

PeeGeeQ Spring Boot Integration Guide

Purpose: Comprehensive guide for integrating PeeGeeQ with Spring Boot applications

Table of Contents

1. [Overview](#)
2. [Core Principles](#)
3. [Dependencies](#)
4. [Configuration](#)
5. [Repository Layer](#)
6. [Service Layer](#)
7. [Consumer Patterns](#)
8. [Bi-Temporal Event Store](#)
9. [Reactive Spring Boot](#)
10. [Example Use Cases](#)
11. [Common Mistakes](#)
12. [Testing](#)
13. [Quick Reference](#)

Overview

PeeGeeQ provides a complete database infrastructure including:

- Connection pool management (via Vert.x)
- Transaction management
- Transactional outbox pattern
- Bi-temporal event store
- Schema migrations
- Health checks and metrics

Key Principle: Your Spring Boot application should **host** PeeGeeQ, not create parallel database infrastructure.

Core Principles

1. Use PeeGeeQ's Public API

 **CORRECT** - Use these public API interfaces in Spring Boot context:

Foundation Classes

PeeGeeQManager - Foundation class that creates everything else

```
// In @Configuration class
@Bean
public PeeGeeQManager peeGeeQManager(MeterRegistry meterRegistry) {
    PeeGeeQConfiguration config = new PeeGeeQConfiguration("development");
    PeeGeeQManager manager = new PeeGeeQManager(config, meterRegistry);
    manager.start();
    return manager;
}
```

PeeGeeQConfiguration - Configuration for PeeGeeQManager

```
// In @Configuration class
PeeGeeQConfiguration config = new PeeGeeQConfiguration("development");
// Loads from peegeeq-development.properties
```

Database Operations

DatabaseService - Entry point for database operations

```
// In @Configuration class - create as bean
@Bean
public DatabaseService databaseService(PeeGeeQManager manager) {
    return new PgDatabaseService(manager);
}

// In @Service class - inject and use
@Service
public class OrderService {
    private final DatabaseService databaseService;

    public OrderService(DatabaseService databaseService) {
        this.databaseService = databaseService;
    }
}
```

ConnectionProvider - Manages connections and transactions

```
// In @Service class - obtain from DatabaseService
@Service
public class OrderService {
    private final DatabaseService databaseService;

    public CompletableFuture<Order> getOrder(String orderId) {
        ConnectionProvider cp = databaseService.getConnectionProvider();

        return cp.withTransaction("peegeeq-main", connection -> {
            // Use connection for all operations
            return orderRepository.findById(orderId, connection);
        }).toCompletionStage().toCompletableFuture();
    }
}
```

Messaging

MessageProducer - Sends messages to queues/outbox

```

// In @Configuration class - create as bean
@Bean
public MessageProducer<OrderEvent> orderEventProducer(QueueFactory factory) {
    return factory.createProducer("orders", OrderEvent.class);
}

// In @Service class - inject and use
@Service
public class OrderService {
    private final MessageProducer<OrderEvent> producer;

    // Simple send (creates its own transaction)
    public CompletableFuture<Void> publishEvent(OrderEvent event) {
        return producer.send(event);
    }

    // For transactional usage, see the "Service Layer" section below
    // which shows how to use ConnectionProvider.withTransaction()
    // to coordinate producer with other operations
}

```

MessageConsumer - Receives messages from queues/outbox

```

// In @Service class - create in @PostConstruct
@Service
public class OrderEventConsumer {
    private final QueueFactory queueFactory;
    private MessageConsumer<OrderEvent> consumer;

    @PostConstruct
    public void startConsumer() {
        consumer = queueFactory.createConsumer(
            "orders",
            OrderEvent.class,
            this::handleOrderEvent
        );
        consumer.start();
    }

    @PreDestroy
    public void stopConsumer() {
        if (consumer != null) {
            consumer.stop();
        }
    }

    private CompletableFuture<Void> handleOrderEvent(OrderEvent event) {
        // Process event
        return CompletableFuture.completedFuture(null);
    }
}

```

MessageHandler - Functional interface for processing messages

```

// Use as method reference
consumer = queueFactory.createConsumer("orders", OrderEvent.class, this::handleOrderEvent);

// Or as lambda
consumer = queueFactory.createConsumer("orders", OrderEvent.class,
    event -> {

```

```

        // Process event
        return CompletableFuture.completedFuture(null);
    }
};

```

QueueFactory - Creates producers and consumers

```

// In @Configuration class - create as bean
@Bean
public QueueFactory queueFactory(DatabaseService databaseService) {
    QueueFactoryProvider provider = new PgQueueFactoryProvider();
    OutboxFactoryRegistrar.registerWith((QueueFactoryRegistrar) provider);
    return provider.createFactory("outbox", databaseService);
}

// In @Service class - inject and use to create producers/consumers
@Service
public class MessagingService {
    private final QueueFactory queueFactory;

    public MessagingService(QueueFactory queueFactory) {
        this.queueFactory = queueFactory;
    }
}

```

QueueFactoryProvider - Provider for creating QueueFactory instances

```

// In @Configuration class
QueueFactoryProvider provider = new PgQueueFactoryProvider();
OutboxFactoryRegistrar.registerWith((QueueFactoryRegistrar) provider);
QueueFactory factory = provider.createFactory("outbox", databaseService);

```

ConsumerGroup - Consumer group for competing consumers pattern

```

// In @Service class - create in @PostConstruct
@Service
public class OrderProcessorGroup {
    private final QueueFactory queueFactory;
    private ConsumerGroup<OrderEvent> consumerGroup;

    @PostConstruct
    public void startConsumerGroup() {
        consumerGroup = queueFactory.createConsumerGroup(
            "order-processors", // Group name
            "orders",           // Topic
            OrderEvent.class,
            this::handleOrderEvent
        );
        consumerGroup.start();
    }

    @PreDestroy
    public void stopConsumerGroup() {
        if (consumerGroup != null) {
            consumerGroup.stop();
        }
    }
}

```

```
}
```

Bi-Temporal Event Store

EventStore - Bi-temporal event storage

```
// In @Configuration class - create as bean
@Bean
public EventStore<OrderEvent> orderEventStore(BiTemporalEventStoreFactory factory) {
    return factory.createEventStore(OrderEvent.class);
}

// In @Service class - inject and use
@Service
public class OrderEventService {
    private final EventStore<OrderEvent> eventStore;

    public CompletableFuture<BiTemporalEvent<OrderEvent>> appendEvent(OrderEvent event) {
        return eventStore.append("OrderCreated", event, Instant.now());
    }

    // Or in transaction:
    public CompletableFuture<BiTemporalEvent<OrderEvent>> appendInTransaction(
        OrderEvent event, SqlConnection connection) {
        return eventStore.appendInTransaction("OrderCreated", event, Instant.now(), connection);
    }
}
```

BiTemporalEventStoreFactory - Factory for creating EventStore instances

```
// In @Configuration class - create as bean
@Bean
public BiTemporalEventStoreFactory eventStoreFactory(PeeGeeQManager manager) {
    return new BiTemporalEventStoreFactory(manager);
}
```

BiTemporalEvent - Event wrapper with temporal dimensions

```
// Returned by EventStore methods
CompletableFuture<BiTemporalEvent<OrderEvent>> future = eventStore.append(...);

BiTemporalEvent<OrderEvent> event = future.get();
OrderEvent payload = event.getPayload();
Instant validTime = event.getValidTime();
Instant transactionTime = event.getTransactionTime();
String eventId = event.getEventId();
```

EventQuery - Query builder for EventStore

```
// In @Service class - use to query events
@Service
public class OrderQueryService {
    private final EventStore<OrderEvent> eventStore;

    public CompletableFuture<List<BiTemporalEvent<OrderEvent>>> getOrderHistory(String orderId) {
```

```

        EventQuery query = EventQuery.forAggregate(orderId);
        return eventStore.query(query);
    }

    public CompletableFuture<List<BiTemporalEvent<OrderEvent>>> getAllEvents() {
        EventQuery query = EventQuery.all();
        return eventStore.query(query);
    }

    public CompletableFuture<List<BiTemporalEvent<OrderEvent>>> getEventsByType(String eventType) {
        EventQuery query = EventQuery.forEventType(eventType);
        return eventStore.query(query);
    }

    public CompletableFuture<List<BiTemporalEvent<OrderEvent>>> getEventsAsOfTime(Instant validTime) {
        EventQuery query = EventQuery.asOfValidTime(validTime);
        return eventStore.query(query);
    }
}

```

Where these come from:

```

peegeeq-api module (interfaces - what you code against)
↓
peegeeq-db module (implementations)
- PgDatabaseService implements DatabaseService
- PgConnectionProvider implements ConnectionProvider
- PgQueueFactory implements QueueFactory

peegeeq-outbox module (implementations)
- OutboxProducer implements MessageProducer
- OutboxConsumer implements MessageConsumer

peegeeq-bitemporal module (implementations)
- BiTemporalEventStoreImpl implements EventStore

```

❌ WRONG - Don't use internal implementation classes:

```

// These are in peegeeq-db, peegeeq-outbox, peegeeq-bitemporal modules
// They are internal implementations - use the interfaces above instead

Pool                // Internal Vert.x pool (use ConnectionProvider)
PgClientFactory     // Internal factory (use DatabaseService)
PgConnectionManager // Internal connection manager (use ConnectionProvider)
OutboxProducer      // Internal implementation (use MessageProducer interface)
OutboxConsumer      // Internal implementation (use MessageConsumer interface)
PgDatabaseService   // Internal implementation (use DatabaseService interface)

```

2. Single Connection Pool

The Object Hierarchy:

```

PeeGeeQManager (created in @Configuration)
↓
DatabaseService (injected into your services)
↓
ConnectionProvider (provides connections and transactions)

```

↓

SqlConnection (the actual database connection used in lambdas)

✅ **CORRECT:** Use only PeeGeeQ's connection pool

Step 1: Spring creates these beans (in your `@Configuration` class):

```
@Bean
public PeeGeeQManager peeGeeQManager(MeterRegistry meterRegistry) {
    PeeGeeQConfiguration config = new PeeGeeQConfiguration("development");
    PeeGeeQManager manager = new PeeGeeQManager(config, meterRegistry);
    manager.start();
    return manager;
}

@Bean
public DatabaseService databaseService(PeeGeeQManager manager) {
    return new PgDatabaseService(manager); // Creates the connection pool internally
}
```

Step 2: Spring injects DatabaseService into your service:

```
@Service
public class OrderService {

    private final DatabaseService databaseService; // ← Injected by Spring

    public OrderService(DatabaseService databaseService) {
        this.databaseService = databaseService;
    }

    public CompletableFuture<Order> getOrder(String orderId) {
        // Get ConnectionProvider from the injected DatabaseService
        ConnectionProvider cp = databaseService.getConnectionProvider();

        // Use withConnection() for single operations (auto-commits)
        return cp.withConnection("peegeeq-main", connection -> {
            String sql = "SELECT * FROM orders WHERE id = $1";
            return connection.preparedQuery(sql)
                .execute(Tuple.of(orderId))
                .map(rowSet -> mapRowToOrder(rowSet.iterator().next()));
        }).toCompletionStage().toCompletableFuture();
    }

    public CompletableFuture<String> createOrder(Order order, OrderEvent event) {
        ConnectionProvider cp = databaseService.getConnectionProvider();

        // Use withTransaction() for multi-step operations
        return cp.withTransaction("peegeeq-main", connection -> {
            // All operations inside this lambda use the SAME connection
            // and participate in the SAME transaction
            return orderRepository.save(order, connection)
                .compose(v -> Future.fromCompletionStage(
                    producer.sendInTransaction(event, connection)
                ))
                .map(v -> order.getId());
        }).toCompletionStage().toCompletableFuture();
    }
}
```

Key Points:

- `DatabaseService` is injected by Spring (created from `PeeGeeQManager`)
- `ConnectionProvider` is obtained from `DatabaseService` (not injected directly)
- `SqlConnection` is provided by the lambda parameter (managed by `PeeGeeQ`)
- All operations use the **same connection pool** created by `PeeGeeQManager`

❌ **WRONG:** Don't create separate connection pools

```
// ❌ No R2DBC connection pools
@Bean
public ConnectionFactory connectionFactory() { ... }

// ❌ No separate Vert.x pools
Vertx vertx = Vertx.vertx();
Pool pool = PgBuilder.pool(...)

// ❌ No JDBC connection pools for application data
@Bean
public DataSource dataSource() { ... }
```

3. Transactional Consistency

The Complete Picture: How objects flow through a transactional operation

```
@Configuration creates beans:
    PeeGeeQManager → DatabaseService → QueueFactory → MessageProducer

@Service receives injected beans:
    DatabaseService (injected)
    MessageProducer (injected)
    OrderRepository (injected)

At runtime:
    DatabaseService.getConnectionProvider() → ConnectionProvider
    ConnectionProvider.withTransaction() → provides SqlConnection to lambda
    All operations use the SAME SqlConnection → ACID guarantees
```

✅ **CORRECT:** Share the same `SqlConnection` across all operations

What "share the same SqlConnection" means:

When you call `withTransaction()`, `PeeGeeQ` gives you a `SqlConnection` object as a lambda parameter. This is a **physical database connection** with an **active transaction**. To ensure all your operations are part of the **same transaction**, you must:

1. **Pass this connection object to every operation** (repository, producer, event store)
2. **Never create a new connection** inside the transaction
3. **Never call** `withTransaction()` **again** inside an existing transaction

Think of it like this:

- ✅ **ONE** `withTransaction()` call = **ONE** database transaction = **ONE** `SqlConnection` object
- ✅ Pass that **SAME** connection variable to all operations
- ✅ All operations commit **together** or rollback **together**


```

// The 'connection' parameter is the SAME object throughout
connectionProvider.withTransaction("client-id", connection -> {
    //                                     ↑
    //                                     This is your SqlConnection

    // Operation 1: Pass 'connection' to repository
    return orderRepository.save(order, connection) // ← Same connection

    // Operation 2: Pass 'connection' to producer
    .compose(v -> Future.fromCompletionStage(
        producer.sendInTransaction(event, connection) // ← Same connection
    ))

    // Operation 3: Pass 'connection' to event store
    .compose(v -> Future.fromCompletionStage(
        eventStore.appendInTransaction("OrderCreated", event, validTime, connection) // ← Same connection
    ));

    // All three operations use the SAME 'connection' variable
    // = All three operations are in the SAME database transaction
    // = If any fails, ALL rollback. If all succeed, ALL commit.
});

```

Complete working example showing object creation and usage:

```

// Step 1: Configuration creates the beans
@Configuration
public class PeeGeeQConfig {

    @Bean
    public PeeGeeQManager peeGeeQManager(MeterRegistry meterRegistry) {
        PeeGeeQConfiguration config = new PeeGeeQConfiguration("development");
        PeeGeeQManager manager = new PeeGeeQManager(config, meterRegistry);
        manager.start();
        return manager;
    }

    @Bean
    public DatabaseService databaseService(PeeGeeQManager manager) {
        return new PgDatabaseService(manager);
    }

    @Bean
    public QueueFactory outboxFactory(DatabaseService databaseService) {
        QueueFactoryProvider provider = new PgQueueFactoryProvider();
        OutboxFactoryRegistrar.registerWith((QueueFactoryRegistrar) provider);
        return provider.createFactory("outbox", databaseService);
    }

    @Bean
    public MessageProducer<OrderEvent> orderEventProducer(QueueFactory factory) {
        return factory.createProducer("orders", OrderEvent.class);
    }
}

// Step 2: Service receives injected beans and uses them transactionally
@Service
public class OrderService {

    private final DatabaseService databaseService; // ← Injected by Spring
    private final MessageProducer<OrderEvent> producer; // ← Injected by Spring
    private final OrderRepository orderRepository; // ← Injected by Spring

```

```

private final OrderItemRepository orderItemRepository; // ← Injected by Spring

public OrderService(DatabaseService databaseService,
                    MessageProducer<OrderEvent> producer,
                    OrderRepository orderRepository,
                    OrderItemRepository orderItemRepository) {
    this.databaseService = databaseService;
    this.producer = producer;
    this.orderRepository = orderRepository;
    this.orderItemRepository = orderItemRepository;
}

public CompletableFuture<String> createOrder(Order order, List<OrderItem> items) {
    // Get ConnectionProvider from injected DatabaseService
    ConnectionProvider connectionProvider = databaseService.getConnectionProvider();

    // Create ONE transaction - all operations use the SAME connection
    return connectionProvider.withTransaction("peegeeq-main", connection -> {
        //                                     ↑
        //                                     This 'connection' parameter is the SAME SqlConnection
        //                                     used by all operations below - ensuring ACID properties

        // Step 1: Save order (uses this connection)
        return orderRepository.save(order, connection) // ← Same connection

        // Step 2: Save order items (uses this connection)
        .compose(savedOrder ->
            orderItemRepository.saveAll(items, connection)) // ← Same connection

        // Step 3: Send outbox event (uses this connection)
        .compose(v -> Future.fromCompletionStage(
            producer.sendInTransaction(
                new OrderCreatedEvent(order),
                connection // ← Same connection - ALL in ONE transaction!
            )
        ))
        .map(v -> order.getId());

        // If ANY step fails, ALL steps rollback together
        // If ALL steps succeed, ALL steps commit together

    }).toCompletionStage().toCompletableFuture();
}
}

```

Key Points:

- **ONE** withTransaction() call = **ONE** database transaction
- The connection parameter is the **SAME** SqlConnection for all operations
- All operations **commit together** or **rollback together** (ACID)
- producer.sendInTransaction(event, connection) joins the existing transaction
- No separate connection pools, no separate transactions

✗ WRONG: Using separate transactions

```

// ✗ WRONG #1 - Each withTransaction() creates a SEPARATE transaction!
connectionProvider.withTransaction("client-1", conn1 -> {
    return orderRepository.save(order, conn1); // Transaction 1
})
.thenCompose(v -> connectionProvider.withTransaction("client-2", conn2 -> {
    return producer.sendInTransaction(event, conn2); // Transaction 2 - NO consistency!
}

```

```

    });
    // Problem: If producer fails, order is already committed!

    // ❌ WRONG #2 - Repository creates its own transaction
    connectionProvider.withTransaction("client-id", connection -> {
        return Future.fromCompletionStage(
            producer.sendInTransaction(event, connection)
        )
        .compose(v -> orderRepository.save(order)); // ❌ No connection passed!
    });
    // Problem: orderRepository.save() doesn't receive the connection,
    // so it creates its own transaction or fails

    // ❌ WRONG #3 - Mixing CompletableFuture and Future without proper conversion
    connectionProvider.withTransaction("client-id", connection -> {
        return producer.sendInTransaction(event, connection) // Returns CompletableFuture
            .thenCompose(v -> orderRepository.save(order, connection)); // ❌ Type mismatch!
    });
    // Problem: withTransaction() expects Future<T>, but you're returning CompletableFuture
    // Must wrap with Future.fromCompletionStage()

```

Dependencies

Maven Configuration

```

<dependencies>
  <!-- PeeGeeQ Core -->
  <dependency>
    <groupId>dev.mars</groupId>
    <artifactId>peegeeq-outbox</artifactId>
    <version>${peegeeq.version}</version>
  </dependency>

  <!-- Spring Boot Starter Web (for non-reactive) -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <!-- OR Spring Boot Starter WebFlux (for reactive) -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
  </dependency>

  <!-- Micrometer for metrics (optional) -->
  <dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
  </dependency>
</dependencies>

```

❌ DO NOT Include R2DBC

```

<!-- ❌ WRONG - Do NOT include these -->
<dependency>

```






```

    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-r2dbc</artifactId>
</dependency>
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>r2dbc-postgresql</artifactId>
</dependency>





```

Why R2DBC is NOT needed:

1. PeeGeeQ replaces ALL R2DBC operations


- o  PeeGeeQ provides reactive database access (Vert.x 5.x reactive patterns)
- o  PeeGeeQ provides connection pooling
- o  PeeGeeQ provides transaction management
- o  PeeGeeQ provides non-blocking I/O
- o  PeeGeeQ works with Spring WebFlux (wrap in Mono / Flux)




2. R2DBC creates incompatible infrastructure

- o  R2DBC creates a **separate connection pool** that cannot share transactions with PeeGeeQ
- o  R2DBC transactions are **isolated** from PeeGeeQ transactions
- o  You **cannot** have transactional consistency between R2DBC operations and PeeGeeQ outbox/event store
- o  This breaks the entire purpose of the transactional outbox pattern

3. What you get with PeeGeeQ instead of R2DBC:

```

// R2DBC would give you:
R2dbcRepository<Order, String> //  Separate connection pool

// PeeGeeQ gives you:
DatabaseService                //  Same connection pool
ConnectionProvider             //  Shared transactions
SqlConnection                  //  Direct Vert.x reactive access

```

Bottom line: PeeGeeQ is a **complete replacement** for R2DBC. You don't need both - PeeGeeQ does everything R2DBC does, plus transactional messaging and bi-temporal event storage.

Configuration

Spring Configuration Class

```

@Configuration
public class PeeGeeQConfig {

    private static final Logger log = LoggerFactory.getLogger(PeeGeeQConfig.class);

    @Autowired
    private ApplicationContext applicationContext;

    /**
     * Create and start PeeGeeQ Manager.
     * This initializes the connection pool, migrations, health checks, etc.
     */
    @Bean
    public PeeGeeQManager peeGeeQManager(

```


```

        @Value("${spring.profiles.active:default}") String profile,
        MeterRegistry meterRegistry) {

    log.info("Creating PeeGeeQ Manager with profile: {}", profile);

    PeeGeeQConfiguration config = new PeeGeeQConfiguration(profile);
    PeeGeeQManager manager = new PeeGeeQManager(config, meterRegistry);
    manager.start();

    log.info("PeeGeeQ Manager started successfully");
    return manager;
}

/**
 *  CORRECT: Expose DatabaseService for application use.
 * This is the entry point for all database operations.
 * Note: PgDatabaseService is the implementation, but we return the interface.
 */
@Bean
public DatabaseService databaseService(PeeGeeQManager manager) {
    log.info("Creating DatabaseService bean");
    return new PgDatabaseService(manager);
}

/**
 * Create outbox factory for producing/consuming events.
 */
@Bean
public QueueFactory outboxFactory(DatabaseService databaseService) {
    log.info("Creating outbox factory");

    QueueFactoryProvider provider = new PgQueueFactoryProvider();
    OutboxFactoryRegistrar.registerWith((QueueFactoryRegistrar) provider);
    QueueFactory factory = provider.createFactory("outbox", databaseService);

    log.info("Outbox factory created successfully");
    return factory;
}

/**
 * Create producer for order events.
 * Note: Cast to MessageProducer interface, not the implementation class.
 */
@Bean
public MessageProducer<OrderEvent> orderEventProducer(QueueFactory factory) {
    log.info("Creating order event producer");
    return factory.createProducer("orders", OrderEvent.class);
}

/**
 * Create bi-temporal event store factory (if using event store).
 */
@Bean
public BiTemporalEventStoreFactory eventStoreFactory(PeeGeeQManager manager) {
    log.info("Creating BiTemporalEventStoreFactory");
    return new BiTemporalEventStoreFactory(manager);
}

/**
 * Create event store for order events (if using bi-temporal event store).
 */
@Bean
public EventStore<OrderEvent> orderEventStore(BiTemporalEventStoreFactory factory) {
    log.info("Creating EventStore for OrderEvent");
    EventStore<OrderEvent> eventStore = factory.createEventStore(OrderEvent.class);
    log.info("EventStore created successfully");
}

```

```

        return eventStore;
    }

    /**
     * Initialize database schema on application startup.
     * Uses ApplicationContext to avoid circular dependency.
     */
    @EventListener(ApplicationReadyEvent.class)
    public void initializeSchema() {
        log.info("Initializing database schema");

        try {
            // Load schema SQL from classpath
            ClassPathResource resource = new ClassPathResource("schema.sql");
            String schemaSql;
            try (BufferedReader reader = new BufferedReader(
                new InputStreamReader(resource.getInputStream(), StandardCharsets.UTF_8))) {
                schemaSql = reader.lines().collect(Collectors.joining("\n"));
            }

            // Get DatabaseService from context (avoids circular dependency)
            DatabaseService databaseService = applicationContext.getBean(DatabaseService.class);
            ConnectionProvider connectionProvider = databaseService.getConnectionProvider();

            // Execute schema using PeeGeeQ's connection
            connectionProvider.withConnection("peegeeq-main", connection ->
                connection.query(schemaSql).execute().mapEmpty()
            )
                .onSuccess(result -> log.info("Database schema initialized successfully"))
                .onFailure(error -> log.error("Failed to initialize schema: {}", error.getMessage(), error))
                .toCompletionStage()
                .toCompletableFuture()
                .get();

        } catch (Exception e) {
            log.error("Error initializing database schema: {}", e.getMessage(), e);
            throw new RuntimeException("Failed to initialize database schema", e);
        }
    }
}

```

Configuration Properties

```

# application.properties
spring.application.name=my-peegeeq-app
spring.profiles.active=development

# PeeGeeQ will load from peegeeq-{profile}.properties
# Example: peegeeq-development.properties

# peegeeq-development.properties
peegeeq.database.host=localhost
peegeeq.database.port=5432
peegeeq.database.name=myapp_dev
peegeeq.database.user=postgres
peegeeq.database.password=postgres
peegeeq.database.pool.maxSize=20

```

```
// ❌ WRONG - Don't inject internal classes
@Bean
public Pool pgPool(PeeGeeQManager manager) {
    return manager.getClientFactory().getConnectionManager()... // Internal API!
}

// ❌ WRONG - Don't create separate R2DBC configuration
@Configuration
@EnableR2dbcRepositories
public class R2dbcConfig {
    @Bean
    public ConnectionFactory connectionFactory() { ... }
}
```

Repository Layer

✅ CORRECT: Repository Using SqlConnection

```
@Repository
public class OrderRepository {

    private static final Logger log = LoggerFactory.getLogger(OrderRepository.class);

    /**
     * Save order using the provided connection.
     * This ensures the operation participates in the caller's transaction.
     */
    public Future<Order> save(Order order, SqlConnection connection) {
        String sql = """
            INSERT INTO orders (id, customer_id, amount, status, created_at)
            VALUES ($1, $2, $3, $4, $5)
            """;

        return connection.preparedQuery(sql)
            .execute(Tuple.of(
                order.getId(),
                order.getCustomerId(),
                order.getAmount(),
                order.getStatus(),
                order.getCreatedAt()
            ))
            .map(result -> {
                log.info("Order saved successfully: {}", order.getId());
                return order;
            });
    }

    /**
     * Find order by ID using the provided connection.
     */
    public Future<Optional<Order>> findById(String id, SqlConnection connection) {
        String sql = "SELECT * FROM orders WHERE id = $1";

        return connection.preparedQuery(sql)
            .execute(Tuple.of(id))
            .map(rowSet -> {
                if (rowSet.size() == 0) {
                    return Optional.empty();
                }
            });
    }
}
```

```

    }
    Row row = rowSet.iterator().next();
    return Optional.of(mapRowToOrder(row));
  });
}

private Order mapRowToOrder(Row row) {
    Order order = new Order();
    order.setId(row.getString("id"));
    order.setCustomerId(row.getString("customer_id"));
    order.setAmount(row.getBigDecimal("amount"));
    order.setStatus(row.getString("status"));
    order.setCreatedAt(row.getLocalDateTime("created_at").toInstant(ZoneOffset.UTC));
    return order;
}
}

```

❌ WRONG Repository Patterns

```

// ❌ WRONG - Don't use R2DBC repositories
@Repository
public interface OrderRepository extends R2dbcRepository<Order, String> {
    // This uses a SEPARATE connection pool from PeeGeeQ!
}

// ❌ WRONG - Don't use JPA repositories
@Repository
public interface OrderRepository extends JpaRepository<Order, String> {
    // This uses JDBC, not PeeGeeQ's reactive pool!
}

// ❌ WRONG - Don't get connection inside repository
@Repository
public class OrderRepository {
    @Autowired
    private DatabaseService databaseService;

    public Future<Order> save(Order order) {
        // ❌ This creates a NEW transaction, not part of caller's transaction!
        return databaseService.getConnectionProvider()
            .withConnection("client-id", connection -> {
                // ...
            });
    }
}

```

Service Layer

✅ CORRECT: Service Using ConnectionProvider

```

@Service
public class OrderService {

    private static final Logger log = LoggerFactory.getLogger(OrderService.class);
    private static final String CLIENT_ID = "peegee-q-main";
}


```



```


private final DatabaseService databaseService;
private final MessageProducer<OrderEvent> orderEventProducer;
private final OrderRepository orderRepository;
private final OrderItemRepository orderItemRepository;

public OrderService(
    DatabaseService databaseService,
    MessageProducer<OrderEvent> orderEventProducer,
    OrderRepository orderRepository,
    OrderItemRepository orderItemRepository) {
    this.databaseService = databaseService;
    this.orderEventProducer = orderEventProducer;
    this.orderRepository = orderRepository;
    this.orderItemRepository = orderItemRepository;
}




/**
 *  CORRECT: Create order with transactional outbox pattern.
 *
 * All operations use the SAME connection from a SINGLE transaction.
 */
public CompletableFuture<String> createOrder(CreateOrderRequest request) {
    log.info("Creating order for customer: {}", request.getCustomerId());

    // Get ConnectionProvider from DatabaseService
    ConnectionProvider connectionProvider = databaseService.getConnectionProvider();

    // Create a single transaction
    return connectionProvider.withTransaction(CLIENT_ID, connection -> {
        Order order = new Order(request);
        String orderId = order.getId();

        // Step 1: Send outbox event (uses this connection)
        return Future.fromCompletionStage(
            orderEventProducer.sendInTransaction(
                new OrderCreatedEvent(request),
                connection //  Same connection
            )
        )
        // Step 2: Save order (uses this connection)
        .compose(v -> orderRepository.save(order, connection))

        // Step 3: Save order items (uses this connection)
        .compose(savedOrder ->
            orderItemRepository.saveAll(orderId, request.getItems(), connection)
        )

        // Step 4: Send additional events (uses this connection)
        .compose(v -> Future.fromCompletionStage(
            orderEventProducer.sendInTransaction(
                new OrderValidatedEvent(orderId),
                connection //  Same connection
            )
        ))
        .map(v -> orderId)
        .onSuccess(id -> log.info(" Order {} created successfully", id))
        .onFailure(error -> log.error(" Order creation failed: {}", error.getMessage()));

    }).toCompletionStage().toCompletableFuture();
}
}

```

WRONG Service Patterns

```
// ❌ WRONG - Using separate transactions
@Service
public class OrderService {

    public CompletableFuture<String> createOrder(CreateOrderRequest request) {
        // ❌ This creates TWO separate transactions!
        return producer.sendWithTransaction(event, TransactionPropagation.CONTEXT)
            .thenCompose(v -> orderRepository.save(order));
    }
}

// ❌ WRONG - Not passing connection to repository
@Service
public class OrderService {

    public CompletableFuture<String> createOrder(CreateOrderRequest request) {
        ConnectionProvider cp = databaseService.getConnectionProvider();

        return cp.withTransaction("client-id", connection -> {
            return Future.fromCompletionStage(
                producer.sendInTransaction(event, connection)
            )
            // ❌ Repository creates its own transaction!
            .compose(v -> Future.fromCompletionStage(
                orderRepository.save(order) // No connection parameter!
            ));
        }).toCompletionStage().toCompletableFuture();
    }
}

// ❌ WRONG - Using @Transactional with R2DBC
@Service
public class OrderService {

    @Transactional // ❌ This is for R2DBC/JDBC, not PeeGeeQ!
    public Mono<String> createOrder(CreateOrderRequest request) {
        return orderRepository.save(order)
            .flatMap(v -> producer.send(event));
    }
}
}
```

Consumer Patterns

✅ CORRECT Consumer Pattern

```
@Service
public class OrderEventConsumer {

    private final QueueFactory outboxFactory;
    private MessageConsumer<OrderEvent> consumer;

    @PostConstruct
    public void startConsumer() {
        consumer = outboxFactory.createConsumer(
            "orders",
            OrderEvent.class,
            this::handleOrderEvent
        );
    }
}
```

```

        consumer.start();
    }

    @PreDestroy
    public void stopConsumer() {
        if (consumer != null) {
            consumer.stop();
        }
    }

    private CompletableFuture<Void> handleOrderEvent(OrderEvent event) {
        // Process the event
        return CompletableFuture.completedFuture(null);
    }
}

```

CORRECT Consumer Group Pattern (Competing Consumers)

```

@Service
public class OrderEventConsumerGroup {

    private final QueueFactory outboxFactory;
    private ConsumerGroup<OrderEvent> consumerGroup;

    @PostConstruct
    public void startConsumerGroup() {
        // Create consumer group - multiple consumers share message processing
        consumerGroup = outboxFactory.createConsumerGroup(
            "order-processors",    // Consumer group name
            "orders",             // Topic name
            OrderEvent.class,     // Message type
            this::handleOrderEvent // Message handler
        );

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

    @PreDestroy
    public void stopConsumerGroup() {
        if (consumerGroup != null) {
            consumerGroup.stop();
        }
    }

    private CompletableFuture<Void> handleOrderEvent(OrderEvent event) {
        // Process the event
        // Multiple instances of this service will share the workload
        return CompletableFuture.completedFuture(null);
    }
}

```

Key Points about ConsumerGroup:

- Multiple consumer instances share message processing (competing consumers pattern)
- Each message is processed by **only one** consumer in the group
- Provides load balancing across multiple service instances
- Different from regular consumers where each consumer gets **all** messages

❌ WRONG Consumer Patterns

```
// ❌ WRONG - Creating consumer without proper lifecycle management
@Service
public class OrderEventConsumer {

    @Autowired
    public OrderEventConsumer(QueueFactory factory) {
        // ❌ Consumer created in constructor, not managed properly!
        MessageConsumer<OrderEvent> consumer = factory.createConsumer(...);
        consumer.start();
    }
}

// ❌ WRONG - Not stopping consumer on shutdown
@Service
public class OrderEventConsumer {

    @PostConstruct
    public void startConsumer() {
        consumer.start();
        // ❌ No @PreDestroy to stop consumer!
    }
}
```

Bi-Temporal Event Store

✅ CORRECT Event Store Pattern

```
@Service
public class OrderEventService {

    private final DatabaseService databaseService;
    private final EventStore<OrderEvent> eventStore;

    public CompletableFuture<BiTemporalEvent<OrderEvent>> recordOrderEvent(OrderEvent event) {
        ConnectionProvider cp = databaseService.getConnectionProvider();

        return cp.withTransaction("peegeeq-main", connection -> {
            // Append event to event store using appendInTransaction()
            return Future.fromCompletionStage(
                eventStore.appendInTransaction(
                    "OrderCreated", // Event type
                    event,          // Event payload
                    event.getValidTime(), // Valid time (when it happened)
                    connection      // ✅ Use transaction connection
                )
            );
        }).toCompletionStage().toCompletableFuture();
    }

    public CompletableFuture<List<BiTemporalEvent<OrderEvent>>> getOrderHistory(String orderId) {
        // Query events using EventQuery
        EventQuery query = EventQuery.forAggregate(orderId);
        return eventStore.query(query);
    }
}
```

```
}
```

✅ CORRECT Combined Outbox + Event Store Pattern

```
@Service
public class OrderService {

    private final DatabaseService databaseService;
    private final MessageProducer<OrderEvent> outboxProducer;
    private final EventStore<OrderEvent> eventStore;
    private final OrderRepository orderRepository;

    public CompletableFuture<String> createOrder(CreateOrderRequest request) {
        ConnectionProvider cp = databaseService.getConnectionProvider();

        return cp.withTransaction("peegeeq-main", connection -> {
            Order order = new Order(request);
            OrderEvent event = new OrderCreatedEvent(order);

            // All three operations in SAME transaction
            return Future.fromCompletionStage(
                // 1. Send to outbox (for immediate processing)
                outboxProducer.sendInTransaction(event, connection)
            )
                .compose(v -> Future.fromCompletionStage(
                    // 2. Append to event store (for historical queries)
                    eventStore.appendInTransaction(
                        "OrderCreated", // Event type
                        event,           // Event payload
                        event.getValidTime(), // Valid time
                        connection       // Same connection
                    )
                ))
                .compose(v ->
                    // 3. Save order data
                    orderRepository.save(order, connection)
                )
                .map(v -> order.getId());
        }).toCompletionStage().toCompletableFuture();
    }
}
```

❌ WRONG Event Store Patterns

```
// ❌ WRONG - Not using transaction connection
@Service
public class OrderEventService {

    public CompletableFuture<BiTemporalEvent<OrderEvent>> recordOrderEvent(OrderEvent event) {
        // ❌ Event store creates its own transaction!
        return eventStore.append(
            "OrderCreated", // Event type
            event,           // Event payload
            event.getValidTime() // Valid time
            // ❌ Missing connection parameter - creates separate transaction!
        );
    }
}
```

```
// ❌ WRONG - Separate transactions for outbox and event store
@Service
public class OrderService {

    public CompletableFuture<String> createOrder(CreateOrderRequest request) {
        // ❌ Two separate transactions - NO consistency!
        return outboxProducer.sendWithTransaction(event, TransactionPropagation.CONTEXT)
            .thenCompose(v -> eventStore.append("OrderCreated", event, event.getValidTime()));
    }
}

// ❌ WRONG - Using wrong method signature
@Service
public class OrderService {

    public CompletableFuture<String> createOrder(CreateOrderRequest request) {
        return connectionProvider.withTransaction("client-id", connection -> {
            // ❌ Wrong method - append() takes eventType, not eventId!
            return Future.fromCompletionStage(
                eventStore.append(order.getId(), event, event.getValidTime(), connection)
            );
        }).toCompletionStage().toCompletableFuture();
    }
}
```

Reactive Spring Boot

For reactive Spring Boot applications (WebFlux), wrap `CompletableFuture` in `Mono`.

ReactiveOutboxAdapter

```
@Component
public class ReactiveOutboxAdapter {

    /**
     * Convert CompletableFuture to Mono.
     */
    public <T> Mono<T> toMono(CompletableFuture<T> future) {
        return Mono.fromFuture(future);
    }

    /**
     * Convert CompletableFuture<Void> to Mono<Void>.
     */
    public Mono<Void> toMonoVoid(CompletableFuture<Void> future) {
        return Mono.fromFuture(future);
    }
}
```

Reactive Service Example

```
@Service
public class OrderService {

    private final DatabaseService databaseService;
    private final MessageProducer<OrderEvent> orderEventProducer;
```

```

private final OrderRepository orderRepository;
private final ReactiveOutboxAdapter adapter;

/**
 *  CORRECT: Reactive service using PeeGeeQ.
 * Wraps CompletableFuture in Mono for Spring WebFlux compatibility.
 */
public Mono<String> createOrder(CreateOrderRequest request) {
    ConnectionProvider connectionProvider = databaseService.getConnectionProvider();

    // Same pattern as non-reactive, but wrapped in Mono
    return adapter.toMono(
        connectionProvider.withTransaction("peegeeq-main", connection -> {
            Order order = new Order(request);

            return Future.fromCompletionStage(
                orderEventProducer.sendInTransaction(
                    new OrderCreatedEvent(request),
                    connection
                )
            )
        })
        .compose(v -> orderRepository.save(order, connection))
        .map(v -> order.getId());

    }).toCompletionStage().toCompletableFuture()
};
}

```

Reactive Controller Example

```

@RestController
@RequestMapping("/api/orders")
public class OrderController {

    private final OrderService orderService;

    @PostMapping
    public Mono<ResponseEntity<OrderResponse>> createOrder(
        @RequestBody CreateOrderRequest request) {

        return orderService.createOrder(request)
            .map(orderId -> ResponseEntity.ok(new OrderResponse(orderId)))
            .onErrorResume(error ->
                Mono.just(ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build())
            );
    }
}

```

Example Use Cases

This section demonstrates various use cases for PeeGeeQ with Spring Boot, showing that you can handle **all types of database operations** without needing R2DBC. The examples are organized by category to help you find patterns relevant to your application.

Category 1: Pure CRUD Operations (No Messaging)

These examples show standard database operations **without** using the outbox pattern or messaging. Use these patterns when you just need regular data access.

Example 1A: Customer Management Service (Non-Reactive)

Purpose: Standard CRUD operations without any messaging

Components:

- Customer.java - Plain POJO (no R2DBC annotations)
- CustomerRepository.java - Vert.x SQL Client CRUD operations
- CustomerService.java - Standard service layer

Operations:

```
// Create
public CompletableFuture<Customer> createCustomer(Customer customer) {
    ConnectionProvider cp = databaseService.getConnectionProvider();
    return cp.withConnection("peegee-main", connection -> {
        String sql = "INSERT INTO customers (id, name, email, created_at) VALUES ($1, $2, $3, $4)";
        return connection.preparedQuery(sql)
            .execute(Tuple.of(customer.getId(), customer.getName(),
                             customer.getEmail(), customer.getCreatedAt()))
            .map(result -> customer);
    }).toCompletionStage().toCompletableFuture();
}

// Read by ID
public CompletableFuture<Optional<Customer>> findById(String id) {
    ConnectionProvider cp = databaseService.getConnectionProvider();
    return cp.withConnection("peegee-main", connection -> {
        String sql = "SELECT * FROM customers WHERE id = $1";
        return connection.preparedQuery(sql)
            .execute(Tuple.of(id))
            .map(rowSet -> {
                if (rowSet.size() == 0) return Optional.empty();
                return Optional.of(mapRowToCustomer(rowSet.iterator().next()));
            });
    }).toCompletionStage().toCompletableFuture();
}

// Update
public CompletableFuture<Customer> updateCustomer(Customer customer) {
    ConnectionProvider cp = databaseService.getConnectionProvider();
    return cp.withConnection("peegee-main", connection -> {
        String sql = "UPDATE customers SET name = $1, email = $2, updated_at = $3 WHERE id = $4";
        return connection.preparedQuery(sql)
            .execute(Tuple.of(customer.getName(), customer.getEmail(),
                             Instant.now(), customer.getId()))
            .map(result -> customer);
    }).toCompletionStage().toCompletableFuture();
}

// Delete
public CompletableFuture<Void> deleteCustomer(String id) {
    ConnectionProvider cp = databaseService.getConnectionProvider();
    return cp.withConnection("peegee-main", connection -> {
        String sql = "DELETE FROM customers WHERE id = $1";
        return connection.preparedQuery(sql)
            .execute(Tuple.of(id))
            .mapEmpty();
    }).toCompletionStage().toCompletableFuture();
}
```



```

}

// List with pagination
public CompletableFuture<List<Customer>> findAll(int page, int size) {
    ConnectionProvider cp = databaseService.getConnectionProvider();
    return cp.withConnection("peegeeq-main", connection -> {
        String sql = "SELECT * FROM customers ORDER BY created_at DESC LIMIT $1 OFFSET $2";
        return connection.preparedQuery(sql)
            .execute(Tuple.of(size, page * size))
            .map(rowSet -> {
                List<Customer> customers = new ArrayList<>();
                rowSet.forEach(row -> customers.add(mapRowToCustomer(row)));
                return customers;
            });
    }).toCompletionStage().toCompletableFuture();
}

// Search by name
public CompletableFuture<List<Customer>> searchByName(String name) {
    ConnectionProvider cp = databaseService.getConnectionProvider();
    return cp.withConnection("peegeeq-main", connection -> {
        String sql = "SELECT * FROM customers WHERE name ILIKE $1 ORDER BY name";
        return connection.preparedQuery(sql)
            .execute(Tuple.of("%" + name + "%"))
            .map(rowSet -> {
                List<Customer> customers = new ArrayList<>();
                rowSet.forEach(row -> customers.add(mapRowToCustomer(row)));
                return customers;
            });
    }).toCompletionStage().toCompletableFuture();
}

// Check existence
public CompletableFuture<Boolean> existsByEmail(String email) {
    ConnectionProvider cp = databaseService.getConnectionProvider();
    return cp.withConnection("peegeeq-main", connection -> {
        String sql = "SELECT COUNT(*) FROM customers WHERE email = $1";
        return connection.preparedQuery(sql)
            .execute(Tuple.of(email))
            .map(rowSet -> rowSet.iterator().next().getLong(0) > 0);
    }).toCompletionStage().toCompletableFuture();
}

```

REST Endpoints:

```

@RestController
@RequestMapping("/api/customers")
public class CustomerController {

    @PostMapping
    public CompletableFuture<ResponseEntity<Customer>> create(@RequestBody Customer customer) {
        return customerService.createCustomer(customer)
            .thenApply(ResponseEntity::ok);
    }

    @GetMapping("/{id}")
    public CompletableFuture<ResponseEntity<Customer>> getById(@PathVariable String id) {
        return customerService.findById(id)
            .thenApply(opt -> opt.map(ResponseEntity::ok)
                .orElse(ResponseEntity.notFound().build()));
    }

    @PutMapping("/{id}")

```

```

public CompletableFuture<ResponseEntity<Customer>> update(
    @PathVariable String id, @RequestBody Customer customer) {
    customer.setId(id);
    return customerService.updateCustomer(customer)
        .thenApply(ResponseEntity::ok);
}







@DeleteMapping("/{id}")
public CompletableFuture<ResponseEntity<Void>> delete(@PathVariable String id) {
    return customerService.deleteCustomer(id)
        .thenApply(v -> ResponseEntity.noContent().build());
}

@GetMapping
public CompletableFuture<ResponseEntity<List<Customer>>> list(
    @RequestParam(defaultValue = "0") int page,
    @RequestParam(defaultValue = "20") int size) {
    return customerService.findAll(page, size)
        .thenApply(ResponseEntity::ok);
}

@GetMapping("/search")
public CompletableFuture<ResponseEntity<List<Customer>>> search(
    @RequestParam String name) {
    return customerService.searchByName(name)
        .thenApply(ResponseEntity::ok);
}
}

```

Key Points:

-  No outbox events needed for simple CRUD
-  Use `withConnection()` for single operations (auto-commits)
-  Use `withTransaction()` for multi-step updates
-  Pagination using LIMIT/OFFSET
-  Full-text search with ILIKE
-  No R2DBC required

Example 1B: Product Catalog Service (Reactive)

Purpose: Reactive CRUD operations without messaging

Components:

- `Product.java` and `Category.java` - Plain POJOs
- `ProductRepository.java` and `CategoryRepository.java` - Vert.x SQL Client
- `ProductService.java` - Reactive service layer

Operations:

```

@Service
public class ProductService {

    private final DatabaseService databaseService;
    private final ReactiveOutboxAdapter adapter;

    // Create product - returns Mono

```

```

public Mono<Product> createProduct(Product product) {
    ConnectionProvider cp = databaseService.getConnectionProvider();
    return adapter.toMono(
        cp.withConnection("peegeeq-main", connection -> {
            String sql = "INSERT INTO products (id, name, category_id, price) VALUES ($1, $2, $3, $4)";
            return connection.preparedQuery(sql)
                .execute(Tuple.of(product.getId(), product.getName(),
                    product.getCategoryId(), product.getPrice()))
                .map(result -> product);
        }).toCompletionStage().toCompletableFuture()
    );
}

// Find with category (JOIN) - returns Mono
public Mono<ProductWithCategory> findByIdWithCategory(String id) {
    ConnectionProvider cp = databaseService.getConnectionProvider();
    return adapter.toMono(
        cp.withConnection("peegeeq-main", connection -> {
            String sql = """
                SELECT p.*, c.name as category_name
                FROM products p
                LEFT JOIN categories c ON p.category_id = c.id
                WHERE p.id = $1
            """;
            return connection.preparedQuery(sql)
                .execute(Tuple.of(id))
                .map(rowSet -> {
                    if (rowSet.size() == 0) return null;
                    return mapRowToProductWithCategory(rowSet.iterator().next());
                });
        }).toCompletionStage().toCompletableFuture()
    );
}

// List all products - returns Flux
public Flux<Product> findAllProducts() {
    ConnectionProvider cp = databaseService.getConnectionProvider();
    return Flux.from(
        adapter.toMono(
            cp.withConnection("peegeeq-main", connection -> {
                String sql = "SELECT * FROM products ORDER BY name";
                return connection.preparedQuery(sql)
                    .execute()
                    .map(rowSet -> {
                        List<Product> products = new ArrayList<>();
                        rowSet.forEach(row -> products.add(mapRowToProduct(row)));
                        return products;
                    });
            }).toCompletionStage().toCompletableFuture()
        ).flatMapMany(Flux::fromIterable)
    );
}

// Batch insert products
public Mono<Integer> importProducts(List<Product> products) {
    ConnectionProvider cp = databaseService.getConnectionProvider();
    return adapter.toMono(
        cp.withTransaction("peegeeq-main", connection -> {
            String sql = "INSERT INTO products (id, name, category_id, price) VALUES ($1, $2, $3, $4)";

            List<Tuple> batch = products.stream()
                .map(p -> Tuple.of(p.getId(), p.getName(), p.getCategoryId(), p.getPrice()))
                .collect(Collectors.toList());

            return connection.preparedQuery(sql)
                .executeBatch(batch)
        })
    );
}

```

```

        .map(result -> products.size());
    }).toCompletionStage().toCompletableFuture()
    );
}

// Conditional update - only if price changed
public Mono<Boolean> updatePriceIfChanged(String id, BigDecimal newPrice) {
    ConnectionProvider cp = databaseService.getConnectionProvider();
    return adapter.toMono(
        cp.withTransaction("peegeeq-main", connection -> {
            String sql = ""
                UPDATE products
                SET price = $1, updated_at = $2
                WHERE id = $3 AND price != $1
            "";
            return connection.preparedQuery(sql)
                .execute(Tuple.of(newPrice, Instant.now(), id))
                .map(result -> result.rowCount() > 0);
        }).toCompletionStage().toCompletableFuture()
    );
}

// Soft delete
public Mono<Void> deactivateProduct(String id) {
    ConnectionProvider cp = databaseService.getConnectionProvider();
    return adapter.toMonoVoid(
        cp.withConnection("peegeeq-main", connection -> {
            String sql = "UPDATE products SET active = false, updated_at = $1 WHERE id = $2";
            return connection.preparedQuery(sql)
                .execute(Tuple.of(Instant.now(), id))
                .mapEmpty();
        }).toCompletionStage().toCompletableFuture()
    );
}

// Count by category
public Mono<Map<String, Long>> countByCategory() {
    ConnectionProvider cp = databaseService.getConnectionProvider();
    return adapter.toMono(
        cp.withConnection("peegeeq-main", connection -> {
            String sql = ""
                SELECT c.name, COUNT(p.id) as product_count
                FROM categories c
                LEFT JOIN products p ON c.id = p.category_id
                GROUP BY c.name
                ORDER BY product_count DESC
            "";
            return connection.preparedQuery(sql)
                .execute()
                .map(rowSet -> {
                    Map<String, Long> counts = new HashMap<>();
                    rowSet.forEach(row -> counts.put(
                        row.getString("name"),
                        row.getLong("product_count")
                    ));
                    return counts;
                });
        }).toCompletionStage().toCompletableFuture()
    );
}
}

```

Reactive Controller:

```

@RestController
@RequestMapping("/api/products")
public class ProductController {

    @PostMapping
    public Mono<ResponseEntity<Product>> create(@RequestBody Product product) {
        return productService.createProduct(product)
            .map(ResponseEntity::ok);
    }

    @GetMapping("/{id}")
    public Mono<ResponseEntity<ProductWithCategory>> getById(@PathVariable String id) {
        return productService.findByIdWithCategory(id)
            .map(ResponseEntity::ok)
            .defaultIfEmpty(ResponseEntity.notFound().build());
    }







    @GetMapping
    public Flux<Product> listAll() {
        return productService.findAllProducts();
    }

    @PostMapping("/import")
    public Mono<ResponseEntity<ImportResult>> importBatch(@RequestBody List<Product> products) {
        return productService.importProducts(products)
            .map(count -> ResponseEntity.ok(new ImportResult(count)));
    }

    @GetMapping("/stats/by-category")
    public Mono<ResponseEntity<Map<String, Long>>> getStatsByCategory() {
        return productService.countByCategory()
            .map(ResponseEntity::ok);
    }
}

```

Key Points:

-  Wrap `CompletableFuture` in `Mono` / `Flux` for Spring WebFlux
-  No R2DBC needed for reactive operations
-  Complex queries with JOINS using Vert.x SQL Client
-  Batch operations with `executeBatch()`
-  Aggregations and GROUP BY queries
-  Conditional updates and soft deletes

Category 2: Read-Heavy Operations

These examples show complex queries, aggregations, and reporting without messaging.

Example 2A: Reporting Service (Non-Reactive)

Purpose: Complex queries, aggregations, and reporting

Components:

- `OrderSummary.java` , `SalesReport.java` , `CustomerStats.java` - DTOs
- `ReportingRepository.java` - Complex queries
- `ReportingService.java` - Report generation

Operations:

```
@Service
public class ReportingService {

    private final DatabaseService databaseService;

    // Sales summary by month
    public CompletableFuture<List<MonthlySales>> getSalesByMonth(
        LocalDate startDate, LocalDate endDate) {
        ConnectionProvider cp = databaseService.getConnectionProvider();
        return cp.withConnection("peegeeq-main", connection -> {
            String sql = """
                SELECT
                    DATE_TRUNC('month', created_at) as month,
                    COUNT(*) as order_count,
                    SUM(amount) as total_sales,
                    AVG(amount) as avg_order_value
                FROM orders
                WHERE created_at >= $1 AND created_at < $2
                GROUP BY DATE_TRUNC('month', created_at)
                ORDER BY month DESC
            """;
            return connection.preparedQuery(sql)
                .execute(Tuple.of(startDate, endDate))
                .map(rowSet -> {
                    List<MonthlySales> results = new ArrayList<>();
                    rowSet.forEach(row -> results.add(mapRowToMonthlySales(row)));
                    return results;
                });
        }).toCompletionStage().toCompletableFuture();
    }

    // Top customers with order details
    public CompletableFuture<List<CustomerStats>> getTopCustomers(int limit) {
        ConnectionProvider cp = databaseService.getConnectionProvider();
        return cp.withConnection("peegeeq-main", connection -> {
            String sql = """
                WITH customer_totals AS (
                    SELECT customer_id,
                           COUNT(*) as order_count,
                           SUM(amount) as total_spent
                    FROM orders
                    GROUP BY customer_id
                )
                SELECT c.id, c.name, c.email,
                       ct.order_count, ct.total_spent
                FROM customers c
                JOIN customer_totals ct ON c.id = ct.customer_id
                ORDER BY ct.total_spent DESC
                LIMIT $1
            """;
            return connection.preparedQuery(sql)
                .execute(Tuple.of(limit))
                .map(rowSet -> {
                    List<CustomerStats> results = new ArrayList<>();
                    rowSet.forEach(row -> results.add(mapRowToCustomerStats(row)));
                    return results;
                });
        }).toCompletionStage().toCompletableFuture();
    }

    // Order details with items (multi-table join)
    public CompletableFuture<OrderWithDetails> getOrderDetails(String orderId) {
```

```

ConnectionProvider cp = databaseService.getConnectionProvider();
return cp.withConnection("peegeeq-main", connection -> {
    String sql = """
        SELECT
            o.id, o.customer_id, o.amount, o.status, o.created_at,
            c.name as customer_name, c.email as customer_email,
            oi.id as item_id, oi.product_id, oi.quantity, oi.price
        FROM orders o
        JOIN customers c ON o.customer_id = c.id
        LEFT JOIN order_items oi ON o.id = oi.order_id
        WHERE o.id = $1
        """;
    return connection.preparedQuery(sql)
        .execute(Tuple.of(orderId))
        .map(rowSet -> mapRowSetToOrderWithDetails(rowSet));
}).toCompletionStage().toCompletableFuture();
}

// Running totals with window functions
public CompletableFuture<List<DailySalesWithRunningTotal>> getDailySalesWithRunningTotal(
    LocalDate startDate, LocalDate endDate) {
    ConnectionProvider cp = databaseService.getConnectionProvider();
    return cp.withConnection("peegeeq-main", connection -> {
        String sql = """
            SELECT
                DATE(created_at) as sale_date,
                SUM(amount) as daily_total,
                SUM(SUM(amount)) OVER (ORDER BY DATE(created_at)) as running_total
            FROM orders
            WHERE created_at >= $1 AND created_at < $2
            GROUP BY DATE(created_at)
            ORDER BY sale_date
            """;
        return connection.preparedQuery(sql)
            .execute(Tuple.of(startDate, endDate))
            .map(rowSet -> {
                List<DailySalesWithRunningTotal> results = new ArrayList<>();
                rowSet.forEach(row -> results.add(mapRowToDailySales(row)));
                return results;
            });
    }).toCompletionStage().toCompletableFuture();
}

// Product performance ranking
public CompletableFuture<List<ProductRanking>> getProductRankings() {
    ConnectionProvider cp = databaseService.getConnectionProvider();
    return cp.withConnection("peegeeq-main", connection -> {
        String sql = """
            SELECT
                p.id, p.name,
                COUNT(oi.id) as times_ordered,
                SUM(oi.quantity) as total_quantity,
                SUM(oi.quantity * oi.price) as total_revenue,
                RANK() OVER (ORDER BY SUM(oi.quantity * oi.price) DESC) as revenue_rank
            FROM products p
            LEFT JOIN order_items oi ON p.id = oi.product_id
            GROUP BY p.id, p.name
            ORDER BY revenue_rank
            """;
        return connection.preparedQuery(sql)
            .execute()
            .map(rowSet -> {
                List<ProductRanking> results = new ArrayList<>();
                rowSet.forEach(row -> results.add(mapRowToProductRanking(row)));
                return results;
            });
    });
}

```

```

        }).toCompletionStage().toCompletableFuture());
    }
}

```

REST Endpoints:

```

@RestController
@RequestMapping("/api/reports")
public class ReportingController {

    @GetMapping("/sales/monthly")
    public CompletableFuture<ResponseEntity<List<MonthlySales>>> getMonthlySales(
        @RequestParam @DateTimeFormat(iso = DateTimeFormat.ISO.DATE) LocalDate startDate,
        @RequestParam @DateTimeFormat(iso = DateTimeFormat.ISO.DATE) LocalDate endDate) {
        return reportingService.getSalesByMonth(startDate, endDate)
            .thenApply(ResponseEntity::ok);
    }

    @GetMapping("/customers/top")
    public CompletableFuture<ResponseEntity<List<CustomerStats>>> getTopCustomers(
        @RequestParam(defaultValue = "10") int limit) {
        return reportingService.getTopCustomers(limit)
            .thenApply(ResponseEntity::ok);
    }









    @GetMapping("/orders/{id}/details")
    public CompletableFuture<ResponseEntity<OrderWithDetails>> getOrderDetails(
        @PathVariable String id) {
        return reportingService.getOrderDetails(id)
            .thenApply(ResponseEntity::ok);
    }

    @GetMapping("/sales/daily-running-total")
    public CompletableFuture<ResponseEntity<List<DailySalesWithRunningTotal>>> getDailyRunningTotal(
        @RequestParam @DateTimeFormat(iso = DateTimeFormat.ISO.DATE) LocalDate startDate,
        @RequestParam @DateTimeFormat(iso = DateTimeFormat.ISO.DATE) LocalDate endDate) {
        return reportingService.getDailySalesWithRunningTotal(startDate, endDate)
            .thenApply(ResponseEntity::ok);
    }

    @GetMapping("/products/rankings")
    public CompletableFuture<ResponseEntity<List<ProductRanking>>> getProductRankings() {
        return reportingService.getProductRankings()
            .thenApply(ResponseEntity::ok);
    }
}

```

Key Points:

-  Complex SQL with aggregations (SUM, AVG, COUNT, GROUP BY)
-  Common Table Expressions (CTEs) for complex queries
-  Window functions (RANK, SUM OVER) for analytics
-  Multi-table JOINS for comprehensive reports
-  Date range queries with proper indexing
-  Read-only operations use `withConnection()` (no transaction overhead)
-  Efficient result mapping to DTOs
-  No JPA/Hibernate needed for reporting

Example 2B: Search Service (Reactive)

Purpose: Full-text search and dynamic filtering

Components:

- SearchCriteria.java - Filter parameters
- SearchResult.java - Paginated results
- SearchRepository.java - Dynamic query builder
- SearchService.java - Search logic

Operations:

```
@Service
public class SearchService {

    private final DatabaseService databaseService;
    private final ReactiveOutboxAdapter adapter;

    // Full-text search with PostgreSQL tsvector
    public Mono<SearchResult<Product>> searchProducts(String query, int page, int size) {
        ConnectionProvider cp = databaseService.getConnectionProvider();
        return adapter.toMono(
            cp.withConnection("peegeeq-main", connection -> {
                String sql = """
                    SELECT *,
                        ts_rank(search_vector, plainto_tsquery('english', $1)) as rank
                    FROM products
                    WHERE search_vector @@ plainto_tsquery('english', $1)
                    ORDER BY rank DESC
                    LIMIT $2 OFFSET $3
                    """;
                return connection.preparedQuery(sql)
                    .execute(Tuple.of(query, size, page * size))
                    .compose(rowSet -> {
                        List<Product> products = new ArrayList<>();
                        rowSet.forEach(row -> products.add(mapRowToProduct(row)));

                        // Get total count
                        String countSql = """
                            SELECT COUNT(*)
                            FROM products
                            WHERE search_vector @@ plainto_tsquery('english', $1)
                            """;
                        return connection.preparedQuery(countSql)
                            .execute(Tuple.of(query))
                            .map(countRowSet -> {
                                long total = countRowSet.iterator().next().getLong(0);
                                return new SearchResult<>(products, total, page, size);
                            });
                    });
            })
        ).toCompletionStage().toCompletableFuture()
    );
}

// Dynamic filtering with multiple criteria
public Mono<List<Product>> filterProducts(SearchCriteria criteria) {
    ConnectionProvider cp = databaseService.getConnectionProvider();
    return adapter.toMono(
        cp.withConnection("peegeeq-main", connection -> {
            // Build dynamic query based on criteria
            StringBuilder sql = new StringBuilder("SELECT * FROM products WHERE 1=1");
        })
    );
}
```

```

List<Object> params = new ArrayList<>();
int paramIndex = 1;

if (criteria.getCategoryId() != null) {
    sql.append(" AND category_id = $").append(paramIndex++);
    params.add(criteria.getCategoryId());
}

if (criteria.getMinPrice() != null) {
    sql.append(" AND price >= $").append(paramIndex++);
    params.add(criteria.getMinPrice());
}

if (criteria.getMaxPrice() != null) {
    sql.append(" AND price <= $").append(paramIndex++);
    params.add(criteria.getMaxPrice());
}

if (criteria.getActive() != null) {
    sql.append(" AND active = $").append(paramIndex++);
    params.add(criteria.getActive());
}

if (criteria.getNamePattern() != null) {
    sql.append(" AND name ILIKE $").append(paramIndex++);
    params.add("%" + criteria.getNamePattern() + "%");
}

sql.append(" ORDER BY ").append(criteria.getSortBy())
    .append(" ").append(criteria.getSortDirection());
sql.append(" LIMIT $").append(paramIndex++);
params.add(criteria.getLimit());

return connection.preparedQuery(sql.toString())
    .execute(Tuple.from(params))
    .map(rowSet -> {
        List<Product> products = new ArrayList<>();
        rowSet.forEach(row -> products.add(mapRowToProduct(row)));
        return products;
    });
}).toCompletionStage().toCompletableFuture()
);
}

// Autocomplete suggestions
public Flux<String> getAutocompleteSuggestions(String prefix, int limit) {
    ConnectionProvider cp = databaseService.getConnectionProvider();
    return Flux.from(
        adapter.toMono(
            cp.withConnection("peegeeq-main", connection -> {
                String sql = """
                    SELECT DISTINCT name
                    FROM products
                    WHERE name ILIKE $1
                    ORDER BY name
                    LIMIT $2
                    """;
                return connection.preparedQuery(sql)
                    .execute(Tuple.of(prefix + "%", limit))
                    .map(rowSet -> {
                        List<String> suggestions = new ArrayList<>();
                        rowSet.forEach(row -> suggestions.add(row.getString("name")));
                        return suggestions;
                    });
            })
        ).toCompletionStage().toCompletableFuture()
    ).flatMapMany(Flux::fromIterable)
}

```

```

    });
}

// Faceted search (aggregations for filters)
public Mono<SearchFacets> getSearchFacets() {
    ConnectionProvider cp = databaseService.getConnectionProvider();
    return adapter.toMono(
        cp.withConnection("peegeeq-main", connection -> {
            String sql = ""
                SELECT
                    c.id as category_id,
                    c.name as category_name,
                    COUNT(p.id) as product_count,
                    MIN(p.price) as min_price,
                    MAX(p.price) as max_price
                FROM categories c
                LEFT JOIN products p ON c.id = p.category_id
                WHERE p.active = true
                GROUP BY c.id, c.name
                ORDER BY c.name
            """;
            return connection.preparedQuery(sql)
                .execute()
                .map(rowSet -> {
                    SearchFacets facets = new SearchFacets();
                    rowSet.forEach(row -> facets.addCategoryFacet(
                        row.getString("category_id"),
                        row.getString("category_name"),
                        row.getLong("product_count"),
                        row.getBigDecimal("min_price"),
                        row.getBigDecimal("max_price")
                    ));
                    return facets;
                });
        }).toCompletionStage().toCompletableFuture()
    );
}
}

```

Reactive Controller:

```

@RestController
@RequestMapping("/api/search")
public class SearchController {

    @GetMapping("/products")
    public Mono<ResponseEntity<SearchResult<Product>>> searchProducts(
        @RequestParam String q,
        @RequestParam(defaultValue = "0") int page,
        @RequestParam(defaultValue = "20") int size) {
        return searchService.searchProducts(q, page, size)
            .map(ResponseEntity::ok);
    }

    @PostMapping("/products/filter")
    public Mono<ResponseEntity<List<Product>>> filterProducts(
        @RequestBody SearchCriteria criteria) {
        return searchService.filterProducts(criteria)
            .map(ResponseEntity::ok);
    }

    @GetMapping("/autocomplete")
    public Flux<String> autocomplete(@RequestParam String prefix) {

```








```

        return searchService.getAutocompleteSuggestions(prefix, 10);
    }

    @GetMapping("/facets")
    public Mono<ResponseEntity<SearchFacets>> getFacets() {
        return searchService.getSearchFacets()
            .map(ResponseEntity::ok);
    }
}

```

Key Points:

-  Full-text search with PostgreSQL `tsvector` and `tsquery`
-  Dynamic query building based on filter criteria
-  Autocomplete with prefix matching
-  Faceted search for filter aggregations
-  Pagination with total count
-  Ranking results by relevance
-  No need for Elasticsearch for basic search

Category 3: Batch Operations and Data Import

These examples show bulk operations and data processing without messaging.

Example 3A: Bulk Data Import (Non-Reactive)

Purpose: Efficient batch operations for data import/export

Operations:

```

@Service
public class DataImportService {

    private final DatabaseService databaseService;

    // Batch insert with transaction
    public CompletableFuture<ImportResult> importCustomers(List<Customer> customers) {
        ConnectionProvider cp = databaseService.getConnectionProvider();
        return cp.withTransaction("peegeeq-main", connection -> {
            String sql = "INSERT INTO customers (id, name, email, created_at) VALUES ($1, $2, $3, $4)";

            // Create batch of tuples
            List<Tuple> batch = customers.stream()
                .map(c -> Tuple.of(c.getId(), c.getName(), c.getEmail(), c.getCreatedAt()))
                .collect(Collectors.toList());

            return connection.preparedQuery(sql)
                .executeBatch(batch)
                .map(result -> new ImportResult(customers.size(), 0));
        }).toCompletionStage().toCompletableFuture();
    }

    // Upsert (INSERT ... ON CONFLICT UPDATE)
    public CompletableFuture<Integer> upsertProducts(List<Product> products) {
        ConnectionProvider cp = databaseService.getConnectionProvider();
        return cp.withTransaction("peegeeq-main", connection -> {
            String sql = ""

```

```

        INSERT INTO products (id, name, category_id, price, updated_at)
        VALUES ($1, $2, $3, $4, $5)
        ON CONFLICT (id) DO UPDATE SET
            name = EXCLUDED.name,
            category_id = EXCLUDED.category_id,
            price = EXCLUDED.price,
            updated_at = EXCLUDED.updated_at
        """;

    List<Tuple> batch = products.stream()
        .map(p -> Tuple.of(p.getId(), p.getName(), p.getCategoryId(),
            p.getPrice(), Instant.now()))
        .collect(Collectors.toList());

    return connection.preparedQuery(sql)
        .executeBatch(batch)
        .map(result -> products.size());
    }).toCompletionStage().toCompletableFuture();
}

// Bulk update with WHERE IN
public CompletableFuture<Integer> deactivateProducts(List<String> productIds) {
    ConnectionProvider cp = databaseService.getConnectionProvider();
    return cp.withTransaction("peegeeq-main", connection -> {
        // Build dynamic IN clause
        String placeholders = IntStream.range(1, productIds.size() + 1)
            .mapToObj(i -> "$" + i)
            .collect(Collectors.joining(", "));

        String sql = "UPDATE products SET active = false WHERE id IN (" + placeholders + ")";

        return connection.preparedQuery(sql)
            .execute(Tuple.from(productIds))
            .map(result -> result.rowCount());
    }).toCompletionStage().toCompletableFuture();
}

// Bulk delete with transaction
public CompletableFuture<Integer> deleteOrders(List<String> orderIds) {
    ConnectionProvider cp = databaseService.getConnectionProvider();
    return cp.withTransaction("peegeeq-main", connection -> {
        // Delete order items first (foreign key constraint)
        String deleteItemsSql = "DELETE FROM order_items WHERE order_id = ANY($1)";

        return connection.preparedQuery(deleteItemsSql)
            .execute(Tuple.of(orderIds.toArray(new String[0])))
            .compose(itemsResult -> {
                // Then delete orders
                String deleteOrdersSql = "DELETE FROM orders WHERE id = ANY($1)";
                return connection.preparedQuery(deleteOrdersSql)
                    .execute(Tuple.of(orderIds.toArray(new String[0])))
                    .map(ordersResult -> ordersResult.rowCount());
            });
    }).toCompletionStage().toCompletableFuture();
}

// Export data to CSV format
public CompletableFuture<String> exportCustomersToCSV() {
    ConnectionProvider cp = databaseService.getConnectionProvider();
    return cp.withConnection("peegeeq-main", connection -> {
        String sql = "SELECT id, name, email, created_at FROM customers ORDER BY created_at";

        return connection.preparedQuery(sql)
            .execute()
            .map(rowSet -> {
                StringBuilder csv = new StringBuilder();

```

```







        csv.append("id,name,email,created_at\n");

        rowSet.forEach(row -> {
            csv.append(row.getString("id")).append(",")
                .append(row.getString("name")).append(",")
                .append(row.getString("email")).append(",")
                .append(row.getLocalDateTime("created_at")).append("\n");
        });

        return csv.toString();
    });
}).toCompletionStage().toCompletableFuture();
}
}

```

Key Points:

-  Batch operations with `executeBatch()` for performance
-  UPSERT with `ON CONFLICT DO UPDATE`
-  Bulk updates with `WHERE IN` or `ANY()`
-  Cascading deletes with proper transaction handling
-  Data export to various formats
-  All operations use transactions for consistency

Category 4: Complex Business Logic

These examples show multi-step business operations with proper transaction management.

Example 4A: Order Fulfillment Workflow (Non-Reactive)

Purpose: Multi-step business process with rollback on failure

Operations:

```

@Service
public class OrderFulfillmentService {

    private final DatabaseService databaseService;

    // Complete order fulfillment workflow
    public CompletableFuture<FulfillmentResult> fulfillOrder(String orderId) {
        ConnectionProvider cp = databaseService.getConnectionProvider();

        return cp.withTransaction("peggeeq-main", connection -> {
            // Step 1: Validate order exists and is pending
            String validateSql = "SELECT * FROM orders WHERE id = $1 AND status = 'PENDING' FOR UPDATE";

            return connection.preparedQuery(validateSql)
                .execute(Tuple.of(orderId))
                .compose(orderRowSet -> {
                    if (orderRowSet.size() == 0) {
                        return Future.failedFuture(new IllegalStateException("Order not found or not pending"));
                    }
                });

            // Step 2: Check inventory for all items
            String inventorySql = ""
                + "SELECT oi.product_id, oi.quantity, i.available_quantity"
                + "FROM order_items oi";

```

```

        JOIN inventory i ON oi.product_id = i.product_id
        WHERE oi.order_id = $1
        """;

return connection.preparedQuery(inventorySql)
    .execute(Tuple.of(orderId))
    .compose(inventoryRowSet -> {
        // Validate sufficient inventory
        for (Row row : inventoryRowSet) {
            int required = row.getInteger("quantity");
            int available = row.getInteger("available_quantity");
            if (available < required) {
                return Future.failedFuture(new IllegalStateException(
                    "Insufficient inventory for product: " + row.getString("product_id")
                ));
            }
        }
    })

    // Step 3: Reserve inventory
    String reserveSql = """
        UPDATE inventory i
        SET available_quantity = available_quantity - oi.quantity,
            reserved_quantity = reserved_quantity + oi.quantity
        FROM order_items oi
        WHERE i.product_id = oi.product_id AND oi.order_id = $1
        """;

return connection.preparedQuery(reserveSql)
    .execute(Tuple.of(orderId))
    .compose(reserveResult -> {
        // Step 4: Update order status
        String updateOrderSql = """
            UPDATE orders
            SET status = 'FULFILLED', fulfilled_at = $1
            WHERE id = $2
            """;

        return connection.preparedQuery(updateOrderSql)
            .execute(Tuple.of(Instant.now(), orderId))
            .compose(updateResult -> {
                // Step 5: Create fulfillment record
                String fulfillmentSql = """
                    INSERT INTO fulfillments (id, order_id, fulfilled_at, status)
                    VALUES ($1, $2, $3, 'COMPLETED')
                    """;

                String fulfillmentId = UUID.randomUUID().toString();
                return connection.preparedQuery(fulfillmentSql)
                    .execute(Tuple.of(fulfillmentId, orderId, Instant.now()))
                    .map(fulfillmentResult ->
                        new FulfillmentResult(orderId, fulfillmentId, "SUCCESS")
                    );
            });
    });

    });
    }).toCompletionStage().toCompletableFuture();
}

// Cancel order and restore inventory
public CompletableFuture<Void> cancelOrder(String orderId) {
    ConnectionProvider cp = databaseService.getConnectionProvider();

    return cp.withTransaction("peegeeq-main", connection -> {
        // Step 1: Validate order can be cancelled
        String validateSql = """

```

```

        SELECT * FROM orders
        WHERE id = $1 AND status IN ('PENDING', 'FULFILLED')
        FOR UPDATE
        """;

    return connection.preparedQuery(validateSql)
        .execute(Tuple.of(orderId))
        .compose(orderRowSet -> {
            if (orderRowSet.size() == 0) {
                return Future.failedFuture(new IllegalStateException("Order cannot be cancelled"));
            }

            String status = orderRowSet.iterator().next().getString("status");

            // Step 2: Restore inventory if order was fulfilled
            if ("FULFILLED".equals(status)) {
                String restoreSql = """
                    UPDATE inventory i
                    SET available_quantity = available_quantity + oi.quantity,
                        reserved_quantity = reserved_quantity - oi.quantity
                    FROM order_items oi
                    WHERE i.product_id = oi.product_id AND oi.order_id = $1
                    """;

                return connection.preparedQuery(restoreSql)
                    .execute(Tuple.of(orderId))
                    .compose(restoreResult -> {
                        // Step 3: Update order status
                        String updateSql = """
                            UPDATE orders
                            SET status = 'CANCELLED', cancelled_at = $1
                            WHERE id = $2
                            """;

                        return connection.preparedQuery(updateSql)
                            .execute(Tuple.of(Instant.now(), orderId))
                            .mapEmpty();
                    });
            } else {
                // Just update status for pending orders
                String updateSql = """
                    UPDATE orders
                    SET status = 'CANCELLED', cancelled_at = $1
                    WHERE id = $2
                    """;

                return connection.preparedQuery(updateSql)
                    .execute(Tuple.of(Instant.now(), orderId))
                    .mapEmpty();
            }
        });
    }).toCompletionStage().toCompletableFuture();
}

```

Key Points:

- ☒ Multi-step business logic in single transaction
- ☒ Validation at each step with proper error handling
- ☒ FOR UPDATE for row-level locking
- ☒ Automatic rollback on any failure
- ☒ Complex state transitions
- ☒ Inventory management with reservations
- ☒ All-or-nothing consistency

Summary: When to Use Each Pattern

Use Case	Pattern	Transaction Type	Example
Simple CRUD	<code>withConnection()</code>	Auto-commit	Customer management
Multi-step updates	<code>withTransaction()</code>	Explicit transaction	Order fulfillment
Read-only queries	<code>withConnection()</code>	No transaction needed	Reports, search
Batch operations	<code>withTransaction()</code> + <code>executeBatch()</code>	Bulk transaction	Data import
Complex business logic	<code>withTransaction()</code> + composition	Multi-step transaction	Order workflow
Reactive operations	Wrap in <code>Mono</code> / <code>Flux</code>	Same as above	Product catalog

Key Takeaway: PeeGeeQ's `DatabaseService` and `ConnectionProvider` handle **all database operations** - you don't need R2DBC, JPA, or any other data access framework. The outbox pattern is **optional** and only needed when you want transactional messaging.

Common Mistakes

❌ Mistake 1: Using R2DBC

Problem:

```
// ❌ WRONG - R2DBC creates separate connection pool
@Repository
public interface OrderRepository extends R2dbcRepository<Order, String> {}

@Service
public class OrderService {
    public Mono<String> createOrder(CreateOrderRequest request) {
        // These use DIFFERENT connection pools - NO transaction consistency!
        return orderRepository.save(new Order(request))
            .flatMap(order -> Mono.fromFuture(
                producer.send(new OrderCreatedEvent(request))
            ));
    }
}
```

Solution:

```
// ✅ CORRECT - Use PeeGeeQ's ConnectionProvider
@Repository
public class OrderRepository {
    public Future<Order> save(Order order, SqlConnection connection) {
```

```

        // Uses PeeGeeQ's connection
    }
}

@Service
public class OrderService {
    public CompletableFuture<String> createOrder(CreateOrderRequest request) {
        return connectionProvider.withTransaction("client-id", connection -> {
            // All operations use SAME connection
            return orderRepository.save(order, connection)
                .compose(v -> Future.fromCompletionStage(
                    producer.sendInTransaction(event, connection)
                ));
        }).toCompletionStage().toCompletableFuture();
    }
}

```

✗ Mistake 2: Using sendWithTransaction()

Problem:

```

// ✗ WRONG - Creates separate transaction
producer.sendWithTransaction(event, TransactionPropagation.CONTEXT)
    .thenCompose(v -> orderRepository.save(order));

```

Solution:

```

// ✓ CORRECT - Joins existing transaction
connectionProvider.withTransaction("client-id", connection -> {
    return Future.fromCompletionStage(
        producer.sendInTransaction(event, connection)
    )
    .compose(v -> orderRepository.save(order, connection));
});

```

✗ Mistake 3: Creating Separate Pool

Problem:

```

// ✗ WRONG - Creating separate pool
@Bean
public Pool vertxPool(PeeGeeQManager manager) {
    return manager.getClientFactory()
        .getConnectionManager()
        .getOrCreateReactivePool("my-pool", ...);
}

```

Solution:

```

// ✓ CORRECT - Use DatabaseService
@Bean
public DatabaseService databaseService(PeeGeeQManager manager) {
    return new PgDatabaseService(manager);
}

```

```
}
```

✗ Mistake 4: Using Pool Directly

Problem:

```
// ✗ WRONG - Using Pool directly
@Service
public class OrderService {
    private final Pool pool; // Internal implementation class

    public CompletableFuture<String> createOrder(CreateOrderRequest request) {
        return pool.withTransaction(connection -> {
            // Bypasses PeeGeeQ's API layer
        }).toCompletionStage().toCompletableFuture();
    }
}
```

Solution:

```
// ✓ CORRECT - Using ConnectionProvider
@Service
public class OrderService {
    private final DatabaseService databaseService; // Public API

    public CompletableFuture<String> createOrder(CreateOrderRequest request) {
        ConnectionProvider cp = databaseService.getConnectionProvider();
        return cp.withTransaction("client-id", connection -> {
            // Uses public API
        }).toCompletionStage().toCompletableFuture();
    }
}
```

✗ Mistake 5: Model Classes with R2DBC Annotations

Problem:

```
// ✗ WRONG - R2DBC annotations
@Table("orders")
public class Order {
    @Id
    private String id;

    @Column("customer_id")
    private String customerId;
}
```

Solution:

```
// ✓ CORRECT - Plain POJO
public class Order {
    private String id;
    private String customerId;
}
```

```

    // No annotations needed - manual SQL mapping in repository
}

```

Testing

Test Configuration

```

@SpringBootTest
@TestPropertySource(properties = {
    "spring.profiles.active=test",
    // Exclude R2DBC auto-configuration (if Spring Boot tries to configure it)
    "spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.r2dbc.R2dbcAutoConfiguration"
})
public class OrderServiceTest {

    @Autowired
    private OrderService orderService;

    @Autowired
    private DatabaseService databaseService;

    @BeforeEach
    public void setup() {
        // Clean database before each test
        ConnectionProvider cp = databaseService.getConnectionProvider();
        cp.withConnection("peegeeq-main", connection ->
            connection.query("TRUNCATE TABLE orders, order_items CASCADE")
                .execute()
                .mapEmpty()
        ).toCompletionStage().toCompletableFuture().join();
    }
}

```

Test Transactional Rollback

```

@Test
public void testBusinessValidationRollback() {
    CreateOrderRequest request = new CreateOrderRequest();
    request.setCustomerId("INVALID_CUSTOMER");
    request.setAmount(new BigDecimal("15000")); // Exceeds limit

    // Should throw exception and rollback
    assertThrows(CompletionException.class, () -> {
        orderService.createOrderWithValidation(request)
            .join();
    });

    // Verify nothing was saved (transaction rolled back)
    ConnectionProvider cp = databaseService.getConnectionProvider();
    Long count = cp.withConnection("peegeeq-main", connection ->
        connection.query("SELECT COUNT(*) FROM orders")
            .execute()
            .map(rowSet -> rowSet.iterator().next().getLong(0))
    ).toCompletionStage().toCompletableFuture().join();

    assertEquals(0L, count, "Order should not exist after rollback");
}

```

```
}
```

Quick Reference

When to Use Each Method

Method	Use Case	Transaction Behavior
withConnection()	Single read/write operation	Auto-commits
withTransaction()	Multiple operations needing consistency	Explicit transaction
sendInTransaction()	Outbox event in existing transaction	Uses provided connection
append() with connection	Event store in existing transaction	Uses provided connection

Correct Dependency Injection

```
@Service
public class MyService {
    // ✅ Inject DatabaseService
    private final DatabaseService databaseService;

    // ✅ Inject MessageProducer (not OutboxProducer)
    private final MessageProducer<MyEvent> producer;

    // ✅ Inject EventStore
    private final EventStore<MyEvent> eventStore;

    // ✅ Inject your repositories
    private final MyRepository repository;

    // ❌ DON'T inject Pool, PgConnectionManager, etc.
}
```

Correct Repository Signature

```
@Repository
public class MyRepository {
    // ✅ CORRECT - Accepts SqlConnection
    public Future<MyEntity> save(MyEntity entity, SqlConnection connection) { ... }

    // ✅ CORRECT - Accepts SqlConnection
    public Future<Optional<MyEntity>> findById(String id, SqlConnection connection) { ... }

    // ❌ WRONG - No connection parameter
    public Future<MyEntity> save(MyEntity entity) { ... }
}
```

Common Mistakes Summary

Mistake	Problem	Solution
Multiple Connection Pools	Creating R2DBC, JDBC, or separate Vert.x pools	Use only PeeGeeQ's ConnectionProvider
Separate Transactions	Each operation creates its own transaction	Use withTransaction() and pass connection
Not Passing Connection	Repository methods don't accept SqlConnection	Always pass connection from service layer
Using R2DBC Repositories	Spring Data R2DBC uses separate connection pool	Write repositories using Vert.x SQL Client
Using @Transactional	Spring's @Transactional is for JDBC/R2DBC	Use ConnectionProvider.withTransaction()
Improper Consumer Lifecycle	Consumers not started/stopped properly	Use @PostConstruct to start, @PreDestroy to stop
Mixing Blocking and Reactive	Using blocking JDBC calls inside PeeGeeQ transactions	Use only Vert.x SQL Client (reactive)

Summary Checklist

Configuration

- ☐ Create `PeeGeeQManager` bean
- ☐ Create `DatabaseService` bean (not `Pool`)
- ☐ Create `QueueFactory` bean
- ☐ Create `MessageProducer` beans (returned from `QueueFactory`)
- ☐ Initialize schema using `ConnectionProvider`
- ☐ Avoid circular dependencies (use `ApplicationContext`)

Dependencies

- ☐ Include `peegee-outbox` dependency
- ☐ Include Spring Boot starter (web or webflux)
- ☐ **DO NOT** include R2DBC dependencies
- ☐ **DO NOT** include separate Vert.x dependencies

Repository Layer

- ☐ Use `SqlConnection` parameter in all methods
- ☐ Return Vert.x `Future` (not `Mono` or `Flux`)
- ☐ Use manual SQL mapping (no annotations)
- ☐ **DO NOT** extend R2DBC repositories

Service Layer

- ☐ Inject `DatabaseService` (not `Pool`)
- ☐ Get `ConnectionProvider` from `DatabaseService`
- ☐ Use `ConnectionProvider.withTransaction()`
- ☐ Pass `connection` to all repository methods
- ☐ Use `sendInTransaction(event, connection)`
- ☐ **DO NOT** use `sendWithTransaction()`

Reactive Applications

- ☐ Create `ReactiveOutboxAdapter` component
- ☐ Wrap `CompletableFuture` in `Mono`
- ☐ Return `Mono` from service methods
- ☐ **DO NOT** use `R2DBC`

Testing

- ☐ Exclude `R2DBC` auto-configuration
- ☐ Test transactional rollback scenarios
- ☐ Verify database state after rollback
- ☐ Clean database between tests

Additional Resources

- **Working Examples:** See `peegeeq-examples/src/main/java/dev/mars/peegeeq/examples/springboot`
- **Reactive Example:** See `peegeeq-examples/src/main/java/dev/mars/peegeeq/examples/springboot2`
- **Test Examples:** See `peegeeq-examples/src/test/java/dev/mars/peegeeq/examples/springboot`

Questions?

If you encounter issues:

1. Check that you're using `DatabaseService` (not `Pool`)
2. Verify all operations use the same `SqlConnection`
3. Ensure you're using `sendInTransaction()` (not `sendWithTransaction()`)
4. Confirm `R2DBC` dependencies are removed
5. Review the working examples in `peegeeq-examples`

Remember: PeeGeeQ provides complete database infrastructure. Your Spring Boot application should host it, not create parallel infrastructure.