Spring Batch

Neden büyük ölçekli batch işlemlerinin `@Scheduled`'ın tek başına sağlayamadığı bir yapıya ihtiyacı olduğu -- Job/Step/chunk-odaklı işleme zihinsel modeli, tek bir CSV-'den-veritabanına sipariş içe aktarma örneği boyunca ItemReader/ItemProcessor/ItemWriter, JobParameters/JobInstance/JobExecution ayrımı, JobRepository ile yeniden başlatılabilirlik, skip/retry fault tolerance, ve `@Scheduled` ile pratik birleşim. Advanced Spring serisinin 4.'sü.

İleri 50 dk

Son güncelleme: 24 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/spring-batch

---

"Task Execution & Scheduling", "bu iş ne zaman başlamalı, ve onu çağıranı bloklamadan nasıl çalıştırırım?" sorusunu cevapladı. Bu ders tamamen farklı bir soruyu cevaplıyor: bir iş gerçekten büyük olduğunda -- bir avuç değil, yüz binlerce kayıt -- onu nasıl yapılandırır, çalıştırır, izler, ve kurtarırsın? Spring Batch, Spring'in bu soruya cevabı, ve ne `@Async`'in ne de `@Scheduled`'ın üzerine kurulu değil; genelde ikisinden BİRİ tarafından tetiklenen ayrı bir konu.

Gerçek Sorun: 500.000 Kayıtlık Gecelik Bir Batch

Gerçek bir gereksinim hayal et: uygulama her gece saat 2'de 500.000 müşteri kaydını işlemek zorunda -- onları veritabanından okumak, her birini doğrulayıp dönüştürmek, sonuçları başka bir yere yazmak, yarı yolda bir hatadan hiçbir şeyi bozmadan kurtulmak, ve 500.000 kaydın hepsini baştan işlemek yerine kaldığı yerden yeniden başlayabilmek.

Saf yaklaşım ilk bakışta açık görünür:

```
@Scheduled(cron = "0 0 2 * * *")
public void processCustomers() {
    // read 500,000 records
    // process them
    // write them
}
```

Bu derlenir, ve birkaç yüz satır için işe bile yarayabilir. Gerçek ölçekte parçalanır: 500.000 kaydı bir kerede belleğe yüklemek israf (ya da imkansız)dır; bunun etrafına tek bir dev transaction koymak ya kaynakları çok uzun süre kilitler, ya da hiç transactional değilse,

herhangi bir hatada veritabanını yarım yazılmış bir durumda bırakır; ve süreç 300.001. kayıtta çökerse, gerçekte ne kadar ilerlediğine dair hiçbir yerde hiçbir kayıt yoktur -- yeniden başlatmak 1. kayıttan başlamak demektir.

`@Scheduled` ve Spring Batch AYNI sorunu çözmüyor -- birinden diğerinin garantilerini bekleyerek ona başvurma.

```
@Scheduled
    → "Bu iş NE ZAMAN başlamalı?"

Spring Batch
    → "Büyük bir batch işlemi NASIL yapılandırılmalı,
       çalıştırılmalı, izlenmeli ve yeniden başlatılmalı?"
```

İkisi rakip değil -- genelde birlikte kullanılırlar, her biri yalnızca gerçekten iyi olduğu kısmı yaparak:

```
@Scheduled
     |
     v
Spring Batch Job'ı başlat
     |
     v
Spring Batch gerçek batch işlemeyi yönetir
```

Bir Zihinsel Model: Job, Step ve Chunk-Odaklı İşleme

Herhangi bir Spring Batch sınıfından ya da annotation'ından önce, bu dersin geri kalanının dolduracağı şekil bu:

```
Job
 |
 +-- Step 1
 |     |
 |     +-- read
 |     +-- process
 |     +-- write
 |
 +-- Step 2
       |
       +-- read
       +-- process
       +-- write
```

Bir **Job**, bütün batch sürecidir -- "bu geceki siparişleri içe aktar," baştan sona. Bir **Step**, o sürecin bir aşamasıdır -- gerçekten ayrı birkaç aşaması olan bir Job'ın (içe aktar, sonra mutabakat yap, sonra raporla) sırayla çalışan birkaç Step'i olur. Bir Step genelde **chunk-odaklıdır**: bir grup öğeyi okur, her birini isteğe bağlı olarak dönüştürür, ve yazar -- ve chunk-odaklı bir step'in içinde, gerçek işi üç rol yapar: bir `ItemReader` öğeleri birer birer okur, bir

`ItemProcessor` her birini isteğe bağlı olarak dönüştürür ya da doğrular, ve bir `ItemWriter` bir bütün grubu tek seferde yazar. Spring Batch'in kendisi bu üçünü süren döngüyü yönetir -- sen her birinin ne yaptığını yazarsın, onları çağıran döngüyü değil.

Çalışan Örnek: Bir CSV Dosyasından Siparişleri İçe Aktarmak

Bu dersin her parçası, her bölüm için yeni bir uydurma örnek yerine, boyunca yeniden kullanılan TEK bir somut senaryoya doğru inşa ediyor: **müşteri siparişlerini bir CSV dosyasından bir veritabanına içe aktarmak.**

```
orderId,customerId,amount
1001,C001,125.50
1002,C002,89.90
1003,C003,250.00
```

Job, `orders.csv`'den her satırı okur, onu bir `Order`'a dönüştürür, doğrular ya da dönüştürür, ve geçerli siparişleri veritabanına yazar -- yukarıdaki zihinsel modeldeki reader/processor /writer rollerinin tam olarak gerçek bir pipeline'a uygulanması:

```
orders.csv
    |
    v
ItemReader
    |
    v
Order
    |
    v
ItemProcessor
    |
    v
İşlenmiş Order
    |
    v
ItemWriter
    |
    v
Veritabanı
```

Chunk İşleme: Gruplar Halinde Okumak, Yazmak ve Commit Etmek

`.chunk(100)`'ü "100 öğeyi işle" diye okuyup geçmek kolaydır -- ama altında gerçekte ne olduğu, bu dersteki tek en önemli mekaniktir.

```
1. öğeyi oku
2. öğeyi oku
3. öğeyi oku
...
100. öğeyi oku
        |
        v
Öğeleri işle
        |
        v
1-100 öğelerini yaz
        |
        v
TRANSACTION'I COMMIT ET
```

Sonra tam olarak aynı döngü, chunk chunk tekrarlanır:

```
101-200
        |
        v
yaz
        |
        v
COMMIT

201-300
        |
        v
yaz
        |
        v
COMMIT
```

Chunk'lama aynı anda birkaç somut nedenden önemlidir: bellek kullanımı sınırlı kalır (500.000 değil, bellekte 100 `Order` nesnesi); her chunk kendi transaction sınırıdır, bu yüzden bir hata, bütün job'a yayılan yarım yazılmış bir karmaşa bırakmaz; ve restart'ı anlamlı kılan tam olarak budur -- chunk sınırları olmadan, "bütün job"dan daha küçük, durumu bilinen bir birim olmazdı.

```
1-100 öğeler     → commit edildi
101-200 öğeler   → commit edildi
201-300 öğeler   → HATA
```

Kavramsal olarak, ilk 200 öğe güvenle commit edilmiştir, ve yalnızca üçüncü chunk başarısız olmuştur -- Spring Batch'in çalışma zamanı metadata'sı (birazdan işlenecek), framework'ün işlemenin 1. chunk'a değil 3. chunk'a ulaştığını bilmesini sağlayan şeydir. Bir restart'ta tam olarak hangi öğelerin yeniden işleneceği, job'ın yapılandırmasına ve kalıcı hale getirilen duruma bağlıdır -- bu ders bu ayrıntıya "Yeniden Başlatılabilirlik ve JobRepository"de geri dönüyor, burada soyut olarak vaat etmek yerine.

Minimal, Tam Bir Job Yapılandırması

Zihinsel model yerine oturduğuna göre, işte sipariş-içe-aktarma örneği için gerçekten tam (ama minimal) bir Job ve Step.

```
import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import org.springframework.batch.core.job.builder.JobBuilder;
import org.springframework.batch.core.repository.JobRepository;
import org.springframework.batch.core.step.builder.StepBuilder;
import org.springframework.batch.item.ItemProcessor;
import org.springframework.batch.item.ItemReader;
import org.springframework.batch.item.ItemWriter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.transaction.PlatformTransactionManager;

record Order(String orderId, String customerId, java.math.BigDecimal amount) {
}

// The two building blocks every Spring Batch job needs: a Job (the whole
// process) built from one or more Steps (a phase of that process). Both
// builders require a JobRepository -- Spring Batch's own store for
// execution state, covered later in this lesson -- because every Job and
// Step it creates is TRACKED there from the moment it runs, not just
// executed and forgotten.
@Configuration
class OrderImportJobConfig {

    @Bean
    public Job orderImportJob(JobRepository jobRepository, Step importStep) {
        return new JobBuilder("orderImportJob", jobRepository)
                .start(importStep)
                .build();
    }

    // chunk(100, transactionManager) is what makes this step CHUNK-ORIENTED:
    // reader/processor/writer are wired together here, and Spring Batch
    // itself drives the read-process-write loop between them in groups of
    // 100 -- see "Chunk Processing" for exactly what that means.
    @Bean
    public Step importStep(JobRepository jobRepository,
                           PlatformTransactionManager transactionManager,
                           ItemReader<Order> orderItemReader,
                           ItemProcessor<Order, Order> orderItemProcessor,
                           ItemWriter<Order> orderItemWriter) {

        return new StepBuilder("importStep", jobRepository)
                .<Order, Order>chunk(100, transactionManager)
                .reader(orderItemReader)
                .processor(orderItemProcessor)
                .writer(orderItemWriter)
                .build();
    }
}
```

Bunu, Spring Batch'in kendisinin çalıştırdığı şekilde yukarıdan aşağıya oku:

```
Job
 ↓
start(importStep)
 ↓
Step
 ↓
chunk(100)
 ↓
reader → processor → writer
```

`orderImportJob(...)`, Job'dır -- burada, tek bir Step (`importStep`) onun bütün sürecidir; bir `JobBuilder`'ın bir `JobRepository`'ye ihtiyacı vardır çünkü, birazdan işleneceği gibi, Spring Batch'in oluşturduğu her Job orada izlenir. `importStep(...)`, gerçek işin tarif edildiği yerdir: `.chunk(100, transactionManager)`, 100 chunk boyutuyla chunk-odaklı işlemeyi bildirir, ve `.reader(...)` / `.processor(...)` / `.writer(...)`, zihinsel modeldeki üç rolü takar -- her biri kendi bean'i olarak sağlanır, sırada ayrıntılandırılıyor.

Sipariş İçe Aktarımı İçin ItemReader, ItemProcessor ve ItemWriter

Yukarıda referans verilen üç rolün her birinin, BU örnek için somut bir implementasyona ihtiyacı var -- CSV satırlarını okumak, geçersiz miktarları filtrelemek, ve veritabanına yazmak.

```
import org.springframework.batch.item.ItemProcessor;
import org.springframework.batch.item.database.JdbcBatchItemWriter;
import org.springframework.batch.item.database.builder.JdbcBatchItemWriterBuilder;
import org.springframework.batch.item.file.FlatFileItemReader;
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.ClassPathResource;

import javax.sql.DataSource;
import java.math.BigDecimal;

record Order(String orderId, String customerId, BigDecimal amount) {
}

// The three pieces a chunk-oriented step actually drives: read one Order
// at a time from orders.csv, optionally transform/validate it, then write
// it. Spring Batch calls these in a loop -- reader.read() repeatedly until
// the chunk size is reached, processor.process() on each item, then a
// SINGLE writer.write(...) call for the whole chunk -- see "Chunk
// Processing" for why that grouping matters.
@Configuration
class OrderImportComponents {

    @Bean
    public FlatFileItemReader<Order> orderItemReader() {
        return new FlatFileItemReaderBuilder<Order>()
```

```
                .name("orderItemReader")
                .resource(new ClassPathResource("orders.csv"))
                .linesToSkip(1) // skip the CSV header row (orderId,customerId,amount)
                .delimited()
                .names("orderId", "customerId", "amount")
                .fieldSetMapper(fieldSet -> new Order(
                        fieldSet.readString("orderId"),
                        fieldSet.readString("customerId"),
                        fieldSet.readBigDecimal("amount")))
                .build();
    }

    // Returning null here is a deliberate Spring Batch convention: it
    // means "filter this item out." A filtered item is silently dropped
    // and never reaches the writer -- this is NOT an error and does not
    // fail the step, unlike the validation failures covered later in
    // "Fault Tolerance: Skip and Retry."
    @Bean
    public ItemProcessor<Order, Order> orderItemProcessor() {
        return order -> {
            if (order.amount().signum() < 0) {
                return null; // filtered out: a negative amount is not written, not an error
            }
            return order;
        };
    }

    @Bean
    public JdbcBatchItemWriter<Order> orderItemWriter(DataSource dataSource) {
        return new JdbcBatchItemWriterBuilder<Order>()
                .dataSource(dataSource)
                .sql("INSERT INTO orders (order_id, customer_id, amount) VALUES (:orderId, :
customerId, :amount)")
                .beanMapped()
                .build();
    }
}
```

`orderItemReader()`, `orders.csv`'yi okuyan, başlık satırını atlayan, her satırı ayırıcıya göre bölen, ve üç sütunu bir `Order`'a eşleyen bir `FlatFileItemReader<Order>` inşa eder. `orderItemProcessor()`, önemli, kaçırması kolay bir Spring Batch davranışını gösteriyor: **bir `ItemProcessor`'dan `null` döndürmek öğeyi FİLTRELER** -- sessizce düşürülür ve asla writer'a ulaşmaz, ki bu tam olarak negatif miktarlı bir siparişin burada hiçbir şeyi başarısız kılmadan hariç tutulma şeklidir. `orderItemWriter()`, her chunk'ın `Order`'larını, satır başına bir insert yerine chunk başına tek, gruplanmış bir SQL insert ile veritabanına yazar.

> Filtreleme (`null` döndürmek) ve başarısız olma (bir exception fırlatmak), farklı sonuçları olan farklı çıktılardır -- filtreleme sessizce tek bir öğeyi hariç tutar ve step normal şekilde devam eder; fırlatmak gerçek bir sorunu işaret eder, ve varsayılan olarak bütün step'i başarısız kılar. Bu dersin ilerisindeki "Fault Tolerance: Skip ve Retry", belirli exception'ları bunun yerine kurtarılabilir yapmayı işliyor.

JobParameters, JobInstance ve JobExecution

AYNI mantıksal job'ı tekrar çalıştırmak, otomatik olarak onu ilk kez çalıştırmakla aynı şey değildir -- Spring Batch'in bunları birbirinden ayırmanın bir yoluna ihtiyacı var, ve bu üç kavram tam olarak bunun için var.

```
Job: orderImportJob
JobParameters:
    file=orders.csv
    businessDate=2026-08-24
```

Belirli bir `JobParameters` kümesiyle birleşen bir `Job`, bir `JobInstance`'ı tanımlar:

```
Job
+
JobParameters
        |
        v
JobInstance
```

Bu önemlidir çünkü bir batch job genelde belirli bir girdiyle ilgilidir, yalnızca "mantığı tekrar çalıştır" değil. İki farklı `businessDate` değeri, gerçekten iki farklı iş parçasını temsil eder:

```
businessDate=2026-08-24
        → bir mantıksal job instance'ı

businessDate=2026-08-25
        → başka bir mantıksal job instance'ı
```

Ama tek bir `JobInstance` bile birden fazla kez DENENEBİLİR -- başarısız olur ve yeniden başlatılırsa, bu hâlâ aynı mantıksal instance'dır, yalnızca ona yeni bir deneme. O deneme bir `JobExecution`'dır:

```
Job
 |
 +-- JobInstance (businessDate=2026-08-24)
       |
       +-- JobExecution #1 → FAILED
       |
       +-- JobExecution #2 → COMPLETED
```

Bir `StepExecution`, aynı fikrin bir seviye altıdır -- belirli bir `JobExecution` içinde, tek bir Step'in kendi denemesi. Spring Batch bunların hepsini (sırada göreceğin gibi) tam olarak, tam olarak aynı `JobInstance` için başarısız bir `JobExecution`'ın başarılı bir tanesinden ayırt edilebilmesi için saklar, her çalışma aynı görünmesin diye.

Yeniden Başlatılabilirlik ve JobRepository

Yukarıdaki tüm defter tutmanın var olma nedeni, bir hata senaryosuyla somutlaşır.

```
500.000 kayıt

1-100.000       ✓
100.001-200.000 ✓
200.001-300.000 ✓
300.001-400.000 ✗ uygulama çöküyor
```

Ciddi bir batch framework'ünün bunu "bütün işlem başarısız oldu, baştan başla" diye ele almak zorunda kalmaması gerekir -- başarısız denemeyi yeniden başlatmayı, çoktan başarılı olmuş ilk 300.000 kaydı körü körüne yeniden işlemek yerine, kaldığı yere yakın bir yerden devam ettirmeyi desteklemeye yetecek çalışma zamanı durumunu kalıcı hale getirebilmelidir. `JobRepository`, tam olarak bunun için var:

```
Batch job'ın
      |
      v
JobRepository
      |
      +-- job execution durumu
      +-- step execution durumu
      +-- çalışma zamanı metadata'sı
      +-- restart'la ilgili durum
```

Spring Batch'i basitçe şunu yazmaktan ayıran şey budur:

```
while (...) {
    read();
    process();
    write();
}
```

Böyle düz bir döngünün kendi ilerlemesine dair hiçbir hafızası yoktur -- 300.001. kayıtta ölürse, hiçbir yerde bu gerçeği kaydeden hiçbir şey yoktur. `JobRepository`, `JobExecution` 'ların ve `StepExecution` 'ların gerçekten kalıcı hale getirildiği yerdir, bir `ExecutionContext` ile birlikte -- bir Step'in ona devam etmekle ilgili ayrıntıları hatırlamak için kullanabileceği küçük bir durum torbası. Bu ders `ExecutionContext` 'i yalnızca restart'ın neden mümkün olduğunu açıklamaya yetecek kadar tanıtıyor; belirli bir restart'ta hangi öğelerin atlanacağı hangi öğelerin yeniden işleneceği, evrensel bir garantiye değil, step'in kendi yapılandırmasına bağlıdır.

Bir Step Başarısız Olduğunda Ne Olur

Somut olarak, chunk-odaklı bir step'in başarısızlık yolu şöyle görünür:

```
Reader
   ↓
Processor
   ↓
Writer
   ↓
Veritabanı hatası
```

Bir chunk içinde tam olarak nerede bir şey ters giderse gitsin doğru olan birkaç şey var: chunk transaction sınırıdır, bu yüzden içindeki bir hata, yalnızca onu tetikleyen tek öğeyi değil, o chunk'ın bütün yazmalarını geri alır (rollback); `JobExecution` / `StepExecution` metadata'sı bu denemenin başarısız olduğunu ve yaklaşık ne kadar ilerlediğini kaydeder; ve bir restart'ın o noktadan devam edip etmediği -- ve ne kadar kesin devam ettiği -- her hatanın tam olarak başarısız olan öğeden otomatik, evrensel bir garantiyle devam ettiği bir şey değil, step'in yapılandırmasına (chunk boyutu, yeniden başlatılabilir olup olmadığı, ne durum sakladığı) bağlıdır.

Fault Tolerance: Skip ve Retry

Ancak temel model -- Job, Step, reader/processor/writer, chunk'lar, repository -- anlam kazandıktan sonra, tek tek kötü öğeler için fault tolerance tanıtmak mantıklı olur.

10.000 kaydın, #532 numaralı kaydının bozuk veri içerdiğini hayal et. Fault tolerance olmadan, o tek kötü kayıt BÜTÜN step'i başarısız kılar. Yapılandırıldığında:

```
import org.springframework.batch.core.Step;
import org.springframework.batch.core.repository.JobRepository;
import org.springframework.batch.core.step.builder.StepBuilder;
import org.springframework.batch.item.ItemProcessor;
import org.springframework.batch.item.ItemReader;
import org.springframework.batch.item.ItemWriter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.dao.TransientDataAccessException;
import org.springframework.transaction.PlatformTransactionManager;

record Order(String orderId, String customerId, java.math.BigDecimal amount) {
}

class InvalidOrderException extends RuntimeException {
    InvalidOrderException(String message) {
        super(message);
    }
}

// The SAME importStep from "A Minimal, Complete Job Configuration", with
// fault tolerance added -- faultTolerant() switches on Spring Batch's
// skip/retry machinery for this step, which is OFF by default.
@Configuration
class FaultTolerantImportStepConfig {

    @Bean
    public Step importStep(JobRepository jobRepository,
                           PlatformTransactionManager transactionManager,
                           ItemReader<Order> orderItemReader,
                           ItemProcessor<Order, Order> orderItemProcessor,
                           ItemWriter<Order> orderItemWriter) {

        return new StepBuilder("importStep", jobRepository)
                .<Order, Order>chunk(100, transactionManager)
                .reader(orderItemReader)
                .processor(orderItemProcessor)
                .writer(orderItemWriter)
                .faultTolerant()
                // A genuinely malformed row (a missing customerId, say)
                // throws InvalidOrderException instead of being silently
                // filtered the way a negative amount is -- skip() lets the
                // STEP continue past it instead of failing the entire job,
                // up to 10 such rows.
                .skip(InvalidOrderException.class)
                .skipLimit(10)
                // A database hiccup is often TEMPORARY -- retry() tries the
                // same write again (up to 3 times) before giving up,
                // instead of treating a transient failure as permanent.
                .retry(TransientDataAccessException.class)
                .retryLimit(3)
                .build();
    }
}
```

`.faultTolerant()`, step için bu davranışı açar -- varsayılan olarak kapalıdır. `.skip(InvalidOrderException.class).skipLimit(10)` şu demektir: bu belirli türden bir hata gerçekleşirse, bütün job'ı hemen başarısız kılma -- o tek öğeyi atla ve devam et, ama toplamda yalnızca 10 atlamaya kadar; o türden 11. başarısızlık yine de step'i başarısız kılar. `.retry(TransientDataAccessException.class).retryLimit(3)` farklı bir şey demektir: bu türden bir hata GEÇİCİ olabilir (kısa bir veritabanı sorunu), bu yüzden vazgeçmeden önce AYNI işlemi 3 kez tekrar dene -- skip gerçekten kötü bir öğeyi tolere etmekle, retry ise aksi hâlde iyi olan bir işlemin muhtemelen geçici bir başarısızlığını tolere etmekle ilgilidir.

@Scheduled'ı Spring Batch ile Birleştirmek

Bu ders doğrudan "Task Execution & Scheduling"in ardından geliyor, bu yüzden iki mekanizmayı açıkça bağlamakta fayda var.

```
import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobParameters;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

import java.time.LocalDate;

// @Scheduled decides WHEN this runs; everything after that line is Spring
// Batch deciding HOW the actual import is executed, tracked, and (if
// needed) restarted -- "Task Execution & Scheduling" and this lesson
// working together, each doing only the job it's actually responsible for.
@Component
class NightlyOrderImportScheduler {

    private final JobLauncher jobLauncher;
    private final Job orderImportJob;

    NightlyOrderImportScheduler(JobLauncher jobLauncher, Job orderImportJob) {
        this.jobLauncher = jobLauncher;
        this.orderImportJob = orderImportJob;
    }

    @Scheduled(cron = "0 0 2 * * *")
    public void launchNightlyImport() throws Exception {
        // JobParameters identify THIS run -- "file" says which CSV to
        // read, "businessDate" says which day's import this is. Together
        // with the Job itself, these parameters are what Spring Batch uses
        // to tell one logical run apart from another -- see "JobParameters,
        // JobInstance, and JobExecution".
        JobParameters jobParameters = new JobParametersBuilder()
                .addString("file", "orders.csv")
                .addLocalDate("businessDate", LocalDate.now())
                .toJobParameters();

        jobLauncher.run(orderImportJob, jobParameters);
    }
}
```

`@Scheduled(cron = "0 0 2 * * *")`, `launchNightlyImport()`'un NE ZAMAN çalıştığına karar verir -- önceki dersten yeni bir şey yok. Yeni olan, onun ne yaptığı: `JobParameters`'ı (öncekinden `file` / `businessDate` çifti) inşa eder, ve onları, Job'ın kendisiyle birlikte, bir `JobLauncher`'a -- gerçekten bir `JobInstance` / `JobExecution` çalıştırmayı başlatan nesneye -- teslim eder.

```
Saat 2:00
   |
   v
@Scheduled
   |
   v
JobLauncher
   |
   v
Spring Batch Job
   |
   v
Step
   |
   v
Reader → Processor → Writer
```

Aklında tutulması gereken ayrım: `@Scheduled` ne zaman başlayacağına karar verir; Spring Batch, başladıktan sonra batch sürecini nasıl çalıştıracağına ve yöneteceğine karar verir. Bu, bu dersin iki merkezi çıkarımından biridir.

@Async vs. @Scheduled vs. Spring Batch

Bu kategorideki üç mekanizma da artık işlendiğine göre, her birinin gerçekte ne için olduğunu kesin olarak belirtmekte fayda var:

```
@Async
    → Bu metodu arka planda çalıştır.

@Scheduled
    → Bu metodu bir zamanlamaya göre başlat.

Spring Batch
    → Step'ler, chunk'lar, transaction'lar, çalışma zamanı
      metadata'sı, hata yönetimi ve yeniden başlatılabilirlik
      dahil, yapılandırılmış bir batch job'ı çalıştır ve yönet.
```

Bunlar aynı sorun için rakip seçenekler değil -- bir arada var olurlar, ve genelde de öyle olur, tam olarak `ScheduledOrderImportLauncher`'ın gösterdiği gibi: `@Scheduled` bir başlatmayı tetikler, ki bu başlatmanın kendisi ne `@Async`'tir ne de tek başına bir batch job'dır, ama bir tanesini başlatır.

Hepsini Bir Araya Getirmek: Gecelik Sipariş İçe Aktarımı, Baştan Sona

```
Gecelik sipariş içe aktarımı

@Scheduled
     |
     v
JobLauncher
     |
     v
Order Import Job
     |
     v
Import Step
     |
     +--> ItemReader
     |
     +--> ItemProcessor
     |
     +--> ItemWriter
     |
     v
Chunk transaction'ı
     |
     v
JobRepository
```

Sade bir dille: zamanlayıcı saat 2'de job'ı tetikler; Spring Batch bu `JobInstance` için `JobExecution`'ı oluşturur ya da tanımlar; step başlar; reader CSV'den `Order`'ları okur; processor her birini doğrular ve dönüştürür (öncekinde işlendiği gibi bazılarını filtreleyerek); writer hayatta kalanları veritabanına yazar; bunların hepsi 100'lük chunk'lar halinde olur; her chunk'ın yazmaları tek bir transaction olarak birlikte commit edilir; çalışma zamanı metadata'sı boyunca `JobRepository`'ye kaydedilir; ve job yarı yolda başarısız olursa, o kalıcı hale getirilen çalışma zamanı durumu, bir restart'ın tamamen baştan başlamaktan kaçınmak için kullanabileceği şeydir.

Spring Batch Ne DEĞİLDİR

Önceki dersle birbirine karışmasını önlemek için birkaç açık negatif hedef: Spring Batch basitçe başka bir zamanlayıcı DEĞİLDİR -- NE ZAMAN çalışacağı konusunda söyleyecek hiçbir şeyi yoktur (bu `@Scheduled`'ın, manuel bir tetikleyicinin, ya da tamamen başka bir zamanlayıcının işidir). Basitçe bir thread pool DEĞİLDİR -- chunk'lama ve transaction'lar onun temel meselesidir, eş zamanlı çalıştırma değil. Veritabanı kayıtları etrafında basitçe bir `while` döngüsü DEĞİLDİR -- `JobRepository`'nin izlenen çalışma zamanı durumu, düz bir döngünün asla sahip olmadığı şeydir. Ve otomatik olarak bir paralel-işleme framework'ü DEĞİLDİR -- bu derste inşa edilenle tam olarak aynı, standart chunk-odaklı bir step sıralı çalışır; gerçek paralellik (partitioning, paralel step'ler), bu dersin yalnızca ADINI verdiği, öğretmediği, ayrı, daha ileri bir konudur.

Best Practices

- Spring Batch'e, gerçek gereksinim büyük bir veri kümesinin yapılandırılmış, izlenebilir, yeniden başlatılabilir işlenmesi olduğunda başvur -- genelde tek başına `@Scheduled`'ın zaten kapsadığı her zamanlanmış görev için değil.
- Bir chunk boyutunu bilinçli olarak seç: verimli olacak kadar büyük, bir rollback ya da restart'ın aşırı iş israf etmeyeceği kadar küçük.
- `ItemProcessor`'ın `null` döndürmesini meşru filtreleme için kullan, ve fırlatılan exception'ları `skip` / `retry`'nin tepki vermesi gereken gerçek başarısızlıklar için sakla.
- `skipLimit` / `retryLimit` değerlerini bilinçli olarak küçük ve gerçekten tolere edilebilir exception türlerine özgü tut -- çıplak bir `.skip(Exception.class)`, gerçek hataları yüksek bir skip sayısının arkasına gizler.
- `@Scheduled`'ın bir job'ın ne zaman başlatılacağına, Spring Batch'in ise nasıl çalışacağına karar vermesine izin ver -- ikisini tek bir metotta birbirine karıştırma.

Yaygın Hatalar

- `@Scheduled`'ın kendisinin Spring Batch'in izleme, chunk'lama ya da restart davranışından herhangi birini sağladığını varsaymak -- yalnızca bir şeyin ne zaman başlayacağına karar verir.
- Bir `Job`'ı (yeniden kullanılabilir tanım) bir `JobExecution` (onu çalıştırmaya bir deneme) ile karıştırmak -- aynı `Job` bean'i zamanla birçok `JobExecution` üretir.
- Bir `JobInstance`'ı (bir `Job` artı onu tanımlayan `JobParameters`'ı) bir `JobExecution` (o belirli instance'ı çalıştırmaya bir deneme) ile karıştırmak -- yeniden denenirse tek bir `JobInstance`'ın birden fazla `JobExecution`'ı olabilir.
- Chunk sınırlarını hesaba katmamak -- gerçekte bütün bir chunk birlikte commit edilirken (ya da geri alınırken), her öğenin bağımsız olarak commit edildiğini varsaymak.
- Her yapılandırmada her hatanın otomatik olarak tam olarak başarısız olan öğeden devam ettiğini varsaymak, bunun yerine step'in kendi restart yapılandırmasına bağlı olduğunu göz ardı etmek.
- Düz, sıralı, chunk-odaklı bir step iyi anlaşılmadan partitioning'e ya da paralel step'lere başvurmak.
- Farklı API'leri göstermek için birkaç ilgisiz uydurma job oluşturmak yerine, bu dersin baştan sona tek bir sipariş-içe-aktarma job'ına sadık kaldığı gibi tek, tutarlı bir örnekte derinlik inşa etmek.
- Uyguladığı çalışma zamanı modelini açıklamadan önce tam Job/Step yapılandırmasını yazmak -- kod, bu dersin başındaki zihinsel model olmadan çok az şey ifade eder.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

Bu dersteki her büyük soyutlama için, cevapladığı soru:

```
Job
→ İşin bütün birimi nedir?

Step
→ Job'ı anlamlı aşamalara nasıl bölerim?

ItemReader
→ Bir öğeyi nasıl birer birer okurum?

ItemProcessor
→ Bir öğeyi nerede dönüştürür/doğrularım?

ItemWriter
→ İşlenmiş öğeleri nerede kalıcı hale getirir/çıktı veririm?

chunk(...)
→ İşlemeyi transaction sınırlarına nasıl gruplarım?

JobRepository
→ Spring Batch çalışma zamanı durumunu nerede tutar?

JobLauncher
→ Bir job'ı nasıl başlatırım?

JobParameters
→ Belirli bir çalışmayı/girdiyi ne tanımlar?

JobExecution
→ Bir çalıştırma denemesi sırasında ne oldu?
```

- Spring Batch, büyük batch işlemlerini yapılandırır, çalıştırır, izler, ve yeniden başlatabilir -- `@Scheduled`'ın "ne zaman"ından gerçekten farklı bir konu.
- Bir `Job`, bir ya da daha fazla `Step`'ten inşa edilir; chunk-odaklı bir `Step`, bir `ItemReader` → `ItemProcessor` → `ItemWriter` pipeline'ını gruplar halinde ("chunk") sürer, her chunk tek bir transaction olarak commit edilir.
- Bir `Job` artı onun `JobParameters`'ı bir `JobInstance`'ı tanımlar; o instance'ı çalıştırmaya her deneme bir `JobExecution`'dır, `JobRepository`'de (her `StepExecution`'la birlikte) izlenir.
- Fault tolerance (`skip`/`retry`), `.faultTolerant()` ile opt-in'dir, ve gerçekten kötü öğeleri (skip) muhtemelen geçici başarısızlıklardan (retry) ayırır.
- `@Scheduled` ve Spring Batch genelde birlikte çalışır: zamanlayıcı bir job'ın ne zaman başlatılacağına karar verir, Spring Batch o job'ın gerçekte nasıl çalışacağına karar verir.

Cheat Sheet

```
// Job + chunk-odaklı Step
@Bean
Job orderImportJob(JobRepository repo, Step importStep) {
    return new JobBuilder("orderImportJob", repo).start(importStep).build();
}

@Bean
Step importStep(JobRepository repo, PlatformTransactionManager tx,
                ItemReader<Order> reader, ItemProcessor<Order, Order> processor,
                ItemWriter<Order> writer) {
    return new StepBuilder("importStep", repo)
            .<Order, Order>chunk(100, tx)
            .reader(reader).processor(processor).writer(writer)
            .build();
}

// ItemProcessor: filtrelemek için null döndür, başarısız/skip için fırlat
ItemProcessor<Order, Order> processor() {
    return order -> order.amount().signum() < 0 ? null : order;
}

// Fault tolerance
.faultTolerant()
.skip(InvalidOrderException.class).skipLimit(10)
.retry(TransientDataAccessException.class).retryLimit(3)

// JobParameters bir JobInstance'ı tanımlar
JobParameters params = new JobParametersBuilder()
        .addString("file", "orders.csv")
        .addLocalDate("businessDate", LocalDate.now())
        .toJobParameters();

// @Scheduled job'ı başlatır; Spring Batch onu çalıştırır
@Scheduled(cron = "0 0 2 * * *")
void launch() throws Exception {
    jobLauncher.run(orderImportJob, params);
}
```

Terimler Sözlüğü

- **Job**: bir ya da daha fazla Step'ten oluşan, bir batch sürecinin genel, yeniden kullanılabilir tanımı.
- **Step**: bir Job'ın bir aşaması, genelde chunk-odaklı (okuma, işleme, yazma).
- **Chunk**: birlikte işlenen ve tek bir transaction olarak commit edilen bir öğe grubu.
- **ItemReader / ItemProcessor / ItemWriter**: chunk-odaklı bir step'in sürdüğü üç rol -- bir öğeyi oku, isteğe bağlı olarak dönüştür/doğrula, bütün bir chunk'ı tek seferde yaz.
- **JobParameters**: bir Job ile birleştiğinde bir JobInstance'ı tanımlayan, tanımlayıcı girdi (bir dosya adı ya da bir business date gibi).
- **JobInstance**: kendi JobParameters'ıyla tanımlanan belirli bir Job çalışması -- aynı JobInstance birden fazla kez denenebilir.

- **JobExecution / StepExecution**: bir JobInstance'ı / içindeki bir Step'i çalıştırmaya bir deneme, bir durumla (örn. FAILED, COMPLETED) izlenir.
- **JobRepository**: Spring Batch'in çalışma zamanı durumunu -- job/step durumu, metadata, ve restart'la ilgili bilgiyi -- kalıcı hale getirdiği yer.
- **JobLauncher**: belirli bir JobParameters kümesiyle bir Job'ın gerçekten çalışmaya başlamasını sağlayan bileşen.
- **Skip / Retry**: fault-tolerance seçenekleri -- skip gerçekten kötü bir öğeyi (bir sınıra kadar) tolere eder; retry, aksi hâlde iyi olan bir işlemin muhtemelen geçici bir başarısızlığını tolere eder.