

Spring Scheduling

Spring Batch

❖ 애자일 개발

✓ 애자일 프로세스의 테넌트

- 고객 만족은 동작하는 소프트웨어를 고객에게 빠르게 전달함으로써 얻어짐
- 개발 단계에 관계없이 변경을 환영
- 동작하는 소프트웨어를 자주 배포
- 비즈니스 담당자와 개발자는 매일 함께 일해야 함
- 동기가 부여된 구성원으로 프로젝트를 구성
- 구성원에게 필요한 도구를 제공하고 작업을 완수할 것이라고 믿음
- 얼굴을 맞대고 이뤄지는 의사 교환이 가장 효과적
- 동작하는 소프트웨어가 진척도의 가장 중요한 척도
- 지속 가능한 개발을 장려 – 팀원 전원이 개발 속도를 꾸준히 유지할 수. 있어야 함
- 우수한 기술력과 좋은 설계 능력을 갖추도록 지속적으로 노력
- 불필요한 작업을 없애 낭비를 최소화
- 자기 조직화 된 팀이 최고의 요구 사항, 아키텍처, 설계를 만들어 냄
- 팀은 정기적으로 어떻게 개선할지 결정하고 이를 반영

Spring Batch

❖ 애자일 개발

✓ 사용자 스토리로 요구 사항 포착

- 사용자 스토리는 요구 사항을 문서화하는 애자일 방법
- 사용자 스토리는 사용자가 원하는 애플리케이션의 기능을 기록하는 것으로 사용자가 시스템과 상호 작용하는 방법을 개발 팀에 전달하고 해당 상호 작용의 테스트 가능한 결과를 기록하는 데 목적이 있음
- 사용자 스토리의 세 가지 주요 부분

◆ title

- 제목은 스토리의 내용과 관련된 단순하고 간결한 설명이어야 함
- 거래 파일 내 정보를 적재한다 또는 인쇄 파일을 생성한다 와 같은 스토리 제목이 좋은 예
- 제목은 GUI와 관련된 내용으로 한정되지는 않음
- GUI가 없다고 해서 사용자 간에 상호 작용할 수 없는 것은 아니기 때문
- GUI가 없을 때 사용자는 문서로 만드는 배치 처리이거나 통신해야 할 (즉 인터페이스해야 할) 특정 외부 시스템

Spring Batch

❖ 애자일 개발

✓ 사용자 스토리로 요구 사항 포착

● 사용자 스토리의 세 가지 주요 부분

◆ native

- 내러티브는 사용자 스토리로 문서화하려는 상호 작용을 사용자 관점에서 간단히 설명한 것
- 일반적으로 Y 상황에서 X 가 무언가를 수행하고 어떤 다른 일이 발생한다와 같은 형식을 가짐

◆ acceptance criteria

- 인수 기준은 스토리의 완료 여부를 판단하는 데 사용할 수 있는 테스트 가능한 요구 사항
- 인수 기준을 유용하게 사용하려면 어떤 방식으로든 검증할 수 있어야 함
- 주관적인 요구 사항이 아니라 개발자가 네, 그렇습니다 또는 아니요 그렇지 않습니다 라고 말할 수 있는 객관적인 항목이어야 함

Spring Batch

❖ 애자일 개발

✓ 사용자 스토리로 요구 사항 포착

● 사용자 스토리 예

◆ 제목: 텔레비전 켜기

◆ 내러티브: 사용자는 텔레비전, 수신기, 케이블 박스를 끈 상태에서 범용 리모컨의 전원 버튼을 누를 수 있고 리모컨은 텔레비전, 수신기, 케이블 박스의 전원을 켜고 텔레비전 쇼를 보도록 구성

◆ 인수 기준

- 범용 리모컨에 전원 버튼이 있음
- 사용자가 전원 버튼을 누르면 다음과 같이 동작
 - 텔레비전 전원이 켜진다.
 - AV 수신기 전원이 켜진다.
 - 케이블 박스 전원이 켜진다.
 - 케이블 박스는 187 채널로 설정될 것이다.
 - AV 수신기는 SAT 입력으로 설정될 것이다.
 - TV는 비디오 1 입력으로 설정될 것이다.

Spring Batch

❖ 애자일 개발

✓ 사용자 스토리로 요구 사항 포착

● 사용자 스토리 예

- ◆ 위 사용자 스토리는 텔레비전 크기 라는 제목으로 시작하는데 해당 제목은 짧지만 내용을 잘 설명하고 있음
- ◆ 그 다음에 내러티브로 이어지는데 위 예에서 내러티브는 사용자가 전원 버튼을 누를 때 발생하는 상황을 설명
- ◆ 마지막으로 인수 기준에는 개발자 및 QA(Quality Assurance) 담당자가 사용할 수 있는 테스트 가능한 요구 사항이 나열돼 있는데 각 기준은 개발자가 쉽게 검사할 수 있는 항목으로 개발자는 개발된 시스템을 대상으로 인수 기준에 적힌 이렇게 해야 한다 나 이렇게 하면 안 된다 라는 항목을 보면서 예 또는 아니요 를 말할 수 있어야 함

● 사용자 스토리(USER STORY) 유스케이스(USE CASES)와의 차이

- ◆ 유스케이스는 다른 익숙한 형태의 요구 사항 문서이며 사용자 스토리와 유사하게 행위자(actor) 중심적
- ◆ 유스케이스는 RUP(Rational Unified Process)의 문서화 양식으로 행위자와 시스템 간의 상호 작용의 모든 측면을 문서화할 때 사용하기 때문에 지나치게 문서 중심적(문서를 위한 문서를 작성하게 됨)이고 포맷도 비대해져 인기를 잃게 됐으며 애자일 개발에서는 유스케이스 대신 사용자 스토리를 사용

Spring Batch

❖ 애자일 개발

✓ 테스트 주도 개발로 설계

- Test-Driven Development은 애자일 실천 방식
- TDD를 사용한다면 개발자는 실패하는 테스트를 먼저 작성한 이후에 코드를 구현해 테스트를 통과시키는 방식
- TDD는 개발자들이 코드를 작성하기 전에 무엇을 코드화 하려고 하는지 생각하도록 설계됐으며 이로써 개발자의 생산성을 높이고 디버거를 적게 사용하며 더 깨끗한 코드를 만들 수 있다는 것이 입증됨
- TDD의 또 다른 장점은 테스트가 실행 가능한 문서 역할을 한다는 점인데 유지 보수가 잘 이뤄지지 않아 오랫동안 현행화 되지 않는 사용자 스토리나 다른 형태의 문서와 달리 자동화 된 테스트는 지속적인 유지보수 활동 중 하나로서 항상 최신화되며 조각이 어떻게 동작하는지 이해하고 싶다면 전체적인 시나리오 아래에서 코드를 개발한 개발자의 의도에 맞게 작성된 단위 테스트를 살펴보면 됨
- TDD는 개발 시에 사용 할 수 있는 훌륭한 도구이긴 하지만 동작 방식을 설명하는 데 최선은 아님

Spring Batch

❖ 애자일 개발

✓ 버전 관리 시스템 사용

- 반드시 필요한 것은 아니지만 모든 개발에 소스 관리 시스템을 사용하는 것이 좋음
- Git 이나 Github 또는 그 외 어떤 다른 형태의 버전 관리 시스템을 사용하든지 상관없이 소스 관리 시스템이 제공하는 기능은 프로그래밍의 생산성을 높이는 데 필수적
- 한 번 배우고 버릴 코드의 소스 관리를 해야 할 이유가 있을까 라고 생각할 수도 있지만 오히려 이 점이 소스 관리를 하는 가장 강력한 이유가 될 수 있음
- 버전 관리 시스템을 사용하면 개발 시 어떤 시도를 하더라도 걱정하지 않아도 되는 안전망이 생기는데 잘 동작하는 코드 를 커밋하고 그 다음에 동작하지 않을지도 모르는 변경을 가해서 변경한 코드가 잘 작동하면 새 버전을 커밋하고 그렇지 않다면 아무 피해 없이 이전 버전으로 롤백 할 수 있음
- 버전 관리 시스템을 사용함으로써 골치 아픈 일을 줄이고 실수는 통제된 환경에서 발생하도록 하는 것이 좋음

Spring Batch

❖ Batch Processing

- ✓ 대부분의 애플리케이션은 사용자와 상호작용(웹 페이지의 링크를 클릭하거나 thick 애 클라이언트가 제공하는 양식에 맞춰 정보를 입력하거나 미들웨어를 통해 메시지를 수신하거나 휴대전화나 태블릿 앱을 손가락으로 터치)하도록 개발
 - thick application: 각종 애플리케이션을 내장해 서버에 보관되는 공유 데이터 관리 이외의 처리를 수행하는 클라이언트
- ✓ 서비스를 개발할 때 API 기능만으로 모든 기능을 개발할 수 없는 상황이 종종 발생
- ✓ API는 사용자 요청을 받아 실행되는 특징이 있기 때문에 애플리케이션 외부에서 이벤트가 발생하여 기능을 실행
- ✓ API는 빠르게 기능을 실행하고 응답해야 하므로 API는 크기가 큰 데이터를 처리 하기에는 적합하지 않음
- ✓ API로 처리하기 어려운 상황
 - 예약 정보 전파: 매일 새벽 2시에 하루 동안의 호텔 예약 데이터를 읽어 가공한 후 각 호텔에 예약 정보를 이메일로 전달
 - 주간 판매 통계: 매주 월요일 새벽 3시에 지난 주에 판매된 데이터를 가공하여 주간 통계를 작성
 - 썸네일 생성: 매시간 새로 등록된 이미지 파일을 조회하여 썸네일 이미지로 변환
 - 웰컴 쿠폰 발송: 10분 간격으로 새로 가입한 사용자의 이메일로 환영 쿠폰을 전달

Spring Batch

❖ Batch Processing

- ✓ 배치 프로세스는 사용자 개입 없이 스케줄링 작업에 따라 여러 자료를 일괄적으로 처리하는 작업을 의미하는데 상호 작용이나 중단 없이 유한한 양의 데이터를 처리하는 것으로 정의하는데 배치 처리가 일단 시작되면 아무런 개입 없이 어떤 형태로든 완료
- ✓ 배치 프로세스는 스케줄링과 트리거, 태스크로 구성
 - 태스크는 실행할 작업
 - 스케줄링은 매일 새벽 2시, 매주 일요일 새벽 2시 처럼 특정 시간을 예약하는 것
 - 트리거는 스케줄에 맞추어 태스크를 실행하는 것
- ✓ 배치 프로세스는 API와 비교했을 때 처리하는 데이터 양이 많은데 예약 정보 전파 와 호텔을 예약하는 API를 비교해 보면 API는 한 사용자가 예약한 정보를 처리하므로 API가 한 번에 처리하는 데이터 양도 예약 정보의 하나지만 배치 프로세스는 하루 동안에 예약된 모든 호텔 정보를 조회하고 이를 분류하고 호텔 별로 다시 재처리하는 작업이므로 API보다 데이터 양이 더 많음
- ✓ 배치 프로세스 중에서도 처리하는 데이터 크기가 비교적 작고 처리 주기가 비교적 짧다면 이를 마이크로 배치라고 하는데 작은 배치를 여러 번 실행하여 원하는 데이터를 뽑는 형태로 웰컴 쿠폰 발송 기능은 10분 간격으로 10분 동안 가입한 사용자의 데이터를 처리하므로 마이크로 배치라고 할 수 있음

Spring Batch

❖ Batch Processing

- ✓ 스프링 프로젝트에서는 배치 프로세스를 위한 여러 가지 기능과 프레임워크를 제공
- ✓ 스프링 배치 프레임워크는 다수의 데이터를 체계적으로 처리하는 기능을 제공
- ✓ Spring Batch를 사용하는 이유
 - 유지 보수 측면 - 테스트 용이성이 추상화 와 같은 이점을 얻을 수 있도록 설계
 - 유연성 - POJO로 업무 로직을 구현하면 해당 로직을 웹 애플리케이션이나 배치 처리에서 사용 가능
 - 커뮤니티 지원
 - 비용

Spring Batch

❖ Batch Processing

✓ Spring Batch 사용 사례

- ETL(추출extract, 변환transform, 적재load)
 - ◆ 한 형식에서 다른 형식으로 데이터를 전환하는 작업은 엔터프라이즈 데이터 처리 에서 많은 부분을 차지
 - ◆ 스프링 배치의 청크 기반 처리 및 압도적인 확장 기능은 ETL 워크로드에 자연스럽게 들어맞음
- Data Migration
 - ◆ 간단한 배치 Job을 기동하고 수행하는 데 많은 코딩이 필요없으며 데이터 마이그 레이션을 수행할 때 당연히 포함해야 할 기능 임에도 대부분 제대로 구현하지 않는 커밋 횟수 측정이나 롤백 기능 등을 제공
- 워크로드 조정
 - ◆ 일반적으로 기업의 배치 처리에 단 하나의 Step만 존재하지는 않으며 분리된 많은 Step을 조정해야 하는데 어떤 파일을 로드 한 다음에 해당 데이터를 이용해 두 가지 독립적인 처리를 수행하고 뒤이어 결과를 하나로 내보낼 수도 있는데 스프링 배치는 이러한 태스크를 아주 잘 조정
 - ◆ Spring Cloud Data Flow 와 스프링 배치를 사용한 composed task 처리는 좋은 예인데 여기서 스프링 배치는 다른 기능을 실행하기 위해 스프링 클라우드 데이터 플로우를 호출하며 무엇을 수행했고 무엇을 수행해야 하는지 추적

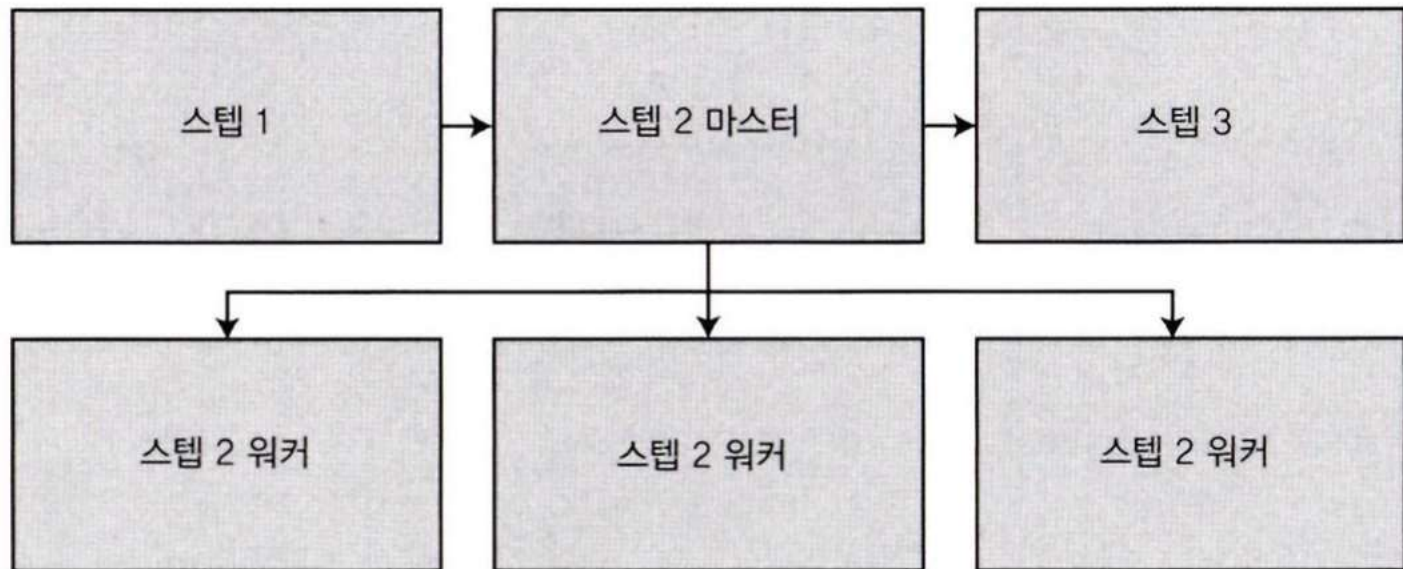
Spring Batch

❖ Batch Processing

✓ Spring Batch 사용 사례

● 병렬 처리

- ◆ Apache Spark, YARN, GridGain, Hazelcast 등과 같은 대부분의 빅 데이터 플랫폼은 최근 멀티 코어 프로세서 와 클라우드로 사용 가능한 한 수많은 서버를 활용 하려 했으나 아파치 스파크와 같은 프레임워크를 사용하려면 알고리즘 또는 데이터 구조에 맞게 코드와 데이터를 변경해야 하는데 스프링 배치는 멀티 코어 또는 멀티 서버에 처리를 분산하는 기능을 제공하며 웹 애플리케이션에서 사용하는 것과 동일한 객체 및 데이터 소스에 접근하는 기능을 제공

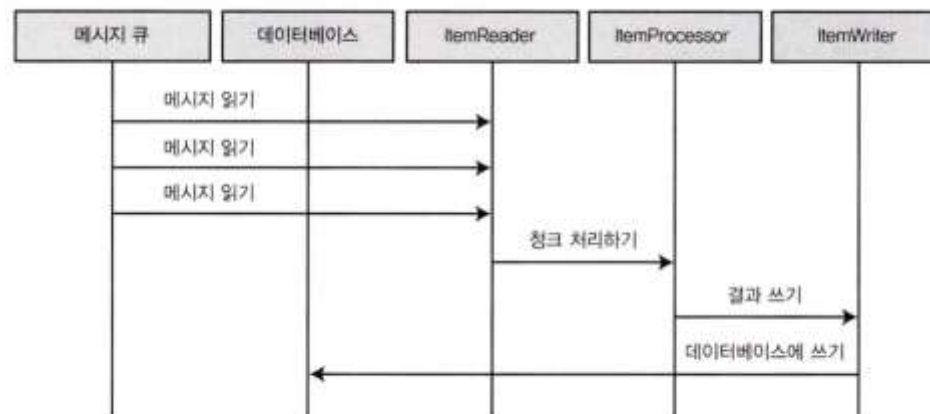


Spring Batch

❖ Batch Processing

✓ Spring Batch 사용 사례

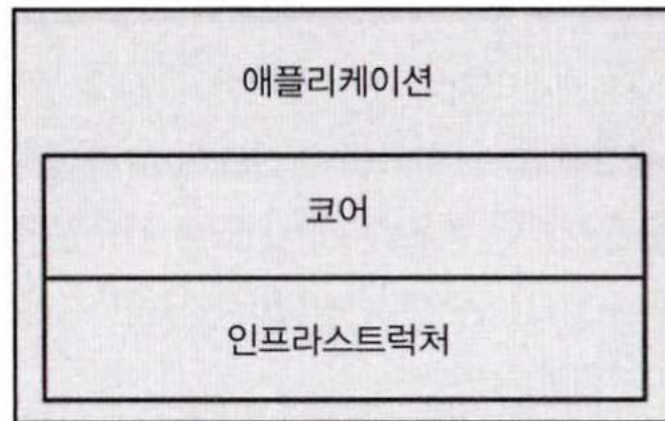
- 무중단 처리 또는 상시 데이터 처리
 - ◆ 시스템이 완전히 일정하거나 또는 거의 일정하게 데이터를 입력받는 사례는 많은데 데이터 누락을 방지하려면 데이터가 들어 오는 속도에 맞게 받아져야 하지만 해당 데이터 처리 시에는 받아들인 데이터를 한 번에 처리할 청크 단위로 모아서 배치 처리를 하는 것이 성능면에서 더 나을 수 있음
 - ◆ 스프링 배치는 신뢰할 수 있고 확장 가능한 방식으로 이런 유형의 처리를 가능하게 하는 도구를 제공
 - ◆ 프레임워크의 기능을 사용하면 queue에서 메시지를 읽은 뒤 청크 단위로 배치 처리를 수행하는 과정을 끝없이 반복할 수 있는데 이러한 솔루션을 처음부터 개발해야 하는 복잡한 상황을 이해할 필요없이 대량 데이터 처리량을 늘릴 수 있음



Spring Batch

❖ Spring Batch Framework

- ✓ 스프링 배치 프레임워크는 일반적인 배치 패턴 및 패러다임을 구현하는 표준 기반 방법으로 액센츄어사 와 스프링 소스 사 간의 협 업으로 개발
- ✓ 스프링 배치에는 데이터 유효성 검증, 출력 포매팅, 복잡한 비즈니스 규칙을 재사용 가능한 방식으로 구현하는 기능, 대규모 데이터 셋 처리 기능이 구현돼 있음
- ✓ 스프링 배치 Architecture



Spring Batch

❖ Spring Batch Framework

✓ 스프링 배치 Architecture

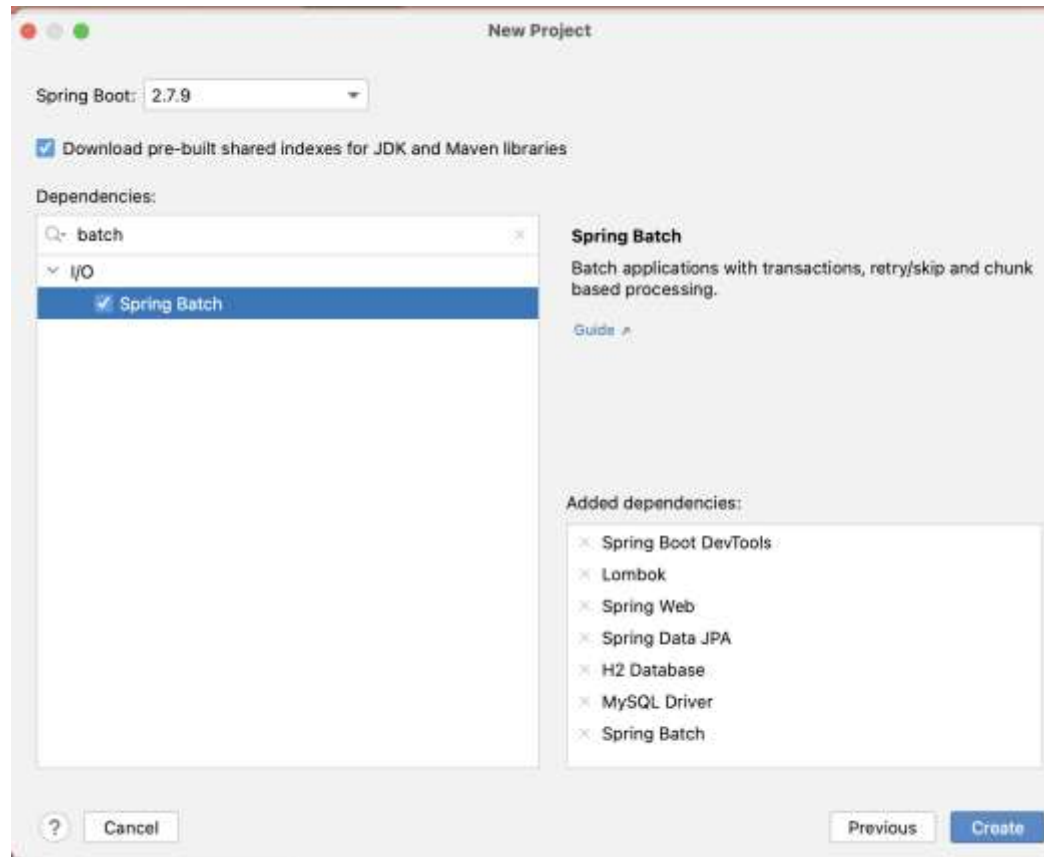
- 스프링 배치는 레이어 구조로 조립된 세 개의 티어로 구성
- 가장 바깥쪽에 애플리케이션 레이어가 있으며 배치 처리 구축에 사용되는 모든 사용자 코드 및 구성이 이 레이어에 포함되고 업무 로직, 서비스 등은 물론 Job 구조화와 관련된 구성까지도 애플리케이션 레이어에 포함되며 애플리케이션 레이어가 최상위에 있는 것이 아니라 다른 두 레이어인 코어 레이어와 인프라스트럭처 레이어를 감싸고 있는데 그 이유는 개발자가 개발하는 대부분의 코드가 코어 레이어와 함께 동작하는 애플리케이션 레이어지만 때로는 커스텀 리더 나 커스텀 라이터 와 같이 인프라스트럭처의 일부를 만들기도 하기 때문
- 애플리케이션 레이어는 다음 레이어인 코어 레이어와 상호 작용하는데 대부분의 시간을 소비
- 코어 레이어에는 배치 도메인을 정의하는 모든 부분이 포함되고 코어 컴포넌트의 요소에는 Job 및 Step 인터페이스와 Job 실행에 사용되는 인터페이스 인 JobLauncher 및 JobParameters 등이 있음
- 이 모든 것의 밑에는 인프라스트럭처 레이어가 있는데 어떤 처리를 수행하려면 파일, 데이터베이스 등으로부터 읽고 쓸 수 있어야 하고 Job 수행에 실패한 이후 재시도될 때 어떤 일을 수행할지를 다룰 수 있어야 하는데 이러한 부분은 공통 인프라스트럭처로 간주되며 프레임워크의 인프라스트럭처 컴포넌트 내에 들어 있다.

Spring Batch

❖ Spring Batch Framework

✓ 프로젝트 생성 및 실행 - BatchWeb

- Spring Boot Version – 2.7.9
- 의존성: Batch, Web, JPA, MySQL, DevTools, Lombok



Spring Batch

❖ Spring Batch Framework

✓ 프로젝트 생성 및 실행 - BatchWeb

● application.yml 파일 생성 및 설정

server:

port: 80

spring:

datasource:

url: jdbc:mysql://localhost:3306/adam?useSSL=false&characterEncoding=UTF-8&serverTimezone=UTC

driver-class-name: com.mysql.cj.jdbc.Driver

username: root

password: wnddkd

jpa:

hibernate:

ddl-auto: update

properties:

hibernate:

format_sql: true

show_sql: true

use_sql_comments: true

Spring Batch

❖ Spring Batch Framework

✓ 프로젝트 생성 및 실행 - BatchWeb

● application.yml 파일 생성 및 설정

#Live Reload

devtools:

livereload:

enabled: true

logging:

level:

org.hibernate.type.descriptor.sql: trace

Spring Batch

- ❖ Spring Batch Framework
 - ✓ 프로젝트 생성 및 실행 - BatchWeb
 - 프로젝트 실행



Spring Batch

❖ 데이터베이스 접속 정보 암호화

- ✓ 암호화를 위한 jasypt 라이브러리의 의존성 설정

implementation 'com.github.ulisesbocchio:jasypt-spring-boot-starter:3.0.4'



Spring Batch

❖ 데이터베이스 접속 정보 암호화

- ✓ 암호화를 위한 jasypt 라이브러리 설정 클래스 생성 – config.JasyptConfig

@Configuration

public class JasyptConfig {

private static final String KEY = "adamsoft";

private static final String ALGORITHM = "PBESWithMD5AndDES";

@Bean("jasyptStringEncryptor")

public StringEncryptor stringEncryptor() {

PooledPBESStringEncryptor encryptor = new PooledPBESStringEncryptor();

SimpleStringPBEConfig config = new SimpleStringPBEConfig();

config.setPassword(KEY); // 암호화할 때 사용하는 키

config.setAlgorithm(ALGORITHM); // 암호화 알고리즘

config.setKeyObtentionIterations("1000"); // 반복할 해싱 회수

config.setPoolSize("1"); // 인스턴스 pool

config.setProviderName("SunJCE"); // 프로바이더

config.setSaltGeneratorClassName("org.jasypt.salt.RandomSaltGenerator"); // salt

생성 클래스 지정

config.setStringOutputType("base64"); // 인코딩

encryptor.setConfig(config); // 설정 정보 Set

return encryptor;

}

}

Spring Batch

❖ 데이터베이스 접속 정보 암호화

- ✓ jasypt 온라인 복호화 사이트에서 암호화: <https://www.devglan.com/online-tools/jasypt-online-encryption-decryption>

Jasypt Online Encryption	Jasypt Online Decryption
Enter plain text to Encrypt	Enter the Jasypt Encrypted Text
root	yePyJ1GNSG57IlzXjP4Ywg
Select Type of Encryption	Select Action Type
Two Way Encryption(With Secret Text)	Decrypt Password
Secret Key To Be Used While Encryption	Secret Key Used during Encryption(Optional)
adamsoft	adamsoft
Encrypt	Match/Decrypt
Encrypted String:	Result:
YFbVNdJin+MJ3+mG6j0i3g==	Invalid encryption key.

Spring Batch

❖ 데이터베이스 접속 정보 암호화

- ✓ application.yml 파일 수정 후 실행

spring:

datasource:

url: jdbc:mysql://localhost:3306/adam?useSSL=false&characterEncoding=UTF-8&serverTimezone=UTC

driver-class-name: com.mysql.cj.jdbc.Driver

username: ENC(YFbVNdJin+MJ3+mG6j0i3g==)

password: ENC(yePyJ1GNSG57I1zXjP4Ywg==)

Spring Batch

❖ 데이터베이스 접속 정보 암호화

- ✓ 원본 데이터 복원을 위한 서비스 클래스 생성 - JasyptService

```
@Component
public class JasyptService {

    @Value("${spring.datasource.username}")
    private String propertiesTestValue;

    public String getPropertiesTestValue() {
        return propertiesTestValue;
    }
}
```

Spring Batch

❖ 데이터베이스 접속 정보 암호화

- ✓ 원본 데이터 복원을 위한 테스트 클래스 생성 – JasyptServiceTests

```
@SpringBootTest
class JasyptTests {

    @Autowired
    private JasyptService jasyptService;

    @Test
    void jasypt_테스트() {
        assertThat("root", is(equalTo(jasyptService.getPropertiesTestValue())));
    }
}
```

Spring Batch

❖ 스케줄링 설정

✓ SchedulingConfigurer를 사용한 TaskScheduler 설정

- 스프링 프레임워크에서는 스케줄링 기능을 확장하는 `org.springframework.scheduling.annotation.SchedulingConfigurer` 인터페이스를 제공하는데 `configureTasks()` 메서드는 `ScheduledTaskRegistrar`를 인자로 받는데 `ScheduledTaskRegistrar`는 `TaskScheduler`를 설정하는 헬퍼 클래스
- 스프링 프레임워크가 스케줄링 기능을 활성화 할 때 `SchedulingConfigurer`에 설정된 `TaskScheduler` 객체를 확인하고 설정된 `TaskScheduler`가 있다면 이를 사용하여 스케줄링 기능을 활성화
- 자바 설정 클래스는 `SchedulingConfigurer` 인터페이스를 구현하도록 설계하고 인터페이스의 `configureTask()` 메서드를 구현하는데 메서드의 인자 `ScheduledTaskRegistrar` 객체에 `TaskScheduler` 구현체를 설정하면 구현체의 스레드를 사용하여 스케줄링 태스크를 실행

Spring Batch

❖ 스케줄링 설정

✓ SchedulingConfigurer를 사용한 TaskScheduler 설정

- SchedulingConfigurer 인터페이스를 구현한 클래스 – config.SchedulingConfig

@EnableScheduling

@Configuration

public class SchedulingConfig implements SchedulingConfigurer {

@Override

public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {

 ThreadPoolTaskScheduler taskScheduler = new ThreadPoolTaskScheduler();

 taskScheduler.setPoolSize(10);

 taskScheduler.setThreadNamePrefix("TaskScheduler-");

 taskScheduler.initialize();

 taskRegistrar.setTaskScheduler(taskScheduler);

}

}

Spring Batch

❖ 스케줄링 설정

- ✓ SchedulingConfigurer를 사용한 TaskScheduler 설정
 - SchedulingConfigurer 인터페이스를 구현한 클래스 – config.SchedulingConfig
 - ◆ SchedulingConfig 자바 설정 클래스는 SchedulingConfigurer를 구현
 - ◆ SchedulingConfigurer의 configureTasks() 메서드 인자 ScheduledTaskRegistrar 객체에 TaskScheduler 구현체를 설정하는데 이때 ScheduledTaskRegistrar의 setTaskScheduler() 메서드를 사용
 - ◆ 두 개 이상의 태스크를 동시에 실행해야 하므로 ThreadPoolTaskScheduler 구현 클래스를 사용
 - ◆ setPoolSize() 메서드는 ThreadPoolTaskScheduler의 스레드 개수를 설정하는데 스레드 이름의 머리말을 설정하려면 setThreadNamePrefix() 메서드를 사용
 - ◆ 설정을 마치면 TaskScheduler의 initialize() 메서드를 사용하여 객체를 초기화

Spring Batch

❖ 스케줄링 설정

✓ SchedulingConfigurer를 사용한 TaskScheduler 설정

● 태스크 설정 – service.scheduledTask

@Slf4j

@Component

public class ScheduledTask {

 @Scheduled(fixedRate= 1000L)

 public void triggerEvent() {

 log.info("1초마다 배치 작업 수행");

 }

}

◆ @Scheduled 애너테이션의 fixedRate는 설정된 간격으로 스케줄링 태스크를 실행하는데 1초(1000밀리초)를 설정했으므로 1초 간격으로 태스크가 실행

◆ @Scheduled 애너테이션이 정의된 triggerEvent() 메서드는 스케줄링 태스크로 실행되는데 1초 간격으로 계속해서 triggerEvent() 메서드가 실행

Spring Batch

❖ 스케줄링 설정

✓ SchedulingConfigurer를 사용한 TaskScheduler 설정

- @Scheduled 애너테이션을 사용하는 규칙
 - ◆ 스케줄 태스크를 정상적으로 실행하려면 스프링 빈의 메서드에 정의
 - ◆ 정의된 메서드의 리턴 타입은 void이며 인자를 정의하면 안됨
 - ◆ 메서드의 접근 제어자는 public
- 내용이 복잡해지면 비즈니스 로직을 실행하는 클래스에 기능을 위임할 수 있는데 ScheduledTask 클래스가 스프링 빈이므로 다른 스프링 빈을 주입받아 사용

Spring Batch

❖ 스케줄링 설정

✓ SchedulingConfigurer를 사용한 TaskScheduler 설정

- 실행 주기를 설정하는 방법 - @Scheduled 어노테이션

```
package org.springframework.scheduling.annotation;
```

```
@Target({ElementType.METHOD, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Repeatable(org.springframework.scheduling.annotation.Schedules.class)
public @interface Scheduled {
    String cron() default "";
    long fixedDelay() default -1;
    String fixedDelayString() default "";
    long fixedRate() default -1;
    String fixedRateString() default "";
    long initialDelay() default -1;
    TimeUnit timeUnit() default TimeUnit.MILLISECONDS;
    String zone() default "";
}
```

Spring Batch

❖ 스케줄링 설정

✓ SchedulingConfigurer를 사용한 TaskScheduler 설정

● 실행 주기를 설정하는 방법 - @Scheduled 어노테이션

- ◆ 스케줄링 설정 속성으로, 크론 표현식(cron expression)을 사용하여 스케줄링을 설정
- ◆ 스케줄링 설정 속성으로 fixedDelay에 설정된 시간만큼 태스크와 태스크 사이의 지연 시간을 설정할 수 있는데 실행한 태스크가 종료되면 설정된 시간만큼 쉬고 다음 태스크를 실행하는데 long 값으로 설정할 때는 fixedDelay 속성을 사용하고 String 값으로 설정할 때는 fixedDelayString 속성을 사용
- ◆ 스케줄링 설정 속성으로 fixedRate에 설정된 시간만큼 태스크와 태스크 사이의 실행 주기를 설정할 수 있는데 태스크를 실행하고 설정된 시간 후 다시 태스크를 실행하는데 long 값으로 설정할 때는 fixedRate 속성을 사용하고 String 값으로 설정할 때는 fixedRateString 속성을 사용
- ◆ 스케줄링 설정 속성으로 스케줄링 기능이 초기화된 후 initialDelay에 설정된 시간만큼 쉬고 첫 태스크를 실행하는데 초기 지연 값이므로 두 번째 실행하는 태스크부터는 영향이 없고 fixedDelay와 fixedRate 속성을 조합하여 사용
- ◆ @Scheduled에 설정된 기본 시간 단위는 밀리초로. 이를 변경할 때 사용하는 속성으로 TimeUnit에 설정된 상수를 사용하여 설정
- ◆ 스케줄링할 수 있는 타임존 값을 설정하는데 설정하지 않으면 서버에 설정된 타임존 값을 사용하고 한국의 타임존 값은 Asia/Seoul이며 설정된 타임존 값은 java.util.TimeZone 클래스에 getTimeZone() 메서드를 사용하여 변경

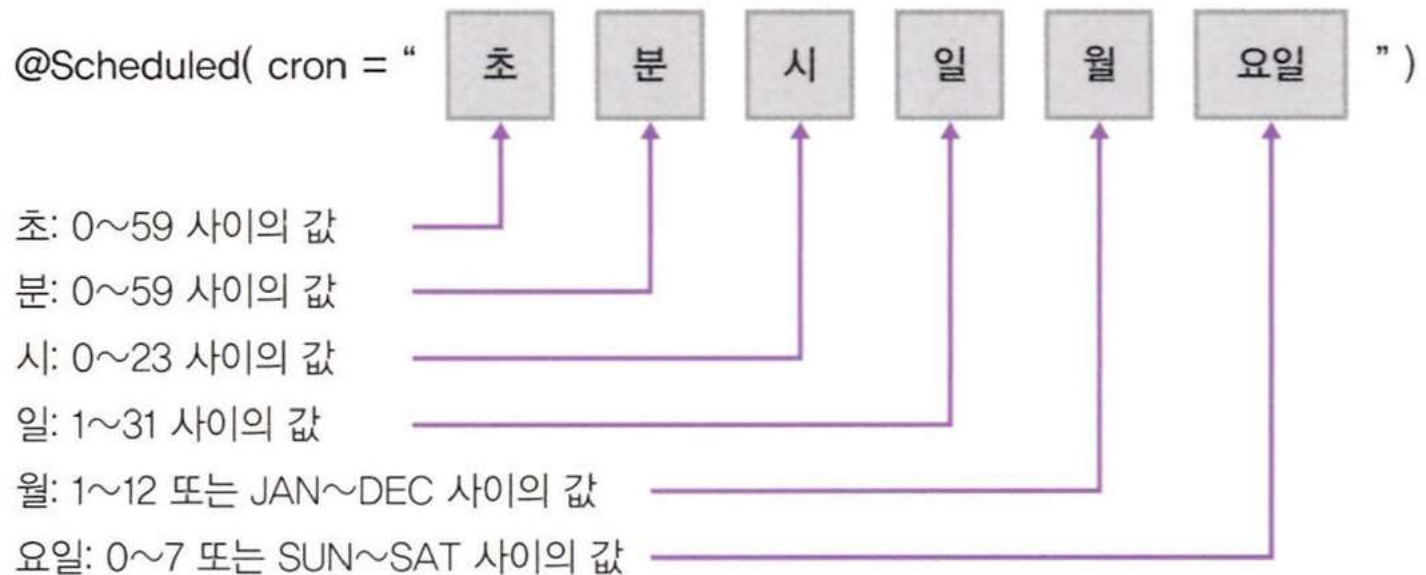
Spring Batch

❖ 스케줄링 설정

✓ SchedulingConfigurer를 사용한 TaskScheduler 설정

● cron 속성 표현식

◆ 표현식의 자리 의미



Spring Batch

❖ 스케줄링 설정

✓ SchedulingConfigurer를 사용한 TaskScheduler 설정

● cron 속성 표현식

◆ 표현식의 자리 의미

- 크론 표현식은 총 여섯 자리로 구성
- 왼쪽부터 첫 번째 자리는 초를 의미하고 마지막 여섯 번째 자리는 요일을 의미
- 기본적으로 각 자리에는 숫자를 사용하여 설정할 수 있으며 자릿수에 따라 시간을 나타내는 문자열이나 크론 표현식 문자를 사용할 수 있음
- 달을 의미하는 자리에는 1월부터 12월을 의미하는 숫자를 입력할 수 있으며 숫자 대신 JAN부터 DEC까지 문자열을 입력해도 되고 요일을 의미하는 자리에는 0 부터 7까지 숫자를 입력할 수 있는데 일요일은 0 이나 7을 의미하며 차례대로 월요일은 1, 토요일은 6을 설정하는데 SUN 부터 SAT 까지 문자열 입력 가능

Spring Batch

❖ 스케줄링 설정

✓ SchedulingConfigurer를 사용한 TaskScheduler 설정

● cron 속성 표현식

◆ 표현식 문자

- *: 모든 값
- ?: 어떤 값이든 상관 없음
- ,: 배열 설정 - 예) 0, 1, 2, 3
- -: 범위 설정 - 예) 10-20(10에서 20 사이)
- /: 초깃값 과 증분값 설정 - 예) 0/10(초깃값0,증분값10)
- #: 요일과 몇 번째를 설정 - 예) 5#2(목요일#두 번째)
- L: 마지막
- W: 스케줄링된 날부터 가까운 평일

Spring Batch

❖ 스케줄링 설정

✓ SchedulingConfigurer를 사용한 TaskScheduler 설정

- cron 속성 표현식

- ◆ 표현식 예제

- @Scheduled(cron="* 0/10 * * * ?"); 10분마다 태스크를 스케줄링하여 실행
- @Scheduled(cron="0 0 2 * * ?"); 매일 새벽 2시 정각마다 태스크를 스케줄링하여 실행
- @Scheduled(cron="0 0 6 * * SUN"); 매주 일요일 오전 6시 정각마다 태스크를 스케줄링하여 실행
- @Scheduled(cron="0 0 1, 2, 3 ? * MON"); 매주 월요일 새벽 1시 00분, 2시 00분, 3시 00분 에 태스크를 스케줄링하여 실행

Spring Batch

❖ 스케줄링 설정

✓ SchedulingConfigurer를 사용한 TaskScheduler 설정

● cron 속성 표현식

◆ 테스트 클래스를 이용해서 확인

```
public class CronExpressionTest {

    @Test
    public void testParse() {
        CronExpression expression = CronExpression.parse("0/5 * * * * ?");
        LocalDateTime nextScheduled =
            expression.next(LocalDateTime.of(2023, 1, 1, 0, 0, 0));

        Assertions.assertEquals("2023-01-01T00:00:05",
            nextScheduled.toString());
        Assertions.assertEquals("2023-01-01T00:00:10",
            expression.next(nextScheduled).toString());
    }
}
```

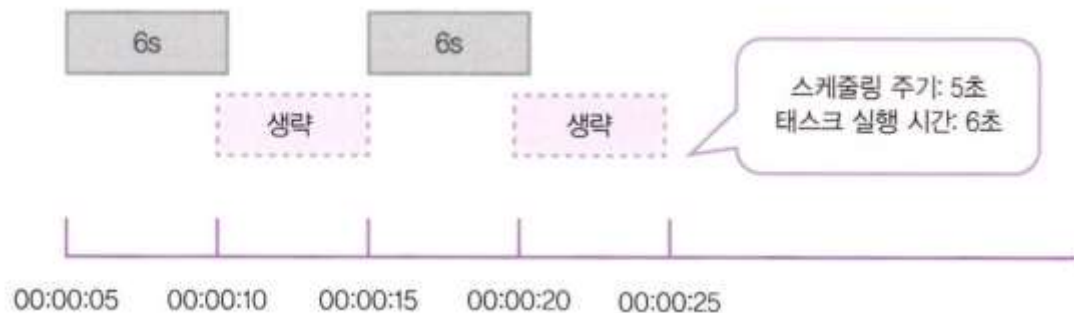
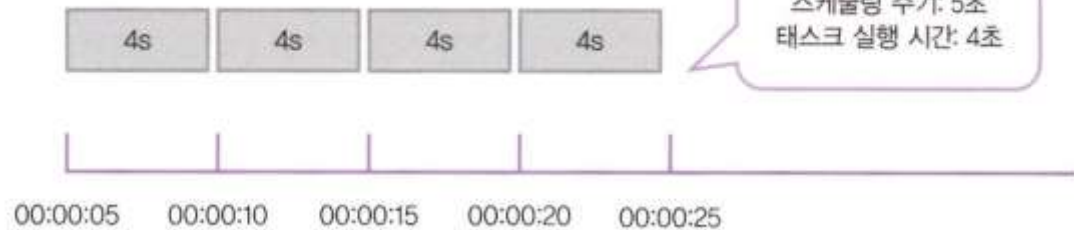
Spring Batch

❖ 스케줄링 설정

✓ SchedulingConfigurer를 사용한 TaskScheduler 설정

- cron 속성 표현식

@Scheduled(cron="0/5 * * * * ?")



Spring Batch

❖ 스케줄링 설정

✓ SchedulingConfigurer를 사용한 TaskScheduler 설정

● 예시

◆ 수행할 태스크

- 신규 가입한 사용자에게 웰컴 쿠폰을 보내는 태스크를 작성
- @Scheduled(cron="0 0/5 * * * ?") 어노테이션을 사용하여 5분 간격으로 태스크를 실행
- User 테이블에는 created_at과 welcome flag 필드를 포함하고 있는데 created_at은 가입한 시간을 저장하고 welcome_flag는 웰컴 메일 송신 여부를 저장

Spring Batch

❖ 스케줄링 설정

✓ SchedulingConfigurer를 사용한 TaskScheduler 설정

● 예시

◆ 수행할 태스크

// 장애에 취약한 코드

```
LocalDateTime from = LocalDateTime.now().withSecond(0).minusMinutes(5);
```

```
LocalDateTime to = LocalDateTime.now().withSecond(0);
```

```
userRepository.findByCreatedAtBetween(from, to);
```

// 장애에 견고한 코드

```
LocalDateTime to = LocalDateTime.now().withSecond(0);
```

```
WelcomeFlag flag = WelcomeFlag.UNSENT;
```

```
Integer fetchCount = 1000;
```

```
userRepository.findByCreatedBeforeAndWelcomeFlagAndSize(toz flag,  
    fetchCount);
```

Spring Batch

❖ 스케줄링 설정

✓ SchedulingConfigurer를 사용한 TaskScheduler 설정

● 예시

◆ 수행할 태스크

- 장애에 취약한 코드는 지난 5분간 데이터만 조회하는 로직으로 개발되어 있어서 00:10:00에 실행 해야 하는 태스크가 생략되었다고 가정하면 00:05~00:10 사이에 가입한 사용자에게는 웰컴 메일을 전송하지 못하고 다음 스케줄인 00:15:00에 실행하는 태스크는 00:10~00:15 사이에 가입한 사용자만 조회한 후 웰컴 메일을 전송하게 되서 결국 몇몇 사용자에게는 이메일을 전송하지 못하는 일이 발생할 수 있음
- 장애에 견고한 코드는 현재 시간보다 작고 WelcomeFlag가 UNSENT인 사용자들만 조회해서 태스크 시스템에 장애가 10시간 동안 발생하고 그 사이에 가입한 사용자가 1만 명이라고 가정하면 fetchcount 인자가 없다면 findBy() 메서드는 한 번에 1만 명을 쿼리할 것이고 1만 명의 데이터를 동시에 처리하면 다시 장애가 발생할 수 있으므로 5분 동안 처리할 수 있는 임의의 숫자를 넣어 적절히 처리하는 것이 중요

Spring Batch

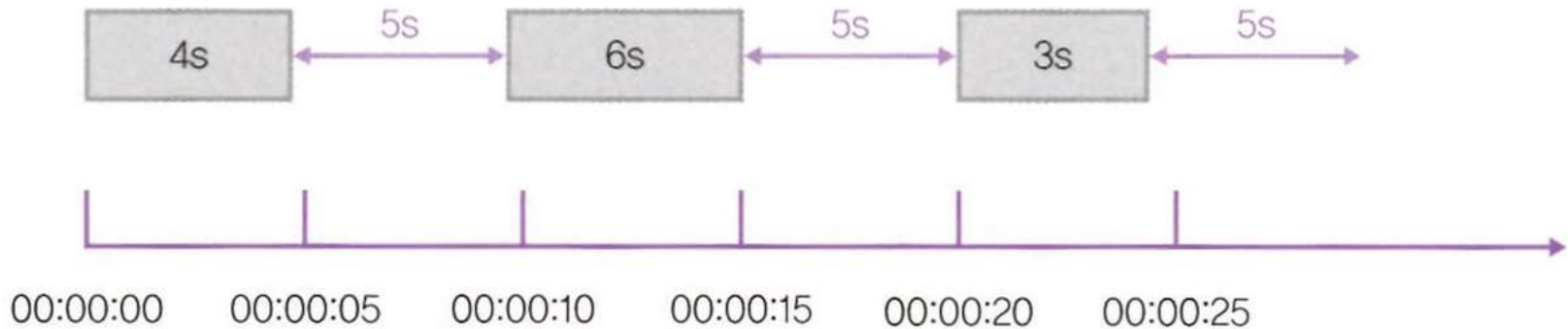
❖ 스케줄링 설정

✓ SchedulingConfigurer를 사용한 TaskScheduler 설정

● fixedDelay

- ◆ fixedDelay 속성은 태스크와 태스크 사이의 시간 간격을 설정하는 데 사용
- ◆ 태스크를 실행한 후 설정된 시간이 지나면 다른 태스크를 실행

@Scheduled(fixedDelay=5000)



Spring Batch

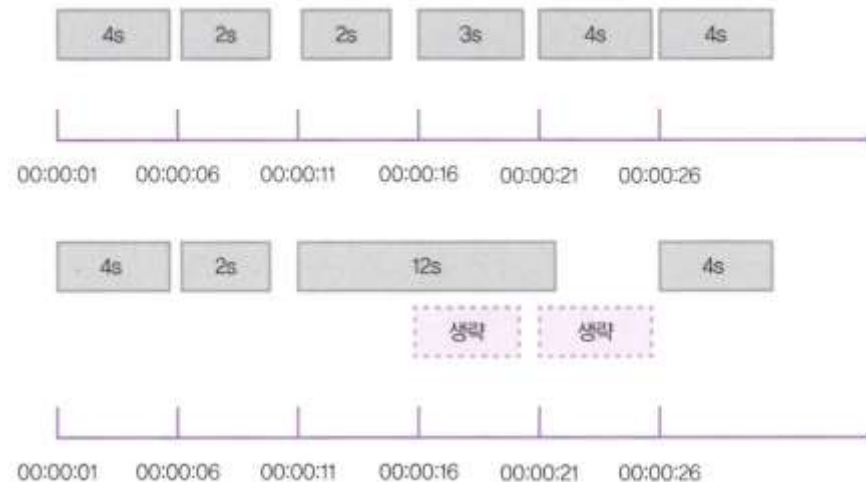
❖ 스케줄링 설정

✓ SchedulingConfigurer를 사용한 TaskScheduler 설정

● fixedRate

- ◆ fixedRate 속성은 설정된 시간 주기에 맞추어 태스크를 실행
- ◆ cron은 정해진 시각에 태스크를 실행하고 fixedRate는 애플리케이션이 시작한 후 5초 간격으로 태스크를 실행
- ◆ fixedRate는 cron처럼 정해진 시각에 태스크를 실행할 수 없음

@Scheduled(fixedRate= 5000)



Spring Batch

❖ 배치 서버 Architecture

- ✓ 일반적으로 배치 프로세스 대상 데이터의 크기가 매우 크거나 여러 가지 태스크를 복합적으로 결합하여 데이터를 가공할 때는 시스템 리소스가 많이 필요하므로 반드시 배치 프로세싱을 위한 서버군을 구성하여 처리
- ✓ 태스크들이 복합적으로 결합해서 데이터를 가공할 때는 태스크 간 의존성 이 있는 경우로 앞선 태스크가 처리한 데이터를 재가공해서 새로운 데이터를 뽑아내는 태스크는 태스크 사이에 의존성이 있다고 하는데 이때는 일반 스프링 프레임워크를 사용하는 것보다 스프링 배치 프레임워크를 사용하는 것이 좋은데 스프링 배치 프레임워크는 배치 프로세싱에 적합한 애플리케이션 구조와 기능들을 제공
- ✓ 비교적 데이터 크기가 작은 마이크로 배치 프로세싱을 대상으로 하는 경우 별도의 배치 서버군을 구성해서 배치 프로세싱을 서비스에서 격리해도 되지만 별도의 서버군을 구축하는 것이 오히려 시스템 낭비일 정도로 작은 배치 프로세스는 REST-API 서버군에서 실행해도 되는데 한 코드 베이스에 API 와 배치 프로세스가 모두 포함되어 있으므로 전체 비즈니스 로직 을 파악하기에는 유리한 장점도 있음
- ✓ 배치 서버를 독립적으로 구성하여 배치 프로세스를 실행하든 API 서버에서 실행하든 간에 장애에 대비하는 것이 필요한데 고가용성을 위해서는 두 대 이상의 서버가 필요
- ✓ 두 대 이상의 배치 서버를 구성할 때 주의할 점은 중복 실행인데 A 서버에서 실행한 태스크를 B 서버에서 동시에 실행하면 데이터가 중복 처리될 가능성이 있으며 동시에 실행한 배치 프로세스는 공유 자원에 데드 락을 발생할 수 있기 때문에 데드락을 회피해야 함

Spring Batch

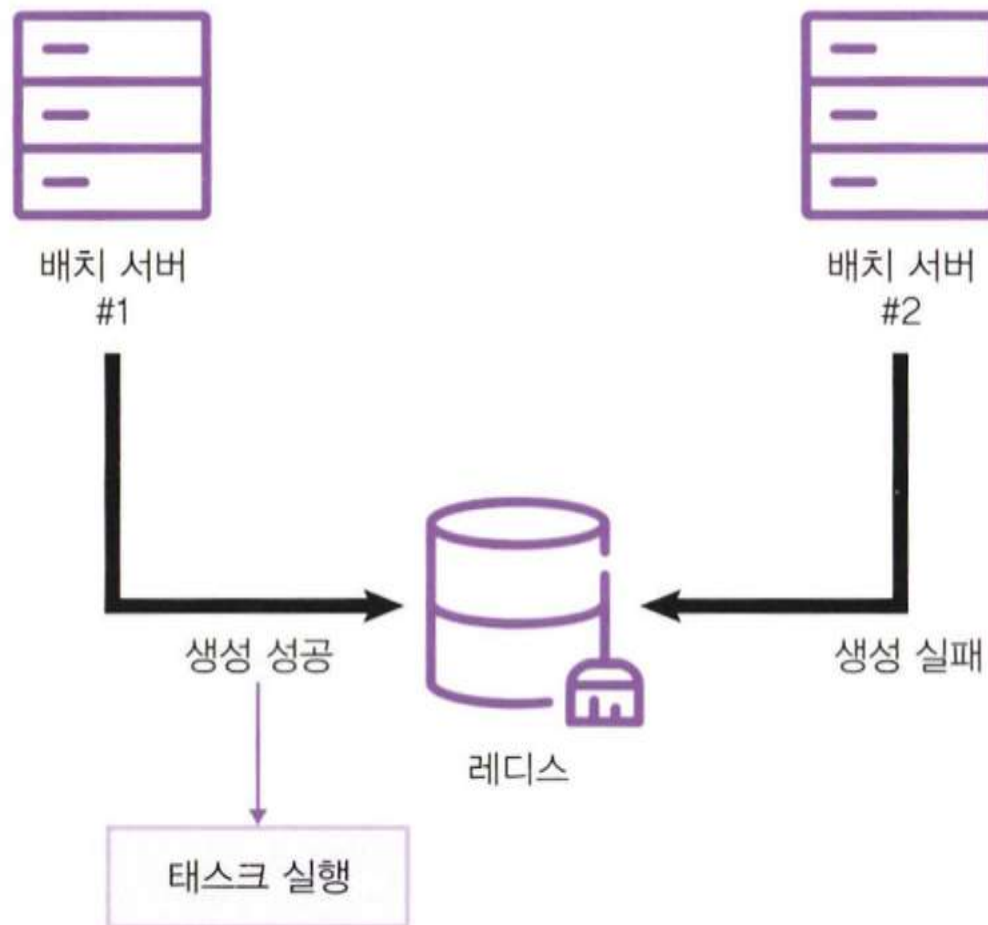
❖ 배치 서버 Architecture

✓ 데드락 회피 방법

- 모든 서버 군의 서버는 NTPD(네트워크 시간 프로토콜 데몬은 네트워크 시간 프로토콜을 사용하여 시간 서버와 동기화하여 시스템 시간을 유지 관리하는 운영 체제 프로그램)를 사용하여 서버 시간을 동기화
- 배치 프로세스의 스케줄링은 @Scheduled 애너테이션의 cron 속성을 사용하여 실행
- 서버 군에 포함된 모든 서버의 배치 프로세스는 모두 같은 시간에 실행
- 중복 실행을 막고자 레디스의 분산 락을 사용하여 가장 먼저 실행된 배치 프로세스가 데이터를 처리
- 장애에 견고한 코드를 작성하여 실행되지 못한 태스크는 자동으로 복구할 수 있도록 작성
- 짧은 주기의 배치 프로세스가 아니라면 수동으로 복구할 수 있는 기능을 개발

Spring Batch

- ❖ 배치 서버 Architecture
 - ✓ 단독 배치 서버

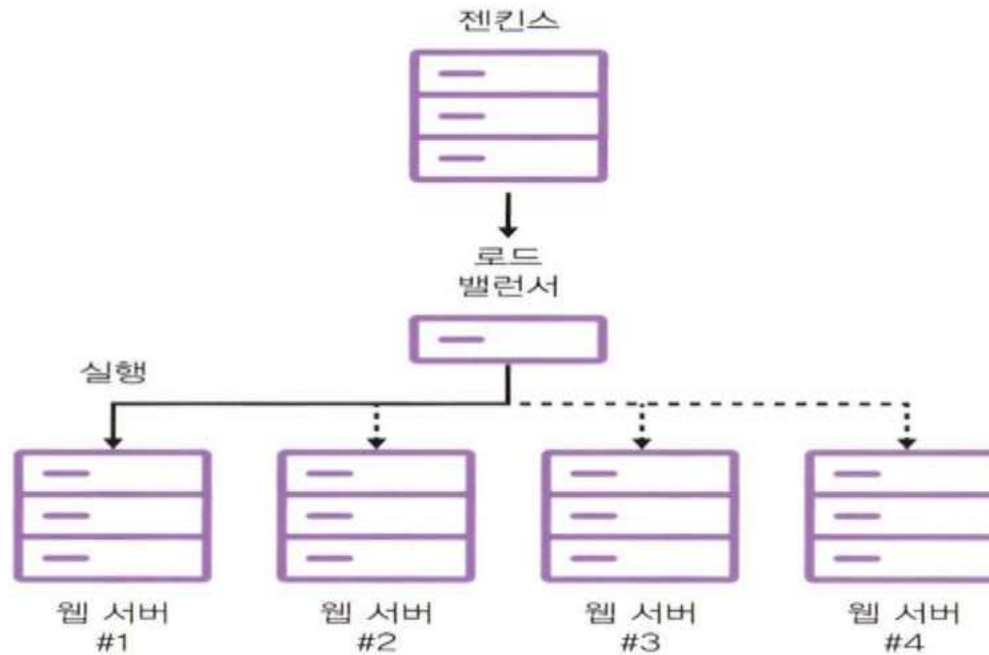


Spring Batch

❖ 배치 서버 Architecture

✓ Jenkins 와 REST-API 서버군

- 젠킨스는 CI/CD 툴로 알려져 있지만 스케줄링과 각종 명령어를 실행할 수 있음
- 젠킨스를 이용하면 스프링 프레임워크의 @Scheduled 기능없이 배치 프로세싱을 구현할 수 있음



Spring Batch

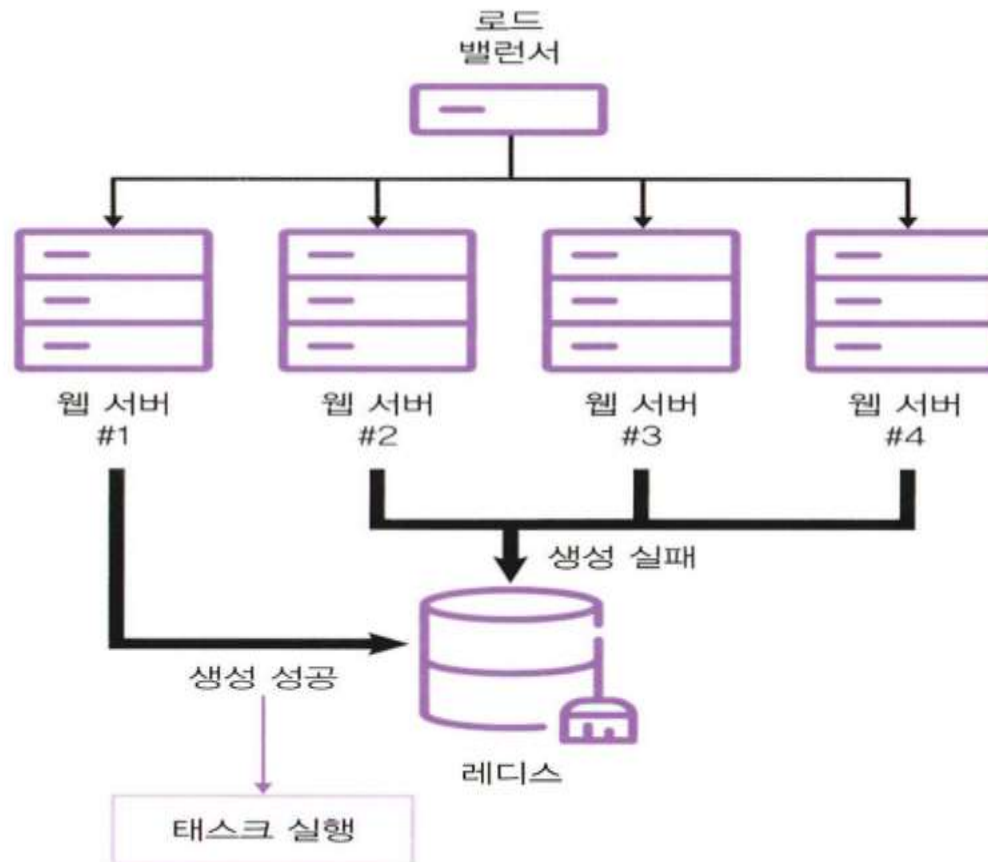
❖ 배치 서버 Architecture

✓ Jenkins 와 REST-API 서버군

- 모든 웹 서버는 REST-API를 제공하고 해당 REST-API를 호출하면 특정 태스크를 실행
- 젠킨스는 REST-API를 호출할 수 있는 기능을 제공하며 REST-API를 호출하는 스케줄을 설정할 수 있으며 이외에도 REST-API 실행 결과를 저장하고 조회할 수 있는 장점도 있기 때문에 젠킨스를 사용하면 @Scheduled 애너테이션 기능처럼 원하는 주기에 원하는 기능을 실행할 수 있음
- 젠킨스는 주기적으로 로드 밸런서에 REST-API를 호출하고 로드 밸런서로 네 개의 서버 중 하나의 서버가 해당 요청을 처리
- REST-API를 제공하는 여러 서버가 로드 밸런서에 포함되어 있기 때문에 웹 서버 #1에 장애가 발생하면 다른 웹 서버들이 요청을 처리할 수 있는데 이처럼 로드 밸런서로 고가용성을 확보할 수 있으며 API 서버 외에 별도의 배치 서버를 구성하지 않아도 되는 장점이 있고 배치 서버를 위한 별도의 코드베이스를 구성하지 않고 하나의 코드베이스를 사용하여 API와 배치 기능을 개발할 수 있음
- 스케줄링과 REST-API 실행을 하는 외부 시스템인 젠킨스 서버도 장애가 발생할 수 있기 때문에 두 대 이상의 젠킨스 서버를 구축하여 고가용성을 확보해야 함
- 전체 시스템 구성상 REST-API 서버 외에도 추가적인 젠킨스 서버군이 필요하지만 이 구성에는 분산락을 생성 할 필요 없으므로 레디스 서버가 없어도 되는 특징이 있음

Spring Batch

- ❖ 배치 서버 Architecture
 - ✓ @Scheduled 와 REST-API 서버군



Spring Batch

❖ 배치 서버 Architecture

✓ @Scheduled 와 REST-API 서버군

- REST-API 서버에 @Scheduled 애너테이션을 사용하여 배치 프로세싱을 실행하는 구조
- 모든 웹 서버에 같은 주기와 같은 시간. 같은 태스크를 실행하므로 분산 락이 필요
- 단독 배치 서버 처럼 코드를 작성하면 되는데 REST-API 코드베이스에 배치 프로세싱 기능을 개발하여 배포하는 것이 다를 뿐
- 별도의 젠킨스 서버나 배치 서버가 필요하지 않지만 배치 프로세싱의 작업 크기가 커지면 특정 REST-API 서버에 부하가 걸릴 수 있다기 때문에 언제든지 별도의 배치 서버로 분리할 수 있도록 준비

Spring Batch

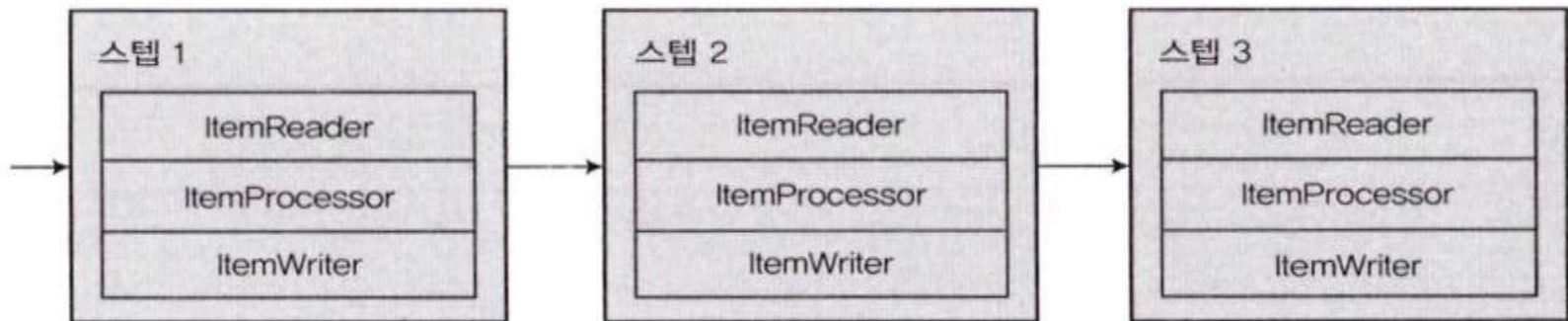
Spring Batch

❖ Spring Batch Framework 용어

✓ Job & Step

● Batch Job

- ◆ Job은 배치 처리 과정을 하나의 단위로 만들어 놓은 객체이며 배치 처리 과정에 있어 전체 계층 최 상단에 위치
- ◆ 자바나 XML을 사용해 구성된 배치 Job은 상태를 수집하고 이전 상태에서 다음 상태로 전환
- ◆ 개념적으로 스프링 배치 Job은 상태 기계에 지나지 않음
- ◆ 업무 시간 이후에 고객의 은행 계좌를 처리하는 사례를 보면 Step 1에서는 다른 시스템에서 수신한 거래 정보 파일을 읽어와 데이터베이스에 저장하고 Step 2에서는 모든 입금 정보를 계좌에 반영한 후 마지막으로 Step 3에서는 모든 출금 정보를 계좌에 반영



Spring Batch

❖ Spring Batch Framework 용어

✓ Job & Step

● Step

- ◆ Job을 구성하는 독립된 작업의 단위
- ◆ Step에는 Tasklet 기반 Step과 chunk 기반 Step이라는 두 가지 주요 Step 유형이 있으며 그 중에 Tasklet 기반 Step의 구조가 더 간단
- ◆ Tasklet 기반 Step을 사용하고 싶으면 Tasklet을 구현하면 되는데 Step이 중지될 때까지 execute 메서드가 계속 반복해서(execute 메서드를 호출할 때마다 독립적인 트랜잭션이 얻어짐) 수행되고 Tasklet 기반 Step은 초기화, 저장 프로시저 실행, 알림 전송 등과 같은 Job에서 일반적으로 사용
- ◆ chunk 기반 Step은 구조가 약간 더 복잡하며 아이템 기반의 처리에 사용하는데 각 chunk 기반 Step은 ItemReader, ItemProcessor, ItemWriter 라는 3개의 주요 부분으로 구성될 수 있는데 Step은 3개의 부분으로 구성될 수 있는데 ItemProcessor가 필수는 아니며 ItemReader와 ItemWriter만으로 구성해 Step을 실행할 수도 있음{데이터 마이그레이션 Job에 일반적으로 사용}

Spring Batch

❖ Spring Batch Framework

✓ Job & Step

● Batch Job 인터페이스

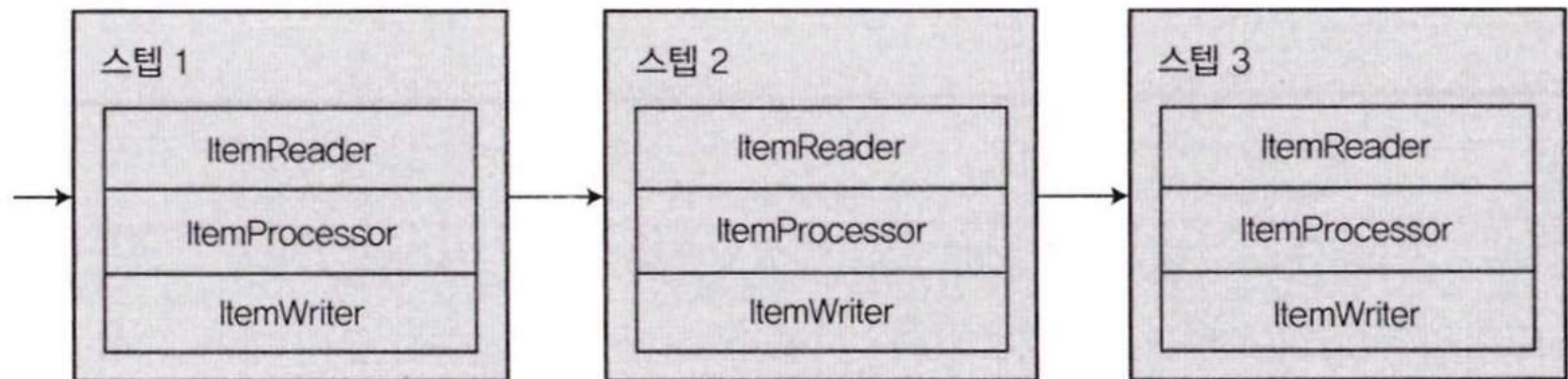
- ◆ `org.springframework.batch.core.Job`: Applicationcontext 내에 구성되는 Job 객체
- ◆ `org.springframework.batch.core.Step`: Applicationcontext 내에 구성되는 Step을 나타내는 객체
- ◆ `org.springframework.batch.core.step.tasklet.Tasklet`: 트랜잭션 내에서 로직이 실행될 수 있는 기능을 제공하는 전략(strategy) 인터페이스
- ◆ `org.springframework.batch.item.ItemReader<T>`: Step 내에서 입력을 제공하는 전략 인터페이스
- ◆ `org.springframework.batch.item.ItemProcessor<T>`: Step 내에서 제공받은 개별 아이템(Item)에 업무 로직, 검증 등을 적용하는 역할을 하는 인터페이스
- ◆ `org.springframework.batch.item.ItemWriter<T>`: Step 내에서 아이템을 저장하는 전략 인터페이스

Spring Batch

❖ Spring Batch Framework

✓ Job 실행

- Job이 실행될 때 스프링 배치의 많은 컴포넌트는 탄력성을 제공하기 위해 서로 상호 작용
- JobRepository
 - ◆ 스프링 배치 아키텍처 내에서 공유되는 주요 컴포넌트인 JobRepository 컴포넌트는 다양한 배치 수행과 관련된 수치 데이터(시작 시간, 종료 시간, 상태, 읽기/쓰기 횟수 등)뿐만 아니라 Job의 상태를 유지 관리
 - ◆ JobRepository 는 일반적으로 관계형 데이터베이스를 사용하며 스프링 배치 내의 대부분의 주요 컴포넌트가 공유



Spring Batch

❖ Spring Batch Framework

✓ Job 실행

● JobLauncher

- ◆ JobLauncher는 Job을 실행하는 역할을 담당
- ◆ JobLauncher는 Job.execute 메서드를 호출하는 역할 이외에도 Job의 재실행 가능 여부 검증, Job의 실행 방법(현재 스레드에서 수행할지 스레드 풀을 통해 실행할지 등), 파라미터 유효성 검증 등의 처리를 수행
- ◆ JobLauncher가 이 한 처리 중에 어떤 처리를 수행할지는 개발자가 구현하기에 따라 달라지는데 스프링 부트 환경이라면 스프링 부트가 즉시 Job을 시작하는 기능을 제공하므로 일반적으로 직접 다룰 필요가 없는 컴포넌트

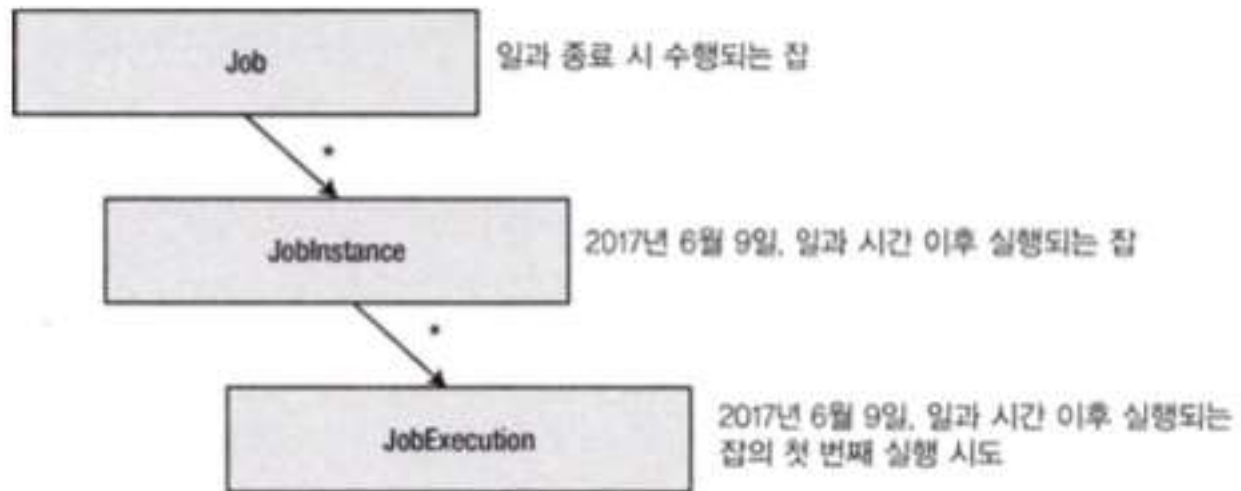
- Job을 실행하면 해당 Job은 각 Step을 실행하고 각 Step이 실행되면 JobRepository는 현재 상태로 갱신되는데 실행된 Step, 현재 상태, 읽은 아이템 및 처리된 아이템 수 등을 모두 JobRepository에 저장
- Job과 Step의 처리 방식은 매우 유사한데 Job은 구성된 Step 목록에 따라 각 Step을 실행
- 여러 아이템으로 이뤄진 청크의 처리가 Step 내에서 완료될 때 스프링 배치는 JobRepository 내에 있는 JobExecution 또는 StepExecution을 현재 상태로 갱신

Spring Batch

❖ Spring Batch Framework

✓ Job 실행

- Step은 ItemReader가 읽은 아이템의 목록을 따라가는데 Step이 각 청크를 처리할 때마다 JobRepository 내 StepExecution의 Step 상태가 업데이트되는데 현재까지의 커밋 수, 시작 및 종료 시간, 기타 다른 정보 등을 JobRepository에 저장
- Job 또는 Step이 완료되면 JobRepository 내에 있는 JobExecution 또는 StepExecution이 최종 상태로 업데이트됨.
- JobExecution과 StepExecution 의 관계



Spring Batch

❖ Spring Batch Framework

✓ Job 실행

- Jobinstance는 스프링 배치 Job의 논리적인 실행
- JobInstance는 Job의 이름 과 Job의 논리적 실행을 위해 제공되는 고유한 식별 파라미터 모음으로써 유일하게 존재
- 거래 명세서를 생성하는 statementGenerator라는 이름을 가진 Job을 예로 들면, statementGenerator Job이 다른 파라미터로 실행될 때마다 새로운 Jobinstance가 생성됨
- JobExecution은 스프링 배치 Job의 실제 실행을 의미
- Job을 구동할 때마다 매번 새로운 JobExecution을 얻게 되는데 어쩔 때는 새로운 Jobinstance를 얻지 못할 수 있음
- 실패한 Job을 재시작해보면 이에 대한 확실한 예를 볼 수 있는데 Job을 처음 실행하면 새로운 Jobinstance 및 JobExecution을 얻지만 실행에 실패한 이후 다시 실행하면 해당 실행은 여전히 동일한 논리적 실행(파라미터가 동일함)이므로 새 Jobinstance를 얻지 못하지만 그 대신 두 번째 실제 실행을 추적하기 위한 새로운 JobExecution을 얻음
- Jobinstance는 여러 개의 JobExecution을 가질 수 있음
- StepExecution은 Step의 실제 실행을 나타내는 반면 stepinstance라는 개념은 존재하지 않으며 일반적으로 JobExecution은 여러 개의 StepExecution과 연관

Spring Batch

❖ Spring Batch Framework

✓ 병렬화

- 가장 단순한 배치 처리 아키텍처는 Job 내의 Step을 처음부터 끝까지 순서대로 단일 스레드에서 실행하는 것
- 스프링 배치는 실 사례에 필요한 다양한 병렬화 방법을 제공
- Job을 병렬화하는 방법에는 다중 스레드 Step을 통한 작업 분할, 전체 Step의 병렬 실행, 비동기 ItemProcessor/ItemWriter 구성, 원격 청킹, 파티셔닝 방법이 있음
- 다중 스레드 Step
 - ◆ 스프링 배치에서 Job은 청크라는 블록 단위로 처리되도록 구성되며 각 청크는 각자 독립적인 트랜잭션으로 처리
 - ◆ 일반적으로 각 청크는 연속해서 처리
 - ◆ 10,000개의 레코드가 있을 때 커밋 수를 50개로 설정했다면 Job은 레코드 1부터 50까지를 처리한 다음에 커밋하고 다시 51에서 100까지 처리하고 커밋하며 이 과정을 10,000 레코드가 모두 처리될 때까지 반복
 - ◆ 스프링 배치를 사용하면 작업을 병렬로 실행해 성능을 향상시킬 수 있는데 이 작업에서 세 개의 스레드를 사용하도록 변경하면 처리량을 이론적으로 3배 늘릴 수 있음

Spring Batch

❖ Spring Batch Framework

✓ 병렬화

● 병렬 Step

- ◆ 입력 파일의 데이터를 읽어오는 한 개의 Step과 데이터베이스에 저장하는 일을 하는 한 개의 Step이 있지만 서로 관련이 없다고 가정하면 파일 하나를 전부 불러와 저장을 완료하기까지 다른 파일의 처리를 미루고 기다릴 필요가 없으므로 병렬 Step 처리 기능을 사용

● 비동기 ItemProcessor/ItemWriter

- ◆ Step 내의 itemProcessor에 병목현상이 발생할 수 있는데 예를 들어 ItemReader가 제공하는 데이터를 가공하기 위해 복잡한 수학 계산을 수행하거나 원격 서비스를 호출해야 할 수도 있는데 이럴 때 일부 Step을 병렬화하는 기능이 유용할 수 있음
- ◆ SynchronousItemProcessor는 ItemProcessor가 호출될 때마다 동일한 스레드에서 실행하게 만들어주는 ItemProcessor 구현체의 데코레이터
- ◆ AsynchronousItemProcessor는 ItemProcessor 호출 결과를 반환하는 대신 각 호출에 대해 `java.util.concurrent.Future`를 반환
- ◆ 현재 청크 내에서 반환된 Future 목록은 AsynchronousItem Writer로 전달
- ◆ AsynchronousItemWriter는 이 상황에서 Step 내에서 실제로 사용해야 하는 ItemWriter의 데코레이터인데 Future를 이용해 실제 결과를 얻어낸 후 이를 위임 ItemWriter에 전달

Spring Batch

❖ Spring Batch Framework

✓ 병렬화

● 원격 청킹

- ◆ 입력은 마스터 노드에서 표준 `ItemReader`를 사용해 이뤄지고 그런 다음 입력은 지속 가능한 통신 형식을 통해 메시지 기반 POJO로 구성된 원격 워커 `ItemProcessor`로 전송
- ◆ 처리가 완료되면 워커는 업데이트된 아이템을 다시 마스터로 보내거나 직접 기록
- ◆ 마스터에서 데이터를 읽고 원격 워커에서 처리한 다음 다시 마스터에게 전송하므로 네트워크 사용량이 매우 많아질 수 있다는 점에 유의해
- ◆ 실제 처리에 비해 I/O 비용이 적은 시나리오에 적합

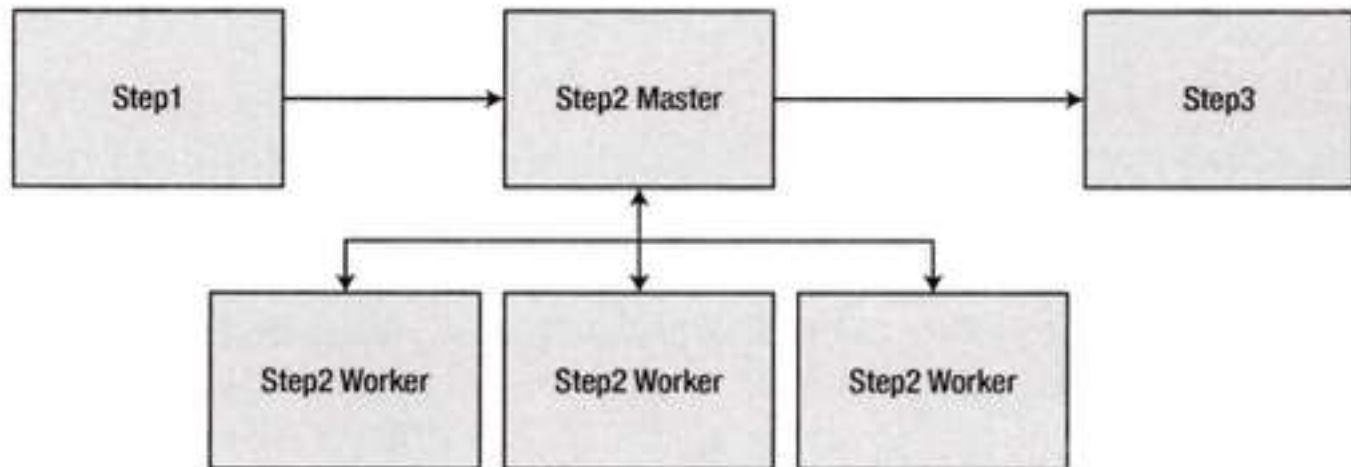
Spring Batch

❖ Spring Batch Framework

✓ 병렬화

● 파티셔닝

- ◆ 스프링 배치는 원격 파티셔닝(마스터 및 원격 워커를 사용) 및 로컬 파티셔닝(워커의 스레드 사용)을 모두 지원
- ◆ 원격 파티셔닝과 원격 청킹의 두 가지 주요 차이점은 원격 파티셔닝을 사용하면 내구성 있는 통신 방법이 필요하지 않으며 마스터는 워커의 Step 수집을 위한 컨트롤러 역할만 수행
- ◆ 이 경우 각 워커의 Step은 독립적으로 동작하며 마치 로컬로 배포된 것처럼 동일하게 구성



Spring Batch

❖ Spring Batch Framework

✓ Job 과 Step을 생성

- BatchWebApplication.java 파일을 수정

`@EnableBatchProcessing`

`@SpringBootApplication`

`public class BatchWebApplication {`

`@Autowired`

`private JobBuilderFactory jobBuilderFactory;`

`@Autowired`

`private StepBuilderFactory stepBuilderFactory;`

Spring Batch

❖ Spring Batch Framework

✓ Job 과 Step을 생성

- BatchWebApplication.java 파일을 수정

```
@Bean
public Step step() {
    return this.stepBuilderFactory.get("step1")
        .tasklet(new Tasklet() {
            @Override
            public RepeatStatus
            execute(StepContribution contribution,

                                ChunkContext chunkContext) {
                System.out.println("Hello,

World!");

                return RepeatStatus.FINISHED;
            }
        }).build();
}
```

Spring Batch

❖ Spring Batch Framework

✓ Job 과 Step을 생성

- BatchWebApplication.java 파일을 수정

```
@Bean
public Job job() {
    return this.jobBuilderFactory.get("job")
        .start(step())
        .build();
}

public static void main(String[] args) {
    SpringApplication.run(HelloWorldApplication.class, args);
}
}
```

Spring Batch

❖ Spring Batch Framework

✓ 프로젝트 생성 및 실행 - HelloWorld

● 실행 과정

- ◆ 스프링 부트에는 JobLauncherCommandLineRunner라는 컴포넌트가 있는데 이 컴포넌트는 스프링 배치가 클래스 경로에 있다면 실행 시에 로딩되며 JobLauncher를 사용해 ApplicationContext에서 찾아낸 모든 Job을 실행
- ◆ main 메서드에서 스프링 부트를 부트스트랩할 때 ApplicationContext가 생성되고 JobLauncherCommandLineRunner가 실행됐으며 Job이 수행
- ◆ Job은 첫 번째 Step을 실행했는데 이때 트랜잭션이 시작됐으며 Tasklet이 실행됐고 결과가 JobRepository에 갱신

Spring Batch

❖ 주요한 배치 개념

✓ 다양한 입력 및 출력 방식

- 스프링 배치의 가장 중요한 기능 중 하나로 다양한 소스를 대상으로 데이터의 입/출력을 처리하는 잘 추상화된 입출력 기능을 들 수 있음
- 은행 거래명세서 Job은 일반적인 텍스트 파일인 플랫 파일과 데이터베이스에서 입력을 받음

✓ 오류 처리

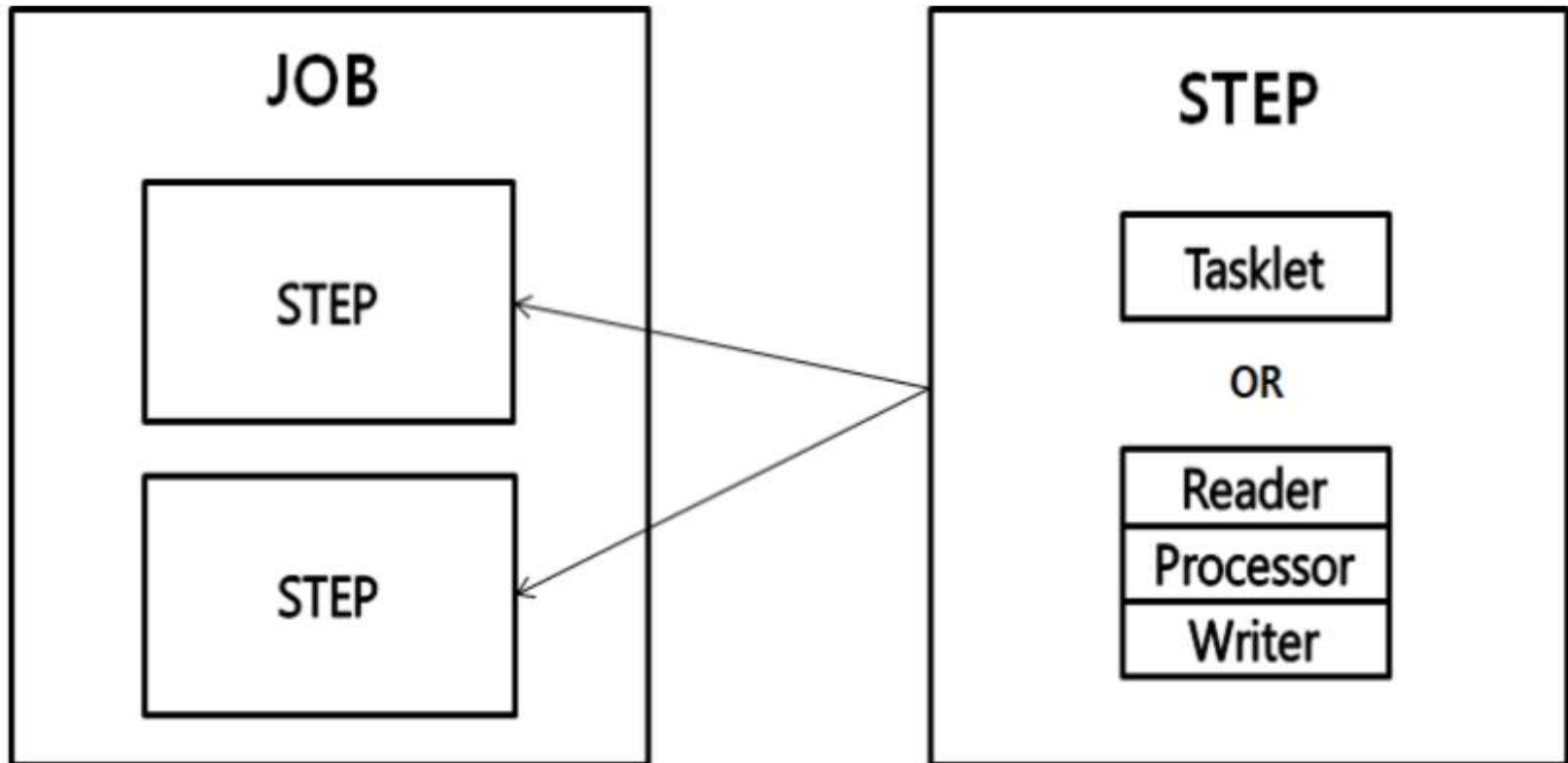
- 배치 처리를 유지보수하면서 겪는 최악의 상황은 배치 처리 중 오류가 발생해 이를 해결해야 하는 상황
- 강력한 오류 처리 능력은 필수적
- 거래명세서 생성 처리는 로깅, 오류가 있는 레코드 건너뛰기, 로직 수행 중 오류 발생 시 해당 로직 재시도하기 등 다양한 시나리오를 다룸

✓ 확장성

- 실제로 배치 처리는 많은 양의 데이터를 처리할 수 있어야 함

Spring Batch

❖ Spring Batch 사용



Spring Batch

❖ Spring Batch 작업 자동 생성되는 테이블

BATCH_JOB_EXECUTION
BATCH_JOB_EXECUTION_CONTEXT
BATCH_JOB_EXECUTION_PARAMS
BATCH_JOB_EXECUTION_SEQ
BATCH_JOB_INSTANCE
BATCH_JOB_SEQ
BATCH_STEP_EXECUTION
BATCH_STEP_EXECUTION_CONTEXT
BATCH_STEP_EXECUTION_SEQ

- ✓ Job을 수행하게 되면 Job의 정보를 저장한 후 이미 수행된 Job의 경우는 재실행하지 않음
- ✓ 재실행을 하고자 하는 경우에는 Job의 이름을 변경해야 하는데 번거롭다면 Job을 생성할 때 `incrementer(new RunIdIncrementer())`를 추가하면 자동으로 Job의 Id가 변경됨

Spring Batch

❖ Spring Batch 작업 자동 생성되는 테이블

- ✓ SEQUENCE: BATCH_JOB_INSTANCE, BATCH_JOB_EXECUTION 및 BATCH_STEP_EXECUTION의 Primary Key는 시퀀스를 이용해서 생성
- ✓ BATCH_JOB_INSTANCE 테이블에는 JobInstance에 관련된 모든 정보가 포함되어 있으며 해당 Table은 전체 계층 구조의 최상위 역할
- ✓ BATCH_JOB_EXECUTION_PARAMS 테이블에는 Job을 실행 시킬 때 사용했던 JobParameters에 대한 정보를 저장
- ✓ BATCH_JOB_EXECUTION 테이블에는 JobExecution에 관련된 모든 정보를 저장하고 있는데 JobExecution은 JobInstance가 실행 될 때마다 시작 시간, 종료 시간, 종료 코드 등 다양한 정보를 가지고 있음
- ✓ BATCH_JOB_EXECUTION 테이블에는 StepExecution에 대한 정보를 저장하고 있는데 BATCH_JOB_EXECUTION 테이블과 여러 면에서 유사하며 STEP을 EXECUTION 정보인 읽은 수, 커밋 수, 스킵 수 등 다양한 정보를 추가로 담고 있음
- ✓ BATCH_JOB_EXECUTION_CONTEXT 테이블에는 JobExecution의ExecutionContext 정보가 들어있는데 ExecutionContext 데이터는 일반적으로 JobInstance가 실패 시 중단된 위치에서 다시 시작할 수 있는 정보를 저장하고 있음
- ✓ BATCH_STEP_EXECUTION_CONTEXT테이블에는 StepExecution의 ExecutionContext 정보가 들어있는데 이 ExecutionContext 데이터는 일반적으로 JobInstance가 실패 시 중단된 위치에서 다시 시작할 수 있는 정보를 저장하고 있음

Spring Batch

❖ 단일 Step 생성

✓ Config 클래스 생성 - SingleStep

```
@Log4j2
@Configuration
@EnableBatchProcessing
@RequiredArgsConstructor
public class SingleStep {

    private final JobBuilderFactory jobBuilderFactory;
    private final StepBuilderFactory stepBuilderFactory;

    @Bean
    public Job Job1(){
        Job exampleJob = jobBuilderFactory.get("Job1")
            .start(Step())
            .build();

        return exampleJob;
    }
}
```

Spring Batch

❖ 단일 Step 생성

- ✓ Config 클래스 생성 - SingleStep

```
@Bean
public Step Step() {
    return stepBuilderFactory.get("step")
        .tasklet((contribution, chunkContext) -> {
            log.info("Step!");
            return RepeatStatus.FINISHED;
        })
        .build();
}
```

Spring Batch

- ❖ 단일 Step 생성
 - ✓ Config 클래스 생성 - SingleStep

```
Executing step: [step]
Step!
Step: [step] executed in 55ms
Job: [SimpleJob: [name=Job1]] completed with the following parameters: [{}] and the following status: [COMPLETED] in 152ms
```

Spring Batch

❖ 다중 Step 생성

✓ Config 클래스 생성 - MultiStep

```
@Slf4j
@Configuration
@EnableBatchProcessing
@RequiredArgsConstructor
public class MultiStep {

    private final JobBuilderFactory jobBuilderFactory;
    private final StepBuilderFactory stepBuilderFactory;

    @Bean
    public Job Job2(){

        Job exampleJob = jobBuilderFactory.get("Job2")
            .start(startStep())
            .next(nextStep())
            .next(lastStep())
            .build();

        return exampleJob;
    }
}
```

Spring Batch

❖ 다중 Step 생성

✓ Config 클래스 생성 - MultiStep

```
@Bean
public Step startStep() {
    return stepBuilderFactory.get("startStep")
        .tasklet((contribution, chunkContext) -> {
            log.info("Start Step!");
            return RepeatStatus.FINISHED;
        })
        .build();
}
```

```
@Bean
public Step nextStep(){
    return stepBuilderFactory.get("nextStep")
        .tasklet((contribution, chunkContext) -> {
            log.info("Next Step!");
            return RepeatStatus.FINISHED;
        })
        .build();
}
```

Spring Batch

❖ 다중 Step 생성

✓ Config 클래스 생성 - MultiStep

```
@Bean
public Step lastStep(){
    return stepBuilderFactory.get("lastStep")
        .tasklet((contribution, chunkContext) -> {
            log.info("Last Step!!");
            return RepeatStatus.FINISHED;
        })
        .build();
}
}
```

Spring Batch

❖ 다중 Step 생성

✓ Config 클래스 생성 - MultiStep

```
Job: [SimpleJob: [name=Job2]] launched with the following parameters: [{}]  
Executing step: [startStep]  
Start Step!  
Step: [startStep] executed in 69ms  
Executing step: [nextStep]  
Next Step!  
Step: [nextStep] executed in 45ms  
Executing step: [lastStep]  
Last Step!!  
Step: [lastStep] executed in 42ms  
Job: [SimpleJob: [name=Job2]] completed with the following parameters: [{}] and the following status: [COMPLETED] in 413ms
```

Spring Batch

❖ Flow Step 사용

✓ Config 클래스 생성 - FlowStep

@Slf4j

@Configuration

@EnableBatchProcessing

@RequiredArgsConstructor

public class FlowStep {

private final JobBuilderFactory jobBuilderFactory;

private final StepBuilderFactory stepBuilderFactory;

Spring Batch

❖ Flow Step 사용

- ✓ Config 클래스 생성 - FlowStep

@Bean

public Job Job3(){

 Job exampleJob = jobBuilderFactory.get("Job3")

 .start(startStep())

 .on("FAILED") //startStep의 ExitStatus가 FAILED일 경우

 .to(failOverStep()) //failOver Step을 실행 시킨다.

 .on("*") //failOver Step의 결과와 상관없이

 .to(writeStep()) //write Step을 실행 시킨다.

 .on("*") //write Step의 결과와 상관없이

 .end() //Flow를 종료시킨다.

 .from(startStep()) //startStep이 FAILED가 아니고

 .on("COMPLETED") //COMPLETED일 경우

 .to(processStep()) //process Step을 실행 시킨다

 .on("*") //process Step의 결과와 상관없이

 .to(writeStep()) // write Step을 실행 시킨다.

 .on("*") //write Step의 결과와 상관없이

 .end() //Flow를 종료 시킨다.

Spring Batch

❖ Flow Step 사용

✓ Config 클래스 생성 - FlowStep

```
.from(startStep()) //startStep의 결과가 FAILED, COMPLETED가 아닌
.on("") //모든 경우
.to(writeStep()) //write Step을 실행시킨다.
.on("") //write Step의 결과와 상관없이
.end() //Flow를 종료시킨다.
.end()
.build();

return exampleJob;
}
```

Spring Batch

❖ Flow Step 사용

✓ Config 클래스 생성 - FlowStep

```
@Bean
public Step startStep() {
    return stepBuilderFactory.get("startStep")
        .tasklet((contribution, chunkContext) -> {
            log.info("Start Step!");

            String result = "COMPLETED";
            //String result = "FAIL";
            //String result = "UNKNOWN";
        });
}
```

Spring Batch

❖ Flow Step 사용

✓ Config 클래스 생성 - FlowStep

```
//Flow에서 on은 RepeatStatus가 아닌 ExitStatus를 바라본다.  
if(result.equals("COMPLETED"))  
    contribution.setExitStatus(ExitStatus.COMPLETED);  
else if(result.equals("FAIL"))  
    contribution.setExitStatus(ExitStatus.FAILED);  
else if(result.equals("UNKNOWN"))  
    contribution.setExitStatus(ExitStatus.UNKNOWN);  
  
return RepeatStatus.FINISHED;  
})  
.build();  
}
```

Spring Batch

❖ Flow Step 사용

✓ Config 클래스 생성 - FlowStep

```
@Bean
public Step failOverStep(){
    return stepBuilderFactory.get("nextStep")
        .tasklet((contribution, chunkContext) -> {
            log.info("FailOver Step!");
            return RepeatStatus.FINISHED;
        })
        .build();
}
```

```
@Bean
public Step processStep(){
    return stepBuilderFactory.get("processStep")
        .tasklet((contribution, chunkContext) -> {
            log.info("Process Step!");
            return RepeatStatus.FINISHED;
        })
        .build();
}
```

Spring Batch

❖ Flow Step 사용

✓ Config 클래스 생성 - FlowStep

```
@Bean
public Step writeStep(){
    return stepBuilderFactory.get("writeStep")
        .tasklet((contribution, chunkContext) -> {
            log.info("Write Step!");
            return RepeatStatus.FINISHED;
        })
        .build();
}
```

Spring Batch

- ❖ Flow Step 사용
 - ✓ Config 클래스 생성 - FlowStep

```
Job: [FlowJob: [name=Job3]] launched with the following parameters: [{}]  
Executing step: [startStep]  
Start Step!  
Step: [startStep] executed in 44ms  
Executing step: [processStep]  
Process Step!  
Step: [processStep] executed in 35ms  
Executing step: [writeStep]  
Write Step!  
Step: [writeStep] executed in 38ms  
Job: [FlowJob: [name=Job3]] completed with the following parameters: [{}]  
and the following status: [COMPLETED] in 311ms
```

Spring Batch

❖ 다양한 Step 설정

✓ startlimit

```
@Bean
@JobScope
public Step Step() throws Exception {
    return stepBuilderFactory.get("Step")
        .startLimit(3)
        .<Member,Member>chunk(10)
        .reader(reader(null))
        .processor(processor(null))
        .writer(writer(null))
        .build();
}
```

- Step에서 startlimit는 해당 Step의 실패 이후 재시작 가능 횟수를 의미
- startlimit 이후 실행에서는 Exception이 발생

Spring Batch

❖ 다양한 Step 설정

✓ Skip

@Bean

@JobScope

public Step Step() throws Exception {

 return stepBuilderFactory.get("Step")

 .<Member,Member>chunk(10)

 .reader(reader(null))

 .processor(processor(null))

 .writer(writer(null))

 .faultTolerant()

 .skipLimit(1) // skip 허용 횟수, 해당 횟수 초과시 Error 발생, Skip 사용시

필수 설정

 .skip(NullPointerException.class)// NullPointerException에 대해선 Skip

 .noSkip(SQLException.class) // SQLException에 대해선 noSkip

 //.skipPolicy(new CustomSkipPolicy) // 사용자가 커스텀하며 Skip Policy

설정 가능

 .build();

}

Spring Batch

❖ 다양한 Step 설정

✓ Retry

@Bean

@JobScope

public Step Step() throws Exception {

 return stepBuilderFactory.get("Step")

 .<Member,Member>chunk(10)

 .reader(reader(null))

 .processor(processor(null))

 .writer(writer(null))

 .faultTolerant()

 .retryLimit(1) //retry 횟수, retry 사용시 필수 설정, 해당 Retry 이후
Exception시 Fail 처리

 .retry(SQLException.class) // SQLException에 대해선 Retry 수행

 .noRetry(NullPointerException.class) // NullPointerException에 no Retry

 //.retryPolicy(new CustomRetryPolicy) // 사용자가 커스텀하며 Retry Policy

설정 가능

 .build();

}

Spring Batch

❖ 다양한 Step 설정

✓ noRollback

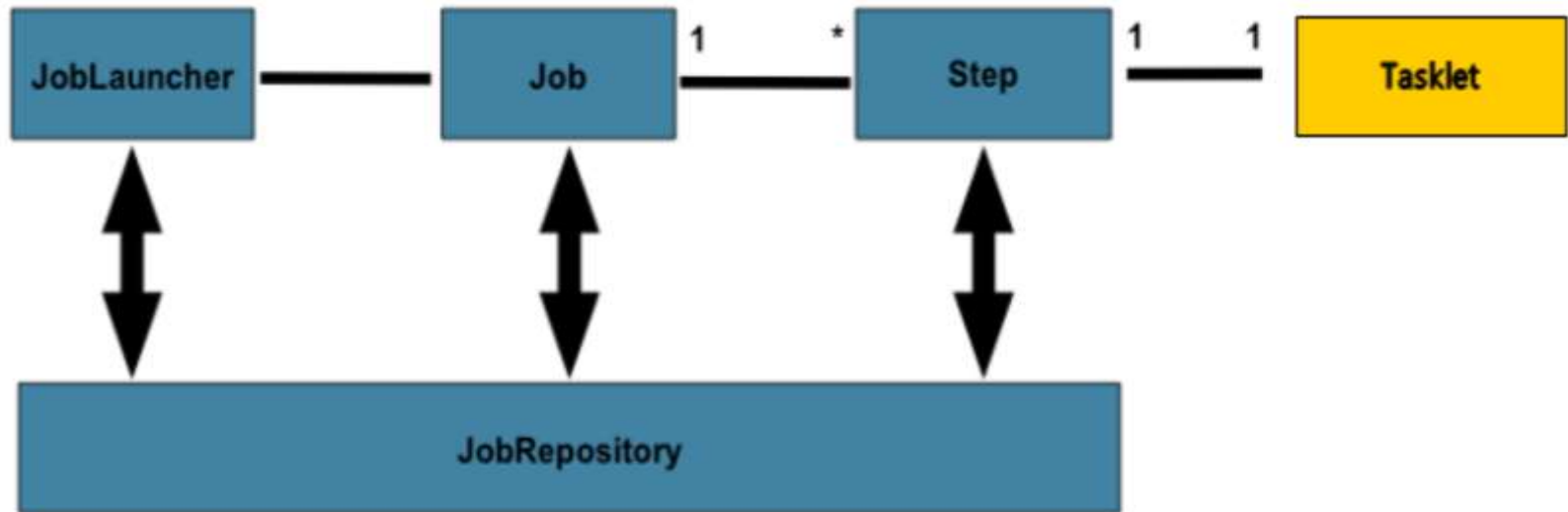
```
@Bean
@JobScope
public Step Step() throws Exception {
    return stepBuilderFactory.get("Step")
        .<Member,Member>chunk(10)
        .reader(reader(null))
        .processor(processor(null))
        .writer(writer(null))
        .faultTolerant()
        .noRollback(NullPointerException.class) // NullPointerException 발생
        rollback이 되지 않게 설정
        .build();
}
```

Spring Batch

❖ STEP을 구성하는 Tasklet과 Chunk 지향 처리

✓ Tasklet

- Tasklet은 하나의 메서드로 구성 되어있는 간단한 인터페이스
- 실패를 알리기 위해 예외를 반환 하거나 throw할 때까지 execute를 반복적으로 호출



Spring Batch

❖ STEP을 구성하는 Tasklet과 Chunk 지향 처리

✓ Tasklet

- Class 안에 Tasklet 구현(Lambda)

```
@Slf4j
@EnableBatchProcessing
@Configuration
@RequiredArgsConstructor
public class TaskletJobConfig{
    public final JobBuilderFactory jobBuilderFactory;

    public final StepBuilderFactory stepBuilderFactory;

    @Bean
    public Job TaskletJob(){
        Job customJob = jobBuilderFactory.get("taskletJob")
            .start(TaskStep())
            .build();
        return customJob;
    }
}
```

Spring Batch

❖ STEP을 구성하는 Tasklet과 Chunk 지향 처리

✓ Tasklet

- Class 안에 Tasklet 구현(Lambda)

```
@Bean
public Step TaskStep(){
    return stepBuilderFactory.get("taskletStep")
        .tasklet((contribution, chunkContext) ->{
            //비즈니스 로직
            for(int idx = 0; idx < 10; idx ++){
                log.info("[idx] = " + idx);
            }
            return RepeatStatus.FINISHED;
        }).build();
}
```

Spring Batch

❖ STEP을 구성하는 Tasklet과 Chunk 지향 처리

✓ Tasklet

- MethodInvokingAdapter를 사용하여 구현

```
@Slf4j
@Service
public class CustomService {

    public void businessLogic() {
        //비즈니스 로직
        for(int idx = 0; idx < 10; idx ++){
            log.info("[idx] = " + idx);
        }
    }
}
```

Spring Batch

❖ STEP을 구성하는 Tasklet과 Chunk 지향 처리

✓ Tasklet

- MethodInvokingAdapter를 사용하여 구현

@Slf4j

@Configuration

@EnableBatchProcessing

@RequiredArgsConstructor

public class TaskletJobConfig {

private final JobBuilderFactory jobBuilderFactory;

private final StepBuilderFactory stepBuilderFactory;

private final CustomService customService;

@Bean

public Job methodInvokingJob() {

return this.jobBuilderFactory.get("methodInvokingJob")

.start(this.methodInvokingStep())

.build();

}

Spring Batch

❖ STEP을 구성하는 Tasklet과 Chunk 지향 처리

✓ Tasklet

- MethodInvokingAdapter를 사용하여 구현

```
@Bean
public Step methodInvokingStep() {
    return this.stepBuilderFactory.get("methodInvokingStep")
        .tasklet(this.methodInvokingTasklet())
        .build();
}

@Bean
@StepScope
public MethodInvokingTaskletAdapter methodInvokingTasklet() {
    MethodInvokingTaskletAdapter methodInvokingTaskletAdapter = new
    MethodInvokingTaskletAdapter();
    methodInvokingTaskletAdapter.setTargetObject(customService);
    methodInvokingTaskletAdapter.setTargetMethod("businessLogic");
    //methodInvokingTaskletAdapter.setArguments(매개 변수);
    return methodInvokingTaskletAdapter;
}
}
```

Spring Batch

❖ STEP을 구성하는 Tasklet과 Chunk 지향 처리

✓ Tasklet

- 외부 클래스를 사용하여 Tasklet 구현

@Slf4j

@Service

public class BusinessTasklet implements Tasklet, StepExecutionListener {

 @Override

 @BeforeStep

 public void beforeStep(StepExecution stepExecution) {

 log.info("Before Step Start!");

 }

 @Override

 @AfterStep

 public ExitStatus afterStep(StepExecution stepExecution) {

 log.info("After Step Start!");

 return ExitStatus.COMPLETED;

 }

Spring Batch

❖ STEP을 구성하는 Tasklet과 Chunk 지향 처리

✓ Tasklet

- 외부 클래스를 사용하여 Tasklet 구현

@Override

```
public RepeatStatus execute(StepContribution stepContribution, ChunkContext  
chunkContext) throws Exception {
```

```
    //비즈니스 로직
```

```
    for (int idx = 0; idx < 10; idx++) {  
        log.info("[idx] = " + idx);  
    }
```

```
    return RepeatStatus.FINISHED;
```

```
}
```

```
}
```

Spring Batch

❖ STEP을 구성하는 Tasklet과 Chunk 지향 처리

✓ Tasklet

- 외부 클래스를 사용하여 Tasklet 구현

```
@Slf4j
@Service
public class ApplicationEventListener implements
    ApplicationListener<ApplicationReadyEvent> {

    @Override
    public void onApplicationEvent(ApplicationReadyEvent event) {

        long timestamp = event.getTimestamp();
        LocalDateTime eventTime =
            LocalDateTime.ofInstant(Instant.ofEpochMilli(timestamp),
                ZoneId.systemDefault());

        log.info("Application is Ready. {}", eventTime);
    }
}
```

Spring Batch

❖ STEP을 구성하는 Tasklet과 Chunk 지향 처리

✓ Tasklet

● 외부 클래스를 사용하여 Tasklet 구현

@Configuration

@EnableBatchProcessing

@RequiredArgsConstructor

public class TaskletJobConfig{

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

private final JobBuilderFactory jobBuilderFactory;

private final StepBuilderFactory stepBuilderFactory;

@Bean

public Job sampleTaskletJob(ApplicationEventListener applicationEventListener,
Step sampleTaskletStep) {

return jobBuilderFactory.get("sampleTaskletJob")

.incrementer(new RunIdIncrementer())

.flow(sampleTaskletStep)

.end()

.build();

}

Spring Batch

❖ STEP을 구성하는 Tasklet과 Chunk 지향 처리

✓ Tasklet

- 외부 클래스를 사용하여 Tasklet 구현

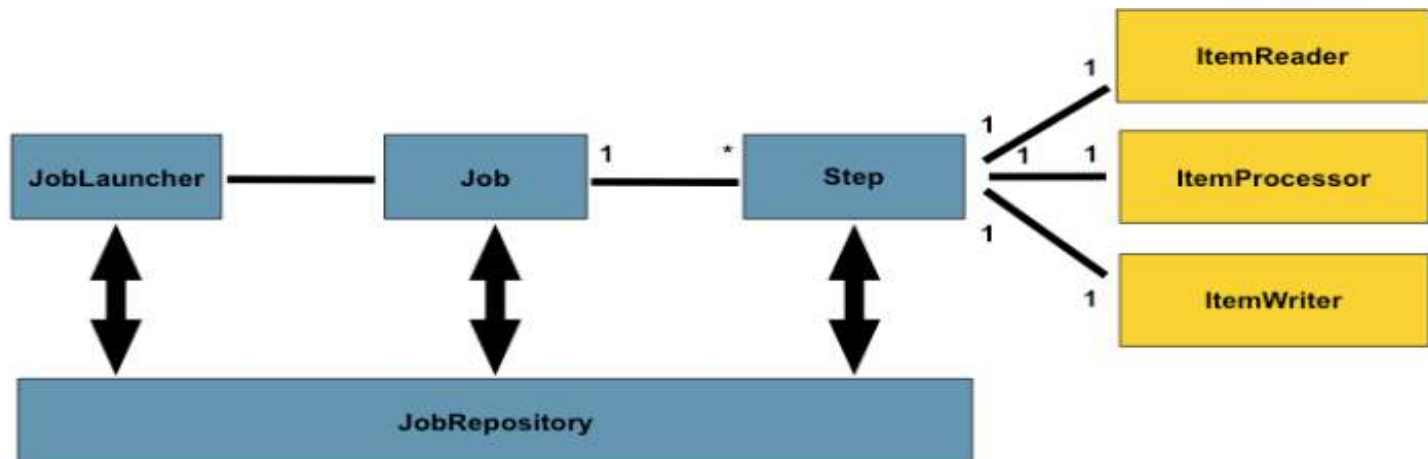
```
@Bean
public Step sampleTaskletStep() {
    return stepBuilderFactory.get("sampleTaskletStep")
        .tasklet(new BusinessTasklet())
        .build();
}
```

Spring Batch

❖ STEP을 구성하는 Tasklet과 Chunk 지향 처리

✓ Chunk

- Spring Batch에서의 Chunk란 처리 되는 커밋 row 수를 의미
- Batch 처리에서 커밋 되는 row 수 라는건 chunk 단위로 Transaction을 수행하기 때문에 실패시 Chunk 단위 만큼 rollback이 됨
- Chunk 지향 처리에서는 3가지 시나리오로 실행
 - ◆ 읽기(Read) - Database에서 배치 처리를 할 Data를 읽어옴
 - ◆ 처리(Processing) - 읽어온 Data를 가공, 처리(필수 사항X)
 - ◆ 쓰기(Write) - 가공, 처리한 데이터를 Database에 저장

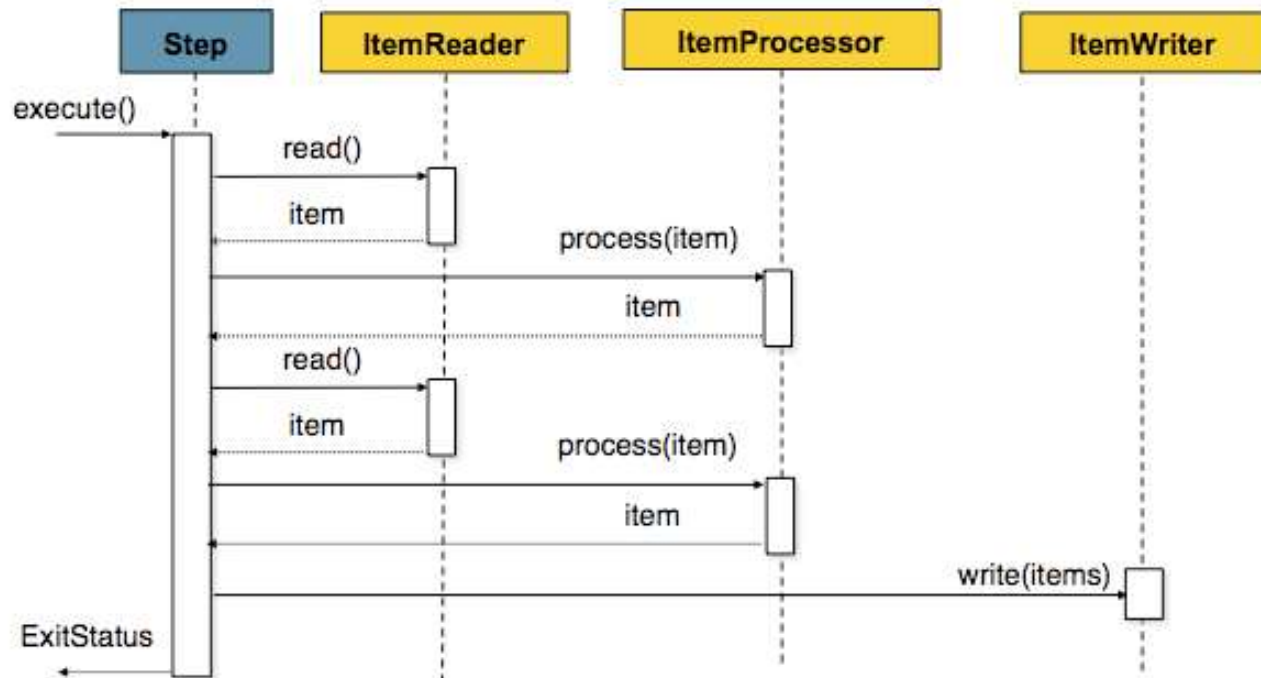


Spring Batch

❖ STEP을 구성하는 Tasklet과 Chunk 지향 처리

✓ Chunk

● 배치가 수행되는 그림



Spring Batch

❖ STEP을 구성하는 Tasklet과 Chunk 지향 처리

✓ Chunk

● 배치로 수행되는 그림

```
List items = new ArrayList();  
for(int i = 0; i < commitInterval; i++){  
    Object item = itemReader.read()  
    Object processedItem = itemProcessor.process(item);  
    items.add(processedItem);  
}  
itemWriter.write(items);
```

Spring Batch

❖ STEP을 구성하는 Tasklet과 Chunk 지향 처리

✓ Chunk

- Spring Batch에는 다양한 ItemReader와 ItemWriter가 존재
- 대용량 배치 처리를 하게 되면 Item을 읽어 올 때 Paging 처리를 하는게 효과적
- Spring Batch Reader에서는 Paging 처리를 지원하고 있으며 적절한 Paging 처리와 Chunk Size(한번에 처리 될 트랜잭션)를 설정하여 더욱 효과적인 배치 처리를 할 수 있음
- 적절한 Paging Size와 Chunk Size
 - ◆ Paging Size가 5이며 Chunk Size가 10일 경우 2번의 Read가 이루어진 후에 1번의 Transaction이 수행되는데 한번의 Transaction을 위해 2번의 쿼리 수행이 발생
 - ◆ Spring Batch에서의 권장 사항 - Setting a fairly large page size and using a commit interval that matches the page size should provide better performance - 페이지 크기를 상당히 크게 설정하고 페이지 크기와 일치하는 커밋 간격을 사용하면 성능이 향상
 - ◆ 한번의 Read 쿼리 수행시 1번의 Transaction을 위해 두 설정의 값을 일치시키는게 가장 좋은 성능 향상 방법이며 특별한 이유가 없는 한 Paging Size와 Chunk Size를 동일하게 설정하는 것을 추천

Spring Batch

- ❖ STEP을 구성하는 Tasklet과 Chunk 지향 처리
 - ✓ Chunk
 - PagingReader 사용 시 주의사항
 - ◆ 페이징 처리 시 각 쿼리에 Offset 과 Limit를 지정해 주어야 하는데 이는 PageSize를 지정하면 Batch에서 Offset과 Limit를 지정해주지만 페이징 처리를 할 때 마다 새로운 쿼리를 실행하기 때문에 데이터 순서가 보장 될 수 있도록 반드시 Order By를 사용하여야 함

Spring Batch

❖ ItemReader

✓ ItemReader 인터페이스

```
package org.springframework.batch.item;  
public interface ItemReader {  
    T read() throws Exception, UnexpectedInputException, ParseException,  
        NonTransientResourceException;  
}
```

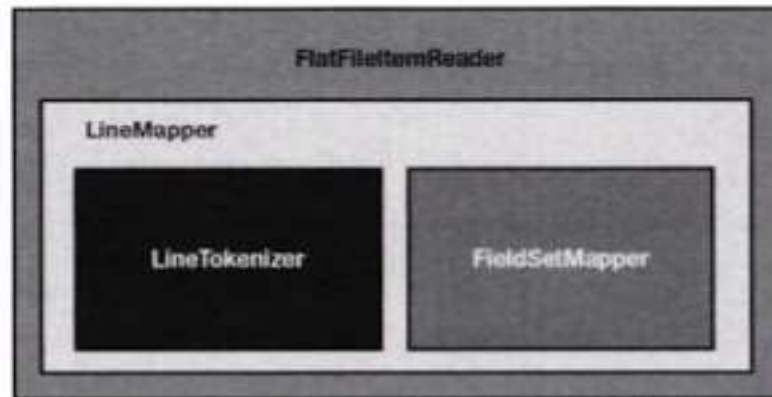
- ItemReader 인터페이스는 strategy interface로 스프링 배치는 처리할 입력 유형에 맞는 여러 구현체를 제공
- 스프링 배치가 구현체를 제공하는 입력 유형은 플랫 파일, 여러 데이터베이스, JMS 리소스, 기타 입력 소스 등으로 다양
- ItemReader 인터페이스나 ItemReader의 하위 인터페이스 중 하나를 구현해서 커스텀 ItemReader를 만들 수도 있음
- 스프링 배치가 ItemReader의 read 메서드를 호출하면 해당 메서드는 스텝 내에서 처리할 아이템 한 개를 반환
- 스텝에서는 아이템 개수를 세어서 청크 내의 데이터가 몇 개나 처리됐는지를 관리
- 해당 아이템은 구성된 특정 ItemProcessor로 전달되며 그 뒤 청크의 일부분에 포함돼 ItemWriter로 전달

Spring Batch

❖ ItemReader

✓ Flat File

- 플랫 파일이란 한 개 또는 그 이상의 레코드가 포함된 특정 파일
- 플랫 파일은 파일의 내용을 봐도 데이터의 의미를 알 수 없다는 점에서 XML 파일과 차이가 있는데 플랫 파일에는 파일 내에 데이터의 포맷이나 의미를 정의하는 메타 데이터가 없지만 XML 파일은 태그를 사용해 데이터에 의미를 부여
- org.springframework.batch.item.file.FlatFileItemReader는 메인 컴포넌트 두 개로 이뤄지는데 하나는 읽어들이 대상 파일을 나타내는 스프링의 Resource이며 다른 하나는 org.springframework.batch.item.file.LineMapper 인터페이스 구현체
- LineMapper는 스프링 JDBC에서 RowMapper가 담당하는 것과 비슷한 역할을 하는데 스프링 JDBC에서 RowMapper를 사용하면 필드의 묶음을 나타내는 Resultset을 객체로 매핑할 수 있음



Spring Batch

❖ ItemReader

✓ Flat File

● FlatFileItemReader 구성 옵션

옵션	타입	기본값	설명
comments	String[]	null	이 문자열 배열에는 파일을 파싱할 때 건너뛰어야 할 주석 줄을 나타내는 접두어를 지정한다.
encoding	String	플랫폼의 기본 Charset	파일에 사용된 문자열 인코딩
lineMapper	LineMapper	null (필수)	이 클래스는 파일의 한 줄을 String으로 읽은 뒤 처리 대상인 도메인 객체(아이템)로 변환한다.
linesToSkip	Int	0	잡을 실행할 때 FlatFileItemReader가 파일을 파싱하기 전에 파일 시작부터 몇 줄을 건너뛴 것인지 지정할 수 있다. 이 숫자는 몇 줄을 건너뛴 것인가를 나타낸다.
record SeparatorPolicy	Record SeparatorPolicy	DefaultRecord SeparatorPolicy	각 줄의 마지막을 정의하는 데 사용한다. 별도로 지정하지 않으면 개행 문자가 레코드의 끝 부분을 나타낸다. 또, 이 클래스는 여러 줄에 걸쳐 따옴표로 감싸진 구문을 다룰 때도 사용할 수 있다.
resource	Resource	null (필수)	읽을 대상 리소스
skippedLines Callback	LineCallback Handler	null	줄을 건너뛴 때 호출되는 콜백 인터페이스. 건너뛴 모든 줄은 이 콜백으로 넘겨진다.

Spring Batch

❖ ItemReader

✓ Flat File

● FlatFileItemReader 구성 옵션

strict	boolean	false	true로 설정하면 리소스를 찾을 수 없을 때 Exception을 던진다.
saveState	boolean	true	이 값이 true이면 재시작이 가능하도록 각 청크 처리 후에 ItemReader의 상태를 저장한다. 다중 스레드 환경에서 사용한다면 false로 지정해야 한다.
name	String	null	ExecutionContext에 저장되는 값의 고유 키를 생성하는 데 사용된다.
maxItemCount	int	Integer.MAX_VALUE	파일에서 읽어들이는 아이템의 최대 개수를 나타낸다.
currentItemCount	int	0	현재 조회 중인 아이템의 순번. 재시작 시 사용된다.

Spring Batch

❖ ItemReader

✓ Flat File

- 파일을 읽을 때는 파일에서 레코드 한 개에 해당하는 문자열이 LineMapper 구현체에 전달되는데 가장 많이 사용하는 LineMapper 구현체는 DefaultLineMapper
- DefaultLineMapper는 파일에서 읽은 원시 String을 대상으로 두 단계 처리를 거쳐 이후 처리에 사용할 도메인 객체로 변환
 - ◆ LineTokenizer 구현체가 해당 줄을 파싱해 `org.springframework.batch.item.file.FieldSet`으로 만든 후 LineTokenizer에 제공되는 String은 파일에서 가져온 한 줄 전체를 나타내는데 레코드 내의 각 필드를 도메인 객체로 매핑하려면 해당 줄을 파싱해 각 필드를 나타내는 데이터의 모음으로 변환할 수 있어야 하는데 스프링 배치의 FieldSet은 한 줄에 해당하는 필드의 모음을 나타내며 이는 데이터베이스로 작업할 때 사용하는 `java.sql.ResultSet`과 유사
 - ◆ FieldSetMapper 구현체는 FieldSet을 도메인 객체로 매핑하는데 LineTokenizer가 한 줄을 여러 필드로 나눴으므로 이제 각 입력 필드를 도메인 객체의 필드로 매핑할 수 있는데 이는 스프링 JDBC에서 RowMapper가 ResultSet의 row를 도메인 객체로 매핑하는 것과 유사

Spring Batch

❖ ItemReader

✓ Flat File

● 고정 너비 파일

◆ customer.txt

Aimee	CHoover	7341Vel Avenue	Mobile	AL35928
Jonas	UGilbert	8852In St.	Saint Paul	MN57321
Regan	MBaxter	4851Nec Av.	Gulfport	MS33193
Octavius	TJohnson	7418Cum Road	Houston	TX51507
Sydnee	NRobinson	894 Ornare. Ave	Olathe	KS25606

Spring Batch

❖ ItemReader

✓ Flat File

● 고정 너비 파일

- ◆ 레거시 메인프레임 시스템에서는 고정 너비 파일을 사용하는 것이 일반적
- ◆ 코볼이나 빅데이터를 비롯한 다른 여러 기술에서 고유의 저장 공간을 정의할 때 고정 너비 파일을 사용하기 때문에 배치를 개발할 때는 고정 너비 파일을 처리할 수 있어야 함
- ◆ 고객 파일 포맷

필드	길이	설명
First Name	11	이름
Middle Initial	1	가운데 이름의 첫 글자
Last Name	10	성
Address Number	4	주소에서 건물 번호 부분
Street	20	거주하는 거리 이름
City	16	거주 도시
State	2	CA(캘리포니아), TX(텍사스)등 주(州)의 두 자리 약자
Zip Code	5	우편번호

Spring Batch

❖ ItemReader

✓ Flat File

● 고정 너비 파일

◆ DTO 클래스 생성 – dto.CustomerDTO

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class CustomerDTO {
    private String firstName;
    private String middleInitial;
    private String lastName;
    private String addressNumber;
    private String street;
    private String city;
    private String state;
    private String zipCode;
}
```

Spring Batch

❖ ItemReader

✓ Flat File

● 고정 너비 파일

◆ 배치 작업 서비스 생성 – service.FlatFileService

@Configuration

@EnableBatchProcessing

@RequiredArgsConstructor

```
public class FlatFileService {
```

```
    private final JobBuilderFactory jobBuilderFactory;
```

```
    private final StepBuilderFactory stepBuilderFactory;
```

Spring Batch

❖ ItemReader

✓ Flat File

● 고정 너비 파일

◆ 배치 작업 서비스 생성 – service.FlatFileService

```
@Bean
@StepScope
public FlatFileItemReader<CustomerDTO> customerItemReader() {
    ClassPathResource resource = new
    ClassPathResource("input/customerFixedWidth.txt");
    return new FlatFileItemReaderBuilder<CustomerDTO>()
        .name("customerItemReader")
        .resource(resource)
        .fixedLength()
        .columns(new Range[]{new Range(1,11), new Range(12, 12), new
    Range(13, 22),
        new Range(23, 26), new Range(27,46), new Range(47,62),
    new Range(63,64),
        new Range(65,69)})
        .names(new String[] {"firstName", "middleInitial", "lastName",
        "addressNumber", "street", "city", "state","zipCode"})
        .targetType(CustomerDTO.class)
        .build();
}
```

Spring Batch

❖ ItemReader

✓ Flat File

● 고정 너비 파일

◆ 배치 작업 서비스 생성 – service.FlatFileService

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

```
@Bean
public Step copyFileStep() {
    return this.stepBuilderFactory.get("copyFileStep")
        .<CustomerDTO, CustomerDTO>chunk(10)
        .reader(customerItemReader())
        .writer(itemWriter())
        .build();
}
```

Spring Batch

❖ ItemReader

✓ Flat File

● 고정 너비 파일

◆ 배치 작업 서비스 생성 – service.FlatFileService

```
@Bean
public Job job() {
    return this.jobBuilderFactory.get("job")
        .start(copyFileStep())
        .build();
}
```

Spring Batch

❖ ItemReader

✓ Flat File

● 고정 너비 파일

◆ 실행 결과

```
CustomerDTO(firstName=Ainee, middleInitial=C, lastName=Hoover, addressNumber=7341, street=Vel Avenue, city=Mobile, state=AL, zipCode=35928)
CustomerDTO(firstName=Jonas, middleInitial=U, lastName=Gilbert, addressNumber=8852, street=In St., city=Saint Paul, state=MN, zipCode=57321)
CustomerDTO(firstName=Regan, middleInitial=M, lastName=Baxter, addressNumber=4851, street=Nec Av., city=Gulfport, state=MS, zipCode=33193)
CustomerDTO(firstName=Octavius, middleInitial=T, lastName=Johnson, addressNumber=7418, street=Cum Road, city=Houston, state=TX, zipCode=51507)
CustomerDTO(firstName=Sydnee, middleInitial=W, lastName=Robinson, addressNumber=894, street=Ornare. Ave, city=Olathe, state=KS, zipCode=25606)
CustomerDTO(firstName=Stuart, middleInitial=K, lastName=Mckenzie, addressNumber=5529, street=Orci Av., city=Nampa, state=ID, zipCode=18562)
CustomerDTO(firstName=Petra, middleInitial=Z, lastName=Lara, addressNumber=8401, street=Et St., city=Georgia, state=GA, zipCode=78323)
CustomerDTO(firstName=Cherokee, middleInitial=T, lastName=Lara, addressNumber=8516, street=Mauris St., city=Seattle, state=WA, zipCode=28720)
CustomerDTO(firstName=Athena, middleInitial=Y, lastName=Burt, addressNumber=4951, street=Mollis Rd., city=Newark, state=DE, zipCode=41034)
CustomerDTO(firstName=Kaitlin, middleInitial=M, lastName=Macias, addressNumber=5715, street=Velit St., city=Chandler, state=AZ, zipCode=86176)
CustomerDTO(firstName=Leroy, middleInitial=X, lastName=Cherry, addressNumber=7810, street=Vulputate St., city=Seattle, state=WA, zipCode=37783)
CustomerDTO(firstName=Connor, middleInitial=W, lastName=Montoya, addressNumber=4122, street=Mauris Av., city=College, state=AK, zipCode=99743)
CustomerDTO(firstName=Byron, middleInitial=X, lastName=Medina, addressNumber=7875, street=At Road, city=Rock Springs, state=WY, zipCode=37733)
CustomerDTO(firstName=Penelope, middleInitial=Y, lastName=Sandoval, addressNumber=2643, street=Fringilla Av., city=College, state=AK, zipCode=99557)
CustomerDTO(firstName=Rashad, middleInitial=V, lastName=Ochoa, addressNumber=6587, street=Lacus Street, city=Flint, state=MI, zipCode=96640)
CustomerDTO(firstName=Jordan, middleInitial=U, lastName=Oneil, addressNumber=170, street=Mattis Ave, city=Bellevue, state=WA, zipCode=44941)
CustomerDTO(firstName=Caesar, middleInitial=D, lastName=Daugherty, addressNumber=7483, street=Libero Ave, city=Frankfort, state=KY, zipCode=56493)
```

Spring Batch

❖ ItemReader

✓ Flat File

- 구분자로 구분된 파일
 - ◆ csv 파일

```
Aimee,C,Hoover,7341,Vel Avenue,Mobile,AL,35928
Jonas,U,Gilbert,8852,In St.,Saint Paul,MN,57321
Regan,M,Baxter,4851,Nec Av.,Gulfport,MS,33193
Octavius,T,Johnson,7418,Cum Road,Houston,TX,51507
Sydnee,N,Robinson,894,Ornare. Ave,Olathe,KS,25606
Stuart,K,Mckenzie,5529,Orci Av.,Nampa,ID,18562
Petra,Z,Lara,8401,Et St.,Georgia,GA,70323
```

Spring Batch

❖ ItemReader

✓ Flat File

● 구분자로 구분된 파일

◆ 매퍼 파일 작성

```
public class CustomerFieldSetMapper implements  
    FieldSetMapper<CustomerDTO> {
```

```
    public CustomerDTO mapFieldSet(FieldSet fieldSet) {  
        CustomerDTO customer = new CustomerDTO();
```

```
        customer.setCity(fieldSet.readString("address"));  
        customer.setCity(fieldSet.readString("city"));  
        customer.setFirstName(fieldSet.readString("firstName"));  
        customer.setLastName(fieldSet.readString("lastName"));  
        customer.setMiddleInitial(fieldSet.readString("middleInitial"));  
        customer.setState(fieldSet.readString("state"));  
        customer.setZipCode(fieldSet.readString("zipCode"));
```

```
        return customer;
```

```
    }  
}
```

Spring Batch

❖ ItemReader

✓ Flat File

● 구분자로 구분된 파일

◆ 배치 파일 작성

@Configuration

@EnableBatchProcessing

@RequiredArgsConstructor

public class CsvFileService {

private final JobBuilderFactory jobBuilderFactory;

private final StepBuilderFactory stepBuilderFactory;

Spring Batch

❖ ItemReader

✓ Flat File

● 구분자로 구분된 파일

◆ 배치 파일 작성

```
@Bean
@StepScope
public FlatFileItemReader<CustomerDTO> customerItemReader() {
    ClassPathResource resource = new ClassPathResource("input/customer.csv");
    return new FlatFileItemReaderBuilder<CustomerDTO>()
        .name("customerItemReader")
        .delimited()
        .names(new String[] {"firstName",
                               "middleInitial",
                               "lastName",
                               "addressNumber",
                               "street",
                               "city",
                               "state",
                               "zipCode"})
        .fieldSetMapper(new CustomerFieldSetMapper())
        .resource(resource)
        .build();
}
```

Spring Batch

❖ ItemReader

✓ Flat File

● 구분자로 구분된 파일

◆ 배치 파일 작성

```
@Bean
public ItemWriter<CustomerDTO> itemWriter() {
    return (items) -> items.forEach(System.out::println);
}
@Bean
public Step copyFileStep() {
    return this.stepBuilderFactory.get("copyFileStep")
        .<CustomerDTO, CustomerDTO>chunk(10)
        .reader(customerItemReader())
        .writer(itemWriter())
        .build();
}
@Bean
public Job job() {
    return this.jobBuilderFactory.get("csvjob")
        .start(copyFileStep())
        .build();
}
}
```

Spring Batch

❖ ItemReader

✓ Flat File

● json 파일

◆ 파일 구성

```
[
  {
    "firstName": "Laura",
    "middleInitial": "0",
    "lastName": "Minella",
    "address": "2039 Wall Street",
    "city": "Omaha",
    "state": "IL",
    "zipCode": "35446",
    "transactions": [
      {
        "accountNumber": 829433,
        "transactionDate": "2010-10-14 05:49:58",
        "amount": 26.08
      }
    ]
  },
]
```

Spring Batch

❖ ItemReader

✓ Flat File

● json 파일

◆ DTO 클래스 – dto.TransactionDTO

@Data

@AllArgsConstructor

@NoArgsConstructor

```
public class TransactionDTO {  
    private String accountNumber;  
    private Date transactionDate;  
    private Double amount;  
}
```

Spring Batch

❖ ItemReader

✓ Flat File

● json 파일

◆ DTO 클래스 – dto.CustomerDTO

@Data

@NoArgsConstructor

@AllArgsConstructor

public class CustomerDTO {

private String firstName;

private String middleInitial;

private String lastName;

private String address;

private String city;

private String state;

private String zipCode;

private List<TransactionDTO> transactions;

}

Spring Batch

❖ ItemReader

✓ Flat File

● json 파일

◆ 배치 파일

@Configuration

@EnableBatchProcessing

@RequiredArgsConstructor

public class JsonJob {

private final JobBuilderFactory jobBuilderFactory;

private final StepBuilderFactory stepBuilderFactory;

Spring Batch

❖ ItemReader

✓ Flat File

● json 파일

◆ 배치 파일

@Bean

@StepScope

```
public JsonItemReader<CustomerDTO> customerFileReader() {  
    ClassPathResource resource = new  
    ClassPathResource("input/customer.json");  
    ObjectMapper objectMapper = new ObjectMapper();  
    objectMapper.setDateFormat(new SimpleDateFormat("yyyy-MM-  
dd hh:mm:ss"));
```

```
    JacksonJsonObjectReader<CustomerDTO> jsonObjectReader =  
    new JacksonJsonObjectReader<>(CustomerDTO.class);  
    jsonObjectReader.setMapper(objectMapper);
```

```
    return new JsonItemReaderBuilder<CustomerDTO>()  
        .name("customerFileReader")  
        .jsonObjectReader(jsonObjectReader)  
        .resource(resource)  
        .build();
```

```
}
```

Spring Batch

❖ ItemReader

✓ Flat File

● json 파일

◆ 배치 파일

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

@Bean
public Step copyFileStep() {
    return this.stepBuilderFactory.get("copyFileStep")
        .<CustomerDTO, CustomerDTO>chunk(10)
        .reader(customerFileReader())
        .writer(itemWriter())
        .build();
}
```

Spring Batch

❖ ItemReader

✓ Flat File

● json 파일

◆ 배치 파일

```
@Bean
public Job job() {
    return this.jobBuilderFactory.get("jsonparsingjob")
        .start(copyFileStep())
        .build();
}
```

Spring Batch

❖ ItemReader

✓ JDBC

- 데이터베이스에 접속해서 샘플 데이터 생성 – 2개의 SQL 파일 이용



Spring Batch

❖ ItemReader

✓ JDBC

● 테이블 구조

	ABC Field ▼	ABC Type ▼	ABC Null ▼	ABC Key ▼	ABC Default ▼	ABC Extra ▼
1	id	bigint	NO	PRI	[NULL]	
2	firstName	varchar(30)	NO		[NULL]	
3	middleInitial	varchar(3)	YES		[NULL]	
4	lastName	varchar(30)	NO		[NULL]	
5	address	varchar(45)	NO		[NULL]	
6	city	varchar(16)	NO		[NULL]	
7	state	char(2)	NO		[NULL]	
8	zipCode	char(5)	YES		[NULL]	

Spring Batch

❖ ItemReader

✓ JDBC

● CustomerDTO 클래스 수정

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class CustomerDTO {
    private Long id;
    private String firstName;
    private String middleInitial;
    private String lastName;
    private String address;
    private String city;
    private String state;
    private String zipCode;
}
```

Spring Batch

❖ ItemReader

✓ JDBC

● CustomerRowMapper 클래스 생성

```
public class CustomerRowMapper implements RowMapper<CustomerDTO> {
```

```
    @Override
```

```
    public CustomerDTO mapRow(ResultSet resultSet, int rowNumber) throws  
        SQLException {
```

```
        CustomerDTO customer = new CustomerDTO();
```

```
        customer.setId(resultSet.getLong("id"));
```

```
        customer.setAddress(resultSet.getString("address"));
```

```
        customer.setCity(resultSet.getString("city"));
```

```
        customer.setFirstName(resultSet.getString("firstName"));
```

```
        customer.setLastName(resultSet.getString("lastName"));
```

```
        customer.setMiddleInitial(resultSet.getString("middleInitial"));
```

```
        customer.setState(resultSet.getString("state"));
```

```
        customer.setZipCode(resultSet.getString("zipCode"));
```

```
        return customer;
```

```
    }
```

```
}
```

Spring Batch

❖ ItemReader

✓ JDBC

● 배치 작업 클래스 생성

@Configuration

@EnableBatchProcessing

@RequiredArgsConstructor

public class JdbcCustomerCursorService {

private final JobBuilderFactory jobBuilderFactory;

private final StepBuilderFactory stepBuilderFactory;

@Bean

public JdbcCursorItemReader<CustomerDTO>

customerItemReader(DataSource dataSource) {

return new JdbcCursorItemReaderBuilder<CustomerDTO>()

.name("customerItemReader")

.dataSource(dataSource)

.sql("select * from CUSTOMER where city = ?")

.rowMapper(new CustomerRowMapper())

.preparedStatementSetter(citySetter())

.build();

}

Spring Batch

❖ ItemReader

✓ JDBC

● 배치 작업 클래스 생성

```
@Bean
@StepScope
public ArgumentPreparedStatementSetter citySetter() {
    return new ArgumentPreparedStatementSetter(new Object [] {"Chicago"});
}
```

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

```
@Bean
public Step copyFileStep() {
    return this.stepBuilderFactory.get("copyFileStep")
        .<CustomerDTO, CustomerDTO>chunk(10)
        .reader(customerItemReader(null))
        .writer(itemWriter())
        .build();
}
```

Spring Batch

❖ ItemReader

✓ JDBC

● 배치 작업 클래스 생성

```
@Bean
public Job job() {
    return this.jobBuilderFactory.get("jdbcjob2")
        .start(copyFileStep())
        .build();
}

}
```

Spring Batch

❖ ItemReader

✓ JDBC

● 결과

```
CustomerDTO(id=297, firstName=Hermione, middleInitial=K, lastName=Kirby, address=599-9125 Et St., city=Chicago, state=IL, zipCode=95546)
```

```
CustomerDTO(id=831, firstName=Oren, middleInitial=Y, lastName=Benson, address=P.O. Box 201, 1204 Sed St., city=Chicago, state=IL, zipCode=91416)
```

Spring Batch

❖ ItemReader

✓ JPA

● Entity 클래스 생성

```
@EnableBatchProcessing  
@SpringBootApplication  
@RequiredArgsConstructor  
public class JpaJob {
```

```
    private final JobBuilderFactory jobBuilderFactory;
```

```
    private final StepBuilderFactory stepBuilderFactory;
```

Spring Batch

❖ ItemReader

✓ JPA

● Entity 클래스 생성

```
@Bean
@StepScope
public JpaPagingItemReader<Customer> customerItemReader(
    EntityManagerFactory entityManagerFactory) {
    Map<String, Object> parameterValues = new HashMap<>();
    parameterValues.put("city", "Chicago");
    return new JpaPagingItemReaderBuilder<Customer>()
        .name("customerItemReader")
        .entityManagerFactory(entityManagerFactory)
        .parameterValues(parameterValues)
        .queryString("select c from Customer c where c.city
= :city")
        .pageSize(3)
        .build();
}
```

Spring Batch

❖ ItemReader

✓ JPA

● Entity 클래스 생성

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

@Bean
public Step copyFileStep() {
    return this.stepBuilderFactory.get("copyFileStep")
        .<Customer, Customer>chunk(10)
        .reader(customerItemReader(null))
        .writer(itemWriter())
        .build();
}

@Bean
public Job job() {
    return this.jobBuilderFactory.get("jpajob9")
        .start(copyFileStep())
        .build();
}
}
```

Spring Batch

❖ ItemReader

✓ JPA

● 결과

```
CustomerDTO(id=297, firstName=Hermione, middleInitial=K, lastName=Kirby, address=599-9125 Et St., city=Chicago, state=IL, zipCode=95546)
```

```
CustomerDTO(id=831, firstName=Oren, middleInitial=Y, lastName=Benson, address=P.O. Box 201, 1204 Sed St., city=Chicago, state=IL, zipCode=91416)
```

Spring Batch

❖ ItemProcessor

✓ ItemProcessor

- 스프링 배치 내에서 입력 데이터를 이용해 어떤 작업을 수행하는 컴포넌트
- 입력 데이터를 받을 때 선택할 수 있는 옵션
 - ◆ 입력 데이터를 다시 쓰는 것으로 특정 시스템에서 다른 시스템으로 데이터를 마이그레이션하거나 데이터베이스에 데이터를 초기 적재하는 작업은 입력 데이터를 읽어서 별도의 추가 처리 없이 쓰기 작업을 수행
 - ◆ 특정 작업 수행
 - 입력의 유효성 검증: ItemProcessor가 유효성 검증을 수행하도록 하면 입력 방법에 상관없이 처리 전에 객체의 유효성 검증을 수행할 수 있음
 - 기존 서비스의 재사용: 입력 데이터를 다루는 기존의 서비스를 재사용
 - 스크립트 실행: ScriptItemProcessor를 사용하면 특정 스크립트를 ItemProcessor로써 실행할 수 있는데 스크립트에 입력으로 아이템을 제공하고 스크립트의 출력을 반환 값으로 가져올 수 있음
 - ItemProcessor의 Chain: 동일한 트랜잭션 내에서 단일 아이템으로 여러 작업을 수행하려는 상황이 있을 수 있는데 단일 클래스 내에서 모든 로직이 수행되도록 커스텀 ItemProcessor를 만들 수도 있지만 개발 로직이 프레임워크에 강하게 결합되기를 피하고자 하는 경우 스프링 배치를 사용해 각 아이템에 대해 순서대로 실행 될 ItemProcessor 목록을 만들 수 있음

Spring Batch

❖ ItemProcessor

✓ ItemProcessor 인터페이스

```
public interface ItemProcessor<I, O> {  
    O process(I item) throws Exception;  
}
```

- process 라는 메서드를 소유하고 있는데 이 메서드는 ItemReader가 읽어들이는 아이템을 전달받아 특정 처리를 수행한 후 결과 아이템을 반환
- ItemProcessor가 받아들이는 입력 아이템의 타입과 반환하는 아이템의 타입이 같을 필요가 없음
- ItemProcessor가 반환하는 타입은 ItemWriter가 입력으로 사용

Spring Batch

❖ ItemProcessor

- ✓ Spring Batch 가 제공하는 ItemProcessor 구현체
 - ValidatingItemProcessor
 - ◆ 입력 데이터의 유효성 검증을 위한 구현체
 - ◆ 유효성 검증에 실패하면 ValidationException이 발생
 - ◆ 유효성 검증 방법은 JSR-303 유효성 검증 도구를 사용

Spring Batch

❖ ItemProcessor

✓ Spring Batch 가 제공하는 ItemProcessor 구현체

- ValidatingItemProcessor

- ◆ 유효성 검사를 위한 유효성 검사 라이브러리 의존성 추가

- implementation 'org.springframework.boot:spring-boot-starter-validation'

Spring Batch

❖ ItemProcessor

✓ Spring Batch 가 제공하는 ItemProcessor 구현체

● ValidatingItemProcessor

◆ CustomerDTO 수정

@Data

@NoArgsConstructor

@AllArgsConstructor

public class CustomerDTO {

@NotNull(message="First name is required")

@Pattern(regexp="[a-zA-Z]+", message="First name must be alphabetical")

private String firstName;

@Size(min=1, max=1)

@Pattern(regexp="[a-zA-Z]", message="Middle initial must be alphabetical")

private String middleInitial;

@NotNull(message="Last name is required")

@Pattern(regexp="[a-zA-Z]+", message="Last name must be alphabetical")

private String lastName;

Spring Batch

❖ ItemProcessor

✓ Spring Batch 가 제공하는 ItemProcessor 구현체

● ValidatingItemProcessor

◆ CustomerDTO 수정

```
@NotNull(message="Address is required")  
@Pattern(regexp="[0-9a-zA-Z\\W. ]+")  
private String address;
```

```
@NotNull(message="City is required")  
@Pattern(regexp="[a-zA-Z\\W. ]+")  
private String city;
```

```
@NotNull(message="State is required")  
@Size(min=2,max=2)  
@Pattern(regexp="[A-Z]{2}")  
private String state;
```

```
@NotNull(message="Zip is required")  
@Size(min=5,max=5)  
@Pattern(regexp="\\W\\d{5}")  
private String zip;
```

```
}
```

Spring Batch

❖ ItemProcessor

✓ Spring Batch 가 제공하는 ItemProcessor 구현체

- ValidatingItemProcessor

- ◆ 배치 파일 작성

- @Configuration

- @EnableBatchProcessing

- @RequiredArgsConstructor

- public class CsvFileService {

- private final JobBuilderFactory jobBuilderFactory;

- private final StepBuilderFactory stepBuilderFactory;

Spring Batch

❖ ItemProcessor

✓ Spring Batch 가 제공하는 ItemProcessor 구현체

● ValidatingItemProcessor

◆ 배치 파일 작성

```
@Bean
@StepScope
public FlatFileItemReader<CustomerDTO> customerItemReader() {
    ClassPathResource resource = new
    ClassPathResource("input/customer_1.csv");
    return new FlatFileItemReaderBuilder<CustomerDTO>()
        .name("customerItemReader")
        .delimited()
        .names(new String[] {"firstName",
            "middleInitial",
            "lastName",
            "address",
            "city",
            "state",
            "zip"})
        .targetType(CustomerDTO.class)
        .resource(resource)
        .build();
}
```

Spring Batch

❖ ItemProcessor

✓ Spring Batch 가 제공하는 ItemProcessor 구현체

- ValidatingItemProcessor

- ◆ 배치 파일 작성

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

```
@Bean
public BeanValidatingItemProcessor<CustomerDTO>
customerDTOBeanValidatingItemProcessor(){
    return new BeanValidatingItemProcessor<>();
}
```

Spring Batch

❖ ItemProcessor

✓ Spring Batch 가 제공하는 ItemProcessor 구현체

● ValidatingItemProcessor

◆ 배치 파일 작성

```
@Bean
public Step copyFileStep() {
    return this.stepBuilderFactory.get("copyFileStep")
        .<CustomerDTO, CustomerDTO>chunk(10)
        .reader(customerItemReader())
        .processor(customerDTOBeanValidatingItemProcessor())
        .writer(itemWriter())
        .build();
}

@Bean
public Job job() {
    return this.jobBuilderFactory.get("processingvalidating")
        .start(copyFileStep())
        .build();
}
```

Spring Batch

❖ ItemProcessor

✓ Spring Batch 가 제공하는 ItemProcessor 구현체

● ValidatingItemProcessor

◆ 결과

```
org.springframework.batch.item.validator.ValidationException: Validation failed for CustomerDTO(firstName=Laura, middleInitial=9S, lastName=Minella, address=8177 4th
Field error in object 'iten' on field 'middleInitial': rejected value [9S]; codes [Size.item.middleInitial,Size.middleInitial,Size.java.lang.String,Size]; arguments
Field error in object 'iten' on field 'middleInitial': rejected value [9S]; codes [Pattern.item.middleInitial,Pattern.middleInitial,Pattern.java.lang.String,Pattern]
at org.springframework.batch.item.validator.SpringValidator.validate(SpringValidator.java:54) ~[spring-batch-infrastructure-4.3.8.jar:4.3.8]
```

Spring Batch

❖ ItemProcessor

- ✓ 기존 서비스를 이용하는 데이터 처리
 - 이름의 첫 글자를 대문자로 변환하는 데이터 처리
 - ◆ 서비스 클래스 생성 – UpperCaseNameService

```
@Service
public class UpperCaseNameService {
    public CustomerDTO upperCase(CustomerDTO customer) {
        CustomerDTO newCustomer = new
        CustomerDTO(customer.getFirstName(), customer.getMiddleInitial(),
        customer.getLastName(),
                                customer.getAddress(), customer.getCity(),
        customer.getState(), customer.getZip());

        newCustomer.setFirstName(newCustomer.getFirstName().toUpper
        Case());
        newCustomer.setMiddleInitial(newCustomer.getMiddleInitial().toU
        pperCase());
        newCustomer.setLastName(newCustomer.getLastName().toUpperCase());

        return newCustomer;
    }
}
```

Spring Batch

❖ ItemProcessor

- ✓ 기존 서비스를 이용하는 데이터 처리
 - 이름을 대문자로 변환하는 데이터 처리
 - ◆ 서비스 클래스 생성 – UpperCaseNameService

```
@Service
public class UpperCaseNameService {
    public CustomerDTO upperCase(CustomerDTO customer) {
        CustomerDTO newCustomer = new
        CustomerDTO(customer.getFirstName(), customer.getMiddleInitial(),
        customer.getLastName(),
                                customer.getAddress(), customer.getCity(),
        customer.getState(), customer.getZip());

        newCustomer.setFirstName(newCustomer.getFirstName().toUpper
        Case());
        newCustomer.setMiddleInitial(newCustomer.getMiddleInitial().toUpperCa
        se());
        newCustomer.setLastName(newCustomer.getLastName().toUpper
        Case());
        return newCustomer;
    }
}
```

Spring Batch

❖ ItemProcessor

- ✓ 기존 서비스를 이용하는 데이터 처리
 - 이름을 대문자로 변환하는 데이터 처리
 - ◆ 배치 파일 수정

```
@Configuration
```

```
@EnableBatchProcessing
```

```
@RequiredArgsConstructor
```

```
public class CsvFileService {
```

```
    private final JobBuilderFactory jobBuilderFactory;
```

```
    private final StepBuilderFactory stepBuilderFactory;
```

Spring Batch

❖ ItemProcessor

- ✓ 기존 서비스를 이용하는 데이터 처리
 - 이름을 대문자로 변환하는 데이터 처리
 - ◆ 배치 파일 수정

```
@Bean
@StepScope
public FlatFileItemReader<CustomerDTO> customerItemReader() {
    ClassPathResource resource = new
    ClassPathResource("input/customer_1.csv");
    return new FlatFileItemReaderBuilder<CustomerDTO>()
        .name("customerItemReader")
        .delimited()
        .names(new String[] {"firstName",
            "middleInitial",
            "lastName",
            "address",
            "city",
            "state",
            "zip"})
        .targetType(CustomerDTO.class)
        .resource(resource)
        .build();
}
```

Spring Batch

❖ ItemProcessor

- ✓ 기존 서비스를 이용하는 데이터 처리
 - 이름을 대문자로 변환하는 데이터 처리
 - ◆ 배치 파일 수정

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

```
@Bean
public BeanValidatingItemProcessor<CustomerDTO>
customerDTOBeanValidatingItemProcessor(){
    return new BeanValidatingItemProcessor<>();
}
```

Spring Batch

❖ ItemProcessor

- ✓ 기존 서비스를 이용하는 데이터 처리
 - 이름을 대문자로 변환하는 데이터 처리
 - ◆ 배치 파일 수정

```
@Bean
public ItemProcessorAdapter<CustomerDTO, CustomerDTO>
itemProcessor(UpperCaseNameService service) {
    ItemProcessorAdapter<CustomerDTO, CustomerDTO> adapter = new
    ItemProcessorAdapter<>();
    adapter.setTargetObject(service);
    adapter.setTargetMethod("upperCase");
    return adapter;
}
```

```
@Bean
public Step copyFileStep() {
    return this.stepBuilderFactory.get("copyFileStep")
        .<CustomerDTO, CustomerDTO>chunk(10)
        .reader(customerItemReader())
        .processor(itemProcessor(null))
        .writer(itemWriter())
        .build();
}
```

Spring Batch

❖ ItemProcessor

- ✓ 기존 서비스를 이용하는 데이터 처리
 - 이름을 대문자로 변환하는 데이터 처리
 - ◆ 배치 파일 수정

```
@Bean
public Job job() {
    return this.jobBuilderFactory.get("processingadapter")
        .start(copyFileStep())
        .build();
}
```

Spring Batch

❖ ItemProcessor

- ✓ 여러 개의 ItemProcessor 적용

@Bean

```
public CompositeltemProcessor<Customer, Customer> itemProcessor() {  
    CompositeltemProcessor<Customer, Customer> itemProcessor = new  
        CompositertemProcessor<>();  
    itemProcessor.setDelegates(Arrays.asList( customerValidatingItemProcessor(),  
        upperCaselteinProcessor(null),  
        lowerCaseltemProcessor(null)));  
    return itemProcessor;  
}
```

Spring Batch

❖ ItemProcessor

✓ 사용자 정의 ItemProcessor

● 아이템 필터링 – 홀수 우편번호만 필터링

◆ 필터링 클래스 작성

```
public class EvenFilteringItemProcessor implements  
    ItemProcessor<CustomerDTO, CustomerDTO> {  
  
    @Override  
    public CustomerDTO process(CustomerDTO item) {  
        return Integer.parseInt(item.getZip()) % 2 == 0 ? null: item;  
    }  
}
```

Spring Batch

❖ ItemProcessor

✓ 사용자 정의 ItemProcessor

- 아이템 필터링 – 홀수 우편번호만 필터링

◆ 배치 파일 수정

```
@Bean
public EvenFilteringItemProcessor itemProcessor() {
    return new EvenFilteringItemProcessor();
}
```

Spring Batch

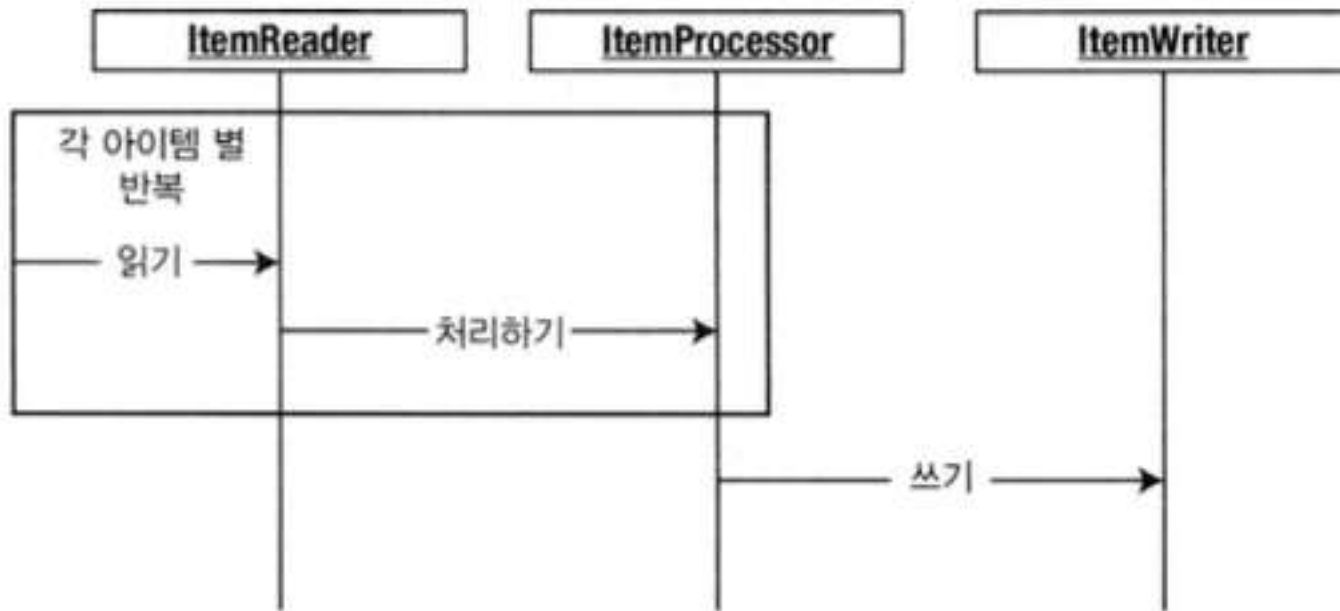
❖ ItemWriter

- ✓ ItemWriter는 스프링 배치의 출력을 담당
- ✓ 스프링 배치 처리 결과를 포맷에 맞게 출력할 필요가 있을 때 스프링 배치가 제공하는 ItemWriter를 사용
- ✓ 종류
 - 파일 기반: 파일 기반으로 출력하도록 설정하는 것은 가장 쉬운 출력 방법이며 배치 처리에 사용되는 가장 일반적인 형식 중 하나
 - RDBMS: 데이터 저장소로 사용되는 관계형 데이터베이스는 대량 데이터를 다루는 작업이 일어나면 특정한 문제가 발생할 수 있음
 - NoSQL: 최근에는 MongoDB, Apache Geode, NeO4i와 같은 NoSQL 저장소도 데이터 저장소로 많이 사용
 - 그 밖의 출력 방식: 시스템은 이메일을 보내고, JMS(Java Messaging Service) 엔드 포인트에 데이터를 쓰고 또 다른 시스템을 통해 데이터를 저장

Spring Batch

❖ ItemWriter

- ✓ 쓰기 작업을 수행할 수 있는 방법을 사용하는 스텝의 동작 방식



Spring Batch

❖ ItemWriter

✓ Flat 파일 쓰기 – csv

● 배치 파일 작성

@Configuration

@EnableBatchProcessing

@RequiredArgsConstructor

public class CsvFileService {

private final JobBuilderFactory jobBuilderFactory;

private final StepBuilderFactory stepBuilderFactory;

Spring Batch

❖ ItemWriter

✓ Flat 파일 쓰기 – txt

● 배치 파일 작성

```
@Bean
@StepScope
public FlatFileItemReader<CustomerDTO> customerItemReader() {
    ClassPathResource resource = new
    ClassPathResource("input/customer_1.csv");
    return new FlatFileItemReaderBuilder<CustomerDTO>()
        .name("customerItemReader")
        .delimited()
        .names(new String[] {"firstName",
            "middleInitial",
            "lastName",
            "address",
            "city",
            "state",
            "zip"})
        .targetType(CustomerDTO.class)
        .resource(resource)
        .build();
}
```

Spring Batch

❖ ItemWriter

- ✓ Flat 파일 쓰기 – txt
 - 배치 파일 작성

```
@Bean
public ItemWriter<CustomerDTO> itemWriter() {
    FileSystemResource resource = new FileSystemResource("output/test-
output.csv");

    return new
FlatFileItemWriterBuilder<CustomerDTO>().name("customerItemWriter")
    .resource(resource)
    .formatted()
    .format("%s %s lives at %s %s in %s, %s.")
    .names(new String[] {"firstName", "lastName", "address", "city", "state",
"zip"})
    .build();
}
```

Spring Batch

❖ ItemWriter

- ✓ Flat 파일 쓰기 – txt
 - 배치 파일 작성

```
@Bean
public Step step() {
    return this.stepBuilderFactory.get("step")
        .<CustomerDTO, CustomerDTO>chunk(10)
        .reader(customerItemReader())
        .writer(itemWriter())
        .build();
}

@Bean
public Job job() {
    return this.jobBuilderFactory.get("writingcsv")
        .start(step())
        .build();
}
}
```

Spring Batch

❖ ItemWriter

✓ Flat 파일 쓰기 – csv

● 배치 파일 작성

@Bean

```
public ItemWriter<CustomerDTO> itemWriter() {  
    FileSystemResource resource = new FileSystemResource("output/test-  
output.csv");  
  
    return new  
    FlatFileItemWriterBuilder<CustomerDTO>().name("customerItemWriter")  
        .resource(resource)  
        .delimited()  
        .delimiter(",")  
        .names(new String[] {"firstName", "lastName", "address", "city", "state",  
"zip"})  
        .build();  
}
```

Spring Batch

❖ ItemWriter

✓ Database 기록 – JDBC 활용

● 데이터베이스 작업

DROP TABLE customer;

```
CREATE TABLE customer (  
  id BIGINT PRIMARY KEY AUTO_INCREMENT,  
  firstName VARCHAR(11) NOT NULL ,  
  middleInitial VARCHAR(3),  
  lastName VARCHAR(10) NOT NULL,  
  address VARCHAR(45) NOT NULL,  
  city VARCHAR(16) NOT NULL,  
  state CHAR(2) NOT NULL,  
  zipCode CHAR(5)  
);
```

Spring Batch

❖ ItemWriter

✓ Database 기록 – JDBC 활용

● 배치 작업의 Writer 수정

@Bean

```
public JdbcBatchItemWriter<CustomerDTO> jdbcCustomerWriter(DataSource
    dataSource) {
    return new JdbcBatchItemWriterBuilder<CustomerDTO>()
        .dataSource(dataSource)
        .sql("INSERT INTO customer(firstName, middleInitial, lastName,
address, city, state, zipCode)
VALUES(:firstName, :middleInitial, :lastName, :address, :city, :state, :zip) ")
        .beanMapped()
        .build();
}
```

@Bean

```
public Step step() {
    return this.stepBuilderFactory.get("step")
        .<CustomerDTO, CustomerDTO>chunk(10)
        .reader(customerItemReader())
        .writer(jdbcCustomerWriter(null))
        .build();
}
```

Spring Batch

❖ ItemWriter

✓ Database 기록 – JDBC 활용

● 결과 확인

	id	first Name	middle Initial	last Name	address	city	state	zip Code
1	1	Richard	N	Darrow	5570 Isabella Ave	St. Louis	IL	58540
2	2	Barack	G	Donnelly	7844 S. Greenwood Ave	Houston	CA	38635
3	3	Ann	Z	Benes	2447 S. Greenwood Ave	Las Vegas	NY	55366
4	4	Laura	9S	Minella	8177 4th Street	Dallas	FL	04119
5	5	Erica	Z	Gates	3141 Farnam Street	Omaha	CA	57640
6	6	Warren	L	Darrow	4686 Mt. Lee Drive	St. Louis	NY	94935
7	7	Warren	M	Williams	6670 S. Greenwood Ave	Hollywood	FL	37288
8	8	Harry	T	Smith	3273 Isabella Ave	Houston	FL	97261
9	9	Steve	O	James	8407 Infinite Loop Drive	Las Vegas	WA	90520
10	10	Erica	Z	Neuberger	513 S. Greenwood Ave	Miami	IL	12778
11	11	Aimee	C	Hoover	7341 Vel Avenue	Mobile	AL	35928

Spring Batch

❖ ItemWriter

✓ Database 기록 – JPA 활용

● Customer Entity 작성

DROP TABLE customer;

```
CREATE TABLE customer (  
  id BIGINT PRIMARY KEY AUTO_INCREMENT,  
  first_name VARCHAR(11) NOT NULL ,  
  middle_initial VARCHAR(3),  
  last_name VARCHAR(10) NOT NULL,  
  address VARCHAR(45) NOT NULL,  
  city VARCHAR(16) NOT NULL,  
  state CHAR(2) NOT NULL,  
  zip_code CHAR(5)  
);
```

Spring Batch

❖ ItemWriter

✓ Database 기록 – JPA 활용

● Customer Entity 작성

```
@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
@Table(name = "customer")
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String firstName;
    private String middleInitial;
    private String lastName;

    private String address;
    private String city;
    private String state;
    private String zipCode;
}
```

Spring Batch

❖ ItemWriter

✓ Database 기록 – JPA 활용

● 배치 작업 수정

```
@Bean
@StepScope
public FlatFileItemReader<Customer> customerItemReader() {
    ClassPathResource resource = new
    ClassPathResource("input/customer_1.csv");
    return new FlatFileItemReaderBuilder<Customer>()
        .name("customerFileReader")
        .resource(resource)
        .delimited()
        .names(new String[] {"firstName",
                               "middleInitial",
                               "lastName",
                               "address",
                               "city",
                               "state",
                               "zip"})
        .targetType(Customer.class)
        .build();
}
```

Spring Batch

❖ ItemWriter

✓ Database 기록 – JPA 활용

● 배치 작업 수정

```
@Bean
public JpaltemWriter<Customer> jpaltemWriter(EntityManagerFactory
entityManagerFactory) {
    JpaltemWriter<Customer> jpaltemWriter = new JpaltemWriter<>();
    jpaltemWriter.setEntityManagerFactory(entityManagerFactory);
    return jpaltemWriter;
}
```

```
@Bean
public Step step() {
    return this.stepBuilderFactory.get("step")
        .<Customer, Customer>chunk(10)
        .reader(customerItemReader())
        .writer(jpaltemWriter(null))
        .build();
}
```

Spring Batch

❖ ItemWriter

✓ Database 기록 – Mongo

● 의존성 추가

implementation 'org.springframework.boot:spring-boot-starter-data-mongodb'



Spring Batch

❖ ItemWriter

✓ Database 기록 – Mongo

- application.yml 파일에 설정 추가

spring:

data:

mongodb:

host: localhost

port: 27017

database: adam

Spring Batch

❖ ItemWriter

✓ Database 기록 – Mongo

● 배치 잡 수정

```
@Bean
public MongolItemWriter<Customer> mongolItemWriter(MongoOperations
    mongoTemplate) {
    return new MongolItemWriterBuilder<Customer>()
        .template(mongoTemplate)
        .build();
}
```

```
@Bean
public Step step() {
    return this.stepBuilderFactory.get("step")
        .<Customer, Customer>chunk(10)
        .reader(customerItemReader())
        .writer(mongolItemWriter(null))
        .build();
}
```

Spring Batch

- ❖ 실행 시 배치 자동 실행 끄기
 - ✓ application.yml 파일에 추가
- ```
spring:
 batch:
 job:
 enabled: false
```



# Spring Batch

## ❖ RestController를 이용한 실행

### ✓ Controller를 생성하고 작성

@RestController

@RequiredArgsConstructor

```
public class JobLauncherController {
```

```
 private final JobLauncher jobLauncher;
```

```
 private final Job job;
```

```
 @RequestMapping("/jobLauncher")
```

```
 public void handle() throws Exception{
```

```
 jobLauncher.run(job, new JobParameters());
```

```
 }
```

```
}
```