Batch Job Processing Architecture Overview

Timesheet Management System

Attribute	Value
Architecture Pattern	Poll-based Database Queue
Processing Model	Asynchronous, Chunked
Concurrency	2 workers, 3 jobs/poll
Database	PostgreSQL 13
Authentication	Keycloak (OAuth2/JWT)
Output Formats	JSON, CSV, XML, PDF, Excel
Delivery Methods	Download, Email, SFTP

Table of Contents

1. Executive Summary

The Batch Job Processing system is a **poll-based, database-queue architecture** designed for generating and delivering reports in the Timesheet Management System. It processes scheduled and on-demand report generation requests, supporting multiple output formats and delivery mechanisms.

Key Features:

- Poll-based job scheduling with 5-second intervals
- Chunked processing for memory-efficient large dataset handling
- Role-based field masking (ADMIN, SUPERVISOR, CASE_WORKER)
- Multiple output formats: JSON, CSV, XML, PDF, Excel
- Job dependency support (parent-child job relationships)
- JWT token-based security throughout the pipeline

2. Architecture Overview

The system follows a layered architecture with clear separation of concerns:

Layer	Components	Responsibility
Client Layer	Web Dashboard, API Clients	User interaction, job submission
API Layer	REST Controllers	Request handling, validation
Service Layer	Business Services	Job queue, processing logic
Scheduler Layer	BatchJobScheduler	Poll, claim, dispatch jobs
Worker Layer	ThreadPoolExecutor	Parallel job execution
Data Layer	PostgreSQL, File System	Job storage, report files
Security Layer	Keycloak, JWT	Authentication, authorization

Architecture Flow:

```
CLIENT (Browser/API)
       |
       v
REST API LAYER (BusinessIntelligenceController)
       |
       v
SERVICE LAYER (JobQueueService)
       |
       v
DATABASE QUEUE (report_jobs table, status='QUEUED')
       |
       | <-- Poll every 5 seconds
       v
BATCH JOB SCHEDULER (claims jobs, status='PROCESSING')
       |
       v
THREAD POOL (2 workers execute BackgroundProcessingService)
       |
       v
OUTPUT (File System / Email / SFTP)
```

3. Tech Stack

Core Technologies

Layer	Technology	Version	Purpose
Runtime	Java	17 (LTS)	Application runtime
Framework	Spring Boot	3.2.0	Application framework
Database	PostgreSQL	13	Data store & job queue
Connection Pool	HikariCP	5.x	DB connection pooling
ORM	Spring Data JPA	6.x	Object-relational mapping
Security	Spring Security	6.x	Auth & authorization
Identity	Keycloak	22.x	OAuth2/OIDC provider

Supporting Libraries

Library	Purpose
Jackson	JSON serialization/deserialization
OpenPDF	PDF report generation
Spring Mail	Email delivery
JSch	SFTP file transfer
Freemarker	Email/report templates
AWS SDK	Secrets Manager integration
Lombok	Boilerplate code reduction

4. Design Choices

4.1 Poll-Based Database Queue vs Message Broker

Choice: Poll-based database queue using PostgreSQL

Rationale:

- Simpler infrastructure (no additional message broker)
- Transactional consistency with job data
- Built-in persistence and durability
- Easier debugging and monitoring via SQL
- Sufficient for expected load (~100 jobs/day)

4.2 Atomic Job Claiming

Choice: Optimistic locking with status check

When a worker claims a job, it atomically checks if status is 'QUEUED' and updates to 'PROCESSING'. This prevents multiple workers from processing the same job.

4.3 Chunked Processing

Choice: Process data in configurable chunks (default: 1000 records)

Benefits:

- Prevents out-of-memory errors on large datasets
- Enables real-time progress tracking
- Allows for cancellation between chunks
- Supports retry at chunk level

4.4 Push-Based Dependency Triggering

Choice: Parent job triggers dependent jobs upon completion

Instead of dependent jobs polling to check if parents are done, the parent job actively triggers dependent jobs when it completes. This provides lower latency and simpler logic.

4.5 JWT Token Capture at Submission

Choice: Store JWT token with job for later processing

The user's JWT token is stored with the job when submitted. When the background worker processes the job, it extracts the user's role and county from the stored token to apply appropriate data filtering and field masking.

5. End-to-End Flow

The following describes the complete flow of a typical batch job from user submission to download:

Step	Component	Action
1	Web Dashboard	User clicks "Generate Report", selects parameters
2	JavaScript	POST /api/bi/reports/generate with JWT token
3	Controller	Validates request, extracts user context from JWT
4	JobQueueService	Creates job with status=QUEUED, returns job ID
5	Dashboard	Displays job ID, starts polling for status
6	BatchJobScheduler	Polls DB every 5s, finds QUEUED job
7	BatchJobScheduler	Claims job (QUEUED → PROCESSING)
8	ThreadPool	Dispatches to BackgroundProcessingService
9	ProcessingService	Loads job, extracts JWT context
10	ProcessingService	Fetches data in chunks, applies masking
11	ProcessingService	Writes to file, updates progress
12	ProcessingService	Marks COMPLETED, triggers dependencies
13	Dashboard	Detects COMPLETED status, shows download button
14	User	Clicks download, receives report file

6. Component Details

Service Components

Component	Responsibility
BusinessIntelligenceController	REST API endpoints for job management
JobQueueService	Queue jobs, claim jobs, update status
BatchJobScheduler	Poll database, dispatch jobs to thread pool
BackgroundProcessingService	Execute job logic, chunked processing
JobDependencyService	Find and trigger dependent jobs
ScheduledReportService	Cron-based scheduled job creation
FieldMaskingService	Role-based field masking
ReportGenerationService	Multi-format report generation

Configuration Components

Component	Purpose
BatchProcessingConfig	Thread pool executor bean
BatchSchedulerProperties	Scheduler configuration (interval, pool size)
JobDependencyConfig	Dependency rules between job types
JobProcessingProperties	Chunk size settings
SecurityConfig	JWT and OAuth2 configuration

7. Data Model

Job Entity (report_jobs table)

Field	Type	Description
job_id	VARCHAR(50)	Primary key (JOB_XXXXXXXX)
status	VARCHAR(20)	QUEUED, PROCESSING, COMPLETED, FAILED, CANCELLED
progress	INTEGER	Completion percentage (0-100)
total_records	BIGINT	Expected total records to process
processed_records	BIGINT	Records processed so far
user_role	VARCHAR(50)	User role from JWT
report_type	VARCHAR(100)	Type of report to generate
data_format	VARCHAR(50)	Output format (JSON, CSV, PDF, etc.)
priority	INTEGER	Job priority (1-10, higher = first)
result_path	TEXT	Path to generated report file
error_message	TEXT	Error details if failed
jwt_token	TEXT	Stored JWT for processing context
parent_job_id	VARCHAR(50)	Parent job ID for dependencies
created_at	TIMESTAMP	Job creation time
started_at	TIMESTAMP	Processing start time
completed_at	TIMESTAMP	Completion time

Job Status State Machine

```
(start) --> QUEUED --> PROCESSING --> COMPLETED
              |            |
              |            +--> FAILED
              |            |
              +------------+--> CANCELLED
```

8. Security Architecture

Authentication Flow

1. User authenticates with Keycloak and receives JWT token
2. User includes JWT in Authorization header for API requests
3. Spring Security validates JWT with Keycloak
4. Role and county claims extracted from token
5. Token stored with job for background processing context

Role-Based Access Control

Role	Data Access	Field Masking
ADMIN	All counties	None (sees all fields)
SUPERVISOR	Assigned counties	Partial (SSN, Address masked)
CASE_WORKER	Single county	Full (all PII masked)

9. Scalability & Performance

Current Configuration

Parameter	Value	Impact
Worker threads	2	Max 2 concurrent jobs
Jobs per poll	3	Max 3 jobs queued per poll
Poll interval	5 seconds	Max 5s latency to start
Chunk size	1000 records	Memory-efficient processing
DB pool	10 connections	Concurrent DB access

Horizontal Scaling

The system supports horizontal scaling by running multiple application instances. The atomic job claiming mechanism (check-and-update in single transaction) ensures that each job is processed by exactly one worker, even across multiple instances. No additional coordination is required.

10. Deployment Architecture

Docker Compose Stack

Service	Image/Build	Port	Purpose
postgres	postgres:13	5432	Primary database
keycloak	keycloak:22	8080	Identity provider
spring-app	Custom build	8080	Main application
mailhog	mailhog/mailhog	8025	Email testing
mock-sftp	Custom build	2222	SFTP testing

JVM Configuration

The application runs with the following JVM settings:
• Heap: -Xmx2g -Xms512m
• GC: -XX:+UseG1GC
• Container-aware memory limits

11. Pros and Cons Analysis

Poll-Based Database Queue

Pros:

+ Simplicity - No additional infrastructure
+ Transactional consistency - Job state and data in same transaction
+ Easy debugging - Query jobs directly via SQL
+ Built-in persistence - Jobs survive restarts
+ Horizontal scaling ready - Atomic claiming works across instances

Cons:

- Polling overhead - Constant DB queries even when idle
- Latency - Up to 5 seconds delay to start jobs
- DB load - Adds load to primary database
- Limited throughput - Not suitable for 1000s of jobs/second

Chunked Processing

Pros:

+ Memory efficient - Processes large datasets without OOM
+ Progress tracking - Real-time progress updates
+ Cancellation support - Can stop between chunks
+ Retry granularity - Retry individual chunks on failure

Cons:

- Complexity - More complex than simple query-all
- Partial failures - Need cleanup of partial files
- Overhead - Multiple DB queries vs single query

12. Future Considerations

Potential Improvements

Message Broker Migration

Move to Kafka/RabbitMQ for higher throughput and event-driven architecture

Externalized Configuration

Move cron profiles to database/config service for dynamic job type registration

Enhanced Monitoring

Prometheus metrics, Grafana dashboards, alerting on job failures

Distributed Tracing

OpenTelemetry integration for end-to-end request tracking

Job Prioritization

Priority queues for urgent jobs, fair scheduling across users

Result Caching

Cache completed reports, serve repeated requests without regeneration

Summary

The Batch Job Processing system is a well-designed, pragmatic solution for report generation workloads. The poll-based database queue provides simplicity and reliability at the cost of some latency and scalability limits. For the expected load (~100 jobs/day), this architecture is appropriate and maintainable.

Key Strengths: Simple architecture, transactional consistency, role-based security, flexible output formats, dependency support

Key Limitations: 5-second poll latency, limited to ~1000 jobs/day, database as queue adds load

Recommended For: Low-to-medium volume batch processing, report generation workloads, systems prioritizing simplicity over throughput