

패스트캠퍼스 포인트 관리하기

1 Spring Batch의 구조

패스트 캠퍼스 포인트 관리하기

1.

Spring Batch의 구조

이 강의는 뭐하는 강의일까?

강의 주제

포인트를 관리하자

포인트 적립 예약 기능을 만들자!

유효기간이 지난 포인트를 만료시키자!

주요 내용

SpringBatch를 사용한 프로젝트를 경험

배치 프로그램 설계와 개발 경험

강의 방식

1. Spring Batch 개념 설명
2. 포인트 관리 프로젝트 개발
3. 프로젝트 업그레이드

부가적인 내용들

Spring Data Jpa 적용

QueryDSL 사용

Docker

Database 세팅

MySQL 쿼리 튜닝

배치 성능 개선

패스트 캠퍼스 포인트 관리하기

1.

Spring Batch의 구조

패스트 캠퍼스 포인트 관리하기

포인트를 예약적립하고 만료해보자

유저와 포인트 지갑

- 1명의 유저는 1개의 포인트 지갑을 가진다.
- 1명의 유저는 N개의 포인트를 적립한 내역을 가지고 있다.
- 적립된 포인트들은 유효기간이 모두 다르다.

포인트 만료

- 유효기간이 만료되면 해당 포인트는 사용불가한 상태가 된다.
- 유효기간은 일단위 까지만 고려하고 시간단위는 무시한다.
- 하루에 한 번 알림을 위해 유효기간이 만료된 총 금액을 구한다.
- 하루에 한 번 알림을 위해 유효기간이 1주일이내로 임박한 포인트금액을 구한다.

포인트 예약 적립

- 패스트 캠퍼스 캐시백 보상이나 이벤트와 같은 이유로 포인트 적립을 예약한다.
- 예약된 포인트는 정해진 일시에 해당 유저에게 적립한다.



Spring Batch

1.

Spring Batch의 구조

다양한 데이터 처리방식

Batch-Processing

데이터 처리시간

정해진 시간에 일괄 처리함

데이터 처리량

정해진 때에 정해진 양의 데이터를
한 번에 처리함

구현 특징

데이터를 처리할 때만 애플리케이션이
Running하도록 구현함

데이터 처리 시 어려움

데이터 볼륨이 너무 큰 경우에
처리가 어려움
처리가 특정한 시간에 집중됨

Real-Time

실시간으로 반응이 일어남
요청이 일어나면 즉시 처리함

요청을 개별적으로 처리함

Web Container에서 동작하도록 구현함

동시에 많은 요청이 일어나는 경우에
대처가 어렵습니다.

Stream-Processing

준실시간으로 반응이 일어남

Stream을 통해 데이터가 들어오면
처리하기 시작함

제 3의 도구를 사용하는 경우가 많음
(Kafka, RabbitMQ 등)

Stream의 input과 output의
flow를 컨트롤 하기 어려움
제 3의 도구와 다수의 stream으로 인해
Fail 처리가 어려움

Spring Batch

1.

Spring Batch의 구조

포인트 관리는 Batch Processing으로...

Batch란 일괄 처리란 뜻을 가지고 있습니다.

데이터를 실시간으로 처리하지 않고 일괄적으로 모아서 한번에 처리하는 방식을 말합니다.

만약에 이런 작업을 항상 Running중인 Web기반으로 구현하다면 매우 비효율적일 것입니다.

일괄 처리를 작업하는 순간에만 애플리케이션이 Running되고 리소스를 사용해야 효율적인 프로그램이라 할 수 있습니다.

포인트를 즉시 적립하는 경우도 있을수 있지만 예약적립과 포인트 유효기간 만료는 매일 한번에 데이터를 처리하는 방식입니다.

즉, 포인트 관리하기 프로젝트는 특정 시기에 특정 데이터들을 처리하면 되는 프로젝트입니다.

따라서 Batch방식으로 구현하는 것이 합리적입니다.

Spring Batch

1.

Spring Batch의 구조

Batch 중에서도 SpringBatch로...

공식문서 <https://spring.io/projects/spring-batch>

위의 공식문서를 정리하면 SpringBatch의 특징은 아래와 같습니다.

1. 가볍고 포괄적인 배치 프레임워크
2. 로깅, 추적, 트랜잭션관리, 작업처리통계, 재시작, 건너뛰기, 리소스 관리 등 대량의 레코드 처리에 필수적인 재사용 가능한 기능을 제공
3. 최적화, 파티셔닝 기술로 대용량 고성능 배치 작업 가능
4. 확장성이 매우 뛰어남

패스트 캠퍼스 포인트 관리 프로젝트를 Spring Batch로 구현하려는 이유는 무엇일까?

포인트를 적립, 만료하는 과정이 특정 일자 기준에 몰아서 처리된다는 의미에서 배치 프로그래밍이 필요합니다.

단순하게 Java Application으로 만드는 것도 가능하지만 Spring기반위에서 구현할수 없으므로 Spring에서 제공하는 많은 것들(위에서 말하는 다양한 특성)을 누릴 수가 없어 생산성이 매우 떨어지게 됩니다. 또한 위에서 말한 스프링 배치의 강력한 기능들을 사용할 수 없다는 것도 개발에 있어서 큰 문제가 됩니다.

→ 사실상 Spring Batch가 가장 적합하다고 판단할 수 있습니다.

Job

1. Spring Batch의 구조

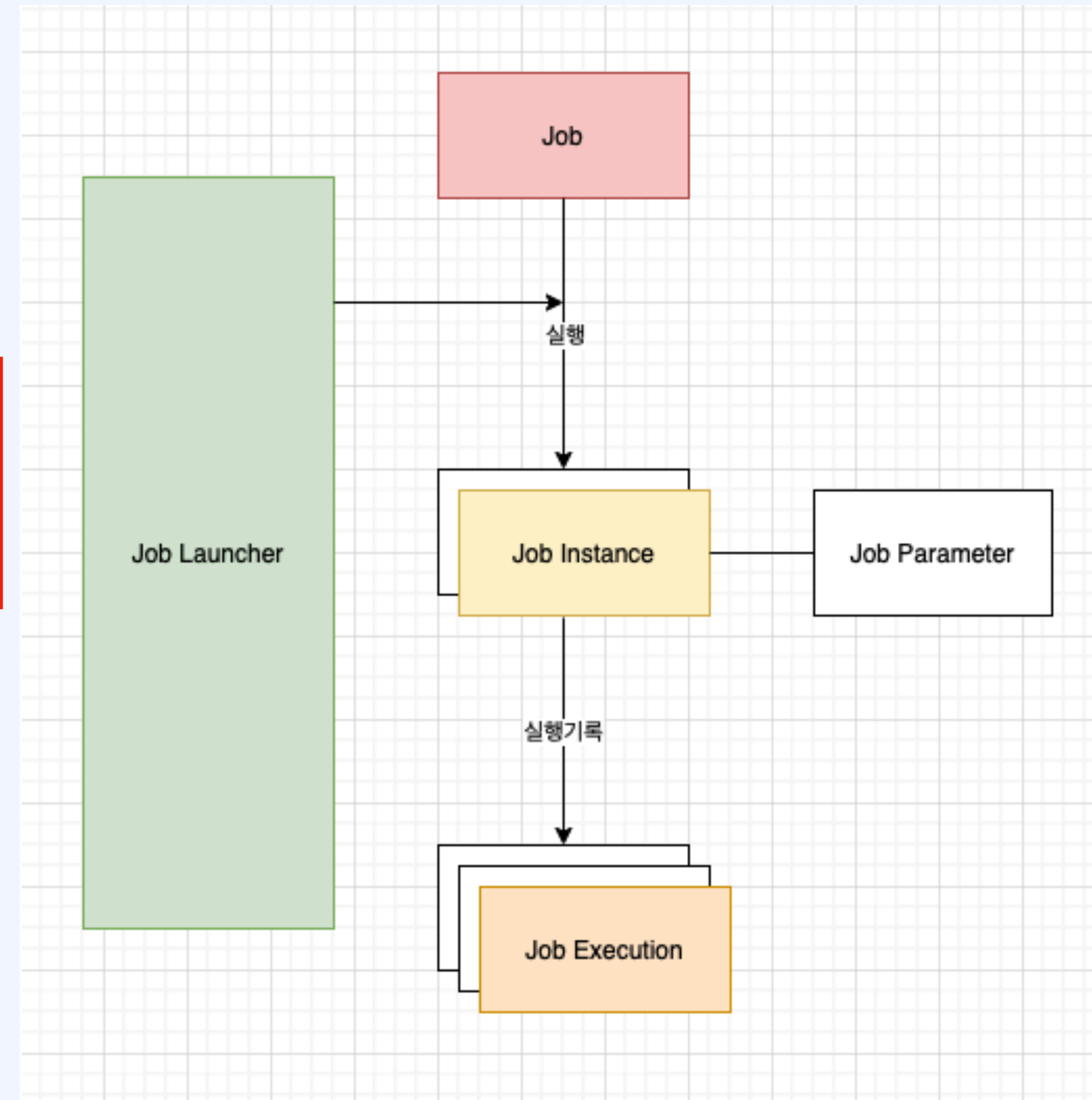
Job이란 무엇일까?

1개의 작업에 대한 명세서입니다.
어디까지가 1개 작업으로 봐야할지 기준이 애매할 수 있습니다.
그러나 그 기준은 상황별로 재량껏 판단할 수 있습니다.

Job의 특징

1개의 Job은 여러개의 Step을 포함할 수 있습니다.
Job name을 통해 Job을 구분할수 있다.
Job name으로 Job을 실행시킬 수 있습니다.
Job을 만드는 빌더는 많지만
JobBuilderFactory로 쉽게 Job을 만들수 있습니다.

Job : 식당을 예약한다.
Step 1 : 전화를 건다.
Step 2 : 예약을 한다.
Step 3 : 예약금을 송금한다.



Job

1. Spring Batch의 구조

JobInstance

Job이 명세서라면 JobInstance는 Job이 실행되어 실체화된 것입니다.
 JobInstance는 배치 처리에서 Job이 실행될 때 하나의 Job 실행 단위입니다.
 같은 Job에 같은 조건(Job Parameters)이면 JobInstance는 동일하다고 판단합니다.
 혹시, Job이 실패해서 다시 같은 조건으로 Job을 실행한다면 같은 JobInstance라고 할 수 있습니다.

```
/**
 * job Instance
 */
JobInstance jobInstance = jobExecution.getJobInstance();
// job 이름
jobInstance.getJobName();
// job instance의 ID
jobInstance.getInstanceId();
```


Job

1. Spring Batch의 구조

JobExecution

JobExecution은 JobInstance의 한번 실행을 뜻합니다.

어떤 Job이 같은 조건으로 1번 실패하고 1번 성공한다면 JobInstance는 1개로 JobExecution은 2개로 다릅니다.

JobExecution은 실패했는지 성공했는지 간에 실제로 실행시킨 사실과 동일한 의미이기 때문에 배치 실행과 관련된 정보를 포함하고 있습니다.

```
/**
 * job Execution
 * 1개의 Job Instance는 여러개의 Job Execution을 가질 수 있다.
 */
// jobExecution의 Job Instance
jobExecution.getJobInstance();
// jobExecution 에서 사용한 Job Parameters
jobExecution.getJobParameters();
// job 시작시간과 종료시간
jobExecution.getStartTime();
jobExecution.getEndTime();
// job의 실행결과 (exit code)
jobExecution.getExitStatus();
// job의 현재상태 (Batch Status)
jobExecution.getStatus();
// job execution context
jobExecution.getExecutionContext();
```

Job

1.

Spring Batch의 구조

JobExecutionContext

1개의 Job내에서 공유하는 공간(context)이다.

1개의 Job의 여러개의 Step이 있다면 그 Step들은 해당 공간을 공유할 수 있다.

```
Map<String, Object> executionContextMap = new HashMap<>();
executionContextMap.put("name", "홍길동");
executionContextMap.put("birth", LocalDate.of(1998, 1, 2));
jobExecution.setExecutionContext(new ExecutionContext(executionContextMap));
```

JobParameters

Job이 시작할 때 필요한 시작 조건을 JobParameters에 넣습니다.

동일한 Job에 JobParameters까지 동일하면 같은 JobInstance입니다.

예시)

상황 : 대한민국의 편의점리스트를 가져와서 저장하는 Job

매일 폐업하고 새로 오픈하는 편의점들 리스트 갱신

밤 10시에는 C편의점리스트 정보최신화

밤 11시에는 G편의점리스트 정보최신화

조건)

밤 10시에는 'JobParameter = C편의점' 로 구동해야됨

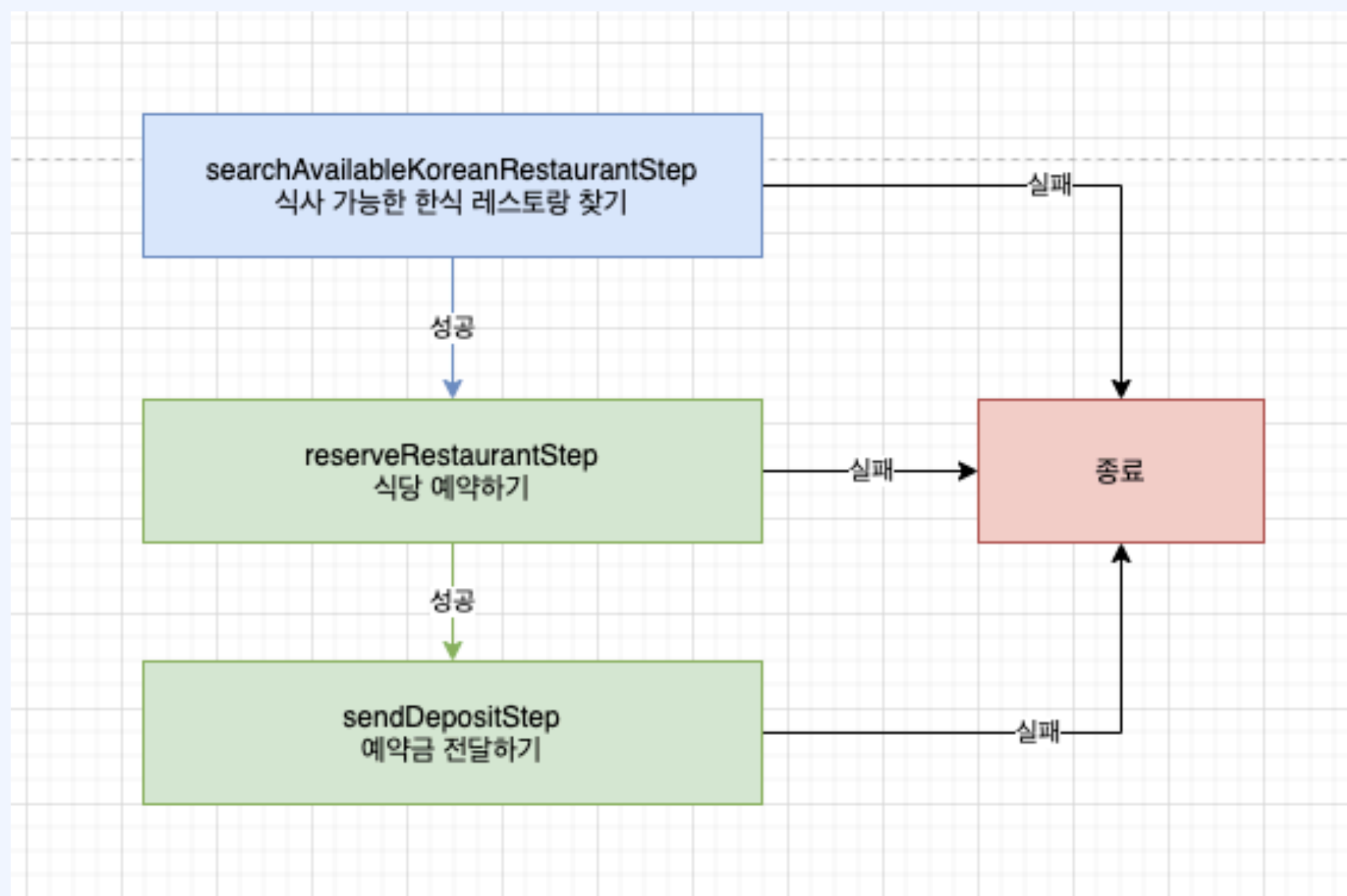
밤 11시에는 'JobParameter = G편의점' 로 구동해야됨

Job

1. Spring Batch의 구조

Job의 구현 예시

이해를 돕기위해 만들어진 예시입니다.
 외국인 관광객들을 위해서 자동으로 예약 가능한 식당을 찾아서 예약해주는 시스템을 만들고자 합니다.
 이 시스템에서는 예약이 가능한 한식 레스토랑을 우선적으로 찾아서 예약해줍니다.



```

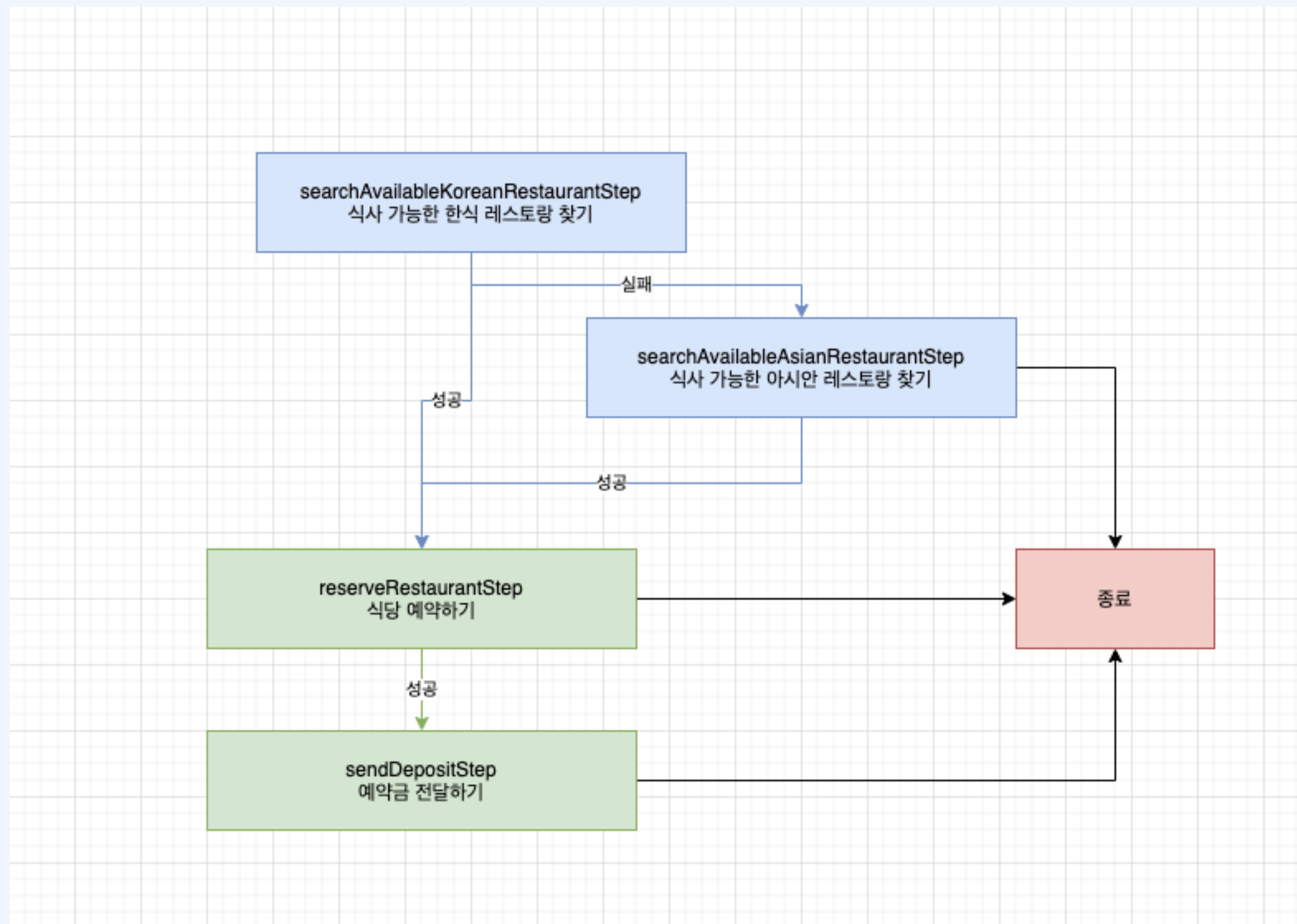
/**
 * 예약 가능한 식당을 자동으로 예약해주는 서비스
 * 단, 외국인 관광객들을 위하여 예약하다보니 한식 레스토랑만 예약한다.
 */
@Bean
public Job reserveRestaurantJob(
    JobBuilderFactory jobBuilderFactory,
    Step searchAvailableKoreanRestaurantStep,
    Step reserveRestaurantStep,
    Step sendDepositStep
) {
    return jobBuilderFactory
        .get("reserveRestaurantJob") // job name
        .start(searchAvailableKoreanRestaurantStep) // step 1
        .next(reserveRestaurantStep) // step 2
        .next(sendDepositStep) // step 3
        .build();
}
    
```

Job

1. Spring Batch의 구조

Job의 구현 예시

식사 가능한 한식 레스토랑이 없으면 식사 가능한 아시안 레스토랑을 찾습니다.



```

@Bean
public Job reserveRestaurantJob(
    // 생략
) {
    return jobBuilderFactory
        .get("reserveRestaurantJob")
        .start(searchAvailableKoreanRestaurantStep)
        .on("FAILED") // searchAvailableKoreanRestaurantStep가 FAILED인 경우
        .to(searchAvailableAsianRestaurantStep) // searchAvailableAsianRestaurantStep 실행
        .on("FAILED") // searchAvailableAsianRestaurantStep가 FAILED인 경우
        .end() // 아무것도 하지않고 flow 종료
        .from(searchAvailableKoreanRestaurantStep)
        .on("*") // searchAvailableKoreanRestaurantStep가 FAILED가 아니라면
        .to(reserveRestaurantStep) // reserveRestaurantStep 실행
        .next(sendDepositStep) // sendDepositStep 실행
        .from(searchAvailableAsianRestaurantStep)
        .on("*") // searchAvailableAsianRestaurantStep가 FAILED가 아니라면
        .to(reserveRestaurantStep) // reserveRestaurantStep 실행
        .next(sendDepositStep) // sendDepositStep 실행
        .end() // job 종료
        .build();
}
  
```

Job

1.

Spring Batch의 구조

JobBuilderFactory를 가져올 수 없다면...?

SpringBatch에서는 EnableBatchProcessing 어노테이션을 달면 아래 bean들을 사용할수 있도록 미리 구현되어있습니다.
SpringBatch 프로젝트를 만들게 된다면 필수적으로 EnableBatchProcessing를 추가해주어야합니다.

```
@EnableBatchProcessing
@SpringBootApplication
public class SpringBatchPracticeApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringBatchPracticeApplication.class);
    }
}
```

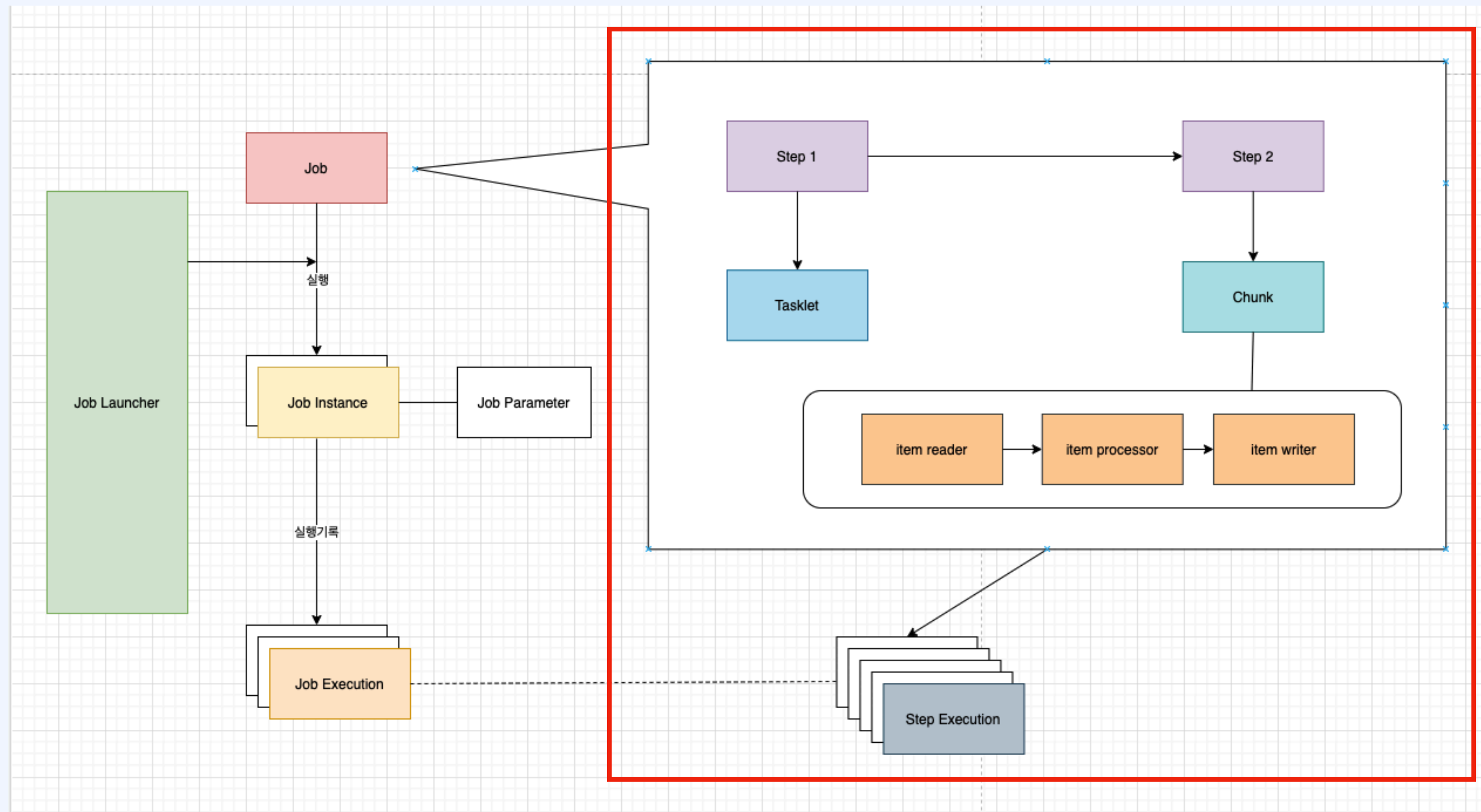
EnableBatchProcessing로 사용할 수 있게 되는 Bean들

JobRepository (bean name "jobRepository")
 JobLauncher (bean name "jobLauncher")
 JobRegistry (bean name "jobRegistry")
 JobExplorer (bean name "jobExplorer")
 PlatformTransactionManager (bean name "transactionManager")
 JobBuilderFactory (bean name "jobBuilders")
 StepBuilderFactory (bean name "stepBuilders")

Step

1. Spring Batch의 구조

Step



Step

1.

Spring Batch의 구조

Step

실질적으로 요청을 처리하는 객체입니다.

Step은 Job과 마찬가지로 행위에 대한 명세서입니다.

1개의 Job에 여러개의 Step을 포함할 수 있습니다.

따라서 1개의 JobExecution에 여러개의 StepExecution을 포함할 수 있습니다.

예시)

Job : 식당을 예약한다.

Step 1. 식당에 전화한다.

Step 2. 예약한다.

Step 3. 예약금을 송금한다.

Step

1.

Spring Batch의 구조

StepExecution

Step이라는 명세서를 실행시켜 실행된 기록입니다.

JobExecution가 Job의 실행 정보를 가지고 있는 것 처럼 StepExecution은 Step의 실행정보를 가지고 있습니다.

```
// Step의 이름
stepExecution.getStepName();

// JobExecution
stepExecution.getJobExecution();

// Step의 시작시간, 종료시간
stepExecution.getStartTime();
stepExecution.getEndTime();

// Execution Context
stepExecution.getExecutionContext();

// Step의 실행 결과
stepExecution.getExitStatus();

// Step의 현재 실행 상태 (Batch Status)
stepExecution.getStatus();
```


Step

1.

Spring Batch의 구조

StepExecutionContext

JobExecutionContext가 1개의 Job에서 공유하는 공간이면
StepExecutionContext는 1개의 Step내에서 공유하는 공간(Context)입니다.

PlatformTransactionManager

StepBuilderFactory로 Step을 정의할때 transactionManager를 받을 수 있습니다.
EnableBatchProcessing를 추가하면 아래와 같이 기본 transactionManager를 사용할 수 있습니다.
(다만, 프로젝트에서 datasource가 여러개인 경우에는 직접 별도로 transactionManager를 만들어서 사용해야 합니다.)

```
@Autowired  
PlatformTransactionManager transactionManager;
```

transactionManager를 StepBuilderFactory에 추가하면 해당 Step은 transactionManager를 사용해서 내부의 transaction을 관리합니다.

transactionManager란?

데이터베이스 트랜잭션은 데이터베이스의 데이터가 변하는 과정이 독립적이며, 일관되고 믿을수 있는걸 보장하는 걸 말합니다.

예를 들면 A가 B에게 기프티콘을 선물했습니다. 그런데 A는 결제가 되었는데 어떤 에러가 발생하여 B는 기프티콘을 받지 못했습니다.

만약에 이것이 트랜잭션으로 관리가 되었다면 이런일은 일어나지 않았을 겁니다.

두 과정을 1개의 트랜잭션으로 묶었다면 B가 기프티콘을 받지 못하면 A의 결제도 없던것이 됩니다.

바로 이런 트랜잭션을 관리해주는 것이 바로 트랜잭션 매니저이고 @EnableBatchProcessing을 통해 자동으로 기본 트랜잭션 매니저를 만들어줍니다.

Step

1. Spring Batch의 구조

JobScope

일반적으로 Scope을 지정하지 않는다면 처음 스프링부트가 시작될 때 모든 bean이 생성됩니다.

Step을 생성하는 코드에 JobScope어노테이션을 달면

Step을 스프링 부트가 시작될 때 바로 만들지 않고 lazy하게(늦게) 연관된 Job이 Step을 실행하는 시점에 만들어 집니다.

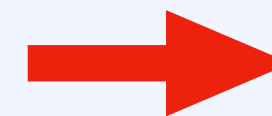
1개의 스프링배치 애플리케이션에 job1, job2, job3이 있고 3개의 job들에 연결된 step도 많이 만들어 두었는데 job1만 실행시킨다면 job1과 연관되지 않은 step들은 굳이 만들 필요가 없습니다.
굳이 필요도 없는 Step을 만드는데 리소스나 시간을 낭비할 필요가 없습니다.

Job1 - Step1-1, Step1-2, Step1-3
Job2 - Step2-1, Step2-2
Job3 - Step3-1, Step3-2

또 다른 이유로는 Job이 실행된 다음에 나중에 결정되는 값이 있다면 늦게 Step을 생성하고 싶을 것입니다.
이런경우에도 JobScope를 사용해 늦게 생성할 수 있습니다.

```
@Bean
@JobScope
public Step sampleStep() {
```

```
@Bean
public Job sampleJob(Step sampleStep) {
```



sampleJob이 실행될 때 sampleStep이 생성

Tasklet

1. Spring Batch의 구조

Tasklet

Step을 구현할 때 구현방법에는 크게 Tasklet 방식이 있고 Chunk 방식이 있습니다.
Tasklet은 Chunk보다 단순한 방식으로 단일 작업을 구현합니다.
아래는 Tasklet Interface입니다. execute를 구현해야합니다.

```
public interface Tasklet {
    @Nullable
    RepeatStatus execute(StepContribution contribution, ChunkContext chunkContext) throws Exception;
}
```

StepBuilderFactory로 Step을 만들 때 tasklet()안에다가 구현한 Tasklet을 넣어주면 됩니다.
아래 예시의 경우에는 sampleStep은 넣어준 sampleTasklet로 동작하게 됩니다.

```
return stepBuilderFactory
    .get("sampleStep")
    .transactionManager(transactionManager)
    .tasklet(sampleTasklet)
    .build();
```

Tasklet

1.

Spring Batch의 구조

Tasklet의 상태

Tasklet의 상태는 계속 진행할지, 끝낼지 두가지로만 표현됩니다.

RepeatStatus.FINISHED가 반환되면 tasklet이 바로 끝나고 RepeatStatus.CONTINUABLE이 반환되면 Tasklet을 다시 실행합니다. 따라서 RepeatStatus.CONTINUABLE를 반환한 예제는 영구적으로 끝나지 않고 계속해서 로그를 남깁니다. null을 반환하면 FINISHED와 동일하게 인지합니다.

```
/**
 * 무한히 종료되지 않는 Tasklet이다.
 * 아래와 같이 구현하면 안된다.
 */
@Bean
@StepScope
public Tasklet sampleTasklet() {
    return (contribution, chunkContext) -> {
        log.info("never ending tasklet");
        return RepeatStatus.CONTINUABLE;
    };
}
```

```
/**
 * finish를 log에 찍고 종료하는 Tasklet이다.
 */
@Bean
@StepScope
public Tasklet sampleTasklet() {
    return (contribution, chunkContext) -> {
        log.info("finish");
        return RepeatStatus.FINISHED;
    };
}
```

Tasklet

1.

Spring Batch의 구조

Tasklet을 만들 때 주의할점

오른쪽의 코드는 문제가 발생할 수 있는 포인트가 있습니다.

- 1 priceRepository.findByDate를 통해서 얼마나 많은 데이터를 가져올지 예측이 불가능합니다.
데이터가 너무 많다면 천만개의 데이터를 조회할 수도 있습니다. 천만개의 데이터를 한번에 가져온다면 메모리 이슈로 인해 처리가 불가해지고 OOM이 발생할 수도 있습니다.
- 2 Tasklet 형식의 Step에 transactionManager를 추가하게 되면 해당 Tasklet은 Transaction에 묶이게 됩니다. 이때 Tasklet에서 너무 많은 데이터를 불러오고 쓰게 되면 Tasklet의 Transaction은 너무 거대해지게되어 Database Transaction 1개가 처리해야할 일이 너무 많아지게 됩니다.

대부분 이런 경우에는 Chunk Processing 을 사용하게 되면 해결됩니다.

```
@Bean
@StepScope
public Tasklet hugeReadTasklet(
    PriceRepository priceRepository
) {
    return (contribution, chunkContext) -> {
        // findByDate로 조회된 데이터가 1천만개 입니다.
        List<Price> prices = priceRepository.findByDate(LocalDate.now());
        // price를 모두 0으로 초기화합니다.
        prices.forEach(price -> price.setAmount(0L));
        // 1천만개의 데이터를 저장합니다.
        priceRepository.saveAll(prices);
        return RepeatStatus.FINISHED;
    };
}
```

Chunk Processing

1.

Spring Batch의 구조

Chunk 프로세싱의 필요성

Batch 프로세싱의 가장 큰 특징이 일괄 처리이면서 동시에 가장 큰 문제가 일괄 처리입니다. 일괄로 한번에 데이터를 처리한다는 것은 시스템의 리소스가 한순간에 많이 필요하다는 것을 말합니다. 오늘 만료 시켜야할 포인트가 십만개면 어떨까요? 서비스가 대성공해서 백만개, 천만개면 어떨까요? 그 어떤 서버도 한번에 천만개를 처리하기 쉽지 않습니다.

이를 해결하기 위해서 spring batch에서는 chunk라는 개념을 만들었습니다. chunk는 일정 개수만큼 잘라서 처리하겠다는 의미로 chunk size가 1000 이면 한번에 1000개씩 처리하고 완료하고 그다음 1000개 처리하고 완료하겠다는 뜻입니다. 이렇게 하면 한순간에는 1000개에 해당하는 리소스만 있으면 됩니다.

우리가 구현하려는 포인트도 마찬가지입니다. 만료하려는 포인트가 얼마나 될지는 예측하기 어렵습니다. 서비스 초기에는 하루에 몇건씩 만료될 수 있고 서비스가 크게 성공해서 수백만건씩 만료될 수 도 있습니다. 하지만 대부분의 경우에는 사업이 잘될 것이라는 예측 속에서 서비스를 만들기 때문에 chunk 프로세스로 처리하도록 구현합니다.

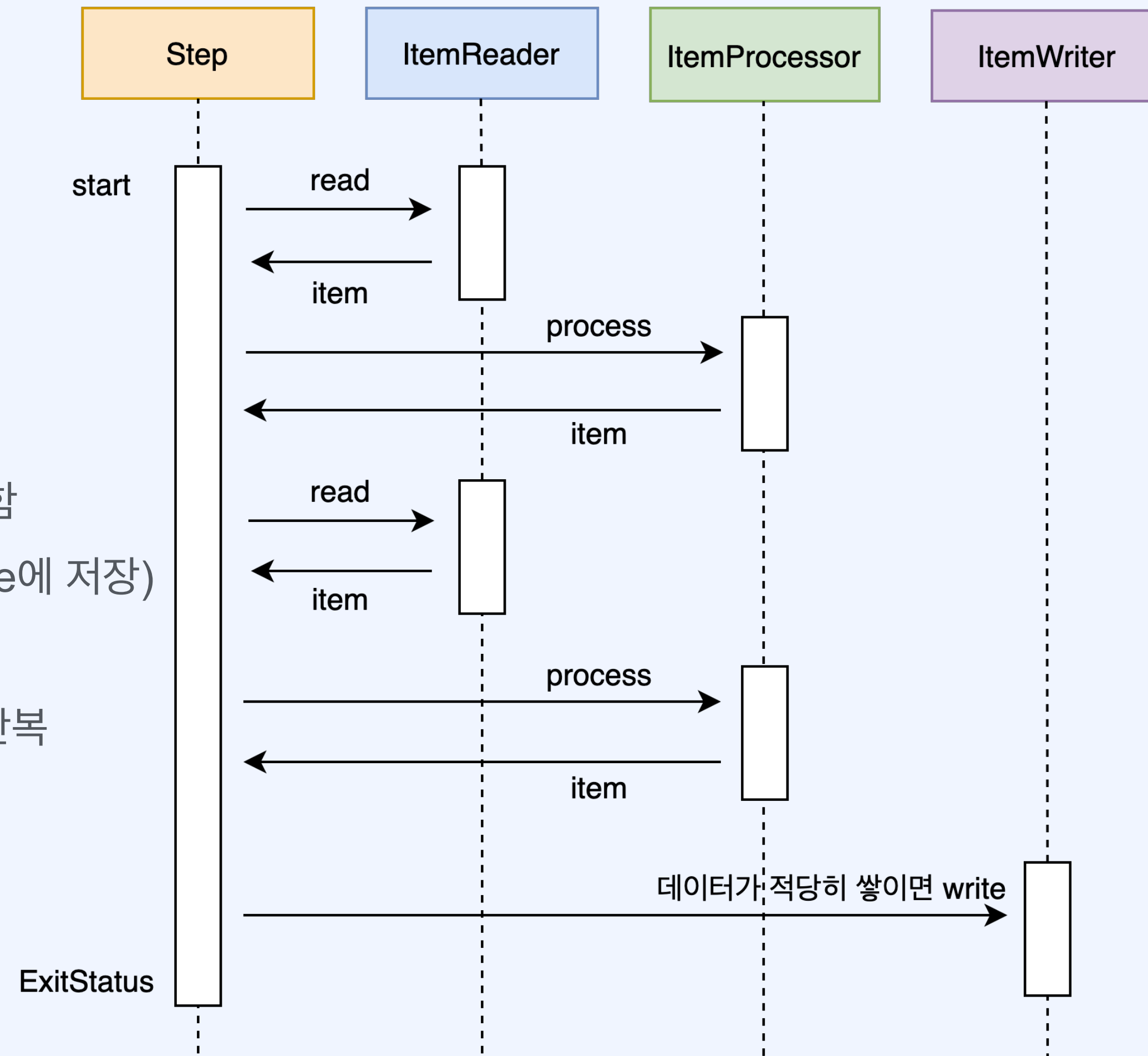
우리도 이번 프로젝트에서는 chunk 프로세스로 spring batch를 구현할 것입니다.

Chunk Processing

1. Spring Batch의 구조

일반적인 Chunk 기반 Step 흐름 이해하기

1. 트랜잭션 시작
2. Item Reader가 데이터 1개 제공하기
3. Item Processor를 통해 데이터 1개를 가공하기
4. chunk size만큼 데이터가 쌓일때 까지 2~3를 반복함
5. Writer에게 데이터 전달하기 (보통의 경우 Database에 저장)
6. 트랜잭션 종료
7. 2이 더이상 진행될수 없을때까지 1~6를 계속 해서 반복



Chunk Processing

1.

Spring Batch의 구조

Chunk Processing 구현방법

Step을 Chunk방식으로 구현하기 위해서는
StepBuilderFactory에서 <A,B>chunk(100)를 사용하면됩니다.
그리고 reader, processor, writer를 넣어주면 됩니다.

Chunk processing을 구현할 때는 ItemReader,
ItemProcessor, ItemWriter를 구현해야 하지만 3가지 모두가 필
수는 아닙니다.

ItemReader와 ItemWriter만 필수이고 ItemProcessor는 구현
하지 않아도 상관 없습니다.

Chunk Processing와 ItemReader, ItemProcessor,
ItemWriter는 아래와 같이 제네릭 Type이 일치해야 합니다.

<A, B>chunk(1000)

ItemReader<A>

ItemProcessor<A, B>

ItemWriter

```
@Bean
@JobScope
public Step saveOrderedPriceStep(
    StepBuilderFactory stepBuilderFactory,
    PlatformTransactionManager transactionManager,
    JpaPagingItemReader<Order> orderReader,
    ItemProcessor<Order, Price> orderToPriceProcessor,
    ItemWriter<Price> priceWriter
) {
    return stepBuilderFactory.get("saveOrderedPriceStep")
        .transactionManager(transactionManager)
        // Order를 read해서 Price로 process한뒤 Price를 Write한다.
        .<Order, Price>chunk(1000)
        // 주문된 데이터 read하기
        .reader(orderReader)
        // 주문정보에서 가격으로 변환
        .processor(orderToPriceProcessor)
        // 가격을 write
        .writer(priceWriter)
        .build();
}
```


Chunk Processing

1.

Spring Batch의 구조

Chunk Size는 얼마가 적당할까요?

Chunk Size는 한번에 끊어서 처리할 수 있는 크기를 말합니다. 그렇다면 그 크기는 얼마가 적당할까요?

이것은 딱히 정해진게 없습니다. 처리하려는 업무의 종류, 코드의 로직, Batch가 구동되는 환경 등에 따라 다릅니다. chunk size가 너무 작으면 일괄처리에서의 효율이 떨어지고 chunk가 너무 커도 리소스문제나 처리량의 한계 등의 문제가 있을 수 있습니다. 적당한 크기의 chunk size를 찾는 것도 Batch성능 개선에 큰 도움이 됩니다.

다시 알아보는 StepExecutionContext

StepExecutionContext를 사용하면 1개의 Step안에서 공유하는 공간을 만들수 있다고 했습니다.

즉, 1개의 Step안에 있는 ItemReader, ItemProcessor, ItemWriter가 같은 공간을 접근할수 있습니다.

다시 알아보는 PlatformTransactionManager

```
@Autowired  
PlatformTransactionManager transactionManager;
```

@EnableBatchProcessing을 달면 기본 트랜잭션 매니저를 가져올수 있고 이걸 StepBuilderFactory에서 등록할 수 있다고 했습니다.

이 트랜잭션 매니저를 StepBuilderFactory에서 사용하게 되면 1개 chunk 단위로 트랜잭션이 생기게 됩니다. 따라서 1개 chunk가 끝나면 일괄적으로 트랜잭션이 끝나게 되고 ItemWriter에서 저장한 모든 대상들의 commit은 chunk가 끝나면 발생합니다.

ItemReader

1.

Spring Batch의 구조

ItemReader

chunk 프로세싱에서 데이터를 제공하는 interface입니다.

ItemReader는 반드시 read메소드를 구현해야합니다.

read메소드를 통해서 ItemProcessor 또는 ItemWriter에게 데이터를 제공합니다.

read메소드가 null을 반환하면 더이상 데이터가 없고 step을 끝내겠다고 판단합니다.

그렇기 때문에 처음부터 null을 반환했다고 하더라도 에러가 나지 않습니다.(단, 데이터가 없으니 Step은 바로 종료되겠죠)

```
public interface ItemReader<T> {  
    @Nullable  
    T read() throws Exception, UnexpectedInputException, ParseException, NonTransientResourceException;  
}
```

ItemReader

ItemReader가 데이터를 가져오는 방법

```
public interface ItemReader<T> {  
    @Nullable  
    T read() throws Exception, UnexpectedInputException, ParseException, NonTransientResourceException;  
}
```

ItemReader의 read를 보면 1개를 반환합니다.

1개씩 계속해서 읽고 더이상 읽을수 없을 때 까지 (null이 나올때까지) 반복합니다. 그렇다고 해서 ItemReader가 데이터를 반드시 1개씩 조회한다는 뜻은 아닙니다.

ItemReader의 데이터 조회 방식을 크게 두가지로 나눌수 있습니다.

1. 정말 1개씩 데이터를 가져와서 read의 결과로 주는 방식이 있습니다.
2. 한번에 대량으로 가져오고 가져온 데이터에서 하나씩 빼주는 방식이 있습니다.
(단, 대량으로 가져올때 최대 가져올수 있는 개수는 정해져있어야합니다.)

ItemReader가 데이터를 가져오는 대상

ItemReader가 데이터를 가져오는 대상은 정말 다양할 수 있습니다.

그리고 원하면 얼마든지 어느곳에서든 데이터를 가져올 수 있는 ItemReader를 만들 수 있습니다.

그러나 보통은 file과 database에서 데이터를 가져오는 경우가 대부분입니다.

ItemReader

1.

Spring Batch의 구조

Flat File

보통 구분자로 나누어져 있는 파일을 읽습니다.

그 예로 csv(comma-separated values) 파일이 있습니다.

name,age,grade

홍길동,20,A

김철수,23,B

```
@Bean
@StepScope
public FlatFileItemReader<Point> pointFlatFileItemReader() {
    return new FlatFileItemReaderBuilder<Point>()
        .name("pointFlatFileItemReader")
        .resource(new FileSystemResource(filePath))
        .delimited()
        .delimiter(",")
        .names("id", "amount")
        .targetType(Point.class)
        .recordSeparatorPolicy(
            new SimpleRecordSeparatorPolicy() {
                @Override
                public String postProcess(String record) {
                    return record.trim();
                }
            }
        )
        .build();
}
```

ItemReader

1. Spring Batch의 구조

데이터베이스

JdbcCursorItemReader

Jdbc Cursor로 조회하여 결과를 Object에 Mapping해서 넣어주는 방식입니다.

```
@Bean
@StepScope
public JdbcCursorItemReader<Point> pointJdbcCursorItemReader() {
    return new JdbcCursorItemReaderBuilder<Point>()
        .fetchSize(1000)
        .dataSource(dataSource)
        .rowMapper(new BeanPropertyRowMapper<>(Point.class))
        .sql("SELECT id, amount FROM point")
        .name("pointJdbcCursorItemReader")
        .build();
}
```

ItemReader

1.

Spring Batch의 구조

데이터베이스

JdbcPagingItemReader

```
@Bean
@StepScope
public JdbcPagingItemReader<Point> pointJdbcPagingItemReader(
    PagingQueryProvider pointQueryProvider
) throws Exception {
    Map<String, Object> parameterValues = new HashMap<>();
    parameterValues.put("amount", 100);
    return new JdbcPagingItemReaderBuilder<Point>()
        .name("pointJdbcPagingItemReader")
        .pageSize(chunkSize)
        .fetchSize(chunkSize)
        .dataSource(dataSource)
        .rowMapper(new BeanPropertyRowMapper<>(Point.class))
        .queryProvider(createQueryProvider)
        .parameterValues(parameterValues)
        .build();
}
```

```
@Bean
@StepScope
public PagingQueryProvider pointQueryProvider() throws Exception {
    SqlPagingQueryProviderFactoryBean queryProvider =
        new SqlPagingQueryProviderFactoryBean();
    queryProvider.setDataSource(dataSource);
    // Database에 맞는 PagingQueryProvider를 선택하기 위해
    queryProvider.setSelectClause("id, amount");
    queryProvider.setFromClause("from point");
    queryProvider.setWhereClause("where amount >= :amount");
    Map<String, Order> sortKeys = new HashMap<>(1);
    sortKeys.put("id", Order.ASCENDING);
    queryProvider.setSortKeys(sortKeys);
    return queryProvider.getObject();
}
```

ItemReader

1. Spring Batch의 구조

데이터베이스

JpaPagingItemReader

Jpa Pagination을 사용하여 데이터를 조회합니다.

실제 Database에 조회하는 쿼리에 추가적인 조건이 붙는다.

MySQL의 경우에는 limit 조건이 붙고

Oracle의 경우에는 rownum 조건이 붙는다.

```
@Bean
@StepScope
public JpaPagingItemReader pointJpaPagingItemReader() {
    return new JpaPagingItemReaderBuilder<Point>()
        .name("pointJpaPagingItemReader")
        .entityManagerFactory(entityManagerFactory())
        .queryString("select p from Point p")
        .pageSize(1000)
        .build();
}
```


ItemReader

1.

Spring Batch의 구조

데이터베이스

RepositoryItemReader

Repository에서 구현한 메소드의 이름을 넣는 방식입니다.
주의할 점은 이 메소드의 인자에 pageable이 포함되어있어서 Pagination을 지원하는 상황이어야합니다.

```
@Bean
@StepScope
public RepositoryItemReader<Point> pointRepositoryItemReader(
    PointRepository pointRepository
) {
    return new RepositoryItemReaderBuilder()
        .repository(pointRepository)
        .methodName("findByAmountGreaterThan")
        .pageSize(1000)
        .maxItemCount(1000)
        .arguments(Arrays.asList(BigInteger.valueOf(100)))
        .sorts(Collections.singletonMap("id", Sort.Direction.ASC))
        .name("pointRepositoryItemReader")
        .build();
}

public interface PointRepository extends JpaRepository<Point, Long> {
    Page<Point> findByAmountGreaterThan(Long amount, Pageable pageable);
}
```


ItemProcessor

1. Spring Batch의 구조

ItemProcessor

ItemProcessor는 ItemReader에서 read()로 넘겨준 데이터를 개별로 가공합니다.

ItemProcessor는 언제 사용할까?

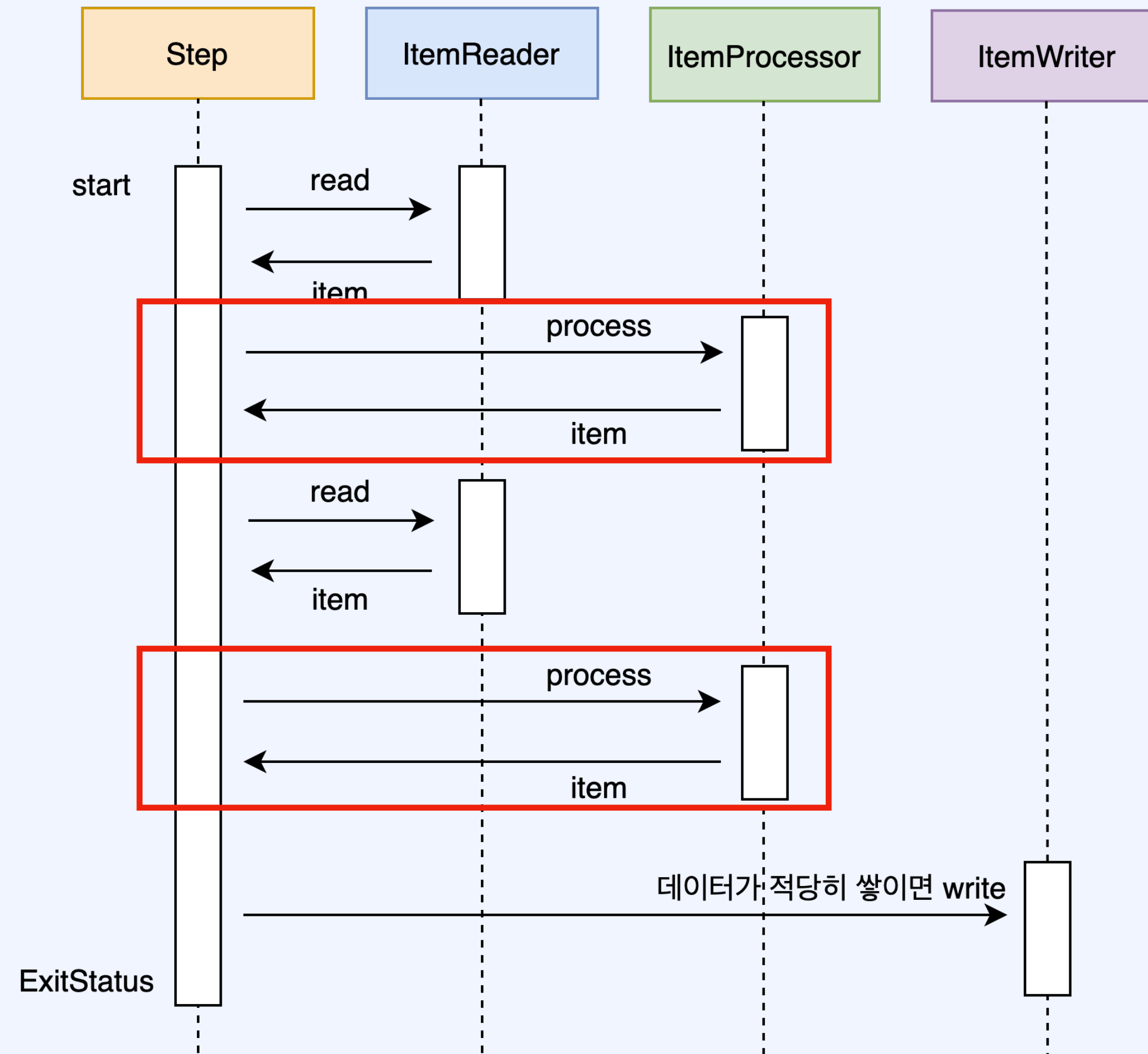
대부분 ItemProcessor는 두가지 이유로 사용합니다.

1. ItemReader가 넘겨준 데이터를 가공하려고 할때
2. 데이터를 Writer로 넘길지 말지 결정할때
(null을 return하면 writer에 전달하지 않음)

ItemProcessor는 필수가 아닙니다.

ItemProcessor가 필요가 없는 경우

1. 정말 ItemProcessor가 필요 없는 경우
예를 들면 ItemReader가 Database에서 데이터를 읽은뒤 수정없이 그대로 ItemWriter에서 File에 Write하는 경우에는 ItemProcessor가 굳이 필요하지 않습니다.
2. ItemReader나 ItemWriter에서 데이터를 직접 가공까지 하는 경우
추천하는 방식은 아니지만 상황에 따라 어쩔수 없이 ItemReader에서 read할 때 수정한 데이터를 넘겨주는 경우도 있고 ItemWriter에서 쓰기 전 데이터를 수정해서 write하는 경우가 있습니다.



ItemProcessor

1. Spring Batch의 구조

ItemProcessor는 어떻게 구현할까?

ItemProcessor는 process를 구현하면 됩니다.

ItemProcessor는 Input과 Output이 있으며 Input(item)을 받아서 Output형식으로 변환한뒤 반환해야합니다.

이때 Input과 Output의 type은 같을 수도 있고 다를 수도 있습니다.

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

```
/**
 * 주문정보(Order)로 10000원이 넘는 가격(Price)을 찾는 Processor
 */
@Bean
@StepScope
public ItemProcessor<Order, Price> findExpensivePriceProcessor(
    PriceRepository priceRepository
) {
    return order -> {
        Price price = priceRepository.findByProductId(order.productId);
        if (price.amount > 10000)
            return price;
        else
            return null // writer에 데이터를 넘기지 않는다.
    };
}
```

ItemProcessor

1. Spring Batch의 구조

CompositeItemProcessor

ItemProcessor 여러개를 체인처럼 연결할 때 사용합니다.

read를 통해 나온 item은 processor1 → processor2 를 연속적으로 통과합니다.

processor의 코드와 역할을 나누고자 할 때 사용합니다.

```
@Bean
@StepScope
public CompositeItemProcessor compositeProcessor(
    ItemProcessor<Order, Order> processor1,
    ItemProcessor<Order, Price> processor2
) {
    List<ItemProcessor> delegates = List.of(processor1, processor2);
    CompositeItemProcessor processor = new CompositeItemProcessor<>();
    processor.setDelegates(delegates);
    return processor;
}
```

ItemWriter

1.

Spring Batch의 구조

ItemWriter

Chunk Processing의 마지막 단계로 Item을 쓰는 단계입니다.

ItemReader와 ItemProcessor를 거쳐 처리된 Item을 Chunk 단위 만큼 처리한 뒤 이를 ItemWriter에 전달합니다.

ItemWriter는 item 1개가 아니라 데이터의 묶음인 Item List를 처리합니다.

ItemWriter가 쓰기를 하는 대상은 그 어떤것도 될수 있습니다.

파일이나 RDBMS, NoSQL에 데이터를 쓸수도 있고 다른 API 호출을 할수도 있습니다.

```
public interface ItemWriter<T> {  
    void write(List<? extends T> items) throws Exception;  
}
```

ItemWriter

1. Spring Batch의 구조

JdbcBatchItemWriter

```
public JdbcBatchItemWriter<Point> jdbcBatchItemWriter() {
    return new JdbcBatchItemWriterBuilder<Point>()
        .dataSource(dataSource)
        .sql("insert into point(point_wallet_id, amount) values (:point_wallet_id, :amount)")
        .beanMapped()
        .build();
}
```

JpaItemWriter

```
@Bean
public JpaItemWriter<Point> jpaItemWriter() {
    JpaItemWriter<Point> jpaItemWriter = new JpaItemWriter<>();
    jpaItemWriter.setEntityManagerFactory(entityManagerFactory);
    return jpaItemWriter;
}
```


ItemWriter

1. Spring Batch의 구조

ItemWriter 커스텀

```
public interface ItemWriter<T> {
    void write(List<? extends T> items) throws Exception;
}
```

```
@Bean
@StepScope
public ItemWriter<Point> expirePointWriter(
    PointRepository pointRepository
) {
    return points -> {
        pointRepository.saveAll(points);
    };
}
```



ItemWriter

1. Spring Batch의 구조

왜 ItemWriter는 List로 처리할까?

```
public interface ItemReader<T> {
    @Nullable
    T read();
}
```

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

```
public interface ItemWriter<T> {
    void write(List<? extends T> items) throws Exception;
}
```

ItemReader의 read와 ItemProcessor를 보면 1개를 반환합니다.

그러나 ItemWriter는 List로 처리하는데 왜그런 걸까요?

ItemWriter에서는 대부분 쓰기작업이 일어납니다.

이런 쓰기 작업을 건별로 처리하면 효율이 떨어지고 성능에 문제가 되는 경우가 많습니다.

예를 들어 ItemWriter에서 Database에 insert한다고 하면 1000개의 데이터를 한번에 저장하는 것과 개별 트랜잭션으로 1개씩 저장하는 것은 많은 성능 차이를 만듭니다.

그럼에도 불구하고 건별로 처리하고자 한다면 items를 받아서 element를 건별로 처리하도록 ItemWriter를 구현하면 됩니다.

Listener

1.

Spring Batch의 구조

Listener

Spring Batch에서는 메인 로직 외에 구간 사이사이에 어떤 일을 처리하고자 할때 Listener를 사용합니다.

예를 들면 Job, Step, chunk, ItemReader, ItemWriter, ItemProcessor의 실행 직전과 직후에 어떤 행위를 할지 정의할 수 있습니다.

Listener의 종류

JobExecutionListener

StepExecutionListener

ChunkListener

ItemReadListener

ItemProcessListener

ItemWriteListener

Listener

1. Spring Batch의 구조

JobExecutionListener

Job의 실행 전 beforeJob / 실행 후 afterJob을 구현합니다.

인자로 JobExecution을 넘겨줍니다.

구현된 JobExecutionListener는 아래처럼 jobBuilderFactory에서 Job을 만들 때 listener()에 포함시켜주면 됩니다.

```
public interface JobExecutionListener {
    void beforeJob(JobExecution jobExecution);
    void afterJob(JobExecution jobExecution);
}
```

```
@Bean
public Job sampleJob(
    JobBuilderFactory jobBuilderFactory,
    JobListener jobListener,
    Step sampleStep
) {
    return jobBuilderFactory.get("sampleJob")
        .listener(jobListener)
        .start(sampleStep)
        .build();
}
```

Listener

1.

Spring Batch의 구조

JobExecutionListener

여기서 jobExecution을 인자로 주는데 job의 상태를 확인하고 관리할 수 있고 jobExecutionContext도 수정할수 있습니다.

```
public class JobListener implements JobExecutionListener {
    @Override
    public void afterJob(JobExecution jobExecution) {
        // jobExecution의 Job Instance
        jobExecution.getJobInstance();
        // jobExecution 에서 사용한 Job Parameters
        jobExecution.getJobParameters();
        // job 시작시간과 종료시간
        jobExecution.getStartTime();
        jobExecution.getEndTime();
        // job의 실행결과 (exit code)
        jobExecution.getExitStatus();
        // job의 현재상태 (Batch Status)
        jobExecution.getStatus();
        // job execution context
        jobExecution.getExecutionContext();
    }
}
```

```
public class JobListener implements JobExecutionListener {
    @Override
    public void beforeJob(JobExecution jobExecution) {
        // job execution context
        jobExecution.getExecutionContext();
        Map<String, Object> executionContextMap = new HashMap<>();
        executionContextMap.put("name", "홍길동");
        executionContextMap.put("birth", LocalDate.of(1998, 1, 2));
        jobExecution.setExecutionContext(
            new ExecutionContext(executionContextMap)
        );
    }
}
```

Listener

1.

Spring Batch의 구조

StepExecutionListener

Step의 실행 전 beforeStep / 실행 후 afterStep을 구현합니다.

인자로 StepExecution을 넘겨줍니다.

구현된 StepExecutionListener는 아래처럼 StepBuilderFactory에서 Step을 만들 때 listener()에 포함시켜주면 됩니다.

```
public interface StepExecutionListener extends StepListener {
    void beforeStep(StepExecution stepExecution);
    @Nullable
    ExitStatus afterStep(StepExecution stepExecution);
}
```

```
@Bean
@JobScope
public Step sampleTaskletStep(
    StepBuilderFactory stepBuilderFactory,
    PlatformTransactionManager transactionManager,
    StepListener stepListener,
    Tasklet sampleTasklet
) {
    return stepBuilderFactory.get("sampleTaskletStep")
        .transactionManager(transactionManager)
        .listener(stepListener)
        .tasklet(sampleTasklet)
        .build();
}
```

Listener

1. Spring Batch의 구조

StepExecutionListener

StepExecution을 인자로 주는데 step의 상태를 확인하고 관리할 수 있고 stepExecutionContext도 수정할수 있습니다.

```
public class StepListener implements StepExecutionListener {
    @Override
    public void afterStep(StepExecution stepExecution) {
        // Step의 이름
        stepExecution.getStepName();
        // JobExecution
        stepExecution.getJobExecution();
        // Step의 시작시간, 종료시간
        stepExecution.getStartTime();
        stepExecution.getEndTime();
        // Execution Context
        stepExecution.getExecutionContext();
        // Step의 실행 결과
        stepExecution.getExitStatus();
        // Step의 현재 실행 상태 (Batch Status)
        stepExecution.getStatus();
    }
}
```

Listener

1.

Spring Batch의 구조

ChunkListener

1개의 chunk가 돌기 전, 후 그리고 에러 발생시 어떤 행위를 할 수 있는지 정의할 수 있습니다.

```
public interface ChunkListener extends StepListener {  
    static final String ROLLBACK_EXCEPTION_KEY = "sb_rollback_exception";  
    void beforeChunk(ChunkContext context);  
    void afterChunk(ChunkContext context);  
    void afterChunkError(ChunkContext context);  
}
```

ItemReadListener

ItemReader가 1개의 Item을 read를 하기 전, 후 그리고 에러발생시 어떤 행위를 할 수 있는지 정의할 수 있습니다.

```
public interface ItemReadListener<T> extends StepListener {  
    void beforeRead();  
    void afterRead(T item);  
    void onReadError(Exception ex);  
}
```

Listener

1. Spring Batch의 구조

ItemProcessListener

ItemProcessor가 1개의 Item을 process 하기 전, 후 그리고 에러발생시 어떤 행위를 할 수 있는지 정의할 수 있습니다.

```
public interface ItemProcessListener<T, S> extends StepListener {
    void beforeProcess(T item);
    void afterProcess(T item, @Nullable S result);
    void onProcessError(T item, Exception e);
}
```

ItemWriteListener

ItemWriter가 item List를 처리하기 전, 후 그리고 에러발생시 어떤 행위를 할 수 있는지 정의할 수 있습니다.

```
public interface ItemWriteListener<S> extends StepListener {
    void beforeWrite(List<? extends S> items);
    void afterWrite(List<? extends S> items);
    void onWriteError(Exception exception, List<? extends S> items);
}
```

SpringBatchTest

1.

Spring Batch의 구조

JobLauncherTestUtils

SpringBatch를 테스트 할 때는 JobLauncherTestUtils를 사용합니다.

jobLauncherTestUtils를 아래와 같이 사용하면 되는데

launchJob을 하게되면 job이 실행됩니다.

```
JobLauncherTestUtils jobLauncherTestUtils = new JobLauncherTestUtils();
jobLauncherTestUtils.setJob(job);
jobLauncherTestUtils.setJobLauncher(jobLauncher);
jobLauncherTestUtils.setJobRepository(jobRepository);
jobLauncherTestUtils.launchJob(jobParameters);
```

패스트 캠퍼스 포인트 만료하기 프로젝트의 *BatchTestSupport* 코드를 참고하면 좋습니다.

JobLauncherTestUtils는 spring-batch-test 의존성을 추가하면 사용할 수 있습니다.

```
testImplementation 'org.springframework.batch:spring-batch-test'
```


SpringBatchTest

1. Spring Batch의 구조

JobLauncherTestUtils

JobLauncherTestUtils.launchJob을 통해 반환되는 값은 JobExecution입니다.

JobExecution의 ExitStatus를 확인함으로써 Job이 성공적으로 끝났는지 확인할 수 있습니다.

```
JobExecution launchJob(Job job, JobParameters jobParameters) throws Exception {
    JobLauncherTestUtils jobLauncherTestUtils = new JobLauncherTestUtils();
    jobLauncherTestUtils.setJob(job);
    jobLauncherTestUtils.setJobLauncher(jobLauncher);
    jobLauncherTestUtils.setJobRepository(jobRepository);
    return jobLauncherTestUtils.launchJob(jobParameters == null ? new JobParametersBuilder().toJobParameters() : jobParameters);
}
```

```
JobExecution execution = launchJob(messageExpiredPointJob, jobParameters);
```

```
// then
```

```
then(execution.getExitStatus()).isEqualTo(ExitStatus.COMPLETED);
```


SpringBatchTest

1.

Spring Batch의 구조

SpringBatchTest

SpringBatchTest 어노테이션은 Spring 4.1 이후부터 추가되었습니다.

@SpringBatchTest를 테스트 클래스 위에 달게되면 테스트하기전에 필요한 수많은 작업들을 미리 해줍니다.

JobLauncherTestUtils도 자동으로 만들어주기 때문에 그냥 Bean으로 가져다 쓰면됩니다.

```
@RunWith(SpringRunner.class)
@SpringBatchTest
@ContextConfiguration(classes = {SampleJobConfiguration.class})
@ActiveProfiles("test")
public class SampleBatchTest {
    @Autowired
    private JobLauncherTestUtils jobLauncherTestUtils;

    @Test
    void launchJobTest() throws Exception {
        JobExecution jobExecution = jobLauncherTestUtils.launchJob(jobParameters);
        then(execution.getExitStatus()).isEqualTo(ExitStatus.COMPLETED);
    }
}
```

SpringBatchTest

1.

Spring Batch의 구조

@SpringBatchTest를 사용할까요?

@SpringBatchTest에 있는 JobLauncherTestUtils를 사용하기 위해서는 테스트가 Scan하는 범위내에 단 1개의 Job만 Bean으로 등록이 되어있어야합니다.

1개의 Job만 Bean으로 등록하는 것은 테스트할 때 @ContextConfiguration(classes=SampleJobConfiguration.class)와 같이 테스트 하고 싶은 Job만 scan범위를 제한하면 됩니다.

그러나 우리가 테스트 코드를 만들다보면 편의상 어떤 Bean이 필요한지 선택하기가 어려워서 @SpringBootTest로 모든 Job을 생성하는 경우가 많습니다. (Bean의 관계가 복잡한 경우에는 더욱 그렇겠죠)

이렇게 되면 @SpringBatchTest를 사용할 수 없기 때문에 선택의 기로에 서게 됩니다.

패스트캠퍼스포인트 만료하기 프로젝트에서는 @SpringBatchTest를 사용하지 않고 JobLauncherTestUtils를 직접 생성해서 테스트하도록 하겠습니다.