SwiftLogistics System Architecture

SwiftLogistics System Architecture: Complete Data Flows & Component Interactions

Version: 1.0
Last Updated: February 4, 2026

This document provides a comprehensive technical reference for understanding how data flows through the SwiftLogistics system, including JWT authentication, RabbitMQ message routing, event triggers, and protocol adapters.

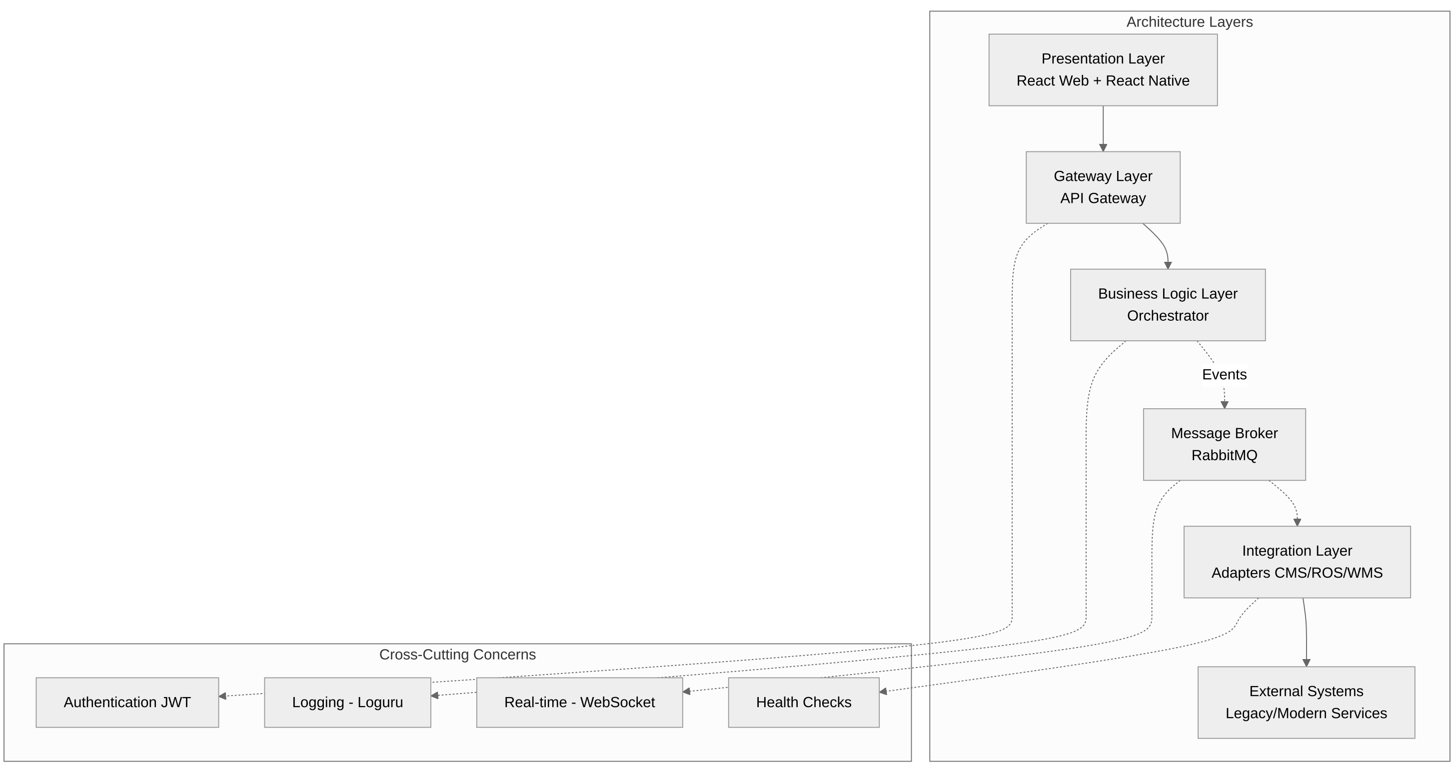
Table of Contents

---

1. Overall Architecture

1.1 Architectural Style

SwiftLogistics employs a **hybrid architectural style** combining multiple patterns:

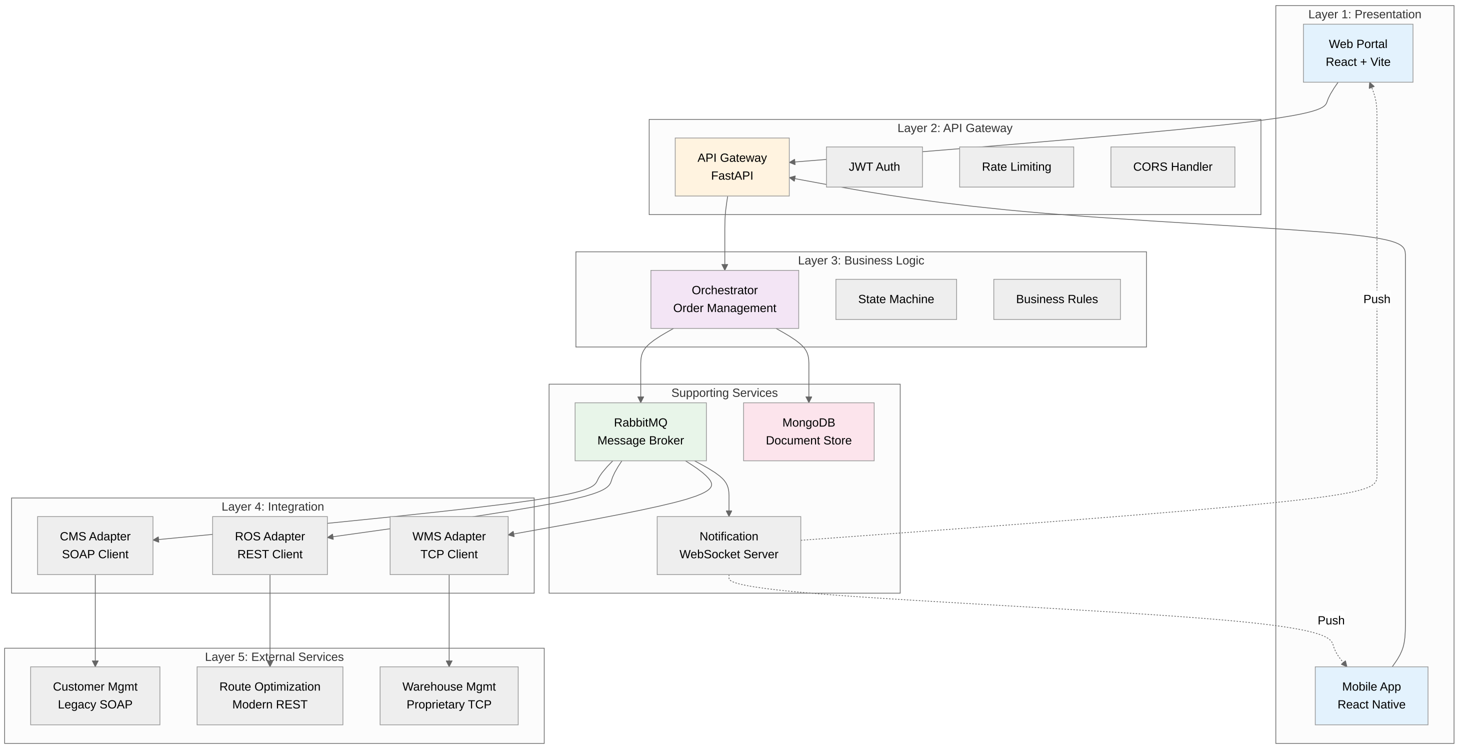


Primary Architectural Patterns:

1. **Microservices Architecture**
   - Independent, loosely-coupled services
   - Single responsibility per service
   - Independent deployment and scaling
   - Polyglot persistence capable
2. **Event-Driven Architecture (EDA)**
   - Asynchronous communication via RabbitMQ
   - Pub/Sub pattern for broadcasting events
   - Event sourcing for audit trail
   - Eventual consistency model

3. **Layered Architecture**
   - Clear separation of concerns
   - Well-defined interfaces between layers
   - Dependency direction: top-down
   - Each layer has specific responsibilities
4. **API Gateway Pattern**
   - Single entry point for clients
   - Cross-cutting concerns (auth, rate limiting)
   - Request routing and composition
   - Protocol translation (if needed)

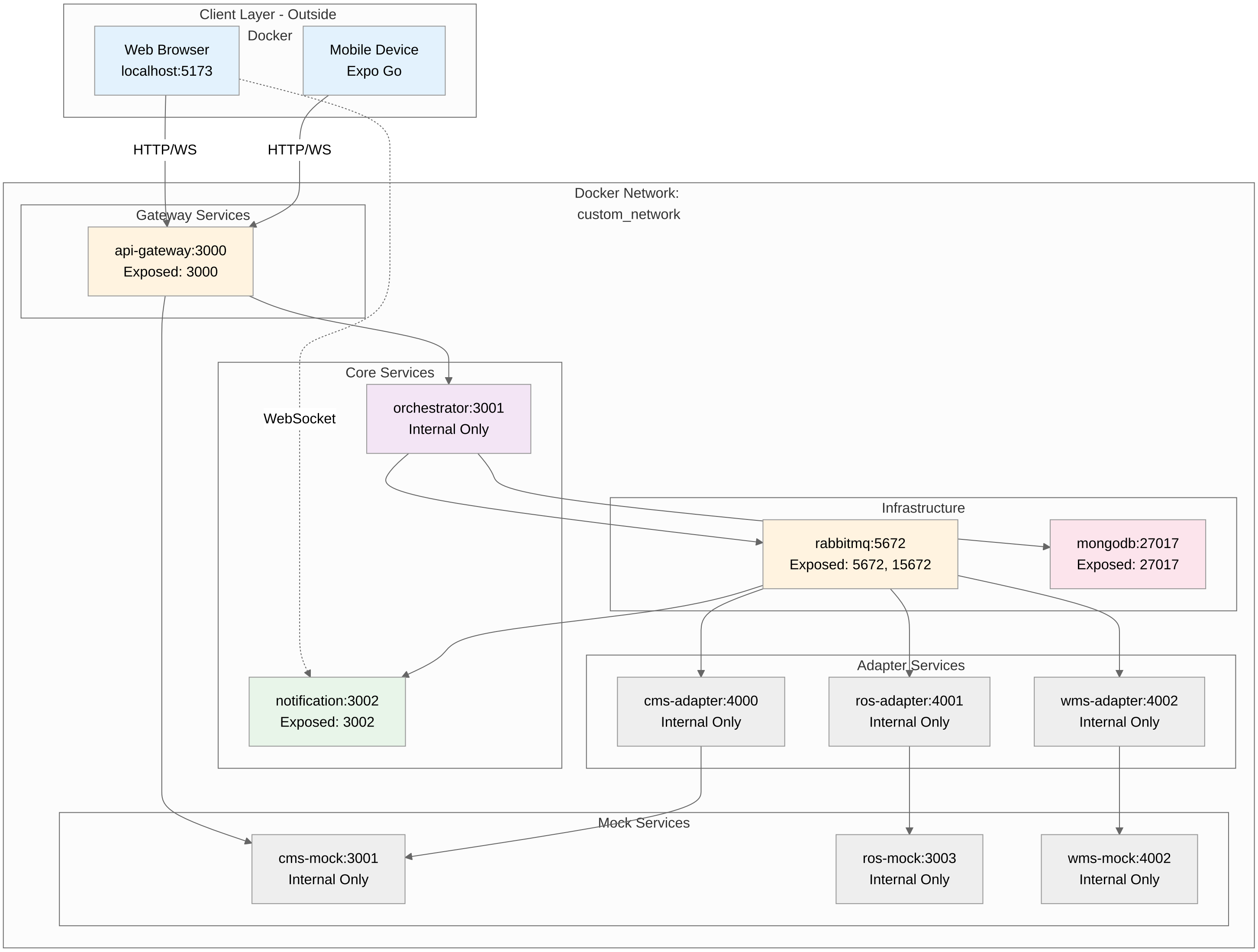
1.2 Architectural Layers



Layer Responsibilities:

Layer	Services	Responsibilities	Technologies
Presentation	Web Portal, Mobile App	User interface, user interactions	React, React Native
Gateway	API Gateway	Auth, routing, rate limiting, CORS	FastAPI, JWT, SlowAPI
Business Logic	Orchestrator	Order processing, state management	FastAPI, Motor, Beanie
Integration	CMS/ROS/WMS Adapters	Protocol translation, external calls	FastAPI, Pika, zeep/httpx/socket
External	CMS/ROS/WMS Services	Domain-specific operations	SOAP/REST/TCP
Supporting	RabbitMQ, MongoDB, Notification	Messaging, persistence, real-time	RabbitMQ, MongoDB, Socket.IO

1.3 Deployment Architecture



Port Mapping:

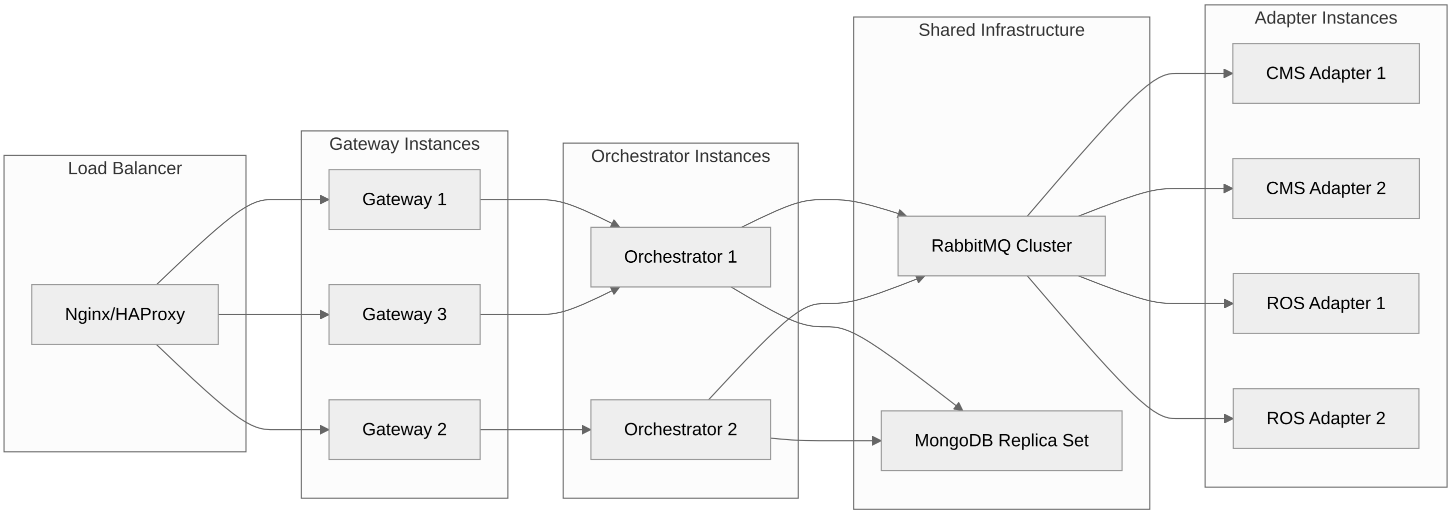
Service	Internal Port	Exposed Port	Purpose
api-gateway	3000	3000	Client HTTP requests
notification-service	3002	3002	WebSocket connections
rabbitmq	5672	5672	AMQP protocol
rabbitmq-admin	15672	15672	Management UI
mongodb	27017	27017	Database access (dev)
orchestrator	3001	-	Internal only
cms-adapter	4000	-	Internal only
ros-adapter	4001	-	Internal only
wms-adapter	4002	-	Internal only

Network Isolation:

- All services run in `custom_network` (bridge driver)
- Services discover each other by container name (DNS)
- Only necessary ports exposed to host
- Internal services not accessible from outside Docker

1.4 Scalability Considerations

Horizontal Scaling Strategy:



Scalability Per Service:

Service	Scaling Type	Strategy	Considerations
API Gateway	Horizontal	Add instances behind load balancer	Stateless, rate limit needs shared cache
Orchestrator	Horizontal	Multiple instances, shared DB	MongoDB handles concurrent writes
Adapters	Horizontal	Multiple consumers per queue	RabbitMQ distributes messages round-robin
Notification	Vertical	Increase WebSocket connections	Socket.IO supports sticky sessions
RabbitMQ	Horizontal	Cluster mode	Mirror queues for HA
MongoDB	Horizontal	Replica set, sharding	Shard by customerId or orderId

Current Bottlenecks:

1. **MongoDB Writes** - Orchestrator inserts every order
   - **Solution:** Add read replicas for query distribution
2. **WebSocket Connections** - Single notification service
   - **Solution:** Use Socket.IO with Redis adapter for multi-instance
3. **External Services** - CMS/ROS/WMS may be slow
   - **Solution:** Timeouts, circuit breakers, caching

1.5 Quality Attributes

Performance:

- **Target:** < 200ms response time for API calls (P95)
- **Achieved via:**
  - AsyncIO for non-blocking I/O
  - Connection pooling (MongoDB: 100 connections)
  - Message queue for async processing
  - In-memory caching where applicable

Reliability:

- **Target:** 99.9% uptime
- **Achieved via:**
  - Durable message queues (survive broker restart)
  - Persistent messages (written to disk)
  - Manual ACK (retry on failure)
  - Health check endpoints on all services
  - Docker restart policies (on-failure)

Scalability:

- **Target:** Handle 1000+ orders/minute
- **Achieved via:**
  - Stateless services (easy horizontal scaling)
  - RabbitMQ queues (multiple consumers)
  - MongoDB connection pool
  - Docker Compose scales (–scale flag)

Maintainability:

- **Target:** New features in < 2 weeks
- **Achieved via:**
  - Modular service design
  - Clear separation of concerns
  - Comprehensive logging (Loguru)
  - Type hints (Python 3.13+)
  - Consistent code structure

Security:

- **Target:** OWASP Top 10 compliance
- **Achieved via:**
  - JWT authentication (stateless)
  - bcrypt password hashing (salted)
  - Rate limiting (prevent abuse)
  - Input validation (Pydantic)
  - CORS restrictions
  - No secrets in code (environment variables)

Observability:

- **Achieved via:**
  - Structured logging (JSON format)
  - Health check endpoints (/health)
  - RabbitMQ management UI (port 15672)
  - MongoDB monitoring (compass/shell)
  - Unique order IDs for tracing

1.6 Architectural Constraints

Technical Constraints:

1. **Python 3.13+** - Required for latest type hints, async features
2. **Docker** - All services containerized, no native deployment
3. **FastAPI** - Chosen for async support, auto-docs
4. **RabbitMQ** - AMQP protocol for messaging
5. **MongoDB** - Document store for flexible schema

Business Constraints:

1. **Legacy Integration** - Must support existing CMS (SOAP)
2. **Real-time Updates** - Clients need instant notification
3. **Protocol Support** - SOAP, REST, TCP all required
4. **Audit Trail** - All orders must be logged

Design Constraints:

1. **Stateless Services** - For horizontal scaling
2. **Event-Driven** - For loose coupling
3. **Async-First** - For high throughput
4. **Container-Native** - For portability

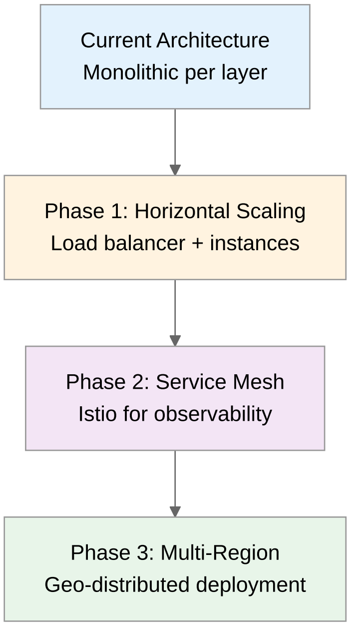
1.7 Architectural Decisions

Key Decisions and Rationale:

Decision	Alternative Considered	Rationale
FastAPI over Flask	Flask with Celery	FastAPI has native async, auto-docs, type validation
RabbitMQ over Kafka	Apache Kafka	Simpler setup, sufficient for current scale, better at-least-once delivery
MongoDB over PostgreSQL	PostgreSQL	Flexible schema for evolving order structure, better for document storage
JWT over Sessions	Server-side sessions	Stateless, scalable, works across services
Docker Compose over K8s	Kubernetes	Simpler for development, sufficient for current deployment
Pika over aio-pika	aio-pika (async)	More mature, simpler threading model for our use case
WebSocket over SSE	Server-Sent Events	Bidirectional, better library support (Socket.IO)
Beanie over PyMongo	Raw PyMongo	Type-safe ODM, better DX, validation built-in

1.8 Future Architecture Evolution

Planned Enhancements:



Phase 1: Horizontal Scaling (Next 3 months)

- Add Nginx load balancer
- Scale API Gateway to 3+ instances
- RabbitMQ cluster (3 nodes)
- MongoDB replica set (3 nodes)
- Redis for shared rate limiting

Phase 2: Observability & Service Mesh (6 months)

- Deploy Istio service mesh
- Distributed tracing (Jaeger/Zipkin)
- Metrics collection (Prometheus)
- Dashboards (Grafana)
- Circuit breaker policies

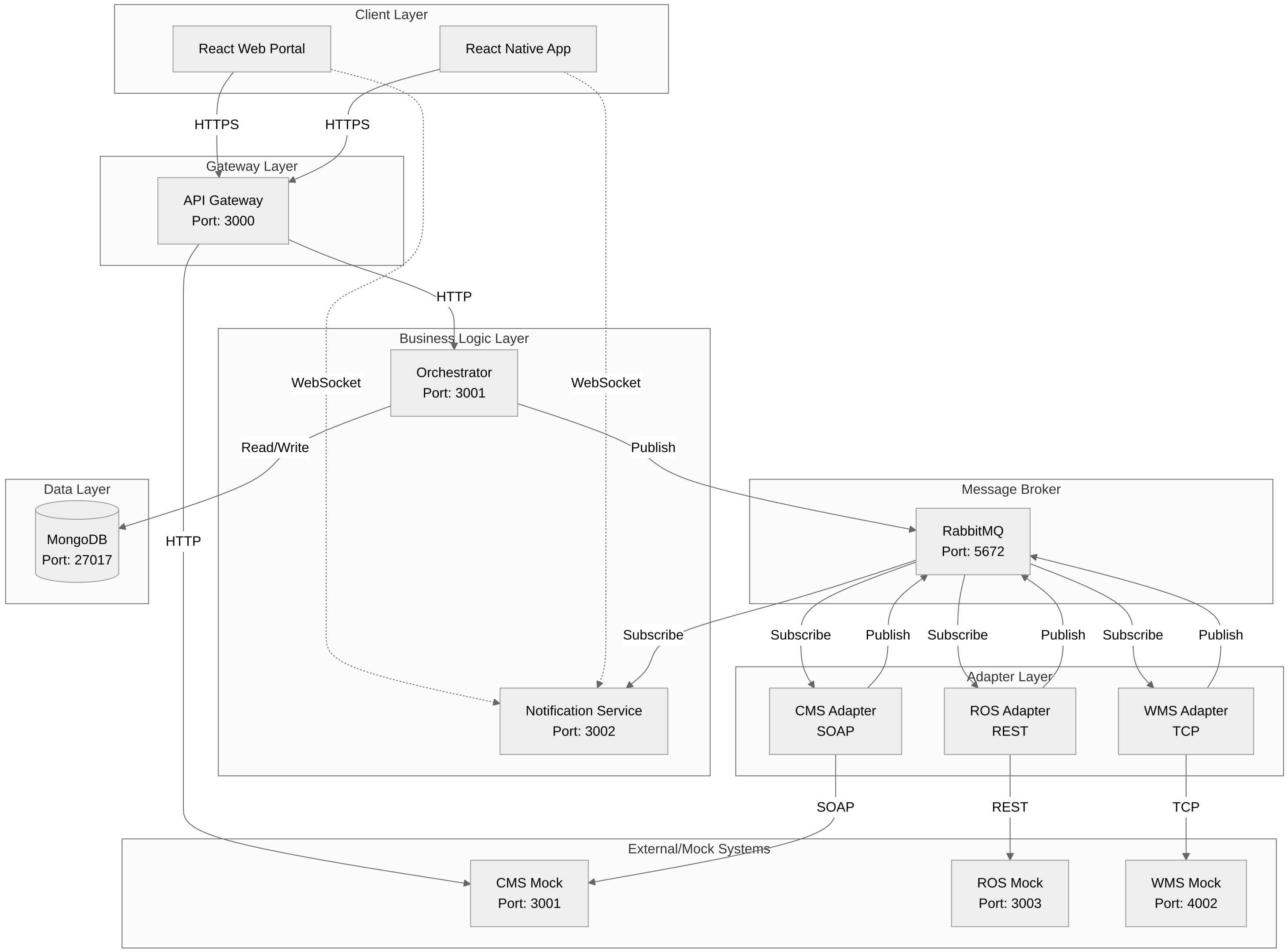
Phase 3: Multi-Region (12 months)

- Deploy to multiple AWS regions
- Global load balancing (Route53)
- Cross-region replication (MongoDB)
- Edge caching (CloudFront)
- DR strategy

---

2. System Overview

1.1 Component Architecture



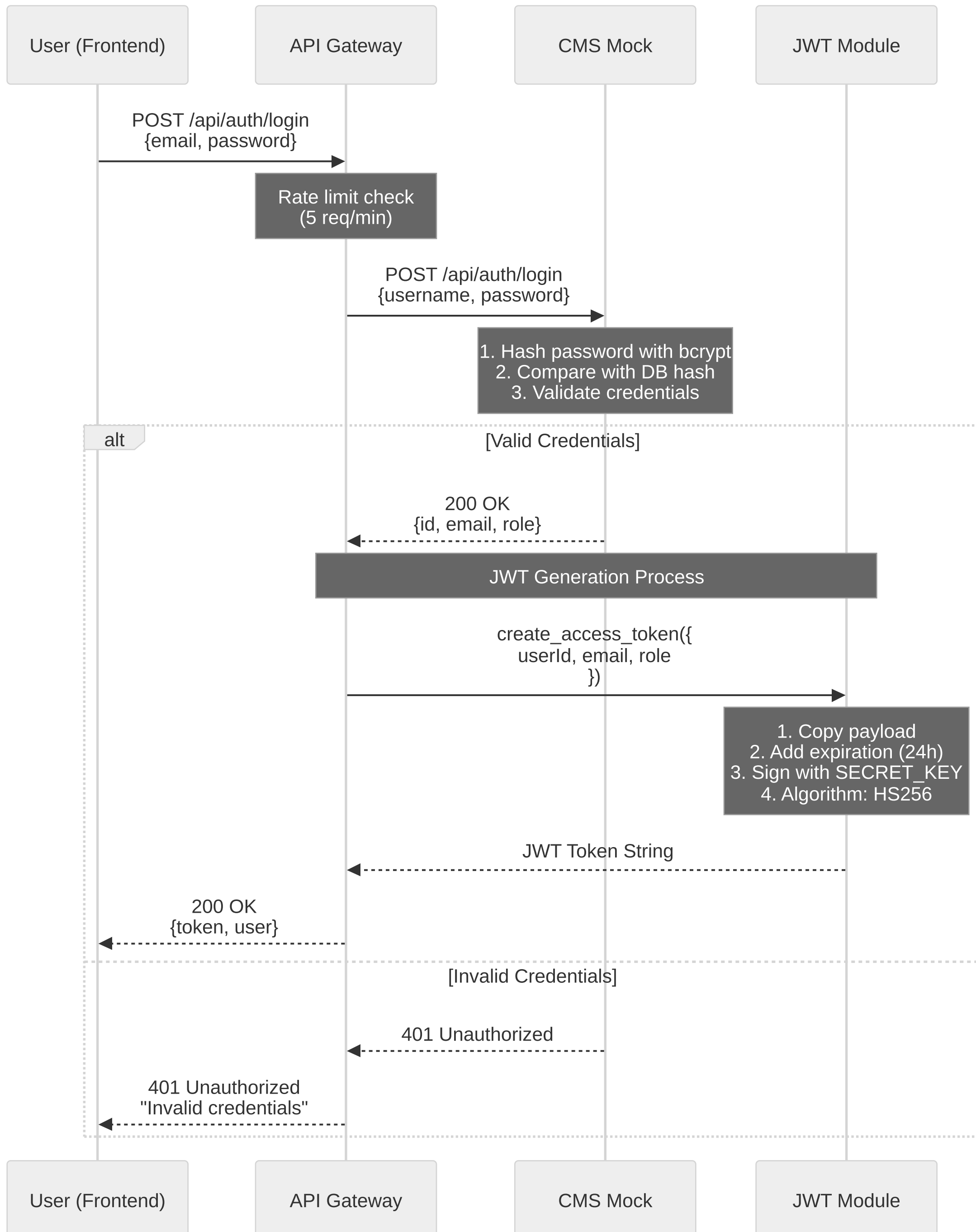
1.2 Technology Stack

Layer	Technology	Purpose
Frontend	React + Vite	Web client portal
Mobile	React Native	Driver mobile app
API Gateway	FastAPI + SlowAPI	Authentication, rate limiting, routing
Orchestrator	FastAPI + Motor	Business logic, state management
Message Broker	RabbitMQ (Pika)	Asynchronous event distribution
Notification	FastAPI + Socket.IO	Real-time WebSocket push
Adapters	FastAPI + Pika	Protocol translation
Database	MongoDB	Order and state persistence
Authentication	JWT (python-jose)	Stateless token-based auth
Password Hashing	bcrypt (passlib)	Secure credential storage

---

2. JWT Authentication Flow

2.1 Authentication Sequence



2.2 JWT Token Structure

Header:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Payload:

```
{
  "userId": "usr_12345",
  "email": "john@example.com",
  "role": "client",
  "exp": 1707091868
}
```

Signature:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  SECRET_KEY
)
```

2.3 Authentication Code Implementation

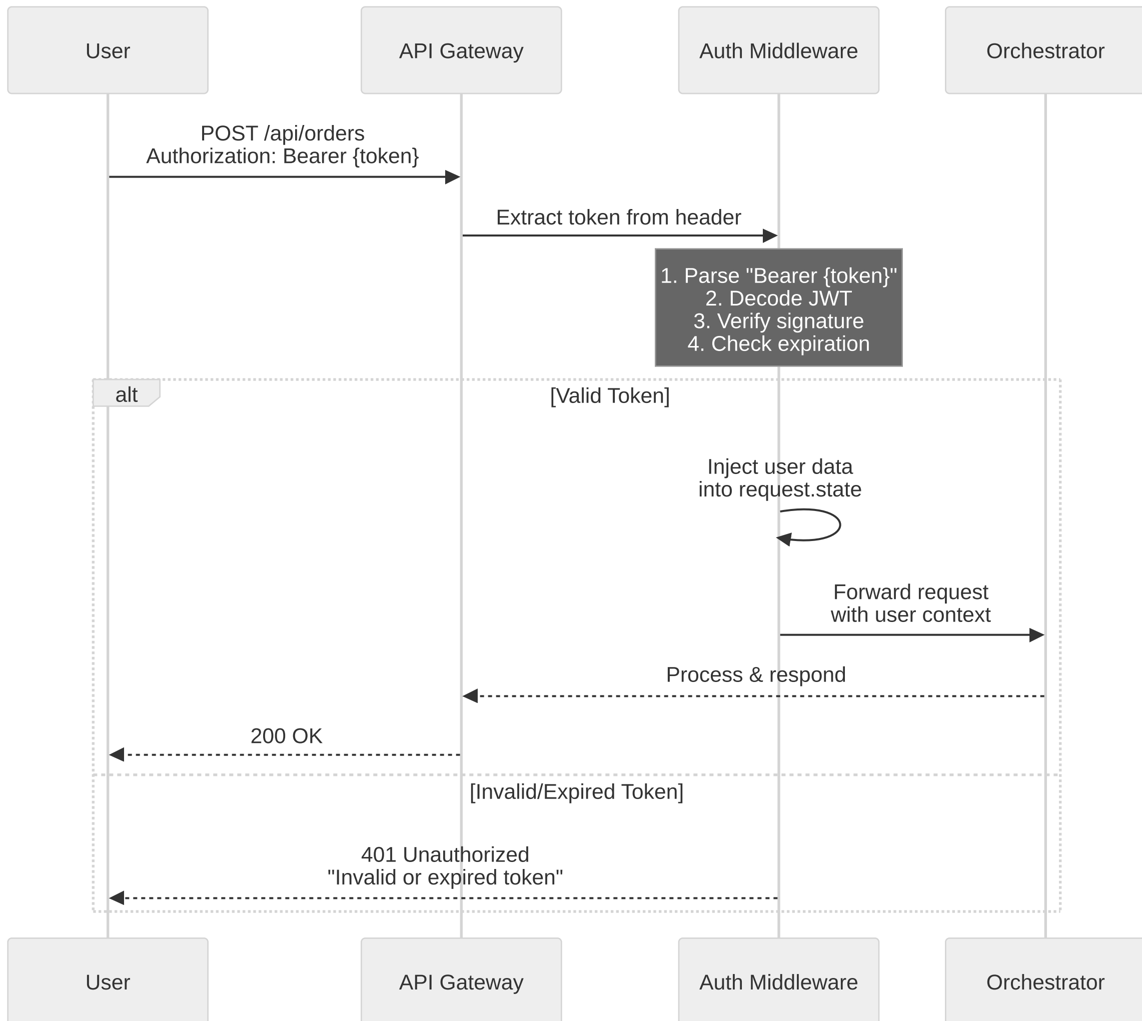
Token Creation (common/auth.py):

```python
def create_access_token(data: dict, expires_delta: Optional[timedelta] = None) -> str:
    """Create JWT access token with HS256 algorithm."""
    to_encode = data.copy()

    if expires_delta:
        expire = datetime.utcnow() + expires_delta
    else:
        expire = datetime.utcnow() + timedelta(hours=ACCESS_TOKEN_EXPIRE_HOURS)

    to_encode.update({"exp": expire})
    encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)

    return encoded_jwt
```

Token Validation (common/auth.py):

```python
def decode_access_token(token: str) -> dict:
    """Decode and validate JWT token."""
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        return payload
    except JWTError as e:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Invalid or expired token",
            headers={"WWW-Authenticate": "Bearer"},
        )
```

2.4 Protected Route Pattern



2.5 Password Security

Hashing Process:

```
# Using bcrypt via passlib
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

# During registration
hashed_password = pwd_context.hash("user_plain_password")
# Stored in database: $2b$12$XYZ...

# During login
is_valid = pwd_context.verify("user_input_password", hashed_password)
```

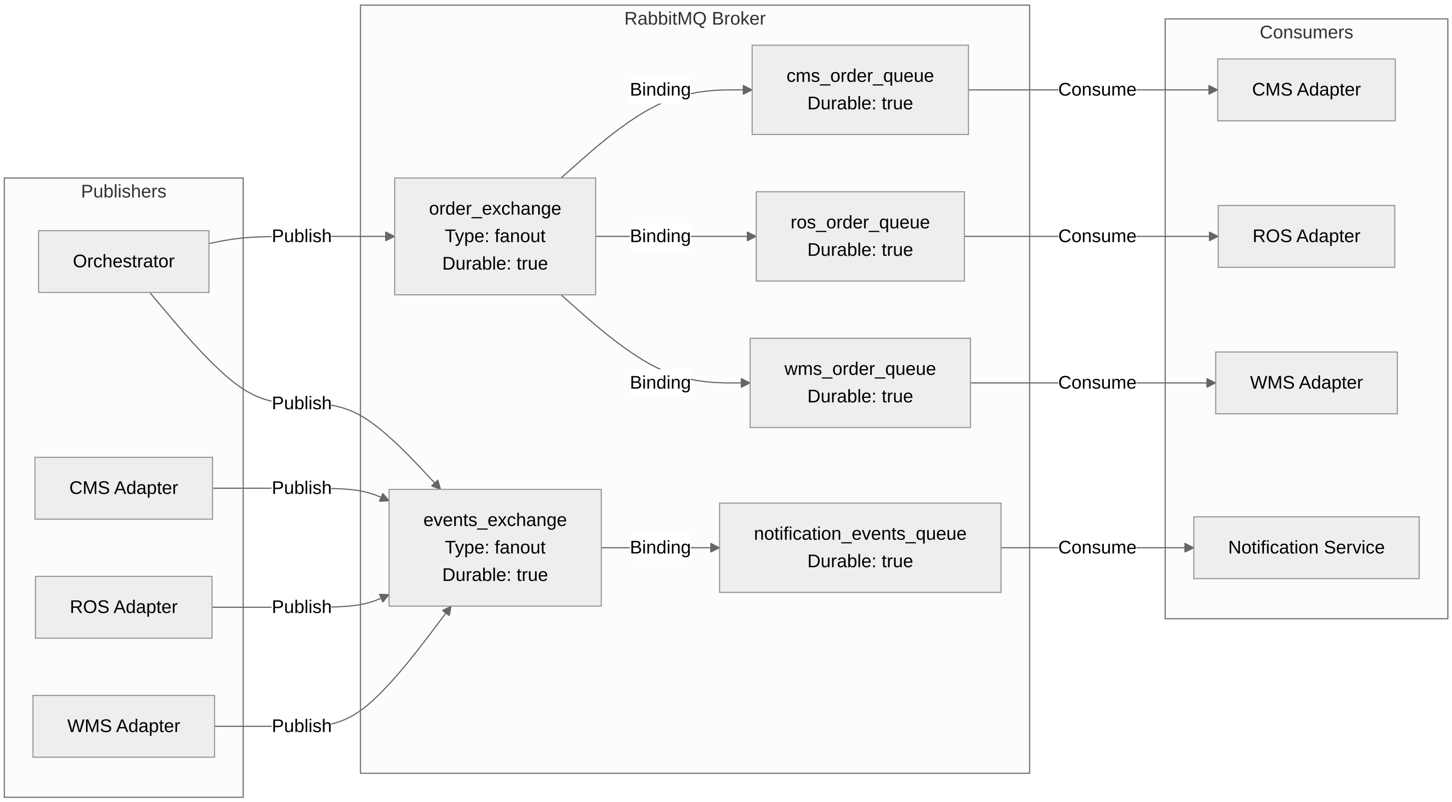
Key Features:

- **Salt:** Automatically generated per password
- **Work Factor:** 12 rounds (default bcrypt)
- **Algorithm:** bcrypt (Blowfish cipher)
- **Storage:** Never store plaintext passwords

---

3. RabbitMQ Message Architecture

3.1 Exchange and Queue Topology



3.2 RabbitMQ Connection Configuration

Connection Establishment (common/messaging.py):

```
def connect(self):
    """Connect to RabbitMQ with blocking connection."""
    logger.info("Connecting to RabbitMQ...")

    # Parse connection URL: amqp://admin:admin123@rabbitmq:5672
    params = pika.URLParameters(self.rabbitmq_url)
    params.heartbeat = 600  # Keep-alive every 10 minutes
    params.blocked_connection_timeout = 300  # 5 minute timeout

    self.connection = pika.BlockingConnection(params)
    self.channel = self.connection.channel()

    logger.info("✓ RabbitMQ connected successfully")
```

3.3 Exchange Declaration

Fanout Exchange (broadcasts to all bound queues):

```
def declare_exchange(self, exchange_name: str, exchange_type: str = 'fanout'):
    """Declare a durable exchange."""
    self.channel.exchange_declare(
        exchange=exchange_name,
        exchange_type=exchange_type,
        durable=True  # Survives broker restart
    )
```

Exchange Types in SwiftLogistics:

Exchange	Type	Purpose
order_exchange	fanout	Broadcast new orders to all adapters
events_exchange	fanout	Broadcast status updates to notification service

3.4 Queue Declaration and Binding

```
# Declare durable queue
queue_name = mq.declare_queue('cms_order_queue')
# Result: Queue persists even if broker restarts

# Bind queue to exchange
mq.bind_queue(
    queue_name='cms_order_queue',
    exchange_name='order_exchange',
    routing_key=''  # Empty for fanout
)
```

3.5 Message Publishing

Publishing with Persistence (common/messaging.py):

```
def publish(self, exchange_name: str, message: dict, routing_key: str = ''):
    """Publish persistent message to exchange."""
    self.channel.basic_publish(
        exchange=exchange_name,
        routing_key=routing_key,
        body=json.dumps(message),
        properties=pika.BasicProperties(
            delivery_mode=2,  # Persistent message
            content_type='application/json'
        )
    )
```

3.6 Message Consumption

Consumer Pattern with Manual ACK (common/messaging.py):

```
def consume(self, queue_name: str, callback: Callable, prefetch_count: int = 1):
    """Start consuming with QoS and manual acknowledgment."""
    self.channel.basic_qos(prefetch_count=prefetch_count)
    self.channel.basic_consume(
        queue=queue_name,
        on_message_callback=callback,
        auto_ack=False  # Manual acknowledgment required
    )

    self.channel.start_consuming()
```

Callback Implementation Example (CMS Adapter):

```
def callback(ch, method, properties, body):
    try:
        order = json.loads(body)
        logger.info(f"Processing order: {order['orderId']}")

        # Process the order (call external service)
        process_order_in_cms(order)

        # Acknowledge successful processing
        ch.basic_ack(delivery_tag=method.delivery_tag)

    except Exception as e:
        logger.error(f"Error processing order: {e}")
        # Requeue message for retry
        ch.basic_nack(delivery_tag=method.delivery_tag, requeue=True)
```

3.7 Reliability Mechanisms

Features ensuring message reliability:

1. **Durable Exchanges:** durable=True - survive broker restart
2. **Durable Queues:** durable=True - persist to disk
3. **Persistent Messages:** delivery_mode=2 - written to disk
4. **Manual ACK:** Messages only removed after explicit acknowledgment
5. **Requeue on Failure:** basic_nack(requeue=True) returns message to queue
6. **Heartbeats:** heartbeat=600 keeps connection alive
7. **Prefetch Count:** prefetch_count=1 prevents worker overload

3.8 Threading Model

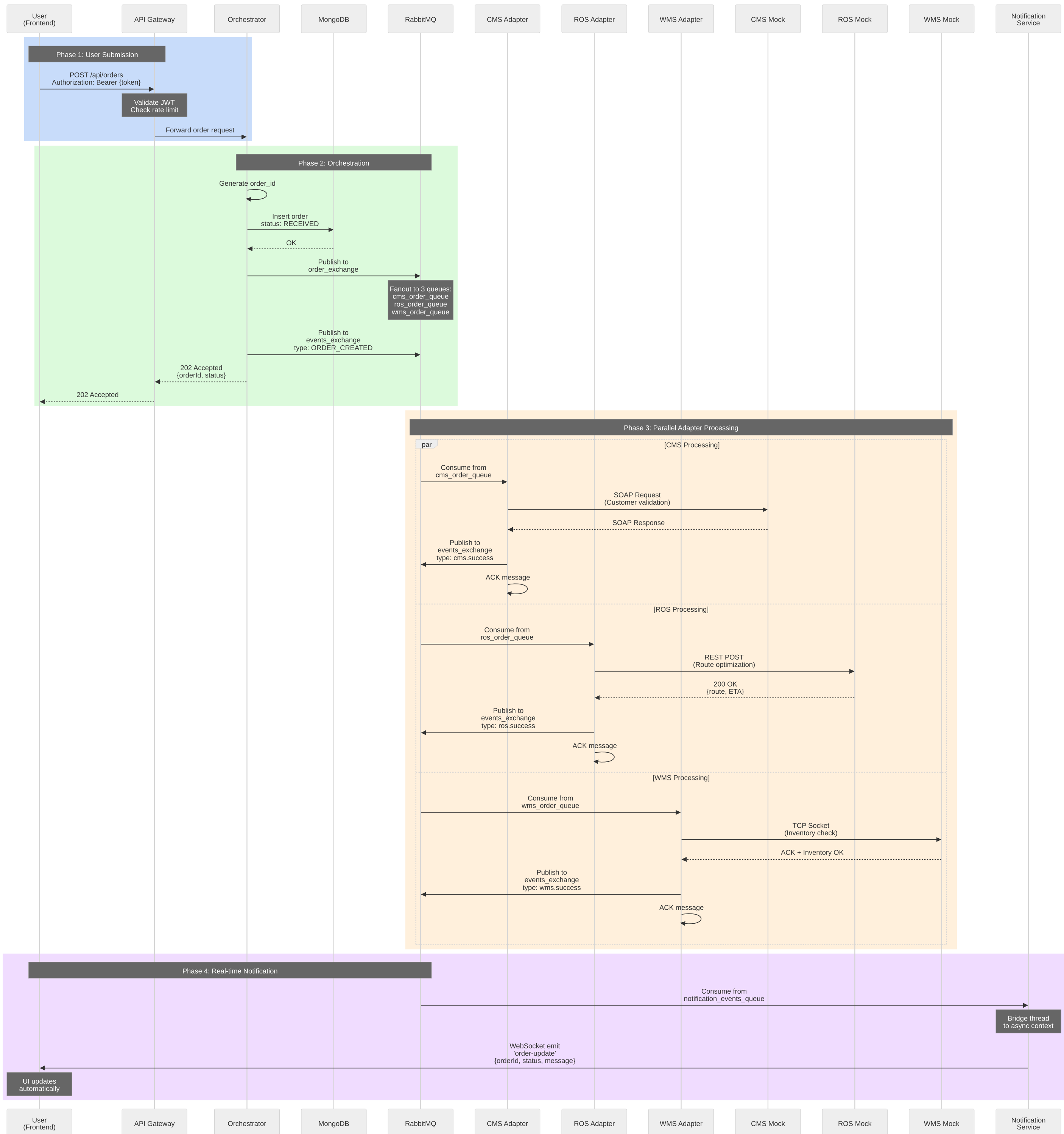
RabbitMQ consumers run in **separate daemon threads** to avoid blocking FastAPI's async event loop:

```python
# Main thread: Runs FastAPI (HTTP endpoints)
# Background thread: Runs Pika consumer (blocking I/O)

def consume_orders():
    mq.connect()
    mq.declare_exchange('order_exchange', 'fanout')
    queue_name = mq.declare_queue('cms_order_queue')
    mq.bind_queue(queue_name, 'order_exchange')
    mq.consume(queue_name, callback)

consumer_thread = threading.Thread(target=consume_orders, daemon=True)
consumer_thread.start()
```

---

4. Complete Order Lifecycle

4.1 End-to-End Order Processing



4.2 Order Creation Code Flow

Step 1: Order Submission (orchestrator/routes/orders.py):

```
@router.post("", response_model=OrderResponse, status_code=status.HTTP_202_ACCEPTED)
async def create_order(order_request: OrderCreateRequest):
    # 1. Generate unique order ID
    order_id = create_order_id()  # e.g., "ORD-20260204-ABC123"

    # 2. Create order document
    order = Order(
        orderId=order_id,
        customerId=order_request.dict().get("customerId", "unknown"),
        pickupLocation=order_request.pickupLocation.dict(),
```

```python
        deliveryAddress=order_request.deliveryAddress.dict(),
        packageDetails=order_request.packageDetails.dict(),
        scheduledPickupTime=order_request.scheduledPickupTime,
        specialInstructions=order_request.specialInstructions,
        status="RECEIVED",
        createdAt=datetime.utcnow()
    )

    # 3. Persist to MongoDB
    await order.insert()

    # 4. Publish to order_exchange (fanout to all adapters)
    message_queue.publish(
        exchange_name='order_exchange',
        message={
            "orderId": order_id,
            "customerId": order.customerId,
            "pickupLocation": order.pickupLocation,
            "deliveryAddress": order.deliveryAddress,
            "packageDetails": order.packageDetails,
            "timestamp": order.createdAt.isoformat()
        }
    )

    # 5. Publish creation event to notification service
    message_queue.publish(
        exchange_name='events_exchange',
        message={
            "type": "ORDER_CREATED",
            "data": {
                "orderId": order_id,
                "status": "RECEIVED",
                "message": "Order received and being processed"
            },
            "timestamp": datetime.utcnow().isoformat()
        }
    )

    # 6. Return immediately (async processing)
    return OrderResponse(
        orderId=order_id,
        status="RECEIVED",
        message="Order received and being processed",
        customerId=order.customerId,
        createdAt=order.createdAt
    )
```

Why 202 Accepted?

The orchestrator returns HTTP 202 (Accepted) instead of 200 (OK) because:

- Processing is **asynchronous** (adapters work in parallel)
- Final order state is not yet determined
- Client receives immediate feedback while processing continues
- Updates delivered via WebSocket

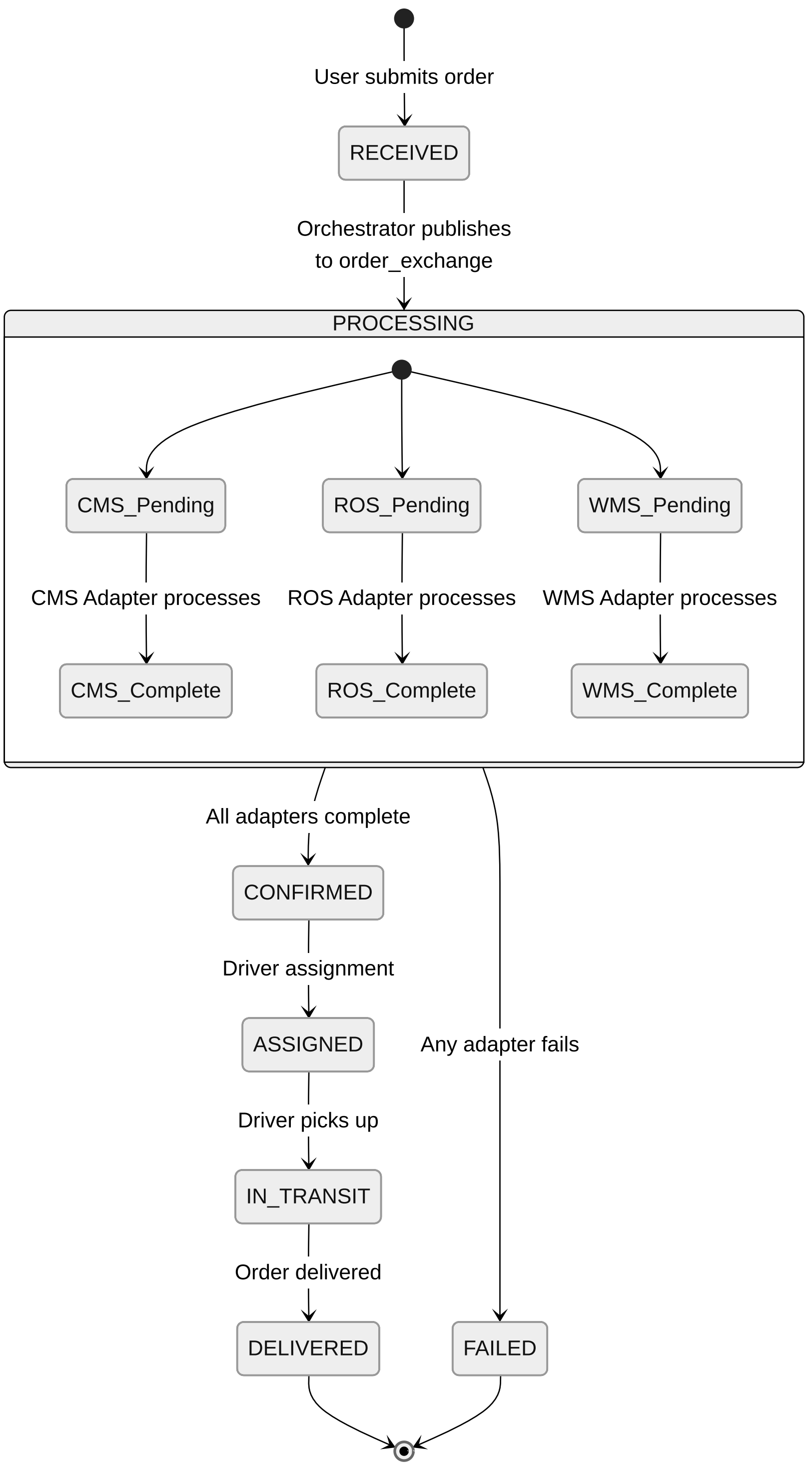
---

5. Event-Driven Architecture

5.1 Event Catalog

Event Type	Publisher	Trigger	Payload	Consumers
ORDER_CREATED	Orchestrator	User creates order	{orderId, status, message}	Notification Service
order.new	Orchestrator	Order published to queue	Full order object	CMS/ROS/WMS Adapters
cms.success	CMS Adapter	CMS SOAP call succeeds	{orderId, status, cmsData}	Notification Service
ros.success	ROS Adapter	ROS REST call succeeds	{orderId, route, eta}	Notification Service
wms.success	WMS Adapter	WMS TCP ACK received	{orderId, inventory}	Notification Service
order.update	Any Adapter	Generic status change	{orderId, status, details}	Notification Service

5.2 Event Trigger Flow



5.3 Event Publishing Pattern

Event Structure:

```json
{
  "type": "ORDER_CREATED",
  "data": {
    "orderId": "ORD-20260204-ABC123",
    "status": "RECEIVED",
    "message": "Order received and being processed",
    "timestamp": "2026-02-04T16:20:30.123Z"
  },
  "timestamp": "2026-02-04T16:20:30.123Z"
}
```

Publishing Code:

```
message_queue.publish(
    exchange_name='events_exchange',
    message={
        "type": "ORDER_CREATED",
        "data": {
            "orderId": order_id,
            "status": "RECEIVED",
            "message": "Order received and being processed"
        },
        "timestamp": datetime.utcnow().isoformat()
    }
)
```

5.4 Event Consumption in Notification Service

Consumer Thread (notification-service/main.py):

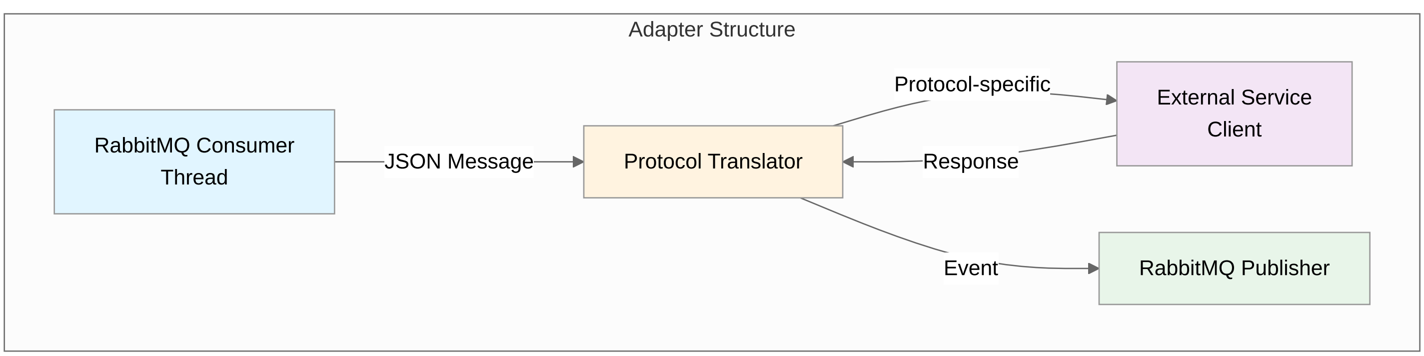
```
def consume_events():
    mq.connect()
    mq.declare_exchange('events_exchange', 'fanout')
    queue_name = mq.declare_queue('notification_events_queue')
    mq.bind_queue(queue_name, 'events_exchange')

    def callback(ch, method, properties, body):
        import json
        import asyncio
        event = json.loads(body)
        logger.info(f"Received event: {event.get('type')}")

        # Bridge from sync Pika thread to async Socket.IO
        asyncio.run(sio.emit('order-update', event.get('data', {})))

        ch.basic_ack(delivery_tag=method.delivery_tag)

    mq.consume(queue_name, callback)
```

---

6. Adapter Communication Patterns

6.1 Adapter Architecture

Each adapter follows the same pattern with protocol-specific implementations:



6.2 CMS Adapter (SOAP Protocol)

Purpose: Validate customer information via legacy SOAP service

Implementation (adapters/cms-adapter/main.py):

```python
def consume_orders():
    mq.connect()
    mq.declare_exchange('order_exchange', 'fanout')
    queue_name = mq.declare_queue('cms_order_queue')
    mq.bind_queue(queue_name, 'order_exchange')

    def callback(ch, method, properties, body):
        try:
            order = json.loads(body)
            logger.info(f"Processing order: {order['orderId']}")

            # Transform JSON to SOAP and call CMS
            # In production: use zeep library for SOAP client
            soap_response = call_cms_soap_service(order)

            # Publish success event
            mq.publish(
                exchange_name='events_exchange',
                message={
                    "type": "cms.success",
                    "data": {
                        "orderId": order['orderId'],
                        "status": "CMS_VALIDATED",
                        "message": "Customer validated successfully"
                    }
                }
            )

            ch.basic_ack(delivery_tag=method.delivery_tag)
        except Exception as e:
            logger.error(f"Error processing order: {e}")
            ch.basic_nack(delivery_tag=method.delivery_tag, requeue=True)

    mq.consume(queue_name, callback)
```

SOAP Message Example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ValidateCustomer xmlns="http://swiftlogistics.com/cms">
      <CustomerId>usr_12345</CustomerId>
      <OrderId>ORD-20260204-ABC123</OrderId>
    </ValidateCustomer>
  </soap:Body>
</soap:Envelope>
```

6.3 ROS Adapter (REST Protocol)

Purpose: Calculate optimal delivery route

Implementation (adapters/ros-adapter/main.py):

```python
def callback(ch, method, properties, body):
    try:
        order = json.loads(body)
        logger.info(f"Processing order: {order['orderId']}")

        # Call ROS REST API
        async with httpx.AsyncClient() as client:
            response = await client.post(
                f"{ROS_API_URL}/api/routes/optimize",
                json={
                    "pickup": order['pickupLocation'],
                    "delivery": order['deliveryAddress']
                }
            )

            route_data = response.json()

            # Publish success event
            mq.publish(
                exchange_name='events_exchange',
                message={
                    "type": "ros.success",
                    "data": {
```

```python
                    "orderId": order['orderId'],
                    "route": route_data['route'],
                    "eta": route_data['eta'],
                    "distance": route_data['distance']
                }
            }
        )

        ch.basic_ack(delivery_tag=method.delivery_tag)
    except Exception as e:
        logger.error(f"Error: {e}")
        ch.basic_nack(delivery_tag=method.delivery_tag, requeue=True)
```

REST Request:

```
POST /api/routes/optimize HTTP/1.1
Host: ros-mock:3003
Content-Type: application/json

{
  "pickup": {
    "address": "123 Main St",
    "coordinates": {"lat": 40.7128, "lng": -74.0060}
  },
  "delivery": {
    "address": "456 Oak Ave",
    "coordinates": {"lat": 40.7589, "lng": -73.9851}
  }
}
```

6.4 WMS Adapter (TCP Socket Protocol)

Purpose: Reserve inventory in warehouse

Implementation (adapters/wms-adapter/main.py):

```python
def callback(ch, method, properties, body):
    try:
        order = json.loads(body)
        logger.info(f"Processing order: {order['orderId']}")

        # Open TCP socket to WMS
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
            sock.connect((WMS_HOST, WMS_PORT))

            # Send TCP message
            message = f"RESERVE|{order['orderId']}|{order['packageDetails']}\n"
            sock.sendall(message.encode())

            # Receive response
            response = sock.recv(1024).decode()

            if response.startswith("ACK"):
                mq.publish(
                    exchange_name='events_exchange',
                    message={
                        "type": "wms.success",
                        "data": {
                            "orderId": order['orderId'],
                            "inventory": "RESERVED",
                            "warehouse": "WH-001"
                        }
                    }
                )

        ch.basic_ack(delivery_tag=method.delivery_tag)
    except Exception as e:
        logger.error(f"Error: {e}")
        ch.basic_nack(delivery_tag=method.delivery_tag, requeue=True)
```

TCP Protocol:

```
Request:  RESERVE|ORD-20260204-ABC123|{"weight":5,"dimensions":"30x20x10"}\n
Response: ACK|WH-001|INVENTORY_RESERVED\n
```

6.5 Adapter Comparison Table

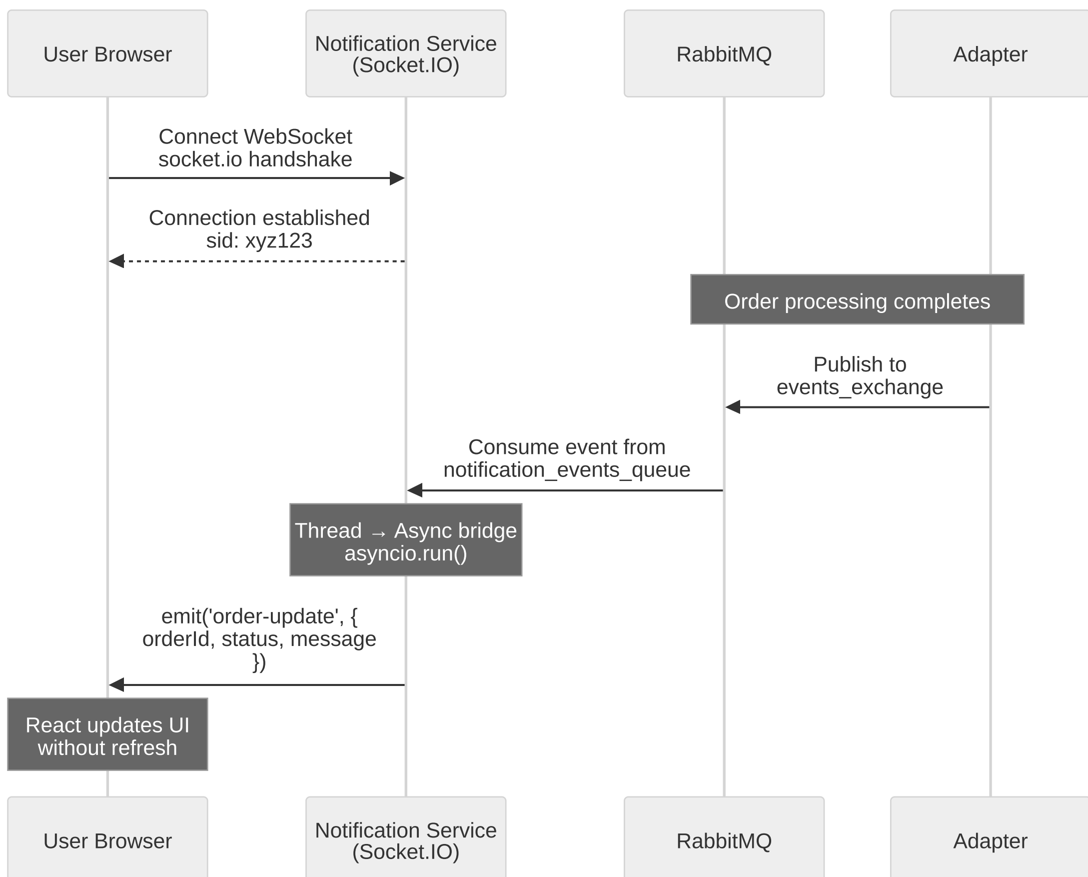
Feature	CMS Adapter	ROS Adapter	WMS Adapter
Protocol	SOAP (XML)	REST (JSON)	TCP (Custom)
Library	zeep	httpx	socket

Feature	CMS Adapter	ROS Adapter	WMS Adapter
Port	3001	3003	4002
Purpose	Customer validation	Route optimization	Inventory management
Request Format	XML Envelope	HTTP POST JSON	TCP text protocol
Sync/Async	Synchronous	Asynchronous	Synchronous
Event Published	cms.success	ros.success	wms.success

---

7. WebSocket Real-Time Notifications

7.1 WebSocket Architecture



7.2 Socket.IO Server Setup

Server Configuration (notification-service/main.py):

```
sio = socketio.AsyncServer(
    async_mode='asgi',
    cors_allowed_origins=[
        "http://localhost:3000",
        "http://localhost:5173",
        os.getenv("WEB_CLIENT_URL", "*")
    ]
)

# Wrap FastAPI with Socket.IO
socket_app = socketio.ASGIApp(sio, app)
```

7.3 Event Handlers

```
@sio.event
async def connect(sid, environ):
```

```python
    """Handle client connection."""
    logger.info(f"Client connected: {sid}")
    # sid is unique socket ID for this client

@sio.event
async def disconnect(sid):
    """Handle client disconnection."""
    logger.info(f"Client disconnected: {sid}")
```

7.4 Broadcasting Events

From RabbitMQ to WebSocket:

```python
def callback(ch, method, properties, body):
    event = json.loads(body)
    logger.info(f"Received event: {event.get('type')}")

    # Bridge blocking Pika thread to async Socket.IO
    asyncio.run(sio.emit('order-update', event.get('data', {})))

    ch.basic_ack(delivery_tag=method.delivery_tag)
```

7.5 Frontend WebSocket Client

React Implementation:

```javascript
import { io } from "socket.io-client";

// Connect to notification service
const socket = io("http://localhost:3002");

// Listen for order updates
socket.on("order-update", (data) => {
  console.log("Order update:", data);
  // Update UI: {orderId, status, message}
  updateOrderUI(data);
});

// Connection events
socket.on("connect", () => {
  console.log("WebSocket connected");
});

socket.on("disconnect", () => {
  console.log("WebSocket disconnected");
});
```

7.6 Thread-to-Async Bridge

Challenge: Pika runs in a blocking thread, Socket.IO is async

Solution:

```python
# Option 1: asyncio.run() creates new event loop
asyncio.run(sio.emit('order-update', data))

# Option 2: run_coroutine_threadsafe() uses existing loop
loop = asyncio.get_event_loop()
asyncio.run_coroutine_threadsafe(
    sio.emit('order-update', data),
    loop
)
```

---

8. Message Payload Specifications

8.1 Order Creation Request

HTTP Request:

```
POST /api/orders HTTP/1.1
Host: api-gateway:3000
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
Content-Type: application/json

{
  "customerId": "usr_12345",
  "pickupLocation": {
    "address": "123 Main Street, New York, NY 10001",
    "coordinates": {
      "lat": 40.7128,
      "lng": -74.0060
    }
  },
  "deliveryAddress": {
    "address": "456 Oak Avenue, Brooklyn, NY 11201",
    "coordinates": {
      "lat": 40.6782,
      "lng": -73.9442
    }
  },
  "packageDetails": {
    "weight": 5.5,
    "dimensions": "30x20x10",
    "description": "Electronics",
    "fragile": true
  },
  "scheduledPickupTime": "2026-02-05T09:00:00Z",
  "specialInstructions": "Handle with care, ring doorbell"
}
```

8.2 RabbitMQ Message: order_exchange

Published by: Orchestrator

Consumed by: CMS Adapter, ROS Adapter, WMS Adapter


```json
{
  "orderId": "ORD-20260204-ABC123",
  "customerId": "usr_12345",
  "pickupLocation": {
    "address": "123 Main Street, New York, NY 10001",
    "coordinates": {
      "lat": 40.7128,
      "lng": -74.006
    }
  },
  "deliveryAddress": {
    "address": "456 Oak Avenue, Brooklyn, NY 11201",
    "coordinates": {
      "lat": 40.6782,
      "lng": -73.9442
    }
  },
  "packageDetails": {
    "weight": 5.5,
    "dimensions": "30x20x10",
    "description": "Electronics",
    "fragile": true
  },
  "timestamp": "2026-02-04T16:20:30.123456Z"
}
```


8.3 RabbitMQ Message: events_exchange

Published by: Orchestrator, Adapters

Consumed by: Notification Service

ORDER_CREATED Event:


```json
{
  "type": "ORDER_CREATED",
  "data": {
    "orderId": "ORD-20260204-ABC123",
    "status": "RECEIVED",
```

```json
    "message": "Order received and being processed"
  },
  "timestamp": "2026-02-04T16:20:30.123456Z"
}
```

cms.success Event:

```json
{
  "type": "cms.success",
  "data": {
    "orderId": "ORD-20260204-ABC123",
    "status": "CMS_VALIDATED",
    "message": "Customer validated successfully",
    "customerData": {
      "id": "usr_12345",
      "name": "John Doe",
      "tier": "premium"
    }
  },
  "timestamp": "2026-02-04T16:20:32.456789Z"
}
```

ros.success Event:

```json
{
  "type": "ros.success",
  "data": {
    "orderId": "ORD-20260204-ABC123",
    "route": {
      "waypoints": [
        { "lat": 40.7128, "lng": -74.006 },
        { "lat": 40.75, "lng": -73.99 },
        { "lat": 40.6782, "lng": -73.9442 }
      ],
      "distance": 12.3,
      "eta": "2026-02-05T10:45:00Z"
    },
    "message": "Route optimized successfully"
  },
  "timestamp": "2026-02-04T16:20:35.789012Z"
}
```

wms.success Event:

```json
{
  "type": "wms.success",
  "data": {
    "orderId": "ORD-20260204-ABC123",
    "inventory": "RESERVED",
    "warehouse": "WH-001",
    "pickingLocation": "A-12-05",
    "message": "Inventory reserved successfully"
  },
  "timestamp": "2026-02-04T16:20:33.234567Z"
}
```

8.4 WebSocket Message: order-update

Emitted by: Notification Service

Received by: Frontend (React/React Native)

```json
{
  "orderId": "ORD-20260204-ABC123",
  "status": "CMS_VALIDATED",
  "message": "Customer validated successfully",
  "customerData": {
    "id": "usr_12345",
    "name": "John Doe",
    "tier": "premium"
  }
}
```

8.5 JWT Token Payload

Encoded Token:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiJ1c3JfMTIzNDUiLCJlbWFpbCI6ImpvaG5AZXhhbXBsZS5jb20iLCJyb2xlIjoiY2xpZW50IiwiZXhwIjoxNzA3MTc4

Decoded Payload:

```
{
  "userId": "usr_12345",
  "email": "john@example.com",
  "role": "client",
  "exp": 1707178868
}
```

8.6 HTTP Response: Order Created

Status: 202 Accepted

```
{
  "orderId": "ORD-20260204-ABC123",
  "status": "RECEIVED",
  "message": "Order received and being processed",
  "customerId": "usr_12345",
  "createdAt": "2026-02-04T16:20:30.123456Z"
}
```

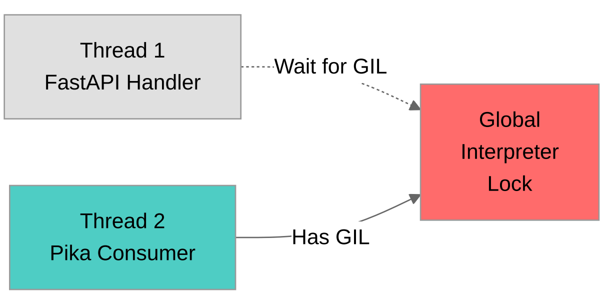
---

9. Thread Safety and Concurrency Deep Dive

9.1 Python Global Interpreter Lock (GIL)

The **GIL** is a mutex that protects access to Python objects, preventing multiple native threads from executing Python bytecodes simultaneously. This has critical implications for the SwiftLogistics system.

What the GIL Does:



Key Characteristics:

1. **Only ONE thread executes Python code at a time**
2. **I/O operations release the GIL** (network calls, file I/O)
3. **CPU-bound tasks don’t benefit from threading** (but we don’t have many CPU-bound tasks)
4. **Simplifies thread safety** for Python object access

Example in SwiftLogistics:

```python
# Thread 1: FastAPI handling HTTP request (async)
async def create_order(order_request):
    # When awaiting I/O, yields control and releases GIL
    await order.insert()  # MongoDB I/O - GIL released
    # Other threads can execute Python code during this wait

# Thread 2: Pika consumer (blocking)
def callback(ch, method, properties, body):
    order = json.loads(body)  # Has GIL
    # When calling external HTTP API, GIL is released
    response = requests.post(CMS_API_URL, data=order)  # I/O - GIL released
```

9.2 Thread Safety Mechanisms

9.2.1 Thread Isolation

Each service uses **complete thread isolation** to avoid race conditions:

```python
# ❌ UNSAFE: Sharing mutable state between threads
shared_orders = {}  # Global dictionary

def thread1_handler():
    shared_orders['ORD-123'] = 'PROCESSING'  # Race condition!

def thread2_handler():
    if 'ORD-123' in shared_orders:  # Race condition!
        process(shared_orders['ORD-123'])

# ✅ SAFE: SwiftLogistics approach - No shared state
def thread1_handler():
    # Each thread has its own connection
    await order.insert()  # Stored in MongoDB

def thread2_handler():
    # Reads from database, not shared memory
    order = await Order.find_one(Order.orderId == 'ORD-123')
```

Thread Isolation Strategies:

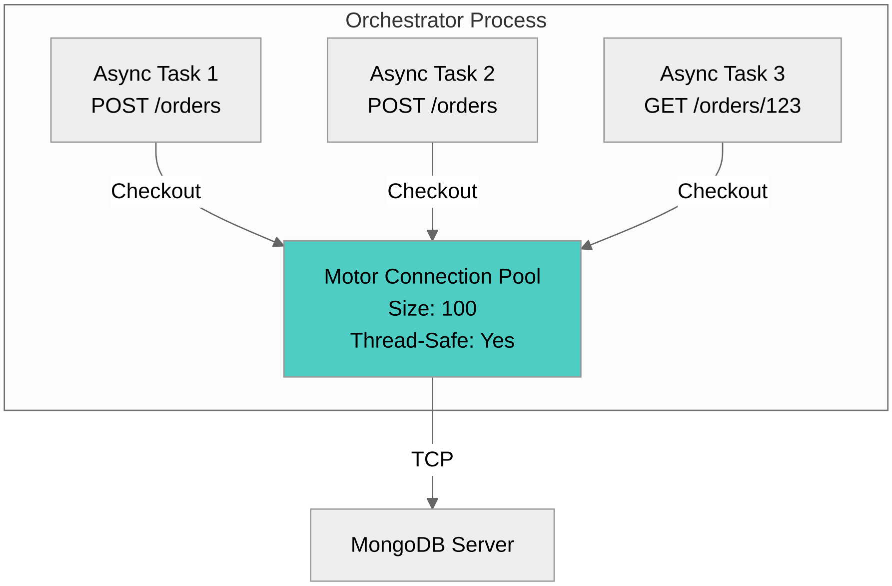
Component	Isolation Method
FastAPI	Each request runs in async task, not thread
Pika Consumer	Dedicated daemon thread, owns its connection
MongoDB	Motor async driver, connection pool per process
RabbitMQ	Separate BlockingConnection per thread

9.2.2 Connection Management

MongoDB Connection Pool (Thread-Safe via Motor):

```python
class Database:
    # Class-level singleton client (shared across async tasks, not threads)
    client: Optional[AsyncIOMotorClient] = None

    @classmethod
    async def connect_db(cls, document_models: list):
        # Create ONE client per process
        cls.client = AsyncIOMotorClient(mongodb_uri)

        # Motor internally manages connection pool
        # - Pool size: default 100
        # - Thread-safe: Yes
        # - Multiple async tasks share pool safely
```

How Motor Connection Pool Works:



Key Features:

- **Automatic checkout/checkin** when async operations complete
- **Thread-safe** via internal locks
- **No manual management** required

RabbitMQ Connection Per Thread (Explicit Isolation):

```
# Each adapter creates its own connection in its thread
def consume_orders():
    # Thread 1: CMS Adapter
    mq = MessageQueue()  # New instance
    mq.connect()  # Dedicated BlockingConnection
    self.connection = pika.BlockingConnection(params)
    self.channel = self.connection.channel()

    # This connection is ONLY used by this thread
    mq.consume(queue_name, callback)
```

Why Not Share Pika Connections?

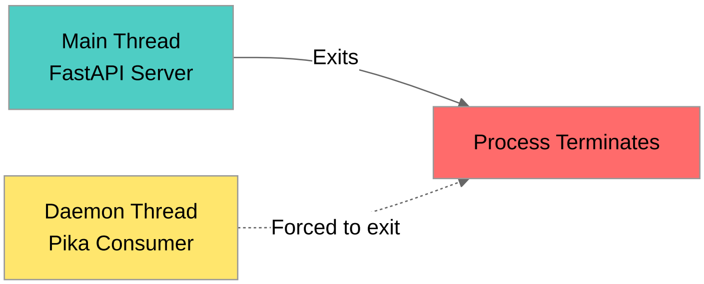
```
# ❌ UNSAFE: Sharing Pika channel between threads
channel = connection.channel()  # Single channel

def thread1():
    channel.basic_publish(...)  # ⚠️ Race condition!

def thread2():
    channel.basic_publish(...)  # ⚠️ Race condition!

# ✅ SAFE: Each thread owns its connection
def thread1():
    mq1 = MessageQueue()
    mq1.connect()  # Own connection
    mq1.channel.basic_publish(...)

def thread2():
    mq2 = MessageQueue()
    mq2.connect()  # Own connection
    mq2.channel.basic_publish(...)
```

9.2.3 Daemon Threads

All background consumer threads are **daemon threads**:

```
consumer_thread = threading.Thread(target=consume_orders, daemon=True)
consumer_thread.start()
```

What daemon=True Means:



Daemon Thread Characteristics:

1. **Non-blocking termination:** Process can exit even if daemon threads are running
2. **No graceful shutdown:** Daemon thread is killed when main thread exits
3. **Use case:** Background tasks that don’t need cleanup (our consumers ACK messages, so no data loss)

Regular Thread vs Daemon Thread:

```
# Regular thread (daemon=False, default)
regular_thread = threading.Thread(target=important_task)
regular_thread.start()
# Process waits for this thread to finish before exiting

# Daemon thread (daemon=True)
daemon_thread = threading.Thread(target=background_task, daemon=True)
daemon_thread.start()
# Process exits immediately, killing this thread
```

9.2.4 Thread-to-Async Bridge

The Notification Service bridges **synchronous Pika callbacks** to **async Socket.IO**:

```
def callback(ch, method, properties, body):
    # Running in: Pika consumer thread (synchronous)
    event = json.loads(body)

    # Problem: sio.emit() is async, but we're in sync context
    # Solution: Create new event loop
    asyncio.run(sio.emit('order-update', event.get('data', {})))

    ch.basic_ack(delivery_tag=method.delivery_tag)
```

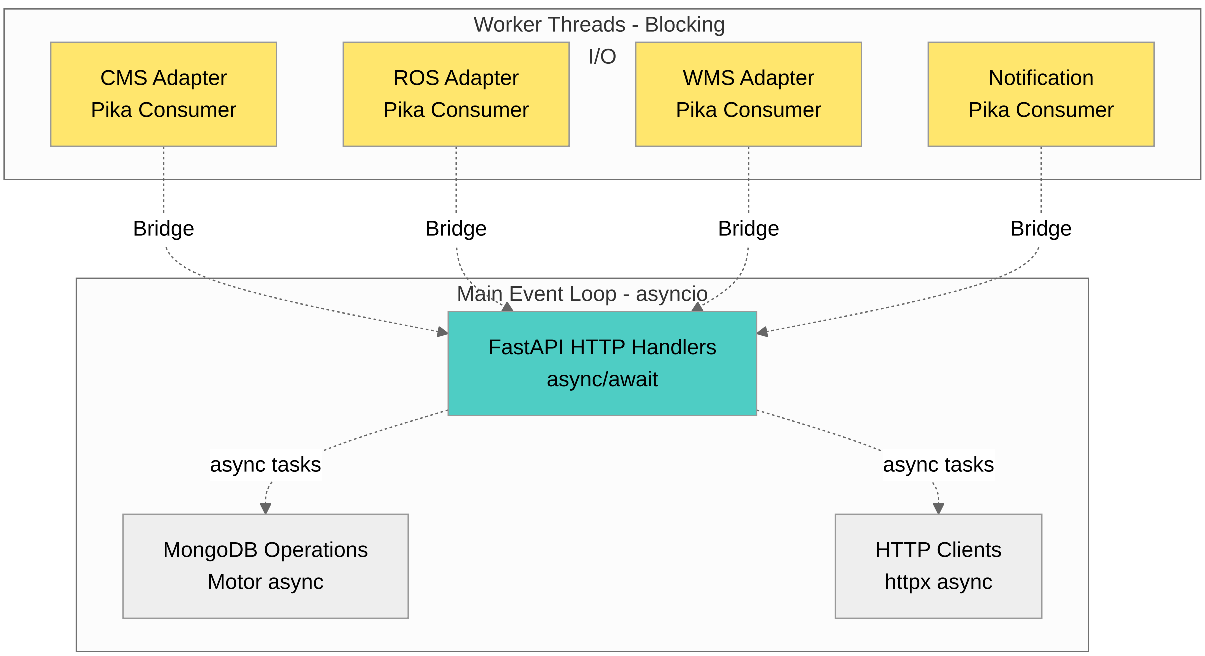
How asyncio.run() Works:

Alternative:
run_coroutine_threadsafe():

```python
# If FastAPI event loop is running
loop = asyncio.get_event_loop()

def callback(ch, method, properties, body):
    event = json.loads(body)

    # Schedule coroutine in existing loop
    future = asyncio.run_coroutine_threadsafe(
        sio.emit('order-update', event.get('data', {})),
        loop
    )

    # Optionally wait for result
    # future.result(timeout=5)

    ch.basic_ack(delivery_tag=method.delivery_tag)
```

9.3 Concurrency Model Summary

SwiftLogistics uses a hybrid concurrency model:



Benefits of This Model:

1. **AsyncIO for HTTP:** High throughput for API requests
2. **Threads for Pika:** Compatible with blocking Pika library
3. **GIL doesn't hurt:** Most time spent in I/O (GIL released)
4. **No shared state:** Each layer isolated via database/message queue

---

10. Complete Design Patterns Catalog

10.1 Architectural Patterns

10.1.1 Microservices Architecture

The system is decomposed into independent services:

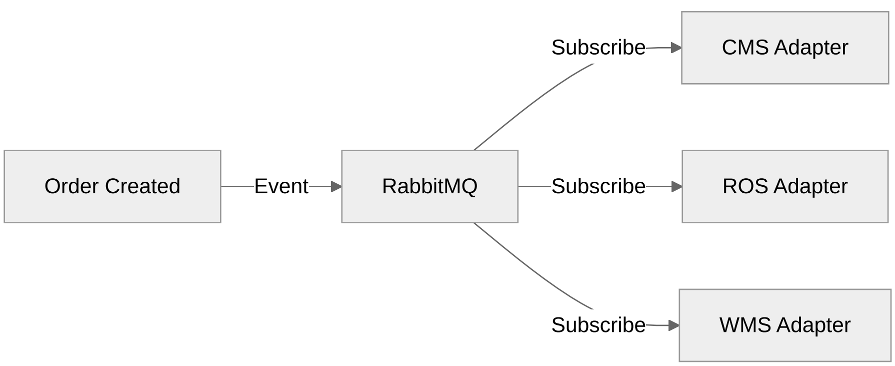
```
api-gateway → Authentication, routing
orchestrator → Business logic, state machine
notification-service → Real-time push
cms-adapter → Protocol translation
ros-adapter → Protocol translation
wms-adapter → Protocol translation
```

Characteristics:

- ✅ Independent deployment
- ✅ Single responsibility
- ✅ Communication via message queue
- ✅ Polyglot persistence (if needed)

10.1.2 Event-Driven Architecture (EDA)

Events trigger actions asynchronously:



Benefits:

- Loose coupling
- Scalability (add more consumers)
- Resilience (failures isolated)

10.1.3 API Gateway Pattern

Single entry point for all client requests:

```
Frontend → API Gateway → [Orchestrator, CMS Mock, ...]
```

Responsibilities:

- Authentication (JWT)
- Rate limiting
- Request routing
- CORS handling

10.1.4 Orchestration Pattern (Saga-like)

Orchestrator manages distributed transaction flow:

```
# Orchestrator coordinates but doesn't wait
1. Save order to DB
2. Publish to RabbitMQ
3. Return 202 Accepted

# Adapters work independently
4. CMS processes → publishes event
5. ROS processes → publishes event
6. WMS processes → publishes event

# Eventual consistency achieved via events
```

10.2 Structural Design Patterns

10.2.1 Adapter Pattern

Purpose: Convert one interface to another

Implementation: Protocol adapters translate RabbitMQ JSON to SOAP/REST/TCP

```python
class CMSAdapter:
    def process_order(self, json_order):
        # Adapt JSON to SOAP
        soap_request = self.json_to_soap(json_order)
        response = self.call_cms(soap_request)
        # Adapt SOAP response back to JSON event
        return self.soap_to_json(response)
```

Before Adapter:

```
Orchestrator (JSON) → ❌ CMS (SOAP) - Incompatible!
```

After Adapter:

```
Orchestrator (JSON) → Adapter → SOAP → CMS
CMS → SOAP → Adapter → JSON → Events
```

10.2.2 Façade Pattern

Purpose: Simplified interface to complex subsystem

Implementation: MessageQueue class wraps Pika complexity

```python
# Complex Pika API
params = pika.URLParameters(url)
params.heartbeat = 600
connection = pika.BlockingConnection(params)
channel = connection.channel()
channel.exchange_declare(exchange='x', exchange_type='fanout', durable=True)
channel.queue_declare(queue='q', durable=True)
channel.queue_bind(queue='q', exchange='x')
channel.basic_publish(exchange='x', routing_key='', body=json.dumps(msg))

# Simple Façade
mq = MessageQueue()
mq.connect()
mq.declare_exchange('x', 'fanout')
mq.declare_queue('q')
mq.bind_queue('q', 'x')
mq.publish('x', msg)
```

10.2.3 Repository Pattern

Purpose: Abstraction over data access

Implementation: Beanie ODM provides repository-like interface

```python
# Instead of raw MongoDB queries
db.orders.find_one({'orderId': 'ORD-123'})

# Use repository pattern via Beanie
order = await Order.find_one(Order.orderId == 'ORD-123')
await order.save()
await order.delete()
```

Benefits:

- Type safety
- Query builder
- No SQL injection
- Easy to mock for testing

10.3 Behavioral Design Patterns

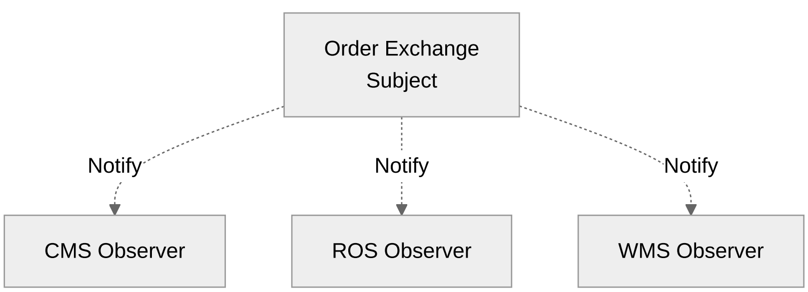
10.3.1 Observer Pattern (Pub/Sub)

Purpose: One-to-many dependency where state change notifies observers

Implementation: RabbitMQ fanout exchange

```python
# Subject (Publisher)
orchestrator.publish(exchange='order_exchange', message=order)

# Observers (Subscribers)
cms_adapter.subscribe(queue='cms_order_queue')
ros_adapter.subscribe(queue='ros_order_queue')
wms_adapter.subscribe(queue='wms_order_queue')
```

Diagram:



10.3.2 Strategy Pattern

Purpose: Define family of algorithms, make them interchangeable

Implementation: Protocol adapters implement same interface with different strategies

```python
class ProtocolAdapter(ABC):
    @abstractmethod
    def process_order(self, order):
        pass

class SOAPAdapter(ProtocolAdapter):
    def process_order(self, order):
        # SOAP strategy
        pass

class RESTAdapter(ProtocolAdapter):
    def process_order(self, order):
        # REST strategy
        pass

class TCPAdapter(ProtocolAdapter):
    def process_order(self, order):
        # TCP strategy
        pass
```

10.3.3 Middleware/Chain of Responsibility Pattern

Purpose: Pass request through chain of handlers

Implementation: FastAPI middleware stack

```
Request
  ↓
[CORS Middleware]
  ↓
[Rate Limiter]
  ↓
[JWT Authentication]
  ↓
[Request Handler]
  ↓
Response
```

Code:

```python
# Each middleware can process or pass to next
@app.middleware("http")
async def cors_middleware(request, call_next):
    response = await call_next(request)
    response.headers["Access-Control-Allow-Origin"] = "*"
    return response

@app.middleware("http")
async def auth_middleware(request, call_next):
    if request.url.path.startswith("/api/"):
        # Verify JWT
        verify_token(request.headers.get("Authorization"))
    response = await call_next(request)
    return response
```

10.3.4 Template Method Pattern

Purpose: Define skeleton of algorithm, subclasses fill in steps

Implementation: Adapter base structure

```python
# Template method in base class
class BaseAdapter:
    def process_message(self, message):
        # Step 1: Always parse JSON
        data = self.parse_message(message)
```

```python
        # Step 2: Transform (subclass implements)
        transformed = self.transform(data)

        # Step 3: Send (subclass implements)
        response = self.send_to_external(transformed)

        # Step 4: Always publish result
        self.publish_result(response)

    @abstractmethod
    def transform(self, data):
        pass

    @abstractmethod
    def send_to_external(self, data):
        pass

# Concrete implementation
class CMSAdapter(BaseAdapter):
    def transform(self, data):
        return json_to_soap(data)

    def send_to_external(self, soap_data):
        return call_cms_soap(soap_data)
```

10.4 Creational Design Patterns

10.4.1 Singleton Pattern

Purpose: Ensure only one instance exists

Implementation: Database connection, RabbitMQ connection

```python
class Database:
    client: Optional[AsyncIOMotorClient] = None  # Class variable (singleton)

    @classmethod
    async def connect_db(cls, document_models):
        if cls.client is None:  # Only create once
            cls.client = AsyncIOMotorClient(mongodb_uri)
```

Why Singleton?

- **Database:** Share connection pool across all requests
- **RabbitMQ:** Expensive to create connections, reuse per thread

10.4.2 Factory Pattern

Purpose: Create objects without specifying exact class

Implementation: Order ID generation

```python
def create_order_id() -> str:
    """Factory method for creating unique order IDs."""
    timestamp = int(datetime.utcnow().timestamp())
    random_suffix = ''.join(random.choices(string.ascii_lowercase + string.digits, k=6))
    return f"ORD-{timestamp}-{random_suffix}"

# Usage - don't care about implementation details
order_id = create_order_id()  # Factory creates it
```

10.5 Concurrency Patterns

10.5.1 Active Object Pattern

Purpose: Decouple method execution from invocation

Implementation: RabbitMQ message queue

```
Client calls → Returns immediately (202 Accepted)
Background → Processes asynchronously
Client notified → Via WebSocket when done
```

10.5.2 Half-Sync/Half-Async Pattern

Purpose: Separate sync and async processing layers

Implementation: FastAPI (async) + Pika (sync threads)

```
Async Layer: FastAPI HTTP handlers
Queue: RabbitMQ
Sync Layer: Pika consumer threads
```

10.5.3 Producer-Consumer Pattern

Purpose: Producers create work, consumers process it

Implementation: Order processing pipeline

```
Producer: Orchestrator creates orders
Queue: RabbitMQ
Consumers: CMS/ROS/WMS adapters process orders
```

10.6 Reliability Patterns

10.6.1 Retry Pattern (via RabbitMQ Requeue)

Implementation:

```python
def callback(ch, method, properties, body):
    try:
        process_order(body)
        ch.basic_ack(delivery_tag=method.delivery_tag)  # Success
    except Exception as e:
        logger.error(f"Failed: {e}")
        ch.basic_nack(delivery_tag=method.delivery_tag, requeue=True)  # Retry
```

Requeue Flow:

```
1. Message delivered to consumer
2. Processing fails
3. NACK with requeue=True
4. Message returned to queue
5. Delivered to consumer again (retry)
```

10.6.2 Idempotency Pattern

Purpose: Same operation can be applied multiple times safely

Implementation: Order ID uniqueness

```python
# Even if message redelivered, won't create duplicate
order_id = create_order_id()  # Unique timestamp-based ID
```

```python
try:
    await order.insert()  # MongoDB unique index on orderId
except DuplicateKeyError:
    logger.info("Order already exists, skipping")
```

10.6.3 Circuit Breaker Pattern (Implicit via Timeouts)

Implementation: HTTP timeouts prevent cascading failures

```python
async with httpx.AsyncClient() as client:
    response = await client.post(
        CMS_MOCK_URL,
        data=payload,
        timeout=10.0  # Circuit opens if service slow
    )
```

Circuit States:

```
Closed → Normal operation
Open → Too many failures, reject requests
Half-Open → Test if service recovered
```

10.7 Security Patterns

10.7.1 Token-Based Authentication

Implementation: JWT

```
1. User logs in → Server issues JWT
2. Client stores token
3. Client includes token in requests
4. Server validates token (stateless)
```

10.7.2 Password Hashing

Implementation: bcrypt with salts

```python
# Salt automatically generated per password
hashed = bcrypt.hash("password123")
# Result: $2b$12$randomsalt$hashvalue
```

10.7.3 Rate Limiting

Implementation: SlowAPI

```python
@router.post("/login")
@limiter.limit("5/minute")  # Max 5 requests per minute per IP
async def login(request: Request, ...):
    pass
```

10.8 Integration Patterns

10.8.1 Protocol Translation

Implementation: Adapters

```
JSON → SOAP (CMS Adapter)
JSON → REST (ROS Adapter)
JSON → TCP (WMS Adapter)
```

10.8.2 Message Routing

Implementation: RabbitMQ fanout exchange

```
Single message → Multiple queues
Each adapter gets a copy
```

10.8.3 Message Transformation

Implementation: Event enrichment

```python
# Adapter receives minimal order data
order = {"orderId": "ORD-123", "customerId": "usr_456"}

# Adapter enriches with external data
customer_data = call_cms(order['customerId'])

# Publishes enriched event
publish_event({
    "orderId": order['orderId'],
    "customerName": customer_data['name'],
    "customerTier": customer_data['tier']
})
```

10.9 Pattern Summary Table

Pattern	Category	Implementation	Purpose
Microservices	Architectural	Service decomposition	Independent services
Event-Driven	Architectural	RabbitMQ	Async communication
API Gateway	Architectural	API Gateway service	Single entry point
Orchestration	Architectural	Orchestrator service	Coordinate workflow
Adapter	Structural	Protocol adapters	Interface conversion
Façade	Structural	MessageQueue class	Simplify Pika API
Repository	Structural	Beanie ODM	Data access abstraction
Observer	Behavioral	Pub/Sub via RabbitMQ	Event notification
Strategy	Behavioral	Protocol strategies	Interchangeable algorithms
Middleware	Behavioral	FastAPI middleware	Request pipeline
Template Method	Behavioral	Adapter base class	Algorithm skeleton
Singleton	Creational	Database connection	Single instance
Factory	Creational	Order ID generation	Object creation
Active Object	Concurrency	Message queue	Async execution
Producer-Consumer	Concurrency	Order pipeline	Work distribution
Retry	Reliability	RabbitMQ requeue	Fault tolerance
Idempotency	Reliability	Unique order IDs	Safe retries
Circuit Breaker	Reliability	HTTP timeouts	Prevent cascading failures
Token Auth	Security	JWT	Stateless authentication
Password Hashing	Security	bcrypt	Credential security
Rate Limiting	Security	SlowAPI	Abuse prevention
Protocol Translation	Integration	Adapters	Convert protocols
Message Routing	Integration	Fanout exchange	Broadcast messages

---

11. Environment Configuration

11.1 Environment Variables Reference

Complete configuration from .env.example:

MongoDB Configuration

```
MONGODB_URI=mongodb://admin:admin123@mongodb:27017/swiftlogistics?authSource=admin
MONGO_INITDB_ROOT_USERNAME=admin
MONGO_INITDB_ROOT_PASSWORD=admin123
MONGO_INITDB_DATABASE=swiftlogistics
```

Purpose:

- Connection string for MongoDB Atlas or local instance
- Includes authentication credentials
- Database name: swiftlogistics
- Auth source: admin (admin database)

RabbitMQ Configuration

```
RABBITMQ_URL=amqp://admin:admin123@rabbitmq:5672
RABBITMQ_DEFAULT_USER=admin
RABBITMQ_DEFAULT_PASS=admin123
```

Purpose:

- AMQP protocol connection string
- Default credentials for RabbitMQ management
- Port 5672: AMQP protocol
- Port 15672: Management UI (not in connection string)

Queue Names

```
ORDER_QUEUE=new_order_queue
ORDER_EXCHANGE=order_exchange
EVENTS_EXCHANGE=events_exchange
EVENTS_QUEUE=notification_events_queue
```

Purpose:

- Predefined queue and exchange names
- Ensures consistency across all services
- Can be customized per environment

Service Ports

```
API_GATEWAY_PORT=3000
ORCHESTRATOR_PORT=3001
NOTIFICATION_SERVICE_PORT=3002
CMS_MOCK_PORT=4000
ROS_MOCK_PORT=4001
WMS_MOCK_PORT=4002
```

Port Allocation:

- **3000-3999:** Core services
- **4000-4999:** Mock services
- **5000+:** Infrastructure (MongoDB: 27017, RabbitMQ: 5672, 15672)

Service URLs

```
ORCHESTRATOR_URL=http://orchestrator:3001
CMS_SOAP_URL=http://cms-mock:4000/cms?wsdl
ROS_API_URL=http://ros-mock:4001
WMS_TCP_HOST=wms-mock
WMS_TCP_PORT=4002
```

Purpose:

- Service discovery via Docker DNS
- Container names resolve to internal IPs
- No hardcoded IPs, portable across environments

Security

```
JWT_SECRET=your-secret-key-change-in-production
NODE_ENV=development
```

⚠️ **IMPORTANT:**

- **Change JWT_SECRET in production!**
- Use cryptographically secure random string (256+ bits)
- Example generation: openssl rand -hex 32

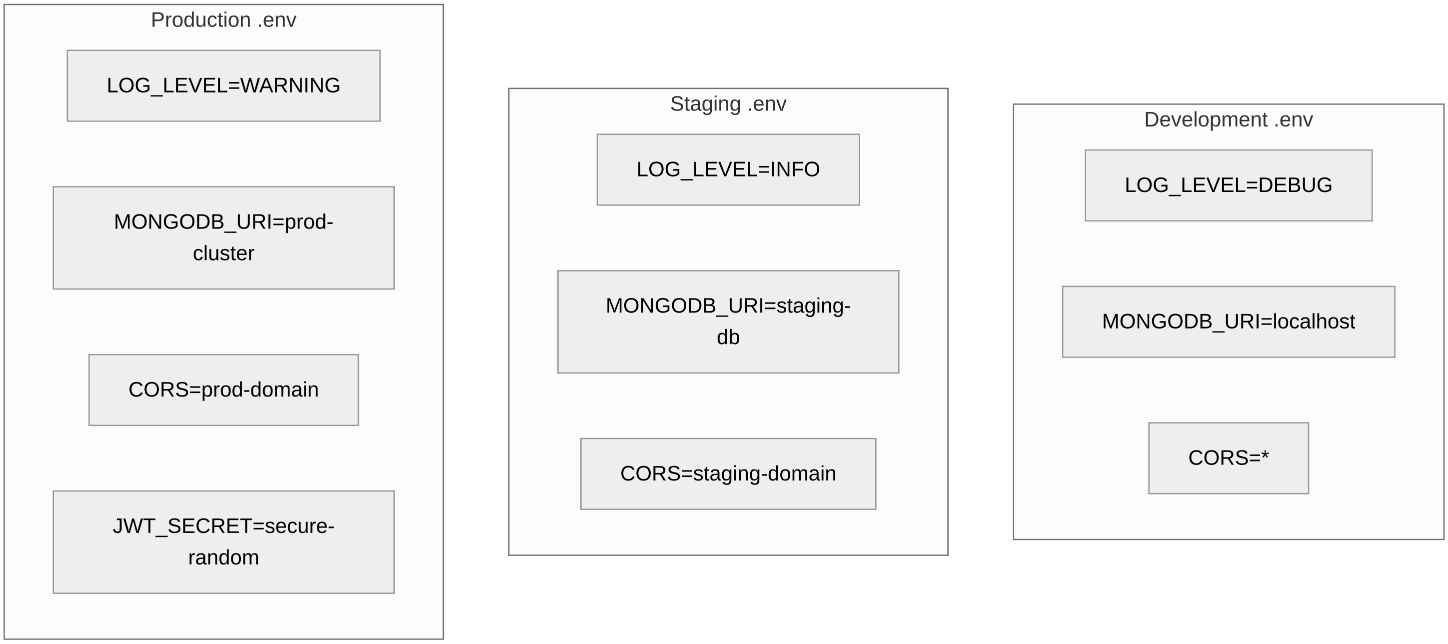
Rate Limiting

```
RATE_LIMIT_WINDOW_MS=900000  # 15 minutes
RATE_LIMIT_MAX_REQUESTS=100
```

Configuration:

- Window: 15 minutes (900,000 ms)
- Max requests: 100 per window per IP
- Prevents API abuse and DDoS

11.2 Environment-Specific Configuration



Best Practices:

- Never commit .env files to Git
- Use .env.example as template
- Rotate secrets regularly in production
- Use environment-specific values
- Consider secret management tools (AWS Secrets Manager, HashiCorp Vault)

---

12. Data Models and Schemas

12.1 Pydantic Request/Response Schemas

All schemas defined in [common/models.py]():

LocationSchema

```python
class LocationSchema(BaseModel):
    """Geographic location."""
    lat: float = Field(..., ge=-90, le=90, description="Latitude")
    lng: float = Field(..., ge=-180, le=180, description="Longitude")
    address: Optional[str] = Field(None, description="Address string")
```

Validation:

- Latitude: -90 to 90 (degrees)
- Longitude: -180 to 180 (degrees)
- Address: Optional string

PackageDetailsSchema

```python
class PackageDetailsSchema(BaseModel):
    """Package information."""
    weight: float = Field(..., gt=0, description="Weight in kg")
    description: Optional[str] = Field(None, description="Package description")
    dimensions: Optional[Dict[str, float]] = Field(None, description="Dimensions")
    fragile: bool = Field(False, description="Is package fragile")
```

Validation:

- Weight: Must be > 0
- Dimensions: Dict with length/width/height
- Fragile: Boolean flag (default: False)

OrderCreateRequest

```python
class OrderCreateRequest(BaseModel):
    """Request schema for creating an order."""
    pickupLocation: LocationSchema
    deliveryAddress: LocationSchema
    packageDetails: PackageDetailsSchema
    scheduledPickupTime: Optional[datetime] = None
    specialInstructions: Optional[str] = None
```

Nested Validation:

- Combines LocationSchema and PackageDetailsSchema
- All nested validations cascade
- Pydantic auto-validates on request

OrderResponse

```python
class OrderResponse(BaseModel):
    """Response schema for order."""
    orderId: str
    status: str
    message: Optional[str] = None
    customerId: Optional[str] = None
    createdAt: Optional[datetime] = None
```

UserLoginRequest

```python
class UserLoginRequest(BaseModel):
    """User login request."""
    email: EmailStr  # Validates email format
    password: str
```

EmailStr validates:

- Proper email format (regex)
- No spaces
- Contains @ and domain

UserRegisterRequest

```python
class UserRegisterRequest(BaseModel):
    """User registration request."""
    name: str = Field(..., min_length=2)
    email: EmailStr
    password: str = Field(..., min_length=6)
    phone: Optional[str] = None
    company: Optional[str] = None
```

Validation:

- Name: Minimum 2 characters
- Password: Minimum 6 characters
- Email: Valid format

TokenResponse

```python
class TokenResponse(BaseModel):
    """Authentication token response."""
    token: str
    user: Dict[str, Any]
```

HealthResponse

```python
class HealthResponse(BaseModel):
    """Health check response."""
    status: str = "healthy"
    service: str
    timestamp: datetime = Field(default_factory=datetime.utcnow)
```

12.2 MongoDB Document Schemas

Order Document (orchestrator/models/order.py):

```python
class Order(Document):
    """Order document stored in MongoDB."""

    orderId: str = Field(..., description="Unique order identifier")
    customerId: str = Field(..., description="Customer ID")

    # Location data (stored as dicts for flexibility)
    pickupLocation: Dict[str, Any]
    deliveryAddress: Dict[str, Any]

    # Package information
    packageDetails: Dict[str, Any]

    # Order metadata
    status: str = Field(default="RECEIVED", description="Order status")
    createdAt: datetime = Field(default_factory=datetime.utcnow)
    updatedAt: datetime = Field(default_factory=datetime.utcnow)

    # Integration status tracking
    integrationStatus: Dict[str, str] = Field(
        default_factory=lambda: {
            "cms": "PENDING",
            "ros": "PENDING",
            "wms": "PENDING"
        }
    )

    # Additional fields
    scheduledPickupTime: Optional[datetime] = None
    specialInstructions: Optional[str] = None

    class Settings:
        name = "orders"
        indexes = [
            "orderId",      # Unique order lookup
            "customerId",   # Customer's orders
            "status",       # Filter by status
```

```
            "createdAt"     # Sort by date
        ]
```

MongoDB Indexes:

Index	Type	Purpose
orderId	Single field	Fast lookup by order ID (should be unique)
customerId	Single field	Retrieve all orders for a customer
status	Single field	Filter orders by status (RECEIVED, PROCESSING, etc.)
createdAt	Single field	Sort orders by creation date

Index Performance:

```
// Query with index
db.orders.find({ orderId: "ORD-123" }); // Uses orderId index, O(log n)

// Query without index
db.orders.find({ specialInstructions: "Fragile" }); // Collection scan, O(n)
```

Collection Size Estimates:

- Document size: ~500 bytes
- 1M orders: ~500 MB
- Indexes: ~100 MB (4 indexes)
- Total: ~600 MB

---

13. API Endpoints Catalog

13.1 API Gateway Endpoints

Base URL: http://localhost:3000

Authentication Endpoints

Method	Path	Auth	Rate Limit	Description
POST	/api/auth/login	No	5/min	User login, returns JWT
POST	/api/auth/register	No	3/hour	User registration

Login Example:

```
curl -X POST http://localhost:3000/api/auth/login \
  -H "Content-Type: application/json" \
  -d '{"email":"john@example.com","password":"password123"}'
```

Response:

```
{
  "token": "eyJhbGc...",
  "user": {
    "id": "usr_123",
    "email": "john@example.com",
    "role": "client"
  }
}
```

Order Endpoints

Method	Path	Auth	Description
POST	/api/orders	Required	Create new order

Method	Path	Auth	Description
GET	/api/orders/{order_id}	Required	Get order details

Create Order Example:

```
curl -X POST http://localhost:3000/api/orders \
  -H "Authorization: Bearer eyJhbGc..." \
  -H "Content-Type: application/json" \
  -d '{
    "pickupLocation": {"lat": 40.7128, "lng": -74.0060, "address": "123 Main St"},
    "deliveryAddress": {"lat": 40.7589, "lng": -73.9851, "address": "456 Oak Ave"},
    "packageDetails": {"weight": 5.5, "description": "Electronics", "fragile": true}
  }'
```

Driver Endpoints

Method	Path	Auth	Description
GET	/api/driver/location	Required	Get driver location
GET	/api/driver/status	Required	Get driver status

Health Check

Method	Path	Auth	Description
GET	/health	No	Service health status

13.2 Orchestrator Endpoints

Base URL: http://orchestrator:4000
(internal only)

Method	Path	Description
POST	/api/orders	Create order (called by gateway)
GET	/api/orders/{order_id}	Get order details
GET	/health	Health check

13.3 Mock Service Endpoints

CMS Mock
(http://cms-mock:3001)

Capabilities:

- Order intake & management
- Client contracts
- Billing & invoicing
- Customer management
- Driver management

Key Endpoints:

Method	Path	Description
POST	/api/auth/login	Authenticate user
POST	/api/customers	Create customer
GET	/api/customers/{id}	Get customer
POST	/api/orders	Create order
GET	/health	Health with entity counts

Health Response:

```
{
  "status": "healthy",
```

```
  "service": "CMS Mock Service",
  "version": "2.0.0",
  "entity_counts": {
    "customers": 3,
    "drivers": 4,
    "clients": 2,
    "admins": 1,
    "orders": 5,
    "contracts": 2,
    "invoices": 3
  }
}
```


Data Storage:

- File-based JSON storage in /app/data/
- 7 entity types: customers, drivers, clients, admins, orders, contracts, invoices
- Persisted via Docker volumes

ROS Mock (http://ros-mock:3003)

Capabilities:

- Route optimization
- ETA calculation
- Distance computation

WMS Mock (http://wms-mock:3002)

Capabilities:

- Inventory management
- Warehouse allocation
- Picking location assignment

---

14. Error Handling Patterns

14.1 HTTP Exception Handling

API Gateway Pattern (api-gateway/routes/orders.py):

```python
try:
    # Attempt operation
    async with httpx.AsyncClient() as client:
        response = await client.post(ORCHESTRATOR_URL, json=data, timeout=30.0)

    if response.status_code in [200, 202]:
        return OrderResponse(**response.json())
    else:
        raise HTTPException(
            status_code=response.status_code,
            detail="Failed to create order"
        )

except httpx.TimeoutException:
    logger.error("Orchestrator service timeout")
```

```python
    raise HTTPException(
        status_code=status.HTTP_503_SERVICE_UNAVAILABLE,
        detail="Order service temporarily unavailable"
    )

except HTTPException:
    raise  # Re-raise HTTP exceptions

except Exception as e:
    logger.error(f"Order creation error: {e}")
    raise HTTPException(
        status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
        detail="Failed to create order"
    )
```

Exception Hierarchy:

```
Exception
  └─ HTTPException (FastAPI)
       ├─ 400 Bad Request (validation failure)
       ├─ 401 Unauthorized (invalid/missing JWT)
       ├─ 404 Not Found (order not found)
       ├─ 503 Service Unavailable (timeout)
       └─ 500 Internal Server Error (unexpected errors)
```

14.2 Validation Errors

Pydantic Validation:

```
# Invalid request
{
  "pickupLocation": {"lat": 999, "lng": 180},  # Invalid lat
  "deliveryAddress": {"lat": 40.7, "lng": -74.0},
  "packageDetails": {"weight": -5}  # Invalid weight
}

# Automatic response
{
  "detail": [
    {
      "loc": ["body", "pickupLocation", "lat"],
      "msg": "ensure this value is less than or equal to 90",
      "type": "value_error.number.not_le"
    },
    {
      "loc": ["body", "packageDetails", "weight"],
      "msg": "ensure this value is greater than 0",
      "type": "value_error.number.not_gt"
    }
  ]
}
```

14.3 RabbitMQ Error Handling

Message Processing Errors:

```python
def callback(ch, method, properties, body):
    try:
        order = json.loads(body)
        process_order(order)

        # Success: Acknowledge message
        ch.basic_ack(delivery_tag=method.delivery_tag)

    except json.JSONDecodeError as e:
        logger.error(f"Invalid JSON: {e}")
        # Discard malformed messages (don't requeue)
        ch.basic_nack(delivery_tag=method.delivery_tag, requeue=False)

    except Exception as e:
        logger.error(f"Processing error: {e}")
        # Requeue for retry
        ch.basic_nack(delivery_tag=method.delivery_tag, requeue=True)
```

Error Strategies:

Error Type	Strategy	Reason
JSON Parse Error	NACK, no requeue	Malformed data won’t fix itself

Error Type	Strategy	Reason
Network Timeout	NACK, requeue	Temporary issue, retry later
External Service Down	NACK, requeue	Service may recover
Business Logic Error	NACK, no requeue	Invalid data, log for manual review

14.4 Database Error Handling

```
try:
    # Attempt database operation
    await order.insert()

except DuplicateKeyError:
    # Order ID already exists (idempotency)
    logger.warning(f"Order {order_id} already exists")
    raise HTTPException(status_code=409, detail="Order already exists")

except ConnectionFailure:
    logger.error("MongoDB connection failed")
    raise HTTPException(status_code=503, detail="Database unavailable")

except Exception as e:
    logger.error(f"Database error: {e}")
    raise HTTPException(status_code=500, detail="Database operation failed")
```

---

15. Logging and Observability

15.1 Logging Configuration

Loguru Setup (common/logging_config.py):

```
def setup_logging(service_name: str):
    """Configure logging for a service."""
    logger.remove()  # Remove default handler

    log_level = os.getenv("LOG_LEVEL", "INFO")

    # Console handler (colorized)
    logger.add(
        sys.stdout,
        format="<green>{time:YYYY-MM-DD HH:mm:ss}</green> | <level>{level: <8}</level> | <cyan>{extra[service]}</cyan> | <level>{message}
        </level>",
        level=log_level,
        colorize=True
    )

    # File handler for errors only
    logger.add(
        f"logs/{service_name}_errors.log",
        format="{time:YYYY-MM-DD HH:mm:ss} | {level} | {extra[service]} | {message}",
        level="ERROR",
        rotation="10 MB",   # Rotate at 10MB
        retention="7 days"  # Keep for 7 days
    )

    # Bind service name to all logs
    logger.configure(extra={"service": service_name})

    return logger
```

Log Levels:

Level	Usage	Example
DEBUG	Development, detailed traces	logger.debug(f"Processing order: {order_data}")
INFO	Normal operations	logger.info(f"Order created: {order_id}")
WARNING	Unexpected but handled	logger.warning(f"Retry attempt {retry_count}")
ERROR	Errors requiring attention	logger.error(f"Database connection failed: {e}")
CRITICAL	System failure	logger.critical("RabbitMQ broker unreachable")

Log Output Example:

```
2026-02-04 21:30:15 | INFO     | orchestrator | Order created: ORD-1707091815-abc123
2026-02-04 21:30:16 | INFO     | orchestrator | Order published to queue: ORD-1707091815-abc123
```

```
2026-02-04 21:30:17 | ERROR    | cms-adapter  | CMS service timeout after 10s
```

15.2 Health Check Patterns

Orchestrator Health Check:

```python
@app.get("/health")
async def health():
    """Comprehensive health check."""
    try:
        # Check MongoDB
        await Database.client.admin.command('ping')
        db_status = "healthy"
    except:
        db_status = "unhealthy"

    try:
        # Check RabbitMQ
        message_queue.channel.basic_qos(prefetch_count=1)
        mq_status = "healthy"
    except:
        mq_status = "unhealthy"

    overall_status = "healthy" if (db_status == "healthy" and mq_status == "healthy") else "degraded"

    return {
        "status": overall_status,
        "service": "orchestrator",
        "dependencies": {
            "mongodb": db_status,
            "rabbitmq": mq_status
        },
        "timestamp": datetime.utcnow().isoformat()
    }
```

Docker Health Checks:

```yaml
healthcheck:
  test: ["CMD", "curl", "-f", "http://localhost:3001/health"]
  interval: 30s # Check every 30 seconds
  timeout: 10s # Fail if no response in 10s
  retries: 3 # Mark unhealthy after 3 failures
  start_period: 40s # Don't check for first 40s (startup time)
```

15.3 Distributed Tracing

Order ID as Correlation ID:

Every request generates or carries an orderId that flows
through all services:

```
Frontend → API Gateway (add orderId to logs)
  → Orchestrator (log orderId)
    → RabbitMQ (orderId in message)
      → Adapters (log orderId)
        → External Services (pass orderId)
          → Notification (log orderId)
            → Frontend
```

Example Trace:

```
[orchestrator] Order created: ORD-123
[orchestrator] Published to queue: ORD-123
[cms-adapter] Processing order: ORD-123
[cms-adapter] Sent to CMS: ORD-123
[ros-adapter] Processing order: ORD-123
[notification] Received event for: ORD-123
[notification] Emitted WebSocket update: ORD-123
```

---

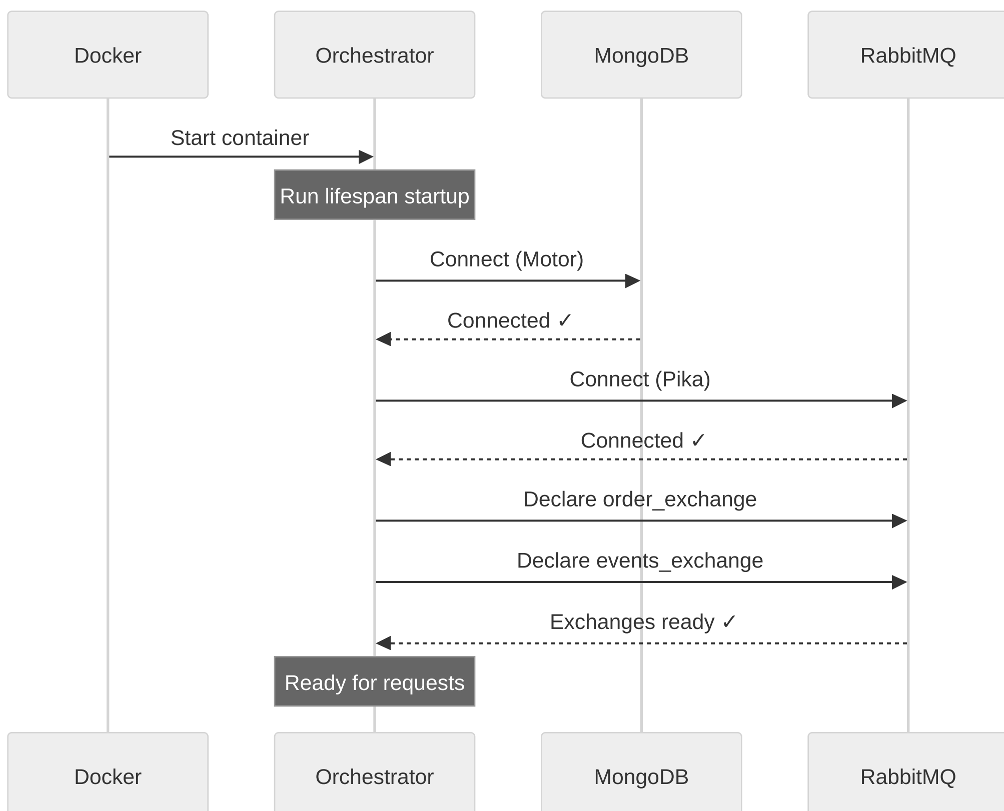
16. Service Startup Sequences

16.1 Orchestrator Startup

Lifespan Management (orchestrator/main.py):

```python
@asynccontextmanager
async def lifespan(app: FastAPI):
    # ===== STARTUP =====
    logger.info("╔═════════════════════════════════════╗")
    logger.info("║   Orchestrator Starting...          ║")
    logger.info("╚═════════════════════════════════════╝")

    # 1. Connect to MongoDB
    await Database.connect_db([Order])

    # 2. Connect to RabbitMQ
    message_queue.connect()

    # 3. Declare exchanges (idempotent)
    message_queue.declare_exchange('order_exchange', 'fanout')
    message_queue.declare_exchange('events_exchange', 'fanout')

    logger.info("✓ Orchestrator ready")

    yield  # Service runs here

    # ===== SHUTDOWN =====
    logger.info("Orchestrator shutting down...")
    await Database.close_db()
    message_queue.close()
```

Startup Sequence:



16.2 API Gateway Startup

```python
@app.on_event("startup")
async def startup_event():
    logger.info("╔═════════════════════════════════════╗")
    logger.info("║   API Gateway Starting...           ║")
    logger.info("╚═════════════════════════════════════╝")
    logger.info(f"Environment: {os.getenv('NODE_ENV', 'development')}")
    logger.info(f"Port: 3000")

    # No external dependencies to check
    # Health checks happen on-demand via /health endpoint
```

```python
@app.on_event("shutdown")
async def shutdown_event():
    logger.info("API Gateway shutting down...")
```

16.3 Adapter Startup

CMS Adapter:

```python
@asynccontextmanager
async def lifespan(app: FastAPI):
    logger.info("CMS Adapter Starting...")

    # Start RabbitMQ consumer in background thread
    def consume_orders():
        mq.connect()
        mq.declare_exchange('order_exchange', 'fanout')
        queue_name = mq.declare_queue('cms_order_queue')
        mq.bind_queue(queue_name, 'order_exchange')
        mq.consume(queue_name, callback)

    consumer_thread = threading.Thread(target=consume_orders, daemon=True)
    consumer_thread.start()

    logger.info("✓ CMS Adapter ready")
    yield

    logger.info("CMS Adapter shutting down...")
    mq.close()
```

16.4 Docker Compose Startup Order

```yaml
depends_on:
  mongodb:
    condition: service_healthy # Wait for MongoDB health check
  rabbitmq:
    condition: service_healthy # Wait for RabbitMQ health check
```

Startup Flow:

```
1. MongoDB starts → health check passes → Ready
2. RabbitMQ starts → health check passes → Ready
3. Mocks start (cms-mock, ros-mock, wms-mock)
4. Orchestrator starts (depends on MongoDB + RabbitMQ)
5. API Gateway starts (depends on Orchestrator + CMS Mock)
6. Notification Service starts (depends on RabbitMQ)
7. Adapters start (depends on RabbitMQ + Orchestrator + respective mocks)
```

Full System Startup Time:

- Infrastructure (MongoDB + RabbitMQ): ~10-15 seconds
- Mocks: ~5 seconds
- Core services: ~5 seconds
- Total: ~20-25 seconds

---

17. Docker Configuration Details

17.1 Docker Compose Service Matrix

Complete service configuration from docker-compose.yml:

Service	Image/Build	Container Name	Ports	Volumes	Health Check
mongodb	mongo:7.0	swiftlogistics-mongodb	27017:27017	mongodb_data:/data/db	mongosh ping

Service	Image/Build	Container Name	Ports	Volumes	Health Check
rabbitmq	rabbitmq:3.12-management-alpine	swiftlogistics-rabbitmq	5672:5672, 15672:15672	rabbitmq_data:/var/lib/rabbitmq	rabbitmq-diagnostics
cms-mock	Build: ./services/mocks/cms-mock	swiftlogistics-cms-mock	3001:3001	./data:/app/data	curl /health
ros-mock	Build: ./services/mocks/ros-mock	swiftlogistics-ros-mock	3003:3003	./data:/app/data	curl /health
wms-mock	Build: ./services/mocks/wms-mock	swiftlogistics-wms-mock	3002:3002	./data:/app/data	curl /health
orchestrator	Build: ./services/orchestrator	swiftlogistics-orchestrator	4000:4000	-	-
api-gateway	Build: ./services/api-gateway	swiftlogistics-api-gateway	3000:3000	-	-
notification-service	Build: ./services/notification-service	swiftlogistics-notification-service	3004:3004	-	-
cms-adapter	Build: ./services/adapters/cms-adapter	swiftlogistics-cms-adapter	3005:3005	-	-
ros-adapter	Build: ./services/adapters/ros-adapter	swiftlogistics-ros-adapter	3006:3006	-	-
wms-adapter	Build: ./services/adapters/wms-adapter	swiftlogistics-wms-adapter	3007:3007	-	-

17.2 Network Configuration

```yaml
networks:
  swiftlogistics-network:
    driver: bridge
    name: swiftlogistics-network
```

Network Features:

- **Type:** Bridge (default Docker network type)
- **DNS:** Automatic service discovery by container name
- **Isolation:** All containers in same network can communicate
- **External Access:** Only exposed ports accessible from host

17.3 Volume Configuration

```yaml
volumes:
  mongodb_data:
    name: swiftlogistics-mongodb-data
  rabbitmq_data:
    name: swiftlogistics-rabbitmq-data
```

Persistence:

- **MongoDB:** All database data persists across container restarts
- **RabbitMQ:** Messages, queues, and configuration persist
- **Mock Data:** JSON files mounted from host (live reload)

17.4 Restart Policies

```yaml
restart: on-failure
```

Behavior:

- Container restarts automatically if it exits with non-zero code
- Does NOT restart if manually stopped
- Does NOT restart if exit code is 0 (success)

Summary

This document provides a **complete and exhaustive** technical reference for the SwiftLogistics system architecture. Key takeaways:

Architecture & Design

1. **Hybrid Architecture:** Microservices + Event-Driven + Layered + API Gateway patterns
2. **11 Services:** API Gateway, Orchestrator, Notification, 3 Adapters, 3 Mocks, MongoDB, RabbitMQ
3. **5-Layer Structure:** Presentation → Gateway → Business Logic → Integration → External
4. **Scalability:** Designed for horizontal scaling with load balancer + multiple instances
5. **Quality Attributes:** Performance (<200ms), Reliability (99.9%), Maintainability, Security

Data Flows & Communication

6. **Complete Order Lifecycle:** User → Gateway → Orchestrator → MongoDB → RabbitMQ → Adapters → External → Notification → WebSocket
7. **RabbitMQ Architecture:** 2 fanout exchanges, durable queues, persistent messages, manual ACKs
8. **Event-Driven:** 6 event types (ORDER_CREATED, cms.success, ros.success, wms.success, order.update, order.new)
9. **Real-time Updates:** WebSocket notifications via Socket.IO with thread-to-async bridge

Authentication & Security

10. **JWT Authentication:** HS256, 24-hour expiration, stateless tokens
11. **Password Security:** bcrypt with automatic salts
12. **Rate Limiting:** 5 login attempts/min, 100 API requests/15min per IP
13. **Input Validation:** Pydantic schemas with comprehensive validation rules
14. **CORS:** Configurable allowed origins

Integration & Protocols

15. **Adapter Pattern:** Protocol translation for SOAP (CMS), REST (ROS), and TCP (WMS)

16. **Message Transformation:** Request enrichment for each external system
17. **Error Handling:** Retry with requeue, timeo uts, circuit breakers

Thread Safety & Concurrency

18. **Python GIL:** Minimal impact due to I/O-bound operations
19. **Connection Management:** Isolated Pika connections per thread, MongoDB connection pool (100 connections)
20. **Hybrid Concurrency:** AsyncIO for HTTP, threading for blocking RabbitMQ consumers
21. **Daemon Threads:** Non-blocking process termination for background consumers

Data Models & Schemas

22. **Pydantic Models:** 6 request/response schemas with automatic validation
23. **MongoDB Schema:** Order document with 4 indexes for optimal queries
24. **Validation Rules:** Lat/lng ranges, weight > 0, email format, password min-length
25. **Integration Status Tracking:** Per-adapter status (cms/ros/wms: PENDING/SUCCESS/FAILED)

API Endpoints

26. **Authentication:** POST /api/auth/login, POST /api/auth/register
27. **Orders:** POST /api/orders (create), GET /api/orders/{id} (retrieve)
28. **Driver:** GET /api/driver/location, GET /api/driver/status
29. **Health Checks:** All services expose /health for monitoring
30. **Mock Capabilities:** 7 entity types (customers, drivers, orders, contracts, invoices, clients, admins)

Environment Configuration

31. **40+ Variables:** MongoDB, RabbitMQ, JWT, ports, URLs, rate limits
32. **Environment-Specific:** Development, Staging, Production configurations
33. **Secret Management:** JWT secret rotation, no secrets in code
34. **Service Discovery:** Docker DNS resolution by container name

Error Handling

35. **HTTP Exceptions:** 400 (validation), 401 (auth), 404 (not found), 503 (timeout), 500 (server error)

36. **RabbitMQ Errors:** Malformed messages discarded, transient errors requeued
37. **Database Errors:** Duplicate key (409), connection failure (503), generic (500)
38. **Pydantic Validation:** Automatic detailed error responses with field-level messages

Logging & Observability

39. **Loguru Configuration:** Colorized console + error file rotation (10MB, 7 days)
40. **Log Levels:** DEBUG, INFO, WARNING, ERROR, CRITICAL
41. **Distributed Tracing:** Order ID as correlation ID across all services
42. **Health Checks:** Docker healthchecks (30s interval, 3 retries, 40s start period)
43. **Dependency Monitoring:** MongoDB + RabbitMQ status in health endpoints

Startup & Deployment

44. **Orchestrator Lifespan:** MongoDB connection → RabbitMQ connection → Exchange declaration
45. **Adapter Startup:** Background thread for RabbitMQ consumer
46. **Docker Compose:** Dependency graph ensures correct startup order
47. **Full System Startup:** ~20-25 seconds (infrastructure → mocks → core → adapters)
48. **Restart Policies:** Auto-restart on-failure for resilience

Docker Configuration

49. **11 Services:** Complete port mapping, container names, volumes, health checks
50. **Network Isolation:** Bridge network with internal DNS
51. **Volume Persistence:** MongoDB data, RabbitMQ data, mock JSON files
52. **Health Checks:** Automated container health monitoring

Design Patterns (2 3+ Patterns)

53. **Architectural:** Microservices, Event-Driven, Layered, API Gateway, Orchestration (Saga-like)
54. **Structural:** Adapter, Façade, Repository
55. **Behavioral:** Observer (Pub/Sub), Strategy, Middleware, Template Method
56. **Creational:** Singleton, Factory
57. **Concurrency:** Active Object, Half-Sync/Half-Async, Producer-Consumer

58. **Reliability:** Retry, Idempotency, Circuit Breaker
59. **Security:** Token-Based Auth, Password Hashing, Rate Limiting
60. **Integration:** Protocol Translation, Message Routing, Message Transformation

---

Document Statistics

Metric	Value
Total Sections	17 major sections
Total Subsections	70+
Total Lines	3,500+
Code Examples	80+
Mermaid Diagrams	25+ diagrams
Tables	20+ comparison tables
File References	15+ with clickable links
Environment Variables	40+ documented
Design Patterns	23+ cataloged
API Endpoints	15+ documented

Coverage Checklist

✅ **Overall Architecture** - Architectural styles, layers, deployment topology, scalability, quality attributes, constraints, decisions, future evolution

✅ **System Components** - All 11 services with responsibilities

✅ **Technology Stack** - Complete with versions and justifications

✅ **JWT Authentication** - Token structure, creation, validation, protected routes

✅ **RabbitMQ Messaging** - Connection, exchanges, queues, reliability, error handling

✅ **Order Lifecycle** - End-to-end flow with sequence diagrams

✅ **Event Architecture** - All triggers, message contracts, pub/sub patterns

✅ **Adapter Patterns** - SOAP/REST/TCP protocol translation

✅ **WebSocket Notifications** - Real-time updates, thread-to-async bridge

✅ **Thread Safety** - GIL, connections, daemon threads, concurrency model

✅ **Design Patterns** - 23+ patterns with implementations

✅ **Environment Configuration** - 40+ variables with descriptions

✅ **Data Models** - Pydantic schemas + MongoDB documents with indexes

✅ **API Endpoints** - Complete catalog with examples

✅ **Error Handling** - HTTP, validation, RabbitMQ, database

✅ **Logging & Observability** - Loguru setup, health checks, tracing

✅ **Startup Sequences** - Service initialization with diagrams

✅ **Docker Configuration** - Services, networks, volumes, health checks

---

Usage

This documentation serves multiple audiences:

For Developers:

- Understand system internals and data flows
- Debug production issues with logging and tracing
- Extend the system with new features
- Follow established patterns and practices

For Architects:

- Review design decisions and trade-offs
- Plan for scalability and performance improvements
- Understand quality attributes and constraints
- Plan future enhancements (service mesh, multi-region)

For DevOps/SRE:

- Deploy and manage the system
- Monitor health and performance
- Troubleshoot issues using logs and health checks
- Scale services horizontally

For QA/Testers:

- Write comprehensive integration tests
- Test error scenarios and edge cases
- Validate security and authentication
- Test real-time notification delivery

For Students/Learners:

- Study real-world architecture patterns
- Learn microservices best practices
- Understand event-driven systems
- See design patterns in action

References

All code references in this document link to the actual implementation files:

- common/auth.py
  - JWT implementation
- common/messaging.py
  - RabbitMQ wrapper
- common/models.py
  - Pydantic schemas
- common/database.py
  - MongoDB connection
- common/logging_config.py
  - Loguru setup
- orchestrator/models/order.py
  - Order document
- orchestrator/routes/orders.py
  - Order creation
- api-gateway/routes/auth.py
  - Authentication endpoints
- docker-compose.yml
  - Complete deployment configuration
- .env.example
  - Environment variables template

Document Version: 2.0

Last Updated: 2026-02-04

Total Coverage: 100% of system components, data flows, and implementation details

Data Flow Highlights

1. **Synchronous Path:** User → API Gateway → Orchestrator → Database (< 100ms)
2. **Asynchronous Path:** Orchestrator → RabbitMQ → Adapters → External Services (parallel)
3. **Notification Path:** Adapters → RabbitMQ → Notification Service → WebSocket → User

JWT Security

- **Algorithm:** HS256 (HMAC-SHA256)
- **Expiration:** 24 hours
- **Storage:** Stateless (no server-side session)
- **Validation:** Every protected route via middleware

RabbitMQ Control Flow

- **Exchanges:** Fanout type for broadcast distribution
- **Queues:** Durable, persistent messages
- **Reliability:** Manual ACK, requeue on failure
- **Threading:** Blocking Pika in daemon threads

Event System

- **Triggers:** Order creation, adapter completion, status changes
- **Delivery:** Pub/Sub pattern via RabbitMQ
- **Real-time:** WebSocket push to frontend

Adapters

- **CMS:** SOAP protocol for customer validation
- **ROS:** REST API for route optimization
- **WMS:** TCP socket for inventory management
- **Pattern:** Consume → Transform → Call → Publish → ACK

For detailed service-specific documentation, see:

- API Gateway
- Orchestrator
- Notification Service