PeeGeeQ Examples Guide

Comprehensive guide to all PeeGeeQ examples, covering 95-98% of system functionality with production-ready patterns.

Table of Contents

Overview

The `peegeeq-examples/` module contains **33 comprehensive examples** (18 main examples + 15 advanced test examples) that demonstrate all aspects of PeeGeeQ functionality. These examples are:

- **Self-contained**: Use TestContainers for easy execution
- **Production-ready**: Demonstrate real-world patterns and best practices
- **Comprehensive**: Cover 95-98% of PeeGeeQ functionality
- **Well-documented**: Extensive inline documentation and explanations
- **Progressively Complex**: Organized from simple concepts to advanced patterns

Coverage Analysis

Feature Category	Coverage	Main Examples	Test Examples
Core Messaging	100%	PeeGeeQExample, ConsumerGroupExample	PeeGeeQExampleTest, AdvancedProducerConsumerGroupTest
Priority Handling	100%	MessagePriorityExample	HighFrequencyProducerConsumerTest
Error Handling	100%	EnhancedErrorHandlingExample	ConsumerGroupResilienceTest
Security	95%	SecurityConfigurationExample	-

Feature Category	Coverage	Main Examples	Test Examples
Performance	95%	PerformanceTuningExample	HighFrequencyProducerConsumerTest, NativeVsOutboxComparisonTest
Integration	90%	IntegrationPatternsExample	MultiConfigurationIntegrationTest
Event Sourcing	100%	BiTemporalEventStoreExample, TransactionalBiTemporalExample	BiTemporalEventStoreExampleTest, TransactionalBiTemporalExampleTest
REST API	100%	RestApiExample, RestApiStreamingExample	RestApiExampleTest
Service Discovery	100%	ServiceDiscoveryExample	ServiceDiscoveryExampleTest
Configuration	100%	AdvancedConfigurationExample, MultiConfigurationExample	MultiConfigurationIntegrationTest
Native Features	100%	NativeVsOutboxComparisonExample	NativeQueueFeatureTest
Testing & Utilities	100%	PeeGeeQExampleRunner	PeeGeeQExampleRunnerTest, ShutdownTest, TestContainersShutdownTest

Getting Started

Recommended Starting Point: PeeGeeQExampleRunner

The **PeeGeeQExampleRunner** is your best entry point to explore PeeGeeQ. It runs all examples sequentially with comprehensive reporting:

```
# Run ALL examples in logical order (recommended for first-time users)
mvn compile exec:java -pl peegeeq-examples

# List all available examples with descriptions
mvn compile exec:java@list-examples -pl peegeeq-examples

# Run specific examples only
mvn compile exec:java -Dexec.mainClass="dev.mars.peegeeq.examples.PeeGeeQExampleRunner" -Dexec.args="self-contained rest-
```

Features:

- **Sequential execution** of all 18 examples in logical complexity order
- **Comprehensive reporting** with timing, success rates, and error details
- **Flexible selection** - run all examples or choose specific ones
- **Error resilience** - continues even if individual examples fail
- **Detailed logging** for troubleshooting and learning

Quick Demo: Self-Contained Demo

For a single comprehensive demonstration without running all examples:

```
# Self-contained demo with Docker PostgreSQL (no setup required)
mvn compile exec:java -Dexec.mainClass="dev.mars.peegeeq.examples.PeeGeeQSelfContainedDemo" -pl peegeeq-examples
```

Beginner Examples

Start here if you're new to PeeGeeQ. These examples introduce core concepts with minimal complexity.

1. PeeGeeQSelfContainedDemo RECOMMENDED FIRST

Complexity: Beginner **Purpose**: Complete demonstration of all PeeGeeQ features in a single, self-contained application

Detailed Description: This example provides a comprehensive demonstration of PeeGeeQ's capabilities without requiring any external setup. It automatically starts a PostgreSQL container using TestContainers and showcases all major features in a single execution.

Key Implementation Details:

- **Automatic PostgreSQL Setup**: Creates and configures a PostgreSQL container with optimized settings (shared memory, performance tuning)
- **Configuration Management**: Demonstrates environment-specific configuration with system property overrides
- **Health Check System**: Shows comprehensive health monitoring including database connectivity, connection pool status, and component health aggregation
- **Metrics Collection**: Implements message processing metrics (sent, received, processed, failed), success rate calculations, and queue depth monitoring
- **Circuit Breaker Pattern**: Demonstrates circuit breaker implementation for consumer error handling with automatic recovery
- **Backpressure Management**: Shows how to handle system overload with backpressure mechanisms
- **Dead Letter Queue**: Implements failed message handling with DLQ management, recovery mechanisms, and statistics tracking

Real-World Scenarios Demonstrated:

- Production-ready configuration management with profile-based settings
- Comprehensive monitoring and alerting setup
- Error handling and recovery patterns for distributed systems
- Performance monitoring and optimization techniques

Technical Features Showcased:

- Native queue implementation using PostgreSQL LISTEN/NOTIFY for real-time processing
- Outbox pattern implementation with transactional guarantees
- Bi-temporal event store with temporal queries and event corrections
- Automatic resource cleanup and container lifecycle management

Key Code Patterns:

```
// Automatic PostgreSQL container setup with performance optimizations
PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>("postgres:15.13-alpine3.20")
    .withDatabaseName("peegeeq_demo")
```

```
        .withUsername("peegeeq_demo")
        .withPassword("peegeeq_demo")
        .withSharedMemorySize(256 * 1024 * 1024L)  // 256MB for better performance
        .withReuse(false);  // Fresh container for each run

    // Configure PeeGeeQ to use the container automatically
    System.setProperty("peegeeq.database.host", postgres.getHost());
    System.setProperty("peegeeq.database.port", String.valueOf(postgres.getFirstMappedPort()));
    System.setProperty("peegeeq.database.name", postgres.getDatabaseName());
```

```
    // Comprehensive feature demonstration in a single method
    private static void runDemo() {
        try (PeeGeeQManager manager = new PeeGeeQManager(
                new PeeGeeQConfiguration("demo"), new SimpleMeterRegistry())) {

            manager.start();

            // Demonstrate all major features systematically
            demonstrateConfiguration(manager);     // Configuration management
            demonstrateHealthChecks(manager);      // Health monitoring
            demonstrateMetrics(manager);           // Performance metrics
            demonstrateCircuitBreaker(manager);    // Resilience patterns
            demonstrateBackpressure(manager);      // Load management
            demonstrateDeadLetterQueue(manager);   // Error handling

            monitorSystemBriefly(manager);         // System monitoring
        }
    }
```

```
mvn compile exec:java -Dexec.mainClass="dev.mars.peegeeq.examples.PeeGeeQSelfContainedDemo" -pl peegeeq-examples
```

2. PeeGeeQExample

Complexity: Beginner **Purpose**: Basic producer/consumer patterns with external PostgreSQL database

Detailed Description: This example demonstrates fundamental PeeGeeQ concepts using an external PostgreSQL database. It serves as the foundation for understanding core messaging patterns and configuration management.

Key Implementation Details:

- **Profile-Based Configuration**: Supports development, production, and test profiles with environment-specific settings
- **Database Connection Management**: Shows proper database connection setup, validation, and error handling
- **Basic Producer/Consumer Patterns**: Implements simple message production and consumption workflows
- **Health Monitoring System**: Demonstrates database connectivity monitoring, connection pool health status, and overall system health aggregation
- **Metrics and Monitoring**: Implements message processing statistics, success rate calculations, and performance tracking
- **Error Handling**: Shows basic retry mechanisms, failure handling, and error classification

Configuration Patterns Demonstrated:

- Environment variable support for cloud-native deployments
- System property overrides for containerized environments
- Configuration validation with helpful error messages
- Profile-specific database settings and connection parameters

Real-World Applications:

- Traditional enterprise messaging setup
- Integration with existing PostgreSQL infrastructure
- Development and testing environment configuration
- Basic monitoring and alerting implementation

Prerequisites: Requires external PostgreSQL database with proper user permissions and database creation

Key Code Patterns:

```
// Profile-based configuration with environment detection
private static void runExample(String profile) {
    logger.info("Starting PeeGeeQ with profile: {}", profile);

    // Configuration automatically loads based on profile
    try (PeeGeeQManager manager = new PeeGeeQManager(
            new PeeGeeQConfiguration(profile), new SimpleMeterRegistry())) {

        manager.start();

        // Systematic demonstration of production features
        demonstrateConfiguration(manager);    // Show configuration management
        demonstrateHealthChecks(manager);     // Database connectivity monitoring
        demonstrateMetrics(manager);          // Message processing statistics
        demonstrateCircuitBreaker(manager);   // Error handling patterns
        demonstrateBackpressure(manager);     // Load management
        demonstrateDeadLetterQueue(manager);  // Failed message handling
    }
}
```

```
// Comprehensive health check implementation
private static void demonstrateHealthChecks(PeeGeeQManager manager) {
    logger.info("=== Health Checks Demo ===");

    var healthService = manager.getHealthService();

    // Check overall system health
    HealthStatus overallHealth = healthService.getOverallHealth();
    logger.info("Overall Health: {} ({})",
        overallHealth.getStatus(), overallHealth.getMessage());

    // Check individual component health
    Map<String, HealthStatus> componentHealth = healthService.getComponentHealth();
    componentHealth.forEach((component, status) ->
        logger.info("  {}: {} - {}", component, status.getStatus(), status.getMessage()));
}
```

```
mvn compile exec:java -Dexec.mainClass="dev.mars.peegeeq.examples.PeeGeeQExample" -pl peegeeq-examples
```

3. SimpleConsumerGroupTest

Complexity: Beginner **Purpose**: Basic consumer group testing patterns and load balancing fundamentals

Detailed Description: This example provides a simple introduction to consumer groups, focusing on basic load balancing and message distribution patterns. It serves as a foundation for understanding more complex consumer group scenarios.

Key Implementation Details:

- **Basic Consumer Group Setup**: Demonstrates simple consumer group creation and configuration
- **Load Balancing Mechanisms**: Shows how messages are distributed among multiple consumers
- **Consumer Coordination**: Implements basic consumer group coordination and membership management
- **Message Distribution**: Demonstrates round-robin and other distribution strategies
- **Testing Patterns**: Shows effective testing strategies for distributed messaging systems

Technical Concepts Covered:

- Consumer group lifecycle management
- Message acknowledgment patterns
- Consumer failure handling and recovery
- Basic scaling patterns for message processing

Learning Objectives:

- Understanding consumer group fundamentals
- Message distribution and load balancing concepts
- Testing strategies for distributed systems
- Basic fault tolerance patterns

Key Code Patterns:

```
// Basic consumer group setup and testing
@Test
public void testSimpleConsumerGroup() throws Exception {
    String queueName = "simple-consumer-group-test";
    String groupName = "test-group";
    int messageCount = 20;
    int consumerCount = 3;

    // Create test messages
    List<TestMessage> testMessages = createTestMessages(messageCount);

    // Set up consumer group with multiple consumers
    ConsumerGroup<TestMessage> consumerGroup = queueFactory.createConsumerGroup(
        groupName, queueName, TestMessage.class);

    // Track processed messages for verification
    ConcurrentMap<String, String> processedMessages = new ConcurrentHashMap<>();
    CountDownLatch latch = new CountDownLatch(messageCount);

    // Add consumers to the group
    for (int i = 0; i < consumerCount; i++) {
        String consumerId = "consumer-" + i;
        consumerGroup.addConsumer(consumerId, message -> {
            TestMessage payload = message.getPayload();

            // Record which consumer processed which message
            processedMessages.put(payload.getId(), consumerId);

            logger.info("Consumer {} processed message: {}",
                consumerId, payload.getId());

            latch.countDown();
```

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

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

    // Send test messages
    MessageProducer<TestMessage> producer = queueFactory.createProducer(
        queueName, TestMessage.class);

    for (TestMessage message : testMessages) {
        producer.send(message);
    }

    // Wait for all messages to be processed
    assertTrue("All messages should be processed within timeout",
        latch.await(30, TimeUnit.SECONDS));

    // Verify load balancing - each consumer should process some messages
    Map<String, Long> messageCountPerConsumer = processedMessages.values()
        .stream()
        .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));

    assertEquals("All consumers should participate", consumerCount,
        messageCountPerConsumer.size());

    // Verify no message was processed twice
    assertEquals("All messages should be processed exactly once",
        messageCount, processedMessages.size());

    consumerGroup.stop();
}
```

Intermediate Examples

These examples build on basic concepts and introduce more sophisticated patterns.

4. ConsumerGroupExample

Complexity: Intermediate **Purpose**: Advanced consumer group patterns with sophisticated load balancing and message routing

Detailed Description: This example demonstrates sophisticated consumer group patterns for real-world distributed messaging scenarios. It shows how to implement multiple consumer groups with different filtering strategies and coordination mechanisms.

Key Implementation Details:

- **Multiple Consumer Groups**: Creates three distinct consumer groups (OrderProcessing, PaymentProcessing, Analytics) each with different purposes and filtering strategies
- **Region-Based Filtering**: OrderProcessing group has region-specific consumers (US, EU, ASIA) that only process messages from their designated regions
- **Priority-Based Processing**: PaymentProcessing group implements priority-based consumers with different processing speeds for high-priority vs normal messages
- **Type-Based Routing**: Analytics group routes messages based on message types (PREMIUM, STANDARD) with specialized handlers
- **Dynamic Consumer Management**: Demonstrates adding and removing consumers dynamically based on load

Advanced Patterns Demonstrated:

- **Message Filtering**: Implements MessageFilter.byRegion(), MessageFilter.byPriority(), and MessageFilter.byType() for content-based routing
- **Consumer Coordination**: Shows how consumers coordinate within groups to avoid duplicate processing
- **Load Balancing**: Demonstrates different load balancing strategies across consumer group members
- **Fault Tolerance**: Implements consumer failure detection and automatic failover mechanisms
- **Scaling Patterns**: Shows how to scale consumer groups up and down based on message volume

Real-World Applications:

- E-commerce order processing with regional distribution
- Payment processing with priority handling
- Analytics and reporting with message type specialization
- Multi-tenant systems with tenant-specific processing

Key Code Patterns:

```
// Creating consumer groups with different filtering strategies
private static void createOrderProcessingGroup(QueueFactory factory) throws Exception {
    ConsumerGroup<OrderEvent> orderGroup = factory.createConsumerGroup(
        "OrderProcessing", "order-events", OrderEvent.class);

    // Region-specific consumers with message filtering
    orderGroup.addConsumer("US-Consumer",
        createOrderHandler("US"),
        MessageFilter.byRegion(Set.of("US")));  // Only US orders

    orderGroup.addConsumer("EU-Consumer",
        createOrderHandler("EU"),
        MessageFilter.byRegion(Set.of("EU")));  // Only EU orders

    orderGroup.addConsumer("ASIA-Consumer",
        createOrderHandler("ASIA"),
        MessageFilter.byRegion(Set.of("ASIA"))); // Only ASIA orders

    orderGroup.start();
    logger.info("Order Processing group started with {} consumers",
        orderGroup.getActiveConsumerCount());
}
```

```
// Priority-based consumer with different processing speeds
private static MessageHandler<OrderEvent> createPaymentHandler(String priority) {
    return message -> {
        OrderEvent event = message.getPayload();
        Map<String, String> headers = message.getHeaders();

        logger.info("[PaymentProcessing-{}] Processing payment for order: {} (priority: {})",
            priority, event.getOrderId(), headers.get("priority"));

        // High priority messages process faster
        int processingTime = "HIGH".equals(priority) ? 50 : 200;
        Thread.sleep(processingTime);

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

5. BiTemporalEventStoreExample

Complexity: Intermediate **Purpose**: Event sourcing with bi-temporal data management and temporal queries

Detailed Description: This example demonstrates advanced event sourcing capabilities using bi-temporal data concepts. It shows how to manage events with both valid time (when events actually occurred) and transaction time (when events were recorded in the system).

Key Implementation Details:

- **Bi-Temporal Event Storage**: Implements events with both valid time and transaction time dimensions for complete temporal tracking
- **Event Appending**: Shows how to append events with proper temporal metadata, correlation IDs, and business keys
- **Temporal Queries**: Demonstrates querying events by time ranges, event types, correlation IDs, and business keys
- **Point-in-Time Reconstruction**: Shows how to reconstruct system state at any point in time using temporal queries
- **Event Corrections**: Implements event correction patterns where historical events can be corrected without losing audit trail
- **Event Subscriptions**: Demonstrates real-time event streaming and notifications for live event processing

Advanced Temporal Concepts:

- **Valid Time**: When the event actually occurred in the real world
- **Transaction Time**: When the event was recorded in the system
- **Temporal Queries**: EventQuery.byTimeRange(), EventQuery.byEventType(), EventQuery.byCorrelationId()
- **Historical Reconstruction**: Ability to see system state as it was at any point in time
- **Event Versioning**: Managing multiple versions of events with corrections and updates

Real-World Applications:

- Financial systems requiring complete audit trails
- Regulatory compliance with temporal data requirements
- System state reconstruction for debugging and analysis
- Event-driven architectures with temporal consistency requirements

Key Code Patterns:

```
// Bi-temporal event storage with both valid time and transaction time
BiTemporalEvent<OrderEvent> event1 = eventStore.append(
    "OrderCreated",                    // Event type
    order1,                           // Event payload
    baseTime,                         // Valid time (when it actually happened)
    Map.of("source", "web", "region", "US"),  // Metadata
    "corr-001",                       // Correlation ID
    "ORDER-001"                       // Business key
).join();

// The system automatically records transaction time (when stored)
logger.info("Event stored - Valid Time: {}, Transaction Time: {}",
    event1.getValidTime(), event1.getTransactionTime());



// Temporal queries - reconstruct state at any point in time
private static void demonstrateTemporalQueries(EventStore<OrderEvent> eventStore) {
    Instant queryTime = Instant.now().minus(30, ChronoUnit.MINUTES);

    // Query events as they were known at a specific time
```

```
        List<BiTemporalEvent<OrderEvent>> historicalEvents = eventStore.query(
            EventQuery.asOfTime(queryTime)  // System state as of 30 minutes ago
                .withEventType("OrderCreated")
                .withLimit(10)
        ).join();

        logger.info("Found {} events as of {}", historicalEvents.size(), queryTime);

        // Query events by valid time range (when they actually occurred)
        List<BiTemporalEvent<OrderEvent>> validTimeEvents = eventStore.query(
            EventQuery.byValidTimeRange(
                queryTime.minus(1, ChronoUnit.HOURS),  // From 1 hour before
                queryTime                              // To query time
            )
        ).join();
    }
```

6. RestApiExample

Complexity: Intermediate **Purpose**: Comprehensive REST API implementation for queue and event store operations

Detailed Description: This example demonstrates a complete REST API implementation for PeeGeeQ, showing how to expose queue operations, event store functionality, and system management through HTTP endpoints.

Key Implementation Details:

- **Database Management API**: Endpoints for database setup, schema initialization, and connection management
- **Queue Operations API**: RESTful endpoints for message production, consumption, and queue management
- **Event Store API**: HTTP interface for event storage, querying, and temporal operations
- **Consumer Group Management**: API endpoints for creating, managing, and monitoring consumer groups
- **Health and Metrics API**: Comprehensive health checks and metrics endpoints for operational monitoring
- **WebSocket Integration**: Real-time messaging capabilities through WebSocket connections

API Endpoints Demonstrated:

- **POST /api/v1/queues/{queueName}/messages** - Message production
- **GET /api/v1/queues/{queueName}/messages** - Message consumption
- **POST /api/v1/eventstores/{storeName}/events** - Event storage
- **GET /api/v1/eventstores/{storeName}/events** - Event querying with temporal parameters
- **POST /api/v1/queues/{queueName}/consumer-groups** - Consumer group creation
- **GET /api/v1/health** - System health checks
- **GET /api/v1/metrics** - System metrics and statistics

Advanced Features:

- **Asynchronous Processing**: Non-blocking HTTP request handling with CompletableFuture
- **Error Handling**: Comprehensive HTTP error responses with proper status codes
- **Request Validation**: Input validation and sanitization for all API endpoints
- **Response Formatting**: Consistent JSON response formats with proper HTTP headers
- **Authentication Ready**: Framework for adding authentication and authorization

Real-World Applications:

- Microservices integration with HTTP-based messaging
- Web application backends requiring messaging capabilities

- API gateways and service meshes integration
- Monitoring and management interfaces for operations teams

Key Code Patterns:

```
// REST API endpoints for queue operations
@RestController
@RequestMapping("/api/v1/queues")
public class QueueController {

    @PostMapping("/{queueName}/messages")
    public CompletableFuture<ResponseEntity<MessageResponse>> sendMessage(
            @PathVariable String queueName,
            @RequestBody MessageRequest request) {

        return queueService.sendMessage(queueName, request)
            .thenApply(result -> ResponseEntity.ok(new MessageResponse(result.getMessageId())))
            .exceptionally(ex -> ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                .body(new MessageResponse("Error: " + ex.getMessage())));
    }

    @GetMapping("/{queueName}/messages")
    public CompletableFuture<ResponseEntity<List<MessageResponse>>> receiveMessages(
            @PathVariable String queueName,
            @RequestParam(defaultValue = "10") int batchSize,
            @RequestParam(defaultValue = "5000") long maxWaitTime) {

        return queueService.receiveMessages(queueName, batchSize, maxWaitTime)
            .thenApply(messages -> ResponseEntity.ok(
                messages.stream()
                    .map(msg -> new MessageResponse(msg.getMessageId(), msg.getPayload()))
                    .collect(Collectors.toList())))
            .exceptionally(ex -> ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                .body(Collections.emptyList()));
    }
}
```

```
// Health check endpoints with comprehensive monitoring
@RestController
@RequestMapping("/api/v1/health")
public class HealthController {

    @GetMapping
    public ResponseEntity<HealthResponse> getOverallHealth() {
        HealthStatus overallHealth = healthService.getOverallHealth();

        HealthResponse response = new HealthResponse(
            overallHealth.getStatus().toString(),
            overallHealth.getMessage(),
            Instant.now()
        );

        HttpStatus httpStatus = overallHealth.getStatus() == HealthStatus.Status.HEALTHY
            ? HttpStatus.OK : HttpStatus.SERVICE_UNAVAILABLE;

        return ResponseEntity.status(httpStatus).body(response);
    }

    @GetMapping("/components")
    public ResponseEntity<Map<String, ComponentHealth>> getComponentHealth() {
        Map<String, HealthStatus> componentHealth = healthService.getComponentHealth();
```

```
            Map<String, ComponentHealth> response = componentHealth.entrySet().stream()
                .collect(Collectors.toMap(
                    Map.Entry::getKey,
                    entry -> new ComponentHealth(
                        entry.getValue().getStatus().toString(),
                        entry.getValue().getMessage(),
                        entry.getValue().getLastChecked()
                    )
                ));

            return ResponseEntity.ok(response);
        }
    }
```

7. MultiConfigurationExample

Complexity: Intermediate **Purpose**: Multi-environment configuration management and deployment strategies

Detailed Description: This example demonstrates sophisticated configuration management patterns for deploying PeeGeeQ across multiple environments (development, staging, production) with environment-specific optimizations and settings.

Key Implementation Details:

- **Environment-Specific Profiles**: Implements separate configuration profiles for development, staging, and production environments
- **Configuration Hierarchy**: Demonstrates how configurations inherit from base settings and override specific values
- **External Configuration Sources**: Shows integration with external configuration systems (environment variables, config files, config servers)
- **Dynamic Configuration**: Implements runtime configuration updates without application restarts
- **Configuration Validation**: Comprehensive validation of configuration values with helpful error messages
- **Environment Detection**: Automatic environment detection and appropriate configuration loading

Configuration Patterns Demonstrated:

- **Base Configuration**: Common settings shared across all environments
- **Environment Overrides**: Environment-specific database connections, performance tuning, security settings
- **Feature Flags**: Environment-based feature enablement and configuration
- **Resource Scaling**: Environment-appropriate connection pool sizes, thread counts, and memory settings
- **Security Configuration**: Environment-specific security policies, encryption settings, and access controls

Advanced Features:

- **Configuration Encryption**: Sensitive configuration values encrypted at rest
- **Configuration Auditing**: Tracking configuration changes and their impact
- **Hot Reloading**: Dynamic configuration updates without service interruption
- **Configuration Templates**: Template-based configuration generation for different environments

Real-World Applications:

- Enterprise deployment pipelines with multiple environments
- Cloud-native applications with environment-specific scaling
- Compliance requirements with environment-specific security settings
- DevOps automation with configuration as code

Key Code Patterns:

```java
// Multi-environment configuration management
public class MultiEnvironmentConfigManager {

    private final Map<String, PeeGeeQConfiguration> environmentConfigs = new HashMap<>();

    public void initializeEnvironments() {
        // Development environment - optimized for development workflow
        PeeGeeQConfiguration devConfig = new PeeGeeQConfiguration("development")
            .withDatabaseConfig(DatabaseConfig.builder()
                .host("localhost")
                .port(5432)
                .database("peegeeq_dev")
                .maxConnections(10)  // Smaller pool for dev
                .connectionTimeout(Duration.ofSeconds(5))
                .build())
            .withQueueConfig(QueueConfig.builder()
                .defaultBatchSize(5)  // Smaller batches for testing
                .maxRetryAttempts(2)  // Fewer retries in dev
                .enableMetrics(true)
                .build());

        // Production environment - optimized for performance and reliability
        PeeGeeQConfiguration prodConfig = new PeeGeeQConfiguration("production")
            .withDatabaseConfig(DatabaseConfig.builder()
                .host(System.getenv("PROD_DB_HOST"))
                .port(Integer.parseInt(System.getenv("PROD_DB_PORT")))
                .database(System.getenv("PROD_DB_NAME"))
                .maxConnections(50)  // Larger pool for production
                .connectionTimeout(Duration.ofSeconds(30))
                .enableSSL(true)     // SSL required in production
                .build())
            .withQueueConfig(QueueConfig.builder()
                .defaultBatchSize(100)  // Larger batches for efficiency
                .maxRetryAttempts(5)    // More retries for reliability
                .enableMetrics(true)
                .enableHealthChecks(true)
                .build());

        environmentConfigs.put("development", devConfig);
        environmentConfigs.put("production", prodConfig);
    }
}



// Environment-specific feature flags and optimizations
private static void demonstrateEnvironmentSpecificFeatures(String environment) {
    PeeGeeQConfiguration config = getConfigurationForEnvironment(environment);

    switch (environment) {
        case "development":
            // Development-specific features
            config.enableDebugLogging(true);
            config.setLogLevel(LogLevel.DEBUG);
            config.enableTestingFeatures(true);
            config.setDatabasePoolSize(5);  // Small pool for dev
            break;

        case "staging":
            // Staging-specific features
            config.enableDebugLogging(false);
            config.setLogLevel(LogLevel.INFO);
            config.enablePerformanceMonitoring(true);
            config.setDatabasePoolSize(20);  // Medium pool for staging
            break;
```

```
            case "production":
                // Production-specific features
                config.enableDebugLogging(false);
                config.setLogLevel(LogLevel.WARN);
                config.enablePerformanceMonitoring(true);
                config.enableSecurityFeatures(true);
                config.setDatabasePoolSize(50);  // Large pool for production
                config.enableCircuitBreaker(true);
                config.enableDeadLetterQueue(true);
                break;
        }

        logger.info("Configured PeeGeeQ for {} environment with {} features",
            environment, config.getEnabledFeatures().size());
    }
```

8. NativeVsOutboxComparisonExample

Complexity: Intermediate **Purpose**: Comprehensive performance comparison and architectural decision guidance

Detailed Description: This example provides detailed performance benchmarking and comparison between PeeGeeQ's native queue implementation and outbox pattern implementation, helping architects make informed decisions about which approach to use.

Key Implementation Details:

- **Performance Benchmarking**: Comprehensive throughput and latency measurements for both implementations
- **Load Testing**: Stress testing both patterns under various load conditions and message volumes
- **Resource Usage Analysis**: Memory, CPU, and database connection usage comparison
- **Scalability Testing**: How each pattern performs as load increases and with multiple consumers
- **Consistency Analysis**: Comparison of consistency guarantees and trade-offs between patterns
- **Use Case Scenarios**: Real-world scenarios where each pattern excels

Benchmarking Methodology:

- **Throughput Testing**: Messages per second under sustained load
- **Latency Measurement**: End-to-end message processing latency
- **Resource Monitoring**: CPU, memory, and database connection usage
- **Concurrent Consumer Testing**: Performance with multiple concurrent consumers
- **Failure Scenario Testing**: Behavior during database failures and recovery

Performance Characteristics Analyzed:

- **Native Queue**: Higher throughput, lower latency, PostgreSQL LISTEN/NOTIFY real-time processing
- **Outbox Pattern**: Better consistency guarantees, transactional safety, easier debugging, automatic stuck message recovery
- **Memory Usage**: Comparison of memory footprint and garbage collection impact
- **Database Load**: Impact on database performance and connection usage

Decision Framework Provided:

- **When to Use Native**: High-throughput scenarios, real-time processing requirements, simple consistency needs
- **When to Use Outbox**: Transactional consistency requirements, complex business logic, audit trail needs
- **Hybrid Approaches**: Using both patterns in the same application for different use cases

Real-World Applications:

- Architecture decision support for new projects
- Performance optimization for existing systems
- Capacity planning and resource allocation
- Technology evaluation and selection processes

Key Code Patterns:

```
// Performance comparison framework
public class NativeVsOutboxComparison {

    public ComparisonResults runPerformanceComparison(int messageCount, int consumerCount) {
        logger.info("Starting performance comparison: {} messages, {} consumers",
            messageCount, consumerCount);

        // Test Native Queue performance
        PerformanceMetrics nativeMetrics = measureNativeQueuePerformance(
            messageCount, consumerCount);

        // Test Outbox Pattern performance
        PerformanceMetrics outboxMetrics = measureOutboxPatternPerformance(
            messageCount, consumerCount);

        // Analyze and compare results
        ComparisonResults results = new ComparisonResults(nativeMetrics, outboxMetrics);

        logger.info("Native Queue: {} msg/sec, {}ms avg latency",
            nativeMetrics.getThroughput(), nativeMetrics.getAverageLatency());
        logger.info("Outbox Pattern: {} msg/sec, {}ms avg latency",
            outboxMetrics.getThroughput(), outboxMetrics.getAverageLatency());

        return results;
    }

    private PerformanceMetrics measureNativeQueuePerformance(int messageCount, int consumerCount) {
        Instant startTime = Instant.now();
        CountDownLatch latch = new CountDownLatch(messageCount);

        // Create native queue with LISTEN/NOTIFY
        NativeQueue<TestMessage> nativeQueue = queueFactory.createNativeQueue(
            "native-perf-test", TestMessage.class);

        // Start consumers
        List<MessageConsumer<TestMessage>> consumers = createConsumers(
            nativeQueue, consumerCount, latch);

        // Send messages and measure
        sendTestMessages(nativeQueue, messageCount);

        try {
            latch.await(60, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        Duration totalTime = Duration.between(startTime, Instant.now());
        return new PerformanceMetrics(messageCount, totalTime, "Native");
    }
}
```

```
// Decision framework implementation
public class ArchitecturalDecisionFramework {

    public PatternRecommendation recommendPattern(UseCase useCase) {
        ScoreCard nativeScore = evaluateNativeQueue(useCase);
        ScoreCard outboxScore = evaluateOutboxPattern(useCase);

        if (nativeScore.getTotalScore() > outboxScore.getTotalScore()) {
            return new PatternRecommendation(
                PatternType.NATIVE_QUEUE,
                nativeScore,
                "Recommended for high-throughput, real-time processing scenarios"
            );
        } else {
            return new PatternRecommendation(
                PatternType.OUTBOX_PATTERN,
                outboxScore,
                "Recommended for transactional consistency and complex business logic"
            );
        }
    }

    private ScoreCard evaluateNativeQueue(UseCase useCase) {
        ScoreCard score = new ScoreCard("Native Queue");

        // Performance factors
        score.addScore("Throughput", useCase.requiresHighThroughput() ? 10 : 5);
        score.addScore("Latency", useCase.requiresLowLatency() ? 10 : 5);
        score.addScore("Real-time", useCase.requiresRealTime() ? 10 : 3);

        // Consistency factors
        score.addScore("ACID Compliance", useCase.requiresACID() ? 3 : 8);
        score.addScore("Transactional Safety", useCase.requiresTransactions() ? 4 : 8);

        // Operational factors
        score.addScore("Simplicity", 8);  // Native is simpler to understand
        score.addScore("Debugging", 6);   // Harder to debug real-time processing

        return score;
    }
}
```

Advanced Examples

These examples demonstrate sophisticated patterns for production systems.

9. MessagePriorityExample

Complexity: Advanced **Purpose**: Comprehensive priority-based message processing with real-world business scenarios

Detailed Description: This example demonstrates sophisticated priority-based message processing patterns, showing how to implement and optimize priority queues for real-world business scenarios with different urgency levels and processing requirements.

Key Implementation Details:

- **Five Priority Levels**: CRITICAL (10), HIGH (7-9), NORMAL (4-6), LOW (1-3), BULK (0) with clear business semantics
- **Priority Queue Configuration**: Optimized queue configuration for priority-based message ordering and processing

- **Real-World Scenario Modeling**: Three comprehensive business scenarios demonstrating practical priority usage
- **Performance Optimization**: Priority queue performance tuning and optimization techniques
- **Priority-Aware Consumers**: Consumers that adjust processing behavior based on message priority
- **Priority Metrics**: Comprehensive metrics and monitoring for priority-based processing

Business Scenarios Demonstrated:

E-Commerce Order Processing Scenario:

- **VIP Customer Orders**: CRITICAL priority for premium customers with expedited processing
- **High-Value Orders**: HIGH priority for orders above certain value thresholds
- **Standard Orders**: NORMAL priority for regular customer orders
- **Bulk Orders**: LOW priority for wholesale and bulk processing
- **Analytics Processing**: BULK priority for order analytics and reporting

Financial Transaction Processing Scenario:

- **Fraud Alerts**: CRITICAL priority for immediate fraud detection and prevention
- **Regulatory Transactions**: HIGH priority for compliance-required transactions
- **Customer Transactions**: NORMAL priority for standard customer-initiated transactions
- **Internal Transfers**: LOW priority for internal account movements
- **Batch Processing**: BULK priority for end-of-day batch operations

System Monitoring and Alerting Scenario:

- **Security Incidents**: CRITICAL priority for security breaches and threats
- **System Failures**: HIGH priority for service outages and critical errors
- **Performance Alerts**: NORMAL priority for performance degradation warnings
- **Maintenance Notifications**: LOW priority for scheduled maintenance alerts
- **Log Processing**: BULK priority for log aggregation and analysis

Advanced Priority Features:

- **Dynamic Priority Assignment**: Runtime priority calculation based on business rules
- **Priority Escalation**: Automatic priority escalation for aging messages
- **Priority-Based SLA**: Different service level agreements based on message priority
- **Priority Metrics**: Detailed metrics on processing times by priority level
- **Priority Queue Optimization**: Database and memory optimization for priority processing

Performance Characteristics Analyzed:

- **Throughput Impact**: How priority queues affect overall message throughput
- **Latency Distribution**: Processing latency analysis by priority level
- **Resource Usage**: Memory and CPU usage patterns with priority processing
- **Scalability**: How priority queues scale with increased message volume

Real-World Applications:

- Customer service systems with priority-based ticket routing
- Financial systems with regulatory priority requirements
- E-commerce platforms with customer tier-based processing
- Monitoring systems with alert severity levels
- Healthcare systems with patient priority classifications

Key Code Patterns:

```
// Priority levels with clear business semantics
private static final int PRIORITY_CRITICAL = 10;  // System alerts, security events
private static final int PRIORITY_HIGH = 8;       // Important business events
private static final int PRIORITY_NORMAL = 5;     // Regular operations
private static final int PRIORITY_LOW = 2;        // Background tasks
private static final int PRIORITY_BULK = 0;       // Batch processing

// Sending priority messages with metadata
private static void sendPriorityMessage(MessageProducer<PriorityMessage> producer,
                                        String messageId, String messageType,
                                        String content, int priority) throws Exception {
    PriorityMessage message = new PriorityMessage(
        messageId, messageType, content, priority, Instant.now(), Map.of());

    // Priority is sent in headers for queue processing
    Map<String, String> headers = new HashMap<>();
    headers.put("priority", String.valueOf(priority));
    headers.put("messageType", messageType);

    producer.send(message, headers).get(5, TimeUnit.SECONDS);
}
```

```
// E-commerce scenario with customer-tier based priorities
private static void demonstrateECommerceScenario(QueueFactory factory) throws Exception {
    MessageProducer<PriorityMessage> producer = factory.createProducer("ecommerce-orders", PriorityMessage.class);

    // VIP customer order - CRITICAL priority
    sendPriorityMessageWithMetadata(producer, "vip-001", "VIP_ORDER",
        "VIP customer premium order", PRIORITY_CRITICAL,
        Map.of("customerType", "VIP", "orderValue", "2500.00"));

    // High-value order - HIGH priority
    sendPriorityMessageWithMetadata(producer, "high-001", "HIGH_VALUE_ORDER",
        "High-value customer order", PRIORITY_HIGH,
        Map.of("customerType", "premium", "orderValue", "850.00"));

    // Regular order - NORMAL priority
    sendPriorityMessageWithMetadata(producer, "reg-001", "REGULAR_ORDER",
        "Standard customer order", PRIORITY_NORMAL,
        Map.of("customerType", "regular", "orderValue", "45.99"));
}
```

10. EnhancedErrorHandlingExample

Complexity: Advanced **Purpose**: Comprehensive error handling patterns for production-grade resilience and fault tolerance

Detailed Description: This example demonstrates sophisticated error handling patterns essential for production distributed systems. It shows how to implement resilient message processing with comprehensive error classification, recovery strategies, and monitoring.

Key Implementation Details:

- **Five Error Handling Strategies**: RETRY, CIRCUIT_BREAKER, DEAD_LETTER, IGNORE, and ALERT with specific use cases for each

- **Exponential Backoff Retry**: Sophisticated retry logic with exponential backoff (1s, 2s, 4s) and jitter to prevent thundering herd
- **Circuit Breaker Integration**: Circuit breaker pattern implementation for preventing cascade failures in distributed systems
- **Dead Letter Queue Management**: Comprehensive DLQ handling with message recovery, statistics tracking, and manual intervention capabilities
- **Poison Message Detection**: Automatic detection and quarantine of messages that consistently fail processing
- **Error Classification System**: Intelligent error categorization to determine appropriate handling strategy

Error Handling Strategies Detailed:

RETRY Strategy:

- **Transient Errors**: Network timeouts, database connection failures, service unavailable errors
- **Exponential Backoff**: 1s, 2s, 4s, 8s with configurable maximum retry attempts
- **Jitter Implementation**: Random delay addition to prevent synchronized retry storms
- **Retry Metrics**: Tracking retry attempts, success rates, and backoff effectiveness

CIRCUIT_BREAKER Strategy:

- **Failure Threshold**: Configurable failure rate threshold for circuit opening
- **Recovery Testing**: Periodic testing of failed services for automatic recovery
- **Fallback Mechanisms**: Alternative processing paths when circuit is open
- **Circuit State Monitoring**: Real-time monitoring of circuit breaker states

DEAD_LETTER Strategy:

- **Permanent Failures**: Parse errors, validation failures, business rule violations
- **DLQ Management**: Organized storage of failed messages with metadata
- **Recovery Workflows**: Manual and automated recovery processes
- **DLQ Analytics**: Analysis of failure patterns and root cause identification

IGNORE Strategy:

- **Non-Critical Errors**: Logging and metrics errors that don't affect business operations
- **Graceful Degradation**: Continuing operation despite non-critical failures
- **Error Logging**: Comprehensive logging for debugging and analysis

ALERT Strategy:

- **Critical System Events**: Security incidents, data corruption, system failures
- **Notification Systems**: Integration with alerting systems (email, SMS, Slack)
- **Escalation Procedures**: Automatic escalation based on error severity and duration

Advanced Error Handling Features:

- **Error Context Preservation**: Maintaining full error context for debugging
- **Error Correlation**: Linking related errors across distributed system components
- **Error Rate Limiting**: Preventing error processing from overwhelming the system
- **Error Recovery Automation**: Automated recovery procedures for common error scenarios

Monitoring and Observability:

- **Error Metrics**: Comprehensive error rate, type, and recovery metrics

- **Error Dashboards**: Real-time visualization of error patterns and trends
- **Error Alerting**: Proactive alerting based on error thresholds and patterns
- **Error Reporting**: Detailed error reports for operations and development teams

Real-World Applications:

- Financial systems requiring high reliability and error recovery
- E-commerce platforms with complex error scenarios
- Healthcare systems with critical error handling requirements
- IoT systems with network reliability challenges
- Microservices architectures with distributed error propagation

Key Code Patterns:

```
// Error handling strategies enumeration
enum ErrorHandlingStrategy {
    RETRY,           // Exponential backoff retry
    CIRCUIT_BREAKER, // Circuit breaker pattern
    DEAD_LETTER,     // Move to dead letter queue
    IGNORE,          // Log and continue
    ALERT            // Send alert and continue
}

// Intelligent error classification
private static ErrorHandlingStrategy classifyError(ProcessingException error) {
    String errorType = error.getErrorType();

    switch (errorType) {
        case "NETWORK_TIMEOUT":
        case "DATABASE_ERROR":
        case "SERVICE_UNAVAILABLE":
            return ErrorHandlingStrategy.RETRY;        // Transient errors

        case "BUSINESS_RULE_ERROR":
        case "VALIDATION_ERROR":
            return ErrorHandlingStrategy.ALERT;        // Business logic issues

        case "PARSE_ERROR":
        case "PERMANENT_ERROR":
            return ErrorHandlingStrategy.DEAD_LETTER;  // Permanent failures

        default:
            return error.isRetryable() ? ErrorHandlingStrategy.RETRY : ErrorHandlingStrategy.IGNORE;
    }
}
```

```
// Exponential backoff retry implementation
private static void handleRetryStrategy(ErrorTestMessage payload, ProcessingException e, int attempt) {
    if (e.isRetryable() && attempt < 2) { // Max 3 attempts
        // Exponential backoff: 1s, 2s, 4s
        long backoffMs = (long) Math.pow(2, attempt) * 1000;
        logger.info("Scheduling retry for message: {} in {}ms", payload.getMessageId(), backoffMs);

        // Schedule retry with exponential backoff
        CompletableFuture.delayedExecutor(backoffMs, TimeUnit.MILLISECONDS).execute(() -> {
            try {
                ErrorTestMessage retryMessage = payload.withIncrementedAttempts();
                producer.send(retryMessage, Map.of("retry", "true", "attempt", String.valueOf(attempt + 1)));
            } catch (Exception ex) {
```

```
                logger.error("Failed to schedule retry", ex);
            }
        });
    } else {
        logger.warn("Max retry attempts reached for message: {}", payload.getMessageId());
        // Move to dead letter queue or alert
    }
}
```

11. FilterErrorHandlingExample

Complexity: Advanced **Purpose**: Enterprise-grade filter error handling with circuit breakers, async retries, and dead letter queues

Detailed Description: This example demonstrates the sophisticated filter error handling system that provides enterprise-grade reliability with intelligent error classification, circuit breaker protection, async retry mechanisms, and dead letter queue integration.

Key Features Demonstrated:

Error Classification System:

- **Transient Error Detection**: Automatic identification of temporary failures that may succeed on retry
- **Permanent Error Detection**: Recognition of persistent failures that won't succeed on retry
- **Pattern-Based Classification**: Configurable error patterns for intelligent classification
- **Exception-Type Classification**: Classification based on exception types

Recovery Strategies:

- **Immediate Rejection**: Fast rejection for permanent errors or high-performance scenarios
- **Retry with Exponential Backoff**: Intelligent retry with increasing delays
- **Dead Letter Queue Integration**: Comprehensive DLQ support with metadata enrichment
- **Circuit Breaker Protection**: Prevents cascading failures during outages

Advanced Features:

- **Async Retry Operations**: Non-blocking retry operations that don't impact throughput
- **Configurable Strategies**: Different strategies per consumer group and error type
- **Comprehensive Monitoring**: Rich metrics for production observability
- **Performance Optimization**: Fast-fail behavior and resource management

Configuration Patterns:

- **High-Reliability Configuration**: For critical business processes
- **High-Performance Configuration**: For high-volume scenarios
- **Balanced Configuration**: For general business applications

Real-World Applications:

- Financial transaction processing with strict reliability requirements
- E-commerce order processing with complex validation rules
- Healthcare systems with critical error handling needs
- IoT systems with network reliability challenges
- Microservices architectures with distributed error propagation

Key Code Patterns:

```
// High-reliability configuration for critical systems
FilterErrorHandlingConfig criticalConfig = 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
FilterErrorHandlingConfig performanceConfig = FilterErrorHandlingConfig.builder()
    .addPermanentErrorPattern("invalid")
    .defaultStrategy(FilterErrorStrategy.REJECT_IMMEDIATELY)
    .maxRetries(1)
    .circuitBreakerEnabled(true)
    .circuitBreakerFailureThreshold(3)
    .circuitBreakerTimeout(Duration.ofSeconds(30))
    .deadLetterQueueEnabled(false)
    .build();

// Consumer with sophisticated error handling
OutboxConsumerGroupMember<PaymentEvent> consumer = new OutboxConsumerGroupMember<>(
    "payment-processor",
    "payment-group",
    "payments",
    paymentHandler,
    paymentValidationFilter,
    null,
    criticalConfig  // Apply enterprise-grade error handling
);

// Monitor error handling metrics
FilterCircuitBreaker.CircuitBreakerMetrics cbMetrics = consumer.getFilterCircuitBreakerMetrics();
System.out.println("Circuit Breaker State: " + cbMetrics.getState());
System.out.println("Failure Rate: " + String.format("%.2f%%", cbMetrics.getFailureRate() * 100));
```

Performance Characteristics:

Scenario	Throughput (msg/sec)	Latency Impact	Resource Usage
Normal Operation	1000	Minimal	Low
Filter Exceptions (20%)	500	Low	Medium

Scenario	Throughput (msg/sec)	Latency Impact	Resource Usage
Circuit Breaker Open	2000	Minimal	Very Low
Async Retries Active	300	Medium	Medium
DLQ Operations	100	High	High

Monitoring and Observability:

- **Circuit Breaker Metrics**: State, failure rates, recovery times
- **Retry Metrics**: Attempt counts, success rates, backoff effectiveness
- **Dead Letter Queue Metrics**: Message counts, success rates, processing times
- **Error Classification Metrics**: Distribution of error types and handling strategies

12. PerformanceTuningExample

Complexity: Advanced **Purpose**: Comprehensive performance optimization techniques for high-throughput messaging systems

Detailed Description: This example demonstrates advanced performance optimization techniques for achieving high-throughput, low-latency messaging with PeeGeeQ. It provides systematic approaches to performance tuning and optimization for production workloads.

Key Implementation Details:

- **Optimized PostgreSQL Container**: Custom PostgreSQL configuration with performance optimizations (shared memory, work memory, connection limits)
- **Connection Pool Optimization**: Advanced connection pool tuning with stress testing and optimal sizing determination
- **Throughput Benchmarking**: Systematic measurement of message throughput capabilities (targeting 10,000+ msg/sec)
- **Latency Optimization**: Comprehensive latency analysis and optimization techniques
- **Batch Processing Optimization**: Optimal batch size determination for different workload patterns
- **Concurrent Processing Optimization**: Thread pool sizing and coordination for maximum throughput
- **Memory Usage Optimization**: Memory profiling and optimization techniques

Performance Optimization Areas:

Database Connection Pool Optimization:

- **Pool Size Tuning**: Systematic testing of different pool sizes under various load conditions
- **Connection Lifecycle Management**: Optimizing connection creation, validation, and cleanup
- **Connection Pool Monitoring**: Real-time monitoring of pool utilization and performance
- **Stress Testing**: Connection pool behavior under extreme load conditions

Throughput Optimization Strategies:

- **Native vs Outbox Comparison**: Performance comparison between implementation patterns
- **Message Serialization Optimization**: Efficient message serialization and deserialization
- **Database Round Trip Minimization**: Reducing database interactions for higher throughput
- **Consumer Thread Pool Tuning**: Optimal thread pool configuration for message processing

Latency Optimization Techniques:

- **Message Size Impact Analysis**: How message size affects processing latency

- **Processing Pipeline Optimization**: Streamlining message processing pipelines
- **Network Latency Minimization**: Reducing network overhead in message processing
- **JVM Optimization**: JVM tuning for low-latency message processing

Batch Processing Optimization:

- **Optimal Batch Size Determination**: Systematic testing to find optimal batch sizes
- **Batch Processing Patterns**: Different batching strategies for various workload types
- **Batch Commit Optimization**: Optimizing database commits for batch operations
- **Batch Error Handling**: Efficient error handling in batch processing scenarios

Concurrent Processing Optimization:

- **Thread Scaling Analysis**: Determining optimal thread counts for different scenarios
- **Thread Pool Configuration**: Advanced thread pool tuning and monitoring
- **Concurrent Consumer Coordination**: Optimizing coordination between concurrent consumers
- **Lock Contention Minimization**: Reducing lock contention in high-concurrency scenarios

Memory Usage Optimization:

- **Memory Profiling**: Systematic memory usage analysis and profiling
- **Garbage Collection Tuning**: JVM garbage collection optimization for messaging workloads
- **Memory Leak Detection**: Identifying and preventing memory leaks in long-running processes
- **Memory-Efficient Data Structures**: Using optimal data structures for memory efficiency

Performance Monitoring and Metrics:

- **Real-Time Performance Metrics**: Comprehensive performance monitoring and alerting
- **Performance Dashboards**: Visual representation of system performance characteristics
- **Performance Regression Detection**: Automated detection of performance regressions
- **Capacity Planning**: Using performance data for capacity planning and scaling decisions

Real-World Applications:

- High-frequency trading systems requiring ultra-low latency
- E-commerce platforms with peak traffic handling requirements
- IoT systems processing millions of sensor messages
- Financial systems with high-throughput transaction processing
- Real-time analytics systems with streaming data processing

Key Code Patterns:

```
// Optimized PostgreSQL container for performance testing
private static PostgreSQLContainer<?> createOptimizedPostgreSQLContainer() {
    return new PostgreSQLContainer<>("postgres:15.13-alpine3.20")
        .withDatabaseName("peegeeq_performance")
        .withUsername("peegeeq_perf")
        .withPassword("peegeeq_perf")
        .withSharedMemorySize(512 * 1024 * 1024L)  // 512MB shared memory
        .withCommand(
            "postgres",
            "-c", "max_connections=200",           // Increased connections
            "-c", "shared_buffers=256MB",          // Larger buffer pool
            "-c", "effective_cache_size=1GB",      // Cache size hint
            "-c", "work_mem=16MB",                 // Work memory per operation
```

```
            "-c", "maintenance_work_mem=64MB",     // Maintenance operations
            "-c", "checkpoint_completion_target=0.9",
            "-c", "wal_buffers=16MB",
            "-c", "default_statistics_target=100"
        );
}



// Throughput measurement with performance metrics
private static PerformanceMetrics measureThroughput(QueueFactory factory, String queueName,
                                                    int messageCount, int consumerCount) throws Exception {
    Instant startTime = Instant.now();
    CountDownLatch latch = new CountDownLatch(messageCount);
    AtomicLong processedCount = new AtomicLong(0);

    // Create multiple consumers for parallel processing
    List<MessageConsumer<PerformanceTestMessage>> consumers = new ArrayList<>();
    for (int i = 0; i < consumerCount; i++) {
        MessageConsumer<PerformanceTestMessage> consumer =
            factory.createConsumer(queueName, PerformanceTestMessage.class);

        consumer.subscribe(message -> {
            processedCount.incrementAndGet();
            latch.countDown();
            return CompletableFuture.completedFuture(null);
        });

        consumers.add(consumer);
    }

    // Send messages and measure throughput
    MessageProducer<PerformanceTestMessage> producer =
        factory.createProducer(queueName, PerformanceTestMessage.class);

    for (int i = 0; i < messageCount; i++) {
        PerformanceTestMessage message = new PerformanceTestMessage("msg-" + i, "test-data");
        producer.send(message);
    }

    latch.await(60, TimeUnit.SECONDS);
    Duration totalTime = Duration.between(startTime, Instant.now());

    return new PerformanceMetrics(messageCount, totalTime, processedCount.get());
}
```

12. ServiceDiscoveryExample

Complexity: Advanced **Purpose**: Multi-instance deployment coordination and service discovery patterns

Detailed Description: This example demonstrates sophisticated service discovery and multi-instance coordination patterns for distributed PeeGeeQ deployments. It shows how to implement service registration, health monitoring, and federated management across multiple instances.

Key Implementation Details:

- **Service Manager Integration**: Complete service manager setup with automatic instance registration and management
- **Multi-Instance Coordination**: Coordination mechanisms for multiple PeeGeeQ instances across different environments
- **Health Monitoring System**: Comprehensive health checking and status monitoring for distributed instances
- **Federated Management**: Federation patterns for connecting multiple PeeGeeQ clusters and regions
- **Load Balancing**: Intelligent load distribution across multiple service instances

- **Environment-Based Filtering**: Instance filtering and routing based on environment and region

Service Discovery Features:

Instance Registration and Management:

- **Automatic Registration**: Automatic service instance registration with metadata
- **Health Check Integration**: Continuous health monitoring with automatic deregistration of unhealthy instances
- **Instance Metadata**: Rich metadata including version, environment, region, and capabilities
- **Dynamic Instance Management**: Runtime addition and removal of service instances

Multi-Environment Support:

- **Environment Isolation**: Separate instance groups for development, staging, and production
- **Region-Based Deployment**: Geographic distribution with region-aware routing
- **Cross-Environment Communication**: Controlled communication patterns between environments
- **Environment-Specific Configuration**: Environment-based configuration and policy management

Federation and Clustering:

- **Cluster Formation**: Automatic cluster formation and membership management
- **Cross-Cluster Communication**: Secure communication between different clusters
- **Cluster Health Monitoring**: Cluster-wide health monitoring and alerting
- **Cluster Load Balancing**: Load distribution across cluster members

Advanced Service Discovery Patterns:

- **Service Mesh Integration**: Integration with service mesh technologies
- **Circuit Breaker Integration**: Circuit breaker patterns for service-to-service communication
- **Service Versioning**: Version-aware service discovery and routing
- **Canary Deployment Support**: Service discovery support for canary and blue-green deployments

Monitoring and Observability:

- **Service Topology Visualization**: Real-time visualization of service topology and relationships
- **Service Performance Metrics**: Performance monitoring across distributed instances
- **Service Dependency Tracking**: Tracking and monitoring service dependencies
- **Service Health Dashboards**: Comprehensive dashboards for service health and performance

Real-World Applications:

- Microservices architectures with dynamic service discovery
- Multi-region deployments with geographic distribution
- Cloud-native applications with auto-scaling requirements
- Enterprise systems with complex deployment topologies
- DevOps environments with continuous deployment pipelines

Key Code Patterns:

```
// Service registration and discovery with health monitoring
public class PeeGeeQServiceManager {

    private final ServiceRegistry serviceRegistry;
```

```
    private final HealthMonitor healthMonitor;

    public void registerService(String serviceName, String environment, String region) {
        ServiceInstance instance = ServiceInstance.builder()
            .serviceName(serviceName)
            .instanceId(generateInstanceId())
            .host(getLocalHostname())
            .port(getServicePort())
            .environment(environment)
            .region(region)
            .metadata(Map.of(
                "version", getServiceVersion(),
                "capabilities", String.join(",", getServiceCapabilities()),
                "startTime", Instant.now().toString()
            ))
            .healthCheckUrl("/health")
            .build();

        // Register with service discovery
        serviceRegistry.register(instance);

        // Start health monitoring
        healthMonitor.startMonitoring(instance, Duration.ofSeconds(30));

        logger.info("Registered service instance: {} in {}/{}",
            instance.getInstanceId(), environment, region);
    }

    public List<ServiceInstance> discoverServices(String serviceName, String environment) {
        // Discover healthy instances only
        return serviceRegistry.discover(serviceName)
            .stream()
            .filter(instance -> environment.equals(instance.getEnvironment()))
            .filter(instance -> healthMonitor.isHealthy(instance.getInstanceId()))
            .collect(Collectors.toList());
    }
}



// Multi-instance coordination and load balancing
public class MultiInstanceCoordinator {

    private final ServiceManager serviceManager;
    private final LoadBalancer loadBalancer;

    public void coordinateMessageProcessing(String queueName, String environment) {
        // Discover available PeeGeeQ instances
        List<ServiceInstance> instances = serviceManager.discoverServices(
            "peegeeq-service", environment);

        if (instances.isEmpty()) {
            throw new ServiceUnavailableException("No PeeGeeQ instances available");
        }

        // Create consumer group across multiple instances
        ConsumerGroupConfig config = ConsumerGroupConfig.builder()
            .groupName("distributed-processors")
            .queueName(queueName)
            .instances(instances)
            .loadBalancingStrategy(LoadBalancingStrategy.ROUND_ROBIN)
            .failoverEnabled(true)
            .healthCheckInterval(Duration.ofSeconds(15))
            .build();

        DistributedConsumerGroup consumerGroup = new DistributedConsumerGroup(config);
```

```
        // Start coordinated processing
        consumerGroup.start();

        // Monitor instance health and rebalance as needed
        scheduleHealthChecksAndRebalancing(consumerGroup);

        logger.info("Started distributed processing across {} instances",
            instances.size());
    }

    private void scheduleHealthChecksAndRebalancing(DistributedConsumerGroup group) {
        ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);

        scheduler.scheduleAtFixedRate(() -> {
            try {
                group.checkInstanceHealth();
                group.rebalanceIfNeeded();
            } catch (Exception e) {
                logger.error("Error during health check and rebalancing", e);
            }
        }, 15, 15, TimeUnit.SECONDS);
    }
}
```

Expert Examples

These examples demonstrate the most sophisticated patterns for complex production systems.

13. IntegrationPatternsExample

Complexity: Expert **Purpose**: Comprehensive implementation of enterprise integration patterns for complex distributed systems

Detailed Description: This example demonstrates sophisticated enterprise integration patterns essential for complex distributed systems and microservices architectures. It implements industry-standard messaging patterns with advanced correlation, routing, and orchestration capabilities.

Key Implementation Details:

- **Request-Reply Pattern**: Synchronous communication with correlation ID management, timeout handling, and response correlation
- **Publish-Subscribe Pattern**: Event broadcasting to multiple subscribers with topic-based routing and subscriber management
- **Message Router Pattern**: Conditional message routing based on content, headers, and business rules
- **Content-Based Router**: Advanced routing decisions based on message content analysis
- **Message Translator**: Message format transformation between different system interfaces
- **Aggregator Pattern**: Collecting and combining related messages from multiple sources

Enterprise Integration Patterns Implemented:

Request-Reply Pattern:

- **Correlation ID Management**: Automatic correlation ID generation and tracking for request-response pairs
- **Timeout Handling**: Configurable timeouts with automatic cleanup of expired requests
- **Response Routing**: Intelligent routing of responses back to original requesters
- **Error Handling**: Comprehensive error handling for failed requests and timeouts

- **Performance Monitoring**: Request-response latency and success rate monitoring

Publish-Subscribe Pattern:

- **Topic Management**: Dynamic topic creation and subscription management
- **Subscriber Registration**: Automatic subscriber registration and deregistration
- **Event Broadcasting**: Efficient event distribution to multiple subscribers
- **Subscription Filtering**: Content-based subscription filtering and routing
- **Subscriber Health Monitoring**: Monitoring subscriber health and automatic cleanup

Message Router Pattern:

- **Rule-Based Routing**: Configurable routing rules based on message content and headers
- **Dynamic Routing**: Runtime routing decision making based on system state
- **Load-Based Routing**: Routing decisions based on destination load and capacity
- **Failover Routing**: Automatic failover to alternative destinations
- **Routing Metrics**: Comprehensive routing performance and decision metrics

Advanced Integration Features:

- **Message Transformation**: Automatic message format transformation between systems
- **Protocol Bridging**: Bridging between different messaging protocols and systems
- **Message Enrichment**: Adding contextual information to messages during routing
- **Message Validation**: Comprehensive message validation and error handling
- **Integration Monitoring**: End-to-end monitoring of integration flows

Microservices Communication Patterns:

- **Service Orchestration**: Coordinating complex business processes across multiple services
- **Service Choreography**: Event-driven service coordination without central orchestration
- **Saga Pattern**: Managing distributed transactions across multiple services
- **Circuit Breaker Integration**: Preventing cascade failures in service communication
- **Service Mesh Integration**: Integration with service mesh technologies for advanced routing

Real-World Applications:

- Enterprise application integration with legacy systems
- Microservices architectures with complex communication patterns
- Event-driven architectures with sophisticated routing requirements
- B2B integration with external partner systems
- API gateway implementations with advanced routing capabilities

Key Code Patterns:

```
// Request-Reply pattern with correlation ID management
private static void demonstrateRequestReplyPattern(QueueFactory factory) throws Exception {
    MessageProducer<RequestMessage> requestProducer =
        factory.createProducer("requests", RequestMessage.class);
    MessageConsumer<ResponseMessage> responseConsumer =
        factory.createConsumer("responses", ResponseMessage.class);

    // Map to track pending requests
    Map<String, CompletableFuture<ResponseMessage>> pendingRequests = new ConcurrentHashMap<>();
```

```
        // Response handler with correlation ID matching
        responseConsumer.subscribe(message -> {
            String correlationId = message.getHeaders().get("correlationId");
            CompletableFuture<ResponseMessage> future = pendingRequests.remove(correlationId);

            if (future != null) {
                future.complete(message.getPayload());
                logger.info("Response received for request: {}", correlationId);
            } else {
                logger.warn("Received response for unknown request: {}", correlationId);
            }

            return CompletableFuture.completedFuture(null);
        });

        // Send request with correlation ID and timeout
        String correlationId = UUID.randomUUID().toString();
        RequestMessage request = new RequestMessage("getData", Map.of("userId", "123"));

        CompletableFuture<ResponseMessage> responseFuture = new CompletableFuture<>();
        pendingRequests.put(correlationId, responseFuture);

        Map<String, String> headers = Map.of("correlationId", correlationId, "replyTo", "responses");
        requestProducer.send(request, headers);

        // Wait for response with timeout
        try {
            ResponseMessage response = responseFuture.get(30, TimeUnit.SECONDS);
            logger.info("Request-Reply completed: {}", response);
        } catch (TimeoutException e) {
            pendingRequests.remove(correlationId);
            logger.error("Request timed out: {}", correlationId);
        }
    }
```

14. SecurityConfigurationExample

Complexity: Expert **Purpose**: Comprehensive security implementation with SSL/TLS, compliance, and production security best practices

Detailed Description: This example demonstrates enterprise-grade security implementation for PeeGeeQ, covering SSL/TLS configuration, certificate management, credential security, compliance requirements, and comprehensive security monitoring.

Key Implementation Details:

- **Complete SSL/TLS Implementation**: End-to-end SSL/TLS configuration for all database connections with certificate validation
- **Certificate Management System**: Comprehensive certificate lifecycle management including generation, validation, rotation, and revocation
- **Credential Security**: Advanced credential management with encryption, rotation, and secure storage patterns
- **Security Event Monitoring**: Real-time security event logging, monitoring, and alerting system
- **Compliance Configuration**: Configuration patterns for GDPR, SOX, HIPAA, and other regulatory compliance requirements
- **Security Audit Framework**: Comprehensive audit logging and reporting for security compliance

Security Implementation Areas:

SSL/TLS Configuration:

- **Database Connection Security**: Complete SSL/TLS setup for PostgreSQL connections with certificate validation

- **Certificate Chain Validation**: Full certificate chain validation including intermediate certificates
- **Cipher Suite Configuration**: Secure cipher suite selection and configuration for optimal security
- **TLS Version Management**: TLS version enforcement and deprecated protocol prevention
- **Certificate Pinning**: Certificate pinning implementation for enhanced security

Certificate Management:

- **Certificate Generation**: Automated certificate generation for development and testing environments
- **Certificate Validation**: Comprehensive certificate validation including expiration, revocation, and trust chain
- **Certificate Rotation**: Automated certificate rotation with zero-downtime deployment
- **Certificate Storage**: Secure certificate storage with proper access controls and encryption
- **Certificate Monitoring**: Proactive monitoring of certificate expiration and health

Credential Management:

- **Password Encryption**: Strong password encryption using industry-standard algorithms
- **Credential Rotation**: Automated credential rotation with configurable schedules
- **Secure Storage**: Integration with secure credential storage systems (HashiCorp Vault, AWS Secrets Manager)
- **Access Control**: Role-based access control for credential management
- **Credential Auditing**: Comprehensive auditing of credential access and modifications

Security Monitoring and Alerting:

- **Security Event Logging**: Detailed logging of all security-related events and activities
- **Real-Time Monitoring**: Real-time monitoring of security events with immediate alerting
- **Anomaly Detection**: Automated detection of unusual security patterns and behaviors
- **Security Dashboards**: Comprehensive security dashboards for operations teams
- **Incident Response**: Automated incident response procedures for security events

Compliance Configuration:

GDPR Compliance:

- **Data Protection**: Implementation of data protection measures and privacy controls
- **Data Retention**: Automated data retention and deletion policies
- **Consent Management**: User consent tracking and management
- **Data Portability**: Data export and portability features
- **Breach Notification**: Automated breach detection and notification procedures

SOX Compliance:

- **Financial Data Protection**: Enhanced protection for financial data and transactions
- **Audit Trail**: Comprehensive audit trails for all financial operations
- **Access Controls**: Strict access controls for financial data and operations
- **Change Management**: Controlled change management processes with approval workflows
- **Segregation of Duties**: Implementation of segregation of duties principles

HIPAA Compliance:

- **PHI Protection**: Protected Health Information (PHI) encryption and access controls
- **Audit Logging**: Detailed audit logging for all PHI access and modifications
- **Access Controls**: Role-based access controls for healthcare data
- **Data Encryption**: End-to-end encryption for healthcare data in transit and at rest

- **Business Associate Agreements**: Framework for business associate compliance

Security Best Practices:

- **Defense in Depth**: Multi-layered security approach with redundant controls
- **Principle of Least Privilege**: Minimal access rights implementation
- **Security by Design**: Security considerations integrated into system design
- **Regular Security Assessments**: Automated security scanning and assessment
- **Security Training**: Security awareness and training programs

Real-World Applications:

- Financial services with strict regulatory requirements
- Healthcare systems with HIPAA compliance needs
- Government systems with security clearance requirements
- Enterprise systems with SOX compliance obligations
- Multi-tenant SaaS platforms with data isolation requirements

Key Code Patterns:

```
// SSL/TLS configuration for secure database connections
public class SecureDatabaseConfiguration {

    public DataSource createSecureDataSource() {
        HikariConfig config = new HikariConfig();

        // Basic connection settings
        config.setJdbcUrl("jdbc:postgresql://localhost:5432/peegeeq_secure");
        config.setUsername("peegeeq_secure_user");
        config.setPassword(getEncryptedPassword());

        // SSL/TLS configuration
        config.addDataSourceProperty("ssl", "true");
        config.addDataSourceProperty("sslmode", "require");
        config.addDataSourceProperty("sslcert", "/path/to/client-cert.pem");
        config.addDataSourceProperty("sslkey", "/path/to/client-key.pem");
        config.addDataSourceProperty("sslrootcert", "/path/to/ca-cert.pem");

        // Security properties
        config.addDataSourceProperty("sslhostnameverifier", "strict");
        config.addDataSourceProperty("sslfactory", "org.postgresql.ssl.DefaultJavaSSLFactory");

        // Connection pool security
        config.setMaximumPoolSize(20);
        config.setConnectionTimeout(30000);
        config.setIdleTimeout(600000);
        config.setMaxLifetime(1800000);

        // Enable connection validation
        config.setConnectionTestQuery("SELECT 1");
        config.setValidationTimeout(5000);

        return new HikariDataSource(config);
    }

    private String getEncryptedPassword() {
        // Retrieve password from secure credential store
        return credentialManager.getDecryptedPassword("peegeeq.database.password");
    }
```

```
}


// Comprehensive security event monitoring and alerting
public class SecurityEventMonitor {

    private final SecurityEventLogger eventLogger;
    private final AlertingService alertingService;

    public void monitorSecurityEvents() {
        // Monitor authentication events
        eventLogger.onAuthenticationEvent(event -> {
            if (event.getType() == AuthenticationEventType.FAILED_LOGIN) {
                handleFailedAuthentication(event);
            } else if (event.getType() == AuthenticationEventType.SUSPICIOUS_ACTIVITY) {
                handleSuspiciousActivity(event);
            }
        });

        // Monitor data access events
        eventLogger.onDataAccessEvent(event -> {
            if (event.isUnauthorizedAccess()) {
                handleUnauthorizedAccess(event);
            }

            // Log all data access for compliance
            auditLogger.logDataAccess(event.getUserId(), event.getResourceId(),
                event.getAccessType(), event.getTimestamp());
        });

        // Monitor configuration changes
        eventLogger.onConfigurationChangeEvent(event -> {
            securityAuditLogger.logConfigurationChange(
                event.getUserId(), event.getChangedProperty(),
                event.getOldValue(), event.getNewValue(), event.getTimestamp());

            if (event.isSecurityRelated()) {
                alertingService.sendSecurityAlert(
                    "Security configuration changed", event.getDetails());
            }
        });
    }

    private void handleFailedAuthentication(AuthenticationEvent event) {
        String userId = event.getUserId();
        int failureCount = authenticationFailureTracker.incrementFailureCount(userId);

        if (failureCount >= 5) {
            // Lock account after 5 failed attempts
            userAccountService.lockAccount(userId, Duration.ofMinutes(30));

            alertingService.sendSecurityAlert(
                "Account locked due to repeated failed authentication attempts",
                Map.of("userId", userId, "failureCount", String.valueOf(failureCount))
            );
        }
    }
}
```

15. TransactionalBiTemporalExample

Complexity: Expert **Purpose**: Advanced event sourcing with transactional consistency and bi-temporal data management

Detailed Description: This example demonstrates the most sophisticated event sourcing patterns, combining ACID transactional guarantees with bi-temporal event storage for complex business scenarios requiring both consistency and temporal data management.

Key Implementation Details:

- **Transactional Event Sourcing**: Complete integration of ACID transactions with event sourcing patterns
- **Bi-Temporal Transactions**: Managing both valid time and transaction time within transactional boundaries
- **Event Correction Workflows**: Sophisticated event correction patterns that maintain transactional integrity
- **Cross-Queue Transactions**: Coordinating transactions across multiple queues and event stores
- **Temporal Data Integrity**: Ensuring consistency in bi-temporal data with transactional guarantees
- **Complex Business Workflows**: Implementation of complex business processes with event sourcing and transactions

Advanced Transactional Patterns:

- **Distributed Transactions**: Coordinating transactions across multiple resources and systems
- **Saga Pattern Implementation**: Long-running business processes with compensating transactions
- **Event Store Transactions**: Transactional event appending with rollback capabilities
- **Queue-Event Store Coordination**: Coordinating message queues with event stores in single transactions
- **Temporal Transaction Isolation**: Isolation levels for bi-temporal data operations

Real-World Applications:

- Financial systems requiring audit trails with transactional consistency
- Healthcare systems with temporal data and regulatory compliance
- Supply chain systems with complex business process coordination
- Insurance systems with policy lifecycle management
- Legal systems with document versioning and audit requirements

Key Code Patterns:

```
// Transactional bi-temporal event sourcing
@Transactional
public class TransactionalBiTemporalEventStore {

    public CompletableFuture<TransactionResult> executeBusinessTransaction(
            BusinessTransaction transaction) {

        return transactionManager.executeInTransaction(() -> {
            List<BiTemporalEvent<?>> events = new ArrayList<>();

            // Process each step of the business transaction
            for (BusinessStep step : transaction.getSteps()) {
                BiTemporalEvent<?> event = processBusinessStep(step);
                events.add(event);

                // Store event with transactional guarantees
                eventStore.append(
                    event.getEventType(),
                    event.getPayload(),
                    step.getValidTime(),      // When it actually happened
                    event.getMetadata(),
                    transaction.getCorrelationId(),
                    step.getBusinessKey()
                ).join();

                // Update related queues within the same transaction
```

```
                if (step.requiresQueueUpdate()) {
                    queueManager.sendMessage(
                        step.getTargetQueue(),
                        createQueueMessage(event),
                        Map.of("transactionId", transaction.getId())
                    ).join();
                }
            }

            // Validate transaction consistency
            validateTransactionConsistency(transaction, events);

            return new TransactionResult(transaction.getId(), events);
        });
    }

    private void validateTransactionConsistency(BusinessTransaction transaction,
                                                List<BiTemporalEvent<?>> events) {
        // Ensure all events have consistent temporal relationships
        for (int i = 1; i < events.size(); i++) {
            BiTemporalEvent<?> current = events.get(i);
            BiTemporalEvent<?> previous = events.get(i - 1);

            if (current.getValidTime().isBefore(previous.getValidTime())) {
                throw new TemporalConsistencyException(
                    "Event valid time sequence violation in transaction: " +
                    transaction.getId());
            }
        }
    }
}



// Event correction within transactional boundaries
@Transactional
public CompletableFuture<CorrectionResult> correctHistoricalEvent(
        String originalEventId, Object correctedPayload, String correctionReason) {

    return transactionManager.executeInTransaction(() -> {
        // Retrieve original event
        BiTemporalEvent<?> originalEvent = eventStore.getEventById(originalEventId)
            .orElseThrow(() -> new EventNotFoundException(originalEventId));

        Instant correctionTime = Instant.now();

        // Create correction event with proper temporal metadata
        BiTemporalEvent<?> correctionEvent = BiTemporalEvent.builder()
            .eventType("EventCorrected")
            .payload(correctedPayload)
            .validTime(originalEvent.getValidTime())  // Same valid time as original
            .transactionTime(correctionTime)          // New transaction time
            .metadata(Map.of(
                "correctionReason", correctionReason,
                "originalEventId", originalEventId,
                "correctedBy", getCurrentUserId(),
                "correctionType", "HISTORICAL_CORRECTION"
            ))
            .correlationId(originalEvent.getCorrelationId())
            .businessKey(originalEvent.getBusinessKey())
            .build();

        // Store correction event
        eventStore.append(correctionEvent).join();

        // Update any dependent events or aggregates
```

```
            updateDependentEvents(originalEvent, correctionEvent);

            // Notify interested parties about the correction
            notificationService.sendCorrectionNotification(
                originalEventId, correctionEvent.getEventId(), correctionReason);

            return new CorrectionResult(originalEventId, correctionEvent.getEventId());
        });
    }
```

16. AdvancedConfigurationExample

Complexity: Expert **Purpose**: Enterprise-grade configuration management with externalization, security, and dynamic updates

Detailed Description: This example demonstrates sophisticated configuration management patterns essential for enterprise production systems, including externalized configuration, security, validation, and dynamic updates.

Key Implementation Details:

- **Externalized Configuration Management**: Complete separation of configuration from application code with external sources
- **Configuration Security**: Encryption, access controls, and secure storage of sensitive configuration data
- **Dynamic Configuration Updates**: Runtime configuration changes without application restarts or downtime
- **Configuration Validation Framework**: Comprehensive validation of configuration values with business rule enforcement
- **Configuration Monitoring**: Real-time monitoring of configuration changes and their system impact
- **Configuration Versioning**: Version control and rollback capabilities for configuration changes

Enterprise Configuration Features:

- **Multi-Source Configuration**: Integration with multiple configuration sources (files, databases, config servers)
- **Environment-Specific Overrides**: Sophisticated environment-based configuration inheritance and overrides
- **Configuration Templates**: Template-based configuration generation for different deployment scenarios
- **Configuration Auditing**: Complete audit trails for all configuration changes and access
- **Configuration Backup and Recovery**: Automated backup and recovery procedures for configuration data

Real-World Applications:

- Large-scale enterprise systems with complex configuration requirements
- Cloud-native applications with dynamic scaling and configuration needs
- Multi-tenant systems with tenant-specific configuration requirements
- Compliance-driven systems with configuration audit and control needs
- DevOps environments with configuration as code requirements

Key Code Patterns:

```
// Advanced configuration management with external sources
public class AdvancedConfigurationManager {

    private final ConfigurationSourceRegistry sourceRegistry;
    private final ConfigurationValidator validator;
    private final ConfigurationEncryption encryption;

    public void initializeAdvancedConfiguration() {
        // Register multiple configuration sources with priorities
        sourceRegistry.register(new EnvironmentVariableSource(), Priority.HIGH);
        sourceRegistry.register(new SystemPropertySource(), Priority.HIGH);
```

```
        sourceRegistry.register(new ConfigServerSource("https://config-server"), Priority.MEDIUM);
        sourceRegistry.register(new DatabaseConfigSource(), Priority.MEDIUM);
        sourceRegistry.register(new FileConfigSource("/etc/peegeeq/config.yml"), Priority.LOW);

        // Load and merge configurations from all sources
        CompositeConfiguration config = loadCompositeConfiguration();

        // Validate configuration against business rules
        ValidationResult validation = validator.validate(config);
        if (!validation.isValid()) {
            throw new ConfigurationValidationException(validation.getErrors());
        }

        // Apply configuration with hot-reload capability
        applyConfigurationWithHotReload(config);
    }

    private CompositeConfiguration loadCompositeConfiguration() {
        CompositeConfiguration.Builder builder = CompositeConfiguration.builder();

        // Load from each source in priority order
        for (ConfigurationSource source : sourceRegistry.getSourcesByPriority()) {
            try {
                Configuration sourceConfig = source.loadConfiguration();
                builder.addSource(source.getName(), sourceConfig, source.getPriority());

                logger.info("Loaded configuration from source: {} ({} properties)",
                    source.getName(), sourceConfig.size());
            } catch (Exception e) {
                logger.warn("Failed to load configuration from source: {}",
                    source.getName(), e);
            }
        }

        return builder.build();
    }
}



// Dynamic configuration updates without restart
public class DynamicConfigurationUpdater {

    private final ConfigurationChangeListener changeListener;
    private final ConfigurationApplier configApplier;

    public void enableDynamicUpdates() {
        // Watch for configuration changes
        configurationWatcher.onConfigurationChange(change -> {
            try {
                // Validate the change
                ValidationResult validation = validator.validateChange(change);
                if (!validation.isValid()) {
                    logger.error("Invalid configuration change rejected: {}",
                        validation.getErrors());
                    return;
                }

                // Apply the change dynamically
                applyConfigurationChange(change);

                // Audit the change
                auditLogger.logConfigurationChange(
                    change.getProperty(),
                    change.getOldValue(),
                    change.getNewValue(),
```

```
                    change.getChangedBy(),
                    Instant.now()
                );

                logger.info("Applied dynamic configuration change: {} = {}",
                    change.getProperty(), change.getNewValue());

            } catch (Exception e) {
                logger.error("Failed to apply configuration change", e);
                // Optionally rollback the change
                rollbackConfigurationChange(change);
            }
        });
    }

    private void applyConfigurationChange(ConfigurationChange change) {
        switch (change.getCategory()) {
            case DATABASE:
                databaseConfigApplier.applyChange(change);
                break;
            case QUEUE:
                queueConfigApplier.applyChange(change);
                break;
            case SECURITY:
                securityConfigApplier.applyChange(change);
                break;
            case PERFORMANCE:
                performanceConfigApplier.applyChange(change);
                break;
            default:
                genericConfigApplier.applyChange(change);
        }
    }
}
```

17. RestApiStreamingExample

Complexity: Expert **Purpose**: Advanced real-time streaming with WebSocket, Server-Sent Events, and connection management at scale

Detailed Description: This example demonstrates sophisticated real-time streaming capabilities, showing how to implement WebSocket and Server-Sent Events for high-performance, scalable real-time messaging systems.

Key Implementation Details:

- **WebSocket Streaming**: Full-duplex real-time messaging with WebSocket connections
- **Server-Sent Events (SSE)**: Unidirectional streaming for live updates and notifications
- **Connection Management**: Sophisticated connection lifecycle management for thousands of concurrent connections
- **Streaming Filters**: Real-time message filtering and routing for streaming connections
- **Consumer Group Streaming**: Integration of consumer groups with streaming connections
- **Connection Scaling**: Patterns for scaling streaming connections across multiple instances

Advanced Streaming Features:

- **Connection Pooling**: Efficient connection pooling and resource management
- **Backpressure Handling**: Managing backpressure in high-volume streaming scenarios
- **Connection Health Monitoring**: Real-time monitoring of connection health and performance
- **Streaming Analytics**: Real-time analytics and metrics for streaming connections
- **Connection Security**: Authentication and authorization for streaming connections

Real-World Applications:

- Real-time trading systems with live market data streaming
- IoT platforms with sensor data streaming
- Social media platforms with live activity feeds
- Gaming platforms with real-time multiplayer communication
- Monitoring systems with live dashboard updates

Key Code Patterns:

```
// WebSocket streaming with message filtering
private static void demonstrateWebSocketStreaming(HttpClient httpClient) throws Exception {
    logger.info("--- WebSocket Streaming ---");

    CountDownLatch wsLatch = new CountDownLatch(1);
    AtomicInteger messageCount = new AtomicInteger(0);

    // WebSocket connection with query parameters for filtering
    String wsUrl = "ws://localhost:8080/api/v1/queues/streaming-demo-setup/live-orders/stream" +
                   "?consumerGroup=ws-consumers" +
                   "&batchSize=5" +
                   "&maxWaitTime=1000" +
                   "&filter.priority=HIGH";  // Only high priority messages

    WebSocket webSocket = httpClient.newWebSocketBuilder()
        .buildAsync(URI.create(wsUrl), new WebSocket.Listener() {
            @Override
            public void onOpen(WebSocket webSocket) {
                logger.info("WebSocket connected for streaming");
                webSocket.request(1);
            }

            @Override
            public CompletionStage<?> onText(WebSocket webSocket, CharSequence data, boolean last) {
                JsonObject message = new JsonObject(data.toString());
                int count = messageCount.incrementAndGet();

                logger.info("WebSocket message #{}: {} (priority: {})",
                    count, message.getString("orderId"), message.getString("priority"));

                if (count >= 3) wsLatch.countDown();
                webSocket.request(1);
                return null;
            }
        }).join();

    wsLatch.await(30, TimeUnit.SECONDS);
    webSocket.sendClose(WebSocket.NORMAL_CLOSURE, "Demo complete");
}
```

```
// Server-Sent Events (SSE) streaming implementation
private static void demonstrateServerSentEvents(WebClient client) throws Exception {
    logger.info("--- Server-Sent Events Streaming ---");

    CountDownLatch sseLatch = new CountDownLatch(1);
    AtomicInteger eventCount = new AtomicInteger(0);

    // SSE endpoint with filtering parameters
    String sseUrl = "/api/v1/queues/streaming-demo-setup/live-orders/stream" +
                   "?consumerGroup=sse-consumers" +
```

```
                    "&batchSize=3" +
                    "&maxWaitTime=2000" +
                    "&filter.region=EU";  // Only EU region messages

        client.get(8080, "localhost", sseUrl)
            .putHeader("Accept", "text/event-stream")
            .putHeader("Cache-Control", "no-cache")
            .as(BodyCodec.pipe(WriteStream.newInstance(new Handler<Buffer>() {
                @Override
                public void handle(Buffer buffer) {
                    String data = buffer.toString();
                    if (data.startsWith("data: ")) {
                        String jsonData = data.substring(6).trim();
                        JsonObject event = new JsonObject(jsonData);
                        int count = eventCount.incrementAndGet();

                        logger.info("SSE event #{}: {} (region: {})",
                            count, event.getString("orderId"), event.getString("region"));

                        if (count >= 3) sseLatch.countDown();
                    }
                }
            })))
            .send(result -> {
                if (result.failed()) {
                    logger.error("SSE connection failed", result.cause());
                    sseLatch.countDown();
                }
            });

        sseLatch.await(30, TimeUnit.SECONDS);
    }
```

18. PeeGeeQExampleRunner

Complexity: Expert **Purpose**: Comprehensive example orchestration and execution framework

Detailed Description: This sophisticated utility provides a complete framework for running and managing all PeeGeeQ examples with advanced reporting, error handling, and execution control.

Key Implementation Details:

- **Example Orchestration**: Intelligent sequencing and execution of all 18 examples in logical order
- **Comprehensive Reporting**: Detailed execution reports with timing, success rates, and performance analysis
- **Error Resilience**: Continues execution even when individual examples fail, with detailed error reporting
- **Selective Execution**: Ability to run specific subsets of examples based on categories or names
- **Performance Analysis**: Comparative performance analysis across different examples
- **Resource Management**: Intelligent resource management and cleanup between example executions

Advanced Execution Features:

- **Example Categorization**: Sophisticated categorization system (Core, REST API, Service Discovery, Advanced)
- **Dependency Management**: Automatic handling of example dependencies and prerequisites
- **Execution Monitoring**: Real-time monitoring of example execution with progress reporting
- **Result Aggregation**: Comprehensive aggregation and analysis of execution results
- **Failure Analysis**: Detailed analysis of failures with troubleshooting recommendations

Real-World Applications:

- Continuous integration testing of messaging system functionality
- Performance benchmarking and regression testing
- Training and demonstration environments
- System validation and acceptance testing
- Development environment setup and validation

Advanced Test Examples

These are sophisticated JUnit test examples that demonstrate advanced patterns and serve as both tests and learning resources.

18. HighFrequencyProducerConsumerTest

Purpose: Performance and load testing for high-frequency scenarios **What it demonstrates**:

- **High-throughput messaging** - Testing system limits with high message volumes
- **Performance measurement** - Comprehensive throughput and latency metrics
- **Load testing patterns** - Systematic load testing methodologies
- **Resource management** - Memory and connection management under load
- **Concurrent processing** - Multi-threaded producer/consumer patterns

19. AdvancedProducerConsumerGroupTest

Purpose: Comprehensive testing of advanced producer and consumer group functionality **What it demonstrates**:

- **Advanced consumer groups** - Complex consumer group scenarios
- **Message filtering** - Content-based message filtering and routing
- **Performance characteristics** - Consumer group performance under various conditions
- **Fault tolerance** - Consumer group behavior during failures

20. ConsumerGroupResilienceTest

Purpose: Testing consumer group resilience and error handling **What it demonstrates**:

- **Failure scenarios** - Testing system behavior under various failure conditions
- **Recovery mechanisms** - Automatic recovery and failover patterns
- **System stability** - Maintaining stability under adverse conditions
- **Error propagation** - How errors propagate through consumer groups

21. NativeQueueFeatureTest

Purpose: Testing native queue features unique to PostgreSQL implementation **What it demonstrates**:

- **PostgreSQL LISTEN/NOTIFY** - Real-time messaging using PostgreSQL features
- **Advisory locks** - Message coordination using PostgreSQL advisory locks
- **Native performance** - Performance characteristics of native implementation
- **Resource management** - PostgreSQL-specific resource management

22-32. Additional Test Examples

- **BiTemporalEventStoreExampleTest** - Bi-temporal event store testing patterns
- **MultiConfigurationIntegrationTest** - Multi-environment integration testing

- **NativeVsOutboxComparisonTest** - Automated performance comparison testing
- **PeeGeeQExampleRunnerTest** - Testing the example runner utility
- **PeeGeeQExampleTest** - Basic functionality testing patterns
- **PeeGeeQSelfContainedDemoTest** - Self-contained demo testing
- **RestApiExampleTest** - REST API testing patterns
- **ServiceDiscoveryExampleTest** - Service discovery testing
- **ShutdownTest** - Graceful shutdown testing patterns
- **TestContainersShutdownTest** - TestContainers lifecycle management
- **TransactionalBiTemporalExampleTest** - Transactional event sourcing testing

Running Examples

Recommended: Use the Example Runner

Run all examples in logical order (best for learning):

```
# Run ALL 18 examples sequentially with comprehensive reporting
mvn compile exec:java -pl peegeeq-examples

# List all available examples with descriptions
mvn compile exec:java@list-examples -pl peegeeq-examples

# Run specific examples only
mvn compile exec:java -Dexec.mainClass="dev.mars.peegeeq.examples.PeeGeeQExampleRunner" -Dexec.args="self-contained rest-
```

Prerequisites

- **Java 21+** - Required for compilation and execution
- **Maven 3.8+** - For dependency management and execution
- **Docker** - Required for TestContainers (most examples)
- **Internet Connection** - For downloading Docker images

Individual Example Execution

```
# Self-contained demo (recommended starting point)
mvn compile exec:java -Dexec.mainClass="dev.mars.peegeeq.examples.PeeGeeQSelfContainedDemo" -pl peegeeq-examples

# Basic example (requires external PostgreSQL)
mvn compile exec:java -Dexec.mainClass="dev.mars.peegeeq.examples.PeeGeeQExample" -pl peegeeq-examples

# Advanced examples
mvn compile exec:java -Dexec.mainClass="dev.mars.peegeeq.examples.MessagePriorityExample" -pl peegeeq-examples
mvn compile exec:java -Dexec.mainClass="dev.mars.peegeeq.examples.EnhancedErrorHandlingExample" -pl peegeeq-examples
mvn compile exec:java -Dexec.mainClass="dev.mars.peegeeq.examples.PerformanceTuningExample" -pl peegeeq-examples
```

Running Test Examples

```
# Run all test examples
mvn test -pl peegeeq-examples
```

```
# Run specific test categories
mvn test -pl peegeeq-examples -Dtest="*HighFrequency*"
mvn test -pl peegeeq-examples -Dtest="*Resilience*"
mvn test -pl peegeeq-examples -Dtest="*Performance*"
```

Example Categories

By Complexity Level

- **Beginner** (3): PeeGeeQSelfContainedDemo **RECOMMENDED**, PeeGeeQExample, SimpleConsumerGroupTest
- **Intermediate** (5): ConsumerGroupExample, BiTemporalEventStoreExample, RestApiExample, MultiConfigurationExample, NativeVsOutboxComparisonExample
- **Advanced** (4): MessagePriorityExample, EnhancedErrorHandlingExample, PerformanceTuningExample, ServiceDiscoveryExample
- **Expert** (6): IntegrationPatternsExample, SecurityConfigurationExample, TransactionalBiTemporalExample, AdvancedConfigurationExample, RestApiStreamingExample, PeeGeeQExampleRunner

By Use Case

- **Getting Started**: PeeGeeQSelfContainedDemo, PeeGeeQExample, PeeGeeQExampleRunner
- **Real-time Messaging**: PeeGeeQExample, ConsumerGroupExample, RestApiStreamingExample
- **Transactional Messaging**: TransactionalBiTemporalExample, NativeVsOutboxComparisonExample
- **Event Sourcing**: BiTemporalEventStoreExample, TransactionalBiTemporalExample
- **Production Deployment**: SecurityConfigurationExample, AdvancedConfigurationExample, MultiConfigurationExample
- **Performance Optimization**: PerformanceTuningExample, NativeVsOutboxComparisonExample, HighFrequencyProducerConsumerTest
- **Integration**: IntegrationPatternsExample, ServiceDiscoveryExample, RestApiExample
- **Testing & Quality**: All test examples (15 total)

By Technology Focus

- **Core API**: PeeGeeQExample, ConsumerGroupExample, SimpleConsumerGroupTest
- **REST/HTTP**: RestApiExample, RestApiStreamingExample
- **Security**: SecurityConfigurationExample
- **Performance**: PerformanceTuningExample, HighFrequencyProducerConsumerTest, NativeVsOutboxComparisonTest
- **Configuration**: AdvancedConfigurationExample, MultiConfigurationExample
- **Service Discovery**: ServiceDiscoveryExample
- **Event Sourcing**: BiTemporalEventStoreExample, TransactionalBiTemporalExample
- **Testing**: All 15 test examples
- **Utilities**: PeeGeeQExampleRunner

Learning Path

Phase 1: Getting Started (45 minutes)

Goal: Understand PeeGeeQ fundamentals and see it in action

1. **PeeGeeQExampleRunner** (5 min) - Get overview of all examples

```
mvn compile exec:java@list-examples -pl peegeeq-examples
```

2. **PeeGeeQSelfContainedDemo RECOMMENDED** (15 min) - See everything in action

```
mvn compile exec:java -Dexec.mainClass="dev.mars.peegeeq.examples.PeeGeeQSelfContainedDemo" -pl peegeeq-examples
```

3. **PeeGeeQExample** (15 min) - Understand basic concepts
4. **SimpleConsumerGroupTest** (10 min) - Learn consumer group basics

Phase 2: Core Features (1.5 hours)

Goal: Master core PeeGeeQ capabilities

5. **ConsumerGroupExample** (20 min) - Advanced consumer patterns
6. **BiTemporalEventStoreExample** (25 min) - Event sourcing capabilities
7. **RestApiExample** (20 min) - HTTP interface usage
8. **MultiConfigurationExample** (15 min) - Configuration management
9. **NativeVsOutboxComparisonExample** (10 min) - Understand trade-offs

Phase 3: Advanced Patterns (2 hours)

Goal: Learn sophisticated messaging patterns

10. **MessagePriorityExample** (25 min) - Priority-based processing
11. **EnhancedErrorHandlingExample** (30 min) - Production error handling
12. **PerformanceTuningExample** (35 min) - Optimization techniques
13. **ServiceDiscoveryExample** (30 min) - Multi-instance deployment

Phase 4: Expert Level (2 hours)

Goal: Master complex production patterns

14. **IntegrationPatternsExample** (30 min) - Enterprise integration patterns
15. **SecurityConfigurationExample** (30 min) - Production security
16. **TransactionalBiTemporalExample** (30 min) - Advanced event sourcing
17. **AdvancedConfigurationExample** (15 min) - Production configuration
18. **RestApiStreamingExample** (15 min) - Real-time streaming

Phase 5: Testing & Advanced Scenarios (1 hour)

Goal: Understand testing patterns and advanced scenarios

19. **Run Test Suite** (30 min) - Execute all test examples

```
mvn test -pl peegeeq-examples
```

20. **Explore Specific Test Areas** (30 min):
    - **HighFrequencyProducerConsumerTest** - Performance testing
    - **ConsumerGroupResilienceTest** - Resilience patterns

- **NativeQueueFeatureTest** - Native PostgreSQL features

Total Learning Time: ~7 hours

- **Quick Overview**: 45 minutes (Phase 1 only)
- **Core Competency**: 2.25 hours (Phases 1-2)
- **Production Ready**: 4.25 hours (Phases 1-3)
- **Expert Level**: 6.25 hours (Phases 1-4)
- **Complete Mastery**: 7.25 hours (All phases)

Ready to start? Begin with the **PeeGeeQExampleRunner** to get an overview, then dive into the **PeeGeeQSelfContainedDemo** to see all features in action!