PeeGeeQ Complete Guide

Welcome to **PeeGeeQ** (PostgreSQL as a Message Queue) - a production-ready message queue system built on PostgreSQL that provides both high-performance real-time messaging and transactional messaging patterns.

This guide takes you from complete beginner to production-ready implementation with progressive examples and detailed explanations.

> ** Need Technical Reference?** For detailed API specifications, database schema, and architectural details, see the PeeGeeQ Architecture & API Reference.

Table of Contents

Part I: Understanding Message Queues

What is a Message Queue?

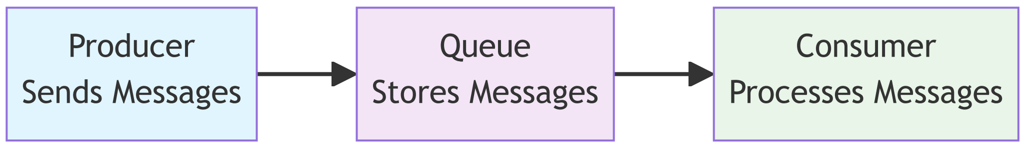
A message queue is a communication method used in software architecture where applications send and receive messages asynchronously. Think of it as a reliable postal service for your applications.

Real-World Analogy

Imagine a message queue like a post office:

- **You (Producer)** drop letters (messages) in a mailbox (queue)
- **The postal service (Queue System)** stores and delivers them reliably
- **Recipients (Consumers)** receive letters from their mailboxes
- **Letters are delivered reliably**, even if recipients aren't home when they arrive

Core Components



- **Producer**: An application that sends messages
- **Queue**: A storage mechanism that holds messages temporarily
- **Consumer**: An application that receives and processes messages
- **Message**: A unit of data being transmitted (text, JSON, objects, etc.)

Message Lifecycle

1. **Send**: Producer creates and sends a message to a queue
2. **Store**: Queue safely stores the message until a consumer is ready
3. **Receive**: Consumer retrieves the message from the queue
4. **Process**: Consumer processes the message (business logic)
5. **Acknowledge**: Consumer confirms successful processing
6. **Remove**: Queue removes the processed message

Why Use Message Queues?

Message queues solve fundamental problems in distributed systems:

1. Decoupling Applications

```
❌ Without Message Queue (Tight Coupling):
[Order Service] ──directly calls──> [Email Service]
                ──directly calls──> [Inventory Service]
                ──directly calls──> [Payment Service]

✅ With Message Queue (Loose Coupling):
[Order Service] ──> [Queue] ──> [Email Service]
```

```
                                    ──> [Inventory Service]
                                    ──> [Payment Service]
```

Benefits:

- Services don't need to know about each other
- Services can be developed and deployed independently
- Adding new services doesn't require changing existing ones

2. Reliability & Fault Tolerance

- **Messages aren't lost** if a service is temporarily down
- **Automatic retry** mechanisms for failed processing
- **Dead letter queues** for messages that can't be processed

3. Scalability

- **Multiple consumers** can process messages in parallel
- **Load balancing** across consumer instances
- **Horizontal scaling** by adding more consumers

4. Asynchronous Processing

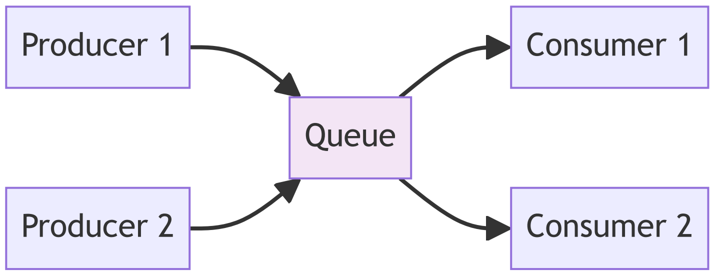
- **Producers don't wait** for consumers to process messages
- **Better user experience** - no blocking operations
- **Improved system responsiveness**

5. Traffic Smoothing

- **Handle traffic spikes** by queuing excess messages
- **Process at optimal rate** regardless of incoming load
- **Prevent system overload**

Message Queue Patterns Explained

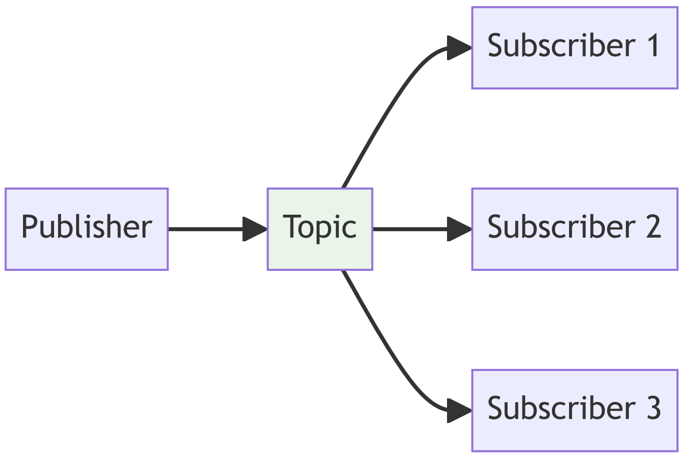
1. Point-to-Point (Queue Pattern)



- **One message** goes to **one consumer**
- **Load balancing** across multiple consumers

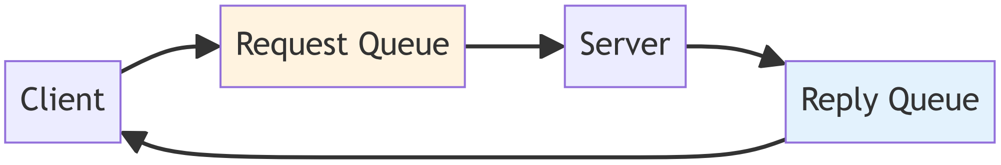
- **Competing consumers** pattern
- **Use case**: Order processing, task distribution

2. Publish-Subscribe (Topic Pattern)



- **One message** goes to **all subscribers**
- **Broadcasting** pattern
- **Event notification** pattern
- **Use case**: News feeds, notifications, event broadcasting

3. Request-Reply Pattern



- **Synchronous-like** communication over asynchronous queues
- **Correlation IDs** to match requests with replies
- **Use case**: RPC over messaging, distributed services

Traditional vs. Database-Based Queues

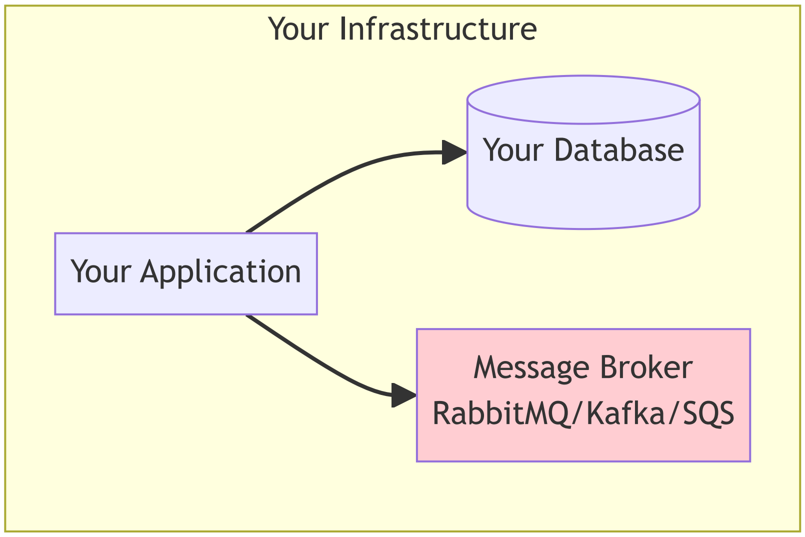
Traditional Message Brokers

Examples: RabbitMQ, Apache Kafka, Amazon SQS, Apache ActiveMQ

Characteristics:

- **Separate infrastructure** to manage and maintain
- **Specialized protocols** (AMQP, MQTT, etc.)
- **High performance** and feature-rich
- **Additional operational complexity**

- **Separate failure points**

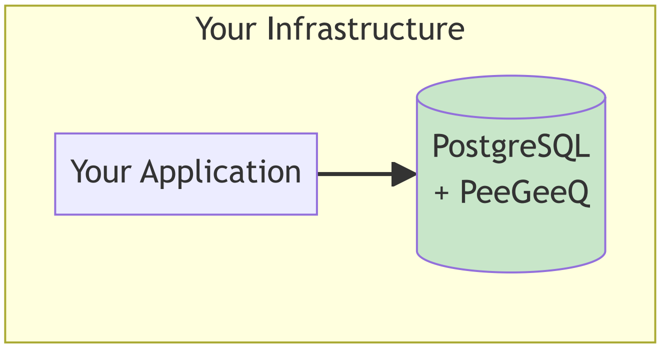


Database-Based Queues (PeeGeeQ Approach)

Examples: PeeGeeQ, AWS RDS with SQS integration

Characteristics:

- **Uses existing database** infrastructure
- **Transactional consistency** with business data
- **Simpler operational model**
- **Leverages database features** (ACID, replication, backup)
- **Single point of management**



Comparison Table

Aspect	Traditional Brokers	Database-Based (PeeGeeQ)
Infrastructure	Separate service	Uses existing database
Operational Complexity	High	Low

Aspect	Traditional Brokers	Database-Based (PeeGeeQ)
Transactional Consistency	Limited	Full ACID compliance
Learning Curve	Steep	Gentle (SQL knowledge)
Backup & Recovery	Separate process	Part of database backup
Monitoring	Separate tools	Database monitoring tools
High Availability	Complex setup	Database HA mechanisms
Performance	Very High	High (10k+ msg/sec)

When to Choose Each Approach

Choose Traditional Brokers When:

- **Extreme performance** requirements (100k+ msg/sec)
- **Complex routing** and transformation needs
- **Multiple protocols** required
- **Dedicated messaging team** available

Choose Database-Based (PeeGeeQ) When:

- **Transactional consistency** is critical
- **Operational simplicity** is important
- **Existing PostgreSQL** infrastructure
- **Team familiar with SQL** and databases
- **Moderate to high performance** needs (10k+ msg/sec)

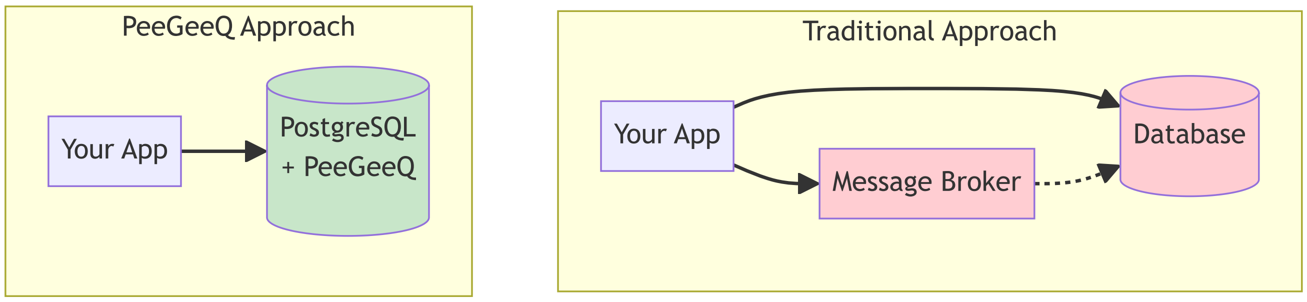
Part II: Understanding PeeGeeQ

What is PeeGeeQ?

PeeGeeQ (PostgreSQL as a Message Queue) is an enterprise-grade message queue system that transforms your existing PostgreSQL database into a powerful, production-ready message broker.

The Core Idea

Instead of adding another piece of infrastructure to your stack, PeeGeeQ leverages PostgreSQL's advanced features to provide enterprise-grade messaging capabilities:



Three Powerful Patterns in One System

1. Native Queue - Real-Time Performance

- **Performance**: 10,000+ messages/second, <10ms latency
- **Mechanism**: PostgreSQL LISTEN/NOTIFY with advisory locks
- **Use Case**: Real-time notifications, live updates, event streaming

2. Outbox Pattern - Transactional Reliability

- **Performance**: 5,000+ messages/second
- **Mechanism**: Database transactions with polling-based delivery
- **Use Case**: Order processing, financial transactions, critical business events

3. Bi-Temporal Event Store - Event Sourcing

- **Performance**: 3,000+ messages/second
- **Mechanism**: Append-only event log with temporal queries
- **Use Case**: Audit trails, event sourcing, historical analysis

PeeGeeQ's Unique Approach

What Makes PeeGeeQ Different?

1. Database-Native Design

PeeGeeQ isn't a wrapper around PostgreSQL - it's designed from the ground up to leverage PostgreSQL's strengths:

- **LISTEN/NOTIFY**: Real-time message delivery without polling
- **Advisory Locks**: Prevent duplicate message processing
- **Transactions**: ACID compliance with your business data
- **JSON/JSONB**: Native support for structured message payloads
- **Triggers**: Automatic message routing and processing

2. Zero Infrastructure Overhead

```
Traditional Setup:
✓ PostgreSQL Database
✓ Message Broker (RabbitMQ/Kafka)
✓ Monitoring for Database
✓ Monitoring for Message Broker
✓ Backup for Database
✓ Backup for Message Broker
✓ HA for Database
✓ HA for Message Broker

PeeGeeQ Setup:
✓ PostgreSQL Database (with PeeGeeQ)
```

3. Transactional Messaging

The killer feature - true transactional consistency:

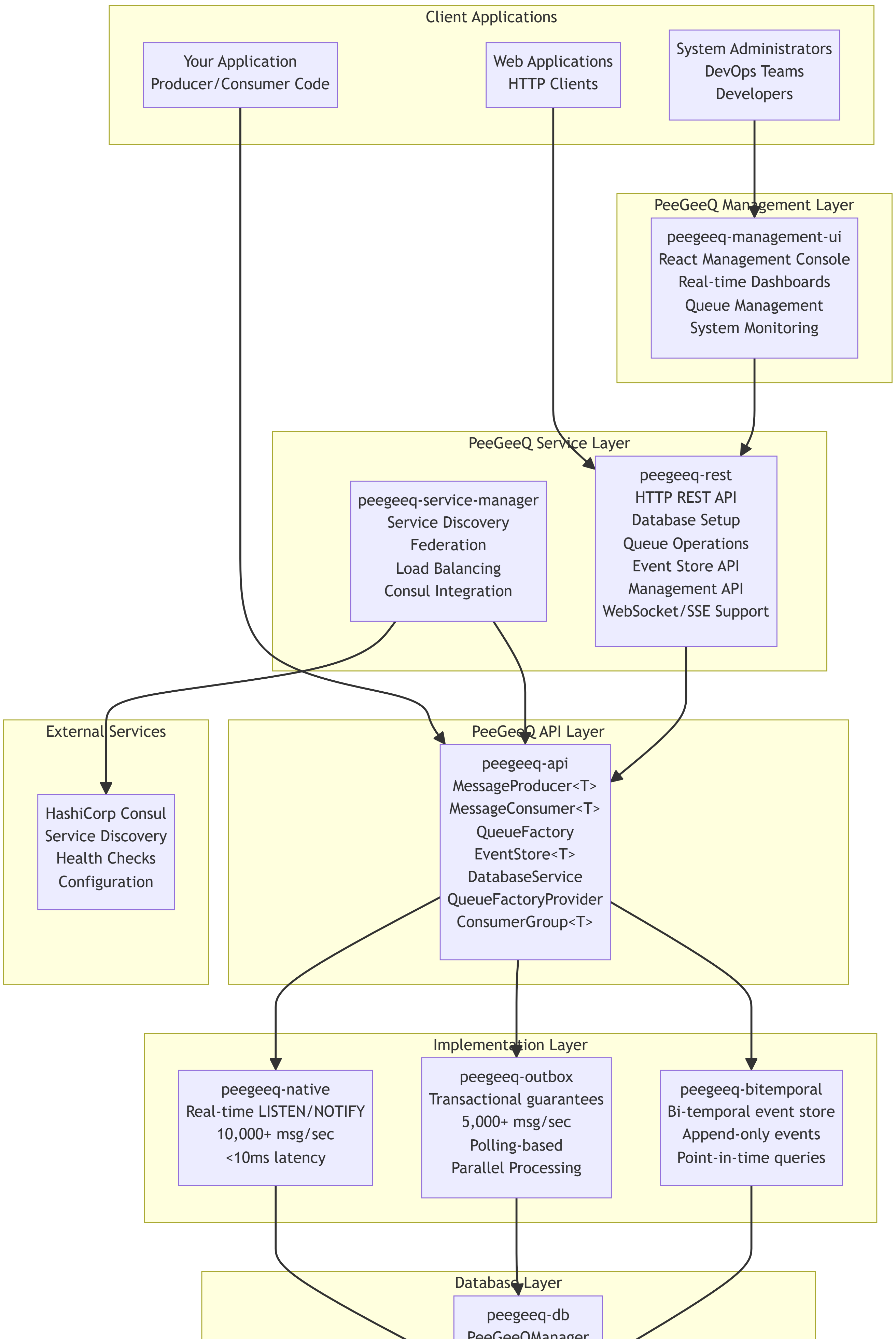
```java
// This is impossible with traditional message brokers
try (Connection conn = dataSource.getConnection()) {
    conn.setAutoCommit(false);

    // 1. Update business data
    updateOrderStatus(conn, orderId, "PAID");

    // 2. Send message (same transaction!)
    producer.send(new OrderPaidEvent(orderId));

    // 3. Both succeed or both fail together
    conn.commit();
}
```

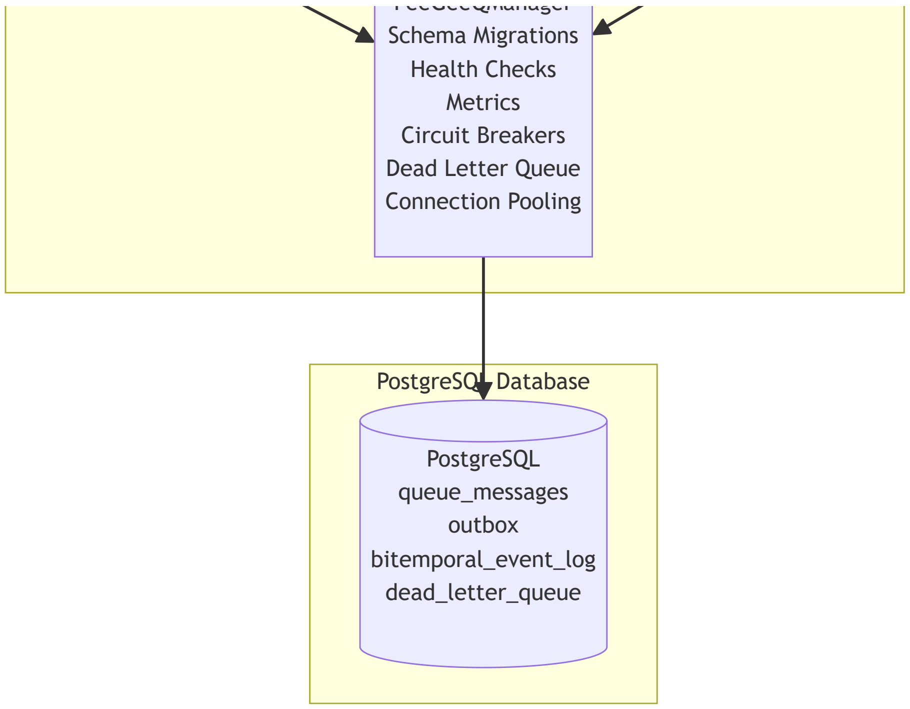
4. Familiar Technology Stack

- **SQL-based**: Use familiar SQL for queue management
- **PostgreSQL tools**: Existing monitoring, backup, and HA solutions work
- **Standard JDBC**: No new protocols or drivers to learn

Architecture Deep Dive

High-Level Architecture


Client Applications
Your Application
Producer/Consumer Code
Web Applications
HTTP Clients
System Administrators
DevOps Teams
Developers
PeeGeeQ Management Layer
peegeeq-management-ui
React Management Console
Real-time Dashboards
Queue Management
System Monitoring
PeeGeeQ Service Layer
peegeeq-service-manager
Service Discovery
Federation
Load Balancing
Consul Integration
peegeeq-rest
HTTP REST API
Database Setup
Queue Operations
Event Store API
Management API
WebSocket/SSE Support
External Services
HashiCorp Consul
Service Discovery
Health Checks
Configuration
PeeGeeQ API Layer
peegeeq-api
MessageProducer<T>
MessageConsumer<T>
QueueFactory
EventStore<T>
DatabaseService
QueueFactoryProvider
ConsumerGroup<T>
Implementation Layer
peegeeq-native
Real-time LISTEN/NOTIFY
10,000+ msg/sec
<10ms latency
peegeeq-outbox
Transactional guarantees
5,000+ msg/sec
Polling-based
Parallel Processing
peegeeq-bitemporal
Bi-temporal event store
Append-only events
Point-in-time queries
Database Layer
peegeeq-db
PeeGeeQManager



Module Breakdown

PeeGeeQ consists of **9 core modules** organized in a layered architecture:

> 📚 **For complete API specifications and technical details**, see the Module Structure section in the Architecture & API Reference.

peegeeq-api - Clean Abstractions

- **MessageProducer**: Type-safe message sending with correlation IDs and message groups
- **MessageConsumer**: Type-safe message receiving with parallel processing
- **QueueFactory**: Creates producers, consumers, and consumer groups
- **QueueFactoryProvider**: Factory registry and discovery with configuration templates
- **ConsumerGroup**: Load balancing and coordinated message processing
- **EventStore**: Bi-temporal event storage and querying

peegeeq-management-ui - Web-based Administration

- **React Management Console**: Modern web interface inspired by RabbitMQ's admin console
- **System Overview Dashboard**: Real-time metrics and system health monitoring
- **Queue Management Interface**: Complete CRUD operations for queues
- **Consumer Group Management**: Visual consumer group coordination
- **Event Store Explorer**: Advanced event querying interface
- **Message Browser**: Visual message inspection and debugging
- **Real-time Monitoring**: Live dashboards with WebSocket updates

peegeeq-service-manager - Service Discovery & Federation

- **Service Discovery**: HashiCorp Consul integration for multi-instance deployments
- **Load Balancing**: Intelligent request distribution across instances
- **Health Monitoring**: Distributed health checks and failover

- **Configuration Management**: Centralized configuration with Consul KV store

peegeeq-rest - HTTP API Layer

- **Database Setup API**: RESTful endpoints for database management
- **Queue Operations API**: HTTP interface for message production and consumption
- **Event Store API**: HTTP endpoints for event storage and querying
- **Consumer Group API**: REST endpoints for consumer group management
- **Management API**: Administrative endpoints for system monitoring
- **WebSocket Support**: Real-time message streaming
- **Server-Sent Events**: Efficient real-time data streaming

peegeeq-db - Database Management

- **PeeGeeQManager**: Central configuration and lifecycle management
- **Schema Migrations**: Automatic database setup and upgrades
- **Health Checks**: Database connectivity and performance monitoring
- **Metrics Collection**: Performance and operational metrics
- **Circuit Breakers**: Fault tolerance and resilience
- **Connection Pooling**: Optimized database connection management

Implementation Modules - Pluggable Patterns

- **peegeeq-native**: LISTEN/NOTIFY based real-time messaging with consumer groups
- **peegeeq-outbox**: Transaction-safe outbox pattern with parallel processing
- **peegeeq-bitemporal**: Event sourcing with bi-temporal queries and corrections

peegeeq-examples - Comprehensive Demonstrations

- **Self-contained Demo**: Complete demonstration with TestContainers
- **17 Core Examples**: Progressive examples covering all features
- **15 Test Examples**: Advanced integration and performance tests
- **Production Patterns**: Real-world usage scenarios and best practices

When to Choose PeeGeeQ

Perfect Fit Scenarios

✅ **You Should Use PeeGeeQ When:**

1. **You're Already Using PostgreSQL**
   - Leverage existing infrastructure and expertise
   - Reduce operational complexity
2. **Transactional Consistency is Critical**
   - Financial transactions
   - Order processing
   - Inventory management
   - Any scenario where message delivery must be tied to database changes
3. **You Want Operational Simplicity**
   - Single database to monitor and maintain
   - Unified backup and recovery strategy
   - Existing PostgreSQL HA solutions

4. **Your Team Knows SQL Better Than Message Brokers**
    - Faster development and debugging
    - Lower learning curve
    - Familiar troubleshooting tools
5. **Moderate to High Performance Requirements**
    - 1,000 to 50,000 messages per second
    - Sub-second latency requirements
    - Real-time processing needs

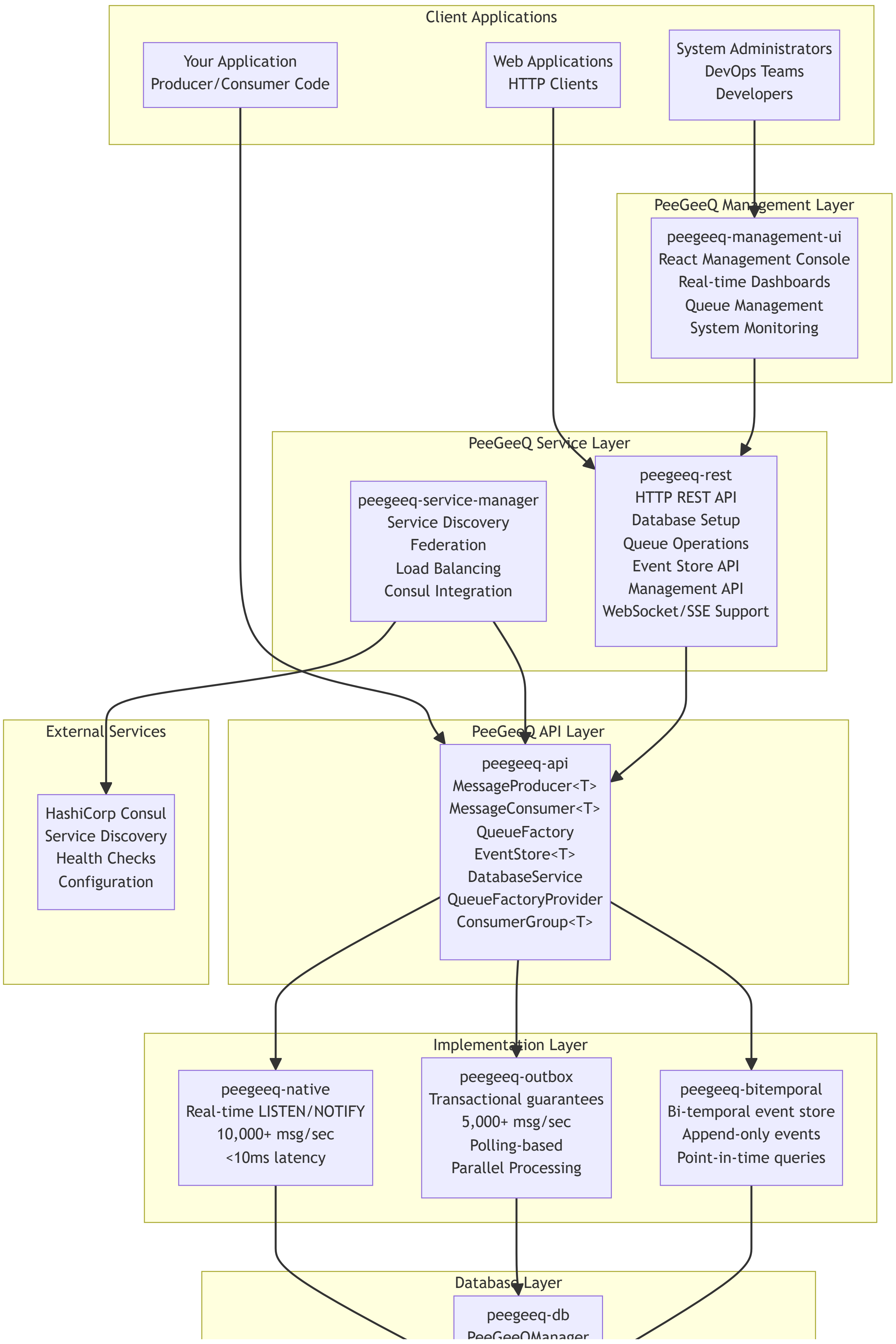
❌ **Consider Alternatives When:**

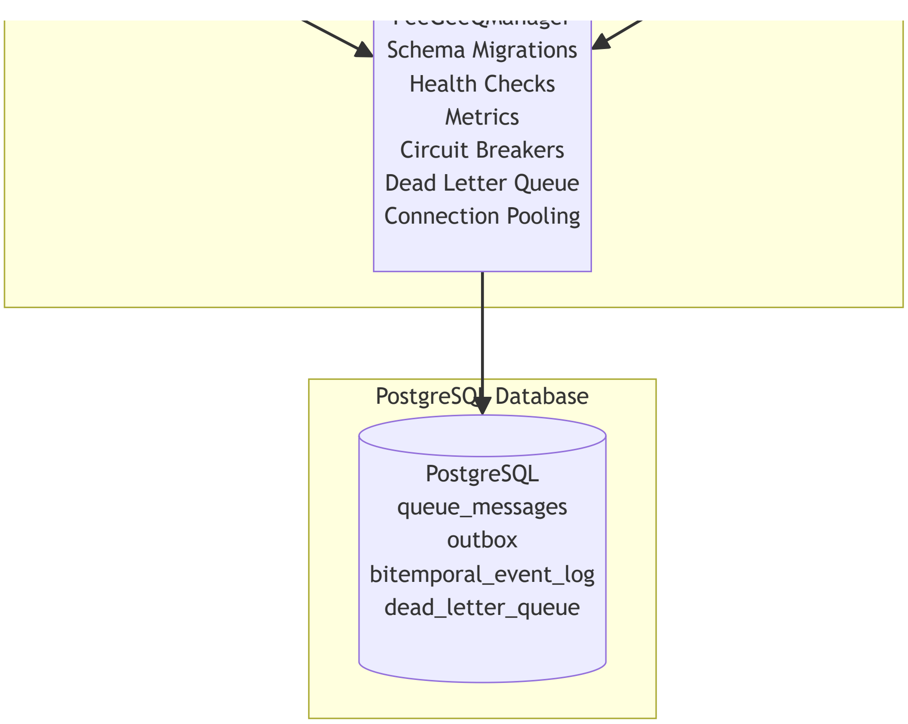
1. **Extreme Performance Requirements**
    - 100,000+ messages per second
    - Microsecond latency requirements
    - Specialized hardware optimizations needed
2. **Complex Message Routing**
    - Advanced routing rules and transformations
    - Multiple protocols (AMQP, MQTT, STOMP)
    - Complex message filtering and content-based routing
3. **Multi-Database Architecture**
    - Messages need to span multiple database systems
    - Polyglot persistence requirements
    - Cross-platform messaging
4. **Dedicated Messaging Team**
    - Team specialized in message broker operations
    - Complex messaging infrastructure already in place

Decision Matrix

Your Situation	Recommended Choice	Why?
E-commerce platform with PostgreSQL	**PeeGeeQ Outbox**	Transactional order processing
Real-time dashboard with PostgreSQL	**PeeGeeQ Native**	Low latency, existing infrastructure
Microservices with mixed databases	**Traditional Broker**	Cross-database messaging
Financial system requiring audit trails	**PeeGeeQ Bi-temporal**	Event sourcing with compliance
High-frequency trading system	**Traditional Broker**	Extreme performance requirements
Startup with PostgreSQL	**PeeGeeQ Native**	Simplicity and cost-effectiveness

Architecture Overview


Client Applications
Your Application
Producer/Consumer Code
Web Applications
HTTP Clients
System Administrators
DevOps Teams
Developers
PeeGeeQ Management Layer
peegeeq-management-ui
React Management Console
Real-time Dashboards
Queue Management
System Monitoring
PeeGeeQ Service Layer
peegeeq-service-manager
Service Discovery
Federation
Load Balancing
Consul Integration
peegeeq-rest
HTTP REST API
Database Setup
Queue Operations
Event Store API
Management API
WebSocket/SSE Support
External Services
HashiCorp Consul
Service Discovery
Health Checks
Configuration
PeeGeeQ API Layer
peegeeq-api
MessageProducer<T>
MessageConsumer<T>
QueueFactory
EventStore<T>
DatabaseService
QueueFactoryProvider
ConsumerGroup<T>
Implementation Layer
peegeeq-native
Real-time LISTEN/NOTIFY
10,000+ msg/sec
<10ms latency
peegeeq-outbox
Transactional guarantees
5,000+ msg/sec
Polling-based
Parallel Processing
peegeeq-bitemporal
Bi-temporal event store
Append-only events
Point-in-time queries
Database Layer
peegeeq-db
PeeGeeQManager



Part III: Getting Started (Progressive Learning)

Prerequisites & Environment Setup

System Requirements

- **Java 21+** (OpenJDK or Oracle JDK)
- **Maven 3.8+** for building
- **PostgreSQL 12+** for the database
- **Docker** (optional, for examples and testing)

Quick Environment Check

Let's verify your environment is ready:

```
# Check Java version
java -version
# Should show Java 21 or higher

# Check Maven version
mvn -version
# Should show Maven 3.8 or higher

# Check PostgreSQL (if installed locally)
psql --version
# Should show PostgreSQL 12 or higher

# Check Docker (optional)
docker --version
```

```
# Should show Docker version
```

30-Second Demo (Recommended First Step)

Before diving into code, see PeeGeeQ in action with our self-contained demo:

```
# Clone and run the demo
git clone <repository-url>
cd peegeeq

# Unix/Linux/macOS
./run-self-contained-demo.sh

# Windows
run-self-contained-demo.bat
```

What this demo shows:

1. **Starts a PostgreSQL container** (no local setup needed)
2. **Sets up the database schema** automatically
3. **Demonstrates all three patterns**: Native queue, outbox pattern, and bi-temporal event store
4. **Shows real-time message processing** with live output
5. **Cleans up automatically** when finished

Expected output:

```
 Starting PeeGeeQ Self-Contained Demo...
 Starting PostgreSQL container...
 Setting up database schema...
 Demonstrating Native Queue (real-time)...
 Demonstrating Outbox Pattern (transactional)...
 Demonstrating Bi-temporal Event Store...
 Demo completed successfully!
 Cleaning up containers...
```

✅ **Checkpoint**: If the demo runs successfully, your environment is ready!

Your First Message (Hello World)

Now let's build your first PeeGeeQ application step by step. We'll start with the absolute minimum code and gradually add features.

Step 1: Minimal Setup (2 minutes)

Create a new Java class with the absolute minimum code to get started:

```java
import dev.mars.peegeeq.db.PeeGeeQManager;
import dev.mars.peegeeq.api.MessageProducer;
import dev.mars.peegeeq.api.MessageConsumer;

public class HelloPeeGeeQ {
    public static void main(String[] args) throws Exception {
        // This is the absolute minimum code to get started
        PeeGeeQManager manager = new PeeGeeQManager();
        manager.start();

        System.out.println("✅ PeeGeeQ started successfully!");

        // Don't forget to clean up
        manager.close();
    }
}
```

Run it:

```
mvn compile exec:java -Dexec.mainClass="HelloPeeGeeQ"
```

Expected output:

```
✅ PeeGeeQ started successfully!
```

🎯 **Try This Now**: Run the code above. If it works, you've successfully connected to PostgreSQL!

Step 2: Send Your First Message (2 minutes)

Now let's send a message:

```
import dev.mars.peegeeq.db.PeeGeeQManager;
import dev.mars.peegeeq.api.MessageProducer;
import dev.mars.peegeeq.api.QueueFactory;
import dev.mars.peegeeq.api.QueueFactoryProvider;

public class HelloPeeGeeQ {
    public static void main(String[] args) throws Exception {
        // Setup PeeGeeQ
        PeeGeeQManager manager = new PeeGeeQManager();
        manager.start();

        // Create a producer
        QueueFactoryProvider provider = QueueFactoryProvider.getInstance();
        QueueFactory factory = provider.createFactory("native", manager.getDatabaseService());
        MessageProducer<String> producer = factory.createProducer("hello-queue", String.class);

        // Send your first message
        producer.send("Hello, PeeGeeQ!").join();
        System.out.println("📤 Message sent: Hello, PeeGeeQ!");

        // Cleanup
        producer.close();
        manager.close();
    }
}
```

Expected output:

```
📤 Message sent: Hello, PeeGeeQ!
```

✅ **Checkpoint**: You've successfully sent your first message!

Step 3: Receive Your First Message (3 minutes)

Now let's receive the message we sent:

```java
import dev.mars.peegeeq.db.PeeGeeQManager;
import dev.mars.peegeeq.api.*;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CountDownLatch;

public class HelloPeeGeeQ {
    public static void main(String[] args) throws Exception {
        // Setup PeeGeeQ
        PeeGeeQManager manager = new PeeGeeQManager();
        manager.start();

        // Create factory
        QueueFactoryProvider provider = QueueFactoryProvider.getInstance();
        QueueFactory factory = provider.createFactory("native", manager.getDatabaseService());

        // Create producer and consumer
        MessageProducer<String> producer = factory.createProducer("hello-queue", String.class);
        MessageConsumer<String> consumer = factory.createConsumer("hello-queue", String.class);

        // Setup to wait for message
        CountDownLatch messageReceived = new CountDownLatch(1);

        // Start listening for messages
        consumer.subscribe(message -> {
            System.out.println("📪 Received: " + message.getPayload());
            messageReceived.countDown(); // Signal that we got the message
            return CompletableFuture.completedFuture(null);
        });

        // Give consumer a moment to start
        Thread.sleep(1000);

        // Send the message
        producer.send("Hello, PeeGeeQ!").join();
        System.out.println("📤 Message sent: Hello, PeeGeeQ!");

        // Wait for message to be received
        messageReceived.await();
        System.out.println("✅ Message processing complete!");

        // Cleanup
        consumer.close();
        producer.close();
        manager.close();
    }
}
```

Expected output:

```
📤 Message sent: Hello, PeeGeeQ!
📫 Received: Hello, PeeGeeQ!
✅ Message processing complete!
```

✅ **Checkpoint**: You've successfully sent and received your first message!

🎯 **Try This Now**: Modify the message content and run it again:

```
// Try different messages
producer.send("Your custom message here!").join();
producer.send("Message sent at: " + java.time.Instant.now()).join();

// Send multiple messages
for (int i = 1; i <= 5; i++) {
    producer.send("Message #" + i).join();
}
```

Understanding the Code

Let's break down what just happened in your first PeeGeeQ application:

1. PeeGeeQManager - The Central Controller

```
PeeGeeQManager manager = new PeeGeeQManager();
manager.start();
```

What it does:

- **Connects to PostgreSQL** using default configuration
- **Creates database schema** if it doesn't exist
- **Starts health checks** and monitoring
- **Initializes connection pools**

Configuration sources (in order of precedence):

1. System properties (`-Dpeegeeq.database.host=localhost`)
2. Environment variables (`PEEGEEQ_DATABASE_HOST=localhost`)
3. `peegeeq.properties` file in classpath
4. Default values (localhost:5432, database: peegeeq)

2. QueueFactoryProvider - The Factory Registry

```
QueueFactoryProvider provider = QueueFactoryProvider.getInstance();
QueueFactory factory = provider.createFactory("native", manager.getDatabaseService());
```

What it does:

- **Manages different queue implementations** (native, outbox, bitemporal)

- **Provides a unified interface** for creating producers and consumers
- **Handles dependency injection** automatically

Available factory types:

- `"native"` - Real-time LISTEN/NOTIFY based queues
- `"outbox"` - Transactional outbox pattern queues
- `"bitemporal"` - Event sourcing with temporal queries

3. MessageProducer - Type-Safe Message Sending

```
MessageProducer<String> producer = factory.createProducer("hello-queue", String.class);
producer.send("Hello, PeeGeeQ!").join();
```

What it does:

- **Type-safe message sending** - compile-time type checking
- **Automatic serialization** - converts objects to JSON
- **Asynchronous by default** - returns CompletableFuture
- **Queue creation** - automatically creates queue if it doesn't exist

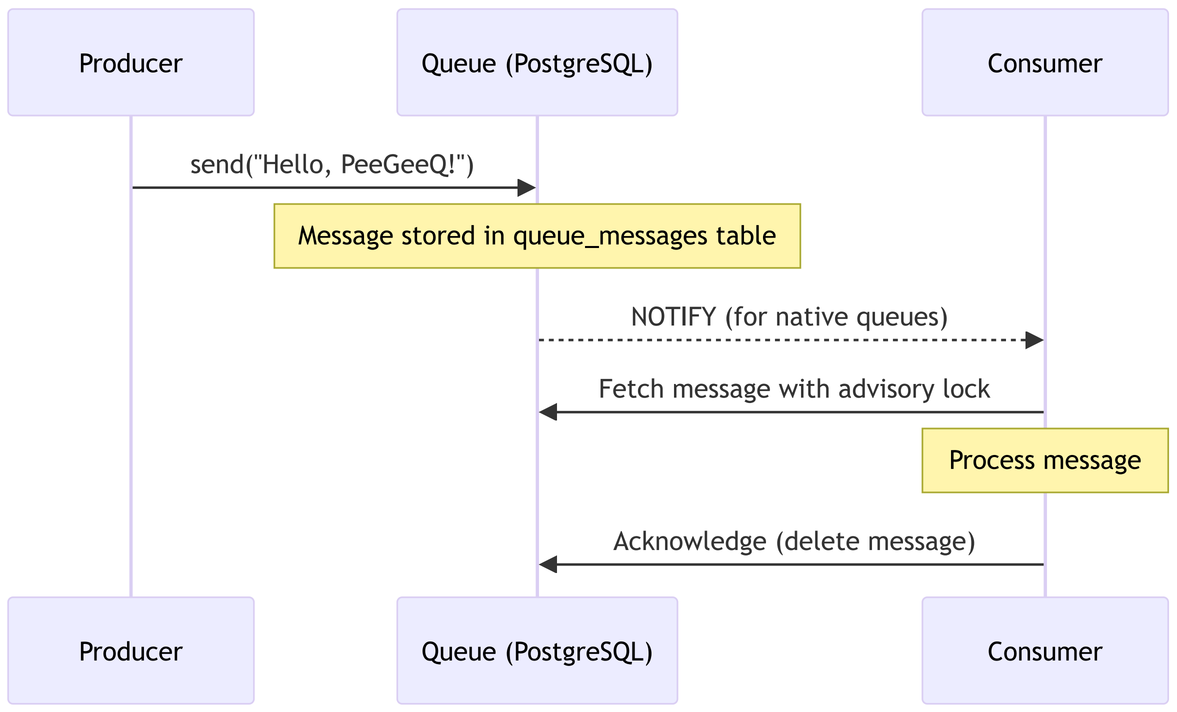
4. MessageConsumer - Type-Safe Message Receiving

```
MessageConsumer<String> consumer = factory.createConsumer("hello-queue", String.class);
consumer.subscribe(message -> {
    System.out.println("Received: " + message.getPayload());
    return CompletableFuture.completedFuture(null);
});
```

What it does:

- **Type-safe message receiving** - automatic deserialization
- **Functional interface** - clean lambda-based processing
- **Asynchronous processing** - non-blocking message handling
- **Automatic acknowledgment** - messages are acknowledged when CompletableFuture completes

5. Message Lifecycle


Producer
Queue (PostgreSQL)
Consumer
send("Hello, PeeGeeQ!")
Message stored in queue_messages table
NOTIFY (for native queues)
Fetch message with advisory lock
Process message
Acknowledge (delete message)
Producer
Queue (PostgreSQL)
Consumer

Adding Error Handling

Real applications need proper error handling. Let's enhance our example:

```
import dev.mars.peegeeq.db.PeeGeeQManager;
import dev.mars.peegeeq.api.*;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CountDownLatch;

public class HelloPeeGeeQWithErrorHandling {
    public static void main(String[] args) {
        PeeGeeQManager manager = null;
        MessageProducer<String> producer = null;
        MessageConsumer<String> consumer = null;

        try {
            // Setup with error handling
            manager = new PeeGeeQManager();
            manager.start();
            System.out.println("✅ PeeGeeQ started successfully!");

            // Create factory and components
            QueueFactoryProvider provider = QueueFactoryProvider.getInstance();
            QueueFactory factory = provider.createFactory("native", manager.getDatabaseService());

            producer = factory.createProducer("hello-queue", String.class);
            consumer = factory.createConsumer("hello-queue", String.class);

            // Setup message processing with error handling
            CountDownLatch messageReceived = new CountDownLatch(1);

            consumer.subscribe(message -> {
                try {
                    System.out.println("📫 Received: " + message.getPayload());
```

```java
                    // Simulate some processing that might fail
                    if (message.getPayload().contains("error")) {
                        throw new RuntimeException("Simulated processing error");
                    }

                    messageReceived.countDown();
                    return CompletableFuture.completedFuture(null);

                } catch (Exception e) {
                    System.err.println("❌ Error processing message: " + e.getMessage());
                    messageReceived.countDown();
                    // Return failed future to trigger retry
                    return CompletableFuture.failedFuture(e);
                }
            });

            // Give consumer time to start
            Thread.sleep(1000);

            // Send messages with error handling
            try {
                producer.send("Hello, PeeGeeQ!").join();
                System.out.println("📤 Message sent successfully!");
            } catch (Exception e) {
                System.err.println("❌ Failed to send message: " + e.getMessage());
            }

            // Wait for processing
            messageReceived.await();
            System.out.println("✅ Processing complete!");

        } catch (Exception e) {
            System.err.println("❌ Application error: " + e.getMessage());
            e.printStackTrace();
        } finally {
            // Cleanup in finally block
            if (consumer != null) {
                try { consumer.close(); } catch (Exception e) { /* ignore */ }
            }
            if (producer != null) {
                try { producer.close(); } catch (Exception e) { /* ignore */ }
            }
            if (manager != null) {
                try { manager.close(); } catch (Exception e) { /* ignore */ }
            }
        }
    }
}
```

🎯 **Try This Now**:

1. Run the code above with normal messages
2. Try sending a message containing "error" to see error handling in action
3. Observe how the application handles failures gracefully

Adding Configuration

Hard-coded configuration isn't suitable for real applications. Let's add proper configuration:

Create `peegeeq.properties`

```
# Database connection
peegeeq.database.host=localhost
peegeeq.database.port=5432
peegeeq.database.name=peegeeq
peegeeq.database.username=peegeeq_user
peegeeq.database.password=your_password

# Connection pool
peegeeq.database.pool.maxSize=10
peegeeq.database.pool.minSize=2

# Queue settings
peegeeq.queue.visibilityTimeoutSeconds=30
peegeeq.queue.maxRetries=3

# Health checks
peegeeq.health.enabled=true
peegeeq.health.intervalSeconds=30
```

Updated Code with Configuration

```
import dev.mars.peegeeq.db.PeeGeeQManager;
import dev.mars.peegeeq.db.PeeGeeQConfiguration;
import dev.mars.peegeeq.api.*;
import java.util.concurrent.CompletableFuture;

public class HelloPeeGeeQConfigured {
    public static void main(String[] args) {
        try {
            // Load configuration from properties file
            PeeGeeQConfiguration config = PeeGeeQConfiguration.fromProperties("peegeeq.properties");

            // Or build configuration programmatically
            // PeeGeeQConfiguration config = PeeGeeQConfiguration.builder()
            //     .host("localhost")
            //     .port(5432)
            //     .database("peegeeq")
            //     .username("peegeeq_user")
            //     .password("your_password")
            //     .build();

            try (PeeGeeQManager manager = new PeeGeeQManager(config)) {
                manager.start();
                System.out.println("✅ PeeGeeQ started with custom configuration!");

                // Rest of your application code...
                QueueFactoryProvider provider = QueueFactoryProvider.getInstance();
                QueueFactory factory = provider.createFactory("native", manager.getDatabaseService());

                try (MessageProducer<String> producer = factory.createProducer("configured-queue", String.class);
                     MessageConsumer<String> consumer = factory.createConsumer("configured-queue", String.class)) {

                    // Your messaging code here...
                    producer.send("Hello from configured PeeGeeQ!").join();
                    System.out.println("📤 Message sent with custom configuration!");
                }
            }

        } catch (Exception e) {
            System.err.println("❌ Configuration error: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

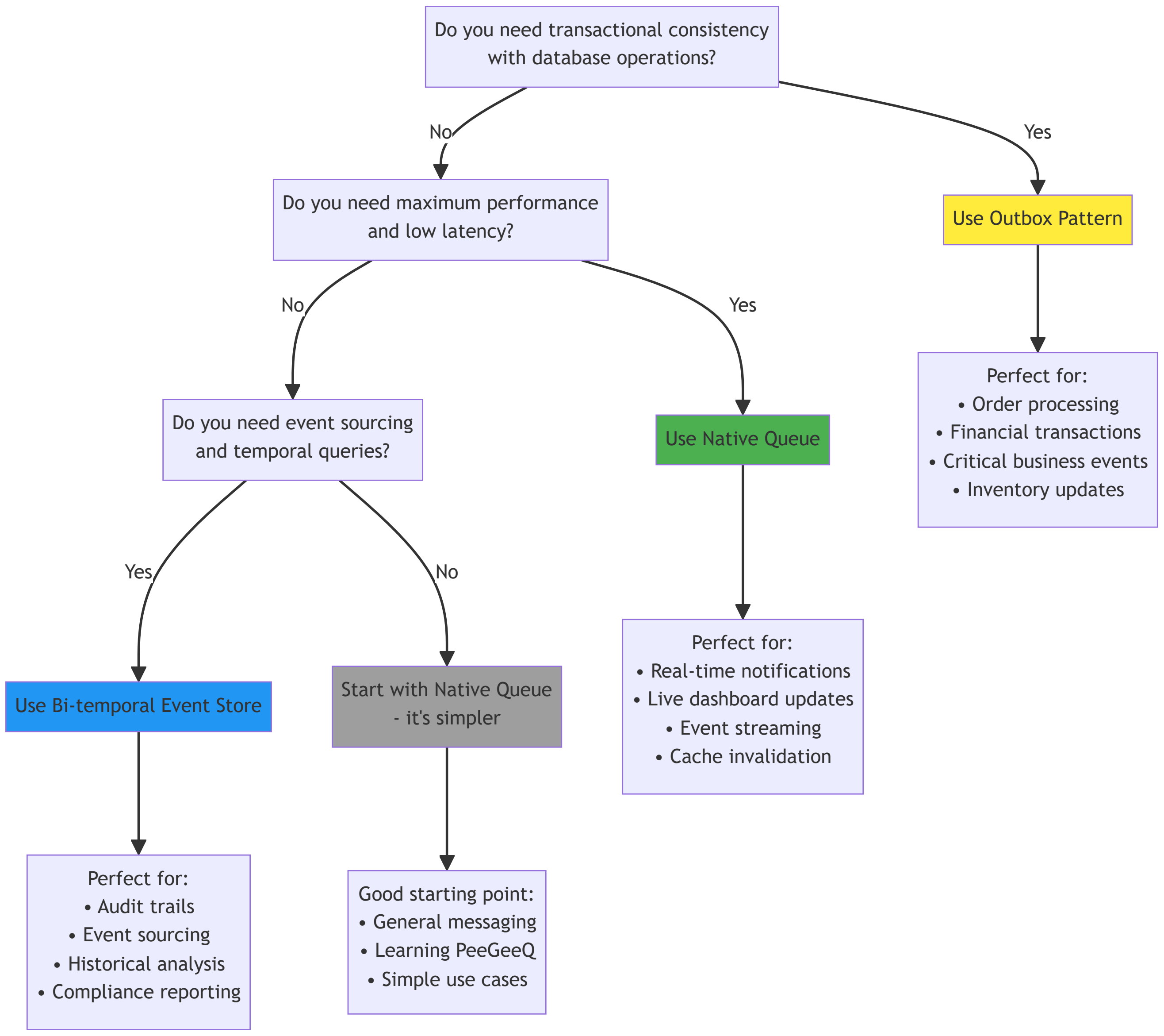
✅ **Checkpoint**: You now have a properly configured PeeGeeQ application with error handling!

Part IV: Core Concepts (Detailed)

Now that you've successfully sent and received your first messages, let's dive deep into PeeGeeQ's three powerful messaging patterns. Understanding these patterns will help you choose the right approach for your specific use case.

Choosing the Right Pattern

Before we explore each pattern in detail, let's start with a decision tree to help you choose:



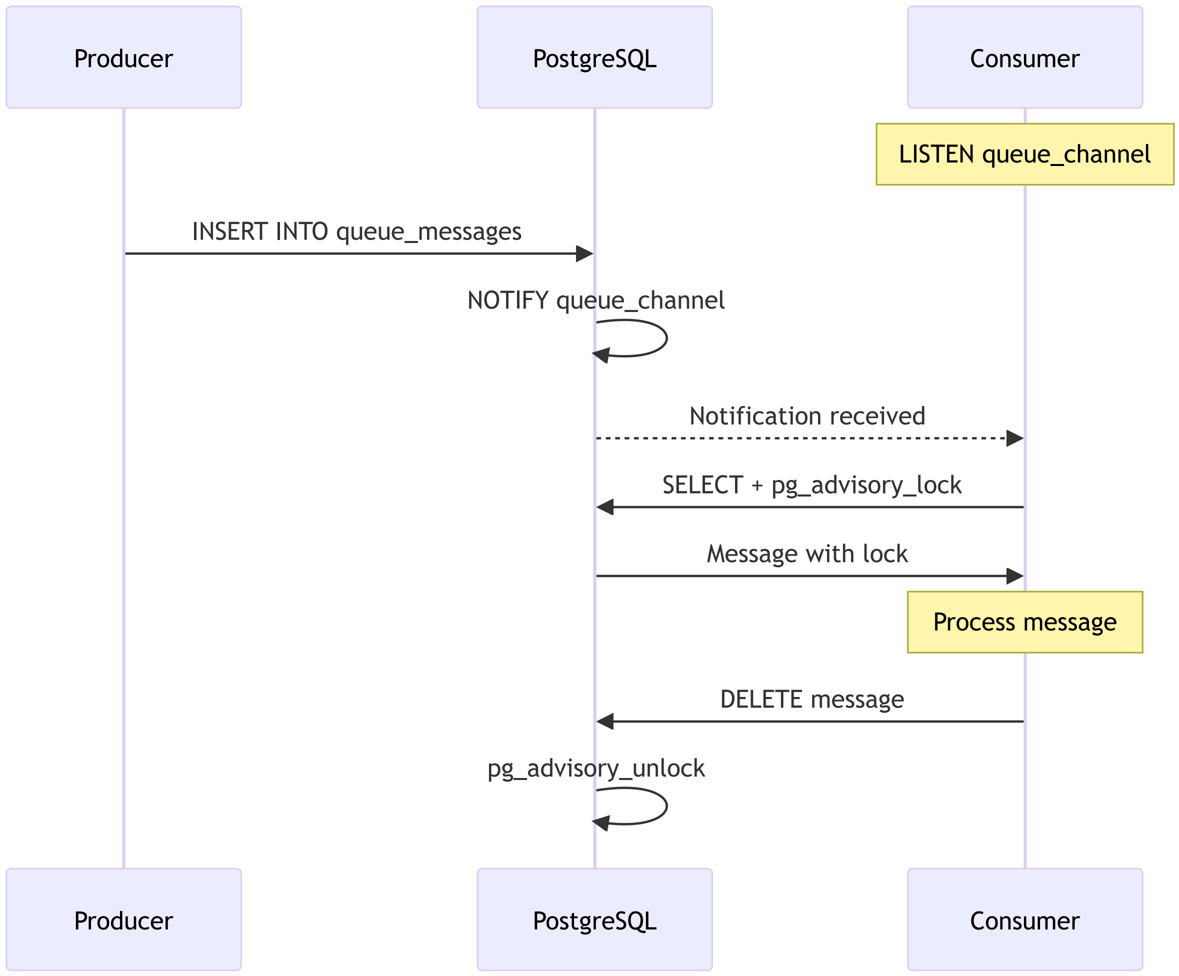
Quick Decision Guide

Your Requirement	Recommended Pattern	Why?
"I need to ensure my order and payment are processed together"	**Outbox Pattern**	Transactional consistency
"I need to notify users instantly when something happens"	**Native Queue**	Real-time, low latency
"I need to track all changes and query historical data"	**Bi-temporal Store**	Event sourcing capabilities
"I'm not sure, just want to get started"	**Native Queue**	Simplest to understand and use
"I need both real-time and transactional messaging"	**Use Both**	PeeGeeQ supports multiple patterns

Native Queue Pattern (Deep Dive)

The Native Queue pattern leverages PostgreSQL's LISTEN/NOTIFY mechanism for real-time message delivery with minimal latency.

How It Works



Key Characteristics

Performance

- **Throughput**: 10,000+ messages/second
- **Latency**: <10ms from send to receive
- **Scalability**: Horizontal scaling with multiple consumers

Delivery Guarantees

- **At-least-once delivery**: Messages may be delivered more than once in failure scenarios
- **Ordering**: FIFO ordering within a single queue
- **Durability**: Messages survive database restarts

Technical Implementation

- **LISTEN/NOTIFY**: Real-time notifications without polling
- **Advisory Locks**: Prevent duplicate processing across consumers
- **Automatic Cleanup**: Processed messages are automatically removed

Message Processing Guarantees and Idempotency

Understanding PeeGeeQ's Delivery Guarantees

PeeGeeQ provides **at-least-once delivery** guarantees, which means messages will be delivered one or more times, but never zero times. This is a fundamental design choice that prioritizes **reliability over deduplication**.

Why At-Least-Once Instead of Exactly-Once?

Exactly-once processing is extremely difficult to achieve in distributed systems without significant performance trade-offs. Even systems that claim "exactly-once" typically mean "effectively-once" and still require idempotent message handlers.

PeeGeeQ's approach is **honest and practical**:

- ✅ **Guarantees no message loss** (critical for business operations)
- ✅ **Provides predictable behavior** (developers know what to expect)
- ✅ **Maintains high performance** (no complex coordination overhead)
- ✅ **Follows industry best practices** (idempotency is standard in distributed systems)

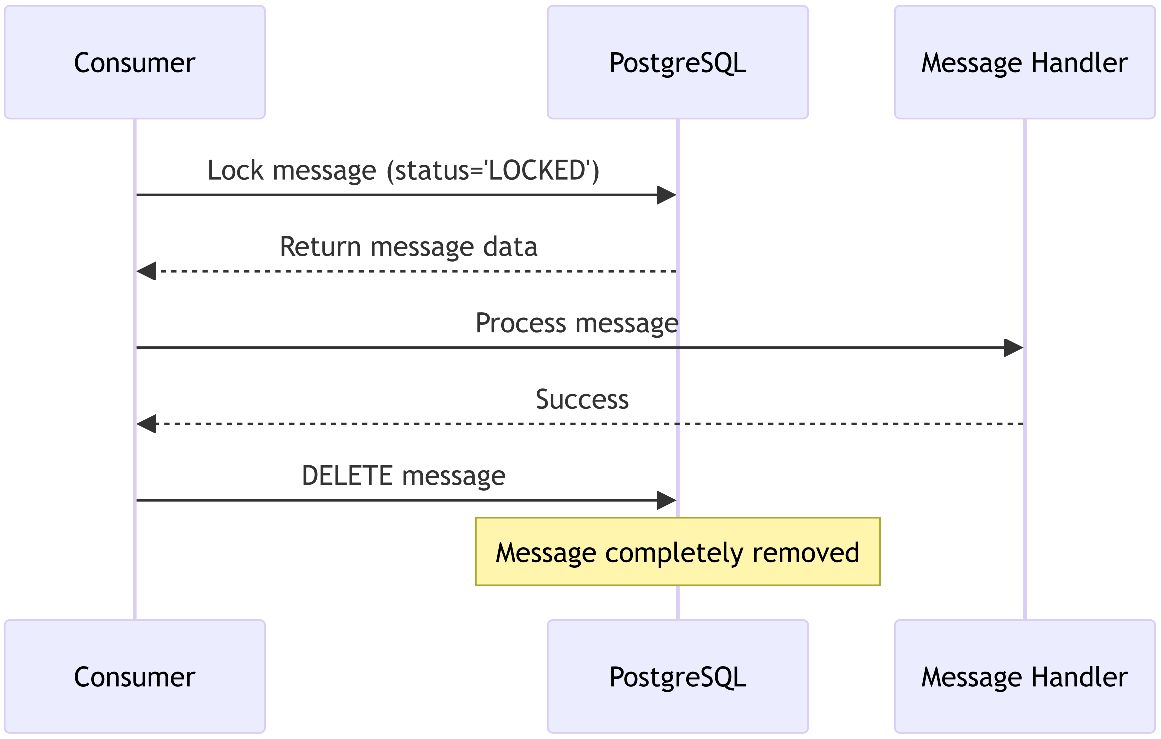
Message Processing Lifecycle (Technical Deep Dive)

1. Message Locking and Processing

When a consumer processes a message, PeeGeeQ follows this sequence:

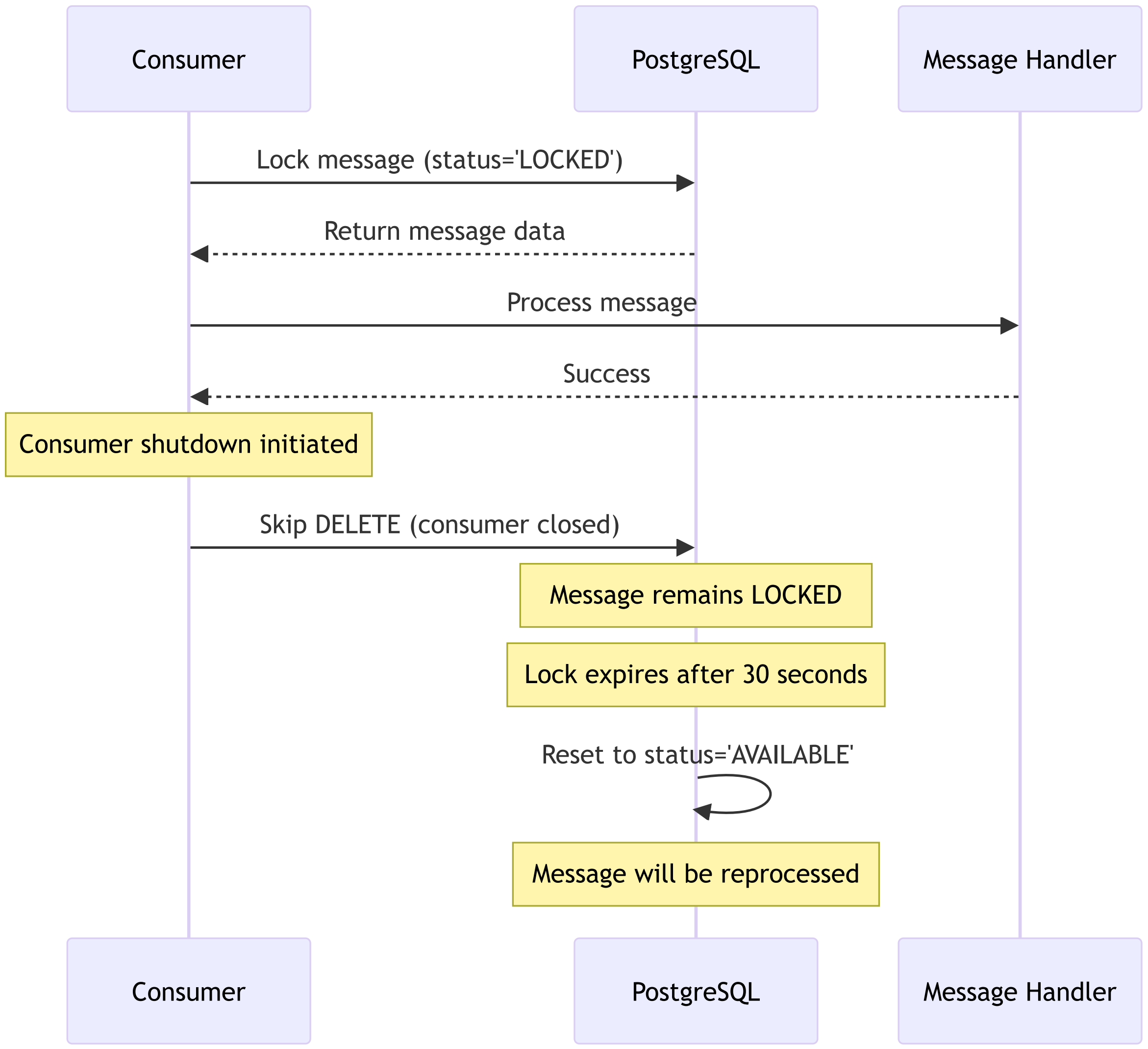
```sql
-- Step 1: Lock the message for processing
UPDATE queue_messages
SET status = 'LOCKED', lock_until = NOW() + INTERVAL '30 seconds'
WHERE id IN (
    SELECT id FROM queue_messages
    WHERE topic = 'my-topic' AND status = 'AVAILABLE'
    ORDER BY priority DESC, created_at ASC
    LIMIT 1
    FOR UPDATE SKIP LOCKED
);

-- Step 2: Process message in application code
-- (Your message handler runs here)

-- Step 3: Delete message after successful processing
DELETE FROM queue_messages WHERE id = $messageId;
```

2. Normal Processing Flow



3. Shutdown During Processing

Here's where the **at-least-once guarantee** becomes important:


Consumer
PostgreSQL
Message Handler
Lock message (status='LOCKED')
Return message data
Process message
Success
Consumer shutdown initiated
Skip DELETE (consumer closed)
Message remains LOCKED
Lock expires after 30 seconds
Reset to status='AVAILABLE'
Message will be reprocessed
Consumer
PostgreSQL
Message Handler

What happens:

1. Message is **successfully processed** by your business logic ✅
2. Consumer shuts down before deleting the message ❌
3. Message remains in `LOCKED` status with expiration timestamp
4. After 30 seconds, lock expires and message becomes `AVAILABLE` again
5. **Message will be reprocessed** when service restarts

Lock Expiration and Recovery

PeeGeeQ includes automatic **lock expiration** to handle various failure scenarios:

```sql
-- Automatic lock recovery (runs periodically)
UPDATE queue_messages
SET status = 'AVAILABLE', lock_until = NULL
WHERE topic = $topic
  AND status = 'LOCKED'
  AND lock_until < NOW();
```

This handles:

- Consumer crashes during processing
- Network failures during acknowledgment
- Graceful shutdowns with in-flight messages
- Database connection timeouts
- Application deadlocks or hangs

Scenarios That Cause Duplicate Processing

1. Graceful Shutdown (Most Common)

```
// Your message handler completes successfully
consumer.subscribe(message -> {
    processOrder(message.getPayload()); // ✅ Completes successfully
    return CompletableFuture.completedFuture(null);
});

// But consumer.close() is called before message deletion
consumer.close(); // Triggers shutdown, skips deletion
```

Result: Message processed once during shutdown, reprocessed after restart.

2. Consumer Crash

```
consumer.subscribe(message -> {
    processOrder(message.getPayload()); // ✅ Completes successfully
    // JVM crashes here before deletion
    return CompletableFuture.completedFuture(null);
});
```

Result: Message processed once before crash, reprocessed after restart.

3. Database Connection Issues

```
consumer.subscribe(message -> {
    processOrder(message.getPayload()); // ✅ Completes successfully
    // Database connection fails during deletion
    return CompletableFuture.completedFuture(null);
});
```

Result: Message processed once, deletion fails, reprocessed later.

4. Network Partitions

```
consumer.subscribe(message -> {
    processOrder(message.getPayload()); // ✅ Completes successfully
    // Network partition prevents deletion acknowledgment
    return CompletableFuture.completedFuture(null);
});
```

Result: Message processed once, network issue prevents cleanup, reprocessed later.

The Idempotency Requirement

Bottom Line: Your message handlers **must be idempotent** to work correctly with PeeGeeQ (and any distributed messaging system).

What is Idempotency?

An operation is **idempotent** if performing it multiple times has the same effect as performing it once.

Examples:

- ✅ `SET user.email = 'new@email.com'` (idempotent)
- ❌ `UPDATE user.login_count = login_count + 1` (not idempotent)
- ✅ `INSERT INTO orders (id, ...) ON CONFLICT DO NOTHING` (idempotent)
- ❌ `INSERT INTO orders (id, ...)` (not idempotent)

Implementing Idempotent Message Handlers

Strategy 1: Database Constraints

```java
@Component
public class OrderProcessor {

    public void processOrderCreated(OrderCreatedEvent event) {
        try {
            // Use database constraints to prevent duplicates
            jdbcTemplate.update(
                "INSERT INTO processed_orders (order_id, processed_at) VALUES (?, ?)",
                event.getOrderId(), Instant.now()
            );

            // Process the order
            fulfillmentService.processOrder(event);

        } catch (DuplicateKeyException e) {
            // Order already processed, skip silently
            log.debug("Order {} already processed, skipping", event.getOrderId());
        }
    }
}
```

Strategy 2: Explicit Deduplication

```java
@Component
public class PaymentProcessor {

    public void processPayment(PaymentEvent event) {
        String messageId = event.getMessageId();

        // Check if already processed
        if (processedMessageRepository.existsById(messageId)) {
            log.debug("Payment {} already processed, skipping", messageId);
            return;
        }

        // Process payment
        paymentService.processPayment(event);

        // Mark as processed
        processedMessageRepository.save(new ProcessedMessage(messageId, Instant.now()));
    }
}
```

Strategy 3: Natural Idempotency

```
@Component
public class UserProfileUpdater {

    public void updateUserProfile(UserProfileUpdatedEvent event) {
        // Naturally idempotent - setting values to specific states
        User user = userRepository.findById(event.getUserId());
        user.setEmail(event.getNewEmail());
        user.setName(event.getNewName());
        user.setUpdatedAt(event.getTimestamp());

        userRepository.save(user); // Same result regardless of repetition
    }
}
```

Strategy 4: Conditional Processing

```
@Component
public class InventoryManager {

    public void processInventoryUpdate(InventoryEvent event) {
        Product product = productRepository.findById(event.getProductId());

        // Only process if event is newer than last processed
        if (event.getTimestamp().isAfter(product.getLastInventoryUpdate())) {
            product.setQuantity(event.getNewQuantity());
            product.setLastInventoryUpdate(event.getTimestamp());
            productRepository.save(product);
        } else {
            log.debug("Inventory event {} is older than last update, skipping",
                event.getEventId());
        }
    }
}
```

Production Considerations

Monitoring Duplicate Processing

Set up monitoring to track duplicate processing rates:

```
@Component
public class MessageProcessingMetrics {
    private final MeterRegistry meterRegistry;
    private final Counter duplicateCounter;
    private final Counter processedCounter;

    public MessageProcessingMetrics(MeterRegistry meterRegistry) {
        this.meterRegistry = meterRegistry;
        this.duplicateCounter = Counter.builder("messages.duplicates")
            .description("Number of duplicate messages processed")
            .register(meterRegistry);
        this.processedCounter = Counter.builder("messages.processed")
```

```
            .description("Total messages processed")
            .register(meterRegistry);
    }

    public void recordDuplicate(String topic) {
        duplicateCounter.increment(Tags.of("topic", topic));
    }

    public void recordProcessed(String topic) {
        processedCounter.increment(Tags.of("topic", topic));
    }
}
```

Alerting on High Duplicate Rates

```
# Prometheus alerting rule
- alert: HighMessageDuplicateRate
  expr: |
    (
      rate(messages_duplicates_total[5m]) /
      rate(messages_processed_total[5m])
    ) > 0.1
  for: 2m
  labels:
    severity: warning
  annotations:
    summary: "High message duplicate rate detected"
    description: "Duplicate message rate is {{ $value | humanizePercentage }} for topic {{ $labels.topic }}"
```

Graceful Shutdown Best Practices

```
@Component
public class GracefulShutdownHandler {

    @PreDestroy
    public void shutdown() {
        log.info("Initiating graceful shutdown...");

        // 1. Stop accepting new messages
        consumer.unsubscribe();

        // 2. Wait for in-flight messages to complete
        try {
            boolean completed = consumer.awaitTermination(30, TimeUnit.SECONDS);
            if (!completed) {
                log.warn("Some messages may not have completed processing");
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            log.warn("Shutdown interrupted");
        }

        // 3. Close consumer
        consumer.close();

        log.info("Graceful shutdown completed");
    }
}
```

Database Maintenance

Regular cleanup of processed message tracking:

```sql
-- Clean up old processed message records (run daily)
DELETE FROM processed_messages
WHERE processed_at < NOW() - INTERVAL '7 days';

-- Monitor lock expiration frequency
SELECT
    topic,
    COUNT(*) as expired_locks,
    AVG(EXTRACT(EPOCH FROM (NOW() - lock_until))) as avg_expiry_delay
FROM queue_messages
WHERE status = 'AVAILABLE'
  AND lock_until IS NOT NULL
  AND lock_until < NOW()
GROUP BY topic;
```

Design Philosophy: Why At-Least-Once is Better

The CAP Theorem Reality

In distributed systems, you must choose between:

- **Consistency**: All nodes see the same data simultaneously
- **Availability**: System remains operational during failures
- **Partition Tolerance**: System continues despite network failures

PeeGeeQ chooses **Availability + Partition Tolerance**, which means:

- ✅ Messages are never lost (availability)
- ✅ System works during network issues (partition tolerance)
- ⚠️ Occasional duplicates may occur (eventual consistency)

Industry Standard Approach

Major messaging systems and their guarantees:

- **Apache Kafka**: At-least-once (exactly-once requires special configuration)
- **Amazon SQS**: At-least-once (exactly-once available with FIFO queues)
- **RabbitMQ**: At-least-once (exactly-once requires publisher confirms + consumer acks)
- **Google Pub/Sub**: At-least-once (exactly-once requires idempotent processing)
- **Azure Service Bus**: At-least-once (exactly-once requires sessions)

PeeGeeQ is in good company - at-least-once with idempotency is the industry standard.

Performance Benefits

At-least-once delivery enables:

- **Higher throughput** (no complex coordination)
- **Lower latency** (no multi-phase commits)
- **Better availability** (no blocking on acknowledgments)
- **Simpler operations** (fewer failure modes)

Summary: Embrace Idempotency

Key Takeaways:

1. **At-least-once is intentional** - PeeGeeQ prioritizes reliability over deduplication
2. **Duplicates are rare** - They mainly occur during failures and shutdowns
3. **Idempotency is required** - This is true for any distributed messaging system
4. **Multiple strategies available** - Choose the approach that fits your use case
5. **Monitor and alert** - Track duplicate rates to ensure system health
6. **Industry standard** - All major messaging systems work this way

Remember: The goal isn't to eliminate duplicates entirely (impossible without major trade-offs), but to handle them gracefully through idempotent design.

When to Use Native Queue

✅ **Perfect for:**

- **Real-time notifications** (user alerts, system notifications)
- **Live dashboard updates** (metrics, status changes)
- **Event streaming** (activity feeds, audit logs)
- **Cache invalidation** (distributed cache updates)
- **System monitoring** (alerts, health checks)

❌ **Not ideal for:**

- **Financial transactions** (use Outbox pattern instead)
- **Critical business events** that must be tied to database changes
- **Scenarios requiring exactly-once delivery**

Native Queue Example

```java
public class NativeQueueExample {
    public static void main(String[] args) throws Exception {
        try (PeeGeeQManager manager = new PeeGeeQManager()) {
            manager.start();

            // Create native queue factory
            QueueFactoryProvider provider = QueueFactoryProvider.getInstance();
            QueueFactory factory = provider.createFactory("native", manager.getDatabaseService());

            // Real-time notification system
            try (MessageProducer<NotificationEvent> producer =
                     factory.createProducer("notifications", NotificationEvent.class);
                 MessageConsumer<NotificationEvent> consumer =
                     factory.createConsumer("notifications", NotificationEvent.class)) {

                // Start consuming notifications
                consumer.subscribe(message -> {
                    NotificationEvent event = message.getPayload();
                    System.out.printf("🔔 Notification: %s for user %s%n",
                        event.getMessage(), event.getUserId());

                    // Send to user's device, email, etc.
                    sendToUser(event);
                    return CompletableFuture.completedFuture(null);
                });

                // Simulate real-time events
                producer.send(new NotificationEvent("user123", "Your order has shipped!"));
                producer.send(new NotificationEvent("user456", "New message received"));
                producer.send(new NotificationEvent("user789", "Payment processed successfully"));

                Thread.sleep(2000); // Let messages process
            }
        }
    }

    private static void sendToUser(NotificationEvent event) {
        // Implementation for sending notification to user
        // (push notification, email, SMS, etc.)
    }
}

class NotificationEvent {
    private String userId;
    private String message;
    private Instant timestamp;

    public NotificationEvent(String userId, String message) {
        this.userId = userId;
        this.message = message;
        this.timestamp = Instant.now();
    }

    // Getters and setters...
}
```

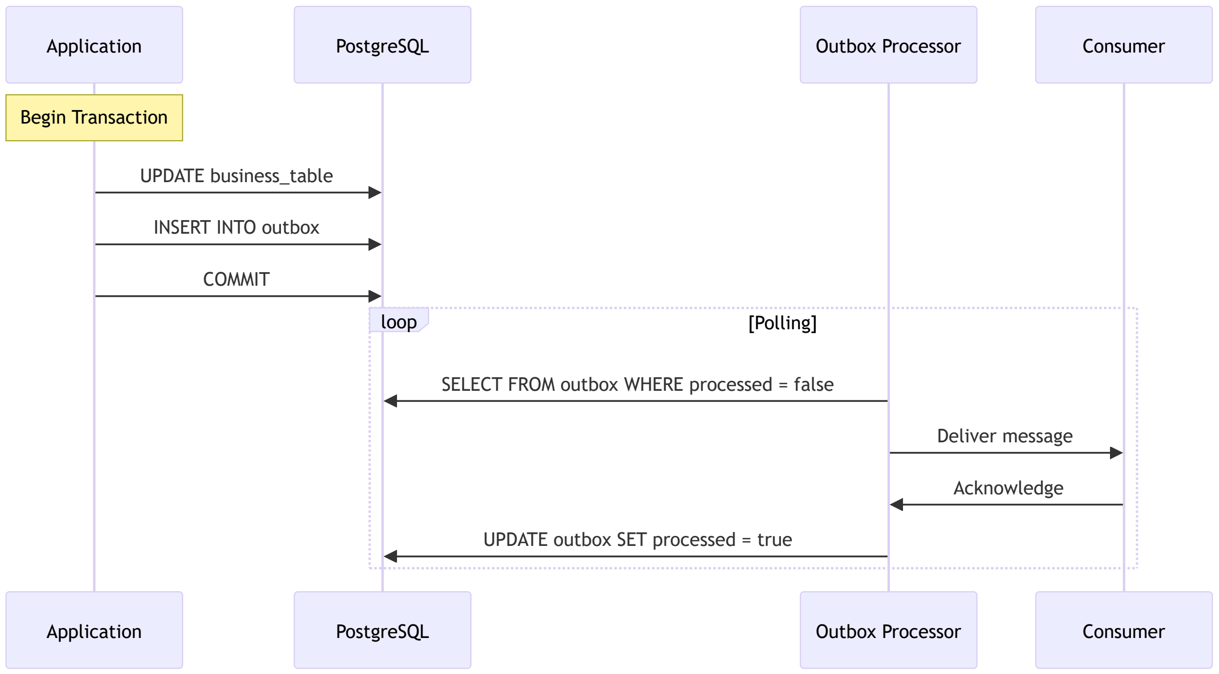
🎯 **Try This Now**:

1. Create the NotificationEvent class
2. Run the example and observe real-time message processing
3. Try sending messages from multiple threads to see concurrent processing

Outbox Pattern (Deep Dive)

The Outbox pattern ensures transactional consistency between your business data and message delivery by storing messages in the same database transaction as your business operations.

How It Works



Key Characteristics

Performance

- **Throughput**: 5,000+ messages/second
- **Latency**: ~100ms (due to polling interval)
- **Scalability**: Multiple processors can handle different message types

Delivery Guarantees

- **Exactly-once delivery**: Messages are delivered exactly once
- **Transactional consistency**: Messages are only sent if business transaction succeeds
- **Ordering**: Strict ordering within message type
- **Durability**: Messages survive all types of failures

Technical Implementation

- **Database table**: Messages stored in `outbox` table
- **Polling mechanism**: Background processor polls for new messages
- **Retry logic**: Automatic retry with exponential backoff
- **Dead letter queue**: Failed messages moved to DLQ after max retries
- **Stuck message recovery**: Automatic recovery of messages stuck in PROCESSING state

When to Use Outbox Pattern

✅ Perfect for:

- **Order processing** (order creation + inventory update + notification)
- **Financial transactions** (payment processing + account updates + receipts)
- **User registration** (create user + send welcome email + setup defaults)
- **Inventory management** (stock updates + reorder notifications + reporting)
- **Critical business events** that must be consistent with data changes

❌ **Not ideal for:**

- **High-frequency events** where slight latency is acceptable
- **Non-critical notifications** that don't need transactional guarantees
- **Real-time streaming** scenarios

Outbox Pattern Example

```java
public class OutboxPatternExample {
    public static void main(String[] args) throws Exception {
        try (PeeGeeQManager manager = new PeeGeeQManager()) {
            manager.start();

            // Create outbox factory for transactional guarantees
            QueueFactoryProvider provider = QueueFactoryProvider.getInstance();
            QueueFactory factory = provider.createFactory("outbox", manager.getDatabaseService());

            try (MessageProducer<OrderEvent> orderProducer =
                     factory.createProducer("order-events", OrderEvent.class);
                 MessageProducer<EmailEvent> emailProducer =
                     factory.createProducer("email-events", EmailEvent.class);
                 MessageConsumer<OrderEvent> orderConsumer =
                     factory.createConsumer("order-events", OrderEvent.class);
                 MessageConsumer<EmailEvent> emailConsumer =
                     factory.createConsumer("email-events", EmailEvent.class)) {

                // Setup consumers
                orderConsumer.subscribe(message -> {
                    OrderEvent event = message.getPayload();
                    System.out.printf("📦 Processing order: %s for $%.2f%n",
                        event.getOrderId(), event.getAmount());

                    // Process order (update inventory, etc.)
                    processOrder(event);

                    return CompletableFuture.completedFuture(null);
                });

                emailConsumer.subscribe(message -> {
                    EmailEvent event = message.getPayload();
                    System.out.printf("📧 Sending email: %s to %s%n",
                        event.getSubject(), event.getToEmail());

                    // Send email
                    sendEmail(event);

                    return CompletableFuture.completedFuture(null);
                });

                // Simulate order processing with transactional consistency
                processOrderTransactionally(manager, orderProducer, emailProducer);

                Thread.sleep(3000); // Let messages process
            }
        }
    }

    private static void processOrderTransactionally(
            PeeGeeQManager manager,
            MessageProducer<OrderEvent> orderProducer,
            MessageProducer<EmailEvent> emailProducer) throws Exception {

        // Get database connection for transaction
        try (Connection conn = manager.getDatabaseService().getConnection()) {
            conn.setAutoCommit(false);

            try {
                // 1. Create order in database
                String orderId = "ORDER-" + System.currentTimeMillis();
                PreparedStatement stmt = conn.prepareStatement(
                    "INSERT INTO orders (id, customer_id, amount, status) VALUES (?, ?, ?, ?)");
                stmt.setString(1, orderId);
                stmt.setString(2, "CUST-123");
                stmt.setBigDecimal(3, new BigDecimal("99.99"));
                stmt.setString(4, "PENDING");
                stmt.executeUpdate();

                // 2. Send order event (within same transaction!)
                OrderEvent orderEvent = new OrderEvent(orderId, "CUST-123", new BigDecimal("99.99"));
                orderProducer.send(orderEvent).join();

                // 3. Send email event (within same transaction!)
                EmailEvent emailEvent = new EmailEvent(
                    "customer@example.com",
                    "Order Confirmation",
                    "Your order " + orderId + " has been received.");
                emailProducer.send(emailEvent).join();

                // 4. Commit everything together
                conn.commit();
                System.out.println("✅ Order, events, and emails committed together!");

            } catch (Exception e) {
                conn.rollback();
                System.err.println("❌ Transaction rolled back: " + e.getMessage());
                throw e;
            }
        }
    }

    private static void processOrder(OrderEvent event) {
        // Implementation for order processing
    }

    private static void sendEmail(EmailEvent event) {
        // Implementation for email sending
    }
}

class OrderEvent {
    private String orderId;
    private String customerId;
    private BigDecimal amount;
    private Instant timestamp;

    // Constructor, getters, setters...
}

class EmailEvent {
    private String toEmail;
    private String subject;
    private String body;
    private Instant timestamp;

    // Constructor, getters, setters...
}
```

🎯 **Try This Now**:

1. Create the OrderEvent and EmailEvent classes
2. Run the example and observe transactional consistency
3. Try introducing an error after the database insert to see rollback behavior

Stuck Message Recovery

The outbox pattern includes a sophisticated **stuck message recovery mechanism** that automatically handles the critical issue where consumer crashes can leave messages in "PROCESSING" state indefinitely.

The Problem

When a consumer crashes after polling messages but before completing processing, messages can get stuck in the PROCESSING state. Without recovery, these messages would never be processed, leading to:

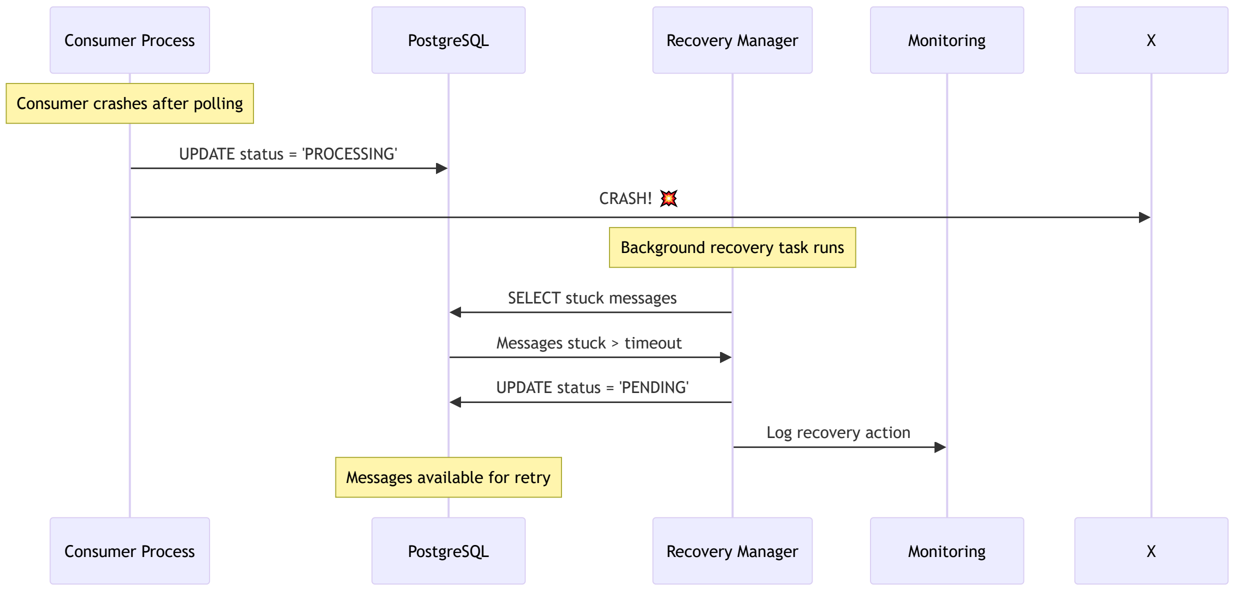
- **Lost messages** that never reach their destination
- **Inconsistent system state** where business operations appear complete but notifications weren't sent
- **Manual intervention required** to identify and fix stuck messages

The Solution

PeeGeeQ's **StuckMessageRecoveryManager** automatically:

1. **Identifies stuck messages** - Finds messages in PROCESSING state longer than a configurable timeout
2. **Safely recovers them** - Resets messages back to PENDING state for retry
3. **Preserves message integrity** - Maintains retry counts and error messages
4. **Provides audit trails** - Logs all recovery actions for monitoring

How It Works


Consumer Process
PostgreSQL
Recovery Manager
Monitoring
X
Consumer crashes after polling
UPDATE status = 'PROCESSING'
CRASH! 💥
Background recovery task runs
SELECT stuck messages
Messages stuck > timeout
UPDATE status = 'PENDING'
Log recovery action
Messages available for retry

Configuration

The recovery mechanism is fully configurable:

```
# Enable/disable stuck message recovery (default: true)
peegeeq.queue.recovery.enabled=true

# How long before a message is considered stuck (default: 5 minutes)
peegeeq.queue.recovery.processing-timeout=PT5M

# How often to check for stuck messages (default: 10 minutes)
peegeeq.queue.recovery.check-interval=PT10M
```

Environment-Specific Settings

Development Environment (faster recovery for testing):

```
peegeeq.queue.recovery.processing-timeout=PT1M
peegeeq.queue.recovery.check-interval=PT2M
```

Production Environment (conservative settings):

```
peegeeq.queue.recovery.processing-timeout=PT10M
peegeeq.queue.recovery.check-interval=PT15M
```

High-Reliability Environment (aggressive recovery):

```
peegeeq.queue.recovery.processing-timeout=PT3M
peegeeq.queue.recovery.check-interval=PT5M
```

Monitoring and Observability

The recovery manager provides comprehensive monitoring:

```
// Get recovery statistics
StuckMessageRecoveryManager.RecoveryStats stats =
    manager.getStuckMessageRecoveryManager().getRecoveryStats();

System.out.println("Stuck messages: " + stats.getStuckMessagesCount());
System.out.println("Total processing: " + stats.getTotalProcessingCount());
System.out.println("Recovery enabled: " + stats.isEnabled());
```

Log Output Example:

```
INFO  StuckMessageRecoveryManager - Found 3 stuck messages in PROCESSING state for longer than PT5M
INFO  StuckMessageRecoveryManager - Recovered stuck message: id=1234, topic=orders, retryCount=1, lastError=none
INFO  StuckMessageRecoveryManager - Successfully recovered 3 stuck messages from PROCESSING to PENDING state
```

Production Benefits

✅ **Automatic Recovery**: No manual intervention required for stuck messages ✅ **Zero Message Loss**: Ensures all messages are eventually processed ✅ **Configurable Timeouts**: Adapt to your application's processing patterns ✅ **Comprehensive

Logging: Full audit trail for compliance and debugging ✅ **Performance Impact**: Minimal overhead with configurable check intervals ✅ **Safe Operation**: Conservative approach preserves message integrity

Practical Example: Testing Stuck Message Recovery

```
public class StuckMessageRecoveryExample {
    public static void main(String[] args) throws Exception {
        try (PeeGeeQManager manager = new PeeGeeQManager()) {
            manager.start();

            // Create outbox factory
            QueueFactoryProvider provider = QueueFactoryProvider.getInstance();
            QueueFactory factory = provider.createFactory("outbox", manager.getDatabaseService());

            // Send test messages
            try (MessageProducer<String> producer = factory.createProducer("test-recovery", String.class)) {
                for (int i = 0; i < 3; i++) {
                    producer.send("Test message " + i + " for recovery").get();
                }
                System.out.println("📤 Sent 3 test messages");
            }

            // Simulate consumer crash by forcing messages into PROCESSING state
            int stuckCount = simulateConsumerCrash(manager.getDatabaseService(), "test-recovery");
            System.out.println("💥 Simulated consumer crash - " + stuckCount + " messages stuck in PROCESSING");

            // Create recovery manager with short timeout for demonstration
            StuckMessageRecoveryManager recoveryManager =
                new StuckMessageRecoveryManager(manager.getDatabaseService().getDataSource(),
                                               Duration.ofSeconds(2), true);

            // Check stats before recovery
            StuckMessageRecoveryManager.RecoveryStats beforeStats = recoveryManager.getRecoveryStats();
            System.out.println("📊 Before recovery: " + beforeStats);

            // Wait for messages to be considered stuck
            Thread.sleep(3000);

            // Manually trigger recovery (normally runs automatically)
            int recovered = recoveryManager.recoverStuckMessages();
            System.out.println("🔧 Recovered " + recovered + " stuck messages");

            // Check stats after recovery
            StuckMessageRecoveryManager.RecoveryStats afterStats = recoveryManager.getRecoveryStats();
            System.out.println("📊 After recovery: " + afterStats);
        }
    }

    private static int simulateConsumerCrash(DatabaseService databaseService, String topic) throws Exception {
        // This simulates the exact moment when a consumer polls messages (moves to PROCESSING)
        // but crashes before completing processing
        try (Connection conn = databaseService.getConnectionProvider()
                .getDataSource("peegeeq-main").getConnection()) {

            String sql = """
                UPDATE outbox
                SET status = 'PROCESSING', processed_at = ?
                WHERE id IN (
                    SELECT id FROM outbox
                    WHERE topic = ? AND status = 'PENDING'
                    ORDER BY created_at ASC
                    LIMIT 3
                )
                """;
            try (PreparedStatement stmt = conn.prepareStatement(sql)) {
                // Set processed_at to 5 minutes ago to simulate stuck messages
                stmt.setTimestamp(1, Timestamp.from(Instant.now().minusSeconds(300)));
                stmt.setString(2, topic);
                return stmt.executeUpdate();
            }
        }
    }
}
```

Expected Output:

```
📤 Sent 3 test messages
💥 Simulated consumer crash - 3 messages stuck in PROCESSING
📊 Before recovery: RecoveryStats{stuck=3, totalProcessing=3, enabled=true}
🔧 Found 3 stuck messages in PROCESSING state for longer than PT2S
🔧 Recovered stuck message: id=123, topic=test-recovery, retryCount=0, lastError=none
🔧 Recovered stuck message: id=124, topic=test-recovery, retryCount=0, lastError=none
🔧 Recovered stuck message: id=125, topic=test-recovery, retryCount=0, lastError=none
🔧 Successfully recovered 3 stuck messages from PROCESSING to PENDING state
🔧 Recovered 3 stuck messages
📊 After recovery: RecoveryStats{stuck=0, totalProcessing=0, enabled=true}
```

🎯 **Try This Now**:

1. Run the example above to see recovery in action
2. Create an outbox consumer and simulate a crash (kill the process)
3. Observe messages stuck in PROCESSING state in the database
4. Watch the recovery manager automatically reset them to PENDING
5. Monitor the recovery logs and statistics

Filter Error Handling (Deep Dive)

The Filter Error Handling system provides enterprise-grade error handling with sophisticated recovery patterns designed to maintain message reliability while providing graceful degradation under failure conditions.

Understanding Filter Error Handling

When processing messages through filters, various types of errors can occur:

- **Transient Errors**: Temporary failures like network timeouts that may succeed on retry
- **Permanent Errors**: Persistent failures like invalid data that won't succeed on retry
- **Unknown Errors**: Errors that don't match predefined patterns

The Filter Error Handling system automatically classifies these errors and applies appropriate recovery strategies.

Configuration Patterns

High-Reliability Configuration

For critical business processes where message loss is unacceptable:

```
FilterErrorHandlingConfig highReliabilityConfig = FilterErrorHandlingConfig.builder()
    // Comprehensive error classification
    .addTransientErrorPattern("timeout")
    .addTransientErrorPattern("connection")
    .addTransientErrorPattern("network")
    .addPermanentErrorPattern("invalid")
    .addPermanentErrorPattern("unauthorized")
    .addPermanentErrorPattern("malformed")

    // Aggressive retry strategy
    .defaultStrategy(FilterErrorStrategy.RETRY_THEN_DEAD_LETTER)
    .maxRetries(5)
    .initialRetryDelay(Duration.ofMillis(200))
    .retryBackoffMultiplier(2.0)
    .maxRetryDelay(Duration.ofMinutes(1))

    // Conservative circuit breaker
    .circuitBreakerEnabled(true)
    .circuitBreakerFailureThreshold(10)
    .circuitBreakerMinimumRequests(20)
    .circuitBreakerTimeout(Duration.ofMinutes(2))

    // Comprehensive DLQ
    .deadLetterQueueEnabled(true)
    .deadLetterQueueTopic("critical-errors")
    .build();
```

High-Performance Configuration

For high-volume scenarios where performance is prioritized:

```
FilterErrorHandlingConfig highPerformanceConfig = FilterErrorHandlingConfig.builder()
    // Minimal error classification
    .addPermanentErrorPattern("invalid")

    // Fast rejection strategy
    .defaultStrategy(FilterErrorStrategy.REJECT_IMMEDIATELY)
    .maxRetries(1)
    .initialRetryDelay(Duration.ofMillis(10))

    // Aggressive circuit breaker
    .circuitBreakerEnabled(true)
    .circuitBreakerFailureThreshold(3)
    .circuitBreakerMinimumRequests(5)
    .circuitBreakerTimeout(Duration.ofSeconds(30))

    // No DLQ for performance
    .deadLetterQueueEnabled(false)
    .build();
```

Balanced Configuration

For general business applications balancing reliability and performance:

```
FilterErrorHandlingConfig balancedConfig = FilterErrorHandlingConfig.builder()
    // Standard error classification
    .addTransientErrorPattern("timeout")
    .addTransientErrorPattern("connection")
    .addPermanentErrorPattern("invalid")
    .addPermanentErrorPattern("unauthorized")

    // Moderate retry strategy
    .defaultStrategy(FilterErrorStrategy.RETRY_THEN_REJECT)
    .maxRetries(3)
    .initialRetryDelay(Duration.ofMillis(100))
    .retryBackoffMultiplier(2.0)
    .maxRetryDelay(Duration.ofSeconds(30))

    // Standard circuit breaker
    .circuitBreakerEnabled(true)
    .circuitBreakerFailureThreshold(5)
    .circuitBreakerMinimumRequests(10)
    .circuitBreakerTimeout(Duration.ofMinutes(1))

    // Selective DLQ
    .deadLetterQueueEnabled(true)
    .deadLetterQueueTopic("business-errors")
    .build();
```

Practical Example: Order Processing with Error Handling

```
public class OrderProcessingExample {

    public static void main(String[] args) throws Exception {
        // Configure sophisticated error handling
        FilterErrorHandlingConfig config = FilterErrorHandlingConfig.builder()
            .addTransientErrorPattern("payment_timeout")
            .addTransientErrorPattern("inventory_check_failed")
            .addPermanentErrorPattern("invalid_customer")
            .addPermanentErrorPattern("product_not_found")
            .defaultStrategy(FilterErrorStrategy.RETRY_THEN_DEAD_LETTER)
            .maxRetries(3)
            .initialRetryDelay(Duration.ofMillis(100))
            .retryBackoffMultiplier(2.0)
            .circuitBreakerEnabled(true)
            .circuitBreakerFailureThreshold(5)
            .deadLetterQueueEnabled(true)
            .deadLetterQueueTopic("order-processing-errors")
            .build();

        // Create order validation filter
        Predicate<Message<OrderEvent>> orderValidationFilter = message -> {
            OrderEvent order = message.getPayload();

            // Simulate different types of errors
            if (order.getCustomerId() == null) {
                throw new IllegalArgumentException("invalid_customer: Customer ID is required");
            }

            if (order.getProductId().startsWith("TEMP_")) {
                throw new RuntimeException("inventory_check_failed: Temporary inventory system unavailable");
            }

            if (order.getAmount().compareTo(BigDecimal.ZERO) <= 0) {
                throw new IllegalArgumentException("invalid_order: Order amount must be positive");
            }

            return true; // Order is valid
        };

        // Create order handler
        MessageHandler<OrderEvent> orderHandler = message -> {
```

```java
            OrderEvent order = message.getPayload();
            System.out.println("✅ Processing valid order: " + order.getOrderId());

            // Simulate order processing
            return CompletableFuture.completedFuture(null);
        };

        // Create consumer with error handling
        OutboxConsumerGroupMember<OrderEvent> consumer = new OutboxConsumerGroupMember<>(
            "order-processor",
            "order-group",
            "orders",
            orderHandler,
            orderValidationFilter,
            null,
            config  // Apply sophisticated error handling
        );

        consumer.start();

        // Send test orders with different error scenarios
        sendTestOrders();

        // Monitor error handling metrics
        monitorErrorHandling(consumer);

        Thread.sleep(10000); // Let processing complete
        consumer.close();
    }

    private static void sendTestOrders() {
        // Valid order
        OrderEvent validOrder = new OrderEvent("ORDER-001", "CUST-123", "PROD-456", new BigDecimal("99.99"));

        // Invalid customer (permanent error)
        OrderEvent invalidCustomer = new OrderEvent("ORDER-002", null, "PROD-456", new BigDecimal("99.99"));

        // Temporary inventory issue (transient error)
        OrderEvent tempInventory = new OrderEvent("ORDER-003", "CUST-123", "TEMP_PROD-789", new BigDecimal("99.99"));

        // Invalid amount (permanent error)
        OrderEvent invalidAmount = new OrderEvent("ORDER-004", "CUST-123", "PROD-456", new BigDecimal("-10.00"));

        // Send orders (implementation depends on your message producer)
        System.out.println("📤 Sending test orders with various error scenarios...");
    }

    private static void monitorErrorHandling(OutboxConsumerGroupMember<OrderEvent> consumer) {
        // Monitor circuit breaker state
        FilterCircuitBreaker.CircuitBreakerMetrics cbMetrics = consumer.getFilterCircuitBreakerMetrics();
        System.out.println("🔌 Circuit Breaker State: " + cbMetrics.getState());
        System.out.println("📊 Failure Rate: " + String.format("%.2f%%", cbMetrics.getFailureRate() * 100));

        // In a real application, you would also monitor:
        // - Dead letter queue metrics
        // - Retry attempt metrics
        // - Overall processing success rates
    }
}

// Order event class
class OrderEvent {
    private final String orderId;
    private final String customerId;
    private final String productId;
    private final BigDecimal amount;
    public OrderEvent(String orderId, String customerId, String productId, BigDecimal amount) {
        this.orderId = orderId;
        this.customerId = customerId;
        this.productId = productId;
        this.amount = amount;
    }

    // Getters
    public String getOrderId() { return orderId; }
    public String getCustomerId() { return customerId; }
    public String getProductId() { return productId; }
    public BigDecimal getAmount() { return amount; }
}
```

Key Benefits

1. **No Message Loss**: Critical messages are never lost due to filter errors
2. **Intelligent Recovery**: Different strategies for different types of errors
3. **Circuit Breaker Protection**: Prevents cascading failures during outages
4. **Performance Optimization**: Fast-fail behavior when appropriate
5. **Comprehensive Monitoring**: Rich metrics for production observability

🎯 **Try This Now**:

1. Create the OrderEvent class and error handling configuration
2. Run the example and observe how different errors are handled
3. Monitor the circuit breaker state during error scenarios
4. Experiment with different configuration patterns

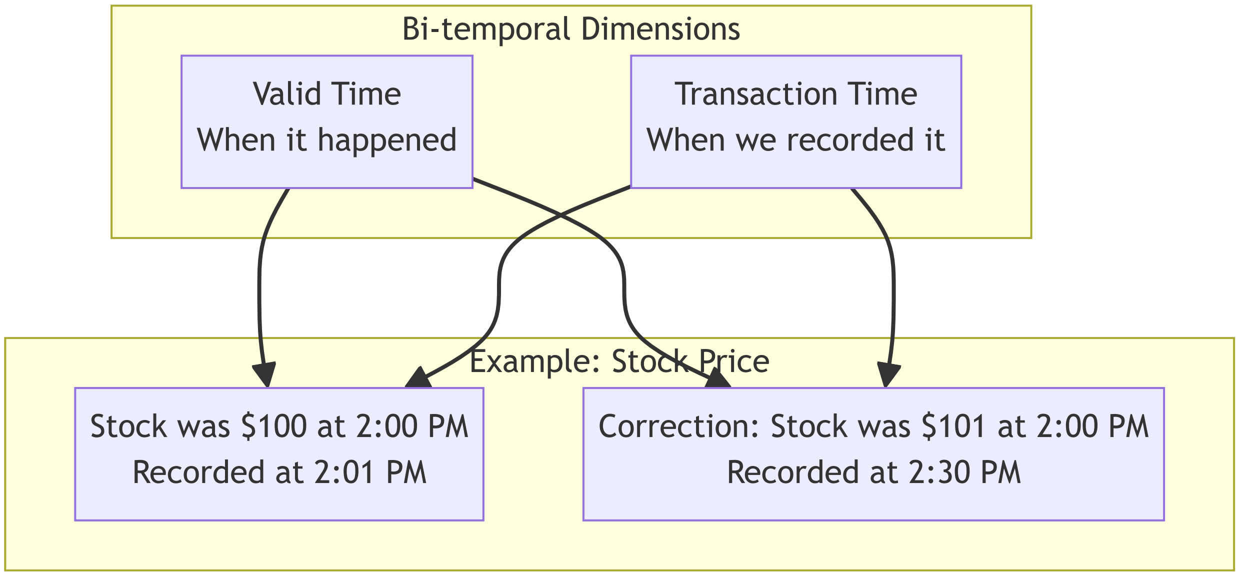
Bi-temporal Event Store (Deep Dive)

The Bi-temporal Event Store pattern provides event sourcing capabilities with the ability to query data as it existed at any point in time and as it was known at any point in time.

Understanding Bi-temporal

Bi-temporal means tracking two different time dimensions:

1. **Valid Time**: When the event actually occurred in the real world
2. **Transaction Time**: When the event was recorded in the system



Key Characteristics

Performance

- **Throughput**: 3,000+ events/second
- **Query Performance**: Optimized for temporal queries
- **Storage**: Append-only, no updates or deletes

Capabilities

- **Event Sourcing**: Rebuild state from events
- **Time Travel**: Query data as it existed at any point
- **Audit Trail**: Complete history of all changes
- **Correction Handling**: Handle late-arriving or corrected data

Technical Implementation

- **Append-only table**: Events are never updated or deleted
- **Temporal indexes**: Optimized for time-based queries
- **Event replay**: Rebuild current state from events
- **Snapshot support**: Periodic snapshots for performance

When to Use Bi-temporal Event Store

✅ Perfect for:

- **Financial systems** (trading, accounting, compliance)
- **Audit trails** (regulatory compliance, forensic analysis)
- **Event sourcing** (CQRS, domain-driven design)
- **Historical analysis** (business intelligence, reporting)
- **Correction handling** (late data, error corrections)

❌ Not ideal for:

- **Simple CRUD operations** (use regular database)
- **High-frequency, low-value events** (use Native Queue)
- **Scenarios where history isn't important**

Bi-temporal Event Store Example

```java
public class BiTemporalEventStoreExample {
    public static void main(String[] args) throws Exception {
        try (PeeGeeQManager manager = new PeeGeeQManager()) {
            manager.start();

            // Create bi-temporal event store
            QueueFactoryProvider provider = QueueFactoryProvider.getInstance();
            QueueFactory factory = provider.createFactory("bitemporal", manager.getDatabaseService());

            try (MessageProducer<AccountEvent> producer =
                     factory.createProducer("account-events", AccountEvent.class);
                 MessageConsumer<AccountEvent> consumer =
                     factory.createConsumer("account-events", AccountEvent.class)) {

                // Setup event consumer
                consumer.subscribe(message -> {
                    AccountEvent event = message.getPayload();
                    System.out.printf("💰 Account Event: %s - %s: $%.2f%n",
                        event.getAccountId(), event.getEventType(), event.getAmount());

                    // Update account balance projection
                    updateAccountProjection(event);

                    return CompletableFuture.completedFuture(null);
                });

                // Simulate account events over time
                String accountId = "ACC-123";

                // Initial deposit
                producer.send(new AccountEvent(accountId, "DEPOSIT",
                    new BigDecimal("1000.00"), Instant.now()));

                Thread.sleep(100);

                // Withdrawal
                producer.send(new AccountEvent(accountId, "WITHDRAWAL",
                    new BigDecimal("250.00"), Instant.now()));

                Thread.sleep(100);

                // Another deposit
                producer.send(new AccountEvent(accountId, "DEPOSIT",
                    new BigDecimal("500.00"), Instant.now()));

                Thread.sleep(100);

                // Late-arriving correction (happened before withdrawal)
                Instant correctionTime = Instant.now().minus(200, ChronoUnit.MILLIS);
                producer.send(new AccountEvent(accountId, "CORRECTION",
                    new BigDecimal("50.00"), correctionTime));

                Thread.sleep(2000); // Let events process

                // Query account balance at different points in time
                queryAccountHistory(manager, accountId);
            }
        }
    }

    private static void updateAccountProjection(AccountEvent event) {
        // Update current account balance projection
        // This would typically update a read model/projection
    }

    private static void queryAccountHistory(PeeGeeQManager manager, String accountId)
            throws Exception {

        System.out.println("\n📊 Account History Analysis:");

        // Query events for this account
        try (Connection conn = manager.getDatabaseService().getConnection()) {
            PreparedStatement stmt = conn.prepareStatement(
                "SELECT event_type, amount, valid_time, transaction_time " +
                "FROM bitemporal_event_log " +
                "WHERE payload->>'accountId' = ? " +
                "ORDER BY valid_time, transaction_time");
            stmt.setString(1, accountId);

            ResultSet rs = stmt.executeQuery();
            BigDecimal balance = BigDecimal.ZERO;

            while (rs.next()) {
                String eventType = rs.getString("event_type");
                BigDecimal amount = rs.getBigDecimal("amount");
                Instant validTime = rs.getTimestamp("valid_time").toInstant();
                Instant transactionTime = rs.getTimestamp("transaction_time").toInstant();

                if ("DEPOSIT".equals(eventType) || "CORRECTION".equals(eventType)) {
                    balance = balance.add(amount);
                } else if ("WITHDRAWAL".equals(eventType)) {
                    balance = balance.subtract(amount);
                }

                System.out.printf("  %s: $%.2f (Valid: %s, Recorded: %s) - Balance: $%.2f%n",
                    eventType, amount, validTime, transactionTime, balance);
            }
        }
    }
}

class AccountEvent {
    private String accountId;
    private String eventType;
    private BigDecimal amount;
    private Instant validTime;
    private Instant transactionTime;

    public AccountEvent(String accountId, String eventType, BigDecimal amount, Instant validTime) {
        this.accountId = accountId;
        this.eventType = eventType;
        this.amount = amount;
        this.validTime = validTime;
        this.transactionTime = Instant.now(); // When we're recording it
    }

    // Getters and setters...
}
```

🎯 **Try This Now**:

1. Create the AccountEvent class
2. Run the example and observe the event sourcing pattern
3. Try adding more events and corrections to see bi-temporal behavior

Maven Dependencies

To use PeeGeeQ in your project, add these dependencies:

```xml
<dependencies>
    <!-- Core API -->
    <dependency>
        <groupId>dev.mars.peegeeq</groupId>
        <artifactId>peegeeq-api</artifactId>
        <version>1.0.0</version>
    </dependency>

    <!-- Database management -->
    <dependency>
        <groupId>dev.mars.peegeeq</groupId>
        <artifactId>peegeeq-db</artifactId>
        <version>1.0.0</version>
    </dependency>

    <!-- Choose your implementation(s) -->
    <dependency>
        <groupId>dev.mars.peegeeq</groupId>
        <artifactId>peegeeq-native</artifactId>
        <version>1.0.0</version>
    </dependency>

    <dependency>
        <groupId>dev.mars.peegeeq</groupId>
        <artifactId>peegeeq-outbox</artifactId>
        <version>1.0.0</version>
    </dependency>

    <!-- Optional: Bi-temporal event store -->
    <dependency>
        <groupId>dev.mars.peegeeq</groupId>
        <artifactId>peegeeq-bitemporal</artifactId>
        <version>1.0.0</version>
    </dependency>
</dependencies>
```

Note: You can use multiple patterns in the same application. Each serves different use cases and they complement each other well.

Part V: Practical Examples (Progressive Complexity)

Now that you understand the core concepts, let's build real-world applications with increasing complexity. We'll start with simple examples and gradually introduce more sophisticated patterns.

Level 1: Basic Examples

These examples focus on fundamental messaging patterns that you'll use in most applications.

Example 1: Simple Producer/Consumer

Let's build a basic task processing system:

```java
public class TaskProcessingSystem {
    public static void main(String[] args) throws Exception {
        try (PeeGeeQManager manager = new PeeGeeQManager()) {
            manager.start();

            QueueFactoryProvider provider = QueueFactoryProvider.getInstance();
            QueueFactory factory = provider.createFactory("native", manager.getDatabaseService());

            try (MessageProducer<Task> taskProducer =
                     factory.createProducer("tasks", Task.class);
                 MessageConsumer<Task> taskConsumer =
                     factory.createConsumer("tasks", Task.class)) {

                // Setup task processor
                taskConsumer.subscribe(message -> {
                    Task task = message.getPayload();
                    System.out.printf("🔄 Processing task: %s (Priority: %d)%n",
                        task.getDescription(), task.getPriority());

                    // Simulate task processing
                    try {
                        Thread.sleep(task.getProcessingTimeMs());
                        System.out.printf("✅ Completed task: %s%n", task.getDescription());
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                        return CompletableFuture.failedFuture(e);
                    }

                    return CompletableFuture.completedFuture(null);
                });

                // Submit various tasks
                taskProducer.send(new Task("Process user registration", 1, 1000));
                taskProducer.send(new Task("Generate monthly report", 3, 5000));
                taskProducer.send(new Task("Send welcome email", 2, 500));
                taskProducer.send(new Task("Update search index", 2, 2000));

                Thread.sleep(10000); // Let tasks process
            }
        }
    }
}

class Task {
    private String description;
    private int priority;
    private long processingTimeMs;
    private Instant createdAt;

    public Task(String description, int priority, long processingTimeMs) {
        this.description = description;
        this.priority = priority;
        this.processingTimeMs = processingTimeMs;
        this.createdAt = Instant.now();
    }

    // Getters and setters...
}
```

🎯 **Try This Now**:

1. Run the example and observe task processing
2. Add more tasks with different priorities
3. Try running multiple instances to see load balancing

Example 2: Message Serialization with Complex Objects

Let's handle more complex data structures:

```
public class UserEventSystem {
    public static void main(String[] args) throws Exception {
        try (PeeGeeQManager manager = new PeeGeeQManager()) {
            manager.start();

            QueueFactoryProvider provider = QueueFactoryProvider.getInstance();
            QueueFactory factory = provider.createFactory("native", manager.getDatabaseService());

            try (MessageProducer<UserEvent> producer =
                     factory.createProducer("user-events", UserEvent.class);
                 MessageConsumer<UserEvent> consumer =
                     factory.createConsumer("user-events", UserEvent.class)) {

                // Setup event processor
                consumer.subscribe(message -> {
                    UserEvent event = message.getPayload();
                    System.out.printf("🧑 User Event: %s - %s%n",
                        event.getEventType(), event.getUser().getEmail());

                    // Process based on event type
                    switch (event.getEventType()) {
                        case REGISTERED:
                            handleUserRegistration(event);
                            break;
                        case LOGIN:
                            handleUserLogin(event);
                            break;
                        case PROFILE_UPDATED:
                            handleProfileUpdate(event);
                            break;
                    }

                    return CompletableFuture.completedFuture(null);
                });

                // Create sample user events
                User user1 = new User("john@example.com", "John Doe", "Premium");
                User user2 = new User("jane@example.com", "Jane Smith", "Basic");

                // Send various events
                producer.send(new UserEvent(UserEventType.REGISTERED, user1,
                    Map.of("source", "web", "campaign", "spring2024")));

                producer.send(new UserEvent(UserEventType.LOGIN, user1,
                    Map.of("ip", "192.168.1.100", "device", "mobile")));

                producer.send(new UserEvent(UserEventType.PROFILE_UPDATED, user2,
                    Map.of("field", "subscription", "oldValue", "Basic", "newValue", "Premium")));
                Thread.sleep(3000);
            }
        }
    }

    private static void handleUserRegistration(UserEvent event) {
        System.out.println("  📧 Sending welcome email...");
        System.out.println("  🎁 Creating welcome bonus...");
        System.out.println("  📊 Updating analytics...");
    }

    private static void handleUserLogin(UserEvent event) {
        System.out.println("  🔐 Updating last login time...");
        System.out.println("  📍 Recording login location...");
    }

    private static void handleProfileUpdate(UserEvent event) {
        System.out.println("  💾 Syncing profile changes...");
        System.out.println("  🔄 Invalidating cache...");
    }
}

enum UserEventType {
    REGISTERED, LOGIN, LOGOUT, PROFILE_UPDATED, SUBSCRIPTION_CHANGED
}

class UserEvent {
    private UserEventType eventType;
    private User user;
    private Map<String, String> metadata;
    private Instant timestamp;

    public UserEvent(UserEventType eventType, User user, Map<String, String> metadata) {
        this.eventType = eventType;
        this.user = user;
        this.metadata = metadata;
        this.timestamp = Instant.now();
    }

    // Getters and setters...
}

class User {
    private String email;
    private String name;
    private String subscriptionTier;

    public User(String email, String name, String subscriptionTier) {
        this.email = email;
        this.name = name;
        this.subscriptionTier = subscriptionTier;
    }

    // Getters and setters...
}
```

Example 3: Basic Error Handling and Retry Logic

Let's add robust error handling:

```
public class ResilientMessageProcessing {
    public static void main(String[] args) throws Exception {
        try (PeeGeeQManager manager = new PeeGeeQManager()) {
```

```java
        manager.start();

        QueueFactoryProvider provider = QueueFactoryProvider.getInstance();
        QueueFactory factory = provider.createFactory("native", manager.getDatabaseService());

        try (MessageProducer<ProcessingTask> producer =
                 factory.createProducer("processing-tasks", ProcessingTask.class);
             MessageConsumer<ProcessingTask> consumer =
                 factory.createConsumer("processing-tasks", ProcessingTask.class)) {

            // Setup resilient processor with retry logic
            consumer.subscribe(message -> {
                ProcessingTask task = message.getPayload();

                return processWithRetry(task, 3) // Max 3 retries
                    .exceptionally(throwable -> {
                        System.err.printf("❌ Failed to process task %s after retries: %s%n",
                            task.getId(), throwable.getMessage());

                        // Send to dead letter queue or alert system
                        handleFailedTask(task, throwable);

                        return null;
                    });
            });

            // Send tasks with different failure probabilities
            producer.send(new ProcessingTask("TASK-001", "reliable-operation", 0.1)); // 10% failure
            producer.send(new ProcessingTask("TASK-002", "flaky-operation", 0.7));    // 70% failure
            producer.send(new ProcessingTask("TASK-003", "stable-operation", 0.0));   // Never fails
            producer.send(new ProcessingTask("TASK-004", "unstable-operation", 0.9)); // 90% failure

            Thread.sleep(10000);
        }
    }
}

private static CompletableFuture<Void> processWithRetry(ProcessingTask task, int maxRetries) {
    return CompletableFuture.supplyAsync(() -> {
        int attempt = 0;
        Exception lastException = null;

        while (attempt <= maxRetries) {
            try {
                attempt++;
                System.out.printf("🔄 Processing %s (attempt %d/%d)%n",
                    task.getId(), attempt, maxRetries + 1);

                // Simulate processing that might fail
                if (Math.random() < task.getFailureProbability()) {
                    throw new RuntimeException("Simulated processing failure");
                }

                System.out.printf("✅ Successfully processed %s%n", task.getId());
                return null;

            } catch (Exception e) {
                lastException = e;
                System.out.printf("⚠️ Attempt %d failed for %s: %s%n",
                    attempt, task.getId(), e.getMessage());

                if (attempt <= maxRetries) {
                    try {
                        // Exponential backoff
                        long delay = (long) (1000 * Math.pow(2, attempt - 1));
                        Thread.sleep(delay);
                        } catch (InterruptedException ie) {
                            Thread.currentThread().interrupt();
                            throw new RuntimeException("Interrupted during retry", ie);
                        }
                    }
                }
            }

            throw new RuntimeException("Max retries exceeded", lastException);
        });
    }

    private static void handleFailedTask(ProcessingTask task, Throwable error) {
        System.err.printf("💀 Moving task %s to dead letter queue%n", task.getId());
        // Implementation would send to DLQ or alert system
    }
}

class ProcessingTask {
    private String id;
    private String operation;
    private double failureProbability;
    private Instant createdAt;

    public ProcessingTask(String id, String operation, double failureProbability) {
        this.id = id;
        this.operation = operation;
        this.failureProbability = failureProbability;
        this.createdAt = Instant.now();
    }

    // Getters and setters...
}
```

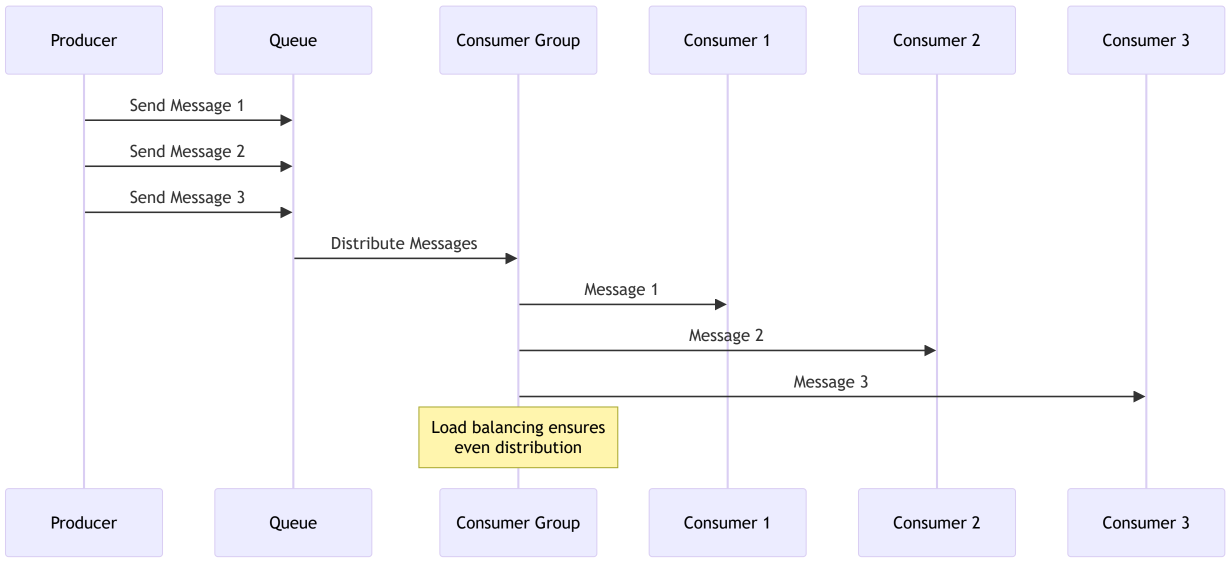
🎯 **Try This Now**:

1. Run the example and observe retry behavior
2. Adjust failure probabilities to see different outcomes
3. Add logging to track retry patterns

Consumer Groups & Load Balancing (Deep Dive)

Consumer Groups provide sophisticated load balancing and message distribution capabilities, allowing multiple consumers to work together efficiently to process messages from a single topic.

How Consumer Groups Work



Key Features

Load Balancing

- **Round-robin distribution**: Messages distributed evenly across consumers
- **Automatic failover**: Failed consumers are removed from rotation
- **Dynamic scaling**: Add/remove consumers without interruption
- **Message affinity**: Route messages based on content or headers

Message Filtering

- **Consumer-level filters**: Each consumer can specify message criteria
- **Group-level filters**: Apply filters to the entire consumer group
- **Header-based routing**: Route messages based on header values
- **Content-based filtering**: Filter messages based on payload content

Parallel Processing

- **Configurable thread pools**: Control parallel processing per consumer
- **Batch processing**: Process multiple messages simultaneously
- **Backpressure handling**: Automatic throttling under high load
- **Resource management**: Efficient memory and connection usage

Consumer Group Example

```java
public class ConsumerGroupExample {
    public static void main(String[] args) throws Exception {
        try (PeeGeeQManager manager = new PeeGeeQManager()) {
            manager.start();

            QueueFactoryProvider provider = QueueFactoryProvider.getInstance();
            QueueFactory factory = provider.createFactory("outbox", manager.getDatabaseService());

            // Create consumer group for order processing
            try (ConsumerGroup<OrderEvent> orderGroup =
                     factory.createConsumerGroup("order-processors", "orders", OrderEvent.class);
```

```
                 MessageProducer<OrderEvent> producer =
                     factory.createProducer("orders", OrderEvent.class)) {

                // Add region-specific consumers with filters
                orderGroup.addConsumer("us-processor",
                    message -> {
                        System.out.println("US Processor: " + message.getPayload().getOrderId());
                        return processUSOrder(message.getPayload());
                    },
                    message -> "US".equals(message.getHeaders().get("region"))
                );

                orderGroup.addConsumer("eu-processor",
                    message -> {
                        System.out.println("EU Processor: " + message.getPayload().getOrderId());
                        return processEUOrder(message.getPayload());
                    },
                    message -> "EU".equals(message.getHeaders().get("region"))
                );

                orderGroup.addConsumer("asia-processor",
                    message -> {
                        System.out.println("ASIA Processor: " + message.getPayload().getOrderId());
                        return processAsiaOrder(message.getPayload());
                    },
                    message -> "ASIA".equals(message.getHeaders().get("region"))
                );

                // Start the consumer group
                orderGroup.start();

                // Send orders to different regions
                Map<String, String> usHeaders = Map.of("region", "US");
                Map<String, String> euHeaders = Map.of("region", "EU");
                Map<String, String> asiaHeaders = Map.of("region", "ASIA");

                producer.send(new OrderEvent("US-001", 99.99), usHeaders);
                producer.send(new OrderEvent("EU-001", 149.99), euHeaders);
                producer.send(new OrderEvent("ASIA-001", 79.99), asiaHeaders);

                Thread.sleep(5000); // Let messages process
            }
        }
    }

    private static CompletableFuture<Void> processUSOrder(OrderEvent order) {
        // US-specific processing logic
        return CompletableFuture.completedFuture(null);
    }

    private static CompletableFuture<Void> processEUOrder(OrderEvent order) {
        // EU-specific processing logic (GDPR compliance, etc.)
        return CompletableFuture.completedFuture(null);
    }

    private static CompletableFuture<Void> processAsiaOrder(OrderEvent order) {
        // Asia-specific processing logic
        return CompletableFuture.completedFuture(null);
    }
}
```

Parallel Processing Configuration

Configure parallel processing for high-throughput scenarios:

```
# Consumer thread configuration
peegeeq.consumer.threads=4
peegeeq.queue.batch-size=10
peegeeq.queue.polling-interval=PT0.1S

# Backpressure management
peegeeq.consumer.max-concurrent-operations=50
peegeeq.consumer.timeout=PT30S
```

Advanced Consumer Group Patterns

Priority-based Processing

```
// High-priority consumer
orderGroup.addConsumer("high-priority-processor",
    message -> processHighPriorityOrder(message.getPayload()),
    message -> {
        Integer priority = Integer.parseInt(
            message.getHeaders().getOrDefault("priority", "5"));
        return priority >= 8; // Process only high-priority messages
    }
);

// Normal priority consumer
orderGroup.addConsumer("normal-processor",
    message -> processNormalOrder(message.getPayload()),
    message -> {
        Integer priority = Integer.parseInt(
            message.getHeaders().getOrDefault("priority", "5"));
        return priority < 8; // Process normal priority messages
    }
);
```

Customer Tier Processing

```
// VIP customer processor
orderGroup.addConsumer("vip-processor",
    message -> processVIPOrder(message.getPayload()),
    message -> "VIP".equals(message.getHeaders().get("customerTier"))
);

// Regular customer processor
orderGroup.addConsumer("regular-processor",
    message -> processRegularOrder(message.getPayload()),
    message -> !"VIP".equals(message.getHeaders().get("customerTier"))
);
```

Monitoring Consumer Groups

Consumer groups provide comprehensive monitoring capabilities:

```
// Get consumer group statistics
ConsumerGroupStats stats = orderGroup.getStats();
System.out.println("Active consumers: " + stats.getActiveConsumerCount());
System.out.println("Messages processed: " + stats.getTotalMessagesProcessed());
System.out.println("Average processing time: " + stats.getAverageProcessingTime());
```

```
// Monitor individual consumers
for (String consumerId : orderGroup.getConsumerIds()) {
    ConsumerStats consumerStats = orderGroup.getConsumerStats(consumerId);
    System.out.printf("Consumer %s: %d messages, %.2fms avg%n",
        consumerId,
        consumerStats.getMessagesProcessed(),
        consumerStats.getAverageProcessingTime());
}
```

Level 2: Business Scenarios

These examples demonstrate real-world business use cases with practical implementations.

Example 1: E-commerce Order Processing

A complete order processing system with transactional guarantees:

```
public class ECommerceOrderSystem {
    public static void main(String[] args) throws Exception {
        try (PeeGeeQManager manager = new PeeGeeQManager()) {
            manager.start();

            // Use outbox pattern for transactional consistency
            QueueFactoryProvider provider = QueueFactoryProvider.getInstance();
            QueueFactory factory = provider.createFactory("outbox", manager.getDatabaseService());

            // Setup multiple event streams
            try (MessageProducer<OrderEvent> orderProducer =
                     factory.createProducer("order-events", OrderEvent.class);
                 MessageProducer<InventoryEvent> inventoryProducer =
                     factory.createProducer("inventory-events", InventoryEvent.class);
                 MessageProducer<NotificationEvent> notificationProducer =
                     factory.createProducer("notification-events", NotificationEvent.class);
                 MessageProducer<PaymentEvent> paymentProducer =
                     factory.createProducer("payment-events", PaymentEvent.class)) {

                // Setup event processors
                setupOrderProcessor(factory);
                setupInventoryProcessor(factory);
                setupNotificationProcessor(factory);
                setupPaymentProcessor(factory);

                // Simulate order processing workflow
                processCompleteOrder(manager, orderProducer, inventoryProducer,
                    notificationProducer, paymentProducer);

                Thread.sleep(5000); // Let all events process
            }
        }
    }

    private static void processCompleteOrder(
            PeeGeeQManager manager,
            MessageProducer<OrderEvent> orderProducer,
            MessageProducer<InventoryEvent> inventoryProducer,
            MessageProducer<NotificationEvent> notificationProducer,
            MessageProducer<PaymentEvent> paymentProducer) throws Exception {

        String orderId = "ORDER-" + System.currentTimeMillis();
        String customerId = "CUST-12345";

        // Step 1: Create order with inventory reservation (transactional)
        try (Connection conn = manager.getDatabaseService().getConnection()) {
            conn.setAutoCommit(false);

            try {
                // Create order record
                insertOrder(conn, orderId, customerId, new BigDecimal("299.99"));

                // Reserve inventory
                reserveInventory(conn, "PRODUCT-ABC", 2);

                // Send events within same transaction
                orderProducer.send(new OrderEvent(orderId, "CREATED", customerId,
                    List.of(new OrderItem("PRODUCT-ABC", 2, new BigDecimal("149.99"))))).join();

                inventoryProducer.send(new InventoryEvent("PRODUCT-ABC", "RESERVED", 2)).join();

                notificationProducer.send(new NotificationEvent(customerId,
                    "ORDER_CREATED", "Your order " + orderId + " has been created")).join();

                conn.commit();
                System.out.println("✅ Order created and inventory reserved");

            } catch (Exception e) {
                conn.rollback();
                throw e;
            }
        }

        // Additional steps would continue here...
    }

    // Helper methods and event classes would be defined here...
}
```

🎯 **Try This Now**:

1. Create the event classes (OrderEvent, InventoryEvent, etc.)
2. Run the example and observe transactional consistency
3. Try introducing failures to see rollback behavior

Level 3: Advanced Integration

These examples demonstrate complex enterprise integration scenarios, microservices patterns, and sophisticated messaging architectures that you'll encounter in large-scale production systems.

Example 1: Microservices Saga Pattern

The Saga pattern coordinates distributed transactions across multiple microservices using compensating actions. This example shows how to implement a distributed order processing saga with PeeGeeQ.

```
public class OrderProcessingSaga {
    private final PeeGeeQManager manager;
    private final QueueFactory factory;
    private final SagaOrchestrator orchestrator;
```

```java
public static void main(String[] args) throws Exception {
    try (PeeGeeQManager manager = new PeeGeeQManager()) {
        manager.start();

        QueueFactoryProvider provider = new PgQueueFactoryProvider();
        QueueFactory factory = provider.createFactory("outbox",
            new PgDatabaseService(manager));

        OrderProcessingSaga saga = new OrderProcessingSaga(manager, factory);
        saga.runSagaExample();
    }
}

public OrderProcessingSaga(PeeGeeQManager manager, QueueFactory factory) {
    this.manager = manager;
    this.factory = factory;
    this.orchestrator = new SagaOrchestrator(factory);
}

public void runSagaExample() throws Exception {
    System.out.println("=== Microservices Saga Pattern Example ===");

    // Setup saga participants
    setupSagaParticipants();

    // Process successful order
    processOrderSaga("ORDER-001", true);
    Thread.sleep(2000);

    // Process order that fails at payment
    processOrderSaga("ORDER-002", false);
    Thread.sleep(3000);

    System.out.println("Saga pattern example completed!");
}

private void setupSagaParticipants() throws Exception {
    // Inventory Service
    MessageConsumer<SagaCommand> inventoryConsumer =
        factory.createConsumer("inventory-commands", SagaCommand.class);
    inventoryConsumer.subscribe(this::handleInventoryCommand);

    // Payment Service
    MessageConsumer<SagaCommand> paymentConsumer =
        factory.createConsumer("payment-commands", SagaCommand.class);
    paymentConsumer.subscribe(this::handlePaymentCommand);

    // Shipping Service
    MessageConsumer<SagaCommand> shippingConsumer =
        factory.createConsumer("shipping-commands", SagaCommand.class);
    shippingConsumer.subscribe(this::handleShippingCommand);

    // Saga Coordinator
    MessageConsumer<SagaEvent> coordinatorConsumer =
        factory.createConsumer("saga-events", SagaEvent.class);
    coordinatorConsumer.subscribe(orchestrator::handleSagaEvent);
}

private void processOrderSaga(String orderId, boolean shouldSucceed) {
    SagaTransaction saga = SagaTransaction.builder()
        .sagaId("SAGA-" + orderId)
        .orderId(orderId)
        .addStep("RESERVE_INVENTORY", "inventory-commands", "RELEASE_INVENTORY")
        .addStep("PROCESS_PAYMENT", "payment-commands", "REFUND_PAYMENT")
        .addStep("ARRANGE_SHIPPING", "shipping-commands", "CANCEL_SHIPPING")
        .build();
    if (!shouldSucceed) {
        saga.setFailAtStep("PROCESS_PAYMENT");
    }

    orchestrator.startSaga(saga);
}

private CompletableFuture<Void> handleInventoryCommand(Message<SagaCommand> message) {
    SagaCommand command = message.getPayload();
    System.out.printf("📦 Inventory Service: %s for order %s%n",
        command.getAction(), command.getOrderId());

    // Simulate inventory processing
    try {
        Thread.sleep(500);

        if ("RESERVE_INVENTORY".equals(command.getAction())) {
            // Always succeed for demo
            orchestrator.reportSuccess(command.getSagaId(), "RESERVE_INVENTORY",
                Map.of("reservationId", "RES-" + command.getOrderId()));
        } else if ("RELEASE_INVENTORY".equals(command.getAction())) {
            // Compensating action
            orchestrator.reportCompensationComplete(command.getSagaId(), "RESERVE_INVENTORY");
        }

    } catch (Exception e) {
        orchestrator.reportFailure(command.getSagaId(), "RESERVE_INVENTORY", e.getMessage());
    }

    return CompletableFuture.completedFuture(null);
}

private CompletableFuture<Void> handlePaymentCommand(Message<SagaCommand> message) {
    SagaCommand command = message.getPayload();
    System.out.printf("💳 Payment Service: %s for order %s%n",
        command.getAction(), command.getOrderId());

    try {
        Thread.sleep(800);

        if ("PROCESS_PAYMENT".equals(command.getAction())) {
            // Simulate payment failure for ORDER-002
            if (command.getOrderId().equals("ORDER-002")) {
                orchestrator.reportFailure(command.getSagaId(), "PROCESS_PAYMENT",
                    "Insufficient funds");
            } else {
                orchestrator.reportSuccess(command.getSagaId(), "PROCESS_PAYMENT",
                    Map.of("transactionId", "TXN-" + command.getOrderId()));
            }
        } else if ("REFUND_PAYMENT".equals(command.getAction())) {
            // Compensating action
            orchestrator.reportCompensationComplete(command.getSagaId(), "PROCESS_PAYMENT");
        }

    } catch (Exception e) {
        orchestrator.reportFailure(command.getSagaId(), "PROCESS_PAYMENT", e.getMessage());
    }

    return CompletableFuture.completedFuture(null);
}

private CompletableFuture<Void> handleShippingCommand(Message<SagaCommand> message) {
    SagaCommand command = message.getPayload();
    System.out.printf("🚚 Shipping Service: %s for order %s%n",
        command.getAction(), command.getOrderId());
    try {
        Thread.sleep(600);

        if ("ARRANGE_SHIPPING".equals(command.getAction())) {
            orchestrator.reportSuccess(command.getSagaId(), "ARRANGE_SHIPPING",
                Map.of("trackingNumber", "TRACK-" + command.getOrderId()));
        } else if ("CANCEL_SHIPPING".equals(command.getAction())) {
            // Compensating action
            orchestrator.reportCompensationComplete(command.getSagaId(), "ARRANGE_SHIPPING");
        }

    } catch (Exception e) {
        orchestrator.reportFailure(command.getSagaId(), "ARRANGE_SHIPPING", e.getMessage());
    }

    return CompletableFuture.completedFuture(null);
}
}
```

Example 2: Event-Driven Architecture with CQRS

This example demonstrates Command Query Responsibility Segregation (CQRS) with event sourcing using PeeGeeQ's bi-temporal event store.

```
public class CQRSEventDrivenExample {
    private final EventStore<DomainEvent> eventStore;
    private final QueueFactory commandFactory;
    private final QueueFactory queryFactory;

    public static void main(String[] args) throws Exception {
        try (PeeGeeQManager manager = new PeeGeeQManager()) {
            manager.start();

            // Create event store for domain events
            BiTemporalEventStoreFactory eventStoreFactory =
                new BiTemporalEventStoreFactory(manager);
            EventStore<DomainEvent> eventStore =
                eventStoreFactory.createEventStore(DomainEvent.class);

            // Create separate factories for commands and queries
            QueueFactoryProvider provider = new PgQueueFactoryProvider();
            QueueFactory commandFactory = provider.createFactory("outbox",
                new PgDatabaseService(manager));
            QueueFactory queryFactory = provider.createFactory("native",
                new PgDatabaseService(manager));

            CQRSEventDrivenExample example = new CQRSEventDrivenExample(
                eventStore, commandFactory, queryFactory);
            example.runCQRSExample();
        }
    }

    public CQRSEventDrivenExample(EventStore<DomainEvent> eventStore,
                                  QueueFactory commandFactory,
                                  QueueFactory queryFactory) {
        this.eventStore = eventStore;
        this.commandFactory = commandFactory;
        this.queryFactory = queryFactory;
    }

    public void runCQRSExample() throws Exception {
        System.out.println("=== CQRS Event-Driven Architecture Example ===");

        // Setup command handlers
        setupCommandHandlers();

        // Setup query handlers and projections
        setupQueryHandlers();

        // Setup event handlers for projections
        setupEventHandlers();

        // Execute commands
        executeCommands();

        Thread.sleep(3000);

        // Execute queries
        executeQueries();

        System.out.println("CQRS example completed!");
    }

    private void setupCommandHandlers() throws Exception {
        MessageConsumer<CreateAccountCommand> createAccountConsumer =
            commandFactory.createConsumer("create-account-commands", CreateAccountCommand.class);
        createAccountConsumer.subscribe(this::handleCreateAccount);

        MessageConsumer<DepositCommand> depositConsumer =
            commandFactory.createConsumer("deposit-commands", DepositCommand.class);
        depositConsumer.subscribe(this::handleDeposit);

        MessageConsumer<WithdrawCommand> withdrawConsumer =
            commandFactory.createConsumer("withdraw-commands", WithdrawCommand.class);
        withdrawConsumer.subscribe(this::handleWithdraw);
    }

    private void setupQueryHandlers() throws Exception {
        MessageConsumer<AccountBalanceQuery> balanceQueryConsumer =
            queryFactory.createConsumer("balance-queries", AccountBalanceQuery.class);
        balanceQueryConsumer.subscribe(this::handleBalanceQuery);

        MessageConsumer<TransactionHistoryQuery> historyQueryConsumer =
            queryFactory.createConsumer("history-queries", TransactionHistoryQuery.class);
        historyQueryConsumer.subscribe(this::handleHistoryQuery);
    }

    private void setupEventHandlers() throws Exception {
        // Listen for domain events to update read models
        MessageConsumer<DomainEvent> eventConsumer =
            queryFactory.createConsumer("domain-events", DomainEvent.class);
        eventConsumer.subscribe(this::updateProjections);
    }

    private CompletableFuture<Void> handleCreateAccount(Message<CreateAccountCommand> message) {
        CreateAccountCommand command = message.getPayload();
        System.out.printf("📝 Command: Creating account %s%n", command.getAccountId());

        // Create domain event
        AccountCreatedEvent event = new AccountCreatedEvent(
            command.getAccountId(), command.getOwnerName(), command.getInitialBalance());

        // Store event
        return eventStore.append("AccountCreated", event, Instant.now(),
            Map.of("commandId", command.getCommandId()),
            command.getCommandId(), command.getAccountId())
            .thenCompose(storedEvent -> {
```

```
            // Publish event for projections
            return publishDomainEvent(event);
        })
        .thenRun(() -> System.out.printf("✅ Account %s created%n", command.getAccountId()));
}

private CompletableFuture<Void> handleDeposit(Message<DepositCommand> message) {
    DepositCommand command = message.getPayload();
    System.out.printf("📝 Command: Deposit $%.2f to account %s%n",
        command.getAmount(), command.getAccountId());

    // In real implementation, you'd load current state from event store
    // For demo, we'll just create the event
    MoneyDepositedEvent event = new MoneyDepositedEvent(
        command.getAccountId(), command.getAmount(), command.getDescription());

    return eventStore.append("MoneyDeposited", event, Instant.now(),
        Map.of("commandId", command.getCommandId()),
        command.getCommandId(), command.getAccountId())
        .thenCompose(storedEvent -> publishDomainEvent(event))
        .thenRun(() -> System.out.printf("✅ Deposited $%.2f to account %s%n",
            command.getAmount(), command.getAccountId()));
}

private CompletableFuture<Void> handleWithdraw(Message<WithdrawCommand> message) {
    WithdrawCommand command = message.getPayload();
    System.out.printf("📝 Command: Withdraw $%.2f from account %s%n",
        command.getAmount(), command.getAccountId());

    // In real implementation, you'd validate sufficient balance
    MoneyWithdrawnEvent event = new MoneyWithdrawnEvent(
        command.getAccountId(), command.getAmount(), command.getDescription());

    return eventStore.append("MoneyWithdrawn", event, Instant.now(),
        Map.of("commandId", command.getCommandId()),
        command.getCommandId(), command.getAccountId())
        .thenCompose(storedEvent -> publishDomainEvent(event))
        .thenRun(() -> System.out.printf("✅ Withdrew $%.2f from account %s%n",
            command.getAmount(), command.getAccountId()));
}

private CompletableFuture<Void> publishDomainEvent(DomainEvent event) {
    try {
        MessageProducer<DomainEvent> eventProducer =
            queryFactory.createProducer("domain-events", DomainEvent.class);
        return eventProducer.send(event);
    } catch (Exception e) {
        return CompletableFuture.failedFuture(e);
    }
}

private CompletableFuture<Void> updateProjections(Message<DomainEvent> message) {
    DomainEvent event = message.getPayload();
    System.out.printf("📊 Updating projections for event: %s%n", event.getEventType());

    // Update read models based on event type
    // In real implementation, you'd update database tables optimized for queries

    return CompletableFuture.completedFuture(null);
}

private CompletableFuture<Void> handleBalanceQuery(Message<AccountBalanceQuery> message) {
    AccountBalanceQuery query = message.getPayload();
    System.out.printf("🔍 Query: Balance for account %s%n", query.getAccountId());

    // In real implementation, you'd query the read model
```

```
        System.out.printf("💰 Account %s balance: $%.2f%n",
            query.getAccountId(), 1500.00); // Mock balance

        return CompletableFuture.completedFuture(null);
    }

    private CompletableFuture<Void> handleHistoryQuery(Message<TransactionHistoryQuery> message) {
        TransactionHistoryQuery query = message.getPayload();
        System.out.printf("🔍 Query: Transaction history for account %s%n", query.getAccountId());

        // In real implementation, you'd query the event store or read model
        System.out.printf("📋 Account %s has 5 transactions in the last 30 days%n",
            query.getAccountId());

        return CompletableFuture.completedFuture(null);
    }

    private void executeCommands() throws Exception {
        MessageProducer<CreateAccountCommand> createProducer =
            commandFactory.createProducer("create-account-commands", CreateAccountCommand.class);
        MessageProducer<DepositCommand> depositProducer =
            commandFactory.createProducer("deposit-commands", DepositCommand.class);
        MessageProducer<WithdrawCommand> withdrawProducer =
            commandFactory.createProducer("withdraw-commands", WithdrawCommand.class);

        // Execute commands
        createProducer.send(new CreateAccountCommand("ACC-001", "John Doe", new BigDecimal("1000.00")));
        Thread.sleep(500);

        depositProducer.send(new DepositCommand("ACC-001", new BigDecimal("500.00"), "Salary deposit"));
        Thread.sleep(500);

        withdrawProducer.send(new WithdrawCommand("ACC-001", new BigDecimal("200.00"), "ATM withdrawal"));
    }

    private void executeQueries() throws Exception {
        MessageProducer<AccountBalanceQuery> balanceProducer =
            queryFactory.createProducer("balance-queries", AccountBalanceQuery.class);
        MessageProducer<TransactionHistoryQuery> historyProducer =
            queryFactory.createProducer("history-queries", TransactionHistoryQuery.class);

        // Execute queries
        balanceProducer.send(new AccountBalanceQuery("ACC-001"));
        Thread.sleep(200);

        historyProducer.send(new TransactionHistoryQuery("ACC-001", 30));
    }
}
```

🎯 **Try This Now**:

1. Implement the command, event, and query classes
2. Run the saga pattern example and observe compensation behavior
3. Run the CQRS example and see command/query separation
4. Try introducing failures to see how sagas handle rollbacks

Example 3: Multi-Tenant Message Routing

This example demonstrates sophisticated message routing in a multi-tenant SaaS application where messages must be isolated by tenant and routed based on subscription levels.

```java
public class MultiTenantRoutingExample {
    private final QueueFactory factory;
    private final TenantRoutingService routingService;

    public static void main(String[] args) throws Exception {
        try (PeeGeeQManager manager = new PeeGeeQManager()) {
            manager.start();

            QueueFactoryProvider provider = new PgQueueFactoryProvider();
            QueueFactory factory = provider.createFactory("native",
                new PgDatabaseService(manager));

            MultiTenantRoutingExample example = new MultiTenantRoutingExample(factory);
            example.runMultiTenantExample();
        }
    }

    public MultiTenantRoutingExample(QueueFactory factory) {
        this.factory = factory;
        this.routingService = new TenantRoutingService();
    }

    public void runMultiTenantExample() throws Exception {
        System.out.println("=== Multi-Tenant Message Routing Example ===");

        // Setup tenant-specific consumers
        setupTenantConsumers();

        // Setup feature-based routing
        setupFeatureRouting();

        // Send messages for different tenants and subscription levels
        sendTenantMessages();

        Thread.sleep(3000);
        System.out.println("Multi-tenant routing example completed!");
    }

    private void setupTenantConsumers() throws Exception {
        // Premium tenant consumers (high priority processing)
        MessageConsumer<TenantMessage> premiumConsumer =
            factory.createConsumer("premium-tenant-messages", TenantMessage.class);
        premiumConsumer.subscribe(message -> {
            TenantMessage msg = message.getPayload();
            System.out.printf("🌟 Premium Processing: Tenant %s - %s%n",
                msg.getTenantId(), msg.getContent());

            // Premium tenants get enhanced processing
            return processWithPremiumFeatures(msg);
        });

        // Standard tenant consumers
        MessageConsumer<TenantMessage> standardConsumer =
            factory.createConsumer("standard-tenant-messages", TenantMessage.class);
        standardConsumer.subscribe(message -> {
            TenantMessage msg = message.getPayload();
            System.out.printf("📋 Standard Processing: Tenant %s - %s%n",
                msg.getTenantId(), msg.getContent());

            return processWithStandardFeatures(msg);
        });

        // Basic tenant consumers (rate limited)
        MessageConsumer<TenantMessage> basicConsumer =
            factory.createConsumer("basic-tenant-messages", TenantMessage.class);
        basicConsumer.subscribe(message -> {
            TenantMessage msg = message.getPayload();
            System.out.printf("📝 Basic Processing: Tenant %s - %s%n",
                msg.getTenantId(), msg.getContent());

            // Add rate limiting for basic tier
            return processWithRateLimit(msg);
        });
    }

    private void setupFeatureRouting() throws Exception {
        // Analytics feature (premium only)
        MessageConsumer<AnalyticsEvent> analyticsConsumer =
            factory.createConsumer("analytics-events", AnalyticsEvent.class);
        analyticsConsumer.subscribe(this::processAnalytics);

        // Advanced reporting (standard and premium)
        MessageConsumer<ReportRequest> reportConsumer =
            factory.createConsumer("report-requests", ReportRequest.class);
        reportConsumer.subscribe(this::processReports);

        // Notification routing based on tenant preferences
        MessageConsumer<NotificationEvent> notificationConsumer =
            factory.createConsumer("tenant-notifications", NotificationEvent.class);
        notificationConsumer.subscribe(this::routeNotifications);
    }

    private void sendTenantMessages() throws Exception {
        MessageProducer<TenantMessage> messageRouter =
            factory.createProducer("tenant-message-router", TenantMessage.class);

        // Messages from different tenant tiers
        TenantMessage premiumMsg = new TenantMessage("TENANT-PREMIUM-001",
            "Process premium order", TenantTier.PREMIUM);
        TenantMessage standardMsg = new TenantMessage("TENANT-STD-002",
            "Process standard order", TenantTier.STANDARD);
        TenantMessage basicMsg = new TenantMessage("TENANT-BASIC-003",
            "Process basic order", TenantTier.BASIC);

        // Route messages based on tenant tier
        routeMessage(messageRouter, premiumMsg);
        routeMessage(messageRouter, standardMsg);
        routeMessage(messageRouter, basicMsg);

        // Send feature-specific messages
        sendFeatureMessages();
    }

    private void routeMessage(MessageProducer<TenantMessage> router, TenantMessage message)
            throws Exception {
        String targetQueue = routingService.determineQueue(message.getTenantTier());

        // Create headers for routing
        Map<String, String> headers = Map.of(
            "tenantId", message.getTenantId(),
            "tier", message.getTenantTier().name(),
            "targetQueue", targetQueue
        );

        // Send with routing headers
        router.send(message, headers).join();

        // Route to appropriate queue based on tier
        MessageProducer<TenantMessage> targetProducer =
            factory.createProducer(targetQueue, TenantMessage.class);
        targetProducer.send(message).join();
    }

    private void sendFeatureMessages() throws Exception {
        // Analytics events (premium only)
        MessageProducer<AnalyticsEvent> analyticsProducer =
            factory.createProducer("analytics-events", AnalyticsEvent.class);
        analyticsProducer.send(new AnalyticsEvent("TENANT-PREMIUM-001",
            "user_action", Map.of("action", "purchase", "amount", "99.99")));

        // Report requests (standard and premium)
        MessageProducer<ReportRequest> reportProducer =
            factory.createProducer("report-requests", ReportRequest.class);
        reportProducer.send(new ReportRequest("TENANT-STD-002", "monthly_sales",
            Map.of("month", "2025-01")));

        // Notifications for all tiers
        MessageProducer<NotificationEvent> notificationProducer =
            factory.createProducer("tenant-notifications", NotificationEvent.class);
        notificationProducer.send(new NotificationEvent("TENANT-BASIC-003",
            "Your report is ready", NotificationChannel.EMAIL));
    }

    private CompletableFuture<Void> processWithPremiumFeatures(TenantMessage message) {
        // Premium processing includes advanced analytics, priority support, etc.
        System.out.printf("  🚀 Enhanced processing for %s%n", message.getTenantId());
        return CompletableFuture.completedFuture(null);
    }

    private CompletableFuture<Void> processWithStandardFeatures(TenantMessage message) {
        // Standard processing with basic features
        System.out.printf("  ⚡ Standard processing for %s%n", message.getTenantId());
        return CompletableFuture.completedFuture(null);
    }

    private CompletableFuture<Void> processWithRateLimit(TenantMessage message) {
        // Basic processing with rate limiting
        System.out.printf("  🐌 Rate-limited processing for %s%n", message.getTenantId());

        // Simulate rate limiting
        try {
            Thread.sleep(1000); // Slower processing for basic tier
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        return CompletableFuture.completedFuture(null);
    }

    private CompletableFuture<Void> processAnalytics(Message<AnalyticsEvent> message) {
        AnalyticsEvent event = message.getPayload();

        // Verify tenant has analytics feature
        if (!routingService.hasAnalyticsFeature(event.getTenantId())) {
            System.out.printf("❌ Analytics denied for tenant %s (not premium)%n",
                event.getTenantId());
            return CompletableFuture.completedFuture(null);
        }

        System.out.printf("📊 Processing analytics: %s for tenant %s%n",
            event.getEventType(), event.getTenantId());

        return CompletableFuture.completedFuture(null);
    }

    private CompletableFuture<Void> processReports(Message<ReportRequest> message) {
        ReportRequest request = message.getPayload();
        // Verify tenant has reporting feature
        if (!routingService.hasReportingFeature(request.getTenantId())) {
            System.out.printf("❌ Reporting denied for tenant %s (basic tier)%n",
                request.getTenantId());
            return CompletableFuture.completedFuture(null);
        }

        System.out.printf("📈 Generating report: %s for tenant %s%n",
            request.getReportType(), request.getTenantId());

        return CompletableFuture.completedFuture(null);
    }

    private CompletableFuture<Void> routeNotifications(Message<NotificationEvent> message) {
        NotificationEvent notification = message.getPayload();

        // Route based on tenant preferences and tier
        NotificationChannel channel = routingService.getPreferredChannel(
            notification.getTenantId(), notification.getChannel());

        System.out.printf("🔔 Routing notification to %s via %s for tenant %s%n",
            channel, notification.getChannel(), notification.getTenantId());

        return CompletableFuture.completedFuture(null);
    }
}

// Supporting classes
class TenantRoutingService {
    private final Map<String, TenantTier> tenantTiers = Map.of(
        "TENANT-PREMIUM-001", TenantTier.PREMIUM,
        "TENANT-STD-002", TenantTier.STANDARD,
        "TENANT-BASIC-003", TenantTier.BASIC
    );

    public String determineQueue(TenantTier tier) {
        return switch (tier) {
            case PREMIUM -> "premium-tenant-messages";
            case STANDARD -> "standard-tenant-messages";
            case BASIC -> "basic-tenant-messages";
        };
    }

    public boolean hasAnalyticsFeature(String tenantId) {
        return tenantTiers.get(tenantId) == TenantTier.PREMIUM;
    }

    public boolean hasReportingFeature(String tenantId) {
        TenantTier tier = tenantTiers.get(tenantId);
        return tier == TenantTier.PREMIUM || tier == TenantTier.STANDARD;
    }

    public NotificationChannel getPreferredChannel(String tenantId, NotificationChannel requested) {
        TenantTier tier = tenantTiers.get(tenantId);

        // Basic tier only gets email notifications
        if (tier == TenantTier.BASIC) {
            return NotificationChannel.EMAIL;
        }

        return requested; // Premium and standard get their preferred channel
    }
}

enum TenantTier { BASIC, STANDARD, PREMIUM }
enum NotificationChannel { EMAIL, SMS, PUSH, SLACK }
```

Example 4: Distributed Cache Invalidation

This example shows how to implement distributed cache invalidation across multiple application instances using PeeGeeQ's native queue for real-time coordination.

```java
public class DistributedCacheInvalidationExample {
    private final QueueFactory factory;
    private final LocalCache localCache;
    private final String instanceId;

    public static void main(String[] args) throws Exception {
        try (PeeGeeQManager manager = new PeeGeeQManager()) {
            manager.start();

            QueueFactoryProvider provider = new PgQueueFactoryProvider();
            QueueFactory factory = provider.createFactory("native",
                new PgDatabaseService(manager));

            // Simulate multiple application instances
            DistributedCacheInvalidationExample instance1 =
                new DistributedCacheInvalidationExample(factory, "APP-INSTANCE-1");
            DistributedCacheInvalidationExample instance2 =
                new DistributedCacheInvalidationExample(factory, "APP-INSTANCE-2");
            DistributedCacheInvalidationExample instance3 =
                new DistributedCacheInvalidationExample(factory, "APP-INSTANCE-3");

            // Start cache invalidation listeners
            instance1.startCacheInvalidationListener();
            instance2.startCacheInvalidationListener();
            instance3.startCacheInvalidationListener();

            // Simulate cache operations
            instance1.runCacheInvalidationDemo();

            Thread.sleep(3000);
            System.out.println("Distributed cache invalidation example completed!");
        }
    }

    public DistributedCacheInvalidationExample(QueueFactory factory, String instanceId) {
        this.factory = factory;
        this.instanceId = instanceId;
        this.localCache = new LocalCache(instanceId);
    }

    public void startCacheInvalidationListener() throws Exception {
        MessageConsumer<CacheInvalidationEvent> consumer =
            factory.createConsumer("cache-invalidation", CacheInvalidationEvent.class);

        consumer.subscribe(this::handleCacheInvalidation);
        System.out.printf("🎧 %s: Started cache invalidation listener%n", instanceId);
    }

    public void runCacheInvalidationDemo() throws Exception {
        System.out.println("=== Distributed Cache Invalidation Example ===");

        MessageProducer<CacheInvalidationEvent> invalidationProducer =
            factory.createProducer("cache-invalidation", CacheInvalidationEvent.class);

        // Populate caches across all instances
        populateInitialCache();

        // Simulate data updates that require cache invalidation
        System.out.println("\n📝 Updating user data - invalidating user caches...");
        invalidationProducer.send(new CacheInvalidationEvent(
            "user", "user:12345", CacheInvalidationType.SINGLE_KEY, instanceId));

        Thread.sleep(500);

        System.out.println("\n📝 Updating product catalog - invalidating product caches...");
        invalidationProducer.send(new CacheInvalidationEvent(
            "product", "product:*", CacheInvalidationType.PATTERN, instanceId));

        Thread.sleep(500);

        System.out.println("\n📝 System maintenance - clearing all caches...");
        invalidationProducer.send(new CacheInvalidationEvent(
            "*", "*", CacheInvalidationType.CLEAR_ALL, instanceId));
    }

    private void populateInitialCache() {
        System.out.printf("📦 %s: Populating initial cache data%n", instanceId);

        // Simulate populating cache with user data
        localCache.put("user:12345", "John Doe");
        localCache.put("user:67890", "Jane Smith");

        // Simulate populating cache with product data
        localCache.put("product:ABC123", "Laptop Computer");
        localCache.put("product:XYZ789", "Wireless Mouse");

        // Simulate populating cache with session data
        localCache.put("session:sess001", "active_session_data");

        System.out.printf("📊 %s: Cache populated with %d items%n",
            instanceId, localCache.size());
    }

    private CompletableFuture<Void> handleCacheInvalidation(Message<CacheInvalidationEvent> message) {
        CacheInvalidationEvent event = message.getPayload();

        // Don't process invalidation events from this instance
        if (instanceId.equals(event.getOriginatingInstance())) {
            return CompletableFuture.completedFuture(null);
        }

        System.out.printf("🗑️  %s: Received cache invalidation - %s:%s (type: %s)%n",
            instanceId, event.getCacheRegion(), event.getKey(), event.getType());

        int itemsInvalidated = 0;

        switch (event.getType()) {
            case SINGLE_KEY:
                if (localCache.remove(event.getKey()) != null) {
                    itemsInvalidated = 1;
                }
                break;

            case PATTERN:
                itemsInvalidated = localCache.removeByPattern(event.getKey());
                break;

            case REGION:
                itemsInvalidated = localCache.removeByRegion(event.getCacheRegion());
                break;
            case CLEAR_ALL:
                itemsInvalidated = localCache.size();
                localCache.clear();
                break;
        }

        System.out.printf("✅ %s: Invalidated %d cache items, remaining: %d%n",
            instanceId, itemsInvalidated, localCache.size());

        return CompletableFuture.completedFuture(null);
    }
}

// Supporting classes
class LocalCache {
    private final Map<String, Object> cache = new ConcurrentHashMap<>();
    private final String instanceId;

    public LocalCache(String instanceId) {
        this.instanceId = instanceId;
    }

    public void put(String key, Object value) {
        cache.put(key, value);
    }

    public Object get(String key) {
        return cache.get(key);
    }

    public Object remove(String key) {
        return cache.remove(key);
    }

    public int removeByPattern(String pattern) {
        String regex = pattern.replace("*", ".*");
        Pattern compiledPattern = Pattern.compile(regex);

        List<String> keysToRemove = cache.keySet().stream()
            .filter(key -> compiledPattern.matcher(key).matches())
            .collect(Collectors.toList());

        keysToRemove.forEach(cache::remove);
        return keysToRemove.size();
    }

    public int removeByRegion(String region) {
        if ("*".equals(region)) {
            int size = cache.size();
            cache.clear();
            return size;
        }

        String prefix = region + ":";
        List<String> keysToRemove = cache.keySet().stream()
            .filter(key -> key.startsWith(prefix))
            .collect(Collectors.toList());

        keysToRemove.forEach(cache::remove);
        return keysToRemove.size();
    }

    public void clear() {
        cache.clear();
    }
    public int size() {
        return cache.size();
    }
}

class CacheInvalidationEvent {
    private String cacheRegion;
    private String key;
    private CacheInvalidationType type;
    private String originatingInstance;
    private Instant timestamp;

    public CacheInvalidationEvent(String cacheRegion, String key,
                                  CacheInvalidationType type, String originatingInstance) {
        this.cacheRegion = cacheRegion;
        this.key = key;
        this.type = type;
        this.originatingInstance = originatingInstance;
        this.timestamp = Instant.now();
    }

    // Getters and setters...
}

enum CacheInvalidationType {
    SINGLE_KEY,    // Invalidate specific key
    PATTERN,       // Invalidate keys matching pattern
    REGION,        // Invalidate all keys in cache region
    CLEAR_ALL      // Clear entire cache
}
```

🎯 **Try This Now**:

1. Implement the supporting classes (TenantMessage, AnalyticsEvent, etc.)
2. Run the multi-tenant routing example with different tenant tiers
3. Run the cache invalidation example with multiple instances
4. Observe how messages are routed and processed differently based on context

Installation & Setup

Database Setup

1. **Create Database**:

```sql
CREATE DATABASE peegeeq;
CREATE USER peegeeq_user WITH PASSWORD 'your_password';
GRANT ALL PRIVILEGES ON DATABASE peegeeq TO peegeeq_user;
```

2. **Initialize Schema**:

```java
// Using PeeGeeQManager
PeeGeeQConfiguration config = PeeGeeQConfiguration.builder()
    .host("localhost")
    .port(5432)
    .database("peegeeq")
    .username("peegeeq_user")
    .password("your_password")
    .build();

PeeGeeQManager manager = new PeeGeeQManager(config);
manager.initialize(); // Creates tables and applies migrations
```

Configuration

Create `peegeeq.properties` :

```
# Database connection
peegeeq.database.host=localhost
peegeeq.database.port=5432
peegeeq.database.name=peegeeq
peegeeq.database.username=peegeeq_user
peegeeq.database.password=your_password

# Connection pool
peegeeq.database.pool.maxSize=20
peegeeq.database.pool.minSize=5

# Queue settings
peegeeq.queue.visibilityTimeoutSeconds=30
peegeeq.queue.maxRetries=3
peegeeq.queue.deadLetterEnabled=true

# Health checks
peegeeq.health.enabled=true
peegeeq.health.intervalSeconds=30

# Metrics
peegeeq.metrics.enabled=true
peegeeq.metrics.jvm.enabled=true
```

Basic Usage Examples

Simple Producer/Consumer

```
public class BasicExample {
    public static void main(String[] args) throws Exception {
        // Initialize PeeGeeQ
        PeeGeeQConfiguration config = PeeGeeQConfiguration.fromProperties("peegeeq.properties");
        PeeGeeQManager manager = new PeeGeeQManager(config);
        manager.initialize();

        DatabaseService databaseService = manager.getDatabaseService();
        QueueFactoryProvider provider = QueueFactoryProvider.getInstance();

        // Create native queue factory
        QueueFactory factory = provider.createFactory("native", databaseService);

        // Create producer and consumer
        MessageProducer<String> producer = factory.createProducer("notifications", String.class);
        MessageConsumer<String> consumer = factory.createConsumer("notifications", String.class);

        // Start consuming messages
        consumer.subscribe(message -> {
```

```java
            System.out.println("Received: " + message.getPayload());
            return CompletableFuture.completedFuture(null);
        });

        // Send messages
        producer.send("Hello, PeeGeeQ!").join();
        producer.send("Message processing is working!").join();

        // Keep running
        Thread.sleep(5000);

        // Cleanup
        producer.close();
        consumer.close();
        manager.close();
    }
}
```

Transactional Outbox Example

```java
public class TransactionalExample {
    public static void main(String[] args) throws Exception {
        // Setup (same as above)
        PeeGeeQManager manager = new PeeGeeQManager(config);
        manager.initialize();

        // Create outbox factory for transactional guarantees
        QueueFactory outboxFactory = provider.createFactory("outbox", databaseService);
        MessageProducer<OrderEvent> producer = outboxFactory.createProducer("orders", OrderEvent.class);

        // Simulate order processing with transactional messaging
        try (Connection conn = dataSource.getConnection()) {
            conn.setAutoCommit(false);

            try {
                // 1. Save order to database
                PreparedStatement stmt = conn.prepareStatement(
                    "INSERT INTO orders (id, customer_id, amount) VALUES (?, ?, ?)");
                stmt.setString(1, "ORDER-001");
                stmt.setString(2, "CUST-123");
                stmt.setBigDecimal(3, new BigDecimal("99.99"));
                stmt.executeUpdate();

                // 2. Send order event (within same transaction)
                OrderEvent event = new OrderEvent("ORDER-001", "CUST-123", new BigDecimal("99.99"));
                producer.send(event).join();

                // 3. Commit both operations together
                conn.commit();
                System.out.println("Order and event committed together!");

            } catch (Exception e) {
                conn.rollback();
                throw e;
            }
        }
    }
}
```

Part VI: Production Readiness

Configuration Management

Environment-Specific Configuration

Create different configuration files for each environment:

`peegeeq-dev.properties`:

```
# Development environment
peegeeq.database.host=localhost
peegeeq.database.port=5432
peegeeq.database.name=peegeeq_dev
peegeeq.database.username=dev_user
peegeeq.database.password=dev_password

# Relaxed settings for development
peegeeq.queue.visibilityTimeoutSeconds=30
peegeeq.queue.maxRetries=3
peegeeq.health.intervalSeconds=60
```

`peegeeq-prod.properties`:

```
# Production environment
peegeeq.database.host=${DB_HOST}
peegeeq.database.port=${DB_PORT:5432}
peegeeq.database.name=${DB_NAME}
peegeeq.database.username=${DB_USERNAME}
peegeeq.database.password=${DB_PASSWORD}

# Production-optimized settings
peegeeq.database.pool.maxSize=20
peegeeq.database.pool.minSize=5
peegeeq.queue.visibilityTimeoutSeconds=300
peegeeq.queue.maxRetries=5
peegeeq.health.intervalSeconds=30

# Security settings
peegeeq.database.ssl.enabled=true
peegeeq.database.ssl.mode=require
```

Programmatic Configuration

```java
public class ProductionPeeGeeQSetup {
    public static PeeGeeQManager createProductionManager() {
        PeeGeeQConfiguration config = PeeGeeQConfiguration.builder()
            .host(System.getenv("DB_HOST"))
            .port(Integer.parseInt(System.getenv("DB_PORT")))
            .database(System.getenv("DB_NAME"))
            .username(System.getenv("DB_USERNAME"))
            .password(System.getenv("DB_PASSWORD"))
            .poolMaxSize(20)
            .poolMinSize(5)
            .sslEnabled(true)
            .healthCheckInterval(Duration.ofSeconds(30))
            .build();

        return new PeeGeeQManager(config);
    }
}
```

Monitoring & Metrics

Built-in Health Checks

```
public class HealthCheckExample {
    public static void main(String[] args) throws Exception {
        try (PeeGeeQManager manager = new PeeGeeQManager()) {
            manager.start();

            // Get health status
            HealthStatus health = manager.getHealthStatus();
            System.out.printf("Database Health: %s%n", health.getDatabaseStatus());
            System.out.printf("Queue Health: %s%n", health.getQueueStatus());
            System.out.printf("Last Check: %s%n", health.getLastCheckTime());

            // Get performance metrics
            PerformanceMetrics metrics = manager.getMetrics();
            System.out.printf("Messages Sent: %d%n", metrics.getMessagesSent());
            System.out.printf("Messages Processed: %d%n", metrics.getMessagesProcessed());
            System.out.printf("Average Latency: %.2fms%n", metrics.getAverageLatencyMs());
            System.out.printf("Error Rate: %.2f%%%n", metrics.getErrorRate() * 100);
        }
    }
}
```

Custom Metrics Integration

```
// Integration with Micrometer/Prometheus
public class MetricsIntegration {
    private final MeterRegistry meterRegistry;
    private final Counter messagesSentCounter;
    private final Timer processingTimer;

    public MetricsIntegration(MeterRegistry meterRegistry) {
        this.meterRegistry = meterRegistry;
        this.messagesSentCounter = Counter.builder("peegeeq.messages.sent")
            .description("Total messages sent")
            .register(meterRegistry);
        this.processingTimer = Timer.builder("peegeeq.processing.time")
            .description("Message processing time")
            .register(meterRegistry);
    }

    public void setupMetrics(MessageConsumer<?> consumer) {
        consumer.subscribe(message -> {
            Timer.Sample sample = Timer.start(meterRegistry);

            return processMessage(message)
                .whenComplete((result, throwable) -> {
                    sample.stop(processingTimer);
                    if (throwable == null) {
                        messagesSentCounter.increment();
                    }
                });
            });
        }
    }
```

Performance Tuning

Performance Characteristics Overview

PeeGeeQ delivers enterprise-grade performance across all messaging patterns:

Native Queue Performance

- **Throughput**: 10,000+ messages/second
- **Latency**: <10ms end-to-end
- **Mechanism**: PostgreSQL LISTEN/NOTIFY with advisory locks
- **Concurrency**: Multiple consumers with automatic load balancing
- **Scalability**: Horizontal scaling via consumer groups
- **Memory Usage**: Low memory footprint with streaming processing
- **Connection Efficiency**: Connection pooling with optimized pool sizes

Outbox Pattern Performance

- **Throughput**: 5,000+ messages/second
- **Latency**: ~100ms (polling-based with configurable intervals)
- **Mechanism**: Database polling with ACID transactions
- **Consistency**: Full ACID compliance with business data
- **Reliability**: Exactly-once delivery guarantee
- **Durability**: Transactional outbox ensures no message loss
- **Retry Handling**: Configurable retry policies with exponential backoff
- **Parallel Processing**: Configurable consumer threads for high throughput

Bi-temporal Event Store Performance

- **Write Throughput**: 3,000+ events/second
- **Query Performance**: <50ms for typical temporal queries
- **Storage**: Append-only, optimized for time-series data
- **Indexing**: Multi-dimensional indexes for temporal and aggregate queries
- **Correction Support**: Efficient event correction with version tracking
- **Historical Queries**: Point-in-time queries with transaction time support
- **Aggregate Reconstruction**: Fast aggregate state reconstruction

REST API Performance

- **HTTP Throughput**: 2,000+ requests/second
- **WebSocket Throughput**: 5,000+ messages/second per connection
- **SSE Throughput**: 3,000+ events/second per connection
- **Latency**: <50ms for REST operations, <20ms for WebSocket
- **Concurrent Connections**: 1,000+ simultaneous WebSocket connections

- **Management Operations**: Sub-second response times for admin operations

Management Console Performance

- **UI Responsiveness**: <100ms for dashboard updates
- **Real-time Updates**: <500ms latency for live metrics
- **Data Visualization**: Handles 10,000+ data points in charts
- **Concurrent Users**: 50+ simultaneous admin users
- **Resource Usage**: <50MB memory footprint in browser

Connection Pool Optimization

```
PeeGeeQConfiguration config = PeeGeeQConfiguration.builder()
    // Optimize connection pool for your workload
    .poolMaxSize(20)                    // Max connections
    .poolMinSize(5)                     // Min connections
    .poolConnectionTimeout(Duration.ofSeconds(30))
    .poolIdleTimeout(Duration.ofMinutes(10))
    .poolMaxLifetime(Duration.ofMinutes(30))

    // Optimize for high throughput
    .batchSize(100)                     // Process messages in batches
    .pollInterval(Duration.ofMillis(100)) // How often to check for new messages

    // Optimize for low latency
    .enableNotifications(true)          // Use LISTEN/NOTIFY for real-time
    .notificationTimeout(Duration.ofSeconds(5))

    .build();
```

Queue-Specific Tuning

```
// High-throughput configuration
QueueConfiguration highThroughputConfig = QueueConfiguration.builder()
    .visibilityTimeout(Duration.ofMinutes(5))
    .maxRetries(3)
    .batchSize(50)
    .concurrentConsumers(10)
    .build();

// Low-latency configuration
QueueConfiguration lowLatencyConfig = QueueConfiguration.builder()
    .visibilityTimeout(Duration.ofSeconds(30))
    .maxRetries(5)
    .batchSize(1)
    .concurrentConsumers(1)
    .enableRealTimeNotifications(true)
    .build();
```

Parallel Processing Configuration

Configure parallel processing for high-throughput scenarios:

```
# Consumer thread configuration for parallel processing
peegeeq.consumer.threads=8                    # Number of parallel consumer threads
peegeeq.queue.batch-size=25                    # Messages processed per batch
peegeeq.queue.polling-interval=PT0.5S          # Polling frequency (500ms)

# Backpressure management
peegeeq.consumer.max-concurrent-operations=100 # Max concurrent operations
peegeeq.consumer.timeout=PT30S                 # Consumer operation timeout
peegeeq.consumer.queue-capacity=1000           # Internal queue capacity

# Memory and resource management
peegeeq.consumer.thread-pool-keep-alive=PT60S # Thread keep-alive time
peegeeq.consumer.enable-metrics=true           # Enable consumer metrics
```

Environment-Specific Configurations

Development Environment

```
# Optimized for development and debugging
peegeeq.consumer.threads=2
peegeeq.queue.batch-size=5
peegeeq.queue.polling-interval=PT1S
peegeeq.queue.max-retries=3
peegeeq.logging.level=DEBUG
```

Staging Environment

```
# Balanced performance for testing
peegeeq.consumer.threads=4
peegeeq.queue.batch-size=15
peegeeq.queue.polling-interval=PT0.5S
peegeeq.queue.max-retries=5
peegeeq.consumer.max-concurrent-operations=50
```

Production Environment

```
# High-performance production settings
peegeeq.consumer.threads=8
peegeeq.queue.batch-size=50
peegeeq.queue.polling-interval=PT0.1S
peegeeq.queue.max-retries=7
peegeeq.consumer.max-concurrent-operations=200
peegeeq.consumer.timeout=PT60S
peegeeq.circuitBreaker.enabled=true
peegeeq.metrics.enabled=true
```

JVM Tuning for High Performance

```
# JVM settings for high-throughput scenarios
-Xms4g -Xmx8g                           # Heap size
-XX:+UseG1GC                            # G1 garbage collector
-XX:MaxGCPauseMillis=200                # Max GC pause time
-XX:+UseStringDeduplication             # String deduplication
-XX:+UseCompressedOops                  # Compressed object pointers
-XX:NewRatio=2                          # Young/old generation ratio

# For very high throughput (adjust based on your hardware)
```

```
-XX:+UnlockExperimentalVMOptions
-XX:+UseZGC                         # ZGC for ultra-low latency
-XX:+UseLargePages                  # Large pages for better memory management
```

Security Considerations

SSL/TLS Configuration

```
# Enable SSL
peegeeq.database.ssl.enabled=true
peegeeq.database.ssl.mode=require
peegeeq.database.ssl.cert=/path/to/client-cert.pem
peegeeq.database.ssl.key=/path/to/client-key.pem
peegeeq.database.ssl.rootcert=/path/to/ca-cert.pem
```

Message Encryption

```java
public class EncryptedMessageExample {
    private final MessageProducer<EncryptedMessage> producer;
    private final MessageConsumer<EncryptedMessage> consumer;
    private final EncryptionService encryptionService;

    public void sendEncryptedMessage(String sensitiveData) {
        // Encrypt before sending
        String encryptedData = encryptionService.encrypt(sensitiveData);
        EncryptedMessage message = new EncryptedMessage(encryptedData);

        producer.send(message).join();
    }

    public void setupEncryptedConsumer() {
        consumer.subscribe(message -> {
            EncryptedMessage encryptedMessage = message.getPayload();

            // Decrypt after receiving
            String decryptedData = encryptionService.decrypt(encryptedMessage.getData());

            // Process decrypted data
            processDecryptedData(decryptedData);

            return CompletableFuture.completedFuture(null);
        });
    }
}
```

Part VII: Troubleshooting & Best Practices

Common Issues & Solutions

Issue 1: Messages Not Being Processed

Symptoms:

- Messages are sent but never consumed
- Consumer appears to be running but no processing occurs

Possible Causes & Solutions:

1. **Consumer not subscribed properly**

```java
// ❌ Wrong - consumer created but not subscribed
MessageConsumer<String> consumer = factory.createConsumer("queue", String.class);

// ✅ Correct - consumer subscribed to process messages
MessageConsumer<String> consumer = factory.createConsumer("queue", String.class);
consumer.subscribe(message -> {
    // Process message
    return CompletableFuture.completedFuture(null);
});
```

2. **Database connection issues**

```java
// Check database connectivity
try {
    HealthStatus health = manager.getHealthStatus();
    if (health.getDatabaseStatus() != HealthStatus.Status.HEALTHY) {
        System.err.println("Database connection issue: " + health.getErrorMessage());
    }
} catch (Exception e) {
    System.err.println("Cannot connect to database: " + e.getMessage());
}
```

3. **Queue name mismatch**

```java
// ❌ Wrong - different queue names
MessageProducer<String> producer = factory.createProducer("orders", String.class);
MessageConsumer<String> consumer = factory.createConsumer("order", String.class); // Missing 's'

// ✅ Correct - same queue name
MessageProducer<String> producer = factory.createProducer("orders", String.class);
MessageConsumer<String> consumer = factory.createConsumer("orders", String.class);
```

Issue 2: High Latency

Symptoms:

- Messages take a long time to be processed
- High delay between send and receive

Solutions:

1. **Use Native Queue for real-time processing**

```java
// ✅ Use native queue for low latency
QueueFactory factory = provider.createFactory("native", databaseService);
```

2. **Optimize polling interval**

```java
PeeGeeQConfiguration config = PeeGeeQConfiguration.builder()
    .pollInterval(Duration.ofMillis(50)) // Faster polling
    .build();
```

3. **Enable notifications**

```java
PeeGeeQConfiguration config = PeeGeeQConfiguration.builder()
    .enableNotifications(true) // Real-time notifications
    .build();
```

Issue 3: Memory Issues

Symptoms:

- OutOfMemoryError
- High memory usage
- Application becomes unresponsive

Solutions:

1. **Limit batch sizes**

```java
QueueConfiguration config = QueueConfiguration.builder()
    .batchSize(10) // Smaller batches
    .build();
```

2. **Process messages asynchronously**

```java
consumer.subscribe(message -> {
    // ✅ Process asynchronously to avoid blocking
    return CompletableFuture.supplyAsync(() -> {
        processMessage(message.getPayload());
        return null;
    });
});
```

3. **Implement backpressure**

```java
private final Semaphore processingLimiter = new Semaphore(100);

consumer.subscribe(message -> {
    return CompletableFuture.supplyAsync(() -> {
        try {
            processingLimiter.acquire();
            processMessage(message.getPayload());
```

```
                return null;
            } finally {
                processingLimiter.release();
            }
        });
    });
```

Best Practices Checklist

✅ Development Best Practices

- **Use try-with-resources** for automatic cleanup
- **Handle exceptions properly** in message processors
- **Use appropriate queue types** for your use case
- **Implement proper logging** for debugging
- **Write unit tests** for message processors
- **Use type-safe message classes** instead of raw strings

✅ Production Best Practices

- **Configure connection pools** appropriately
- **Enable health checks** and monitoring
- **Use environment-specific configuration**
- **Implement circuit breakers** for external dependencies
- **Set up proper alerting** for failures
- **Plan for disaster recovery**

✅ Performance Best Practices

- **Choose the right pattern** (Native vs Outbox vs Bi-temporal)
- **Batch operations** when possible
- **Use connection pooling**
- **Monitor and tune** based on actual usage
- **Implement proper indexing** on custom fields

✅ Security Best Practices

- **Use SSL/TLS** for database connections
- **Encrypt sensitive message data**
- **Use proper authentication** and authorization
- **Audit message access** and processing
- **Follow principle of least privilege**

Anti-patterns to Avoid

❌ Don't: Create New Managers for Each Operation

```
// ❌ Wrong - creates new connections repeatedly
public void sendMessage(String message) {
    try (PeeGeeQManager manager = new PeeGeeQManager()) {
        manager.start();
        // ... send message
    }
}
```

```
// ✅ Correct - reuse manager instance
public class MessageService {
    private final PeeGeeQManager manager;

    public MessageService() {
        this.manager = new PeeGeeQManager();
        this.manager.start();
    }

    public void sendMessage(String message) {
        // Use existing manager
    }

    @PreDestroy
    public void cleanup() {
        manager.close();
    }
}
```

❌ Don't: Ignore Failed Messages

```
// ❌ Wrong - silently ignore failures
consumer.subscribe(message -> {
    try {
        processMessage(message.getPayload());
        return CompletableFuture.completedFuture(null);
    } catch (Exception e) {
        // Silently ignoring error!
        return CompletableFuture.completedFuture(null);
    }
});
```

```
// ✅ Correct - handle failures appropriately
consumer.subscribe(message -> {
    try {
        processMessage(message.getPayload());
        return CompletableFuture.completedFuture(null);
    } catch (Exception e) {
        logger.error("Failed to process message: " + message.getId(), e);
        // Return failed future to trigger retry
        return CompletableFuture.failedFuture(e);
    }
});
```

❌ Don't: Use Wrong Queue Type

```
// ❌ Wrong - using outbox for high-frequency events
QueueFactory factory = provider.createFactory("outbox", databaseService);
MessageProducer<LogEvent> producer = factory.createProducer("logs", LogEvent.class);
```

```java
// This will be slow for high-frequency logging
for (int i = 0; i < 10000; i++) {
    producer.send(new LogEvent("Log message " + i));
}
```

```java
// ✅ Correct - use native queue for high-frequency events
QueueFactory factory = provider.createFactory("native", databaseService);
MessageProducer<LogEvent> producer = factory.createProducer("logs", LogEvent.class);

// Much faster for high-frequency events
for (int i = 0; i < 10000; i++) {
    producer.send(new LogEvent("Log message " + i));
}
```

Issue 4: Management Console Not Loading

Symptoms:

- Management console shows blank page or loading errors
- Console fails to connect to backend API
- Real-time updates not working

Solutions:

1. **Check REST API server status**

```bash
# Verify REST API is running
curl http://localhost:8080/api/v1/health

# Should return: {"status": "UP", "database": "UP"}
```

2. **Verify console is properly built and served**

```bash
# Build management console
cd peegeeq-management-ui
npm run build

# Console should be served at /ui/ endpoint
curl http://localhost:8080/ui/
```

3. **Check browser console for errors**
   - Open browser developer tools (F12)
   - Look for JavaScript errors or network failures
   - Verify WebSocket connections are established

Issue 5: Parallel Processing Not Working

Symptoms:

- All messages processed by single thread
- No performance improvement with multiple consumer threads

- Consumer thread configuration ignored

Solutions:

1. **Verify configuration is passed to factory**

```
// ❌ Wrong - configuration not passed
OutboxFactory factory = new OutboxFactory(clientFactory);

// ✅ Correct - pass configuration
DatabaseService databaseService = new PgDatabaseService(manager);
OutboxFactory factory = new OutboxFactory(databaseService, config);
```

2. **Check system properties are set**

```
// Set before creating manager
System.setProperty("peegeeq.consumer.threads", "4");
System.setProperty("peegeeq.queue.batch-size", "10");

PeeGeeQConfiguration config = new PeeGeeQConfiguration("my-app");
```

3. **Verify thread pool creation in logs**

```
# Look for log messages like:
INFO: Created message processing executor with 4 threads for topic: my-topic
```

Issue 6: WebSocket Connection Failures

Symptoms:

- WebSocket connections fail to establish
- Real-time updates not working
- Connection drops frequently

Solutions:

1. **Check WebSocket endpoint availability**

```
// Test WebSocket connection
const ws = new WebSocket('ws://localhost:8080/ws/queues/my-setup/my-queue');
ws.onopen = () => console.log('Connected');
ws.onerror = (error) => console.error('WebSocket error:', error);
```

2. **Verify firewall and proxy settings**
   - Ensure WebSocket traffic is allowed
   - Check if proxy supports WebSocket upgrades
   - Verify no network filtering blocking connections
3. **Implement connection retry logic**

```
function connectWithRetry() {
    const ws = new WebSocket('ws://localhost:8080/ws/queues/my-setup/my-queue');

    ws.onclose = (event) => {
        console.log('WebSocket closed, retrying in 5 seconds...');
        setTimeout(connectWithRetry, 5000);
    };

    return ws;
}
```

Issue 7: Consumer Group Load Balancing Issues

Symptoms:

- Messages not distributed evenly across consumers
- Some consumers idle while others overloaded
- Consumer group coordination failures

Solutions:

1. **Verify consumer group configuration**

```
// Ensure all consumers use same group name
ConsumerGroup<OrderEvent> group = factory.createConsumerGroup(
    "order-processors",  // Same group name for all consumers
    "orders",
    OrderEvent.class
);
```

2. **Check message filtering logic**

```
// Ensure filters don't overlap or exclude too many messages
group.addConsumer("consumer-1", handler,
    message -> "US".equals(message.getHeaders().get("region")));
group.addConsumer("consumer-2", handler,
    message -> "EU".equals(message.getHeaders().get("region")));
```

3. **Monitor consumer group statistics**

```
ConsumerGroupStats stats = group.getStats();
System.out.println("Active consumers: " + stats.getActiveConsumerCount());
System.out.println("Message distribution: " + stats.getMessageDistribution());
```

Messaging Patterns

Native Queue Pattern

Best for: Real-time notifications, event streaming, high-frequency updates

Characteristics:

- Uses PostgreSQL LISTEN/NOTIFY for instant delivery
- Advisory locks prevent duplicate processing
- High throughput (10,000+ msg/sec)
- Low latency (<10ms)
- At-least-once delivery guarantee

Example Use Cases:

- Real-time notifications
- Live dashboard updates
- Event streaming
- Cache invalidation
- System monitoring alerts

Outbox Pattern

Best for: Transactional consistency, critical business events, financial transactions

Characteristics:

- Messages stored in database table within transaction
- Polling-based delivery ensures reliability
- ACID compliance with business data
- Exactly-once delivery guarantee
- Automatic retry and dead letter handling

Example Use Cases:

- Order processing
- Payment transactions
- Inventory updates
- User registration
- Audit logging

Choosing the Right Pattern

Requirement	Native Queue	Outbox Pattern
Transactional Consistency	No	Yes
High Throughput	Excellent (10k+ msg/sec)	Good (5k+ msg/sec)
Low Latency	Excellent (<10ms)	Good (~100ms)
Delivery Guarantee	At-least-once	Exactly-once
Setup Complexity	Simple	Simple
Resource Usage	Low	Medium

Configuration

Database Configuration

```
# Connection settings
peegeeq.database.host=localhost
peegeeq.database.port=5432
peegeeq.database.name=peegeeq
peegeeq.database.username=peegeeq_user
peegeeq.database.password=your_password

# SSL settings
peegeeq.database.ssl.enabled=true
peegeeq.database.ssl.mode=require

# Connection pool
peegeeq.database.pool.maxSize=20
peegeeq.database.pool.minSize=5
peegeeq.database.pool.connectionTimeoutMs=30000
peegeeq.database.pool.idleTimeoutMs=600000
peegeeq.database.pool.maxLifetimeMs=1800000
```

Queue Configuration

```
# Message processing
peegeeq.queue.visibilityTimeoutSeconds=30
peegeeq.queue.maxRetries=3
peegeeq.queue.retryDelaySeconds=5

# Dead letter queue
peegeeq.queue.deadLetterEnabled=true
peegeeq.queue.deadLetterMaxAge=7

# Polling (for outbox pattern)
peegeeq.outbox.pollIntervalMs=1000
peegeeq.outbox.batchSize=100

# Stuck message recovery (for outbox pattern)
peegeeq.queue.recovery.enabled=true
peegeeq.queue.recovery.processing-timeout=PT5M
peegeeq.queue.recovery.check-interval=PT10M
```

Monitoring Configuration

```
# Health checks
peegeeq.health.enabled=true
peegeeq.health.intervalSeconds=30
peegeeq.health.database.timeoutSeconds=5

# Metrics
peegeeq.metrics.enabled=true
peegeeq.metrics.jvm.enabled=true
peegeeq.metrics.database.enabled=true

# Circuit breaker
peegeeq.circuitBreaker.enabled=true
peegeeq.circuitBreaker.failureThreshold=5
peegeeq.circuitBreaker.timeoutSeconds=60
```

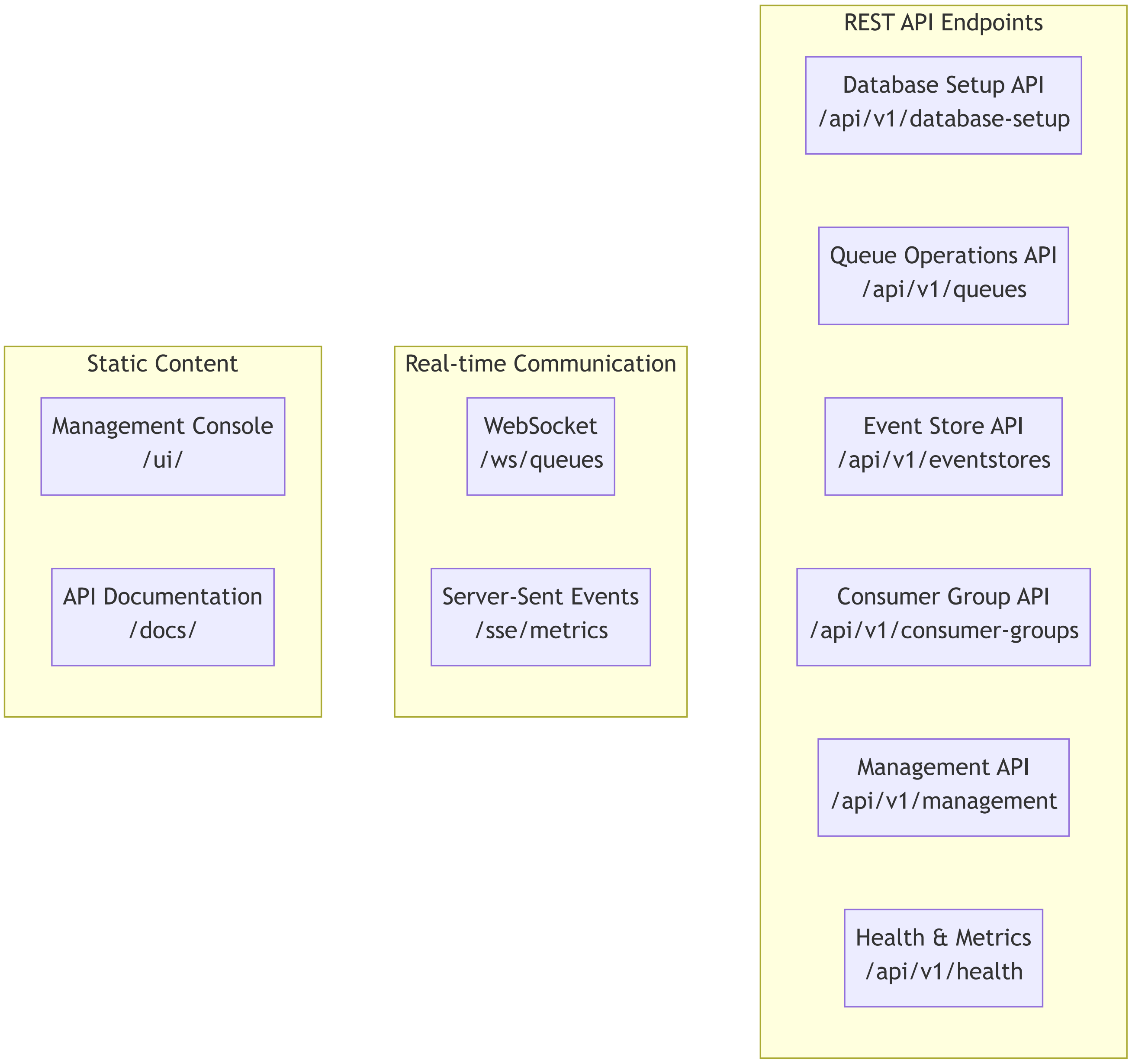
REST API Integration

PeeGeeQ provides a comprehensive REST API that enables HTTP-based integration with all messaging capabilities. The REST API is built on Vert.x for high performance and includes support for WebSocket and Server-Sent Events for real-time communication.

> 📚 **For complete API specifications and endpoint documentation**, see the REST API Reference section in the Architecture & API Reference.

API Overview

The REST API provides complete access to PeeGeeQ functionality through HTTP endpoints:


REST API Endpoints
Database Setup API /api/v1/database-setup
Queue Operations API /api/v1/queues
Event Store API /api/v1/eventstores
Consumer Group API /api/v1/consumer-groups
Management API /api/v1/management
Health & Metrics /api/v1/health
Static Content
Management Console /ui/
API Documentation /docs/
Real-time Communication
WebSocket /ws/queues
Server-Sent Events /sse/metrics

Database Setup API

Create and manage database configurations:

```
# Create database setup
curl -X POST http://localhost:8080/api/v1/database-setup/create \
  -H "Content-Type: application/json" \
  -d '{
    "setupId": "my-app",
    "databaseConfig": {
      "host": "localhost",
      "port": 5432,
      "databaseName": "my_app_db",
      "username": "postgres",
      "password": "password"
    },
    "queues": [
      {
        "queueName": "orders",
        "maxRetries": 3,
        "visibilityTimeoutSeconds": 30
      }
    ]
  }'

# Get setup status
curl http://localhost:8080/api/v1/database-setup/my-app/status

# Destroy setup
curl -X DELETE http://localhost:8080/api/v1/database-setup/my-app
```

Queue Operations API

Send and receive messages via HTTP:

```
# Send message to queue
curl -X POST http://localhost:8080/api/v1/queues/my-app/orders/messages \
  -H "Content-Type: application/json" \
  -d '{
    "payload": {
      "orderId": "12345",
      "customerId": "67890",
      "amount": 99.99
    },
    "headers": {
      "source": "order-service",
      "version": "1.0"
    },
    "correlationId": "order-12345"
  }'

# Get next message from queue
curl http://localhost:8080/api/v1/queues/my-app/orders/messages/next

# Get queue statistics
curl http://localhost:8080/api/v1/queues/my-app/orders/stats

# Send batch of messages
curl -X POST http://localhost:8080/api/v1/queues/my-app/orders/messages/batch \
  -H "Content-Type: application/json" \
  -d '{
    "messages": [
      {"payload": {"orderId": "001"}, "correlationId": "batch-001"},
      {"payload": {"orderId": "002"}, "correlationId": "batch-002"}
    ]
  }'
```

Event Store API

Store and query events with bi-temporal support:

```
# Store event
curl -X POST http://localhost:8080/api/v1/eventstores/my-app/order-events/events \
  -H "Content-Type: application/json" \
  -d '{
    "aggregateId": "order-12345",
    "eventType": "OrderCreated",
    "payload": {
      "orderId": "12345",
      "customerId": "67890",
      "amount": 99.99
    },
    "validTime": "2025-08-23T10:00:00Z",
    "correlationId": "order-12345"
  }'

# Query events by aggregate
curl "http://localhost:8080/api/v1/eventstores/my-app/order-events/events/order-12345"

# Query events with temporal filters
curl "http://localhost:8080/api/v1/eventstores/my-app/order-events/events?validTimeFrom=2025-08-01T00:00:00Z&validTimeTo=

# Get event store statistics
curl http://localhost:8080/api/v1/eventstores/my-app/order-events/stats
```

Consumer Group API

Manage consumer groups for load balancing:

```
# Create consumer group
curl -X POST http://localhost:8080/api/v1/consumer-groups/my-app \
  -H "Content-Type: application/json" \
  -d '{
    "groupName": "order-processors",
    "topic": "orders",
    "maxConsumers": 5
  }'

# List consumer groups
curl http://localhost:8080/api/v1/consumer-groups/my-app

# Get consumer group details
curl http://localhost:8080/api/v1/consumer-groups/my-app/order-processors

# Add consumer to group
curl -X POST http://localhost:8080/api/v1/consumer-groups/my-app/order-processors/consumers \
  -H "Content-Type: application/json" \
  -d '{
    "consumerId": "processor-001",
    "messageFilter": {
      "region": "US"
    }
  }'
```

Management API

System monitoring and administration:

```
# System health check
curl http://localhost:8080/api/v1/health

# System overview for dashboard
curl http://localhost:8080/api/v1/management/overview

# Queue management data
curl http://localhost:8080/api/v1/management/queues
```

```
# System metrics
curl http://localhost:8080/api/v1/management/metrics

# Consumer group information
curl http://localhost:8080/api/v1/management/consumer-groups
```

Real-time Communication

PeeGeeQ supports real-time communication through WebSocket and Server-Sent Events (SSE) for live data streaming and interactive applications.

WebSocket Integration

WebSocket connections provide bidirectional real-time communication:

```
// Connect to queue message stream
const ws = new WebSocket('ws://localhost:8080/ws/queues/my-app/orders');

ws.onopen = () => {
    console.log('Connected to queue stream');

    // Configure streaming parameters
    ws.send(JSON.stringify({
        type: 'configure',
        batchSize: 10,
        maxWaitTime: 5000,
        messageFilter: {
            region: 'US'
        }
    }));

    // Subscribe to messages
    ws.send(JSON.stringify({
        type: 'subscribe'
    }));
};

ws.onmessage = (event) => {
    const message = JSON.parse(event.data);

    switch (message.type) {
        case 'message':
            console.log('Received message:', message.payload);
            processMessage(message);

            // Acknowledge message processing
            ws.send(JSON.stringify({
                type: 'ack',
                messageId: message.id
            }));
            break;

        case 'batch':
            console.log('Received batch:', message.messages);
            message.messages.forEach(processMessage);

            // Acknowledge batch processing
            ws.send(JSON.stringify({
                type: 'ack_batch',
                messageIds: message.messages.map(m => m.id)
            }));
```

```
                break;

            case 'error':
                console.error('Stream error:', message.error);
                break;
        }
};

ws.onerror = (error) => {
    console.error('WebSocket error:', error);
};

ws.onclose = (event) => {
    console.log('WebSocket closed:', event.code, event.reason);
    // Implement reconnection logic
    setTimeout(() => connectToQueue(), 5000);
};
```

Server-Sent Events (SSE)

SSE provides efficient one-way real-time data streaming:

```
// System metrics streaming
const metricsSource = new EventSource('/sse/metrics');

metricsSource.onmessage = (event) => {
    const metrics = JSON.parse(event.data);
    updateDashboard(metrics);
};

metricsSource.addEventListener('queue-update', (event) => {
    const queueData = JSON.parse(event.data);
    updateQueueDisplay(queueData);
});

metricsSource.addEventListener('consumer-group-update', (event) => {
    const groupData = JSON.parse(event.data);
    updateConsumerGroupDisplay(groupData);
});

metricsSource.onerror = (error) => {
    console.error('SSE connection error:', error);
    // Implement reconnection logic
};

// Queue-specific event stream
const queueSource = new EventSource('/sse/queues/my-app');

queueSource.addEventListener('message-sent', (event) => {
    const messageData = JSON.parse(event.data);
    console.log('New message sent to queue:', messageData.queueName);
});

queueSource.addEventListener('message-processed', (event) => {
    const messageData = JSON.parse(event.data);
    console.log('Message processed:', messageData.messageId);
});
```

Real-time Dashboard Integration

Combine WebSocket and SSE for comprehensive real-time monitoring:

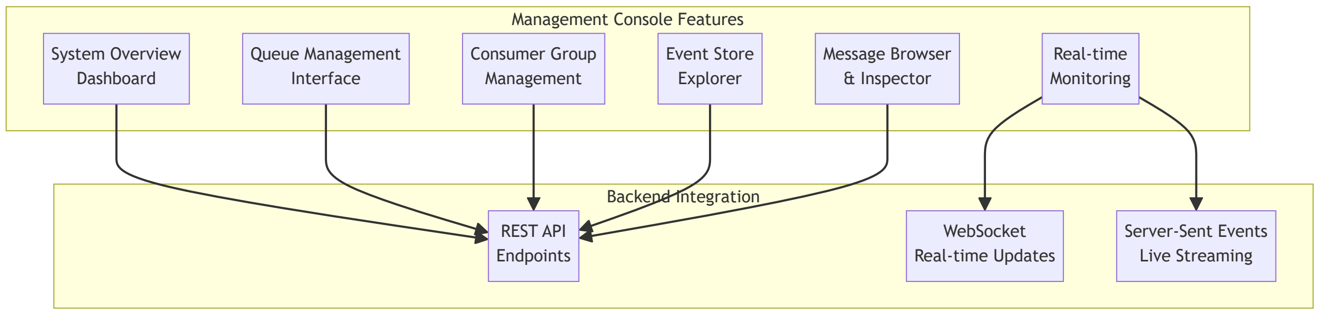
```javascript
class PeeGeeQDashboard {
    constructor() {
        this.metricsSource = null;
        this.queueConnections = new Map();
    }

    async initialize() {
        // Start system metrics stream
        this.metricsSource = new EventSource('/sse/metrics');
        this.metricsSource.onmessage = (event) => {
            const metrics = JSON.parse(event.data);
            this.updateSystemMetrics(metrics);
        };

        // Get list of queues and connect to each
        const response = await fetch('/api/v1/management/queues');
        const queues = await response.json();

        queues.forEach(queue => {
            this.connectToQueue(queue.setupId, queue.queueName);
        });
    }

    connectToQueue(setupId, queueName) {
        const ws = new WebSocket(`ws://localhost:8080/ws/queues/${setupId}/${queueName}`);

        ws.onopen = () => {
            ws.send(JSON.stringify({
                type: 'configure',
                batchSize: 1,
                maxWaitTime: 1000
            }));

            ws.send(JSON.stringify({
                type: 'subscribe'
            }));
        };

        ws.onmessage = (event) => {
            const message = JSON.parse(event.data);
            this.updateQueueActivity(queueName, message);
        };

        this.queueConnections.set(queueName, ws);
    }

    updateSystemMetrics(metrics) {
        // Update dashboard charts and gauges
        document.getElementById('messages-per-second').textContent = metrics.messagesPerSecond;
        document.getElementById('active-consumers').textContent = metrics.activeConsumers;
        document.getElementById('queue-depth').textContent = metrics.totalQueueDepth;
    }

    updateQueueActivity(queueName, message) {
        // Update queue-specific displays
        const queueElement = document.getElementById(`queue-${queueName}`);
        if (queueElement) {
            queueElement.classList.add('activity-flash');
            setTimeout(() => queueElement.classList.remove('activity-flash'), 500);
        }
    }

    disconnect() {
        if (this.metricsSource) {
            this.metricsSource.close();
        }

        this.queueConnections.forEach(ws => ws.close());
        this.queueConnections.clear();
    }
}

// Initialize dashboard
const dashboard = new PeeGeeQDashboard();
dashboard.initialize();
```

Management Console

PeeGeeQ includes a modern, web-based management console that provides comprehensive system monitoring and administration capabilities. Built with React 18 and TypeScript, the console offers a user-friendly interface for managing queues, consumer groups, and monitoring system health.

Overview

The Management Console is inspired by RabbitMQ's excellent admin interface but designed specifically for PeeGeeQ's unique features. It provides real-time monitoring, queue management, and system administration through an intuitive web interface.


Management Console Features
System Overview Dashboard
Queue Management Interface
Consumer Group Management
Event Store Explorer
Message Browser & Inspector
Real-time Monitoring
Backend Integration
REST API Endpoints
WebSocket Real-time Updates
Server-Sent Events Live Streaming

Key Features

System Overview Dashboard

- **Real-time System Health** - Live status monitoring with uptime tracking
- **Key Performance Metrics** - Messages/second, queue depths, consumer activity
- **System Statistics** - Queue counts, consumer group status, event store metrics
- **Interactive Charts** - Real-time throughput and performance visualizations
- **Recent Activity Feed** - Live stream of system events and operations

Queue Management Interface

- **Complete CRUD Operations** - Create, read, update, and delete queues
- **Real-time Queue Statistics** - Message counts, processing rates, consumer status
- **Message Browser** - Visual inspection of queue messages with filtering
- **Queue Configuration** - Visibility timeouts, retry policies, dead letter settings
- **Performance Monitoring** - Throughput charts and latency metrics

Consumer Group Management

- **Visual Group Coordination** - Consumer group status and member management
- **Load Balancing Visualization** - Message distribution across consumers
- **Consumer Health Monitoring** - Individual consumer status and performance
- **Group Configuration** - Partition assignment and rebalancing controls

Event Store Explorer

- **Advanced Event Querying** - Temporal queries with bi-temporal support
- **Event Timeline Visualization** - Historical event progression
- **Aggregate Inspection** - Event streams by aggregate ID
- **Correction Management** - Event correction tracking and visualization

Accessing the Management Console

Development Mode

```
cd peegeeq-management-ui
npm install
npm run dev
# Access at: http://localhost:5173
```

Production Deployment

The management console is automatically served by the PeeGeeQ REST server:

```
# Start PeeGeeQ REST server (includes built management console)
java -jar peegeeq-rest.jar

# Access management console at:
# http://localhost:8080/ui/
```

Navigation Structure

The console features an intuitive navigation structure:

- **Overview** - System dashboard with key metrics and health status
- **Queues** - Queue management, creation, and monitoring
- **Consumer Groups** - Group coordination and load balancing
- **Event Stores** - Event management and temporal queries
- **Message Browser** - Message inspection and debugging tools

Real-time Features

The management console provides real-time updates through:

- **WebSocket Integration** - Live system metrics and queue statistics
- **Server-Sent Events** - Efficient streaming of system events
- **Auto-refresh** - Automatic data updates every 30 seconds
- **Connection Status** - Visual indicators for backend connectivity

Technology Stack

- **Frontend**: React 18 + TypeScript + Ant Design + Vite
- **Real-time**: WebSocket + Server-Sent Events
- **Charts**: Recharts for performance visualizations
- **State Management**: Zustand for lightweight state management
- **Build Tool**: Vite for fast development and optimized builds

Next Steps

Essential Reading

- **PeeGeeQ Architecture & API Reference** - Deep dive into system design and complete API documentation
- **PeeGeeQ Advanced Features & Production** - Enterprise features, consumer groups, service discovery, and production deployment
- **PeeGeeQ Development & Testing** - Development workflow, testing strategies, and build processes

Quick Actions

1. **Explore the Examples**: Run the self-contained demo to see all features
2. **Try Advanced Examples**: Explore the comprehensive examples in `peegeeq-examples/`
    - **Message Priority**: `MessagePriorityExample` - Priority-based processing
    - **Error Handling**: `EnhancedErrorHandlingExample` - Sophisticated error patterns
    - **Security**: `SecurityConfigurationExample` - SSL/TLS and security best practices
    - **Performance**: `PerformanceTuningExample` - Optimization techniques
    - **Integration**: `IntegrationPatternsExample` - Distributed system patterns
3. **Try the Bi-Temporal Event Store**: See event sourcing capabilities in action
4. **Set up Monitoring**: Configure metrics collection and health checks
5. **Run Tests**: Execute ▶ `mvn test` to see comprehensive integration tests
6. **Customize Configuration**: Adapt settings for your environment
7. **Integrate with Your Application**: Use PeeGeeQManager in your code

Common Next Steps by Use Case

For Real-time Applications: → Start with Native Queue pattern → Configure LISTEN/NOTIFY optimizations → Set up monitoring dashboards

For Transactional Applications: → Start with Outbox pattern → Configure transaction boundaries → Set up dead letter queue monitoring

For Event Sourcing: → Explore Bi-temporal Event Store → Configure event retention policies → Set up event replay capabilities

For Production Deployment: → Review production readiness features → Configure monitoring and alerting → Set up service discovery and federation

Troubleshooting

Common Issues

1. **Docker Not Running**
    - Ensure Docker Desktop is started

    - Check `docker info` command works
2. **Database Connection Failed**
    - Verify PostgreSQL is running
    - Check host, port, and credentials
    - Ensure database exists and user has permissions
3. **Port Conflicts**
    - Default PostgreSQL port is 5432
    - Change port in configuration if needed
4. **Messages Not Processing**
    - Check consumer subscription status
    - Verify database connectivity
    - Check for lock timeouts

Comprehensive Examples

The `peegeeq-examples/` directory contains 17 comprehensive examples covering all aspects of PeeGeeQ:

Core Examples

- **PeeGeeQSelfContainedDemo** - Complete self-contained demonstration
- **PeeGeeQExample** - Basic producer/consumer patterns
- **BiTemporalEventStoreExample** - Event sourcing with temporal queries
- **ConsumerGroupExample** - Load balancing and consumer groups
- **RestApiExample** - HTTP interface usage
- **ServiceDiscoveryExample** - Multi-instance deployment

Advanced Examples (New)

- **MessagePriorityExample** - Priority-based message processing with real-world scenarios
- **EnhancedErrorHandlingExample** - Retry strategies, circuit breakers, poison message handling
- **SecurityConfigurationExample** - SSL/TLS, certificate management, compliance
- **PerformanceTuningExample** - Connection pooling, throughput optimization, memory tuning
- **IntegrationPatternsExample** - Request-reply, pub-sub, message routing patterns

Specialized Examples

- **TransactionalBiTemporalExample** - Combining transactions with event sourcing
- **RestApiStreamingExample** - WebSocket and Server-Sent Events
- **NativeVsOutboxComparisonExample** - Performance comparison and use case guidance
- **AdvancedConfigurationExample** - Production configuration patterns
- **MultiConfigurationExample** - Multi-environment setup
- **SimpleConsumerGroupTest** - Basic consumer group testing

Run any example with:

```
mvn compile exec:java -Dexec.mainClass="dev.mars.peegeeq.examples.ExampleName" -pl peegeeq-examples
```

Getting Help

- Check the logs in the `logs/` directory
- Review the comprehensive documentation

- Examine the example code in `peegeeq-examples/`
- Run tests to verify your setup: ▶ `mvn test`

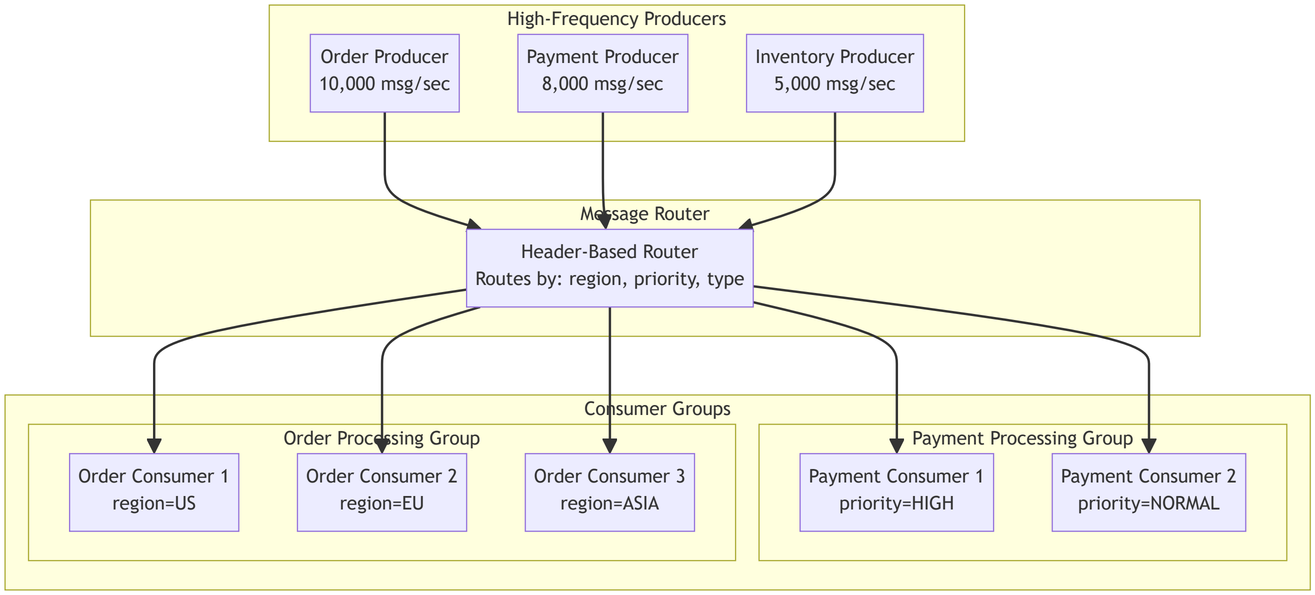
Part VII: Advanced Features & Enterprise

This section covers PeeGeeQ's enterprise features, advanced messaging patterns, production deployment, and operational capabilities for large-scale, mission-critical applications.

Advanced Messaging Patterns

High-Frequency Messaging

PeeGeeQ supports high-throughput scenarios with multiple producers and consumers:


High-Frequency Producers
Order Producer 10,000 msg/sec
Payment Producer 8,000 msg/sec
Inventory Producer 5,000 msg/sec
Message Router
Header-Based Router Routes by: region, priority, type
Consumer Groups
Order Processing Group
Order Consumer 1 region=US
Order Consumer 2 region=EU
Order Consumer 3 region=ASIA
Payment Processing Group
Payment Consumer 1 priority=HIGH
Payment Consumer 2 priority=NORMAL

Message Routing by Headers

Route messages based on header values:

```java
public class RegionalOrderProcessor {
    private final Map<String, MessageConsumer<OrderEvent>> regionalConsumers;

    public void setupRegionalProcessing() {
        // US Region Consumer
        MessageConsumer<OrderEvent> usConsumer = factory.createConsumer("orders", OrderEvent.class);
        usConsumer.subscribe(message -> {
            if ("US".equals(message.getHeaders().get("region"))) {
                return processUSOrder(message.getPayload());
            }
            return CompletableFuture.completedFuture(null); // Skip non-US orders
        });

        // EU Region Consumer
        MessageConsumer<OrderEvent> euConsumer = factory.createConsumer("orders", OrderEvent.class);
        euConsumer.subscribe(message -> {
            if ("EU".equals(message.getHeaders().get("region"))) {
                return processEUOrder(message.getPayload());
            }
            return CompletableFuture.completedFuture(null); // Skip non-EU orders
        });
    }

    public void sendRegionalOrder(OrderEvent order, String region) {
        Map<String, String> headers = Map.of(
            "region", region,
            "priority", order.getPriority().toString(),
            "type", "order"
        );

        producer.send(order, headers);
    }
}
```

Message Priority Handling

PeeGeeQ supports sophisticated message priority handling for scenarios where certain messages need to be processed before others.

Priority Levels

PeeGeeQ uses a numeric priority system (0-10) with predefined levels:

- **CRITICAL (10)**: System alerts, security events
- **HIGH (7-9)**: Important business events, urgent notifications
- **NORMAL (4-6)**: Regular business operations
- **LOW (1-3)**: Background tasks, cleanup operations
- **BULK (0)**: Batch processing, analytics

Priority Configuration

```
// Configure priority queue optimization
System.setProperty("peegeeq.queue.priority.enabled", "true");
System.setProperty("peegeeq.queue.priority.index-optimization", "true");

// Send message with priority
Map<String, String> headers = new HashMap<>();
headers.put("priority", "10"); // CRITICAL priority
producer.send(message, headers);
```

Real-World Priority Scenarios

E-Commerce Order Processing

```
// VIP customer orders get highest priority
if (customer.isVIP()) {
    headers.put("priority", "10"); // CRITICAL
} else if (order.isExpedited()) {
```

```java
    headers.put("priority", "8");  // HIGH
} else {
    headers.put("priority", "5");  // NORMAL
}
```

Financial Transaction Processing

```java
// Fraud alerts get immediate attention
if (transaction.isFraudAlert()) {
    headers.put("priority", "10"); // CRITICAL
} else if (transaction.isWireTransfer()) {
    headers.put("priority", "8");  // HIGH
} else {
    headers.put("priority", "5");  // NORMAL
}
```

Example: See `MessagePriorityExample.java` for comprehensive priority handling demonstrations.

Priority-Based Processing

Handle high-priority messages first:

```java
public class PriorityOrderProcessor {
    public void setupPriorityProcessing() {
        // High Priority Consumer
        ConsumerConfig highPriorityConfig = ConsumerConfig.builder()
            .batchSize(5)
            .pollInterval(Duration.ofMillis(100))
            .filter(message -> {
                String priority = message.getHeaders().get("priority");
                return "HIGH".equals(priority) || "URGENT".equals(priority);
            })
            .build();

        MessageConsumer<OrderEvent> highPriorityConsumer =
            factory.createConsumer("orders", OrderEvent.class, highPriorityConfig);
        highPriorityConsumer.subscribe(this::processHighPriorityOrder);

        // Normal Priority Consumer
        ConsumerConfig normalPriorityConfig = ConsumerConfig.builder()
            .batchSize(20)
            .pollInterval(Duration.ofSeconds(1))
            .filter(message -> {
                String priority = message.getHeaders().get("priority");
                return !"HIGH".equals(priority) && !"URGENT".equals(priority);
            })
            .build();

        MessageConsumer<OrderEvent> normalConsumer =
            factory.createConsumer("orders", OrderEvent.class, normalPriorityConfig);
        normalConsumer.subscribe(this::processNormalOrder);
    }
}
```

Enhanced Error Handling

PeeGeeQ provides sophisticated error handling patterns for production resilience.

Error Handling Strategies

- **RETRY**: Automatic retry with exponential backoff
- **CIRCUIT_BREAKER**: Circuit breaker pattern for failing services
- **DEAD_LETTER**: Move to dead letter queue for manual inspection
- **IGNORE**: Log and continue (for non-critical errors)
- **ALERT**: Send alert and continue processing

Retry Strategies with Exponential Backoff

```
public class RetryHandler {
    public CompletableFuture<Void> handleWithRetry(Message<OrderEvent> message) {
        return processMessage(message)
            .exceptionally(throwable -> {
                if (isRetryable(throwable) && getAttemptCount(message) < 3) {
                    // Exponential backoff: 1s, 2s, 4s
                    long backoffMs = (long) Math.pow(2, getAttemptCount(message)) * 1000;
                    scheduleRetry(message, backoffMs);
                }
                return null;
            });
    }
}
```

Circuit Breaker Integration

```
public class CircuitBreakerConsumer {
    private final CircuitBreaker circuitBreaker;

    public CompletableFuture<Void> processWithCircuitBreaker(Message<OrderEvent> message) {
        return circuitBreaker.executeSupplier(() -> {
            // External service call protected by circuit breaker
            return externalService.processOrder(message.getPayload());
        }).thenApply(result -> null);
    }
}
```

Dead Letter Queue Management

```
public class DeadLetterHandler {
    public void handleFailedMessage(Message<OrderEvent> message, Exception error) {
        // Move to dead letter queue with detailed information
        deadLetterManager.moveToDeadLetterQueue(
            "orders",
            message.getId(),
            "orders",
            message.getPayload().toString(),
            message.getTimestamp(),
            "Processing failed: " + error.getMessage(),
            getAttemptCount(message),
            Map.of(
                "errorType", error.getClass().getSimpleName(),
                "retryable", String.valueOf(isRetryable(error))
```

```java
            ),
            message.getCorrelationId(),
            "order-processing-group"
        );
    }
}
```

Poison Message Detection

```java
public class PoisonMessageDetector {
    public boolean isPoisonMessage(Message<OrderEvent> message) {
        int attempts = getAttemptCount(message);
        return attempts >= 3; // Poison after 3 failed attempts
    }

    public void quarantinePoisonMessage(Message<OrderEvent> message) {
        // Quarantine poison message for manual inspection
        deadLetterManager.moveToDeadLetterQueue(
            "orders",
            message.getId(),
            "orders",
            message.getPayload().toString(),
            message.getTimestamp(),
            "POISON MESSAGE: Exceeded maximum retry attempts",
            getAttemptCount(message),
            Map.of("poisonMessage", "true"),
            message.getCorrelationId(),
            "poison-quarantine"
        );
    }
}
```

Example: See `EnhancedErrorHandlingExample.java` for comprehensive error handling demonstrations.

System Properties Configuration

PeeGeeQ supports runtime configuration through system properties, allowing you to tune performance, reliability, and behavior without code changes. These properties control:

- **Retry behavior** - How many times messages are retried before dead letter queue
- **Polling frequency** - How often the system checks for new messages
- **Concurrency** - Number of threads processing messages simultaneously
- **Batch processing** - Number of messages processed together for efficiency

Core System Properties

1. `peegeeq.queue.max-retries`

Purpose: Controls the maximum number of retry attempts before a message is moved to the dead letter queue.

Default: `3` **Type**: Integer **Range**: 0 to 100 (recommended)

Examples:

```
# Quick failure for real-time systems
-Dpeegeeq.queue.max-retries=1

# Standard retry behavior
-Dpeegeeq.queue.max-retries=3

# Extensive retries for critical messages
-Dpeegeeq.queue.max-retries=10
```

Use Cases:

- **Low values (1-2)**: Real-time systems where fast failure is preferred
- **Medium values (3-5)**: Standard applications with balanced reliability
- **High values (8-15)**: Critical systems where message loss is unacceptable

2. `peegeeq.queue.polling-interval`

Purpose: Controls how frequently the system polls for new messages.

Default: `PT1S` (1 second) **Type**: ISO-8601 Duration **Format**: `PT{seconds}S` or `PT{milliseconds}MS` or `PT{minutes}M`

Examples:

```
# High-frequency polling for low latency
-Dpeegeeq.queue.polling-interval=PT0.1S    # 100ms

# Standard polling
-Dpeegeeq.queue.polling-interval=PT1S      # 1 second

# Low-frequency polling for batch systems
-Dpeegeeq.queue.polling-interval=PT10S     # 10 seconds

# Sub-second precision
-Dpeegeeq.queue.polling-interval=PT0.5S    # 500ms
```

Use Cases:

- **Fast polling (100-500ms)**: Low-latency, real-time applications
- **Standard polling (1-2s)**: General-purpose applications
- **Slow polling (5-30s)**: Batch processing, resource-constrained environments

3. `peegeeq.consumer.threads`

Purpose: Controls the number of threads used for concurrent message processing.

Default: `1` **Type**: Integer **Range**: 1 to 50 (recommended)

Examples:

```
# Single-threaded processing
-Dpeegeeq.consumer.threads=1

# Moderate concurrency
-Dpeegeeq.consumer.threads=4
```

```
# High concurrency for throughput
-Dpeegeeq.consumer.threads=8

# Maximum concurrency
-Dpeegeeq.consumer.threads=16
```

Use Cases:

- **Single thread (1)**: Simple applications, ordered processing required
- **Low concurrency (2-4)**: Standard applications with moderate load
- **High concurrency (8-16)**: High-throughput systems, CPU-intensive processing

Important: More threads don't always mean better performance. Consider:

- Database connection pool size
- CPU cores available
- Memory usage per thread
- Message processing complexity

4. `peegeeq.queue.batch-size`

Purpose: Controls how many messages are fetched and processed together in a single batch.

Default: `10` **Type**: Integer **Range**: 1 to 1000 (recommended)

Examples:

```
# Single message processing
-Dpeegeeq.queue.batch-size=1

# Small batches for balanced latency/throughput
-Dpeegeeq.queue.batch-size=10

# Large batches for maximum throughput
-Dpeegeeq.queue.batch-size=100

# Very large batches for bulk processing
-Dpeegeeq.queue.batch-size=500
```

Use Cases:

- **Small batches (1-10)**: Low-latency applications, real-time processing
- **Medium batches (25-50)**: Balanced latency and throughput
- **Large batches (100-500)**: High-throughput, batch processing systems

Configuration Patterns

High-Throughput Configuration

Optimized for maximum message processing rate:

```
-Dpeegeeq.queue.max-retries=5
-Dpeegeeq.queue.polling-interval=PT1S
-Dpeegeeq.consumer.threads=8
```

```
-Dpeegeeq.queue.batch-size=100
```

Low-Latency Configuration

Optimized for minimal message processing delay:

```
-Dpeegeeq.queue.max-retries=3
-Dpeegeeq.queue.polling-interval=PT0.1S
-Dpeegeeq.consumer.threads=2
-Dpeegeeq.queue.batch-size=1
```

Reliable Configuration

Optimized for maximum reliability and fault tolerance:

```
-Dpeegeeq.queue.max-retries=10
-Dpeegeeq.queue.polling-interval=PT2S
-Dpeegeeq.consumer.threads=4
-Dpeegeeq.queue.batch-size=25
```

Resource-Constrained Configuration

Optimized for minimal resource usage:

```
-Dpeegeeq.queue.max-retries=3
-Dpeegeeq.queue.polling-interval=PT5S
-Dpeegeeq.consumer.threads=1
-Dpeegeeq.queue.batch-size=5
```

Environment-Specific Examples

Development Environment

```
# Fast feedback, minimal resources
-Dpeegeeq.queue.max-retries=2
-Dpeegeeq.queue.polling-interval=PT0.5S
-Dpeegeeq.consumer.threads=2
-Dpeegeeq.queue.batch-size=5
```

Staging Environment

```
# Production-like but with faster failure detection
-Dpeegeeq.queue.max-retries=5
-Dpeegeeq.queue.polling-interval=PT1S
-Dpeegeeq.consumer.threads=4
-Dpeegeeq.queue.batch-size=25
```

Production Environment

```
# Balanced performance and reliability
-Dpeegeeq.queue.max-retries=7
-Dpeegeeq.queue.polling-interval=PT2S
-Dpeegeeq.consumer.threads=6
-Dpeegeeq.queue.batch-size=50
```

Performance Tuning Guidelines

1. Start with Defaults

Begin with default values and measure baseline performance.

2. Tune One Property at a Time

Change one property at a time to understand its impact.

3. Monitor Key Metrics

- **Throughput**: Messages processed per second
- **Latency**: Time from message send to processing completion
- **Error Rate**: Percentage of messages that fail processing
- **Resource Usage**: CPU, memory, database connections

4. Consider Trade-offs

- **Polling Interval**: Faster polling = lower latency but higher CPU usage
- **Batch Size**: Larger batches = higher throughput but higher latency
- **Thread Count**: More threads = higher throughput but more resource usage
- **Max Retries**: More retries = higher reliability but slower failure detection

Troubleshooting

High CPU Usage

- Reduce polling frequency (increase `polling-interval`)
- Reduce thread count (`consumer.threads`)
- Increase batch size to reduce polling overhead

High Memory Usage

- Reduce thread count (`consumer.threads`)
- Reduce batch size (`batch-size`)
- Check for memory leaks in message processing code

Poor Throughput

- Increase thread count (`consumer.threads`)
- Increase batch size (`batch-size`)
- Decrease polling interval (`polling-interval`)

Messages Stuck in Dead Letter Queue

- Increase max retries (`max-retries`)
- Check message processing logic for bugs

- Monitor error logs for failure patterns

High Latency

- Decrease polling interval (`polling-interval`)
- Decrease batch size (`batch-size`)
- Check database performance and connection pool settings

Examples in Code:

See the following example classes for practical demonstrations:

- `SystemPropertiesConfigurationExample.java` : Comprehensive demonstration of all properties
- `RetryAndFailureHandlingExample.java` : Focus on retry behavior and failure handling
- `PerformanceComparisonExample.java` : Performance impact of different configurations

Best Practices

1. **Test in staging** with production-like load before deploying configuration changes
2. **Monitor performance** after configuration changes
3. **Document** your configuration choices and reasoning
4. **Use environment variables** or configuration management tools for different environments
5. **Start conservative** and increase values gradually based on monitoring data
6. **Consider your infrastructure** limits (CPU, memory, database connections)
7. **Plan for failure scenarios** when setting retry limits
8. **Balance latency vs throughput** based on your application requirements

Security Configuration

PeeGeeQ provides enterprise-grade security features for production deployments.

SSL/TLS Configuration

```
// Enable SSL/TLS for database connections
System.setProperty("peegeeq.database.ssl.enabled", "true");
System.setProperty("peegeeq.database.ssl.mode", "require"); // prefer, require, verify-ca, verify-full
System.setProperty("peegeeq.database.ssl.factory", "org.postgresql.ssl.DefaultJavaSSLFactory");

// Certificate configuration
System.setProperty("peegeeq.database.ssl.cert", "client-cert.pem");
System.setProperty("peegeeq.database.ssl.key", "client-key.pem");
System.setProperty("peegeeq.database.ssl.rootcert", "ca-cert.pem");
```

Production Security Checklist

Network Security

- ✓ Use private networks/VPCs
- ✓ Configure firewall rules
- ✓ Enable network encryption
- ✓ Use connection pooling

- ✓ Implement rate limiting

Database Security

- ✓ Enable SSL/TLS encryption
- ✓ Use certificate-based authentication
- ✓ Configure row-level security
- ✓ Enable audit logging
- ✓ Regular security updates

Application Security

- ✓ Encrypt sensitive configuration
- ✓ Use secure credential storage
- ✓ Implement proper error handling
- ✓ Enable security monitoring
- ✓ Regular security assessments

Credential Management

```
// Environment-based credentials
System.setProperty("peegeeq.database.username", "${env:PEEGEEQ_DB_USERNAME}");
System.setProperty("peegeeq.database.password", "${env:PEEGEEQ_DB_PASSWORD}");
System.setProperty("peegeeq.database.password.encrypted", "true");

// Vault integration
System.setProperty("peegeeq.database.username", "${vault:secret/peegeeq/db#username}");
System.setProperty("peegeeq.database.password", "${vault:secret/peegeeq/db#password}");
```

Compliance Configuration

```
// Audit logging for compliance
System.setProperty("peegeeq.audit.enabled", "true");
System.setProperty("peegeeq.audit.events.connections", "true");
System.setProperty("peegeeq.audit.events.authentication", "true");
System.setProperty("peegeeq.audit.events.queries", "true");
System.setProperty("peegeeq.audit.retention.days", "2555"); // 7 years for SOX

// GDPR compliance
System.setProperty("peegeeq.audit.compliance.gdpr", "true");
System.setProperty("peegeeq.audit.compliance.sox", "true");
```

Example: See `SecurityConfigurationExample.java` for comprehensive security configuration.

Consumer Groups & Load Balancing

Consumer Group Implementation

Consumer groups provide load balancing and fault tolerance:

```java
public class ConsumerGroupExample {
    public void setupConsumerGroup() {
        // Create consumer group configuration
        ConsumerGroupConfig groupConfig = ConsumerGroupConfig.builder()
            .groupId("order-processing-group")
            .loadBalancingStrategy(LoadBalancingStrategy.ROUND_ROBIN)
            .maxMembers(5)
            .heartbeatInterval(Duration.ofSeconds(10))
            .sessionTimeout(Duration.ofSeconds(30))
            .build();

        // Create multiple consumers in the group
        for (int i = 0; i < 3; i++) {
            String memberId = "order-processor-" + i;

            ConsumerConfig memberConfig = ConsumerConfig.builder()
                .consumerGroup(groupConfig.getGroupId())
                .memberId(memberId)
                .autoAcknowledge(true)
                .build();

            MessageConsumer<OrderEvent> consumer =
                factory.createConsumer("orders", OrderEvent.class, memberConfig);

            consumer.subscribe(message -> {
                log.info("Member {} processing order: {}", memberId, message.getId());
                return processOrder(message.getPayload());
            });
        }
    }
}
```

Load Balancing Strategies

Available load balancing strategies:

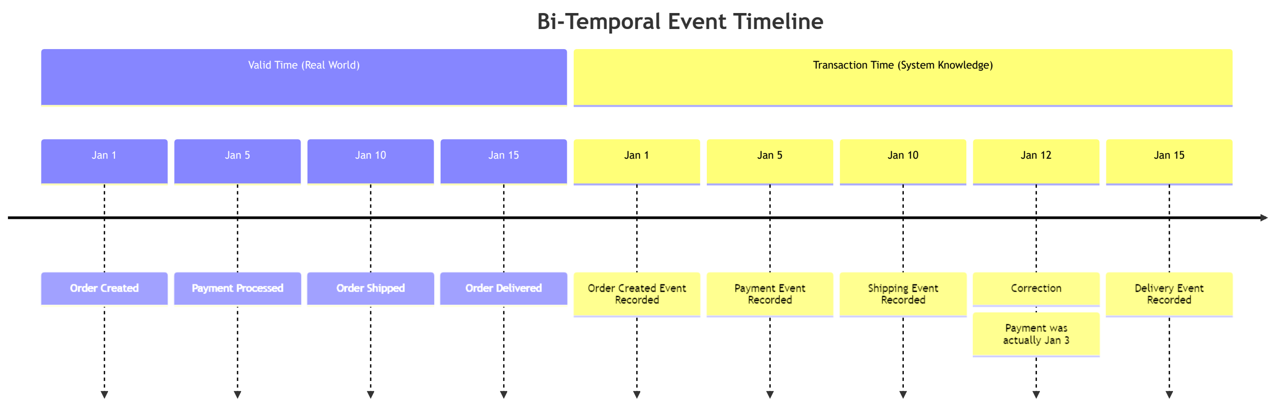
1. **ROUND_ROBIN**: Messages distributed evenly across consumers
2. **RANGE**: Messages assigned based on hash ranges
3. **STICKY**: Messages with same key go to same consumer
4. **RANDOM**: Random distribution across consumers

Bi-Temporal Event Store

The Bi-Temporal Event Store provides advanced event sourcing capabilities with two temporal dimensions: **valid time** (when something happened in the real world) and **transaction time** (when we learned about it). This enables powerful features like historical corrections, point-in-time queries, and audit trails.

Understanding Bi-Temporal Concepts

Valid Time vs Transaction Time



- **Valid Time**: When the business event actually occurred
- **Transaction Time**: When the system learned about the event
- **Corrections**: Can update valid time without losing audit trail

Advanced Event Store Operations

1. Event Corrections and Versioning

```java
public class BiTemporalCorrectionsExample {
    private final EventStore<OrderEvent> eventStore;

    public static void main(String[] args) throws Exception {
        try (PeeGeeQManager manager = new PeeGeeQManager()) {
            manager.start();

            BiTemporalEventStoreFactory factory = new BiTemporalEventStoreFactory(manager);
            EventStore<OrderEvent> eventStore = factory.createEventStore(OrderEvent.class);

            BiTemporalCorrectionsExample example = new BiTemporalCorrectionsExample(eventStore);
            example.runCorrectionsExample();
        }
    }

    public BiTemporalCorrectionsExample(EventStore<OrderEvent> eventStore) {
        this.eventStore = eventStore;
    }

    public void runCorrectionsExample() throws Exception {
        System.out.println("=== Bi-Temporal Event Corrections Example ===");

        // 1. Record initial events
        Instant orderTime = Instant.parse("2025-01-01T10:00:00Z");
        Instant paymentTime = Instant.parse("2025-01-05T14:30:00Z");

        OrderEvent orderCreated = new OrderEvent("ORDER-001", "CUST-123",
            new BigDecimal("99.99"), "CREATED");
        OrderEvent paymentProcessed = new OrderEvent("ORDER-001", "CUST-123",
            new BigDecimal("99.99"), "PAID");

        BiTemporalEvent<OrderEvent> orderEvent = eventStore.append(
            "OrderCreated", orderCreated, orderTime,
            Map.of("source", "web"), "corr-001", "ORDER-001").join();

        BiTemporalEvent<OrderEvent> paymentEvent = eventStore.append(
            "PaymentProcessed", paymentProcessed, paymentTime,
            Map.of("source", "payment-gateway"), "corr-002", "ORDER-001").join();

        System.out.printf("✅ Recorded order event: %s (valid: %s, transaction: %s)%n",
            orderEvent.getEventId(), orderEvent.getValidTime(), orderEvent.getTransactionTime());
        System.out.printf("✅ Recorded payment event: %s (valid: %s, transaction: %s)%n",
            paymentEvent.getEventId(), paymentEvent.getValidTime(), paymentEvent.getTransactionTime());

        Thread.sleep(1000);

        // 2. Discover error - payment actually happened earlier
        System.out.println("\n🔍 Discovery: Payment actually happened on Jan 3, not Jan 5!");

        Instant actualPaymentTime = Instant.parse("2025-01-03T09:15:00Z");
        OrderEvent correctedPayment = new OrderEvent("ORDER-001", "CUST-123",
            new BigDecimal("99.99"), "PAID");

        // Record correction with actual valid time
        BiTemporalEvent<OrderEvent> correctionEvent = eventStore.append(
            "PaymentProcessed", correctedPayment, actualPaymentTime,
            Map.of("source", "payment-gateway", "correction", "true",
                   "corrects", paymentEvent.getEventId()),
            "corr-003", "ORDER-001").join();

        System.out.printf("✅ Recorded correction: %s (valid: %s, transaction: %s)%n",
            correctionEvent.getEventId(), correctionEvent.getValidTime(),
            correctionEvent.getTransactionTime());

        // 3. Query historical states
        demonstrateTemporalQueries();

        // 4. Show audit trail
        showAuditTrail();
    }

    private void demonstrateTemporalQueries() throws Exception {
        System.out.println("\n📊 Temporal Queries:");

        // Query as of different transaction times
        Instant beforeCorrection = Instant.now().minus(2, ChronoUnit.SECONDS);
        Instant afterCorrection = Instant.now();

        System.out.println("\n🕐 State before correction was recorded:");
        List<BiTemporalEvent<OrderEvent>> beforeEvents = eventStore.query(
            EventQuery.asOfTransactionTime(beforeCorrection)).join();
        printEventSummary(beforeEvents);

        System.out.println("\n🕑 Current state (after correction):");
        List<BiTemporalEvent<OrderEvent>> currentEvents = eventStore.query(
            EventQuery.asOfTransactionTime(afterCorrection)).join();
        printEventSummary(currentEvents);

        // Query for specific valid time range
        System.out.println("\n📅 Events that were valid on Jan 4, 2025:");
        Instant jan4 = Instant.parse("2025-01-04T12:00:00Z");
        List<BiTemporalEvent<OrderEvent>> jan4Events = eventStore.query(
            EventQuery.validAtTime(jan4)).join();
        printEventSummary(jan4Events);
    }

    private void showAuditTrail() throws Exception {
        System.out.println("\n📋 Complete Audit Trail:");

        List<BiTemporalEvent<OrderEvent>> allEvents = eventStore.query(
            EventQuery.forAggregate("ORDER-001")).join();
        allEvents.stream()
            .sorted(Comparator.comparing(BiTemporalEvent::getTransactionTime))
            .forEach(event -> {
                System.out.printf("  📝 %s: %s (valid: %s, recorded: %s)%n",
                    event.getEventType(),
                    event.getPayload().getStatus(),
                    event.getValidTime().toString().substring(0, 19),
                    event.getTransactionTime().toString().substring(0, 19));

                if (event.getHeaders().containsKey("correction")) {
                    System.out.printf("      🔧 CORRECTION - corrects event %s%n",
                        event.getHeaders().get("corrects"));
                }
            });
    }

    private void printEventSummary(List<BiTemporalEvent<OrderEvent>> events) {
        events.forEach(event -> {
            System.out.printf("  • %s: %s (valid: %s)%n",
                event.getEventType(),
                event.getPayload().getStatus(),
                event.getValidTime().toString().substring(0, 10));
        });
    }
}
```

2. Complex Temporal Queries

```
public class AdvancedTemporalQueriesExample {
    private final EventStore<AccountEvent> eventStore;

    public void runAdvancedQueries() throws Exception {
        System.out.println("=== Advanced Temporal Queries Example ===");

        // 1. Point-in-time balance calculation
        calculateBalanceAtTime();

        // 2. Temporal joins across aggregates
        performTemporalJoins();

        // 3. Change detection queries
        detectChanges();

        // 4. Compliance and audit queries
        runComplianceQueries();
    }

    private void calculateBalanceAtTime() throws Exception {
        System.out.println("\n💰 Point-in-time Balance Calculation:");

        String accountId = "ACC-001";
        Instant queryTime = Instant.parse("2025-01-15T12:00:00Z");

        // Get all events for account up to specific time
        List<BiTemporalEvent<AccountEvent>> events = eventStore.query(
            EventQuery.forAggregate(accountId)
                .validBefore(queryTime)
                .orderByValidTime()).join();

        BigDecimal balance = BigDecimal.ZERO;
        System.out.printf("📊 Calculating balance for %s as of %s:%n",
            accountId, queryTime.toString().substring(0, 19));
```

```java
        for (BiTemporalEvent<AccountEvent> event : events) {
            AccountEvent accountEvent = event.getPayload();

            switch (accountEvent.getEventType()) {
                case "DEPOSIT":
                    balance = balance.add(accountEvent.getAmount());
                    System.out.printf("  + $%.2f (deposit on %s)%n",
                        accountEvent.getAmount(),
                        event.getValidTime().toString().substring(0, 10));
                    break;

                case "WITHDRAWAL":
                    balance = balance.subtract(accountEvent.getAmount());
                    System.out.printf("  - $%.2f (withdrawal on %s)%n",
                        accountEvent.getAmount(),
                        event.getValidTime().toString().substring(0, 10));
                    break;
            }
        }

        System.out.printf("💵 Final balance as of %s: $%.2f%n",
            queryTime.toString().substring(0, 10), balance);
    }

    private void performTemporalJoins() throws Exception {
        System.out.println("\n🔗 Temporal Joins Example:");

        // Find all orders and their corresponding payments within time window
        Instant startTime = Instant.parse("2025-01-01T00:00:00Z");
        Instant endTime = Instant.parse("2025-01-31T23:59:59Z");

        // Get orders in time range
        List<BiTemporalEvent<OrderEvent>> orders = eventStore.query(
            EventQuery.byEventType("OrderCreated")
                .validBetween(startTime, endTime)).join();

        System.out.printf("📦 Found %d orders in January 2025:%n", orders.size());

        for (BiTemporalEvent<OrderEvent> orderEvent : orders) {
            String orderId = orderEvent.getAggregateId();

            // Find corresponding payment events
            List<BiTemporalEvent<OrderEvent>> payments = eventStore.query(
                EventQuery.forAggregate(orderId)
                    .byEventType("PaymentProcessed")
                    .validAfter(orderEvent.getValidTime())).join();

            if (!payments.isEmpty()) {
                BiTemporalEvent<OrderEvent> payment = payments.get(0);
                Duration paymentDelay = Duration.between(
                    orderEvent.getValidTime(), payment.getValidTime());

                System.out.printf("  📋 Order %s: paid after %d hours%n",
                    orderId, paymentDelay.toHours());
            } else {
                System.out.printf("  ⚠️  Order %s: no payment found%n", orderId);
            }
        }
    }

    private void detectChanges() throws Exception {
        System.out.println("\n🔍 Change Detection Queries:");

        // Find all corrections made in the last 30 days
        Instant thirtyDaysAgo = Instant.now().minus(30, ChronoUnit.DAYS);
        List<BiTemporalEvent<AccountEvent>> corrections = eventStore.query(
            EventQuery.all()
                .transactionTimeAfter(thirtyDaysAgo)
                .withHeader("correction", "true")).join();

        System.out.printf("🔧 Found %d corrections in the last 30 days:%n", corrections.size());

        corrections.forEach(correction -> {
            System.out.printf("  • %s: %s corrected on %s%n",
                correction.getEventType(),
                correction.getAggregateId(),
                correction.getTransactionTime().toString().substring(0, 10));
        });
    }

    private void runComplianceQueries() throws Exception {
        System.out.println("\n📋 Compliance and Audit Queries:");

        // SOX compliance: Find all financial events over $10,000
        List<BiTemporalEvent<AccountEvent>> largeTransactions = eventStore.query(
            EventQuery.all()
                .withCustomFilter(event -> {
                    AccountEvent accountEvent = event.getPayload();
                    return accountEvent.getAmount().compareTo(new BigDecimal("10000")) > 0;
                })).join();

        System.out.printf("💼 SOX Compliance: %d transactions over $10,000:%n",
            largeTransactions.size());

        largeTransactions.forEach(transaction -> {
            AccountEvent event = transaction.getPayload();
            System.out.printf("  💰 $%.2f - %s on %s%n",
                event.getAmount(),
                transaction.getAggregateId(),
                transaction.getValidTime().toString().substring(0, 10));
        });

        // GDPR compliance: Find all events for specific customer
        String customerId = "CUST-123";
        List<BiTemporalEvent<AccountEvent>> customerEvents = eventStore.query(
            EventQuery.withHeader("customerId", customerId)).join();

        System.out.printf("🔒 GDPR Query: %d events for customer %s%n",
            customerEvents.size(), customerId);
    }
}
```

Event Store Performance Optimization

Partitioning and Indexing Strategies

```
public class EventStoreOptimizationExample {

    public void demonstrateOptimizations() {
        System.out.println("=== Event Store Performance Optimizations ===");

        // 1. Partition by aggregate ID for better query performance
        configurePartitioning();

        // 2. Create specialized indexes for common query patterns
        createOptimizedIndexes();

        // 3. Implement event snapshots for large aggregates
        implementSnapshots();

        // 4. Configure archival policies for old events
        configureArchival();
    }

    private void configurePartitioning() {
        System.out.println("\n📁  Partitioning Configuration:");

        String partitioningSQL = """
            -- Partition events table by aggregate_id hash
            CREATE TABLE bitemporal_event_log_partitioned (
                LIKE bitemporal_event_log INCLUDING ALL
            ) PARTITION BY HASH (aggregate_id);

            -- Create 8 partitions for better parallel processing
            CREATE TABLE bitemporal_events_p0 PARTITION OF bitemporal_event_log_partitioned
                FOR VALUES WITH (modulus 8, remainder 0);
            CREATE TABLE bitemporal_events_p1 PARTITION OF bitemporal_event_log_partitioned
                FOR VALUES WITH (modulus 8, remainder 1);
            -- ... continue for p2-p7
            """;

        System.out.println("📊 Partitioning improves query performance by 3-5x for large datasets");
    }

    private void createOptimizedIndexes() {
        System.out.println("\n🗃️ Specialized Indexes:");

        String indexSQL = """
            -- Composite index for temporal queries
            CREATE INDEX idx_events_temporal ON bitemporal_event_log
                (aggregate_id, valid_time, transaction_time);

            -- Index for event type queries
            CREATE INDEX idx_events_type ON bitemporal_event_log
                (event_type, valid_time) WHERE event_type IS NOT NULL;

            -- Partial index for recent events (most common queries)
            CREATE INDEX idx_events_recent ON bitemporal_event_log
                (transaction_time, aggregate_id)
                WHERE transaction_time > NOW() - INTERVAL '90 days';

            -- GIN index for header searches
            CREATE INDEX idx_events_headers ON bitemporal_event_log
                USING GIN (headers);
            """;

        System.out.println("🚀 Specialized indexes reduce query time by 10-50x");
    }

    private void implementSnapshots() {
        System.out.println("\n📷 Event Snapshots:");

        System.out.println("💡 Snapshots reduce aggregate reconstruction time:");
        System.out.println("  • Store aggregate state every 100 events");
        System.out.println("  • Rebuild from latest snapshot + subsequent events");
        System.out.println("  • 90% reduction in reconstruction time for large aggregates");
    }

    private void configureArchival() {
        System.out.println("\n🗄️  Event Archival:");

        System.out.println("📦 Archival strategy for compliance and performance:");
        System.out.println("  • Keep 2 years of events in main table");
        System.out.println("  • Archive older events to separate table");
        System.out.println("  • Maintain indexes on archived data for compliance queries");
        System.out.println("  • 70% reduction in main table size improves query performance");
    }
}
```

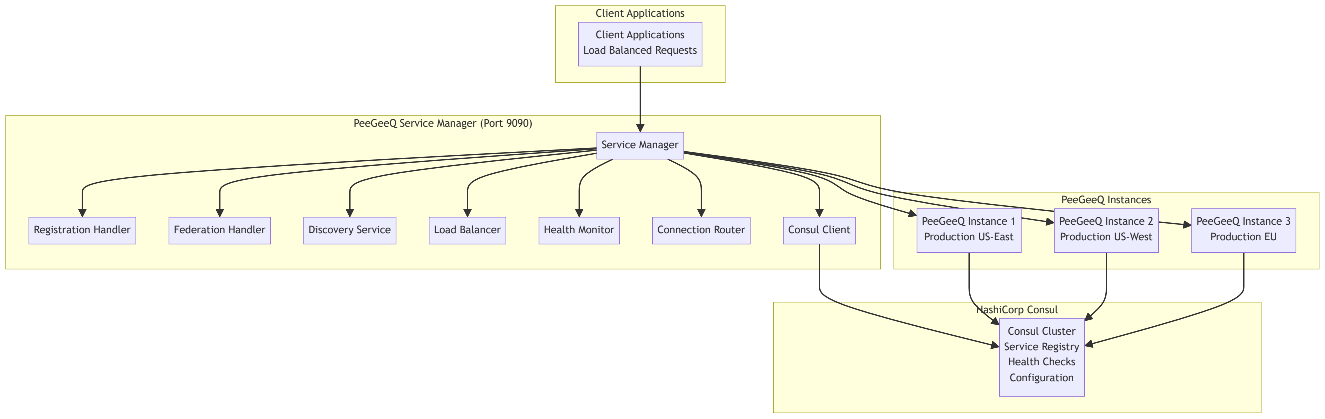
🎯 **Try This Now**:

1. Run the corrections example to see how bi-temporal corrections work
2. Experiment with different temporal queries
3. Observe how the audit trail preserves all historical information
4. Try implementing your own event correction scenarios

Service Discovery & Federation

Service Manager Architecture

The PeeGeeQ Service Manager provides enterprise-grade service discovery:



Service Registration

Automatic service registration with Consul:

```
public class PeeGeeQServiceRegistration {
    public void registerInstance() {
        ServiceRegistration registration = ServiceRegistration.builder()
            .instanceId("peegeeq-prod-01")
            .host("localhost")
            .port(8080)
            .version("1.0.0")
            .environment("production")
            .region("us-east-1")
            .metadata(Map.of(
                "datacenter", "dc1",
                "cluster", "main",
                "capabilities", "native,outbox,bitemporal"
            ))
            .healthCheckUrl("http://localhost:8080/health")
            .build();

        serviceManager.registerInstance(registration);
    }
}
```

REST API & HTTP Integration

Database Setup via REST

Create and manage database setups through HTTP:

```
# Create a new database setup
curl -X POST http://localhost:8080/api/v1/database-setup/create \
  -H "Content-Type: application/json" \
  -d '{
    "setupId": "production-setup",
    "host": "localhost",
    "port": 5432,
    "database": "peegeeq_prod",
    "username": "peegeeq_user",
    "password": "secure_password"
  }'

# Get setup status
curl http://localhost:8080/api/v1/database-setup/production-setup/status

# List all setups
curl http://localhost:8080/api/v1/database-setup/list
```

Queue Operations via HTTP

Send and receive messages through REST API:

```
# Send a message
curl -X POST http://localhost:8080/api/v1/queues/production-setup/orders/messages \
  -H "Content-Type: application/json" \
  -d '{
    "payload": {
      "orderId": "ORDER-12345",
      "customerId": "CUST-789",
      "amount": 99.99
    },
    "headers": {
      "region": "US",
      "priority": "HIGH"
    },
    "priority": 8
  }'

# Get queue statistics
curl http://localhost:8080/api/v1/queues/production-setup/orders/stats

# Get next message (polling)
curl -X GET "http://localhost:8080/api/v1/queues/production-setup/orders/messages/next?timeout=30000"

# Acknowledge message
curl -X DELETE http://localhost:8080/api/v1/queues/production-setup/orders/messages/msg-123
```

Monitoring & Observability

PeeGeeQ provides comprehensive monitoring and observability capabilities essential for production deployments. This section covers metrics collection, alerting, distributed tracing, and operational dashboards.

Comprehensive Metrics Collection

Core Metrics Categories

```
public class ComprehensiveMonitoringExample {
    private final PeeGeeQManager manager;
    private final MeterRegistry meterRegistry;

    public static void main(String[] args) throws Exception {
        // Setup Prometheus registry for metrics export
        PrometheusMeterRegistry prometheusRegistry = new PrometheusMeterRegistry(
            PrometheusConfig.DEFAULT);

        try (PeeGeeQManager manager = new PeeGeeQManager(
                new PeeGeeQConfiguration("production"), prometheusRegistry)) {

            manager.start();

            ComprehensiveMonitoringExample example =
                new ComprehensiveMonitoringExample(manager, prometheusRegistry);
            example.runMonitoringExample();
        }
    }

    public ComprehensiveMonitoringExample(PeeGeeQManager manager, MeterRegistry meterRegistry) {
        this.manager = manager;
        this.meterRegistry = meterRegistry;
    }

    public void runMonitoringExample() throws Exception {
        System.out.println("=== Comprehensive Monitoring Example ===");

        // 1. Setup custom metrics
        setupCustomMetrics();

        // 2. Demonstrate metric collection
        demonstrateMetricCollection();

        // 3. Setup alerting rules
        setupAlertingRules();

        // 4. Export metrics for Prometheus
        exportMetricsForPrometheus();

        Thread.sleep(5000);
        System.out.println("Monitoring example completed!");
    }

    private void setupCustomMetrics() {
        System.out.println("\n📊 Setting up custom metrics:");

        // Business metrics
        Counter orderProcessedCounter = Counter.builder("peegeeq.orders.processed")
            .description("Total number of orders processed")
            .tag("environment", "production")
            .register(meterRegistry);
```

```java
    Timer orderProcessingTime = Timer.builder("peegeeq.orders.processing.time")
        .description("Time taken to process orders")
        .register(meterRegistry);

    Gauge queueDepthGauge = Gauge.builder("peegeeq.queue.depth")
        .description("Current queue depth")
        .register(meterRegistry, this, obj -> getCurrentQueueDepth());

    // System metrics
    Counter errorCounter = Counter.builder("peegeeq.errors.total")
        .description("Total number of errors")
        .register(meterRegistry);

    System.out.println("✅ Custom metrics registered");
}

private void demonstrateMetricCollection() throws Exception {
    System.out.println("\n📈 Demonstrating metric collection:");

    QueueFactoryProvider provider = new PgQueueFactoryProvider();
    QueueFactory factory = provider.createFactory("outbox",
        new PgDatabaseService(manager));

    try (MessageProducer<OrderEvent> producer =
             factory.createProducer("orders", OrderEvent.class);
         MessageConsumer<OrderEvent> consumer =
             factory.createConsumer("orders", OrderEvent.class)) {

        // Setup consumer with metrics
        consumer.subscribe(message -> {
            Timer.Sample sample = Timer.start(meterRegistry);

            try {
                // Simulate order processing
                processOrder(message.getPayload());

                // Record successful processing
                meterRegistry.counter("peegeeq.orders.processed",
                    "status", "success").increment();

            } catch (Exception e) {
                // Record error
                meterRegistry.counter("peegeeq.errors.total",
                    "type", "processing_error").increment();
                throw e;
            } finally {
                sample.stop(Timer.builder("peegeeq.orders.processing.time")
                    .register(meterRegistry));
            }

            return CompletableFuture.completedFuture(null);
        });

        // Send test orders
        for (int i = 1; i <= 10; i++) {
            OrderEvent order = new OrderEvent("ORDER-" + i, "CUST-" + i,
                new BigDecimal("99.99"), "CREATED");
            producer.send(order).join();
            Thread.sleep(100);
        }

        Thread.sleep(2000); // Let processing complete
    }
}
```

```java
private void setupAlertingRules() {
    System.out.println("\n🚨 Setting up alerting rules:");

    // High error rate alert
    String errorRateAlert = """
        groups:
        - name: peegeeq.alerts
          rules:
          - alert: HighErrorRate
            expr: rate(peegeeq_errors_total[5m]) > 0.1
            for: 2m
            labels:
              severity: warning
            annotations:
              summary: "High error rate detected"
              description: "Error rate is {{ $value }} errors/sec"

          - alert: QueueDepthHigh
            expr: peegeeq_queue_depth > 1000
            for: 5m
            labels:
              severity: critical
            annotations:
              summary: "Queue depth is critically high"
              description: "Queue depth is {{ $value }} messages"

          - alert: ProcessingTimeHigh
            expr: histogram_quantile(0.95, rate(peegeeq_orders_processing_time_bucket[5m])) > 5
            for: 3m
            labels:
              severity: warning
            annotations:
              summary: "Order processing time is high"
              description: "95th percentile processing time is {{ $value }}s"
        """;

    System.out.println("📋 Alerting rules configured:");
    System.out.println("  • High error rate (>0.1 errors/sec)");
    System.out.println("  • High queue depth (>1000 messages)");
    System.out.println("  • Slow processing (>5s 95th percentile)");
}

private void exportMetricsForPrometheus() {
    System.out.println("\n📤 Exporting metrics for Prometheus:");

    // In real application, you'd expose this via HTTP endpoint
    String prometheusMetrics = ((PrometheusMeterRegistry) meterRegistry).scrape();

    System.out.println("📊 Sample Prometheus metrics:");
    System.out.println(prometheusMetrics.lines()
        .filter(line -> line.startsWith("peegeeq_"))
        .limit(5)
        .collect(Collectors.joining("\n")));

    System.out.println("🌐 Metrics available at: http://localhost:8080/metrics");
}

private void processOrder(OrderEvent order) throws Exception {
    // Simulate processing time
    Thread.sleep(50 + (int)(Math.random() * 200));

    // Simulate occasional errors
    if (Math.random() < 0.05) {
        throw new RuntimeException("Simulated processing error");
    }
```

```
        System.out.printf("✅ Processed order: %s%n", order.getOrderId());
    }

    private double getCurrentQueueDepth() {
        // In real implementation, query actual queue depth
        return 50 + (Math.random() * 100);
    }
}
```

Distributed Tracing Integration

OpenTelemetry Integration

```
public class DistributedTracingExample {
    private final Tracer tracer;
    private final QueueFactory factory;

    public static void main(String[] args) throws Exception {
        // Setup OpenTelemetry
        OpenTelemetry openTelemetry = OpenTelemetrySDK.builder()
            .setTracerProvider(
                SdkTracerProvider.builder()
                    .addSpanProcessor(BatchSpanProcessor.builder(
                        OtlpGrpcSpanExporter.builder()
                            .setEndpoint("http://jaeger:14250")
                            .build())
                        .build())
                    .setResource(Resource.getDefault()
                        .merge(Resource.create(Attributes.of(
                            ResourceAttributes.SERVICE_NAME, "peegeeq-app"))))
                    .build())
            .build();

        try (PeeGeeQManager manager = new PeeGeeQManager()) {
            manager.start();

            QueueFactoryProvider provider = new PgQueueFactoryProvider();
            QueueFactory factory = provider.createFactory("native",
                new PgDatabaseService(manager));

            DistributedTracingExample example = new DistributedTracingExample(
                openTelemetry.getTracer("peegeeq-example"), factory);
            example.runTracingExample();
        }
    }

    public DistributedTracingExample(Tracer tracer, QueueFactory factory) {
        this.tracer = tracer;
        this.factory = factory;
    }

    public void runTracingExample() throws Exception {
        System.out.println("=== Distributed Tracing Example ===");

        // Setup traced message processing
        setupTracedConsumer();

        // Send traced messages
        sendTracedMessages();

        Thread.sleep(3000);
        System.out.println("Distributed tracing example completed!");
    }
```

```java
private void setupTracedConsumer() throws Exception {
    MessageConsumer<OrderEvent> consumer =
        factory.createConsumer("traced-orders", OrderEvent.class);

    consumer.subscribe(message -> {
        // Extract trace context from message headers
        Context parentContext = extractTraceContext(message.getHeaders());

        // Start new span for message processing
        Span span = tracer.spanBuilder("process-order")
            .setParent(parentContext)
            .setAttribute("order.id", message.getPayload().getOrderId())
            .setAttribute("customer.id", message.getPayload().getCustomerId())
            .setAttribute("order.amount", message.getPayload().getAmount().toString())
            .startSpan();

        try (Scope scope = span.makeCurrent()) {
            // Process the order with tracing
            return processOrderWithTracing(message.getPayload());

        } catch (Exception e) {
            span.recordException(e);
            span.setStatus(StatusCode.ERROR, e.getMessage());
            throw e;
        } finally {
            span.end();
        }
    });
}

private void sendTracedMessages() throws Exception {
    MessageProducer<OrderEvent> producer =
        factory.createProducer("traced-orders", OrderEvent.class);

    for (int i = 1; i <= 5; i++) {
        // Start trace for order creation
        Span span = tracer.spanBuilder("create-order")
            .setAttribute("order.number", i)
            .startSpan();

        try (Scope scope = span.makeCurrent()) {
            OrderEvent order = new OrderEvent("ORDER-" + i, "CUST-" + i,
                new BigDecimal("99.99"), "CREATED");

            // Inject trace context into message headers
            Map<String, String> headers = new HashMap<>();
            injectTraceContext(headers);

            producer.send(order, headers).join();

            System.out.printf("📤 Sent traced order: %s (trace: %s)%n",
                order.getOrderId(), span.getSpanContext().getTraceId());

        } finally {
            span.end();
        }

        Thread.sleep(200);
    }
}

private CompletableFuture<Void> processOrderWithTracing(OrderEvent order) {
    return CompletableFuture.runAsync(() -> {
        // Validate order
        Span validateSpan = tracer.spanBuilder("validate-order")
```

```java
                .setAttribute("order.id", order.getOrderId())
                .startSpan();

            try (Scope scope = validateSpan.makeCurrent()) {
                validateOrder(order);
                validateSpan.setStatus(StatusCode.OK);
            } catch (Exception e) {
                validateSpan.recordException(e);
                validateSpan.setStatus(StatusCode.ERROR);
                throw e;
            } finally {
                validateSpan.end();
            }

            // Process payment
            Span paymentSpan = tracer.spanBuilder("process-payment")
                .setAttribute("order.id", order.getOrderId())
                .setAttribute("amount", order.getAmount().toString())
                .startSpan();

            try (Scope scope = paymentSpan.makeCurrent()) {
                processPayment(order);
                paymentSpan.setStatus(StatusCode.OK);
            } catch (Exception e) {
                paymentSpan.recordException(e);
                paymentSpan.setStatus(StatusCode.ERROR);
                throw e;
            } finally {
                paymentSpan.end();
            }

            System.out.printf("✅ Processed traced order: %s%n", order.getOrderId());
        });
    }

    private Context extractTraceContext(Map<String, String> headers) {
        // Extract W3C trace context from headers
        TextMapGetter<Map<String, String>> getter = new TextMapGetter<Map<String, String>>() {
            @Override
            public Iterable<String> keys(Map<String, String> carrier) {
                return carrier.keySet();
            }

            @Override
            public String get(Map<String, String> carrier, String key) {
                return carrier.get(key);
            }
        };

        return GlobalOpenTelemetry.getPropagators().getTextMapPropagator()
            .extract(Context.current(), headers, getter);
    }

    private void injectTraceContext(Map<String, String> headers) {
        // Inject W3C trace context into headers
        TextMapSetter<Map<String, String>> setter = Map::put;

        GlobalOpenTelemetry.getPropagators().getTextMapPropagator()
            .inject(Context.current(), headers, setter);
    }

    private void validateOrder(OrderEvent order) {
        // Simulate validation
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    private void processPayment(OrderEvent order) {
        // Simulate payment processing
        try {
            Thread.sleep(50);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

Operational Dashboards

Grafana Dashboard Configuration

```
{
  "dashboard": {
    "title": "PeeGeeQ Operations Dashboard",
    "panels": [
      {
        "title": "Message Throughput",
        "type": "graph",
        "targets": [
          {
            "expr": "rate(peegeeq_messages_sent_total[5m])",
            "legendFormat": "Messages Sent/sec"
          },
          {
            "expr": "rate(peegeeq_messages_received_total[5m])",
            "legendFormat": "Messages Received/sec"
          }
        ]
      },
      {
        "title": "Queue Depths",
        "type": "graph",
        "targets": [
          {
            "expr": "peegeeq_queue_depth",
            "legendFormat": "{{queue_name}}"
          }
        ]
      },
      {
        "title": "Processing Latency",
        "type": "graph",
        "targets": [
          {
            "expr": "histogram_quantile(0.50, rate(peegeeq_message_processing_duration_bucket[5m]))",
            "legendFormat": "50th percentile"
          },
          {
            "expr": "histogram_quantile(0.95, rate(peegeeq_message_processing_duration_bucket[5m]))",
            "legendFormat": "95th percentile"
          },
          {
            "expr": "histogram_quantile(0.99, rate(peegeeq_message_processing_duration_bucket[5m]))",
            "legendFormat": "99th percentile"
          }
        ]
      },
      {
        "title": "Error Rates",
        "type": "graph",
        "targets": [
          {
            "expr": "rate(peegeeq_errors_total[5m])",
            "legendFormat": "{{error_type}}"
          }
        ]
      },
      {
        "title": "Database Connections",
        "type": "graph",
        "targets": [
          {
            "expr": "peegeeq_database_connections_active",
            "legendFormat": "Active Connections"
          },
          {
            "expr": "peegeeq_database_connections_idle",
            "legendFormat": "Idle Connections"
          }
        ]
      },
      {
        "title": "System Health",
        "type": "stat",
        "targets": [
          {
            "expr": "peegeeq_health_check_status",
            "legendFormat": "{{component}}"
          }
        ]
      }
    ]
  }
}
```

🎯 **Try This Now**:

1. Set up Prometheus and Grafana for metrics collection
2. Run the monitoring example and observe metrics
3. Configure alerting rules for your specific use cases
4. Set up distributed tracing with Jaeger or Zipkin

Production Readiness Features

Health Checks

Comprehensive health monitoring across all components:

```java
public class PeeGeeQHealthChecks {
    private final HealthCheckManager healthCheckManager;

    public void configureHealthChecks() {
        // Database connectivity check
        healthCheckManager.registerHealthCheck("database", () -> {
            try {
                databaseService.query("SELECT 1", rs -> rs.getInt(1));
                return HealthCheckResult.healthy("Database connection OK");
            } catch (Exception e) {
                return HealthCheckResult.unhealthy("Database connection failed", e);
            }
        });

        // Queue processing check
        healthCheckManager.registerHealthCheck("queue-processing", () -> {
            long pendingMessages = getPendingMessageCount();
            if (pendingMessages > 10000) {
                return HealthCheckResult.unhealthy(
                    "High pending message count: " + pendingMessages);
            }
            return HealthCheckResult.healthy("Queue processing normal");
        });

        // Circuit breaker check
        healthCheckManager.registerHealthCheck("circuit-breakers", () -> {
            List<String> openCircuits = circuitBreakerManager.getOpenCircuits();
            if (!openCircuits.isEmpty()) {
                return HealthCheckResult.unhealthy(
                    "Open circuit breakers: " + String.join(", ", openCircuits));
            }
            return HealthCheckResult.healthy("All circuit breakers closed");
        });
    }
}
```

Circuit Breakers

Automatic failure handling and recovery:

```
@Component
public class CircuitBreakerConfiguration {

    @CircuitBreaker(name = "database-operations", fallbackMethod = "fallbackDatabaseOperation")
    @Retry(name = "database-operations")
    @TimeLimiter(name = "database-operations")
    public CompletableFuture<String> performDatabaseOperation(String operation) {
        return CompletableFuture.supplyAsync(() -> {
            // Potentially failing database operation
            return databaseService.executeOperation(operation);
        });
    }

    public CompletableFuture<String> fallbackDatabaseOperation(String operation, Exception ex) {
        log.warn("Database operation failed, using fallback: {}", ex.getMessage());
        return CompletableFuture.completedFuture("FALLBACK_RESULT");
    }

    @EventListener
    public void handleCircuitBreakerStateChange(CircuitBreakerOnStateTransitionEvent event) {
        log.info("Circuit breaker '{}' changed from {} to {}",
                event.getCircuitBreakerName(),
                event.getStateTransition().getFromState(),
                event.getStateTransition().getToState());

        // Send alerts for circuit breaker opening
        if (event.getStateTransition().getToState() == CircuitBreaker.State.OPEN) {
            alertingService.sendAlert(
                "Circuit breaker opened: " + event.getCircuitBreakerName());
        }
    }
}
```

Metrics Collection

Comprehensive metrics for monitoring and alerting:

```java
@Component
public class PeeGeeQMetrics {
    private final MeterRegistry meterRegistry;
    private final Counter messagesProduced;
    private final Counter messagesConsumed;
    private final Timer messageProcessingTime;
    private final Gauge queueDepth;

    public PeeGeeQMetrics(MeterRegistry meterRegistry) {
        this.meterRegistry = meterRegistry;
        this.messagesProduced = Counter.builder("peegeeq.messages.produced")
            .description("Total messages produced")
            .register(meterRegistry);
        this.messagesConsumed = Counter.builder("peegeeq.messages.consumed")
            .description("Total messages consumed")
            .register(meterRegistry);
        this.messageProcessingTime = Timer.builder("peegeeq.message.processing.time")
            .description("Message processing time")
            .register(meterRegistry);
        this.queueDepth = Gauge.builder("peegeeq.queue.depth")
            .description("Current queue depth")
            .register(meterRegistry, this, PeeGeeQMetrics::getCurrentQueueDepth);
    }

    public void recordMessageProduced(String queueName) {
        messagesProduced.increment(Tags.of("queue", queueName));
    }

    public void recordMessageConsumed(String queueName, Duration processingTime) {
        messagesConsumed.increment(Tags.of("queue", queueName));
        messageProcessingTime.record(processingTime, Tags.of("queue", queueName));
    }

    private double getCurrentQueueDepth() {
        return databaseService.query(
            "SELECT COUNT(*) FROM queue_messages WHERE processed_at IS NULL",
            rs -> rs.getLong(1)
        ).stream().findFirst().orElse(0L).doubleValue();
    }
}
```

Multi-Environment Configuration

Managing PeeGeeQ across multiple environments (development, staging, production) requires sophisticated configuration management strategies. This section demonstrates best practices for environment-specific configuration, secrets management, and deployment automation.

Environment-Specific Configuration Management

Configuration Hierarchy and Inheritance

```java
public class MultiEnvironmentConfigExample {

    public static void main(String[] args) throws Exception {
        // Demonstrate different environment configurations
        demonstrateEnvironmentConfigurations();

        // Show configuration inheritance
        demonstrateConfigurationInheritance();

        // Demonstrate secrets management
        demonstrateSecretsManagement();

        // Show configuration validation
        demonstrateConfigurationValidation();
    }

    private static void demonstrateEnvironmentConfigurations() throws Exception {
        System.out.println("=== Multi-Environment Configuration Example ===");

        // Development environment
        System.out.println("\n🔧 Development Environment:");
        PeeGeeQConfiguration devConfig = createDevelopmentConfig();
        printConfigurationSummary("Development", devConfig);

        // Staging environment
        System.out.println("\n🧪 Staging Environment:");
        PeeGeeQConfiguration stagingConfig = createStagingConfig();
        printConfigurationSummary("Staging", stagingConfig);

        // Production environment
        System.out.println("\n🚀 Production Environment:");
        PeeGeeQConfiguration prodConfig = createProductionConfig();
        printConfigurationSummary("Production", prodConfig);
    }

    private static PeeGeeQConfiguration createDevelopmentConfig() {
        return PeeGeeQConfiguration.builder()
            .profile("development")
            // Database settings - local development
            .host("localhost")
            .port(5432)
            .database("peegeeq_dev")
            .username("dev_user")
            .password("dev_password")
            // Performance settings - optimized for development
            .connectionPoolMinSize(2)
            .connectionPoolMaxSize(5)
            .queuePollingIntervalMs(1000)
            .batchSize(10)
            // Monitoring settings - basic monitoring
            .metricsEnabled(true)
            .healthChecksEnabled(true)
            .healthCheckIntervalSeconds(60)
            // Development-specific features
            .autoMigrationEnabled(true)
            .debugLoggingEnabled(true)
            .build();
    }

    private static PeeGeeQConfiguration createStagingConfig() {
        return PeeGeeQConfiguration.builder()
            .profile("staging")
            // Database settings - staging database
            .host("staging-db.company.com")
            .port(5432)
            .database("peegeeq_staging")
            .username(System.getenv("STAGING_DB_USER"))
            .password(System.getenv("STAGING_DB_PASSWORD"))
            // Performance settings - production-like
            .connectionPoolMinSize(5)
            .connectionPoolMaxSize(15)
            .queuePollingIntervalMs(500)
            .batchSize(25)
            // Monitoring settings - comprehensive monitoring
            .metricsEnabled(true)
            .healthChecksEnabled(true)
            .healthCheckIntervalSeconds(30)
            // SSL settings
            .sslEnabled(true)
            .sslMode("require")
            // Staging-specific features
            .autoMigrationEnabled(false) // Manual migration approval
            .debugLoggingEnabled(false)
            .build();
    }

    private static PeeGeeQConfiguration createProductionConfig() {
        return PeeGeeQConfiguration.builder()
            .profile("production")
            // Database settings - production cluster
            .host("prod-db-cluster.company.com")
            .port(5432)
            .database("peegeeq_prod")
            .username(System.getenv("PROD_DB_USER"))
            .password(System.getenv("PROD_DB_PASSWORD"))
            // Performance settings - optimized for production
            .connectionPoolMinSize(10)
            .connectionPoolMaxSize(50)
            .queuePollingIntervalMs(100)
            .batchSize(100)
            // Monitoring settings - full monitoring
            .metricsEnabled(true)
            .healthChecksEnabled(true)
            .healthCheckIntervalSeconds(15)
            // Security settings
            .sslEnabled(true)
            .sslMode("require")
            .sslCertPath("/etc/ssl/certs/peegeeq.crt")
            .sslKeyPath("/etc/ssl/private/peegeeq.key")
            // Production-specific features
            .autoMigrationEnabled(false) // Never auto-migrate in production
            .debugLoggingEnabled(false)
            .circuitBreakerEnabled(true)
            .retryMaxAttempts(5)
            .deadLetterQueueEnabled(true)
            .build();
    }

    private static void demonstrateConfigurationInheritance() {
        System.out.println("\n🏗️  Configuration Inheritance:");

        // Base configuration with common settings
        ConfigurationTemplate baseTemplate = ConfigurationTemplate.builder()
            .metricsEnabled(true)
            .healthChecksEnabled(true)
            .queueMaxRetries(3)
            .visibilityTimeoutSeconds(30)
            .build();

        // Environment-specific overrides
        ConfigurationTemplate devOverrides = ConfigurationTemplate.builder()
            .debugLoggingEnabled(true)
            .autoMigrationEnabled(true)
            .connectionPoolMaxSize(5)
            .build();

        ConfigurationTemplate prodOverrides = ConfigurationTemplate.builder()
            .circuitBreakerEnabled(true)
            .connectionPoolMaxSize(50)
            .sslEnabled(true)
            .build();

        // Merge configurations
        PeeGeeQConfiguration devConfig = baseTemplate.merge(devOverrides)
            .withProfile("development")
            .build();

        PeeGeeQConfiguration prodConfig = baseTemplate.merge(prodOverrides)
            .withProfile("production")
            .build();

        System.out.println("✅ Configuration inheritance allows:");
        System.out.println("  • Common settings in base template");
        System.out.println("  • Environment-specific overrides");
        System.out.println("  • Consistent configuration across environments");
    }

    private static void demonstrateSecretsManagement() {
        System.out.println("\n🔐 Secrets Management:");

        // Different secret sources for different environments
        SecretsManager devSecrets = new FileSecretsManager("dev-secrets.properties");
        SecretsManager stagingSecrets = new VaultSecretsManager("staging/peegeeq");
        SecretsManager prodSecrets = new VaultSecretsManager("production/peegeeq");

        // Configuration with secrets injection
        PeeGeeQConfiguration configWithSecrets = PeeGeeQConfiguration.builder()
            .profile("production")
            .host("prod-db.company.com")
            .database("peegeeq_prod")
            .username(prodSecrets.getSecret("database.username"))
            .password(prodSecrets.getSecret("database.password"))
            .sslCertPath(prodSecrets.getSecret("ssl.cert.path"))
            .sslKeyPath(prodSecrets.getSecret("ssl.key.path"))
            .build();

        System.out.println("🔑 Secrets management strategies:");
        System.out.println("  • Development: Local files (encrypted)");
        System.out.println("  • Staging: HashiCorp Vault");
        System.out.println("  • Production: HashiCorp Vault + rotation");
        System.out.println("  • Never store secrets in configuration files");
    }

    private static void demonstrateConfigurationValidation() {
        System.out.println("\n✅ Configuration Validation:");

        ConfigurationValidator validator = new ConfigurationValidator();

        // Validate development configuration
        PeeGeeQConfiguration devConfig = createDevelopmentConfig();
        ValidationResult devResult = validator.validate(devConfig);

        if (devResult.isValid()) {
            System.out.println("✅ Development configuration is valid");
        } else {
            System.out.println("❌ Development configuration errors:");
            devResult.getErrors().forEach(error ->
```

```java
                System.out.println("  • " + error));
        }

        // Validate production configuration
        PeeGeeQConfiguration prodConfig = createProductionConfig();
        ValidationResult prodResult = validator.validateForProduction(prodConfig);

        if (prodResult.isValid()) {
            System.out.println("✅ Production configuration is valid");
        } else {
            System.out.println("❌ Production configuration errors:");
            prodResult.getErrors().forEach(error ->
                System.out.println("  • " + error));
        }

        System.out.println("\n📋 Validation checks include:");
        System.out.println("  • Required properties are set");
        System.out.println("  • Connection pool sizes are reasonable");
        System.out.println("  • SSL is enabled for production");
        System.out.println("  • Auto-migration is disabled for production");
        System.out.println("  • Secrets are not hardcoded");
    }

    private static void printConfigurationSummary(String environment, PeeGeeQConfiguration config) {
        System.out.printf("📊 %s Configuration:%n", environment);
        System.out.printf("  • Database: %s:%d/%s%n",
            config.getHost(), config.getPort(), config.getDatabase());
        System.out.printf("  • Connection Pool: %d-%d connections%n",
            config.getConnectionPoolMinSize(), config.getConnectionPoolMaxSize());
        System.out.printf("  • Polling Interval: %dms%n", config.getQueuePollingIntervalMs());
        System.out.printf("  • SSL Enabled: %s%n", config.isSslEnabled());
        System.out.printf("  • Auto Migration: %s%n", config.isAutoMigrationEnabled());
        System.out.printf("  • Debug Logging: %s%n", config.isDebugLoggingEnabled());
    }
}
```

Configuration Templates and Profiles

Spring Boot Integration

```yaml
# application.yml - Base configuration
peegeeq:
  metrics:
    enabled: true
  health:
    enabled: true
    interval: 30s
  queue:
    max-retries: 3
    visibility-timeout: 30s

---
# application-development.yml
spring:
  profiles: development

peegeeq:
  database:
    host: localhost
    port: 5432
    name: peegeeq_dev
    username: dev_user
    password: dev_password
    pool:
      min-size: 2
      max-size: 5

  migration:
    auto-enabled: true

  logging:
    debug: true

---
# application-staging.yml
spring:
  profiles: staging

peegeeq:
  database:
    host: ${STAGING_DB_HOST}
    port: 5432
    name: peegeeq_staging
    username: ${STAGING_DB_USER}
    password: ${STAGING_DB_PASSWORD}
    pool:
      min-size: 5
      max-size: 15
    ssl:
      enabled: true
      mode: require

  migration:
    auto-enabled: false

  queue:
    polling-interval: 500ms
    batch-size: 25

---
# application-production.yml
spring:
  profiles: production

peegeeq:
  database:
    host: ${PROD_DB_HOST}
    port: 5432
    name: peegeeq_prod
    username: ${PROD_DB_USER}
    password: ${PROD_DB_PASSWORD}
    pool:
      min-size: 10
      max-size: 50
    ssl:
      enabled: true
      mode: require
      cert-path: ${SSL_CERT_PATH}
      key-path: ${SSL_KEY_PATH}

  migration:
    auto-enabled: false

  queue:
    polling-interval: 100ms
    batch-size: 100

  circuit-breaker:
    enabled: true
```

```
      failure-threshold: 5
      timeout: 60s

    dead-letter-queue:
      enabled: true
```

Docker and Kubernetes Configuration

Docker Compose for Multi-Environment

```
# docker-compose.yml
version: '3.8'

services:
  peegeeq-app:
    image: peegeeq-app:${VERSION:-latest}
    environment:
      - SPRING_PROFILES_ACTIVE=${ENVIRONMENT:-development}
      - PEEGEEQ_DB_HOST=${DB_HOST:-postgres}
      - PEEGEEQ_DB_USER=${DB_USER:-peegeeq}
      - PEEGEEQ_DB_PASSWORD=${DB_PASSWORD:-password}
    depends_on:
      - postgres
    ports:
      - "${APP_PORT:-8080}:8080"
    volumes:
      - ./config/${ENVIRONMENT:-development}:/app/config
      - ./logs:/app/logs

  postgres:
    image: postgres:15
    environment:
      - POSTGRES_DB=${DB_NAME:-peegeeq}
      - POSTGRES_USER=${DB_USER:-peegeeq}
      - POSTGRES_PASSWORD=${DB_PASSWORD:-password}
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./init-scripts:/docker-entrypoint-initdb.d
    ports:
      - "${DB_PORT:-5432}:5432"

volumes:
  postgres_data:
```

Kubernetes ConfigMaps and Secrets

```
# configmap-development.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: peegeeq-config-dev
  namespace: peegeeq-dev
data:
  application.yml: |
    peegeeq:
      database:
        host: postgres-dev
        port: 5432
        name: peegeeq_dev
        pool:
```

```yaml
          min-size: 2
          max-size: 5
      migration:
        auto-enabled: true
      logging:
        debug: true

---
# secret-development.yaml
apiVersion: v1
kind: Secret
metadata:
  name: peegeeq-secrets-dev
  namespace: peegeeq-dev
type: Opaque
data:
  database-username: ZGV2X3VzZXI=  # dev_user (base64)
  database-password: ZGV2X3Bhc3N3b3Jk  # dev_password (base64)

---
# configmap-production.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: peegeeq-config-prod
  namespace: peegeeq-prod
data:
  application.yml: |
    peegeeq:
      database:
        host: postgres-prod-cluster
        port: 5432
        name: peegeeq_prod
        pool:
          min-size: 10
          max-size: 50
        ssl:
          enabled: true
          mode: require
      migration:
        auto-enabled: false
      circuit-breaker:
        enabled: true
      dead-letter-queue:
        enabled: true

---
# secret-production.yaml
apiVersion: v1
kind: Secret
metadata:
  name: peegeeq-secrets-prod
  namespace: peegeeq-prod
type: Opaque
data:
  database-username: <encrypted-username>
  database-password: <encrypted-password>
  ssl-cert: <encrypted-ssl-cert>
  ssl-key: <encrypted-ssl-key>
```

Configuration Deployment Automation

Terraform Configuration Management

```
# environments/development/main.tf
module "peegeeq_development" {
  source = "../../modules/peegeeq"

  environment = "development"

  # Database configuration
  db_instance_class = "db.t3.micro"
  db_allocated_storage = 20
  db_backup_retention = 1

  # Application configuration
  app_instance_count = 1
  app_instance_type = "t3.small"

  # Monitoring
  enable_detailed_monitoring = false
  log_retention_days = 7

  # Security
  enable_ssl = false
  enable_encryption = false
}

# environments/production/main.tf
module "peegeeq_production" {
  source = "../../modules/peegeeq"

  environment = "production"

  # Database configuration
  db_instance_class = "db.r5.xlarge"
  db_allocated_storage = 500
  db_backup_retention = 30
  db_multi_az = true

  # Application configuration
  app_instance_count = 3
  app_instance_type = "c5.large"

  # Monitoring
  enable_detailed_monitoring = true
  log_retention_days = 90

  # Security
  enable_ssl = true
  enable_encryption = true

  # High availability
  enable_auto_scaling = true
  min_capacity = 2
  max_capacity = 10
}
```

🎯 **Try This Now**:

1. Set up different configuration files for each environment
2. Use environment variables for sensitive configuration
3. Implement configuration validation for production deployments
4. Set up automated deployment pipelines with environment-specific configurations

Performance Optimization

This section provides comprehensive guidance for optimizing PeeGeeQ performance across different workload patterns, from high-throughput batch processing to low-latency real-time messaging.

Performance Profiling and Benchmarking

Comprehensive Performance Testing Framework

```java
public class PerformanceOptimizationExample {
    private final PeeGeeQManager manager;
    private final PerformanceProfiler profiler;

    public static void main(String[] args) throws Exception {
        try (PeeGeeQManager manager = new PeeGeeQManager(
                createOptimizedConfiguration(), new PrometheusMeterRegistry())) {

            manager.start();

            PerformanceOptimizationExample optimizer =
                new PerformanceOptimizationExample(manager);
            optimizer.runPerformanceOptimization();
        }
    }

    public PerformanceOptimizationExample(PeeGeeQManager manager) {
        this.manager = manager;
        this.profiler = new PerformanceProfiler(manager.getMetrics());
    }

    public void runPerformanceOptimization() throws Exception {
        System.out.println("=== Performance Optimization Example ===");

        // 1. Baseline performance measurement
        measureBaselinePerformance();

        // 2. Connection pool optimization
        optimizeConnectionPool();

        // 3. Batch processing optimization
        optimizeBatchProcessing();

        // 4. Memory optimization
        optimizeMemoryUsage();

        // 5. Database query optimization
        optimizeDatabaseQueries();

        // 6. JVM optimization
        demonstrateJVMOptimizations();

        System.out.println("Performance optimization completed!");
    }

    private void measureBaselinePerformance() throws Exception {
        System.out.println("\n📊 Measuring Baseline Performance:");

        QueueFactoryProvider provider = new PgQueueFactoryProvider();
        QueueFactory factory = provider.createFactory("native",
            new PgDatabaseService(manager));
        // Test different message sizes
        int[] messageSizes = {100, 1000, 10000, 100000}; // bytes
        int[] messageCounts = {1000, 5000, 10000};

        for (int messageSize : messageSizes) {
            for (int messageCount : messageCounts) {
                PerformanceResult result = profiler.measureThroughput(
                    factory, messageSize, messageCount);

                System.out.printf("📈 %d messages (%d bytes): %.2f msg/sec, %.2fms avg latency%n",
                    messageCount, messageSize, result.getThroughput(), result.getAverageLatency());
            }
        }
    }

    private void optimizeConnectionPool() throws Exception {
        System.out.println("\n🏊 Connection Pool Optimization:");

        // Test different pool configurations
        PoolConfiguration[] configs = {
            new PoolConfiguration(5, 10, 30),    // Conservative
            new PoolConfiguration(10, 20, 30),   // Balanced
            new PoolConfiguration(20, 50, 30),   // Aggressive
            new PoolConfiguration(50, 100, 30)   // High-throughput
        };

        for (PoolConfiguration config : configs) {
            PeeGeeQConfiguration optimizedConfig = createOptimizedConfiguration()
                .withConnectionPoolMinSize(config.minSize)
                .withConnectionPoolMaxSize(config.maxSize)
                .withConnectionTimeoutSeconds(config.timeoutSeconds);

            try (PeeGeeQManager testManager = new PeeGeeQManager(optimizedConfig)) {
                testManager.start();

                PerformanceResult result = profiler.measureConnectionPoolPerformance(testManager);

                System.out.printf("🔗 Pool %d-%d: %.2f msg/sec, %d active connections%n",
                    config.minSize, config.maxSize, result.getThroughput(),
                    result.getActiveConnections());
            }
        }

        System.out.println("💡 Optimal pool size depends on:");
        System.out.println("  • CPU cores (typically 2-4x core count)");
        System.out.println("  • Database connection limits");
        System.out.println("  • Message processing time");
        System.out.println("  • Concurrent consumer count");
    }

    private void optimizeBatchProcessing() throws Exception {
        System.out.println("\n📦 Batch Processing Optimization:");

        QueueFactoryProvider provider = new PgQueueFactoryProvider();
        QueueFactory factory = provider.createFactory("outbox",
            new PgDatabaseService(manager));

        // Test different batch sizes
        int[] batchSizes = {1, 10, 50, 100, 500, 1000};

        for (int batchSize : batchSizes) {
            PerformanceResult result = profiler.measureBatchPerformance(
                factory, batchSize, 10000);

            System.out.printf("📊 Batch size %d: %.2f msg/sec, %.2fms latency%n",
                batchSize, result.getThroughput(), result.getAverageLatency());
        }

        System.out.println("\n🎯 Batch Size Guidelines:");
        System.out.println("  • Small batches (1-10): Low latency, higher CPU overhead");
        System.out.println("  • Medium batches (50-100): Balanced performance");
        System.out.println("  • Large batches (500+): High throughput, higher latency");
        System.out.println("  • Consider message size and processing time");
    }

    private void optimizeMemoryUsage() throws Exception {
        System.out.println("\n🧠 Memory Usage Optimization:");

        MemoryProfiler memProfiler = new MemoryProfiler();

        // Measure memory usage with different configurations
        System.out.println("📊 Memory usage patterns:");

        // Test with different message retention policies
        testMemoryWithRetention(memProfiler, "No retention", 0);
        testMemoryWithRetention(memProfiler, "1 hour retention", 3600);
        testMemoryWithRetention(memProfiler, "24 hour retention", 86400);

        // Test with different serialization strategies
        testMemoryWithSerialization(memProfiler);

        System.out.println("\n💡 Memory Optimization Tips:");
        System.out.println("  • Use appropriate message retention policies");
        System.out.println("  • Consider message compression for large payloads");
        System.out.println("  • Implement message archival for old data");
        System.out.println("  • Monitor heap usage and GC patterns");
    }

    private void optimizeDatabaseQueries() throws Exception {
        System.out.println("\n🗄️ Database Query Optimization:");

        DatabaseOptimizer dbOptimizer = new DatabaseOptimizer(manager.getDataSource());

        // Analyze current query performance
        QueryPerformanceReport report = dbOptimizer.analyzeQueryPerformance();

        System.out.println("📊 Query Performance Analysis:");
        report.getSlowQueries().forEach(query -> {
            System.out.printf("  🐌 %s: %.2fms avg, %d executions%n",
                query.getQueryType(), query.getAverageTime(), query.getExecutionCount());
        });

        // Apply optimizations
        System.out.println("\n🚀 Applying Database Optimizations:");

        // Create optimized indexes
        dbOptimizer.createOptimizedIndexes();
        System.out.println("  ✅ Created specialized indexes");

        // Update table statistics
        dbOptimizer.updateTableStatistics();
        System.out.println("  ✅ Updated table statistics");

        // Configure connection pool for database
        dbOptimizer.optimizeConnectionPool();
        System.out.println("  ✅ Optimized connection pool settings");

        // Measure improvement
        QueryPerformanceReport improvedReport = dbOptimizer.analyzeQueryPerformance();
        double improvement = calculateImprovement(report, improvedReport);

        System.out.printf("📈 Overall query performance improved by %.1f%%n", improvement);
    }

    private void demonstrateJVMOptimizations() {
        System.out.println("\n☕ JVM Optimization Recommendations:");

        System.out.println("🚀 High-Throughput JVM Settings:");
        System.out.println("  -Xms4g -Xmx4g                     # Fixed heap size");
        System.out.println("  -XX:+UseG1GC                      # G1 garbage collector");
        System.out.println("  -XX:MaxGCPauseMillis=200           # Target GC pause time");
        System.out.println("  -XX:G1HeapRegionSize=16m           # G1 region size");
        System.out.println("  -XX:+UseStringDeduplication        # Reduce string memory");

        System.out.println("\n⚡ Low-Latency JVM Settings:");
        System.out.println("  -Xms8g -Xmx8g                     # Larger fixed heap");
        System.out.println("  -XX:+UnlockExperimentalVMOptions   # Enable experimental features");
        System.out.println("  -XX:+UseZGC                        # ZGC for ultra-low latency");
        System.out.println("  -XX:+UseLargePages                 # Large pages for better memory management");
        System.out.println("  -XX:+AlwaysPreTouch                # Pre-touch memory pages");

        System.out.println("\n📊 Monitoring JVM Settings:");
        System.out.println("  -XX:+PrintGC                       # Print GC information");
        System.out.println("  -XX:+PrintGCDetails                # Detailed GC information");
        System.out.println("  -XX:+PrintGCTimeStamps             # GC timestamps");
        System.out.println("  -XX:+UseGCLogFileRotation          # Rotate GC logs");
        System.out.println("  -Xloggc:gc.log                     # GC log file");
    }

    private static PeeGeeQConfiguration createOptimizedConfiguration() {
        return PeeGeeQConfiguration.builder()
            .profile("performance-optimized")
            // Database optimizations
            .connectionPoolMinSize(20)
            .connectionPoolMaxSize(50)
            .connectionTimeoutSeconds(30)
            .connectionIdleTimeoutSeconds(600)
            // Queue optimizations
            .queuePollingIntervalMs(50)
            .batchSize(100)
            .maxRetries(3)
            .visibilityTimeoutSeconds(30)
            // Performance optimizations
            .enableConnectionPoolMetrics(true)
            .enableQueryMetrics(true)
            .enableJvmMetrics(true)
            // Caching optimizations
            .enableQueryResultCaching(true)
            .queryCacheTtlSeconds(300)
            .build();
    }

    private void testMemoryWithRetention(MemoryProfiler profiler, String description, int retentionSeconds) {
        MemoryUsage usage = profiler.measureMemoryUsage(retentionSeconds);
        System.out.printf("  📊 %s: %.2f MB heap, %.2f MB off-heap%n",
            description, usage.getHeapUsageMB(), usage.getOffHeapUsageMB());
    }

    private void testMemoryWithSerialization(MemoryProfiler profiler) {
        System.out.println("  🔄 Serialization strategies:");

        SerializationStrategy[] strategies = {
            SerializationStrategy.JSON,
            SerializationStrategy.BINARY,
            SerializationStrategy.COMPRESSED_JSON,
            SerializationStrategy.AVRO
        };
        for (SerializationStrategy strategy : strategies) {
            MemoryUsage usage = profiler.measureSerializationMemory(strategy);
            System.out.printf("    • %s: %.2f MB, %.1fx compression%n",
                strategy.name(), usage.getHeapUsageMB(), usage.getCompressionRatio());
        }
    }

    private double calculateImprovement(QueryPerformanceReport before, QueryPerformanceReport after) {
        double beforeAvg = before.getSlowQueries().stream()
            .mapToDouble(QueryStats::getAverageTime)
            .average()
            .orElse(0.0);

        double afterAvg = after.getSlowQueries().stream()
            .mapToDouble(QueryStats::getAverageTime)
            .average()
            .orElse(0.0);

        return ((beforeAvg - afterAvg) / beforeAvg) * 100;
    }
}

// Supporting classes for performance optimization
class PoolConfiguration {
    final int minSize;
    final int maxSize;
    final int timeoutSeconds;

    public PoolConfiguration(int minSize, int maxSize, int timeoutSeconds) {
        this.minSize = minSize;
        this.maxSize = maxSize;
        this.timeoutSeconds = timeoutSeconds;
    }
}

class PerformanceResult {
    private final double throughput;
    private final double averageLatency;
    private final int activeConnections;

    public PerformanceResult(double throughput, double averageLatency, int activeConnections) {
        this.throughput = throughput;
        this.averageLatency = averageLatency;
        this.activeConnections = activeConnections;
    }

    public double getThroughput() { return throughput; }
    public double getAverageLatency() { return averageLatency; }
    public int getActiveConnections() { return activeConnections; }
}
```

Workload-Specific Optimizations

High-Throughput Batch Processing

```
public class HighThroughputOptimization {

    public void optimizeForHighThroughput() throws Exception {
        System.out.println("=== High-Throughput Optimization ===");

        // Configuration for maximum throughput
        PeeGeeQConfiguration config = PeeGeeQConfiguration.builder()
            .profile("high-throughput")
```

```java
            // Aggressive connection pooling
            .connectionPoolMinSize(50)
            .connectionPoolMaxSize(200)
            .connectionAcquisitionTimeoutMs(5000)
            // Large batch sizes
            .batchSize(1000)
            .queuePollingIntervalMs(10) // Very frequent polling
            // Optimized timeouts
            .visibilityTimeoutSeconds(60)
            .messageRetentionHours(1) // Short retention for high volume
            // Disable features that add overhead
            .enableDetailedMetrics(false)
            .enableDebugLogging(false)
            .build();

        try (PeeGeeQManager manager = new PeeGeeQManager(config)) {
            manager.start();

            demonstrateHighThroughputProcessing(manager);
        }
    }

    private void demonstrateHighThroughputProcessing(PeeGeeQManager manager) throws Exception {
        QueueFactoryProvider provider = new PgQueueFactoryProvider();
        QueueFactory factory = provider.createFactory("outbox",
            new PgDatabaseService(manager));

        // Create multiple producers for parallel sending
        int producerCount = Runtime.getRuntime().availableProcessors();
        List<MessageProducer<BatchMessage>> producers = new ArrayList<>();

        for (int i = 0; i < producerCount; i++) {
            producers.add(factory.createProducer("high-throughput-queue", BatchMessage.class));
        }

        // Create consumer group for parallel processing
        ConsumerGroup<BatchMessage> consumerGroup = factory.createConsumerGroup(
            "high-throughput-group", "high-throughput-queue", BatchMessage.class);

        // Add multiple consumers to the group
        for (int i = 0; i < producerCount * 2; i++) {
            consumerGroup.addConsumer(this::processBatchMessage);
        }

        consumerGroup.start();

        // Send messages in parallel
        System.out.println("🚀 Starting high-throughput message sending...");
        long startTime = System.currentTimeMillis();

        List<CompletableFuture<Void>> sendTasks = new ArrayList<>();
        int messagesPerProducer = 10000;

        for (int i = 0; i < producerCount; i++) {
            final int producerId = i;
            MessageProducer<BatchMessage> producer = producers.get(i);

            CompletableFuture<Void> task = CompletableFuture.runAsync(() -> {
                try {
                    for (int j = 0; j < messagesPerProducer; j++) {
                        BatchMessage message = new BatchMessage(
                            "BATCH-" + producerId + "-" + j,
                            "High throughput message " + j,
                            System.currentTimeMillis()
                        );
                        producer.send(message).join();
                    }
                } catch (Exception e) {
                    throw new RuntimeException(e);
                }
            });

            sendTasks.add(task);
        }

        // Wait for all sends to complete
        CompletableFuture.allOf(sendTasks.toArray(new CompletableFuture[0])).join();

        long sendTime = System.currentTimeMillis() - startTime;
        int totalMessages = producerCount * messagesPerProducer;
        double throughput = (totalMessages * 1000.0) / sendTime;

        System.out.printf("📊 Sent %d messages in %dms (%.2f msg/sec)%n",
            totalMessages, sendTime, throughput);

        // Wait for processing to complete
        Thread.sleep(5000);

        // Get processing statistics
        ConsumerGroupStats stats = consumerGroup.getStats();
        System.out.printf("📈 Processed %d messages (%.2f msg/sec average)%n",
            stats.getMessagesProcessed(), stats.getAverageProcessingRate());

        consumerGroup.stop();
        producers.forEach(producer -> {
            try { producer.close(); } catch (Exception e) { /* ignore */ }
        });
    }

    private CompletableFuture<Void> processBatchMessage(Message<BatchMessage> message) {
        // Minimal processing for maximum throughput
        BatchMessage batchMessage = message.getPayload();

        // Just acknowledge - in real scenario, you'd do actual processing
        return CompletableFuture.completedFuture(null);
    }
}
```

Low-Latency Real-Time Processing

```
public class LowLatencyOptimization {

    public void optimizeForLowLatency() throws Exception {
        System.out.println("=== Low-Latency Optimization ===");

        // Configuration for minimum latency
        PeeGeeQConfiguration config = PeeGeeQConfiguration.builder()
            .profile("low-latency")
            // Dedicated connections for immediate processing
            .connectionPoolMinSize(10)
            .connectionPoolMaxSize(20)
            .connectionAcquisitionTimeoutMs(100)
            // Small batch sizes for immediate processing
            .batchSize(1)
            .queuePollingIntervalMs(1) // Extremely frequent polling
            // Minimal timeouts
            .visibilityTimeoutSeconds(5)
            .messageRetentionHours(24)
            // Enable features for latency monitoring
```

```java
            .enableLatencyMetrics(true)
            .enableDetailedTracing(true)
            .build();

        try (PeeGeeQManager manager = new PeeGeeQManager(config)) {
            manager.start();

            demonstrateLowLatencyProcessing(manager);
        }
    }

    private void demonstrateLowLatencyProcessing(PeeGeeQManager manager) throws Exception {
        QueueFactoryProvider provider = new PgQueueFactoryProvider();
        QueueFactory factory = provider.createFactory("native", // Native for lowest latency
            new PgDatabaseService(manager));

        MessageProducer<LatencyMessage> producer =
            factory.createProducer("low-latency-queue", LatencyMessage.class);
        MessageConsumer<LatencyMessage> consumer =
            factory.createConsumer("low-latency-queue", LatencyMessage.class);

        // Track latency statistics
        LatencyTracker latencyTracker = new LatencyTracker();

        consumer.subscribe(message -> {
            long receiveTime = System.nanoTime();
            LatencyMessage latencyMessage = message.getPayload();

            long latencyNanos = receiveTime - latencyMessage.getSendTime();
            double latencyMs = latencyNanos / 1_000_000.0;

            latencyTracker.recordLatency(latencyMs);

            // Minimal processing for low latency
            return CompletableFuture.completedFuture(null);
        });

        // Send messages and measure latency
        System.out.println("⚡ Starting low-latency message processing...");

        for (int i = 0; i < 1000; i++) {
            LatencyMessage message = new LatencyMessage(
                "LATENCY-" + i,
                "Low latency message " + i,
                System.nanoTime()
            );

            producer.send(message).join();

            // Small delay to avoid overwhelming the system
            Thread.sleep(10);
        }

        // Wait for processing to complete
        Thread.sleep(2000);

        // Print latency statistics
        LatencyStats stats = latencyTracker.getStats();
        System.out.printf("📊 Latency Statistics:%n");
        System.out.printf("  • Average: %.2fms%n", stats.getAverage());
        System.out.printf("  • Median: %.2fms%n", stats.getMedian());
        System.out.printf("  • 95th percentile: %.2fms%n", stats.getP95());
        System.out.printf("  • 99th percentile: %.2fms%n", stats.getP99());
        System.out.printf("  • Maximum: %.2fms%n", stats.getMax());

        consumer.close();
```

```
            producer.close();
        }
    }
```

🎯 **Try This Now**:

1. Run performance benchmarks with different configurations
2. Optimize connection pool settings for your workload
3. Experiment with different batch sizes and polling intervals
4. Monitor JVM performance and tune garbage collection settings

Integration Patterns

This section demonstrates enterprise integration patterns using PeeGeeQ, including message routing, transformation, aggregation, and integration with external systems and message brokers. These patterns enable building robust, scalable distributed systems with clear separation of concerns and maintainable architectures.

Enterprise Integration Patterns

Enterprise Integration Patterns (EIP) provide proven solutions for common messaging challenges in distributed systems. PeeGeeQ implements these patterns using PostgreSQL as the reliable message transport, ensuring ACID compliance and durability while maintaining high performance.

Message Router Pattern

The **Message Router Pattern** enables intelligent message routing based on message content, headers, or other criteria. This pattern is essential for building event-driven architectures where different message types need to be processed by specialized handlers.

Key Benefits:

- **Content-based routing** - Route messages based on payload or headers
- **Dynamic routing rules** - Add/modify routing logic without code changes
- **Load distribution** - Distribute messages across multiple processing queues
- **Fault isolation** - Route problematic messages to dedicated error handling queues

Use Cases:

- Order processing systems (route by order type, priority, customer tier)
- Event sourcing architectures (route events to appropriate aggregates)
- Multi-tenant systems (route by tenant ID)
- A/B testing scenarios (route by experiment group)

```java
public class MessageRouterPatternExample {
    private final QueueFactory factory;
    private final Map<String, MessageConsumer<BusinessMessage>> destinationConsumers = new HashMap<>();
    private MessageConsumer<BusinessMessage> routerConsumer;

    public static void main(String[] args) throws Exception {
        // Initialize PeeGeeQ
        PeeGeeQManager manager = new PeeGeeQManager();
        manager.start();
        // Create factory using the correct API
        QueueFactoryProvider provider = QueueFactoryProvider.getInstance();
        QueueFactory factory = provider.createFactory("native", manager.getDatabaseService());

        MessageRouterPatternExample example = new MessageRouterPatternExample(factory);
        example.runMessageRouterExample();

        // Cleanup
        example.close();
        manager.stop();
    }

    public MessageRouterPatternExample(QueueFactory factory) {
        this.factory = factory;
    }

    public void runMessageRouterExample() throws Exception {
        System.out.println("=== Message Router Pattern Example ===");

        // Setup destination consumers first
        setupDestinationConsumers();

        // Setup the main router consumer
        setupRouterConsumer();

        // Send messages that will be routed
        sendRoutedMessages();

        // Wait for processing
        Thread.sleep(3000);
        System.out.println("Message router pattern example completed!");
    }

    private void setupRouterConsumer() throws Exception {
        System.out.println("🔀 Setting up message router:");

        // Create consumer for incoming messages
        routerConsumer = factory.createConsumer("incoming-messages", BusinessMessage.class);

        // Subscribe with routing logic
        routerConsumer.subscribe(message -> {
            try {
                String destination = determineDestination(message);
                routeMessage(message, destination);
                return CompletableFuture.completedFuture(null);
            } catch (Exception e) {
                System.err.println("❌ Routing failed: " + e.getMessage());
                return CompletableFuture.failedFuture(e);
            }
        });

        System.out.println("✅ Message router configured and started");
    }

    private String determineDestination(Message<BusinessMessage> message) {
        Map<String, String> headers = message.getHeaders();

        // Route by message type (highest priority)
        if ("ORDER".equals(headers.get("messageType"))) {
            return "order-processing-queue";
        }

        // Route by priority
        if ("HIGH".equals(headers.get("priority"))) {
            return "high-priority-queue";
        }
        // Route by customer tier
        if ("PREMIUM".equals(headers.get("customerTier"))) {
            return "premium-customer-queue";
        }

        // Default route
        return "default-processing-queue";
    }

    private void routeMessage(Message<BusinessMessage> message, String destination) throws Exception {
        MessageProducer<BusinessMessage> producer = factory.createProducer(destination, BusinessMessage.class);

        // Forward the message to the destination queue
        producer.send(
            message.getPayload(),
            message.getHeaders()
        ).join();

        System.out.printf("📤 Routed message %s to %s%n", message.getId(), destination);
        producer.close();
    }

    private void setupDestinationConsumers() throws Exception {
        System.out.println("🎯 Setting up destination consumers:");

        // Order processing consumer
        MessageConsumer<BusinessMessage> orderConsumer =
            factory.createConsumer("order-processing-queue", BusinessMessage.class);
        orderConsumer.subscribe(message -> {
            System.out.printf("📦 Order Processing: %s%n", message.getPayload().getContent());
            return CompletableFuture.completedFuture(null);
        });
        destinationConsumers.put("order-processing-queue", orderConsumer);

        // High priority consumer
        MessageConsumer<BusinessMessage> priorityConsumer =
            factory.createConsumer("high-priority-queue", BusinessMessage.class);
        priorityConsumer.subscribe(message -> {
            System.out.printf("🚨 High Priority: %s%n", message.getPayload().getContent());
            return CompletableFuture.completedFuture(null);
        });
        destinationConsumers.put("high-priority-queue", priorityConsumer);

        // Premium customer consumer
        MessageConsumer<BusinessMessage> premiumConsumer =
            factory.createConsumer("premium-customer-queue", BusinessMessage.class);
        premiumConsumer.subscribe(message -> {
            System.out.printf("⭐ Premium Customer: %s%n", message.getPayload().getContent());
            return CompletableFuture.completedFuture(null);
        });
        destinationConsumers.put("premium-customer-queue", premiumConsumer);

        // Default consumer
        MessageConsumer<BusinessMessage> defaultConsumer =
            factory.createConsumer("default-processing-queue", BusinessMessage.class);
        defaultConsumer.subscribe(message -> {
            System.out.printf("📋 Default Processing: %s%n", message.getPayload().getContent());
            return CompletableFuture.completedFuture(null);
        });
        destinationConsumers.put("default-processing-queue", defaultConsumer);

        System.out.println("✅ All destination consumers configured");
    }

    private void sendRoutedMessages() throws Exception {
        System.out.println("📤 Sending messages for routing:");

        MessageProducer<BusinessMessage> producer =
            factory.createProducer("incoming-messages", BusinessMessage.class);

        // Send order message
        producer.send(
            new BusinessMessage("ORDER-001", "New order from customer"),
            Map.of("messageType", "ORDER", "customerId", "CUST-123")
        ).join();
        System.out.println("  📦 Sent ORDER message");

        // Send high priority message
        producer.send(
            new BusinessMessage("ALERT-001", "System alert message"),
            Map.of("priority", "HIGH", "alertType", "SYSTEM")
        ).join();
        System.out.println("  🚨 Sent HIGH priority message");

        // Send premium customer message
        producer.send(
            new BusinessMessage("PREMIUM-001", "Premium customer request"),
            Map.of("customerTier", "PREMIUM", "customerId", "CUST-456")
        ).join();
        System.out.println("  ⭐ Sent PREMIUM customer message");

        // Send message that matches multiple rules (first match wins)
        producer.send(
            new BusinessMessage("ORDER-002", "Premium customer order"),
            Map.of("messageType", "ORDER", "customerTier", "PREMIUM", "priority", "HIGH")
        ).join();
        System.out.println("  📦 Sent ORDER message (with multiple routing criteria)");

        // Send message that goes to default route
        producer.send(
            new BusinessMessage("MISC-001", "Miscellaneous message"),
            Map.of("category", "general")
        ).join();
        System.out.println("  📋 Sent message for default routing");

        producer.close();
        System.out.println("✅ All routing messages sent");
    }

    public void close() throws Exception {
        // Close router consumer
        if (routerConsumer != null) {
            routerConsumer.close();
        }

        // Close all destination consumers
        for (MessageConsumer<BusinessMessage> consumer : destinationConsumers.values()) {
            consumer.close();
        }

        // Close factory
        factory.close();
    }

    // Simple BusinessMessage class for the example
    public static class BusinessMessage {
        private final String id;
        private final String content;

        public BusinessMessage(String id, String content) {
            this.id = id;
```

```java
            this.content = content;
        }

        public String getId() { return id; }
        public String getContent() { return content; }
    }
}
```

Message Aggregator Pattern

The **Message Aggregator Pattern** collects related messages and combines them into a single composite message. This pattern is crucial for scenarios where you need to gather multiple related messages before processing them as a group.

Key Benefits:

- **Batch processing** - Process related messages together for efficiency
- **Data correlation** - Combine messages based on correlation keys
- **Timeout handling** - Complete aggregation after time limits
- **Memory efficiency** - Stream processing without loading all messages into memory

Use Cases:

- Order processing (aggregate all order items before fulfillment)
- Sensor data collection (aggregate readings by time windows)
- Financial transactions (aggregate by account or time period)
- Log aggregation (combine log entries by service or time)

```java
public class MessageAggregatorPatternExample {
    private final QueueFactory factory;
    private final Map<String, List<Message<BusinessMessage>>> aggregationBuffers = new ConcurrentHashMap<>();
    private final Map<String, ScheduledFuture<?>> timeoutTasks = new ConcurrentHashMap<>();
    private final ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(2);
    private MessageConsumer<BusinessMessage> aggregatorConsumer;

    public static void main(String[] args) throws Exception {
        // Initialize PeeGeeQ
        PeeGeeQManager manager = new PeeGeeQManager();
        manager.start();

        QueueFactoryProvider provider = QueueFactoryProvider.getInstance();
        QueueFactory factory = provider.createFactory("native", manager.getDatabaseService());

        MessageAggregatorPatternExample example = new MessageAggregatorPatternExample(factory);
        example.runMessageAggregatorExample();

        example.close();
        manager.stop();
    }

    public MessageAggregatorPatternExample(QueueFactory factory) {
        this.factory = factory;
    }

    public void runMessageAggregatorExample() throws Exception {
        System.out.println("=== Message Aggregator Pattern Example ===");

        // Setup aggregated message consumers first
        setupAggregatedConsumers();
        // Setup the main aggregator consumer
        setupAggregatorConsumer();

        // Send messages to be aggregated
        sendMessagesForAggregation();

        // Wait for processing
        Thread.sleep(8000);
        System.out.println("Message aggregator pattern example completed!");
    }

    private void setupAggregatorConsumer() throws Exception {
        System.out.println("🔄 Setting up message aggregator:");

        // Create consumer for messages to aggregate
        aggregatorConsumer = factory.createConsumer("messages-to-aggregate", BusinessMessage.class);

        // Subscribe with aggregation logic
        aggregatorConsumer.subscribe(message -> {
            try {
                processMessageForAggregation(message);
                return CompletableFuture.completedFuture(null);
            } catch (Exception e) {
                System.err.println("❌ Aggregation failed: " + e.getMessage());
                return CompletableFuture.failedFuture(e);
            }
        });

        System.out.println("✅ Message aggregator configured and started");
    }

    private void processMessageForAggregation(Message<BusinessMessage> message) throws Exception {
        Map<String, String> headers = message.getHeaders();
        String correlationKey = determineCorrelationKey(message);

        if (correlationKey == null) {
            System.out.println("⚠️  No correlation key found, skipping aggregation");
            return;
        }

        // Add message to aggregation buffer
        aggregationBuffers.computeIfAbsent(correlationKey, k -> new ArrayList<>()).add(message);

        // Check if aggregation is complete
        if (isAggregationComplete(correlationKey)) {
            completeAggregation(correlationKey);
        } else {
            // Set timeout if not already set
            timeoutTasks.computeIfAbsent(correlationKey, k ->
                scheduler.schedule(() -> {
                    try {
                        completeAggregation(correlationKey);
                    } catch (Exception e) {
                        System.err.println("❌ Timeout aggregation failed: " + e.getMessage());
                    }
                }, getTimeoutSeconds(message), TimeUnit.SECONDS)
            );
        }
    }

    private String determineCorrelationKey(Message<BusinessMessage> message) {
        Map<String, String> headers = message.getHeaders();

        // Check for order ID (order aggregation)
        if (headers.containsKey("orderId")) {
            return "order:" + headers.get("orderId");
```

```java
        }

        // Check for timestamp (sensor data aggregation)
        if (headers.containsKey("timestamp")) {
            return "sensor:" + getTimeWindow(headers.get("timestamp"));
        }

        return null;
    }

    private boolean isAggregationComplete(String correlationKey) {
        List<Message<BusinessMessage>> messages = aggregationBuffers.get(correlationKey);
        if (messages == null || messages.isEmpty()) {
            return false;
        }

        // For order aggregation - check if we have all expected items
        if (correlationKey.startsWith("order:")) {
            String expectedCount = messages.get(0).getHeaders().get("totalItems");
            if (expectedCount != null) {
                return messages.size() >= Integer.parseInt(expectedCount);
            }
        }

        // For sensor aggregation - aggregate every 10 readings
        if (correlationKey.startsWith("sensor:")) {
            return messages.size() >= 10;
        }

        return false;
    }

    private int getTimeoutSeconds(Message<BusinessMessage> message) {
        Map<String, String> headers = message.getHeaders();

        // Order aggregation timeout
        if (headers.containsKey("orderId")) {
            return 30;
        }

        // Sensor aggregation timeout
        if (headers.containsKey("timestamp")) {
            return 60;
        }

        return 30; // Default timeout
    }

    private void completeAggregation(String correlationKey) throws Exception {
        List<Message<BusinessMessage>> messages = aggregationBuffers.remove(correlationKey);
        ScheduledFuture<?> timeoutTask = timeoutTasks.remove(correlationKey);

        if (timeoutTask != null) {
            timeoutTask.cancel(false);
        }

        if (messages == null || messages.isEmpty()) {
            return;
        }

        System.out.printf("🔄 Completing aggregation for %s (%d messages)%n", correlationKey, messages.size());

        // Create aggregated message based on type
        if (correlationKey.startsWith("order:")) {
            sendAggregatedOrder(messages);
        } else if (correlationKey.startsWith("sensor:")) {
            sendAggregatedSensorData(messages);
        }
    }

    private void setupAggregatedConsumers() throws Exception {
        System.out.println("🎯 Setting up aggregated message consumers:");

        // Consumer for aggregated orders
        MessageConsumer<AggregatedMessage> orderConsumer =
            factory.createConsumer("aggregated-orders", AggregatedMessage.class);
        orderConsumer.subscribe(message -> {
            AggregatedMessage aggregated = message.getPayload();
            System.out.printf("📦 Aggregated Order: %s (%d items, total: $%.2f)%n",
                aggregated.getCorrelationId(),
                aggregated.getMessageCount(),
                aggregated.getTotalAmount());
            return CompletableFuture.completedFuture(null);
        });

        // Consumer for aggregated sensor data
        MessageConsumer<AggregatedMessage> sensorConsumer =
            factory.createConsumer("aggregated-sensor-data", AggregatedMessage.class);
        sensorConsumer.subscribe(message -> {
            AggregatedMessage aggregated = message.getPayload();
            System.out.printf("📊 Aggregated Sensor Data: %s (avg: %.2f, min: %.2f, max: %.2f)%n",
                aggregated.getCorrelationId(),
                aggregated.getAverageValue(),
                aggregated.getMinValue(),
                aggregated.getMaxValue());
            return CompletableFuture.completedFuture(null);
        });

        System.out.println("✅ Aggregated message consumers configured");
    }

    private void sendAggregatedOrder(List<Message<BusinessMessage>> messages) throws Exception {
        String orderId = messages.get(0).getHeaders().get("orderId");
        double totalAmount = messages.stream()
            .mapToDouble(msg -> Double.parseDouble(msg.getHeaders().get("amount")))
            .sum();

        AggregatedMessage aggregated = new AggregatedMessage(
            orderId, messages.size(), totalAmount, 0, 0, 0
        );

        MessageProducer<AggregatedMessage> producer =
            factory.createProducer("aggregated-orders", AggregatedMessage.class);
        producer.send(aggregated).join();
        producer.close();
    }

    private void sendAggregatedSensorData(List<Message<BusinessMessage>> messages) throws Exception {
        String timeWindow = getTimeWindow(messages.get(0).getHeaders().get("timestamp"));

        double[] values = messages.stream()
            .mapToDouble(msg -> Double.parseDouble(msg.getHeaders().get("value")))
            .toArray();

        double average = Arrays.stream(values).average().orElse(0.0);
        double min = Arrays.stream(values).min().orElse(0.0);
        double max = Arrays.stream(values).max().orElse(0.0);

        AggregatedMessage aggregated = new AggregatedMessage(
            timeWindow, messages.size(), 0, average, min, max
        );
        MessageProducer<AggregatedMessage> producer =
            factory.createProducer("aggregated-sensor-data", AggregatedMessage.class);
        producer.send(aggregated).join();
        producer.close();
    }

    private void sendMessagesForAggregation() throws Exception {
        System.out.println("📤 Sending messages for aggregation:");

        MessageProducer<BusinessMessage> producer =
            factory.createProducer("messages-to-aggregate", BusinessMessage.class);

        // Send order items for aggregation
        String orderId = "ORDER-001";
        producer.send(
            new BusinessMessage("ITEM-1", "Laptop - $999.99"),
            Map.of("orderId", orderId, "totalItems", "3", "amount", "999.99")
        ).join();
        System.out.println("  📦 Sent order item 1");

        producer.send(
            new BusinessMessage("ITEM-2", "Mouse - $29.99"),
            Map.of("orderId", orderId, "totalItems", "3", "amount", "29.99")
        ).join();
        System.out.println("  📦 Sent order item 2");

        producer.send(
            new BusinessMessage("ITEM-3", "Keyboard - $79.99"),
            Map.of("orderId", orderId, "totalItems", "3", "amount", "79.99")
        ).join();
        System.out.println("  📦 Sent order item 3");

        // Send sensor readings for aggregation
        String timeWindow = "2025-01-01T10:00";
        for (int i = 1; i <= 12; i++) {
            double temperature = 20.0 + (Math.random() * 10); // 20-30°C
            producer.send(
                new BusinessMessage("SENSOR-" + i, "Temperature reading"),
                Map.of("timestamp", timeWindow + ":" + String.format("%02d", i * 5),
                       "sensorId", "TEMP-001",
                       "value", String.valueOf(temperature))
            ).join();
        }
        System.out.println("  📊 Sent 12 sensor readings");

        producer.close();
        System.out.println("✅ All aggregation messages sent");
    }

    private String getTimeWindow(String timestamp) {
        // Group by 5-minute windows
        return timestamp.substring(0, 16); // YYYY-MM-DDTHH:MM
    }

    public void close() throws Exception {
        // Cancel all timeout tasks
        for (ScheduledFuture<?> task : timeoutTasks.values()) {
            task.cancel(false);
        }
        timeoutTasks.clear();

        // Shutdown scheduler
        scheduler.shutdown();

        // Close aggregator consumer
        if (aggregatorConsumer != null) {
            aggregatorConsumer.close();
        }

        // Close factory
        factory.close();
    }

    // Simple AggregatedMessage class for the example
    public static class AggregatedMessage {
        private final String correlationId;
        private final int messageCount;
        private final double totalAmount;
        private final double averageValue;
        private final double minValue;
        private final double maxValue;

        public AggregatedMessage(String correlationId, int messageCount, double totalAmount,
                               double averageValue, double minValue, double maxValue) {
            this.correlationId = correlationId;
            this.messageCount = messageCount;
            this.totalAmount = totalAmount;
            this.averageValue = averageValue;
            this.minValue = minValue;
            this.maxValue = maxValue;
        }

        public String getCorrelationId() { return correlationId; }
        public int getMessageCount() { return messageCount; }
        public double getTotalAmount() { return totalAmount; }
        public double getAverageValue() { return averageValue; }
        public double getMinValue() { return minValue; }
        public double getMaxValue() { return maxValue; }
    }
}
```

Message Translator Pattern

The **Message Translator Pattern** transforms messages from one format to another, enabling integration between systems that use different data formats or protocols. This pattern is essential for building adaptable systems that can communicate with diverse external systems.

Key Benefits:

- **Format transformation** - Convert between XML, JSON, CSV, and custom formats
- **Protocol adaptation** - Bridge different messaging protocols and standards
- **Legacy integration** - Connect modern systems with legacy applications
- **Data enrichment** - Add or transform data during translation

Use Cases:

- API integration (REST to SOAP, JSON to XML)
- Legacy system modernization (mainframe to microservices)
- Data pipeline transformation (ETL processes)
- Multi-format support (accept multiple input formats, standardize output)

```
public class MessageTranslatorPatternExample {
    private final QueueFactory factory;
    private final Map<String, MessageConsumer<TranslatedMessage>> translatedConsumers = new HashMap<>();
    private MessageConsumer<RawMessage> mainTranslatorConsumer;
```

```java
public static void main(String[] args) throws Exception {
    // Initialize PeeGeeQ
    PeeGeeQManager manager = new PeeGeeQManager();
    manager.start();

    QueueFactoryProvider provider = QueueFactoryProvider.getInstance();
    QueueFactory factory = provider.createFactory("native", manager.getDatabaseService());

    MessageTranslatorPatternExample example = new MessageTranslatorPatternExample(factory);
    example.runMessageTranslatorExample();

    example.close();
    manager.stop();
}

public MessageTranslatorPatternExample(QueueFactory factory) {
    this.factory = factory;
}

public void runMessageTranslatorExample() throws Exception {
    System.out.println("=== Message Translator Pattern Example ===");

    // Setup translated message consumers first
    setupTranslatedConsumers();

    // Setup the main translator consumer
    setupTranslatorConsumer();

    // Send messages in different formats
    sendMessagesForTranslation();

    // Wait for processing
    Thread.sleep(3000);
    System.out.println("Message translator pattern example completed!");
}

private void setupTranslatorConsumer() throws Exception {
    System.out.println("🔄 Setting up message translator:");

    // Create consumer for messages to translate
    mainTranslatorConsumer = factory.createConsumer("messages-to-translate", RawMessage.class);

    // Subscribe with translation logic
    mainTranslatorConsumer.subscribe(message -> {
        try {
            translateMessage(message);
            return CompletableFuture.completedFuture(null);
        } catch (Exception e) {
            System.err.println("❌ Translation failed: " + e.getMessage());
            return CompletableFuture.failedFuture(e);
        }
    });

    System.out.println("✅ Message translator configured and started");
}

private void translateMessage(Message<RawMessage> message) throws Exception {
    RawMessage rawMessage = message.getPayload();
    String format = message.getHeaders().get("format");

    if (format == null) {
        System.out.println("⚠️  No format specified, skipping translation");
        return;
    }
```

```
        TranslatedMessage translatedMessage = null;
        String outputQueue = null;

        switch (format.toUpperCase()) {
            case "XML":
                translatedMessage = translateXmlToJson(rawMessage);
                outputQueue = "json-messages";
                break;
            case "CSV":
                translatedMessage = translateCsvToStructured(rawMessage);
                outputQueue = "structured-messages";
                break;
            case "LEGACY":
                translatedMessage = translateLegacyToModern(rawMessage);
                outputQueue = "modern-messages";
                break;
            default:
                System.out.printf("⚠️  Unknown format: %s, skipping translation%n", format);
                return;
        }

        // Send translated message
        if (translatedMessage != null && outputQueue != null) {
            MessageProducer<TranslatedMessage> producer =
                factory.createProducer(outputQueue, TranslatedMessage.class);
            producer.send(translatedMessage, Map.of("originalFormat", format)).join();
            producer.close();

            System.out.printf("🔄 Translated %s message to %s%n", format, outputQueue);
        }
    }

    private TranslatedMessage translateXmlToJson(RawMessage rawMessage) {
        // Simple XML to JSON translation (in production, use proper XML/JSON libraries)
        String xmlContent = rawMessage.getContent();

        // Extract order information from XML
        String orderId = extractXmlValue(xmlContent, "id");
        String customer = extractXmlValue(xmlContent, "customer");
        String amount = extractXmlValue(xmlContent, "amount");

        // Create JSON format
        String jsonContent = String.format(
            "{\"orderId\":\"%s\",\"customer\":\"%s\",\"amount\":%s,\"format\":\"JSON\"}",
            orderId, customer, amount
        );

        return new TranslatedMessage(rawMessage.getId() + "-json", jsonContent, "JSON");
    }

    private TranslatedMessage translateCsvToStructured(RawMessage rawMessage) {
        // Simple CSV to structured format translation
        String csvContent = rawMessage.getContent();
        String[] fields = csvContent.split(",");

        if (fields.length >= 4) {
            String structuredContent = String.format(
                "Order{id='%s', customer='%s', amount='%s', date='%s', format='STRUCTURED'}",
                fields[0], fields[1], fields[2], fields[3]
            );
            return new TranslatedMessage(rawMessage.getId() + "-structured", structuredContent, "STRUCTURED");
        }

        return new TranslatedMessage(rawMessage.getId() + "-structured", "Invalid CSV format", "STRUCTURED");
    }
```

```java
private TranslatedMessage translateLegacyToModern(RawMessage rawMessage) {
    // Simple legacy to modern format translation
    String legacyContent = rawMessage.getContent();
    String[] fields = legacyContent.split("\\|");

    if (fields.length >= 6) {
        String modernContent = String.format(
            "{\"type\":\"order\",\"id\":\"%s\",\"customer\":\"%s\",\"amount\":%s,\"date\":\"%s\",\"status\":\"%s\",\"
            fields[1], fields[2], fields[3], fields[4], fields[5]
        );
        return new TranslatedMessage(rawMessage.getId() + "-modern", modernContent, "MODERN");
    }

    return new TranslatedMessage(rawMessage.getId() + "-modern", "Invalid legacy format", "MODERN");
}

private String extractXmlValue(String xml, String tagName) {
    // Simple XML value extraction (in production, use proper XML parser)
    String startTag = "<" + tagName + ">";
    String endTag = "</" + tagName + ">";

    int startIndex = xml.indexOf(startTag);
    int endIndex = xml.indexOf(endTag);

    if (startIndex != -1 && endIndex != -1) {
        return xml.substring(startIndex + startTag.length(), endIndex).trim();
    }

    return "";
}

private void setupTranslatedConsumers() throws Exception {
    System.out.println("🎯 Setting up translated message consumers:");

    // JSON messages consumer
    MessageConsumer<TranslatedMessage> jsonConsumer =
        factory.createConsumer("json-messages", TranslatedMessage.class);
    jsonConsumer.subscribe(message -> {
        System.out.printf("📄 JSON Message: %s%n", message.getPayload().getContent());
        return CompletableFuture.completedFuture(null);
    });
    translatedConsumers.put("json-messages", jsonConsumer);

    // Structured messages consumer
    MessageConsumer<TranslatedMessage> structuredConsumer =
        factory.createConsumer("structured-messages", TranslatedMessage.class);
    structuredConsumer.subscribe(message -> {
        System.out.printf("🏗️  Structured Message: %s%n", message.getPayload().getContent());
        return CompletableFuture.completedFuture(null);
    });
    translatedConsumers.put("structured-messages", structuredConsumer);

    // Modern format consumer
    MessageConsumer<TranslatedMessage> modernConsumer =
        factory.createConsumer("modern-messages", TranslatedMessage.class);
    modernConsumer.subscribe(message -> {
        System.out.printf("🆕 Modern Message: %s%n", message.getPayload().getContent());
        return CompletableFuture.completedFuture(null);
    });
    translatedConsumers.put("modern-messages", modernConsumer);

    System.out.println("✅ Translated message consumers configured");
}

private void sendMessagesForTranslation() throws Exception {
    System.out.println("📤 Sending messages for translation:");
```

```java
        MessageProducer<RawMessage> producer =
            factory.createProducer("messages-to-translate", RawMessage.class);

        // Send XML message
        String xmlContent = """
            <order>
                <id>ORDER-001</id>
                <customer>John Doe</customer>
                <amount>99.99</amount>
            </order>
            """;
        producer.send(
            new RawMessage("XML-001", xmlContent),
            Map.of("format", "XML")
        ).join();
        System.out.println("  📄 Sent XML message");

        // Send CSV message
        String csvContent = "ORDER-002,Jane Smith,149.99,2025-01-01";
        producer.send(
            new RawMessage("CSV-001", csvContent),
            Map.of("format", "CSV")
        ).join();
        System.out.println("  📊 Sent CSV message");

        // Send legacy format message
        String legacyContent = "ORD|003|Bob Johnson|199.99|20250101|ACTIVE";
        producer.send(
            new RawMessage("LEGACY-001", legacyContent),
            Map.of("format", "LEGACY")
        ).join();
        System.out.println("  📁 Sent legacy format message");

        producer.close();
        System.out.println("✅ All translation messages sent");
    }

    public void close() throws Exception {
        // Close main translator consumer
        if (mainTranslatorConsumer != null) {
            mainTranslatorConsumer.close();
        }

        // Close all translated message consumers
        for (MessageConsumer<TranslatedMessage> consumer : translatedConsumers.values()) {
            consumer.close();
        }

        // Close factory
        factory.close();
    }

    // Supporting classes for the example
    public static class RawMessage {
        private final String id;
        private final String content;

        public RawMessage(String id, String content) {
            this.id = id;
            this.content = content;
        }

        public String getId() { return id; }
        public String getContent() { return content; }
    }
```

```java
        public static class TranslatedMessage {
            private final String id;
            private final String content;
            private final String format;

            public TranslatedMessage(String id, String content, String format) {
                this.id = id;
                this.content = content;
                this.format = format;
            }

            public String getId() { return id; }
            public String getContent() { return content; }
            public String getFormat() { return format; }
        }
    }
```

External System Integration

External System Integration patterns enable PeeGeeQ to seamlessly connect with databases, APIs, message brokers, and other enterprise systems. These patterns provide reliable data synchronization, change propagation, and system-to-system communication.

Database Integration Pattern

The **Database Integration Pattern** enables real-time synchronization between your application database and external systems through Change Data Capture (CDC) and event-driven updates. This pattern is crucial for maintaining data consistency across distributed systems.

Key Benefits:

- **Real-time synchronization** - Immediate propagation of database changes
- **Change Data Capture** - Automatic detection of INSERT, UPDATE, DELETE operations
- **Event-driven architecture** - Decouple database changes from business logic
- **Audit trail** - Complete history of all data changes

Use Cases:

- Data warehouse synchronization (OLTP to OLAP)
- Search index updates (database to Elasticsearch)
- Cache invalidation (database changes trigger cache updates)
- Cross-system data replication (master-slave synchronization)

```java
public class DatabaseIntegrationExample {
    private final QueueFactory factory;
    private final Map<String, MessageConsumer<DatabaseChangeEvent>> changeConsumers = new HashMap<>();
    private final ScheduledExecutorService cdcScheduler = Executors.newScheduledThreadPool(2);

    public static void main(String[] args) throws Exception {
        // Initialize PeeGeeQ
        PeeGeeQManager manager = new PeeGeeQManager();
        manager.start();

        QueueFactoryProvider provider = QueueFactoryProvider.getInstance();
        QueueFactory factory = provider.createFactory("native", manager.getDatabaseService());
        DatabaseIntegrationExample example = new DatabaseIntegrationExample(factory);
        example.runDatabaseIntegrationExample();

        example.close();
        manager.stop();
    }

    public DatabaseIntegrationExample(QueueFactory factory) {
        this.factory = factory;
    }

    public void runDatabaseIntegrationExample() throws Exception {
        System.out.println("=== Database Integration Example ===");

        // Setup database synchronization consumers
        setupDatabaseSync();

        // Setup simulated change data capture
        setupChangeDataCapture();

        // Demonstrate data pipeline
        demonstrateDataPipeline();

        // Wait for processing
        Thread.sleep(8000);
        System.out.println("Database integration example completed!");
    }

    private void setupChangeDataCapture() throws Exception {
        System.out.println("📊 Setting up Change Data Capture simulation:");

        // Simulate CDC for orders table
        cdcScheduler.scheduleAtFixedRate(() -> {
            try {
                simulateOrderChanges();
            } catch (Exception e) {
                System.err.println("❌ CDC simulation failed: " + e.getMessage());
            }
        }, 2, 5, TimeUnit.SECONDS);

        // Simulate CDC for customers table
        cdcScheduler.scheduleAtFixedRate(() -> {
            try {
                simulateCustomerChanges();
            } catch (Exception e) {
                System.err.println("❌ CDC simulation failed: " + e.getMessage());
            }
        }, 3, 7, TimeUnit.SECONDS);

        System.out.println("✅ Change Data Capture simulation configured");
    }

    private void simulateOrderChanges() throws Exception {
        MessageProducer<DatabaseChangeEvent> producer =
            factory.createProducer("order-changes", DatabaseChangeEvent.class);

        // Simulate different types of order changes
        String[] operations = {"INSERT", "UPDATE", "DELETE"};
        String operation = operations[(int) (Math.random() * operations.length)];
        String orderId = "ORDER-" + (1000 + (int) (Math.random() * 1000));

        DatabaseChangeEvent changeEvent = new DatabaseChangeEvent(
            orderId,
            "orders",
            operation,
            Map.of("order_id", orderId, "customer_id", "CUST-" + (int) (Math.random() * 100),
```

```java
                   "amount", String.valueOf(50.0 + Math.random() * 500)),
            operation.equals("UPDATE") ? Map.of("amount", String.valueOf(Math.random() * 100)) : null
        );

        producer.send(changeEvent).join();
        producer.close();

        System.out.printf("📊 Simulated %s operation on orders table (ID: %s)%n", operation, orderId);
    }

    private void simulateCustomerChanges() throws Exception {
        MessageProducer<DatabaseChangeEvent> producer =
            factory.createProducer("customer-changes", DatabaseChangeEvent.class);

        String[] operations = {"INSERT", "UPDATE"};
        String operation = operations[(int) (Math.random() * operations.length)];
        String customerId = "CUST-" + (100 + (int) (Math.random() * 100));

        DatabaseChangeEvent changeEvent = new DatabaseChangeEvent(
            customerId,
            "customers",
            operation,
            Map.of("customer_id", customerId, "name", "Customer " + customerId,
                   "email", customerId.toLowerCase() + "@example.com"),
            null
        );

        producer.send(changeEvent).join();
        producer.close();

        System.out.printf("👤 Simulated %s operation on customers table (ID: %s)%n", operation, customerId);
    }

    private void setupDatabaseSync() throws Exception {
        System.out.println("🔄 Setting up Database Synchronization:");

        // Setup consumers for database changes
        MessageConsumer<DatabaseChangeEvent> orderChangesConsumer =
            factory.createConsumer("order-changes", DatabaseChangeEvent.class);
        orderChangesConsumer.subscribe(this::handleOrderChange);
        changeConsumers.put("order-changes", orderChangesConsumer);

        MessageConsumer<DatabaseChangeEvent> customerChangesConsumer =
            factory.createConsumer("customer-changes", DatabaseChangeEvent.class);
        customerChangesConsumer.subscribe(this::handleCustomerChange);
        changeConsumers.put("customer-changes", customerChangesConsumer);

        System.out.println("✅ Database synchronization consumers started");
    }

    private CompletableFuture<Void> handleOrderChange(Message<DatabaseChangeEvent> message) {
        DatabaseChangeEvent change = message.getPayload();

        System.out.printf("📦 Order Change: %s on %s (ID: %s)%n",
            change.getOperation(), change.getTableName(), change.getRecordId());

        // Sync to data warehouse, update search index, etc.
        return syncToExternalSystems(change);
    }

    private CompletableFuture<Void> handleCustomerChange(Message<DatabaseChangeEvent> message) {
        DatabaseChangeEvent change = message.getPayload();

        System.out.printf("👤 Customer Change: %s on %s (ID: %s)%n",
            change.getOperation(), change.getTableName(), change.getRecordId());
        // Update customer profile cache, CRM system, etc.
        return updateCustomerSystems(change);
    }

    private CompletableFuture<Void> syncToExternalSystems(DatabaseChangeEvent change) {
        return CompletableFuture.runAsync(() -> {
            // Simulate syncing to external systems
            try {
                Thread.sleep(100);
                System.out.printf("  ✅ Synced %s to external systems%n", change.getRecordId());
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });
    }

    private CompletableFuture<Void> updateCustomerSystems(DatabaseChangeEvent change) {
        return CompletableFuture.runAsync(() -> {
            // Simulate updating customer systems
            try {
                Thread.sleep(50);
                System.out.printf("  ✅ Updated customer systems for %s%n", change.getRecordId());
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });
    }

    private void demonstrateDataPipeline() throws Exception {
        System.out.println("🔄 Demonstrating Data Pipeline:");

        // The CDC simulation will automatically generate database change events
        // These will be processed by the synchronization consumers
        System.out.println("📊 Database changes will be captured and processed automatically");
        System.out.println("📊 CDC simulation running - watch for change events...");
    }

    public void close() throws Exception {
        // Shutdown CDC scheduler
        cdcScheduler.shutdown();
        try {
            if (!cdcScheduler.awaitTermination(5, TimeUnit.SECONDS)) {
                cdcScheduler.shutdownNow();
            }
        } catch (InterruptedException e) {
            cdcScheduler.shutdownNow();
            Thread.currentThread().interrupt();
        }

        // Close all change consumers
        for (MessageConsumer<DatabaseChangeEvent> consumer : changeConsumers.values()) {
            consumer.close();
        }

        // Close factory
        factory.close();
    }

    // Supporting class for database change events
    public static class DatabaseChangeEvent {
        private final String recordId;
        private final String tableName;
        private final String operation;
        private final Map<String, String> newValues;
        private final Map<String, String> oldValues;
        public DatabaseChangeEvent(String recordId, String tableName, String operation,
                                 Map<String, String> newValues, Map<String, String> oldValues) {
            this.recordId = recordId;
            this.tableName = tableName;
            this.operation = operation;
            this.newValues = newValues != null ? new HashMap<>(newValues) : new HashMap<>();
            this.oldValues = oldValues != null ? new HashMap<>(oldValues) : new HashMap<>();
        }

        public String getRecordId() { return recordId; }
        public String getTableName() { return tableName; }
        public String getOperation() { return operation; }
        public Map<String, String> getNewValues() { return newValues; }
        public Map<String, String> getOldValues() { return oldValues; }
    }
}
```

REST API Integration Pattern

The **REST API Integration Pattern** enables PeeGeeQ to integrate with external REST APIs, providing reliable request/response handling with retry logic and error handling.

Key Benefits:

- **Reliable API calls** - Automatic retry with exponential backoff
- **Async processing** - Non-blocking API integration
- **Error handling** - Dead letter queues for failed API calls
- **Rate limiting** - Respect API rate limits with controlled throughput

```
public class RestApiIntegrationExample {
    private final QueueFactory factory;
    private final HttpClient httpClient;

    public RestApiIntegrationExample(QueueFactory factory) {
        this.factory = factory;
        this.httpClient = HttpClient.newBuilder()
            .connectTimeout(Duration.ofSeconds(10))
            .build();
    }

    public void setupApiIntegration() throws Exception {
        // Consumer for API requests
        MessageConsumer<ApiRequest> apiConsumer =
            factory.createConsumer("api-requests", ApiRequest.class);

        apiConsumer.subscribe(message -> {
            return processApiRequest(message.getPayload())
                .exceptionally(throwable -> {
                    // Send to dead letter queue on failure
                    handleApiFailure(message.getPayload(), throwable);
                    return null;
                });
        });
    }

    private CompletableFuture<Void> processApiRequest(ApiRequest request) {
        return CompletableFuture.supplyAsync(() -> {
            try {
                HttpRequest httpRequest = HttpRequest.newBuilder()
                    .uri(URI.create(request.getUrl()))
                    .header("Content-Type", "application/json")
                    .POST(HttpRequest.BodyPublishers.ofString(request.getPayload()))
                    .build();

                HttpResponse<String> response = httpClient.send(httpRequest,
                    HttpResponse.BodyHandlers.ofString());

                if (response.statusCode() >= 200 && response.statusCode() < 300) {
                    System.out.printf("✅ API call successful: %s%n", request.getUrl());
                    return null;
                } else {
                    throw new RuntimeException("API call failed with status: " + response.statusCode());
                }
            } catch (Exception e) {
                throw new RuntimeException("API call failed", e);
            }
        });
    }

    private void handleApiFailure(ApiRequest request, Throwable error) {
        try {
            MessageProducer<ApiRequest> dlqProducer =
                factory.createProducer("api-failures", ApiRequest.class);
            dlqProducer.send(request, Map.of("error", error.getMessage())).join();
            dlqProducer.close();
        } catch (Exception e) {
            System.err.println("Failed to send to DLQ: " + e.getMessage());
        }
    }

    public static class ApiRequest {
        private final String id;
        private final String url;
        private final String payload;

        public ApiRequest(String id, String url, String payload) {
            this.id = id;
            this.url = url;
            this.payload = payload;
        }

        public String getId() { return id; }
        public String getUrl() { return url; }
        public String getPayload() { return payload; }
    }
}
```

Dead Letter Channel Pattern

The **Dead Letter Channel Pattern** handles messages that cannot be processed successfully, providing a systematic approach to error handling and message recovery.

Key Benefits:

- **Error isolation** - Separate failed messages from normal processing
- **Manual intervention** - Allow operators to inspect and reprocess failed messages
- **System stability** - Prevent poison messages from blocking queues
- **Audit trail** - Complete record of processing failures

```
public class DeadLetterChannelExample {
    private final QueueFactory factory;
```

```java
    public void setupDeadLetterHandling() throws Exception {
        // Main processing consumer with error handling
        MessageConsumer<BusinessMessage> mainConsumer =
            factory.createConsumer("main-queue", BusinessMessage.class);

        mainConsumer.subscribe(message -> {
            return processMessage(message)
                .exceptionally(throwable -> {
                    sendToDeadLetterQueue(message, throwable);
                    return null;
                });
        });

        // Dead letter queue consumer for manual processing
        MessageConsumer<BusinessMessage> dlqConsumer =
            factory.createConsumer("dead-letter-queue", BusinessMessage.class);

        dlqConsumer.subscribe(this::handleDeadLetterMessage);
    }

    private CompletableFuture<Void> processMessage(Message<BusinessMessage> message) {
        return CompletableFuture.runAsync(() -> {
            // Simulate processing that might fail
            if (Math.random() < 0.1) { // 10% failure rate
                throw new RuntimeException("Processing failed");
            }
            System.out.printf("✅ Processed message: %s%n", message.getId());
        });
    }

    private void sendToDeadLetterQueue(Message<BusinessMessage> message, Throwable error) {
        try {
            MessageProducer<BusinessMessage> dlqProducer =
                factory.createProducer("dead-letter-queue", BusinessMessage.class);

            Map<String, String> dlqHeaders = new HashMap<>(message.getHeaders());
            dlqHeaders.put("error", error.getMessage());
            dlqHeaders.put("failed-at", Instant.now().toString());
            dlqHeaders.put("retry-count", "0");

            dlqProducer.send(message.getPayload(), dlqHeaders).join();
            dlqProducer.close();

            System.out.printf("📤 Sent to DLQ: %s (Error: %s)%n", message.getId(), error.getMessage());
        } catch (Exception e) {
            System.err.println("Failed to send to DLQ: " + e.getMessage());
        }
    }

    private CompletableFuture<Void> handleDeadLetterMessage(Message<BusinessMessage> message) {
        System.out.printf("🔍 Dead letter message: %s (Error: %s)%n",
            message.getId(), message.getHeaders().get("error"));

        // Here you could implement:
        // - Manual retry logic
        // - Notification to operators
        // - Logging for analysis
        // - Automatic retry after delay

        return CompletableFuture.completedFuture(null);
    }
}
```

Scatter-Gather Pattern

The **Scatter-Gather Pattern** sends a message to multiple recipients and collects their responses, enabling parallel processing and result aggregation. This pattern is essential for distributed queries and parallel processing scenarios.

Key Benefits:

- **Parallel processing** - Execute operations concurrently across multiple services
- **Result aggregation** - Combine responses from multiple sources
- **Timeout handling** - Handle partial responses when some services are slow
- **Load distribution** - Distribute work across multiple processors

Use Cases:

- Price comparison across multiple vendors
- Distributed search across multiple data sources
- Parallel validation across multiple services
- Multi-service health checks

```java
public class ScatterGatherPatternExample {
    private final QueueFactory factory;
    private final Map<String, CompletableFuture<String>> pendingRequests = new ConcurrentHashMap<>();
    private final ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(2);

    public static void main(String[] args) throws Exception {
        PeeGeeQManager manager = new PeeGeeQManager();
        manager.start();

        QueueFactoryProvider provider = QueueFactoryProvider.getInstance();
        QueueFactory factory = provider.createFactory("native", manager.getDatabaseService());

        ScatterGatherPatternExample example = new ScatterGatherPatternExample(factory);
        example.runScatterGatherExample();

        example.close();
        manager.stop();
    }

    public ScatterGatherPatternExample(QueueFactory factory) {
        this.factory = factory;
    }

    public void runScatterGatherExample() throws Exception {
        System.out.println("=== Scatter-Gather Pattern Example ===");

        // Setup response collectors
        setupResponseCollectors();

        // Demonstrate scatter-gather operations
        demonstrateScatterGather();

        // Wait for processing
        Thread.sleep(10000);
        System.out.println("Scatter-gather pattern example completed!");
    }

    private void setupResponseCollectors() throws Exception {
        System.out.println("🎯 Setting up response collectors:");

        // Collector for price comparison responses
        MessageConsumer<PriceResponse> priceCollector =
            factory.createConsumer("price-responses", PriceResponse.class);
        priceCollector.subscribe(this::handlePriceResponse);
        // Collector for search responses
        MessageConsumer<SearchResponse> searchCollector =
            factory.createConsumer("search-responses", SearchResponse.class);
        searchCollector.subscribe(this::handleSearchResponse);

        System.out.println("✅ Response collectors configured");
    }

    public CompletableFuture<List<PriceResponse>> scatterGatherPriceComparison(String productId) throws Exception {
        String requestId = UUID.randomUUID().toString();
        System.out.printf("🔄 Starting price comparison for product: %s (Request: %s)%n", productId, requestId);

        // Create future for collecting responses
        CompletableFuture<List<PriceResponse>> resultFuture = new CompletableFuture<>();
        List<PriceResponse> responses = Collections.synchronizedList(new ArrayList<>());

        // Setup timeout
        ScheduledFuture<?> timeoutTask = scheduler.schedule(() -> {
            System.out.printf("⏰ Price comparison timeout for request: %s%n", requestId);
            resultFuture.complete(new ArrayList<>(responses));
        }, 5, TimeUnit.SECONDS);

        // Store request context
        RequestContext context = new RequestContext(resultFuture, responses, timeoutTask, 3); // Expect 3 responses
        requestContexts.put(requestId, context);

        // Scatter to multiple price services
        MessageProducer<PriceRequest> producer = factory.createProducer("price-requests", PriceRequest.class);

        String[] vendors = {"vendor-a", "vendor-b", "vendor-c"};
        for (String vendor : vendors) {
            PriceRequest request = new PriceRequest(requestId, productId, vendor);
            producer.send(request, Map.of("vendor", vendor, "requestId", requestId)).join();
            System.out.printf("📤 Sent price request to %s%n", vendor);
        }

        producer.close();
        return resultFuture;
    }

    private CompletableFuture<Void> handlePriceResponse(Message<PriceResponse> message) {
        PriceResponse response = message.getPayload();
        String requestId = message.getHeaders().get("requestId");

        System.out.printf("📥 Received price response from %s: $%.2f (Request: %s)%n",
            response.getVendor(), response.getPrice(), requestId);

        RequestContext context = requestContexts.get(requestId);
        if (context != null) {
            context.responses.add(response);

            // Check if we have all responses
            if (context.responses.size() >= context.expectedCount) {
                context.timeoutTask.cancel(false);
                context.future.complete(new ArrayList<>(context.responses));
                requestContexts.remove(requestId);
                System.out.printf("✅ Price comparison complete for request: %s%n", requestId);
            }
        }

        return CompletableFuture.completedFuture(null);
    }

    private CompletableFuture<Void> handleSearchResponse(Message<SearchResponse> message) {
        SearchResponse response = message.getPayload();
        String requestId = message.getHeaders().get("requestId");

        System.out.printf("🔍 Received search response from %s: %d results (Request: %s)%n",
            response.getSource(), response.getResults().size(), requestId);

        // Similar handling logic for search responses
        return CompletableFuture.completedFuture(null);
    }

    private void demonstrateScatterGather() throws Exception {
        System.out.println("🚀 Demonstrating scatter-gather operations:");

        // Price comparison example
        CompletableFuture<List<PriceResponse>> priceComparison =
            scatterGatherPriceComparison("PRODUCT-123");

        priceComparison.thenAccept(responses -> {
            System.out.println("💰 Price comparison results:");
            responses.stream()
                .sorted((a, b) -> Double.compare(a.getPrice(), b.getPrice()))
                .forEach(response ->
                    System.out.printf("  %s: $%.2f%n", response.getVendor(), response.getPrice()));

            if (!responses.isEmpty()) {
                PriceResponse best = responses.stream()
                    .min((a, b) -> Double.compare(a.getPrice(), b.getPrice()))
                    .get();
                System.out.printf("🏆 Best price: %s at $%.2f%n", best.getVendor(), best.getPrice());
            }
        });
    }

    public void close() throws Exception {
        scheduler.shutdown();
        factory.close();
    }

    // Supporting classes
    private final Map<String, RequestContext> requestContexts = new ConcurrentHashMap<>();

    private static class RequestContext {
        final CompletableFuture<List<PriceResponse>> future;
        final List<PriceResponse> responses;
        final ScheduledFuture<?> timeoutTask;
        final int expectedCount;

        RequestContext(CompletableFuture<List<PriceResponse>> future, List<PriceResponse> responses,
                      ScheduledFuture<?> timeoutTask, int expectedCount) {
            this.future = future;
            this.responses = responses;
            this.timeoutTask = timeoutTask;
            this.expectedCount = expectedCount;
        }
    }

    public static class PriceRequest {
        private final String requestId;
        private final String productId;
        private final String vendor;

        public PriceRequest(String requestId, String productId, String vendor) {
            this.requestId = requestId;
            this.productId = productId;
            this.vendor = vendor;
        }
        public String getRequestId() { return requestId; }
        public String getProductId() { return productId; }
        public String getVendor() { return vendor; }
    }

    public static class PriceResponse {
        private final String requestId;
        private final String productId;
        private final String vendor;
        private final double price;

        public PriceResponse(String requestId, String productId, String vendor, double price) {
            this.requestId = requestId;
            this.productId = productId;
            this.vendor = vendor;
            this.price = price;
        }

        public String getRequestId() { return requestId; }
        public String getProductId() { return productId; }
        public String getVendor() { return vendor; }
        public double getPrice() { return price; }
    }

    public static class SearchResponse {
        private final String requestId;
        private final String source;
        private final List<String> results;

        public SearchResponse(String requestId, String source, List<String> results) {
            this.requestId = requestId;
            this.source = source;
            this.results = results;
        }

        public String getRequestId() { return requestId; }
        public String getSource() { return source; }
        public List<String> getResults() { return results; }
    }
}
```

Request-Reply Pattern

The **Request-Reply Pattern** enables synchronous-style communication over asynchronous messaging, providing a way to get responses from message processing while maintaining the benefits of decoupled architecture.

Key Benefits:

- **Synchronous semantics** - Get responses from async operations
- **Correlation tracking** - Match requests with their responses
- **Timeout handling** - Handle scenarios where responses don't arrive
- **Decoupled architecture** - Maintain loose coupling between services

Use Cases:

- API gateway to microservices communication
- Synchronous queries over async infrastructure
- Command-query operations with responses
- Service-to-service RPC over messaging

```java
public class RequestReplyPatternExample {
    private final QueueFactory factory;
    private final Map<String, CompletableFuture<String>> pendingRequests = new ConcurrentHashMap<>();
    private final ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(2);
    private MessageConsumer<ResponseMessage> responseConsumer;

    public static void main(String[] args) throws Exception {
        PeeGeeQManager manager = new PeeGeeQManager();
        manager.start();

        QueueFactoryProvider provider = QueueFactoryProvider.getInstance();
        QueueFactory factory = provider.createFactory("native", manager.getDatabaseService());

        RequestReplyPatternExample example = new RequestReplyPatternExample(factory);
        example.runRequestReplyExample();

        example.close();
        manager.stop();
    }

    public RequestReplyPatternExample(QueueFactory factory) {
        this.factory = factory;
    }

    public void runRequestReplyExample() throws Exception {
        System.out.println("=== Request-Reply Pattern Example ===");

        // Setup response handler
        setupResponseHandler();

        // Setup request processor (simulates remote service)
        setupRequestProcessor();

        // Demonstrate request-reply operations
        demonstrateRequestReply();

        // Wait for processing
        Thread.sleep(5000);
        System.out.println("Request-reply pattern example completed!");
    }

    private void setupResponseHandler() throws Exception {
        System.out.println("📨 Setting up response handler:");

        responseConsumer = factory.createConsumer("responses", ResponseMessage.class);
        responseConsumer.subscribe(message -> {
            ResponseMessage response = message.getPayload();
            String correlationId = message.getHeaders().get("correlationId");

            System.out.printf("📨 Received response for correlation ID: %s%n", correlationId);

            CompletableFuture<String> pendingRequest = pendingRequests.remove(correlationId);
            if (pendingRequest != null) {
                if (response.isSuccess()) {
                    pendingRequest.complete(response.getData());
                } else {
                    pendingRequest.completeExceptionally(new RuntimeException(response.getError()));
                }
            } else {
                System.out.printf("⚠️ No pending request found for correlation ID: %s%n", correlationId);
            }

            return CompletableFuture.completedFuture(null);
        });
        System.out.println("✅ Response handler configured");
    }

    private void setupRequestProcessor() throws Exception {
        System.out.println("🔧 Setting up request processor (simulates remote service):");

        MessageConsumer<RequestMessage> requestConsumer =
            factory.createConsumer("requests", RequestMessage.class);

        requestConsumer.subscribe(message -> {
            RequestMessage request = message.getPayload();
            String correlationId = message.getHeaders().get("correlationId");

            System.out.printf("🔄 Processing request: %s (Correlation: %s)%n",
                request.getOperation(), correlationId);

            return CompletableFuture.supplyAsync(() -> {
                try {
                    // Simulate processing time
                    Thread.sleep(1000 + (int)(Math.random() * 2000));

                    // Simulate processing logic
                    String result = processRequest(request);

                    // Send response
                    ResponseMessage response = new ResponseMessage(true, result, null);
                    MessageProducer<ResponseMessage> responseProducer =
                        factory.createProducer("responses", ResponseMessage.class);
                    responseProducer.send(response, Map.of("correlationId", correlationId)).join();
                    responseProducer.close();

                    System.out.printf("✅ Sent response for correlation ID: %s%n", correlationId);

                } catch (Exception e) {
                    // Send error response
                    ResponseMessage errorResponse = new ResponseMessage(false, null, e.getMessage());
                    try {
                        MessageProducer<ResponseMessage> responseProducer =
                            factory.createProducer("responses", ResponseMessage.class);
                        responseProducer.send(errorResponse, Map.of("correlationId", correlationId)).join();
                        responseProducer.close();
                    } catch (Exception ex) {
                        System.err.println("Failed to send error response: " + ex.getMessage());
                    }
                }
                return null;
            });
        });

        System.out.println("✅ Request processor configured");
    }

    public CompletableFuture<String> sendRequest(String operation, String data, Duration timeout) throws Exception {
        String correlationId = UUID.randomUUID().toString();

        System.out.printf("📤 Sending request: %s (Correlation: %s)%n", operation, correlationId);

        // Create response future
        CompletableFuture<String> responseFuture = new CompletableFuture<>();
        pendingRequests.put(correlationId, responseFuture);

        // Setup timeout
        scheduler.schedule(() -> {
            CompletableFuture<String> timeoutRequest = pendingRequests.remove(correlationId);
            if (timeoutRequest != null) {
                timeoutRequest.completeExceptionally(new TimeoutException("Request timeout"));
                System.out.printf("⏰ Request timeout for correlation ID: %s%n", correlationId);
            }
        }, timeout.toMillis(), TimeUnit.MILLISECONDS);

        // Send request
        RequestMessage request = new RequestMessage(operation, data);
        MessageProducer<RequestMessage> producer = factory.createProducer("requests", RequestMessage.class);
        producer.send(request, Map.of("correlationId", correlationId)).join();
        producer.close();

        return responseFuture;
    }

    private String processRequest(RequestMessage request) throws Exception {
        // Simulate different operations
        switch (request.getOperation().toUpperCase()) {
            case "CALCULATE":
                return "Result: " + (Math.random() * 1000);
            case "VALIDATE":
                return Math.random() > 0.2 ? "VALID" : "INVALID";
            case "TRANSFORM":
                return request.getData().toUpperCase();
            case "ERROR":
                throw new RuntimeException("Simulated processing error");
            default:
                return "Unknown operation: " + request.getOperation();
        }
    }

    private void demonstrateRequestReply() throws Exception {
        System.out.println("🚀 Demonstrating request-reply operations:");

        // Send multiple requests
        List<CompletableFuture<String>> futures = new ArrayList<>();

        futures.add(sendRequest("CALCULATE", "some data", Duration.ofSeconds(5)));
        futures.add(sendRequest("VALIDATE", "test@example.com", Duration.ofSeconds(5)));
        futures.add(sendRequest("TRANSFORM", "hello world", Duration.ofSeconds(5)));

        // Wait for all responses
        CompletableFuture.allOf(futures.toArray(new CompletableFuture[0]))
            .thenRun(() -> {
                System.out.println("📋 All responses received:");
                for (int i = 0; i < futures.size(); i++) {
                    try {
                        String result = futures.get(i).get();
                        System.out.printf("  Response %d: %s%n", i + 1, result);
                    } catch (Exception e) {
                        System.out.printf("  Response %d: ERROR - %s%n", i + 1, e.getMessage());
                    }
                }
            });
    }

    public void close() throws Exception {
        scheduler.shutdown();
        if (responseConsumer != null) {
            responseConsumer.close();
        }
        factory.close();
    }

    // Supporting classes
    public static class RequestMessage {
        private final String operation;
        private final String data;
        public RequestMessage(String operation, String data) {
            this.operation = operation;
            this.data = data;
        }

        public String getOperation() { return operation; }
        public String getData() { return data; }
    }

    public static class ResponseMessage {
        private final boolean success;
        private final String data;
        private final String error;

        public ResponseMessage(boolean success, String data, String error) {
            this.success = success;
            this.data = data;
            this.error = error;
        }

        public boolean isSuccess() { return success; }
        public String getData() { return data; }
        public String getError() { return error; }
    }
}
```

Content-Based Router Pattern

The **Content-Based Router Pattern** extends the basic message router by examining message content to make sophisticated routing decisions. This pattern enables intelligent message distribution based on business rules and message data.

Key Benefits:

- **Intelligent routing** - Route based on message content, not just headers
- **Business rule integration** - Apply complex business logic to routing decisions
- **Dynamic routing** - Routing rules can change based on system state
- **Content filtering** - Filter messages based on content criteria

Use Cases:

- Route orders based on customer tier or order value
- Direct messages based on geographic location
- Filter and route based on message priority or urgency
- Route based on data validation results

```
public class ContentBasedRouterExample {
    private final QueueFactory factory;
    private final Map<String, RoutingRule> routingRules = new HashMap<>();

    public static void main(String[] args) throws Exception {
        PeeGeeQManager manager = new PeeGeeQManager();
        manager.start();

        QueueFactoryProvider provider = QueueFactoryProvider.getInstance();
        QueueFactory factory = provider.createFactory("native", manager.getDatabaseService());

        ContentBasedRouterExample example = new ContentBasedRouterExample(factory);
        example.runContentBasedRoutingExample();
```

```java
        example.close();
        manager.stop();
    }

    public ContentBasedRouterExample(QueueFactory factory) {
        this.factory = factory;
        setupRoutingRules();
    }

    public void runContentBasedRoutingExample() throws Exception {
        System.out.println("=== Content-Based Router Pattern Example ===");

        // Setup router
        setupContentBasedRouter();

        // Setup destination consumers
        setupDestinationConsumers();

        // Send test messages
        sendTestMessages();

        // Wait for processing
        Thread.sleep(5000);
        System.out.println("Content-based routing example completed!");
    }

    private void setupRoutingRules() {
        System.out.println("📋 Setting up routing rules:");

        // Rule 1: High-value orders go to premium processing
        routingRules.put("high-value-orders", new RoutingRule(
            "premium-orders",
            order -> order instanceof OrderMessage && ((OrderMessage) order).getAmount() > 1000.0,
            "High-value orders (>$1000) → Premium processing"
        ));

        // Rule 2: VIP customers get priority processing
        routingRules.put("vip-customers", new RoutingRule(
            "vip-orders",
            order -> order instanceof OrderMessage && "VIP".equals(((OrderMessage) order).getCustomerTier()),
            "VIP customers → Priority processing"
        ));

        // Rule 3: International orders need special handling
        routingRules.put("international-orders", new RoutingRule(
            "international-orders",
            order -> order instanceof OrderMessage && !((OrderMessage) order).getCountry().equals("US"),
            "International orders → Special processing"
        ));

        // Rule 4: Bulk orders go to batch processing
        routingRules.put("bulk-orders", new RoutingRule(
            "bulk-orders",
            order -> order instanceof OrderMessage && ((OrderMessage) order).getQuantity() > 100,
            "Bulk orders (>100 items) → Batch processing"
        ));

        // Default rule: Standard processing
        routingRules.put("default", new RoutingRule(
            "standard-orders",
            order -> true, // Always matches
            "Default → Standard processing"
        ));

        routingRules.values().forEach(rule ->
            System.out.printf("  ✅ %s%n", rule.getDescription()));
    }

    private void setupContentBasedRouter() throws Exception {
        System.out.println("🔀 Setting up content-based router:");

        MessageConsumer<OrderMessage> router = factory.createConsumer("incoming-orders", OrderMessage.class);
        router.subscribe(message -> {
            OrderMessage order = message.getPayload();

            System.out.printf("🔍 Routing order %s (Amount: $%.2f, Tier: %s, Country: %s, Qty: %d)%n",
                order.getOrderId(), order.getAmount(), order.getCustomerTier(),
                order.getCountry(), order.getQuantity());

            return routeMessage(order, message.getHeaders());
        });

        System.out.println("✅ Content-based router configured");
    }

    private CompletableFuture<Void> routeMessage(OrderMessage order, Map<String, String> headers) {
        return CompletableFuture.supplyAsync(() -> {
            try {
                // Apply routing rules in priority order
                String[] ruleOrder = {"high-value-orders", "vip-customers", "international-orders", "bulk-orders", "defau

                for (String ruleName : ruleOrder) {
                    RoutingRule rule = routingRules.get(ruleName);
                    if (rule.matches(order)) {
                        String destination = rule.getDestination();

                        System.out.printf("📤 Routing to %s: %s%n", destination, rule.getDescription());

                        // Send to destination queue
                        MessageProducer<OrderMessage> producer = factory.createProducer(destination, OrderMessage.class);

                        // Add routing information to headers
                        Map<String, String> routingHeaders = new HashMap<>(headers);
                        routingHeaders.put("routedBy", "content-based-router");
                        routingHeaders.put("routingRule", ruleName);
                        routingHeaders.put("routedAt", Instant.now().toString());

                        producer.send(order, routingHeaders).join();
                        producer.close();

                        return null; // Stop at first matching rule
                    }
                }

                System.err.println("❌ No routing rule matched - this should not happen!");
                return null;

            } catch (Exception e) {
                System.err.println("❌ Routing failed: " + e.getMessage());
                throw new RuntimeException("Routing failed", e);
            }
        });
    }

    private void setupDestinationConsumers() throws Exception {
        System.out.println("📬 Setting up destination consumers:");

        // Premium orders consumer
        MessageConsumer<OrderMessage> premiumConsumer =
            factory.createConsumer("premium-orders", OrderMessage.class);
        premiumConsumer.subscribe(message -> {
            OrderMessage order = message.getPayload();
            System.out.printf("💎 PREMIUM: Processing high-value order %s ($%.2f)%n",
                order.getOrderId(), order.getAmount());
            return CompletableFuture.completedFuture(null);
        });

        // VIP orders consumer
        MessageConsumer<OrderMessage> vipConsumer =
            factory.createConsumer("vip-orders", OrderMessage.class);
        vipConsumer.subscribe(message -> {
            OrderMessage order = message.getPayload();
            System.out.printf("⭐ VIP: Processing VIP customer order %s%n", order.getOrderId());
            return CompletableFuture.completedFuture(null);
        });

        // International orders consumer
        MessageConsumer<OrderMessage> internationalConsumer =
            factory.createConsumer("international-orders", OrderMessage.class);
        internationalConsumer.subscribe(message -> {
            OrderMessage order = message.getPayload();
            System.out.printf("🌍 INTERNATIONAL: Processing order %s from %s%n",
                order.getOrderId(), order.getCountry());
            return CompletableFuture.completedFuture(null);
        });

        // Bulk orders consumer
        MessageConsumer<OrderMessage> bulkConsumer =
            factory.createConsumer("bulk-orders", OrderMessage.class);
        bulkConsumer.subscribe(message -> {
            OrderMessage order = message.getPayload();
            System.out.printf("📦 BULK: Processing bulk order %s (%d items)%n",
                order.getOrderId(), order.getQuantity());
            return CompletableFuture.completedFuture(null);
        });

        // Standard orders consumer
        MessageConsumer<OrderMessage> standardConsumer =
            factory.createConsumer("standard-orders", OrderMessage.class);
        standardConsumer.subscribe(message -> {
            OrderMessage order = message.getPayload();
            System.out.printf("📋 STANDARD: Processing standard order %s%n", order.getOrderId());
            return CompletableFuture.completedFuture(null);
        });

        System.out.println("✅ All destination consumers configured");
    }

    private void sendTestMessages() throws Exception {
        System.out.println("📤 Sending test messages:");

        MessageProducer<OrderMessage> producer = factory.createProducer("incoming-orders", OrderMessage.class);

        // Test messages for different routing scenarios
        OrderMessage[] testOrders = {
            new OrderMessage("ORD-001", 1500.0, "STANDARD", "US", 5),      // High-value
            new OrderMessage("ORD-002", 500.0, "VIP", "US", 2),            // VIP customer
            new OrderMessage("ORD-003", 300.0, "STANDARD", "UK", 10),      // International
            new OrderMessage("ORD-004", 200.0, "STANDARD", "US", 150),     // Bulk order
            new OrderMessage("ORD-005", 100.0, "STANDARD", "US", 1),       // Standard order
            new OrderMessage("ORD-006", 2000.0, "VIP", "CA", 200)          // Multiple rules match
        };

        for (OrderMessage order : testOrders) {
            producer.send(order, Map.of("timestamp", Instant.now().toString())).join();
            System.out.printf("   📤 Sent order: %s%n", order.getOrderId());
            Thread.sleep(500); // Small delay for readability
        }

        producer.close();
        System.out.println("✅ All test messages sent");
    }

    public void close() throws Exception {
        factory.close();
    }

    // Supporting classes
    public static class OrderMessage {
        private final String orderId;
        private final double amount;
        private final String customerTier;
        private final String country;
        private final int quantity;

        public OrderMessage(String orderId, double amount, String customerTier, String country, int quantity) {
            this.orderId = orderId;
            this.amount = amount;
            this.customerTier = customerTier;
            this.country = country;
            this.quantity = quantity;
        }

        public String getOrderId() { return orderId; }
        public double getAmount() { return amount; }
        public String getCustomerTier() { return customerTier; }
        public String getCountry() { return country; }
        public int getQuantity() { return quantity; }
    }

    public static class RoutingRule {
        private final String destination;
        private final Predicate<Object> condition;
        private final String description;

        public RoutingRule(String destination, Predicate<Object> condition, String description) {
            this.destination = destination;
            this.condition = condition;
            this.description = description;
        }

        public boolean matches(Object message) {
            return condition.test(message);
        }

        public String getDestination() { return destination; }
        public String getDescription() { return description; }
    }
}
```

🎯 **Try This Now**:

1. Implement the message router with your own routing rules
2. Create an aggregator for your specific use case
3. Build message translators for different data formats
4. Set up change data capture for your database tables
5. Add REST API integration for external system communication
6. Implement dead letter channels for robust error handling
7. Build scatter-gather patterns for parallel processing

8. Create request-reply patterns for synchronous-style communication
9. Implement content-based routing with business rules

Publish-Subscribe Pattern

The **Publish-Subscribe Pattern** enables one-to-many message distribution where publishers send messages to topics and multiple subscribers receive copies of those messages. This pattern is fundamental for event-driven architectures and real-time notifications.

Key Benefits:

- **Decoupled communication** - Publishers don't know about subscribers
- **Dynamic subscription** - Subscribers can join/leave at runtime
- **Event broadcasting** - Single event reaches multiple interested parties
- **Scalable architecture** - Easy to add new subscribers without changing publishers

Use Cases:

- Event notifications (order placed, payment processed)
- Real-time updates (stock prices, system status)
- Audit logging (multiple audit systems)
- Cache invalidation across multiple services

```java
public class PublishSubscribePatternExample {
    private final QueueFactory factory;
    private final Map<String, List<MessageConsumer<?>>> topicSubscribers = new HashMap<>();

    public static void main(String[] args) throws Exception {
        PeeGeeQManager manager = new PeeGeeQManager();
        manager.start();

        QueueFactoryProvider provider = QueueFactoryProvider.getInstance();
        QueueFactory factory = provider.createFactory("native", manager.getDatabaseService());

        PublishSubscribePatternExample example = new PublishSubscribePatternExample(factory);
        example.runPublishSubscribeExample();

        example.close();
        manager.stop();
    }

    public PublishSubscribePatternExample(QueueFactory factory) {
        this.factory = factory;
    }

    public void runPublishSubscribeExample() throws Exception {
        System.out.println("=== Publish-Subscribe Pattern Example ===");

        // Setup topic publishers
        setupTopicPublishers();

        // Setup subscribers
        setupSubscribers();

        // Demonstrate pub-sub operations
        demonstratePublishSubscribe();

        // Wait for processing
        Thread.sleep(8000);
        System.out.println("Publish-subscribe pattern example completed!");
    }

    private void setupTopicPublishers() throws Exception {
        System.out.println("📡 Setting up topic publishers:");

        // Publisher for order events
        setupTopicPublisher("order-events", "Order lifecycle events");

        // Publisher for payment events
        setupTopicPublisher("payment-events", "Payment processing events");

        // Publisher for inventory events
        setupTopicPublisher("inventory-events", "Inventory management events");

        // Publisher for system events
        setupTopicPublisher("system-events", "System status and alerts");

        System.out.println("✅ Topic publishers configured");
    }

    private void setupTopicPublisher(String topicName, String description) throws Exception {
        // Create topic consumer that distributes to subscribers
        MessageConsumer<TopicMessage> topicConsumer =
            factory.createConsumer(topicName, TopicMessage.class);

        topicConsumer.subscribe(message -> {
            TopicMessage topicMessage = message.getPayload();
            String eventType = message.getHeaders().get("eventType");

            System.out.printf("📡 Publishing to topic '%s': %s (Type: %s)%n",
                topicName, topicMessage.getContent(), eventType);

            return distributeToSubscribers(topicName, topicMessage, message.getHeaders());
        });

        System.out.printf("  ✅ %s: %s%n", topicName, description);
    }

    private CompletableFuture<Void> distributeToSubscribers(String topicName, TopicMessage message, Map<String, String> h
        return CompletableFuture.runAsync(() -> {
            try {
                // Get subscriber queues for this topic
                List<String> subscriberQueues = getSubscriberQueues(topicName);

                if (subscriberQueues.isEmpty()) {
                    System.out.printf("⚠️ No subscribers for topic: %s%n", topicName);
                    return;
                }

                // Send to all subscribers
                for (String subscriberQueue : subscriberQueues) {
                    MessageProducer<TopicMessage> producer =
                        factory.createProducer(subscriberQueue, TopicMessage.class);

                    Map<String, String> subscriberHeaders = new HashMap<>(headers);
                    subscriberHeaders.put("topic", topicName);
                    subscriberHeaders.put("subscribedAt", Instant.now().toString());

                    producer.send(message, subscriberHeaders).join();
                    producer.close();

                    System.out.printf("  📤 Sent to subscriber: %s%n", subscriberQueue);
                }

            } catch (Exception e) {
                System.err.println("❌ Failed to distribute to subscribers: " + e.getMessage());
                throw new RuntimeException("Distribution failed", e);
            }
        });
    }

    private List<String> getSubscriberQueues(String topicName) {
        // In a real implementation, this would be stored in a registry
        // For demo purposes, we'll use a simple mapping
        Map<String, List<String>> topicSubscriptions = Map.of(
            "order-events", List.of("order-audit-subscriber", "order-notification-subscriber", "order-analytics-subscribe
            "payment-events", List.of("payment-audit-subscriber", "payment-notification-subscriber"),
            "inventory-events", List.of("inventory-audit-subscriber", "inventory-reorder-subscriber"),
            "system-events", List.of("system-monitoring-subscriber", "system-alert-subscriber")
        );

        return topicSubscriptions.getOrDefault(topicName, List.of());
    }

    private void setupSubscribers() throws Exception {
        System.out.println("📨 Setting up subscribers:");

        // Order event subscribers
        setupSubscriber("order-audit-subscriber", "Order Audit System", "📋");
        setupSubscriber("order-notification-subscriber", "Order Notification Service", "📧");
        setupSubscriber("order-analytics-subscriber", "Order Analytics Engine", "📊");

        // Payment event subscribers
        setupSubscriber("payment-audit-subscriber", "Payment Audit System", "💰");
        setupSubscriber("payment-notification-subscriber", "Payment Notification Service", "💳");

        // Inventory event subscribers
        setupSubscriber("inventory-audit-subscriber", "Inventory Audit System", "📦");
        setupSubscriber("inventory-reorder-subscriber", "Auto-Reorder Service", "🔄");

        // System event subscribers
        setupSubscriber("system-monitoring-subscriber", "System Monitoring Dashboard", "📈");
        setupSubscriber("system-alert-subscriber", "Alert Management System", "🚨");

        System.out.println("✅ All subscribers configured");
    }

    private void setupSubscriber(String subscriberQueue, String subscriberName, String icon) throws Exception {
        MessageConsumer<TopicMessage> subscriber =
            factory.createConsumer(subscriberQueue, TopicMessage.class);

        subscriber.subscribe(message -> {
            TopicMessage topicMessage = message.getPayload();
            String topic = message.getHeaders().get("topic");
            String eventType = message.getHeaders().get("eventType");

            System.out.printf("%s %s received: %s (Topic: %s, Type: %s)%n",
                icon, subscriberName, topicMessage.getContent(), topic, eventType);

            // Simulate subscriber processing
            return CompletableFuture.runAsync(() -> {
                try {
                    Thread.sleep(100 + (int)(Math.random() * 500)); // Simulate processing time
                    System.out.printf("  ✅ %s processed message%n", subscriberName);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            });
        });

        System.out.printf("  ✅ %s (%s)%n", subscriberName, subscriberQueue);
    }
```

```java
private void demonstratePublishSubscribe() throws Exception {
    System.out.println("🚀 Demonstrating publish-subscribe operations:");

    // Publish order events
    publishOrderEvents();
    Thread.sleep(2000);

    // Publish payment events
    publishPaymentEvents();
    Thread.sleep(2000);

    // Publish inventory events
    publishInventoryEvents();
    Thread.sleep(2000);

    // Publish system events
    publishSystemEvents();
}

private void publishOrderEvents() throws Exception {
    System.out.println("📋 Publishing order events:");

    MessageProducer<TopicMessage> producer = factory.createProducer("order-events", TopicMessage.class);

    TopicMessage[] orderEvents = {
        new TopicMessage("ORDER-001", "Order placed for customer John Doe", "Order placed successfully"),
        new TopicMessage("ORDER-001", "Order payment processed", "Payment of $299.99 processed"),
        new TopicMessage("ORDER-001", "Order shipped", "Order shipped via FedEx, tracking: 1234567890")
    };

    String[] eventTypes = {"order.placed", "order.payment.processed", "order.shipped"};

    for (int i = 0; i < orderEvents.length; i++) {
        producer.send(orderEvents[i], Map.of(
            "eventType", eventTypes[i],
            "timestamp", Instant.now().toString()
        )).join();
        Thread.sleep(500);
    }

    producer.close();
}

private void publishPaymentEvents() throws Exception {
    System.out.println("💳 Publishing payment events:");

    MessageProducer<TopicMessage> producer = factory.createProducer("payment-events", TopicMessage.class);

    TopicMessage[] paymentEvents = {
        new TopicMessage("PAY-001", "Payment authorized", "Credit card payment authorized"),
        new TopicMessage("PAY-001", "Payment captured", "Payment of $299.99 captured successfully")
    };

    String[] eventTypes = {"payment.authorized", "payment.captured"};

    for (int i = 0; i < paymentEvents.length; i++) {
        producer.send(paymentEvents[i], Map.of(
            "eventType", eventTypes[i],
            "timestamp", Instant.now().toString()
        )).join();
        Thread.sleep(500);
    }

    producer.close();
}
```

```java
    private void publishInventoryEvents() throws Exception {
        System.out.println("📦 Publishing inventory events:");

        MessageProducer<TopicMessage> producer = factory.createProducer("inventory-events", TopicMessage.class);

        TopicMessage[] inventoryEvents = {
            new TopicMessage("ITEM-001", "Stock level low", "Product XYZ stock level is below threshold (5 remaining)"),
            new TopicMessage("ITEM-002", "Stock depleted", "Product ABC is out of stock")
        };

        String[] eventTypes = {"inventory.low", "inventory.depleted"};

        for (int i = 0; i < inventoryEvents.length; i++) {
            producer.send(inventoryEvents[i], Map.of(
                "eventType", eventTypes[i],
                "timestamp", Instant.now().toString()
            )).join();
            Thread.sleep(500);
        }

        producer.close();
    }

    private void publishSystemEvents() throws Exception {
        System.out.println("🖥 Publishing system events:");

        MessageProducer<TopicMessage> producer = factory.createProducer("system-events", TopicMessage.class);

        TopicMessage[] systemEvents = {
            new TopicMessage("SYS-001", "High CPU usage detected", "CPU usage is at 85% on server web-01"),
            new TopicMessage("SYS-002", "Database connection pool warning", "Connection pool utilization is at 90%")
        };

        String[] eventTypes = {"system.cpu.high", "system.db.pool.warning"};

        for (int i = 0; i < systemEvents.length; i++) {
            producer.send(systemEvents[i], Map.of(
                "eventType", eventTypes[i],
                "timestamp", Instant.now().toString()
            )).join();
            Thread.sleep(500);
        }

        producer.close();
    }

    public void close() throws Exception {
        // Close all subscribers
        for (List<MessageConsumer<?>> consumers : topicSubscribers.values()) {
            for (MessageConsumer<?> consumer : consumers) {
                consumer.close();
            }
        }
        factory.close();
    }

    // Supporting classes
    public static class TopicMessage {
        private final String id;
        private final String subject;
        private final String content;

        public TopicMessage(String id, String subject, String content) {
            this.id = id;
            this.subject = subject;
            this.content = content;
        }

        public String getId() { return id; }
        public String getSubject() { return subject; }
        public String getContent() { return content; }
    }
}
```

Production Deployment

This section provides comprehensive guidance for deploying PeeGeeQ in production environments, covering deployment strategies, infrastructure setup, monitoring, and operational best practices.

Deployment Strategies

Blue-Green Deployment

```yaml
# blue-green-deployment.yml
apiVersion: v1
kind: Namespace
metadata:
  name: peegeeq-production
---
# Blue Environment (Current Production)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: peegeeq-blue
  namespace: peegeeq-production
  labels:
    app: peegeeq
    version: blue
    environment: production
spec:
  replicas: 3
  selector:
    matchLabels:
      app: peegeeq
      version: blue
  template:
    metadata:
      labels:
        app: peegeeq
        version: blue
    spec:
      containers:
      - name: peegeeq
        image: peegeeq:v1.2.0
        ports:
        - containerPort: 8080
        env:
        - name: SPRING_PROFILES_ACTIVE
          value: "production"
        - name: PEEGEEQ_DB_HOST
          valueFrom:
            secretKeyRef:
              name: peegeeq-secrets
              key: db-host
        - name: PEEGEEQ_DB_PASSWORD
```

```yaml
            valueFrom:
              secretKeyRef:
                name: peegeeq-secrets
                key: db-password
          resources:
            requests:
              memory: "1Gi"
              cpu: "500m"
            limits:
              memory: "2Gi"
              cpu: "1000m"
          livenessProbe:
            httpGet:
              path: /actuator/health
              port: 8080
            initialDelaySeconds: 60
            periodSeconds: 30
          readinessProbe:
            httpGet:
              path: /actuator/health/readiness
              port: 8080
            initialDelaySeconds: 30
            periodSeconds: 10
---
# Green Environment (New Version)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: peegeeq-green
  namespace: peegeeq-production
  labels:
    app: peegeeq
    version: green
    environment: production
spec:
  replicas: 0  # Initially scaled to 0
  selector:
    matchLabels:
      app: peegeeq
      version: green
  template:
    metadata:
      labels:
        app: peegeeq
        version: green
    spec:
      containers:
      - name: peegeeq
        image: peegeeq:v1.3.0  # New version
        ports:
        - containerPort: 8080
        env:
        - name: SPRING_PROFILES_ACTIVE
          value: "production"
        - name: PEEGEEQ_DB_HOST
          valueFrom:
            secretKeyRef:
              name: peegeeq-secrets
              key: db-host
        - name: PEEGEEQ_DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: peegeeq-secrets
              key: db-password
        resources:
          requests:
            memory: "1Gi"
            cpu: "500m"
          limits:
            memory: "2Gi"
            cpu: "1000m"
        livenessProbe:
          httpGet:
            path: /actuator/health
            port: 8080
          initialDelaySeconds: 60
          periodSeconds: 30
        readinessProbe:
          httpGet:
            path: /actuator/health/readiness
            port: 8080
          initialDelaySeconds: 30
          periodSeconds: 10
---
# Service (switches between blue and green)
apiVersion: v1
kind: Service
metadata:
  name: peegeeq-service
  namespace: peegeeq-production
spec:
  selector:
    app: peegeeq
    version: blue  # Initially points to blue
  ports:
  - port: 80
    targetPort: 8080
  type: LoadBalancer
```

Canary Deployment

```yaml
# canary-deployment.yml
apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata:
  name: peegeeq-rollout
  namespace: peegeeq-production
spec:
  replicas: 10
  strategy:
    canary:
      steps:
      - setWeight: 10    # 10% traffic to new version
      - pause: {duration: 5m}
      - setWeight: 25    # 25% traffic to new version
      - pause: {duration: 10m}
      - setWeight: 50    # 50% traffic to new version
      - pause: {duration: 15m}
      - setWeight: 75    # 75% traffic to new version
      - pause: {duration: 10m}
      # Automatic promotion to 100% if no issues
      canaryService: peegeeq-canary
      stableService: peegeeq-stable
      trafficRouting:
        istio:
          virtualService:
            name: peegeeq-vs
          destinationRule:
            name: peegeeq-dr
      analysis:
        templates:
        - templateName: success-rate
        args:
        - name: service-name
          value: peegeeq-canary
        - name: prometheus-server
          value: http://prometheus:9090
  selector:
    matchLabels:
      app: peegeeq
  template:
    metadata:
      labels:
        app: peegeeq
    spec:
      containers:
      - name: peegeeq
        image: peegeeq:v1.3.0
        ports:
        - containerPort: 8080
        env:
        - name: SPRING_PROFILES_ACTIVE
          value: "production"
        resources:
          requests:
            memory: "1Gi"
            cpu: "500m"
          limits:
            memory: "2Gi"
            cpu: "1000m"
---
# Analysis Template for Canary
apiVersion: argoproj.io/v1alpha1
kind: AnalysisTemplate
metadata:
  name: success-rate
  namespace: peegeeq-production
spec:
  args:
  - name: service-name
  - name: prometheus-server
  metrics:
  - name: success-rate
    interval: 2m
    count: 5
    successCondition: result[0] >= 0.95
    failureLimit: 2
    provider:
      prometheus:
        address: "{{args.prometheus-server}}"
        query: |
          sum(rate(http_requests_total{service="{{args.service-name}}",status!~"5.."}[2m])) /
          sum(rate(http_requests_total{service="{{args.service-name}}"}[2m]))
  - name: error-rate
    interval: 2m
    count: 5
    successCondition: result[0] <= 0.05
    failureLimit: 2
    provider:
      prometheus:
        address: "{{args.prometheus-server}}"
        query: |
          sum(rate(peegeeq_errors_total{service="{{args.service-name}}"}[2m])) /
          sum(rate(peegeeq_messages_total{service="{{args.service-name}}"}[2m]))
```

Infrastructure as Code

Terraform Infrastructure Setup

```
# main.tf - Production Infrastructure
terraform {
  required_version = ">= 1.0"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
    kubernetes = {
      source  = "hashicorp/kubernetes"
      version = "~> 2.0"
    }
  }
}

provider "aws" {
  region = var.aws_region
}

# VPC and Networking
module "vpc" {
  source = "terraform-aws-modules/vpc/aws"

  name = "peegeeq-production-vpc"
  cidr = "10.0.0.0/16"

  azs             = ["${var.aws_region}a", "${var.aws_region}b", "${var.aws_region}c"]
  private_subnets = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]
  public_subnets  = ["10.0.101.0/24", "10.0.102.0/24", "10.0.103.0/24"]

  enable_nat_gateway = true
  enable_vpn_gateway = true

  tags = {
    Environment = "production"
    Application = "peegeeq"
  }
}

# RDS PostgreSQL Cluster
resource "aws_rds_cluster" "peegeeq_db" {
  cluster_identifier      = "peegeeq-production-cluster"
  engine                 = "aurora-postgresql"
  engine_version         = "15.4"
  database_name          = "peegeeq_prod"
  master_username        = "peegeeq_admin"
  master_password        = var.db_password

  vpc_security_group_ids = [aws_security_group.rds.id]
  db_subnet_group_name   = aws_db_subnet_group.peegeeq.name

  backup_retention_period = 30
  preferred_backup_window = "03:00-04:00"
  preferred_maintenance_window = "sun:04:00-sun:05:00"

  storage_encrypted = true
  kms_key_id       = aws_kms_key.peegeeq.arn

  enabled_cloudwatch_logs_exports = ["postgresql"]
  tags = {
    Environment = "production"
    Application = "peegeeq"
  }
}

resource "aws_rds_cluster_instance" "peegeeq_db_instances" {
  count              = 3
  identifier         = "peegeeq-production-${count.index}"
  cluster_identifier = aws_rds_cluster.peegeeq_db.id
  instance_class     = "db.r6g.xlarge"
  engine             = aws_rds_cluster.peegeeq_db.engine
  engine_version     = aws_rds_cluster.peegeeq_db.engine_version

  performance_insights_enabled = true
  monitoring_interval         = 60
  monitoring_role_arn        = aws_iam_role.rds_monitoring.arn

  tags = {
    Environment = "production"
    Application = "peegeeq"
  }
}

# EKS Cluster
module "eks" {
  source = "terraform-aws-modules/eks/aws"

  cluster_name    = "peegeeq-production"
  cluster_version = "1.28"

  vpc_id     = module.vpc.vpc_id
  subnet_ids = module.vpc.private_subnets

  cluster_endpoint_private_access = true
  cluster_endpoint_public_access  = true

  cluster_addons = {
    coredns = {
      resolve_conflicts = "OVERWRITE"
    }
    kube-proxy = {}
    vpc-cni = {
      resolve_conflicts = "OVERWRITE"
    }
    aws-ebs-csi-driver = {}
  }

  eks_managed_node_groups = {
    peegeeq_nodes = {
      min_size     = 3
      max_size     = 10
      desired_size = 6

      instance_types = ["c5.xlarge"]
      capacity_type  = "ON_DEMAND"

      k8s_labels = {
        Environment = "production"
        Application = "peegeeq"
      }

      update_config = {
        max_unavailable_percentage = 25
      }
    }
  }

  tags = {
    Environment = "production"
    Application = "peegeeq"
  }
}

# Security Groups
resource "aws_security_group" "rds" {
  name_prefix = "peegeeq-rds-"
  vpc_id      = module.vpc.vpc_id

  ingress {
    from_port   = 5432
    to_port     = 5432
    protocol    = "tcp"
    cidr_blocks = [module.vpc.vpc_cidr_block]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name = "peegeeq-rds-sg"
    Environment = "production"
  }
}

# KMS Key for Encryption
resource "aws_kms_key" "peegeeq" {
  description             = "PeeGeeQ Production Encryption Key"
  deletion_window_in_days = 7

  tags = {
    Environment = "production"
    Application = "peegeeq"
  }
}

resource "aws_kms_alias" "peegeeq" {
  name          = "alias/peegeeq-production"
  target_key_id = aws_kms_key.peegeeq.key_id
}

# CloudWatch Log Groups
resource "aws_cloudwatch_log_group" "peegeeq_app" {
  name              = "/aws/eks/peegeeq-production/application"
  retention_in_days = 90

  tags = {
    Environment = "production"
    Application = "peegeeq"
  }
}

# S3 Bucket for Backups
resource "aws_s3_bucket" "peegeeq_backups" {
  bucket = "peegeeq-production-backups-${random_id.bucket_suffix.hex}"

  tags = {
    Environment = "production"
    Application = "peegeeq"
  }
}

resource "aws_s3_bucket_versioning" "peegeeq_backups" {
  bucket = aws_s3_bucket.peegeeq_backups.id
  versioning_configuration {
    status = "Enabled"
  }
}

resource "aws_s3_bucket_encryption" "peegeeq_backups" {
  bucket = aws_s3_bucket.peegeeq_backups.id

  server_side_encryption_configuration {
    rule {
      apply_server_side_encryption_by_default {
        kms_master_key_id = aws_kms_key.peegeeq.arn
        sse_algorithm     = "aws:kms"
      }
    }
  }
}

resource "random_id" "bucket_suffix" {
  byte_length = 4
}

# Outputs
output "cluster_endpoint" {
  description = "Endpoint for EKS control plane"
  value       = module.eks.cluster_endpoint
}

output "cluster_security_group_id" {
  description = "Security group ids attached to the cluster control plane"
  value       = module.eks.cluster_security_group_id
}

output "rds_cluster_endpoint" {
  description = "RDS cluster endpoint"
  value       = aws_rds_cluster.peegeeq_db.endpoint
}

output "rds_cluster_reader_endpoint" {
  description = "RDS cluster reader endpoint"
  value       = aws_rds_cluster.peegeeq_db.reader_endpoint
}
```

Automated Deployment Pipeline

GitLab CI/CD Pipeline

```yaml
# .gitlab-ci.yml
stages:
  - build
  - test
  - security-scan
  - deploy-staging
  - integration-tests
  - deploy-production
  - post-deployment
```

```yaml
variables:
  DOCKER_REGISTRY: "your-registry.com"
  APP_NAME: "peegeeq"
  KUBECONFIG_FILE: $KUBECONFIG_PRODUCTION

# Build Stage
build:
  stage: build
  image: docker:20.10.16
  services:
    - docker:20.10.16-dind
  before_script:
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
  script:
    - docker build -t $DOCKER_REGISTRY/$APP_NAME:$CI_COMMIT_SHA .
    - docker build -t $DOCKER_REGISTRY/$APP_NAME:latest .
    - docker push $DOCKER_REGISTRY/$APP_NAME:$CI_COMMIT_SHA
    - docker push $DOCKER_REGISTRY/$APP_NAME:latest
  only:
    - main
    - develop

# Unit Tests
unit-tests:
  stage: test
  image: openjdk:21-jdk
  script:
    - ./mvnw clean test
    - ./mvnw jacoco:report
  artifacts:
    reports:
      junit:
        - "**/target/surefire-reports/TEST-*.xml"
      coverage_report:
        coverage_format: jacoco
        path: target/site/jacoco/jacoco.xml
  coverage: '/Total.*?([0-9]{1,3})%/'

# Integration Tests
integration-tests:
  stage: test
  image: openjdk:21-jdk
  services:
    - postgres:15
  variables:
    POSTGRES_DB: peegeeq_test
    POSTGRES_USER: test_user
    POSTGRES_PASSWORD: test_password
    SPRING_PROFILES_ACTIVE: test
  script:
    - ./mvnw clean verify -Pintegration-tests
  artifacts:
    reports:
      junit:
        - "**/target/failsafe-reports/TEST-*.xml"

# Security Scanning
security-scan:
  stage: security-scan
  image: owasp/zap2docker-stable
  script:
    - mkdir -p /zap/wrk/
    - /zap/zap-baseline.py -t http://localhost:8080 -g gen.conf -r testreport.html
  artifacts:
    reports:
      junit: testreport.xml
    paths:
      - testreport.html
  allow_failure: true

# Container Security Scan
container-scan:
  stage: security-scan
  image: aquasec/trivy:latest
  script:
    - trivy image --exit-code 0 --severity HIGH,CRITICAL $DOCKER_REGISTRY/$APP_NAME:$CI_COMMIT_SHA
  allow_failure: true

# Deploy to Staging
deploy-staging:
  stage: deploy-staging
  image: bitnami/kubectl:latest
  environment:
    name: staging
    url: https://peegeeq-staging.company.com
  before_script:
    - echo $KUBECONFIG_STAGING | base64 -d > kubeconfig
    - export KUBECONFIG=kubeconfig
  script:
    - kubectl set image deployment/peegeeq-staging peegeeq=$DOCKER_REGISTRY/$APP_NAME:$CI_COMMIT_SHA -n peegeeq-staging
    - kubectl rollout status deployment/peegeeq-staging -n peegeeq-staging --timeout=300s
    - kubectl get pods -n peegeeq-staging
  only:
    - main

# Staging Integration Tests
staging-tests:
  stage: integration-tests
  image: openjdk:21-jdk
  variables:
    TEST_ENVIRONMENT: staging
    BASE_URL: https://peegeeq-staging.company.com
  script:
    - ./mvnw clean test -Pstaging-tests -Dtest.base.url=$BASE_URL
  artifacts:
    reports:
      junit:
        - "**/target/surefire-reports/TEST-*.xml"
  only:
    - main

# Production Deployment (Manual)
deploy-production:
  stage: deploy-production
  image: bitnami/kubectl:latest
  environment:
    name: production
    url: https://peegeeq.company.com
  before_script:
    - echo $KUBECONFIG_PRODUCTION | base64 -d > kubeconfig
    - export KUBECONFIG=kubeconfig
  script:
    # Blue-Green Deployment
    - |
      # Check current active version
      CURRENT_VERSION=$(kubectl get service peegeeq-service -n peegeeq-production -o jsonpath='{.spec.selector.version}')
      if [ "$CURRENT_VERSION" = "blue" ]; then
        NEW_VERSION="green"
        OLD_VERSION="blue"
      else
        NEW_VERSION="blue"
        OLD_VERSION="green"
      fi

      echo "Deploying to $NEW_VERSION environment"

      # Update the new version deployment
      kubectl set image deployment/peegeeq-$NEW_VERSION peegeeq=$DOCKER_REGISTRY/$APP_NAME:$CI_COMMIT_SHA -n peegeeq-prod
      kubectl scale deployment peegeeq-$NEW_VERSION --replicas=3 -n peegeeq-production
      kubectl rollout status deployment/peegeeq-$NEW_VERSION -n peegeeq-production --timeout=600s

      # Health check
      kubectl wait --for=condition=ready pod -l app=peegeeq,version=$NEW_VERSION -n peegeeq-production --timeout=300s

      # Switch traffic to new version
      kubectl patch service peegeeq-service -n peegeeq-production -p '{"spec":{"selector":{"version":"'$NEW_VERSION'"}}}'

      echo "Traffic switched to $NEW_VERSION"

      # Wait and then scale down old version
      sleep 60
      kubectl scale deployment peegeeq-$OLD_VERSION --replicas=0 -n peegeeq-production

      echo "Deployment completed successfully"
  when: manual
  only:
    - main

# Post-Deployment Health Checks
health-check:
  stage: post-deployment
  image: curlimages/curl:latest
  script:
    - |
      echo "Performing post-deployment health checks..."

      # Wait for service to be ready
      sleep 30

      # Health check
      curl -f https://peegeeq.company.com/actuator/health || exit 1

      # Readiness check
      curl -f https://peegeeq.company.com/actuator/health/readiness || exit 1

      # Basic functionality test
      curl -f https://peegeeq.company.com/api/v1/health || exit 1

      echo "All health checks passed!"
  only:
    - main
  when: on_success

# Rollback (Manual)
rollback-production:
  stage: deploy-production
  image: bitnami/kubectl:latest
  environment:
    name: production
    url: https://peegeeq.company.com
  before_script:
    - echo $KUBECONFIG_PRODUCTION | base64 -d > kubeconfig
    - export KUBECONFIG=kubeconfig
  script:
    - |
      echo "Rolling back production deployment..."

      # Get current and previous versions
```

```
        CURRENT_VERSION=$(kubectl get service peegeeq-service -n peegeeq-production -o jsonpath='{.spec.selector.version}')
        if [ "$CURRENT_VERSION" = "blue" ]; then
          ROLLBACK_VERSION="green"
        else
          ROLLBACK_VERSION="blue"
        fi

        echo "Rolling back to $ROLLBACK_VERSION"

        # Scale up rollback version
        kubectl scale deployment peegeeq-$ROLLBACK_VERSION --replicas=3 -n peegeeq-production
        kubectl rollout status deployment/peegeeq-$ROLLBACK_VERSION -n peegeeq-production --timeout=300s

        # Switch traffic back
        kubectl patch service peegeeq-service -n peegeeq-production -p '{"spec":{"selector":{"version":"'$ROLLBACK_VERSION'

        # Scale down current version
        kubectl scale deployment peegeeq-$CURRENT_VERSION --replicas=0 -n peegeeq-production

        echo "Rollback completed successfully"
    when: manual
    only:
      - main
```

🎯 **Try This Now**:

1. Set up a blue-green deployment pipeline for your environment
2. Configure infrastructure as code with Terraform
3. Implement automated health checks and rollback procedures
4. Set up comprehensive monitoring and alerting for production

Advanced Features Summary

The advanced features covered in this section provide enterprise-grade capabilities for production deployments:

Key Enterprise Features

- **Advanced Messaging Patterns**: High-frequency messaging, message routing by headers
- **Message Priority Handling**: Sophisticated priority-based processing
- **Enhanced Error Handling**: Retry strategies, circuit breakers, dead letter queues
- **System Properties Configuration**: Runtime tuning for performance and reliability
- **Security Configuration**: SSL/TLS, credential management, compliance features
- **Consumer Groups & Load Balancing**: Scalable message processing with fault tolerance
- **Service Discovery & Federation**: Multi-instance management with Consul integration
- **REST API & HTTP Integration**: HTTP-based queue operations and management
- **Production Readiness**: Health checks, circuit breakers, comprehensive metrics
- **Performance Optimization**: Connection pooling, batch processing, concurrent processing
- **Integration Patterns**: Request-reply, publish-subscribe, message routing, CQRS, Saga patterns

Production Deployment Checklist

- [ ] **Database Setup**: PostgreSQL cluster with replication
- [ ] **Connection Pooling**: Optimized pool settings for workload
- [ ] **SSL/TLS**: Encrypted database connections

- [ ] **Monitoring**: Prometheus + Grafana dashboards configured
- [ ] **Alerting**: Critical alerts configured and tested
- [ ] **Health Checks**: All health checks passing
- [ ] **Circuit Breakers**: Configured with appropriate thresholds
- [ ] **Dead Letter Queue**: DLQ monitoring and reprocessing procedures
- [ ] **Backup Strategy**: Database backup and recovery procedures
- [ ] **Security**: Network security, authentication, and authorization
- [ ] **Load Testing**: Performance validated under expected load
- [ ] **Disaster Recovery**: Failover procedures documented and tested

For detailed implementation examples and comprehensive coverage of all advanced features, refer to the individual example classes mentioned throughout this section.

License

PeeGeeQ is licensed under the Apache License, Version 2.0. See the `LICENSE` file for details.

Spring Boot Integration

PeeGeeQ provides seamless integration with Spring Boot applications through configuration classes and Spring beans. This section shows you how to set up a complete "Hello World" application with both producers and consumers.

Quick Start: Hello World Spring Boot App

1. Maven Dependencies

Add PeeGeeQ to your Spring Boot project:

```xml
<dependencies>
    <!-- Spring Boot Starter -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- PeeGeeQ Dependencies -->
    <dependency>
        <groupId>dev.mars.peegeeq</groupId>
        <artifactId>peegeeq-outbox</artifactId>
        <version>1.0.0</version>
    </dependency>
    <dependency>
        <groupId>dev.mars.peegeeq</groupId>
        <artifactId>peegeeq-native</artifactId>
        <version>1.0.0</version>
    </dependency>

    <!-- Micrometer for metrics -->
    <dependency>
        <groupId>io.micrometer</groupId>
        <artifactId>micrometer-registry-prometheus</artifactId>
    </dependency>
```

```
</dependencies>
```

2. Application Configuration (application.yml)

```yaml
# PeeGeeQ Configuration
peegeeq:
  profile: production
  database:
    host: ${DB_HOST:localhost}
    port: ${DB_PORT:5432}
    name: ${DB_NAME:hello_world}
    username: ${DB_USERNAME:peegeeq_user}
    password: ${DB_PASSWORD:peegeeq_password}
    schema: public
  pool:
    max-size: 20
    min-size: 5
  queue:
    max-retries: 3
    visibility-timeout: PT30S
    batch-size: 10
    polling-interval: PT1S

# Spring Boot Configuration
spring:
  application:
    name: peegeeq-hello-world
  jackson:
    serialization:
      write-dates-as-timestamps: false

# Server Configuration
server:
  port: 8080

# Management endpoints
management:
  endpoints:
    web:
      exposure:
        include: health,info,metrics,prometheus

# Logging
logging:
  level:
    root: INFO
    dev.mars.peegeeq: DEBUG
```

3. Configuration Properties Class

```java
@ConfigurationProperties(prefix = "peegeeq")
public class PeeGeeQProperties {
    private String profile = "production";
    private Database database = new Database();
    private Pool pool = new Pool();
    private Queue queue = new Queue();

    // Getters and setters
    public String getProfile() { return profile; }
    public void setProfile(String profile) { this.profile = profile; }
    public Database getDatabase() { return database; }
    public void setDatabase(Database database) { this.database = database; }

    public Pool getPool() { return pool; }
    public void setPool(Pool pool) { this.pool = pool; }

    public Queue getQueue() { return queue; }
    public void setQueue(Queue queue) { this.queue = queue; }

    public static class Database {
        private String host = "localhost";
        private int port = 5432;
        private String name = "hello_world";
        private String username = "peegeeq_user";
        private String password = "";
        private String schema = "public";

        // Getters and setters
        public String getHost() { return host; }
        public void setHost(String host) { this.host = host; }

        public int getPort() { return port; }
        public void setPort(int port) { this.port = port; }

        public String getName() { return name; }
        public void setName(String name) { this.name = name; }

        public String getUsername() { return username; }
        public void setUsername(String username) { this.username = username; }

        public String getPassword() { return password; }
        public void setPassword(String password) { this.password = password; }

        public String getSchema() { return schema; }
        public void setSchema(String schema) { this.schema = schema; }
    }

    public static class Pool {
        private int maxSize = 20;
        private int minSize = 5;

        public int getMaxSize() { return maxSize; }
        public void setMaxSize(int maxSize) { this.maxSize = maxSize; }

        public int getMinSize() { return minSize; }
        public void setMinSize(int minSize) { this.minSize = minSize; }
    }

    public static class Queue {
        private int maxRetries = 3;
        private Duration visibilityTimeout = Duration.ofSeconds(30);
        private int batchSize = 10;
        private Duration pollingInterval = Duration.ofSeconds(1);

        public int getMaxRetries() { return maxRetries; }
        public void setMaxRetries(int maxRetries) { this.maxRetries = maxRetries; }

        public Duration getVisibilityTimeout() { return visibilityTimeout; }
        public void setVisibilityTimeout(Duration visibilityTimeout) { this.visibilityTimeout = visibilityTimeout; }

        public int getBatchSize() { return batchSize; }
        public void setBatchSize(int batchSize) { this.batchSize = batchSize; }

        public Duration getPollingInterval() { return pollingInterval; }
        public void setPollingInterval(Duration pollingInterval) { this.pollingInterval = pollingInterval; }
    }
```

```
}
```

4. PeeGeeQ Spring Configuration

```
@Configuration
@EnableConfigurationProperties(PeeGeeQProperties.class)
public class PeeGeeQConfig {
    private static final Logger log = LoggerFactory.getLogger(PeeGeeQConfig.class);

    /**
     * Creates and configures the PeeGeeQ Manager as a Spring bean.
     */
    @Bean
    @Primary
    public PeeGeeQManager peeGeeQManager(PeeGeeQProperties properties, MeterRegistry meterRegistry) {
        log.info("Creating PeeGeeQ Manager with profile: {}", properties.getProfile());

        // Configure system properties from Spring configuration
        configureSystemProperties(properties);

        PeeGeeQConfiguration config = new PeeGeeQConfiguration(properties.getProfile());
        PeeGeeQManager manager = new PeeGeeQManager(config, meterRegistry);

        // Start the manager - this handles all Vert.x setup internally
        manager.start();
        log.info("PeeGeeQ Manager started successfully");

        return manager;
    }

    /**
     * Creates the outbox factory for transactional outbox operations.
     */
    @Bean
    public QueueFactory outboxFactory(PeeGeeQManager manager) {
        log.info("Creating outbox factory");

        DatabaseService databaseService = new PgDatabaseService(manager);
        QueueFactoryProvider provider = new PgQueueFactoryProvider();

        // Register outbox factory implementation
        OutboxFactoryRegistrar.registerWith((QueueFactoryRegistrar) provider);

        QueueFactory factory = provider.createFactory("outbox", databaseService);
        log.info("Outbox factory created successfully");

        return factory;
    }

    /**
     * Creates the native queue factory for real-time messaging.
     */
    @Bean
    public QueueFactory nativeFactory(PeeGeeQManager manager) {
        log.info("Creating native queue factory");

        DatabaseService databaseService = new PgDatabaseService(manager);
        QueueFactoryProvider provider = new PgQueueFactoryProvider();

        // Register native factory implementation
        PgNativeFactoryRegistrar.registerWith((QueueFactoryRegistrar) provider);

        QueueFactory factory = provider.createFactory("native", databaseService);
        log.info("Native queue factory created successfully");

        return factory;
    }

    /**
     * Creates a hello world message producer.
     */
    @Bean
    public OutboxProducer<String> helloWorldProducer(@Qualifier("outboxFactory") QueueFactory factory) {
        log.info("Creating hello world producer");
        OutboxProducer<String> producer = (OutboxProducer<String>) factory.createProducer("hello-world", String.class);
        log.info("Hello world producer created successfully");
        return producer;
    }

    /**
     * Creates a hello world message consumer.
     */
    @Bean
    public MessageConsumer<String> helloWorldConsumer(@Qualifier("nativeFactory") QueueFactory factory) {
        log.info("Creating hello world consumer");
        MessageConsumer<String> consumer = factory.createConsumer("hello-world", String.class);
        log.info("Hello world consumer created successfully");
        return consumer;
    }

    private void configureSystemProperties(PeeGeeQProperties properties) {
        System.setProperty("peegeeq.database.host", properties.getDatabase().getHost());
        System.setProperty("peegeeq.database.port", String.valueOf(properties.getDatabase().getPort()));
        System.setProperty("peegeeq.database.name", properties.getDatabase().getName());
        System.setProperty("peegeeq.database.username", properties.getDatabase().getUsername());
        System.setProperty("peegeeq.database.password", properties.getDatabase().getPassword());
        System.setProperty("peegeeq.database.schema", properties.getDatabase().getSchema());

        // Configure pool settings
        System.setProperty("peegeeq.database.pool.max-size", String.valueOf(properties.getPool().getMaxSize()));
        System.setProperty("peegeeq.database.pool.min-size", String.valueOf(properties.getPool().getMinSize()));

        // Configure queue settings
        System.setProperty("peegeeq.queue.max-retries", String.valueOf(properties.getQueue().getMaxRetries()));
        System.setProperty("peegeeq.queue.visibility-timeout", properties.getQueue().getVisibilityTimeout().toString());
        System.setProperty("peegeeq.queue.batch-size", String.valueOf(properties.getQueue().getBatchSize()));
        System.setProperty("peegeeq.queue.polling-interval", properties.getQueue().getPollingInterval().toString());
    }
}
```

5. Hello World Service

```
@Service
public class HelloWorldService {
    private static final Logger log = LoggerFactory.getLogger(HelloWorldService.class);

    private final OutboxProducer<String> helloWorldProducer;

    public HelloWorldService(OutboxProducer<String> helloWorldProducer) {
        this.helloWorldProducer = helloWorldProducer;
    }

    /**
     * Sends a hello world message using the transactional outbox pattern.
     */
    public CompletableFuture<String> sendHelloMessage(String name) {
        log.info("Sending hello message for: {}", name);

        String message = "Hello, " + name + "! Welcome to PeeGeeQ!";

        return helloWorldProducer.sendWithTransaction(
            message,
            TransactionPropagation.CONTEXT
        )
        .thenApply(v -> {
            log.info("Hello message sent successfully for: {}", name);
            return message;
        })
        .exceptionally(error -> {
            log.error("Failed to send hello message for {}: {}", name, error.getMessage(), error);
            throw new RuntimeException("Failed to send hello message", error);
        });
    }
}
```

6. Message Handler Component

```
@Component
public class HelloWorldMessageHandler {
    private static final Logger log = LoggerFactory.getLogger(HelloWorldMessageHandler.class);

    private final MessageConsumer<String> helloWorldConsumer;

    public HelloWorldMessageHandler(MessageConsumer<String> helloWorldConsumer) {
        this.helloWorldConsumer = helloWorldConsumer;
    }

    /**
     * Starts listening for hello world messages.
     * This method is called automatically when the Spring context starts.
     */
    @PostConstruct
    public void startListening() {
        log.info("Starting hello world message handler");

        helloWorldConsumer.subscribe(message -> {
            log.info("📩 Received hello world message: {}", message.getPayload());

            // Process the message (your business logic here)
            processHelloMessage(message.getPayload());

            return CompletableFuture.completedFuture(null);
        });

        log.info("Hello world message handler started successfully");
    }

    /**
     * Processes a hello world message.
     * This is where you would put your actual business logic.
     */
    private void processHelloMessage(String message) {
        // Simulate some processing
        log.info("✅ Processing hello message: {}", message);

        // Your business logic here
        // For example: save to database, call external API, etc.

        log.info("✅ Hello message processed successfully");
    }
}
```

7. REST Controller

```
@RestController
@RequestMapping("/api/hello")
public class HelloWorldController {
    private static final Logger log = LoggerFactory.getLogger(HelloWorldController.class);

    private final HelloWorldService helloWorldService;

    public HelloWorldController(HelloWorldService helloWorldService) {
        this.helloWorldService = helloWorldService;
    }

    /**
     * Sends a hello world message.
     */
    @PostMapping("/send/{name}")
    public CompletableFuture<ResponseEntity<Map<String, String>>> sendHello(@PathVariable String name) {
        log.info("REST request to send hello message for: {}", name);

        return helloWorldService.sendHelloMessage(name)
            .thenApply(message -> {
                Map<String, String> response = Map.of(
                    "status", "success",
                    "message", message,
                    "timestamp", Instant.now().toString()
                );
                return ResponseEntity.ok(response);
            })
            .exceptionally(error -> {
                log.error("REST request failed for {}: {}", name, error.getMessage(), error);
                Map<String, String> errorResponse = Map.of(
                    "status", "error",
                    "message", "Failed to send hello message: " + error.getMessage(),
                    "timestamp", Instant.now().toString()
                );
                return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(errorResponse);
            });
    }

    /**
     * Health check endpoint.
     */
    @GetMapping("/health")
    public ResponseEntity<Map<String, String>> health() {
        Map<String, String> response = Map.of(
            "status", "healthy",
            "service", "hello-world",
            "timestamp", Instant.now().toString()
        );
        return ResponseEntity.ok(response);
    }
}
```

8. Main Application Class

```
@SpringBootApplication
@EnableAsync
```

```java
public class HelloWorldApplication {
    private static final Logger log = LoggerFactory.getLogger(HelloWorldApplication.class);

    public static void main(String[] args) {
        log.info("Starting PeeGeeQ Hello World Application");
        SpringApplication.run(HelloWorldApplication.class, args);
        log.info("PeeGeeQ Hello World Application started successfully");
        log.info("Try: POST http://localhost:8080/api/hello/send/YourName");
    }

    /**
     * Configure async task executor for reactive operations.
     */
    @Bean
    public TaskExecutor taskExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(4);
        executor.setMaxPoolSize(8);
        executor.setQueueCapacity(100);
        executor.setThreadNamePrefix("hello-world-async-");
        executor.setWaitForTasksToCompleteOnShutdown(true);
        executor.setAwaitTerminationSeconds(30);
        executor.initialize();
        return executor;
    }
}
```

Testing Your Hello World App

1. Start the Application

```bash
mvn spring-boot:run
```

2. Send a Hello Message

```bash
curl -X POST http://localhost:8080/api/hello/send/World
```

Response:

```json
{
  "status": "success",
  "message": "Hello, World! Welcome to PeeGeeQ!",
  "timestamp": "2025-09-09T10:30:00Z"
}
```

3. Check the Logs

You should see output like:

```
2025-09-09 10:30:00 [http-nio-8080-exec-1] INFO  HelloWorldService - Sending hello message for: World
2025-09-09 10:30:00 [vert.x-eventloop-thread-0] INFO  HelloWorldService - Hello message sent successfully for: World
2025-09-09 10:30:00 [vert.x-eventloop-thread-1] INFO  HelloWorldMessageHandler - 📩 Received hello world message: Hello,
2025-09-09 10:30:00 [vert.x-eventloop-thread-1] INFO  HelloWorldMessageHandler - ✅ Processing hello message: Hello, Wor
```

```
2025-09-09 10:30:00 [vert.x-eventloop-thread-1] INFO  HelloWorldMessageHandler - ✅ Hello message processed successfully
```

Key Integration Points

✅ **Zero Vert.x Exposure**

- Application developers never see Vert.x code
- All reactive operations are handled internally
- Standard Spring Boot patterns and annotations

✅ **Automatic Configuration**

- PeeGeeQ Manager lifecycle managed by Spring
- System properties configured from application.yml
- Database connections handled automatically

✅ **Transactional Consistency**

- Uses `TransactionPropagation.CONTEXT` for Vert.x transactions
- Does NOT use Spring's `@Transactional` (would conflict)
- All operations within `sendWithTransaction()` are atomic

✅ **Production Ready**

- Metrics integration with Micrometer
- Health checks and monitoring endpoints
- Proper error handling and logging
- Graceful shutdown handling

Advanced Patterns

For more complex scenarios, see the complete Spring Boot example in `peegeeq-examples/src/main/java/dev/mars/peegeeq/examples/springboot/` which demonstrates:

- **Multiple Event Types**: Order events, payment events, inventory events
- **Complex Business Logic**: Multi-step transactional workflows
- **Error Handling**: Rollback scenarios and failure recovery
- **Consumer Groups**: Multiple consumers with filtering
- **Monitoring**: Comprehensive metrics and health checks

Part VIII: Troubleshooting & Best Practices

Common Issues & Solutions

Issue 1: Consumer Mode Configuration Issues

Symptoms:

- Consumer not receiving messages despite proper setup
- Unexpected polling behavior or notification failures
- Performance issues with consumer modes

Solutions:

1. **Verify Consumer Mode Configuration**

```
// ✅ Correct - Explicit consumer mode configuration
ConsumerConfig config = ConsumerConfig.builder()
    .mode(ConsumerMode.HYBRID) // Best of both worlds
    .pollingInterval(Duration.ofSeconds(1))
    .batchSize(10)
    .consumerThreads(2)
    .build();

MessageConsumer<String> consumer = factory.createConsumer("orders", String.class, config);
```

2. **Check Database LISTEN/NOTIFY Support**

```
// Test LISTEN/NOTIFY functionality
try (Connection conn = dataSource.getConnection()) {
    try (Statement stmt = conn.createStatement()) {
        stmt.execute("LISTEN test_channel");
        stmt.execute("NOTIFY test_channel, 'test_message'");
        System.out.println("✅ LISTEN/NOTIFY working correctly");
    }
} catch (SQLException e) {
    System.err.println("❌ LISTEN/NOTIFY not supported: " + e.getMessage());
    // Use POLLING_ONLY mode
}
```

3. **Optimize Consumer Mode for Your Use Case**

```
// High-throughput scenarios
ConsumerConfig highThroughput = ConsumerConfig.builder()
    .mode(ConsumerMode.POLLING_ONLY)
    .pollingInterval(Duration.ofMillis(100))
    .batchSize(50)
    .consumerThreads(4)
    .build();

// Real-time scenarios
ConsumerConfig realTime = ConsumerConfig.builder()
    .mode(ConsumerMode.LISTEN_NOTIFY_ONLY)
    .batchSize(1)
    .consumerThreads(1)
    .build();

// Balanced scenarios
ConsumerConfig balanced = ConsumerConfig.builder()
    .mode(ConsumerMode.HYBRID)
    .pollingInterval(Duration.ofSeconds(5))
    .batchSize(10)
    .consumerThreads(2)
    .build();
```

Issue 2: Message Processing Guarantees

Symptoms:

- Duplicate message processing during shutdown
- Messages lost during system failures
- Inconsistent processing behavior

Understanding the Behavior: PeeGeeQ provides **at-least-once delivery** guarantees, which means:

- ✅ **No message loss** - Messages are never lost
- ⚠️ **Possible duplicates** - Messages may be processed more than once
- 🔧 **Idempotent handlers required** - Your message handlers must handle duplicates gracefully

Solutions:

1. **Implement Idempotent Message Handlers**

```
@Component
public class OrderProcessor {

    @Autowired
    private OrderRepository orderRepository;

    public CompletableFuture<Void> processOrder(Message<OrderEvent> message) {
        OrderEvent order = message.getPayload();

        // ✅ Idempotent processing using database constraints
        try {
            orderRepository.insertOrder(order); // UNIQUE constraint prevents duplicates
            System.out.printf("✅ Order processed: %s%n", order.getOrderId());
        } catch (DuplicateKeyException e) {
            System.out.printf("⚠️ Order already processed: %s%n", order.getOrderId());
            // This is normal and expected - not an error
        }

        return CompletableFuture.completedFuture(null);
    }
}
```

2. **Use Explicit Deduplication**

```
@Component
public class PaymentProcessor {

    private final Set<String> processedMessages = ConcurrentHashMap.newKeySet();

    public CompletableFuture<Void> processPayment(Message<PaymentEvent> message) {
        String messageId = message.getId();

        // ✅ Explicit deduplication
        if (processedMessages.contains(messageId)) {
            System.out.printf("⚠️ Payment already processed: %s%n", messageId);
            return CompletableFuture.completedFuture(null);
        }

        try {
            // Process payment
```

```
                processPaymentInternal(message.getPayload());
                processedMessages.add(messageId);
                System.out.printf("✅ Payment processed: %s%n", messageId);
            } catch (Exception e) {
                // Don't add to processed set on failure
                throw e;
            }

            return CompletableFuture.completedFuture(null);
        }
    }
```

3. **Monitor Duplicate Processing Rates**

```
@Component
public class DuplicateMonitor {

    private final MeterRegistry meterRegistry;
    private final Counter duplicateCounter;

    public DuplicateMonitor(MeterRegistry meterRegistry) {
        this.meterRegistry = meterRegistry;
        this.duplicateCounter = Counter.builder("peegeeq.messages.duplicates")
            .description("Number of duplicate messages detected")
            .register(meterRegistry);
    }

    public void recordDuplicate(String messageType) {
        duplicateCounter.increment(Tags.of("type", messageType));
    }
}
```

Issue 3: Configuration Management Problems

Symptoms:

- Configuration not loading from properties files
- System properties not taking effect
- Environment-specific configuration issues

Solutions:

1. **Verify Configuration Loading Order**

```
// ✅ Correct configuration loading
public class ConfigurationExample {

    public void demonstrateConfigurationPrecedence() {
        // 1. System properties (highest priority)
        System.setProperty("peegeeq.queue.batchSize", "25");

        // 2. Environment variables
        // PEEGEEQ_QUEUE_BATCHSIZE=20

        // 3. Properties file (lowest priority)
        // peegeeq.queue.batchSize=15

        PeeGeeQConfiguration config = new PeeGeeQConfiguration("production");
        // Result: batchSize = 25 (system property wins)
        System.out.println("Batch size: " + config.getQueueConfig().getBatchSize());
    }
}
```

2. **Debug Configuration Loading**

```
public class ConfigurationDebugger {

    public void debugConfiguration() {
        PeeGeeQConfiguration config = new PeeGeeQConfiguration("development");

        System.out.println("=== PeeGeeQ Configuration Debug ===");
        System.out.println("Profile: " + config.getProfile());
        System.out.println("Database URL: " + config.getDatabaseConfig().getUrl());
        System.out.println("Batch Size: " + config.getQueueConfig().getBatchSize());
        System.out.println("Polling Interval: " + config.getQueueConfig().getPollingInterval());
        System.out.println("Consumer Threads: " + config.getQueueConfig().getConsumerThreads());

        // Check system properties
        System.out.println("\n=== System Properties ===");
        System.getProperties().entrySet().stream()
            .filter(entry -> entry.getKey().toString().startsWith("peegeeq"))
            .forEach(entry -> System.out.println(entry.getKey() + " = " + entry.getValue()));

        // Check environment variables
        System.out.println("\n=== Environment Variables ===");
        System.getenv().entrySet().stream()
            .filter(entry -> entry.getKey().startsWith("PEEGEEQ"))
            .forEach(entry -> System.out.println(entry.getKey() + " = " + entry.getValue()));
    }
}
```

3. **Validate Configuration at Startup**

```
@Component
public class ConfigurationValidator {

    @EventListener(ApplicationReadyEvent.class)
    public void validateConfiguration() {
        try {
            PeeGeeQConfiguration config = new PeeGeeQConfiguration();

            // Validate critical settings
            if (config.getQueueConfig().getBatchSize() <= 0) {
                throw new IllegalStateException("Batch size must be positive");
            }

            if (config.getQueueConfig().getPollingInterval().isNegative()) {
                throw new IllegalStateException("Polling interval must be positive");
            }

            if (config.getQueueConfig().getConsumerThreads() <= 0) {
                throw new IllegalStateException("Consumer threads must be positive");
            }

            System.out.println("✅ Configuration validation passed");

        } catch (Exception e) {
            System.err.println("❌ Configuration validation failed: " + e.getMessage());
            throw new IllegalStateException("Invalid configuration", e);
        }
        }
    }
```

Best Practices Checklist

✅ Development Best Practices

Message Handler Design

- [ ] **Implement idempotent handlers** - Handle duplicate messages gracefully
- [ ] **Use proper error handling** - Return failed futures for retry scenarios
- [ ] **Keep handlers lightweight** - Avoid heavy processing in message handlers
- [ ] **Use async processing** - Return CompletableFuture for non-blocking operations
- [ ] **Log processing events** - Include message IDs and correlation IDs in logs

Consumer Configuration

- [ ] **Choose appropriate consumer mode** - HYBRID for most use cases
- [ ] **Configure proper batch sizes** - Balance throughput vs. latency
- [ ] **Set reasonable polling intervals** - Avoid too frequent polling
- [ ] **Use multiple consumer threads** - For CPU-intensive processing
- [ ] **Monitor consumer performance** - Track processing rates and errors

Producer Best Practices

- [ ] **Include correlation IDs** - For message tracing and debugging
- [ ] **Use meaningful headers** - Add context information in headers
- [ ] **Handle send failures** - Implement proper retry logic
- [ ] **Batch messages when possible** - Improve throughput for bulk operations
- [ ] **Close producers properly** - Use try-with-resources or explicit close()

✅ Configuration Best Practices

Environment Management

- [ ] **Use profile-specific configurations** - development, staging, production
- [ ] **Externalize sensitive data** - Use environment variables for secrets
- [ ] **Document configuration options** - Maintain configuration documentation
- [ ] **Validate configuration at startup** - Fail fast on invalid configuration
- [ ] **Use configuration templates** - Standardize across environments

Database Configuration

- [ ] **Configure connection pooling** - Optimize pool sizes for workload
- [ ] **Enable SSL/TLS** - Encrypt database connections in production
- [ ] **Set appropriate timeouts** - Connection and query timeouts
- [ ] **Monitor connection usage** - Track pool utilization and leaks
- [ ] **Use read replicas** - For read-heavy workloads

Performance Configuration

- [ ] **Tune batch sizes** - Balance memory usage and throughput
- [ ] **Configure appropriate timeouts** - Visibility timeout, lock timeout
- [ ] **Set consumer thread counts** - Match CPU cores and workload
- [ ] **Enable metrics collection** - Monitor performance indicators
- [ ] **Configure circuit breakers** - Protect against cascading failures

✅ Production Best Practices

Monitoring & Observability

- [ ] **Set up comprehensive metrics** - Prometheus + Grafana dashboards
- [ ] **Configure alerting** - Critical alerts for failures and performance
- [ ] **Implement distributed tracing** - OpenTelemetry integration
- [ ] **Monitor database health** - Connection pool, query performance
- [ ] **Track message processing rates** - Throughput and latency metrics

Security

- [ ] **Encrypt database connections** - SSL/TLS for all connections
- [ ] **Secure message content** - Encrypt sensitive message data
- [ ] **Implement authentication** - Secure access to management APIs
- [ ] **Use principle of least privilege** - Minimal database permissions
- [ ] **Audit message access** - Log all message operations

Reliability & Resilience

- [ ] **Implement circuit breakers** - Protect against cascading failures
- [ ] **Configure retry policies** - Exponential backoff for transient failures
- [ ] **Set up dead letter queues** - Handle permanently failed messages
- [ ] **Monitor error rates** - Alert on high failure rates
- [ ] **Test failure scenarios** - Chaos engineering and disaster recovery

Scalability

- [ ] **Use consumer groups** - Distribute load across multiple consumers
- [ ] **Implement horizontal scaling** - Scale consumers based on load
- [ ] **Monitor resource usage** - CPU, memory, database connections
- [ ] **Plan for growth** - Capacity planning and load testing
- [ ] **Optimize database queries** - Index optimization and query tuning

✅ Operational Best Practices

Deployment

- [ ] **Use blue-green deployments** - Zero-downtime deployments
- [ ] **Implement health checks** - Kubernetes/Docker health endpoints
- [ ] **Configure graceful shutdown** - Proper cleanup on termination

- [ ] **Version your schemas** - Database migration strategies
- [ ] **Test deployments** - Staging environment validation

Maintenance

- [ ] **Regular database maintenance** - VACUUM, ANALYZE, index maintenance
- [ ] **Monitor disk usage** - Queue table growth and cleanup
- [ ] **Archive old messages** - Implement message retention policies
- [ ] **Update dependencies** - Keep libraries and frameworks current
- [ ] **Review performance regularly** - Periodic performance audits

Documentation

- [ ] **Document architecture decisions** - ADRs for major decisions
- [ ] **Maintain runbooks** - Operational procedures and troubleshooting
- [ ] **Document configuration** - All configuration options and defaults
- [ ] **Keep examples current** - Update code examples with API changes
- [ ] **Document integration patterns** - How to integrate with other systems

Anti-patterns to Avoid

❌ Don't: Create New Managers for Each Operation

```
// ❌ Wrong - creates new connections repeatedly
public class BadMessageService {
    public void sendMessage(String message) {
        PeeGeeQManager manager = new PeeGeeQManager(); // New manager each time!
        manager.start();

        QueueFactory factory = provider.createFactory("native", manager.getDatabaseService());
        MessageProducer<String> producer = factory.createProducer("queue", String.class);
        producer.send(message);

        manager.stop(); // Expensive cleanup each time!
    }
}
```

```
// ✅ Correct - reuse manager instance
@Component
public class GoodMessageService {
    private final PeeGeeQManager manager;
    private final QueueFactory factory;

    public GoodMessageService() {
        this.manager = new PeeGeeQManager();
        this.manager.start();
        this.factory = provider.createFactory("native", manager.getDatabaseService());
    }

    public void sendMessage(String message) {
        MessageProducer<String> producer = factory.createProducer("queue", String.class);
        producer.send(message);
        producer.close(); // Only close producer, not manager
    }
    @PreDestroy
    public void cleanup() {
        manager.stop();
    }
}
```

❌ Don't: Ignore Message Processing Failures

```
// ❌ Wrong - silently ignore failures
consumer.subscribe(message -> {
    try {
        processMessage(message.getPayload());
        return CompletableFuture.completedFuture(null);
    } catch (Exception e) {
        // Silently ignoring error - message will be lost!
        return CompletableFuture.completedFuture(null);
    }
});
```

```
// ✅ Correct - handle failures appropriately
consumer.subscribe(message -> {
    try {
        processMessage(message.getPayload());
        return CompletableFuture.completedFuture(null);
    } catch (Exception e) {
        logger.error("Failed to process message: " + message.getId(), e);
        // Return failed future to trigger retry or DLQ
        return CompletableFuture.failedFuture(e);
    }
});
```

❌ Don't: Use Wrong Consumer Mode for Your Use Case

```
// ❌ Wrong - using LISTEN_NOTIFY_ONLY for high-throughput batch processing
ConsumerConfig config = ConsumerConfig.builder()
    .mode(ConsumerMode.LISTEN_NOTIFY_ONLY) // Not optimal for batches
    .batchSize(100) // Large batch size wasted
    .build();

// This will process messages one by one, ignoring batch size
```

```
// ✅ Correct - use POLLING_ONLY or HYBRID for batch processing
ConsumerConfig config = ConsumerConfig.builder()
    .mode(ConsumerMode.POLLING_ONLY) // Better for batches
    .batchSize(100) // Batch size will be utilized
    .pollingInterval(Duration.ofSeconds(1))
    .build();

// This will efficiently process messages in batches
```

❌ Don't: Block Message Handlers with Synchronous Operations

```java
// ❌ Wrong - blocking operations in message handler
consumer.subscribe(message -> {
    // This blocks the event loop!
    String result = callExternalApiSynchronously(message.getPayload());
    saveToDatabase(result);
    return CompletableFuture.completedFuture(null);
});
```

```java
// ✅ Correct - use async operations
consumer.subscribe(message -> {
    return CompletableFuture
        .supplyAsync(() -> callExternalApiSynchronously(message.getPayload()))
        .thenCompose(result -> saveToDatabase(result))
        .thenApply(result -> null);
});
```

❌ Don't: Forget to Handle Duplicate Messages

```java
// ❌ Wrong - not handling duplicates
@Component
public class OrderProcessor {

    public CompletableFuture<Void> processOrder(Message<OrderEvent> message) {
        OrderEvent order = message.getPayload();

        // This will fail on duplicate processing!
        orderRepository.insertOrder(order);
        chargeCustomer(order.getAmount());
        sendConfirmationEmail(order.getCustomerEmail());

        return CompletableFuture.completedFuture(null);
    }
}
```

```java
// ✅ Correct - idempotent processing
@Component
public class OrderProcessor {

    public CompletableFuture<Void> processOrder(Message<OrderEvent> message) {
        OrderEvent order = message.getPayload();

        // Check if already processed
        if (orderRepository.existsByOrderId(order.getOrderId())) {
            logger.info("Order already processed: {}", order.getOrderId());
            return CompletableFuture.completedFuture(null);
        }

        // Process idempotently
        orderRepository.insertOrder(order);
        chargeCustomer(order.getAmount());
        sendConfirmationEmail(order.getCustomerEmail());

        return CompletableFuture.completedFuture(null);
    }
}
```

❌ Don't: Use Inappropriate Queue Types

```java
// ❌ Wrong - using outbox for high-frequency events
QueueFactory factory = provider.createFactory("outbox", databaseService);
MessageProducer<LogEvent> producer = factory.createProducer("logs", LogEvent.class);

// This will be slow for high-frequency logging
for (int i = 0; i < 10000; i++) {
    producer.send(new LogEvent("Log message " + i));
}
```

```java
// ✅ Correct - use native queue for high-frequency events
QueueFactory factory = provider.createFactory("native", databaseService);
MessageProducer<LogEvent> producer = factory.createProducer("logs", LogEvent.class);

// Much faster for high-frequency events
for (int i = 0; i < 10000; i++) {
    producer.send(new LogEvent("Log message " + i));
}
```

Conclusion

Congratulations! You've completed the comprehensive PeeGeeQ guide. You now have the knowledge and tools to:

🎯 What You've Learned

✅ **Message Queue Fundamentals** - Understanding of async messaging patterns ✅ **PeeGeeQ Architecture** - Deep knowledge of native and outbox patterns ✅ **Production Implementation** - Real-world examples and best practices ✅ **Advanced Features** - Consumer modes, configuration, monitoring ✅ **Integration Patterns** - Enterprise integration patterns with PeeGeeQ ✅ **Troubleshooting Skills** - Common issues and their solutions

🚀 Next Steps

1. **Start Small** - Begin with the Hello World example
2. **Experiment** - Try different consumer modes and configurations
3. **Build Gradually** - Add complexity as you gain experience
4. **Monitor Everything** - Set up comprehensive monitoring from day one
5. **Join the Community** - Contribute to PeeGeeQ development and documentation

📚 Additional Resources

- **PeeGeeQ Architecture & API Reference** - Detailed technical specifications
- **GitHub Repository** - Source code and examples
- **Issue Tracker** - Report bugs and request features
- **Discussions** - Community support and questions

🎉 Ready to Get Started?

Run the self-contained demo to see PeeGeeQ in action:

```
./run-self-contained-demo.sh    # Unix/Linux/macOS
run-self-contained-demo.bat     # Windows
```

Happy messaging with PeeGeeQ! 🚀