19. 스프링 배치(Spring Batch)

Spring Batch 구조

Spring Batch 는 **대용량 배치 처리**를 지원하는 Spring 기반의 프레임워크다.
데이터 일괄 처리, ETL(Extract, Transform, Load), 정산/집계/보고서 생성 등에 널리 사용된다.

Spring Batch는 다음과 같은 핵심 특징을 가진다:

✅ 대용량 데이터 처리 가능
✅ 트랜잭션 관리, 재시작(Resume), Skip/Retry 기능 내장
✅ Job/Step 구성을 통한 유연한 파이프라인 구성
✅ 모니터링 및 관리 기능 제공

1️⃣ 기본 구성 흐름

```
JobLauncher → Job → Step(s) → Chunk 기반 처리 or Tasklet 기반 처리
```

👉 **Job** 단위로 전체 배치 실행 → 각 **Step** 이 개별 작업 단위 담당 → Step 내부에서 Chunk/Tasklet 방식으로 처리 🚀.

2️⃣ 주요 구성 요소

구성 요소	설명
Job	배치 작업 전체 단위
Step	Job 내의 작업 단위 (1개 이상 존재 가능)
JobLauncher	Job 실행을 트리거하는 컴포넌트
JobRepository	Job 실행/Step 실행 상태, 메타데이터 저장 (DB 기반)
JobExplorer	실행 중인 Job/Step 상태 조회 API 제공
JobInstance	동일 JobParameter 로 실행된 Job 의 논리적 인스턴스
JobExecution	JobInstance의 1회 실행 기록
StepExecution	Step의 1회 실행 기록

3️⃣ Step 구성 방식

1️⃣ Chunk 기반 처리 (가장 일반적)

```
Step → ItemReader → ItemProcessor → ItemWriter
```

N 개 단위(Chunk Size)로 읽고 → 처리하고 → 쓰기 트랜잭션 커밋.

- ItemReader: 데이터 소스에서 **데이터 읽기**
- ItemProcessor: 읽은 데이터에 대한 **변환/가공 처리**
- ItemWriter: 처리된 데이터를 **저장/쓰기**

2️⃣ Tasklet 기반 처리

- 단일 Tasklet → 1 Step 에서 **단순 반복 없는 작업 수행**.

예:

```
파일 압축 / 정리 / 상태 업데이트 등
```

```
@Bean
public Step exampleStep() {
    return stepBuilderFactory.get("exampleStep")
        .tasklet((contribution, chunkContext) -> {
            // Tasklet logic here
            return RepeatStatus.FINISHED;
        })
        .build();
}
```

4️⃣ 실행 흐름

```
Application → JobLauncher.run(Job, JobParameters)
    ↓
JobInstance 생성 or 조회
    ↓
JobExecution 생성 (1회 실행 단위)
    ↓
각 Step 순차 실행 (StepExecution 생성)
    ↓
Chunk 기반 or Tasklet 기반 처리 실행
    ↓
모든 Step 정상 종료 시 → JobExecution 완료
    ↓
JobRepository 에 실행 결과 기록
```

5️⃣ 주요 Bean 구성 예시

```
@Configuration
@EnableBatchProcessing
public class BatchConfig {

    @Autowired
    private JobBuilderFactory jobBuilderFactory;

    @Autowired
```

```
    private StepBuilderFactory stepBuilderFactory;

    @Bean
    public Job exampleJob() {
        return jobBuilderFactory.get("exampleJob")
            .start(exampleStep())
            .build();
    }

    @Bean
    public Step exampleStep() {
        return stepBuilderFactory.get("exampleStep")
            .<String, String>chunk(10)
            .reader(itemReader())
            .processor(itemProcessor())
            .writer(itemWriter())
            .build();
    }

    @Bean
    public ItemReader<String> itemReader() {
        return new ListItemReader<>(Arrays.asList("a", "b", "c"));
    }

    @Bean
    public ItemProcessor<String, String> itemProcessor() {
        return item -> item.toUpperCase();
    }

    @Bean
    public ItemWriter<String> itemWriter() {
        return items -> items.forEach(System.out::println);
    }
}
```

👉 **Chunk(10)** 으로 10개 단위로 트랜잭션 Commit 🚀.

6️⃣ 메타데이터 관리

Spring Batch는 기본적으로 **JobRepository** 를 통해 메타데이터 관리:

- BATCH_JOB_INSTANCE
- BATCH_JOB_EXECUTION
- BATCH_JOB_EXECUTION_PARAMS
- BATCH_STEP_EXECUTION
- BATCH_STEP_EXECUTION_CONTEXT

👉 배치 작업 재시작 시 **메타데이터 기반으로 어디까지 처리했는지** 자동 관리 가능.

---

7 구성 요소 시각화

```
+----------------+
|  JobLauncher   |
+----------------+
        |
        v
+----------------+        +--------------------+
|     Job        |-----> |   JobRepository     |
+----------------+        +--------------------+
        |
        v
+----------------+
|      Step      |
+----------------+
        |
        v
+--------+  +---------------+  +----------+
| Reader |→ | Processor(opt)|→|  Writer   |
+--------+  +---------------+  +----------+
```

---

8 Best Practice

영역	전략
Job 설계	1 Job = 1 Business Unit 기준으로 구성 권장
Step 설계	Chunk 기반 → 데이터 처리 트랜잭션 단위 최적화
재시작 설계	Restart 가능하도록 Idempotent 설계 필수
Logging	StepExecutionContext 활용 → 처리 중 상태 정보 저장 가능
Job Parameter	필수 입력값 (날짜 등)은 JobParameter 로 명시 관리 권장
Schedule 연계	Quartz, @Scheduled, 외부 CI/CD Job과 연계 가능

---

9 결론

✅ Spring Batch 는 대규모 데이터 배치 처리에 최적화된 프레임워크다.
✅ Job → Step → Chunk/Tasklet → **표준화된 구조**로 설계 가능.
✅ 트랜잭션 관리, Skip/Retry, 재시작(Resume) 기능을 기본 지원.
✅ 다양한 DataSource (DB, File, API 등) 연계 가능.
✅ **정산 처리 / ETL / 보고서 생성 / 통계 처리** 등에서 광범위하게 사용.

Job, Step, Reader, Processor, Writer

Spring Batch에서 배치 처리의 핵심은:

✅ 전체 작업 단위: **Job**
✅ 작업 단위 내 단계: **Step**
✅ 데이터 흐름 구성 요소: **ItemReader** → **ItemProcessor** → **ItemWriter**

이 구조는 **대용량 데이터 처리, ETL, 정산 작업, 보고서 생성** 등에 최적화된 패턴이다.

1️⃣ Job

정의

- **배치 처리의 전체 단위**
- 여러 개의 **Step** 으로 구성됨
- 실행 상태, 파라미터, 재시작 가능 여부 등을 관리

주요 특징

- `JobInstance`: 동일한 `JobName` + `JobParameters` 로 실행된 논리적 인스턴스
- `JobExecution`: JobInstance 의 실제 실행(1회 실행 기록)

구성 예시

```
@Bean
public Job exampleJob() {
    return jobBuilderFactory.get("exampleJob")
        .start(step1())
        .next(step2())
        .build();
}
```

👉 `start()` → 첫 Step
👉 `next()` → 다음 Step 순차 연결 가능

실행 흐름

```
JobLauncher.run(Job, JobParameters)
    ↓
JobInstance 생성
    ↓
JobExecution 생성
    ↓
Step 1 실행 → 완료 후 Step 2 실행 → ...
    ↓
전체 JobExecution 상태 기록
```

2️⃣ Step

정의

- **Job 내부의 단일 실행 단위**
- 하나의 Step 은 1개의 작업 논리적 흐름 담당
- 다음 두 가지 방식으로 구현 가능:

방식	설명
Tasklet 기반	단순 반복 없는 단일 작업 (파일 압축, 통계 업데이트 등)
Chunk 기반	대량 데이터 처리 시 사용 (Reader → Processor → Writer 흐름)

Chunk 기반 Step 예시

```
@Bean
public Step exampleStep() {
    return stepBuilderFactory.get("exampleStep")
        .<String, String>chunk(100)   // Chunk Size = 100
        .reader(itemReader())
        .processor(itemProcessor())
        .writer(itemWriter())
        .build();
}
```

👉 Chunk 기반 처리 → **100건 단위로 트랜잭션 Commit**.

3️⃣ ItemReader

정의

- **데이터를 읽어오는 컴포넌트**
- Spring Batch 가 처리할 **Input 데이터를 순차적으로 제공**함

종류

구현체	사용 대상
JdbcCursorItemReader	DB Cursor 기반 Read
JdbcPagingItemReader	DB Paging 기반 Read
FlatFileItemReader	CSV, TXT 등 파일 Read
StaxEventItemReader	XML Read
JpaPagingItemReader	JPA 기반 Read
Custom 구현	API, Kafka 등 외부 시스템 Read

예시

```
@Bean
public ItemReader<String> itemReader() {
    return new ListItemReader<>(Arrays.asList("A", "B", "C", "D"));
}
```

👉 Reader 는 **null 리턴 시 Step 종료**.

4️⃣ ItemProcessor

정의

- **읽어온 데이터를 변환/가공/필터링** 하는 컴포넌트
- 입력값 → 출력값(혹은 null 반환으로 Skip) 변환 수행
- Chunk 내에서 **각 Item 단위로 호출됨**.

예시

```
@Bean
public ItemProcessor<String, String> itemProcessor() {
    return item -> item.toLowerCase();
}
```

활용 예시

- 데이터 포맷 변환
- 특정 조건에 따른 필터링 (→ null 반환 시 해당 Item 은 Writer 에 전달되지 않음)
- 외부 API 연동 → 보강 정보 추가
- Data Enrichment

👉 Processor 는 **비즈니스 로직 적용 핵심 포인트**.

5️⃣ ItemWriter

정의

- Processor 를 거친 데이터를 최종 저장/쓰기 처리
- **Chunk 단위로 호출**됨 → 예: Chunk Size = 100 → 100개 Item 묶음으로 Writer 호출
- 데이터 저장 대상:

대상	구현체 예시
DB	JdbcBatchItemWriter, JpaItemWriter
File	FlatFileItemWriter

대상	구현체 예시
Message Queue	Custom 구현 필요
API 호출	Custom 구현 필요

예시

```
@Bean
public ItemWriter<String> itemWriter() {
    return items -> items.forEach(System.out::println);
}
```

👉 실제 서비스에서는 DB Insert/Update, 파일 기록, API 전송 등에 사용.

6️⃣ Chunk 기반 처리 흐름

```
Step 시작
    ↓
Reader → Item 1 → Processor → Result 1 → (배치 목록에 담음)
Reader → Item 2 → Processor → Result 2 → (배치 목록에 담음)
...
Item N → Processor → Result N → (배치 목록에 담음)
    ↓
ChunkSize 도달 → Writer 호출 → Result 1~N 묶음 저장
    ↓
Chunk 반복
    ↓
Reader → null 반환 시 Step 종료
```

👉 Chunk 처리 덕분에 **대량 데이터 처리 시 트랜잭션 관리 최적화** 가능 🚀.

7️⃣ 구성 흐름 정리

```
JobLauncher
    ↓
Job
    ↓
Step (Chunk 기반)
    ↓
[ ItemReader → ItemProcessor → ItemWriter ]
```

각 구성 요소는 **Bean 으로 선언하고 재사용 가능** → 유연한 Pipeline 구성 가능.

8️⃣ Best Practice

구성 요소	전략
ItemReader	null 반환 시 Step 정상 종료 → Reader 구현 시 주의
ItemProcessor	필터링, 변환 로직 구현 → 최대한 경량화 필요
ItemWriter	Writer 내부에서 반드시 **Batch 처리** 권장 (단건 Write는 비효율 발생)
Chunk Size	처리 대상 데이터 특성에 맞게 튜닝 필요 (작으면 Commit 빈번, 크면 Memory 사용량 증가)
트랜잭션 관리	Chunk 기반으로 Rollback 가능 → Skip/Retry 전략 적용 가능

9️⃣ 결론

✅ **Job** → 전체 배치 단위, **Step** → 개별 작업 단위
✅ Chunk 기반 Step → 대량 데이터 처리에 최적화
✅ ItemReader → 데이터 원천에서 데이터 읽기
✅ ItemProcessor → 데이터 가공/변환/필터링
✅ ItemWriter → 최종 데이터 저장/쓰기 처리
✅ 구성 단순하고 유연 → 다양한 Batch 처리 Pipeline 설계 가능.

Chunk 처리 / Tasklet 처리

Spring Batch Step 에는 두 가지 주요 처리 모델이 있다:

✅ **Chunk-Oriented Processing (Chunk 처리)**
✅ **Tasklet-Oriented Processing (Tasklet 처리)**

각 모델은 배치 처리의 **성격, 데이터 특성, 트랜잭션 관리 방식**에 따라 적합한 경우가 다르다.
→ 어떤 경우에 어떤 방식을 선택해야 하는지 명확하게 이해하고 설계해야 한다.

1️⃣ Chunk 처리란?

기본 개념

- **대량 데이터 처리** 시 사용하는 방식
- Reader → Processor → Writer → **Chunk Size 단위로 트랜잭션 커밋**
- 하나의 Chunk 내에서:
    - Reader 가 Item 여러 개 읽음 → Processor 로 처리 → Writer 에 일괄 저장
- 트랜잭션은 **Chunk 단위**로 묶어서 관리됨.

구조 흐름

```
Step
    → Chunk (ex. size=100)
        → ItemReader → ItemProcessor → ItemWriter
    → Commit
    → Next Chunk 반복
```

코드 예시

```
@Bean
public Step chunkStep() {
    return stepBuilderFactory.get("chunkStep")
        .<String, String>chunk(100)
        .reader(itemReader())
        .processor(itemProcessor())
        .writer(itemWriter())
        .build();
}
```

👉 **100개 단위로 트랜잭션 Commit.**

특징

✅ 트랜잭션 경계 명확 (Chunk 단위)
✅ Skip/Retry 지원
✅ 재시작(Resume) 지원 → 실패한 Chunk 이후부터 재실행 가능
✅ 대용량 데이터 처리에 최적화
✅ Item 기반 Batch 처리 표준 패턴

2️⃣ Tasklet 처리란?

기본 개념

- **단일 태스크(작업)** 를 처리하기 위한 방식
- Reader/Processor/Writer 구성 없이 → **Tasklet 하나로 전체 로직 수행**
- 주로 다음과 같은 작업에 사용:

```
파일 이동/압축/삭제
DB 통계 값 업데이트
단일 API 호출 처리
디렉토리 정리
Trigger 용도 Step
```

구조 흐름

```
Step
    → Tasklet 실행 → RepeatStatus (FINISHED or CONTINUABLE)
```

코드 예시

```
@Bean
public Step taskletStep() {
    return stepBuilderFactory.get("taskletStep")
        .tasklet((contribution, chunkContext) -> {
            System.out.println("Tasklet 작업 수행 중...");
            return RepeatStatus.FINISHED;
        })
        .build();
}
```

👉 간단한 로직은 람다 Tasklet 으로 바로 구현 가능 🚀.

특징

✅ 단순/단일 작업에 적합
✅ 트랜잭션 경계 → **Tasklet 전체 단위로 커밋됨**
✅ Chunk 처리가 필요 없는 경우 빠르게 구현 가능
✅ Reader/Processor/Writer 필요 없음
✅ 프로그래밍적 유연성 ↑

3️⃣ Chunk vs Tasklet 비교표

항목	Chunk 처리	Tasklet 처리
주요 목적	대량 데이터 처리	단일 작업 처리
트랜잭션 단위	Chunk Size 단위 Commit	Tasklet 전체 단위 Commit
구성 요소	ItemReader, ItemProcessor, ItemWriter 필요	Tasklet (단일 메서드) 필요
처리 패턴	반복적 처리에 최적화	일회성 작업에 최적화
Skip/Retry	지원	지원 안함 (Tasklet 내부에서 수동 구현 가능)
Restart 지원	지원	지원 (단, 상태 관리 직접 구현 필요)
대표 사례	ETL 처리, DB → DB 전송, 파일 → DB 전송	파일 압축, FTP 전송, 디렉토리 정리, API 호출

---

4 설계 기준

상황	추천 처리 방식
대량 데이터 처리 (10만 건 이상)	Chunk 처리
트랜잭션을 Chunk Size 로 제어하고 싶을 때	Chunk 처리
단순 파일 이동/압축/정리 작업	Tasklet 처리
외부 API 단일 호출 후 상태 업데이트	Tasklet 처리
주기적 DB 통계값 계산 → 저장	Tasklet 처리
정산/집계 처리 (Row 기반으로 처리 필요)	Chunk 처리
실패 후 Resume(재시작) 필요	Chunk 처리 권장

---

5 Chunk 처리 주의사항

- Chunk Size 는 **성능에 직접적인 영향** → 적절히 튜닝 필요
  - 너무 작으면 → 트랜잭션 빈번 → 성능 저하
  - 너무 크면 → 메모리 사용량 증가
- Skip/Retry 설정 시 **ItemWriter 재처리 유의**
  - Writer 구현 시 **Idempotent(멱등성)** 보장 중요
- Reader → null 리턴 시 Step 정상 종료

---

6 Tasklet 처리 주의사항

- **트랜잭션 제어 필요 시 명확히 설정**
  - 전체 Tasklet 작업이 트랜잭션으로 묶임
  - 내부 작업 시 반드시 중간 상태 관리 고려 필요
- StepExecutionContext 활용 시 상태 저장 가능
  - Tasklet 작업 중 Resume 구현 가능 (ex. 큰 파일 처리 Tasklet)
- 너무 복잡한 Tasklet 구현은 지양
  - 복잡한 반복 작업은 Chunk 처리로 전환 권장

---

7 결론

✅ **Chunk 처리** 는 대량 데이터 처리, ETL 등에 필수적인 표준 패턴이다.
✅ **Tasklet 처리** 는 단일 작업(파일 처리, API 호출 등)에 적합하다.
✅ 둘은 상호보완적으로 구성 가능 → 하나의 Job 에 Chunk Step + Tasklet Step 병행 구성 多.
✅ 설계 단계에서 **데이터 처리 패턴과 트랜잭션 경계**에 맞춰 적절한 방식을 선택해야 한다.

DB 연동 배치 / 파일 기반 배치

1️⃣ 스프링 배치 기본 구조 복습

- **Job** → 배치 작업의 단위
- **Step** → Job 안의 단계적 처리 단위
- **ItemReader** → 데이터 읽기 담당
- **ItemProcessor** → 데이터 처리 담당
- **ItemWriter** → 데이터 쓰기 담당
- **JobRepository, JobLauncher** → 배치 실행과 상태 관리
- **ExecutionContext** → Step 간 상태 저장

2️⃣ DB 연동 배치

✨ 목적

- RDB에서 데이터를 **대량으로 읽어와서 가공 후 저장** 또는 **다른 DB로 이동**
- 실시간 처리가 아닌 **일괄 처리(배치)** 형태

✨ 구성

1. ItemReader

구현체	설명
`JdbcCursorItemReader`	커서 기반 Streaming (대용량)
`JpaPagingItemReader`	JPA 기반 페이징 처리
`JdbcPagingItemReader`	순수 JDBC 기반 페이징
`HibernateCursorItemReader`	Hibernate 기반 커서 처리

2. ItemWriter

구현체	설명
`JdbcBatchItemWriter`	JDBC batch insert/update 지원
`JpaItemWriter`	JPA 기반 persist/merge
`HibernateItemWriter`	Hibernate 기반 persist/merge
`StoredProcedureItemWriter`	프로시저 호출로 데이터 저장

✨ 주요 고려사항

- **트랜잭션 범위** 관리 → Chunk 단위
- **대량 처리 시 메모리 관리** 중요
- **DB 락 주의** → `JdbcCursorItemReader`는 주의 필요
- **병렬 처리 (Partitioning)** 사용 가능

✨ 예시 코드 (JpaPagingItemReader 기반)

```
@Bean
public JpaPagingItemReader<Member> memberReader(EntityManagerFactory emf) {
    JpaPagingItemReader<Member> reader = new JpaPagingItemReader<>();
    reader.setEntityManagerFactory(emf);
    reader.setQueryString("SELECT m FROM Member m WHERE m.status = :status");

    Map<String, Object> parameters = new HashMap<>();
    parameters.put("status", "ACTIVE");
    reader.setParameterValues(parameters);
    reader.setPageSize(100);
    return reader;
}
```

3️⃣ 파일 기반 배치

✨ 목적

- **CSV, TSV, Fixed-Length, XML, JSON** 파일 → DB 저장 또는 반대 방향
- **파일 → 파일 변환** (ETL)

✨ 구성

1. ItemReader

구현체	설명
`FlatFileItemReader`	CSV, TXT, TSV 등 flat file 처리
`StaxEventItemReader`	XML 파일 처리
JSON Reader (커스텀 구현 필요)	JSON → Jackson 이용 직접 구현

2. ItemWriter

구현체	설명
`FlatFileItemWriter`	CSV, TXT, TSV 등 flat file 쓰기
`StaxEventItemWriter`	XML 파일 쓰기

구현체	설명
JSON Writer (커스텀 구현 필요)	JSON → Jackson 이용 직접 구현

✨ 주요 고려사항

- **파일 인코딩** (UTF-8, EUC-KR 등)
- **헤더/푸터 라인 처리**
- **에러 발생 시 롤백 전략** 설정
- **대용량 파일일 경우 ChunkSize 조정 및 Buffer 전략**

✨ 예시 코드 (FlatFileItemReader + FlatFileItemWriter)

Reader 설정 (CSV 파일 읽기)

```
@Bean
public FlatFileItemReader<Customer> customerReader() {
    FlatFileItemReader<Customer> reader = new FlatFileItemReader<>();
    reader.setResource(new ClassPathResource("customer.csv"));
    reader.setLinesToSkip(1); // Header skip
    reader.setLineMapper(new DefaultLineMapper<Customer>() {{
        setLineTokenizer(new DelimitedLineTokenizer() {{
            setNames("id", "name", "email");
        }});
        setFieldSetMapper(new BeanWrapperFieldSetMapper<Customer>() {{
            setTargetType(Customer.class);
        }});
    }});
    return reader;
}
```

Writer 설정 (CSV 파일 쓰기)

```
@Bean
public FlatFileItemWriter<Customer> customerWriter() {
    FlatFileItemWriter<Customer> writer = new FlatFileItemWriter<>();
    writer.setResource(new FileSystemResource("output/customers_output.csv"));
    writer.setLineAggregator(new DelimitedLineAggregator<Customer>() {{
        setDelimiter(",");
        setFieldExtractor(new BeanWrapperFieldExtractor<Customer>() {{
            setNames(new String[] { "id", "name", "email" });
        }});
    }});
    return writer;
}
```

4️⃣ 실전 활용 시 패턴

패턴 유형	추천 기술 요소
DB → DB	`JdbcCursorItemReader` + `JdbcBatchItemWriter`
DB → CSV	`JpaPagingItemReader` + `FlatFileItemWriter`
CSV → DB	`FlatFileItemReader` + `JpaItemWriter`
DB → JSON	`JpaPagingItemReader` + Custom JSON Writer
파일 변환 (CSV → CSV)	`FlatFileItemReader` + `FlatFileItemWriter`

5️⃣ 확장 고려사항

- **배치 재실행 전략**: 이미 처리된 데이터 skip (idempotent)
- **오류 발생 시 skip 정책** 설정
- **대량 데이터일 경우 Partitioning, Parallel Step 적용**
- **모니터링**: ExecutionContext + JobRepository 활용

스케줄링 연계, 실패 복구

1️⃣ 스케줄링 연계란?

✨ 목적

- **정해진 시간 / 주기**에 배치 작업 자동 실행
- **운영 자동화**를 위한 필수 구성
- 예시:
  - 매일 새벽 3시 정산
  - 매 5분마다 로그 수집
  - 매주 금요일 리포트 생성

✨ 주요 방법

방법	특징
`@Scheduled` (Spring 제공)	간단한 주기성 작업에 적합
Quartz Scheduler	고급 스케줄링 기능 지원 (분산, 트랜잭션 보장 등)
외부 스케줄링 (cron + systemd, Kubernetes CronJob)	시스템 단에서 관리 (보다 독립적)

1️⃣.1 `@Scheduled` 사용법

```
@EnableScheduling
@Configuration
public class BatchScheduler {

    private final JobLauncher jobLauncher;
    private final Job myBatchJob;

    public BatchScheduler(JobLauncher jobLauncher, Job myBatchJob) {
        this.jobLauncher = jobLauncher;
        this.myBatchJob = myBatchJob;
    }

    @Scheduled(cron = "0 0 3 * * ?") // 매일 새벽 3시
    public void runBatchJob() throws Exception {
        JobParameters params = new JobParametersBuilder()
                .addLong("time", System.currentTimeMillis())
                .toJobParameters();
        jobLauncher.run(myBatchJob, params);
    }
}
```

주의사항

- **JobParameters 반드시 고유**해야 함 → `time` 추가
- 동일 JobParameters 로 실행하면 **이미 완료됨**으로 간주 → 재실행 안 됨

1️⃣.2 Quartz Scheduler 연계

장점

- **분산 스케줄링 가능**
- Job 실행 이력 관리
- 미완료 시 재시도 기능 내장

구성 예시

1. `QuartzJobBean` 상속
2. `JobLauncher` 통해 Spring Batch Job 실행

```
@Component
public class QuartzBatchJob extends QuartzJobBean {

    @Autowired
    private JobLauncher jobLauncher;

    @Autowired
    private Job myBatchJob;

```

```
    @Override
    protected void executeInternal(JobExecutionContext context) throws
JobExecutionException {
        try {
            JobParameters params = new JobParametersBuilder()
                    .addLong("time", System.currentTimeMillis())
                    .toJobParameters();
            jobLauncher.run(myBatchJob, params);
        } catch (Exception e) {
            throw new JobExecutionException(e);
        }
    }
}
```

→ Quartz Trigger 와 JobDetail 설정 필요 (자바 Config 또는 XML)

1️⃣.3 외부 스케줄링 활용 (systemd / Kubernetes CronJob 등)

```
# Linux crontab 예시
0 3 * * * java -jar my-batch.jar
```

- 장점: Application 이 다운되더라도 OS 수준에서 관리 가능
- 운영 환경에서는 보통 Quartz or 외부 스케줄링 + 모니터링 조합 사용

2️⃣ 실패 복구 전략

✨ 목적

- 배치 실행 도중 **장애 발생 시 복구 가능하게 설계**
- "처음부터 다시"가 아닌 **중단된 지점부터 복원 가능**
- 운영 신뢰성 확보

2️⃣.1 ExecutionContext 활용

- StepExecutionContext 에 상태 저장 → 다음 실행 시 복원 가능

구성

```
reader.open(stepExecution.getExecutionContext());
processor.open(stepExecution.getExecutionContext());
writer.open(stepExecution.getExecutionContext());
```

- FlatFileItemReader, JpaPagingItemReader 등은 기본적으로 내부에 복구 지원 옵션 있음 (`saveState = true`)

```
reader.setSaveState(true);
```

→ 자동으로 ExecutionContext 에 **lastReadPosition** 저장됨

2️⃣.2 Incrementer 활용

- JobParameter 에 `RunIdIncrementer` 적용 → 매 실행마다 고유 ID 생성

```
@Bean
public Job myBatchJob(JobBuilderFactory jobBuilderFactory, Step myStep) {
    return jobBuilderFactory.get("myBatchJob")
            .incrementer(new RunIdIncrementer())
            .flow(myStep)
            .end()
            .build();
}
```

- 이 구조를 사용하면 동일 JobParameters 로 재실행 가능

2️⃣.3 Restartable Job 설계

요소	적용 방법
`saveState = true`	Reader / Writer 에 적용
Incrementer 사용	JobParameter 관리
JobRepository 사용	JobExecution 상태 기록 / 조회 가능
`JobExplorer` 사용	기존 JobExecution 확인 후 재시작 가능

상태 관리 흐름

```
STARTED → STOPPED → STARTED → COMPLETED / FAILED
```

- STOPPED/FAILED 상태 → 이후 **재시작 시 이어서 복구 가능**

2️⃣.4 재처리 정책 구성

- `skipPolicy` → 에러 발생 시 skip
- `retryPolicy` → 재시도
- `faultTolerant()` → skip/retry 조합 가능

```
stepBuilderFactory.get("myStep")
    .<InputType, OutputType>chunk(100)
    .reader(reader)
    .processor(processor)
    .writer(writer)
    .faultTolerant()
    .skip(Exception.class)
    .skipLimit(10)
    .retry(Exception.class)
    .retryLimit(3)
    .build();
```

- 예시:
  - 최대 10건까지 skip 허용
  - 에러 발생 시 최대 3회까지 retry

---

3 실전 운영 Tips

✅ 스케줄링 구성 시

- 운영 서버에서는 **Quartz + External Scheduler** 병행 추천
- 개발/테스트 단계에서는 **@Scheduled** 로 빠르게 테스트 가능

✅ 실패 복구 구성 시

- **saveState, ExecutionContext 적극 활용**
- JobRepository 반드시 유지 (H2 사용 말고 MySQL/Postgres 권장)
- 실패 시 운영툴에서 수동 재시작 → 이어서 복구 가능

✅ 모니터링

- **Actuator** 로 Job 상태 노출 가능 (`/actuator/batch` 등)
- JobExecution 상태를 DB 기반 모니터링 페이지 구성 권장

정리하면:

기능	주 용도 / 목적
@Scheduled	간단한 주기적 실행
Quartz	고급 스케줄링, 분산 환경 지원
External Scheduler (cron/systemd/K8S)	독립적, 시스템 수준 제어
saveState	Step 단위 상태 복구
RetryPolicy	장애 시 재시도 지원
skipPolicy	일부 건 skip 처리

기능	주 용도 / 목적
JobExplorer / JobRepository	Job 상태 관리, 재시작 지원