# Java-Based Spring Batch Service for Migration

## 1. Overview

The migration job is implemented using Spring Batch, a lightweight, scalable batch-processing framework. It is designed to extract data from the on-premises i-scanned-image table, transform it, and load it into Amazon S3 (for BLOBs) and Amazon DynamoDB (for metadata).

## 2. Spring Batch Architecture

The migration job follows the ETL (Extract, Transform, Load) pattern and consists of the following key components:

### 2.1 Job Definition

|  |  |
| --- | --- |
| Parameter | Value |
| Job Name | iScannedImageMigrationJob |
| Job Type | Scheduled batch job |
| Execution Mode | Runs as a Spring Boot application deployed on an on-premises server |
| Scheduler | Cron-based execution using Spring Scheduler |

### 2.2 Step Breakdown

|  |  |  |
| --- | --- | --- |
| Step | Description | Component Used |
| Step 1: Read Data | Fetches records from Oracle's i-scanned-image table. | Spring Batch ItemReader (JDBC Cursor) |
| Step 2: Transform Data | Extracts BLOBs, converts them into S3-compatible format, and prepares metadata. | Custom Data Processor |
| Step 3: Upload BLOBs to S3 | Stores document binaries in an S3 bucket with a generated GUID. | AWS SDK for Java (S3 PutObject) |
| Step 4: Store Metadata in DynamoDB | Stores metadata including GUID, S3 path, and timestamps. | AWS SDK for Java (DynamoDB PutItem) |
| Step 5: Logging & Error Handling | Logs failed records, retries on failure, and generates a migration report. | Custom Error Handler |

## 3. Batch Processing Configuration

|  |  |  |
| --- | --- | --- |
| Parameter | Value | Description |
| Chunk Size | 1000 | Number of records processed in each batch |
| Thread Pool Size | 5 | Number of parallel threads for processing |
| Retry Mechanism | 3 Attempts | Retries failed records up to 3 times before logging |
| Commit Interval | 500 | Commits transactions after processing 500 records |

## 4. Spring Batch Configuration Example (Code Snippet)

@Configuration  
@EnableBatchProcessing  
public class BatchConfig {  
   
 @Autowired  
 private JobBuilderFactory jobBuilderFactory;  
   
 @Autowired  
 private StepBuilderFactory stepBuilderFactory;  
  
 @Bean  
 public ItemReader<ScannedImage> itemReader() {  
 return new JdbcCursorItemReaderBuilder<ScannedImage>()  
 .dataSource(dataSource)  
 .name("scannedImageReader")  
 .sql("SELECT id, document\_blob, metadata FROM i\_scanned\_image")  
 .rowMapper(new ScannedImageRowMapper())  
 .build();  
 }  
  
 @Bean  
 public ItemProcessor<ScannedImage, ProcessedImage> itemProcessor() {  
 return new ScannedImageProcessor();  
 }  
  
 @Bean  
 public ItemWriter<ProcessedImage> itemWriter() {  
 return new S3AndDynamoDBWriter();  
 }  
  
 @Bean  
 public Step migrationStep() {  
 return stepBuilderFactory.get("migrationStep")  
 .<ScannedImage, ProcessedImage>chunk(1000)  
 .reader(itemReader())  
 .processor(itemProcessor())  
 .writer(itemWriter())  
 .build();  
 }  
  
 @Bean  
 public Job migrationJob() {  
 return jobBuilderFactory.get("iScannedImageMigrationJob")  
 .incrementer(new RunIdIncrementer())  
 .flow(migrationStep())  
 .end()  
 .build();  
 }  
}

## 5. Monitoring and Error Handling

• Logging: All batch job execution details are logged in an on-prem database table and AWS CloudWatch.  
• Retry Mechanism: If an S3 upload or DynamoDB write fails, Spring Batch will retry 3 times before marking it as a failure.  
• Dead Letter Queue (DLQ): Failed records are stored separately for manual processing.  
• Audit Reports: A summary report is generated after each job execution and stored in S3.

## 6. Performance Considerations

• Parallel Execution: Uses multi-threading to improve performance.  
• Asynchronous Uploads: BLOBs are uploaded to S3 asynchronously while metadata is stored in DynamoDB.  
• Connection Pooling: Database connections are managed efficiently using a connection pool.  
• Batch Size Tuning: Optimal batch sizes (1000 records per commit) ensure better memory utilization.