Batch Processing Framework

Functional Specification Document

Document Version: 1.0

Date: November 29, 2025

Project: Batch Processing Framework for DTO Persistence

Technology Stack: Java 17, Jakarta Persistence 3.1, JUnit 5, Mockito 5

1. Executive Summary

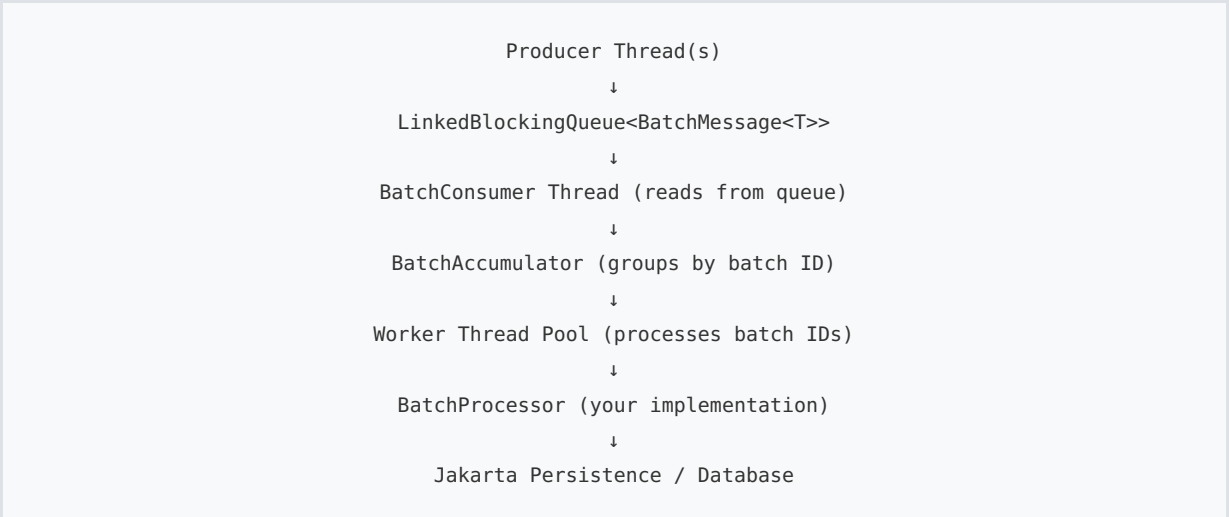
This document specifies the functional requirements for a multi-threaded batch processing framework designed to consume Data Transfer Objects (DTOs) from a producer, group them by batch ID, split them into configurable sub-batches, and persist them to a database using Jakarta Persistence with transactional integrity and failure recovery.

Key Objectives:

- Process high-volume DTO streams efficiently
- Maintain transactional consistency with configurable batch sizes
- Handle failures gracefully with automatic cleanup
- Support parallel processing of multiple batch IDs
- Ensure graceful shutdown without data loss

2. System Architecture

2.1 High-Level Architecture



2.2 Component Overview

Component	Responsibility	Threading
LinkedBlockingQueue	Thread-safe message buffer between producer and consumer	Multi-producer, single-consumer
BatchConsumer	Drains queue, accumulates messages by batch ID	1 dedicated thread (configurable)
BatchAccumulator	Groups messages by batch ID, splits into sub-batches	Thread-safe (ConcurrentHashMap)
BatchWorker	Processes all sub-batches for a single batch ID	Thread pool (configurable size)
BatchProcessor	Persists a single sub-batch to database	Called by BatchWorker
CleanupStrategy	Removes partial writes on failure	Called by BatchWorker

3. Functional Requirements

3.1 Message Structure

Each message in the system is wrapped in a `BatchMessage<T>` containing:

Field	Type	Description
batchId	String	Unique identifier for the logical batch (required, not null)
payload	T (generic DTO)	The actual data object to persist (null for end-of-batch messages)
endOfBatch	boolean	Flag indicating this is the final message for the batch ID

3.2 Batch Processing Flow

3.2.1 Message Production

1. Producer wraps each DTO in a `BatchMessage.data(batchId, dto)`
2. Producer adds message to `LinkedBlockingQueue`
3. After all DTOs for a batch ID are queued, producer sends `BatchMessage.endOfBatch(batchId)`

```
// Producer example String batchId = "ORDER-2025-001"; for (OrderDTO order : orders) { queue.put(BatchMessage.data(batchId, order)); } queue.put(BatchMessage.endOfBatch(batchId)); // Required!
```

3.2.2 Message Consumption

1. BatchConsumer polls queue with configurable timeout (default: 1 second)
2. Data messages are added to BatchAccumulator keyed by batch ID
3. End-of-batch messages mark the batch ID as "complete"
4. Complete batch IDs are submitted to worker thread pool

3.2.3 Batch Accumulation and Splitting

Splitting Logic:

- Configuration parameter: `maxBatchSize` (default: 5000)
- When a batch ID is marked complete, calculate sub-batches:

```
totalItems = accumulated messages for batch ID subBatchCount = ceiling(totalItems / maxBatchSize)
Example 1: 100 items, max 5000 → 1 sub-batch (100 items) Example 2: 10,000 items, max 5000 → 2
sub-batches (5000 + 5000) Example 3: 7,500 items, max 5000 → 2 sub-batches (5000 + 2500)
```

Each sub-batch receives:

- The same batch ID
- A sequence number (1, 2, 3, ...)
- A list of DTO items

3.2.4 Batch Processing

Step	Action	Thread
1	BatchWorker created for batch ID	Main coordinator thread
2	Retrieve all sub-batches from accumulator	Worker thread
3	Call processor.beforeBatchId(batchId)	Worker thread
4	Process sub-batch 1 sequentially	Worker thread
5	If success, process sub-batch 2, etc.	Worker thread
6	If failure, trigger cleanup and stop	Worker thread
7	Call processor.afterBatchId(batchId)	Worker thread

Success Path: All sub-batches for batch ID complete successfully → processor.afterBatchId() called → cleanupStrategy.markComplete() called

Failure Path: Sub-batch N fails → remaining sub-batches purged → cleanupStrategy.cleanup(batchId, N) called → continue with next batch ID

4. Failure Handling Specification

4.1 Failure Scenarios

Scenario	System Response	Data State
Sub-batch persistence fails	Catch exception, purge remaining sub-batches, trigger cleanup	Partial writes removed by CleanupStrategy
Database	BatchProcessor throws exception, cleanup	Transaction rolled back, cleanup removes

connection lost	triggered	any committed data
Unexpected runtime error	Catch in BatchWorker, log error, trigger cleanup	Cleanup called with sequence = -1
Thread interruption	Graceful shutdown, drain queue, complete in-flight batches	No data loss

4.2 Cleanup Requirements

The `CleanupStrategy.cleanup(String batchId, int failedSequenceNumber)` implementation must:

1. Delete ALL persisted data for the given batch ID (all sequences)
2. Handle partial transactions (some sub-batches may be committed)
3. Be idempotent (safe to call multiple times)
4. Complete within reasonable time (synchronous operation)

```
// Cleanup implementation requirement public void cleanup(String batchId, int failedSeq) { EntityTransaction tx = em.getTransaction(); tx.begin(); try { // Delete ALL entities for this batch ID em.createQuery("DELETE FROM YourEntity WHERE batchId = :id") .setParameter("id", batchId) .executeUpdate(); tx.commit(); } catch (Exception e) { if (tx.isActive()) tx.rollback(); throw new RuntimeException("Cleanup failed", e); } }
```

4.3 Error Recovery

After cleanup, the framework:

- Logs the failure with batch ID and sequence number
- Continues processing the next batch ID (does NOT halt)
- Removes failed batch ID from accumulator
- Does NOT retry (retry logic is producer's responsibility)

5. Threading Model

5.1 Thread Pools

Pool	Size	Purpose	Configuration
Consumer Pool	1 (typical)	Read from queue, build batches	Builder.consumerThreads()
Worker Pool	4 (default)	Process batch IDs in parallel	Builder.workerThreads()

5.2 Concurrency Guarantees

- **Thread Safety:** All shared data structures use concurrent collections
- **Isolation:** Each batch ID processed by exactly one worker at a time
- **Ordering:** Sub-batches for same batch ID processed sequentially
- **Parallelism:** Different batch IDs can process simultaneously

5.3 Synchronization Points

Operation	Synchronization	Reason
Add message to accumulator	Synchronized method	Prevent race conditions on batch lists
Retrieve and remove batch	Synchronized method	Atomic read-then-remove operation
Mark batch complete	ConcurrentHashMap.newKeySet()	Thread-safe set operations
Queue operations	LinkedBlockingQueue (internal)	Built-in thread safety

6. Configuration Specification

6.1 Required Configuration

Parameter	Type	Required	Description
queue	BlockingQueue<BatchMessage<T>>	Yes	Message queue (typically LinkedBlockingQueue)
processor	BatchProcessor<T>	Yes	Persistence implementation
maxBatchSize	int	No	Maximum items per sub-batch (default: 5000)
cleanupStrategy	CleanupStrategy	No	Cleanup implementation (default: NoOpCleanupStrategy)
consumerThreads	int	No	Consumer pool size (default: 1)
workerThreads	int	No	Worker pool size (default: 4)

6.2 Configuration Example

```
BatchCoordinator<OrderDTO> coordinator = new BatchCoordinator.Builder<OrderDTO>() .queue(new LinkedBlockingQueue<>(10000)) .processor(new OrderBatchProcessor(entityManager)) .cleanupStrategy(new OrderCleanupStrategy(entityManager)) .maxBatchSize(5000) .consumerThreads(1) .workerThreads(8) .build(); coordinator.start();
```

7. Interface Specifications

7.1 BatchProcessor Interface

```
public interface BatchProcessor<T> { /** * Process a single sub-batch within a transaction. * Must be idempotent if possible. * * @param batch The sub-batch to persist * @throws BatchProcessingException if persistence fails */ void process(Batch<T> batch) throws BatchProcessingException; /** * Called before processing first sub-batch of a batch ID. * Optional hook for initialization. */ default void beforeBatchId(String batchId) {} /** * Called after all sub-batches successfully complete. * Optional hook for finalization. */ default void afterBatchId(String batchId) {} }
```

7.2 CleanupStrategy Interface

```
public interface CleanupStrategy { /** * Remove partial writes for a failed batch ID. * * @param batchId The batch ID that failed * @param failedSequenceNumber Which sub-batch failed (or -1 for unexpected errors) */ void cleanup(String batchId, int failedSequenceNumber); /** * Mark batch ID as successfully completed (optional hook). */ default void markComplete(String batchId) {} }
```

7.3 Transaction Handling Requirements

The `BatchProcessor.process()` implementation must:

- Begin a new transaction for each sub-batch
- Persist all items in the sub-batch
- Flush changes to database
- Commit transaction on success
- Rollback transaction on any exception
- Throw `BatchProcessingException` on failure

8. Operational Requirements

8.1 Startup Sequence

1. Create `LinkedBlockingQueue`
2. Implement `BatchProcessor` and `CleanupStrategy`
3. Build `BatchCoordinator` with configuration
4. Call `coordinator.start()`
5. Start producer threads

8.2 Shutdown Sequence

1. Stop all producers (no new messages to queue)
2. Call `coordinator.shutdown(timeoutSeconds)`
3. Framework drains remaining messages from queue
4. Framework completes all in-flight batch processing
5. Consumer and worker threads terminate gracefully
6. Returns `true` if shutdown completed within timeout

> **Graceful Shutdown Guarantee:** No messages in queue are lost if shutdown timeout is sufficient. In-flight batches complete processing before threads terminate.

8.3 Monitoring and Observability

The framework provides logging at key points:

- Message received (batch ID, data/end-of-batch)
- Batch ID marked complete
- Sub-batch creation (count, sizes)
- Processing started/completed (batch ID, sequence)

- Failures (batch ID, sequence, exception)
- Cleanup triggered (batch ID, sequence)
- Shutdown initiated/completed

Current Implementation: `System.out.println()` (replace with SLF4J for production)

9. Performance Characteristics

9.1 Throughput Factors

Factor	Impact	Tuning
maxBatchSize	Larger = fewer transactions, higher throughput	Balance with transaction timeout limits
workerThreads	More threads = more parallel batch IDs	Match to DB connection pool size
Queue capacity	Larger = more buffering, handles bursts	Balance with memory constraints
DB performance	Primary bottleneck	Optimize batch inserts, indexes, etc.

9.2 Expected Performance

- **Queue latency:** < 1ms (in-memory, non-blocking)
- **Consumer latency:** < 100ms (poll timeout)
- **Processing latency:** Dominated by database transaction time
- **Throughput:** Constrained by (workerThreads × DB transaction rate)

9.3 Scalability Limits

Resource	Limit	Impact
Memory	Queue size × message size + in-flight batches	OOM if queue fills faster than processing
Threads	consumerThreads + workerThreads	Context switching overhead if too high
DB connections	Must match or exceed workerThreads	Connection pool exhaustion

10. Data Flow Examples

10.1 Example 1: Simple Batch (No Splitting)

```
Input: - 100 OrderDTO messages with batch ID "ORDER-001" - 1 end-of-batch message for "ORDER-001"
Configuration: - maxBatchSize = 5000 Processing: 1. Consumer receives 100 data messages,
accumulates 2. Consumer receives end-of-batch, marks "ORDER-001" complete 3. BatchAccumulator
creates 1 sub-batch (100 items, sequence 1) 4. Worker processes sub-batch 1 → SUCCESS 5.
processor.afterBatchId("ORDER-001") called 6. cleanupStrategy.markComplete("ORDER-001") called
Result: 100 items persisted in 1 transaction
```

10.2 Example 2: Split Batch

```
Input: - 10,000 OrderDTO messages with batch ID "ORDER-002" - 1 end-of-batch message for "ORDER-002" Configuration: - maxBatchSize = 5000 Processing: 1. Consumer receives 10,000 data messages, accumulates 2. Consumer receives end-of-batch, marks "ORDER-002" complete 3. BatchAccumulator creates 2 sub-batches: - Sub-batch 1: 5000 items, sequence 1 - Sub-batch 2: 5000 items, sequence 2 4. Worker processes sub-batch 1 → SUCCESS (5000 items committed) 5. Worker processes sub-batch 2 → SUCCESS (5000 items committed) 6. processor.afterBatchId("ORDER-002") called 7. cleanupStrategy.markComplete("ORDER-002") called Result: 10,000 items persisted in 2 transactions
```

10.3 Example 3: Failure with Cleanup

```
Input: - 10,000 OrderDTO messages with batch ID "ORDER-003" - 1 end-of-batch message for "ORDER-003" Configuration: - maxBatchSize = 5000 Processing: 1. Consumer receives 10,000 data messages, accumulates 2. Consumer receives end-of-batch, marks "ORDER-003" complete 3. BatchAccumulator creates 2 sub-batches 4. Worker processes sub-batch 1 → SUCCESS (5000 items committed) 5. Worker processes sub-batch 2 → FAILURE (database constraint violation) 6. Worker catches exception, logs error 7. cleanupStrategy.cleanup("ORDER-003", 2) called 8. Cleanup deletes all 5000 items from sub-batch 1 9. Worker continues with next batch ID Result: 0 items persisted (all-or-nothing for batch ID)
```

10.4 Example 4: Parallel Batch IDs

```
Input (interleaved): - 500 messages for "ORDER-001" - 500 messages for "ORDER-002" - 500 messages for "ORDER-003" - End-of-batch for all three Configuration: - maxBatchSize = 5000 - workerThreads = 4 Processing: 1. Consumer accumulates messages by batch ID 2. All three batch IDs marked complete 3. Three workers process in parallel: - Worker 1: ORDER-001 (500 items in 1 sub-batch) - Worker 2: ORDER-002 (500 items in 1 sub-batch) - Worker 3: ORDER-003 (500 items in 1 sub-batch) 4. All complete successfully Result: 1500 items persisted in 3 parallel transactions
```

11. Testing Requirements

11.1 Unit Test Coverage

Component	Test Cases
BatchMessage	Construction, validation, equality, null handling
BatchAccumulator	Grouping, splitting, edge cases (0, 1, exact max, over max)
BatchConsumer	Queue draining, end-of-batch handling, shutdown, interruption
BatchWorker	Success path, failure scenarios, cleanup invocation, hooks
BatchCoordinator	Lifecycle, configuration validation, graceful shutdown

11.2 Integration Test Scenarios

- End-to-end: Producer → Queue → Consumer → Worker → Mock Processor
- Split batches: 10K messages, verify 2 sub-batches processed
- Multiple batch IDs: Verify parallel processing
- Failure handling: Simulate DB error, verify cleanup
- Graceful shutdown: Verify queue drained, no data loss

11.3 Performance Testing

- Load test with production-like volumes
- Measure throughput at various maxBatchSize settings
- Verify no memory leaks under sustained load
- Test shutdown under heavy load

12. Limitations and Constraints

12.1 Known Limitations

- **No automatic retry:** Failed batches are not retried (producer responsibility)
- **In-memory only:** No persistent queue (messages lost on JVM crash)
- **Single JVM:** Not distributed across multiple servers
- **Synchronous cleanup:** Cleanup blocks worker thread
- **No partial success:** Batch ID is all-or-nothing

12.2 Design Constraints

- DTOs must be thread-safe (immutable recommended)
- Batch IDs must be unique per logical batch
- End-of-batch message is mandatory for each batch ID
- Sub-batches for same batch ID processed sequentially (cannot parallelize)
- CleanupStrategy must be idempotent

12.3 Resource Requirements

- Java 17 or higher
- Jakarta Persistence 3.1
- Database with ACID transaction support
- Sufficient heap memory for queue + in-flight batches
- DB connection pool sized for worker threads

13. Extensibility Points

13.1 Customization Interfaces

Interface	Purpose	Implementation Required
BatchProcessor<T>	Persistence logic	Yes
CleanupStrategy	Rollback/cleanup logic	Recommended
BatchRepository<T,E>	Abstract Jakarta repo	Optional (convenience)

13.2 Extension Examples

- **Custom validation:** Override beforeBatchId() to validate batch
- **Metrics:** Add counters in processor hooks
- **Auditing:** Log in afterBatchId() for compliance
- **Circuit breaker:** Wrap processor to fail fast on DB issues
- **Prioritization:** Use PriorityBlockingQueue instead of LinkedBlockingQueue

14. Compliance and Standards

14.1 Technology Standards

- **Jakarta Persistence (JPA) 3.1:** Standard persistence API
- **Java Concurrency Utilities:** ExecutorService, BlockingQueue
- **Maven:** Standard build tool
- **JUnit 5:** Standard testing framework

14.2 Code Quality Standards

- All public methods documented with JavaDoc
- Exception handling at all integration points
- Thread safety ensured via concurrent collections
- Comprehensive unit test coverage (>50 test cases)
- Integration tests for end-to-end scenarios

15. Glossary

Term	Definition
Batch ID	Unique string identifier grouping related DTOs into a logical batch
Sub-batch	A partition of a logical batch, limited by maxBatchSize, processed in single transaction
Sequence Number	1-based index of sub-batches within a batch ID (e.g., 1, 2, 3 for 3 sub-batches)
End-of-batch Message	Special message signaling no more data for a batch ID
DTO	Data Transfer Object - POJO carrying data to be persisted
Graceful Shutdown	Orderly termination draining queue and completing in-flight work
Cleanup	Removal of partial database writes after batch failure
Worker Thread	Thread responsible for processing all sub-batches of a batch ID
Consumer Thread	Thread that drains the queue and builds batches

16. Appendix: Configuration Examples

16.1 Low-Volume Configuration

```
// Scenario: 100-1000 items/minute, single batch ID at a time BatchCoordinator coordinator = new BatchCoordinator.Builder<T>() .queue(new LinkedBlockingQueue<>(1000)) .processor(processor) .maxBatchSize(1000) .consumerThreads(1) .workerThreads(2) .build();
```

16.2 High-Volume Configuration

```
// Scenario: 100K+ items/minute, multiple concurrent batch IDs BatchCoordinator coordinator = new BatchCoordinator.Builder<T>() .queue(new LinkedBlockingQueue<>(50000)) .processor(processor) .maxBatchSize(10000) .consumerThreads(1) .workerThreads(16) .build();
```

16.3 Memory-Constrained Configuration

```
// Scenario: Limited heap, small transactions preferred BatchCoordinator coordinator = new BatchCoordinator.Builder<T>() .queue(new LinkedBlockingQueue<>(5000)) .processor(processor) .maxBatchSize(500) .consumerThreads(1) .workerThreads(4) .build();
```

Document Status: Final

Last Updated: November 29, 2025

Contact: Batch Framework Development Team