

SPRING

SPRING BATCH

FABIO TADASHI MIYASATO

2

LISTA DE FIGURAS

Figura 2.1 – Spring Initializr.....	8
Figura 2.2 – Resultado da execução do Job	11
Figura 2.3 – Divisão de responsabilidade de steps do tipo “chunck”	12
Figura 2.4 – Representação da sequência de execução de um step do tipo “chunck”.	13
Figura 2.5 – Resultado da execução do Job	19
Figura 2.6 – Conteúdo do arquivo “livros_vendidos.csv”	19
Figura 2.7 – Representação da tabela física no banco de dados.....	19

LISTA DE CÓDIGOS-FONTE

Código-fonte 2.1 – Configuração do acesso ao banco de dados H2	8
Código-fonte 2.2 – Estrutura do Job.....	9
Código-fonte 2.3 – Configuração do caminho do arquivo a ser excluído no application.yml.....	10
Código-fonte 2.4 – Estrutura do Step	10
Código-fonte 2.5 – Configuração do Job	11
Código-fonte 2.6 – Representação da entidade Livro.	13
Código-fonte 2.7 – Script para criação da tabela física que representa o objeto Livro	14
Código-fonte 2.8 – Estrutura da classe BatchChunkApplication	15
Código-fonte 2.9 – Estrutura do método “itemReader”	15
Código-fonte 2.10 – Configuração do nome do arquivo que será manipulado no “application.yml”.	15
Código-fonte 2.11 – Parametrização do Item Processor.....	16
Código-fonte 2.12 – Estrutura do método “itemWriter”	17
Código-fonte 2.13 – Criação do step do tipo Chunk.....	18
Código-fonte 2.14 – Estrutura de um Job.....	18

SUMÁRIO

2 SPRING BATCH	5
2.1 O que é um Batch?	5
2.2 E o Spring Batch?	5
2.3 Job	6
2.4 Step	7
2.5 Configuração Spring Batch.....	7
2.6 Tasklet.....	9
2.7 Chunk.....	12
2.7.1 Item Reader.....	14
2.7.2 Item Processor	16
2.7.3 Item Writer.....	16
2.7.4 Step.....	17
2.7.5 Job	18
2.8 Conclusão	20
REFERÊNCIAS.....	21
GLOSSÁRIO	22

2 SPRING BATCH

2.1 O que é um Batch?

“É um processo com quantidade delimitada de dados sem interação ou interrupção.”

Michael Minella

The Definitive Guide To Spring Batch

Michael Minella é o *co-lead* do Spring Batch e a maior referência no assunto. É importante entender essa definição para compreender o funcionamento e os casos de uso para o Spring Batch.

Um processo com quantidade delimitada de dados, ou seja, o processo batch, não é um processamento contínuo. Ao definir e iniciar um Job, os dados, seja arquivo ou banco de dados, devem ter uma quantidade definida de dados a serem processados, porém não é necessário informar ao processo a quantidade de dados.

O processo batch, também conhecido como processamento em lote, não tem interação após ser iniciado e não deve ser interrompido manualmente. O framework de processamento batch deve tratar eventuais interrupções de maneira previamente configurada.

2.2 E o Spring Batch?

Conforme a definição da documentação oficial, trata-se de um “framework de batch leve e abrangente para desenvolvimento de aplicações batch robustas para o dia a dia dos sistemas corporativos”. Seguindo a filosofia do framework, o Spring Batch permite processar lotes de dados de maneira simples, autoconfigurável e, o mais importante, pronta para produção.

O Spring Batch implementa a especificação JSR-352 da JEE, na qual são definidos os artefatos e ciclo de vida de uma aplicação para processamento em lotes.

Entre os principais casos de uso para o Spring Batch estão: processamento de lotes de arquivos periodicamente, processos batch concorrentes com altos níveis de paralelização e processamento de dados corporativos orientados por mensageria.

2.3 Job

O Spring Batch tem diversos componentes e conceitos específicos que precisam ser entendidos antes de iniciar o desenvolvimento. O Job talvez seja o mais importante, pois ele é a definição do que será executado pela aplicação. Trata-se do processo completo que vai tratar a quantidade delimitada de dados, citados anteriormente.

O Job representa o escopo do processamento. Em outras palavras, é como um plano (ou configuração) de tudo o que deve acontecer. Cada Job é composto por um ou mais Steps (que serão discutidos posteriormente).

Um Job pode receber diversos parâmetros de execução, por exemplo, no caso de uma livraria que tenha fechamento de vendas do mês, ele pode executar mensalmente e receber como parâmetro o mês a ser fechado.

Cada Job parametrizado por mês é chamado de Job Instance no Spring Batch. Então, se tivermos o Job “fechamento_vendas”, teremos os Job Instances “fechamento_venda_janeiro”, “fechamento_venda_fevereiro e assim por diante.

Por sua vez, cada Job Instance tem uma ou mais execuções denominadas Job Executions. Por exemplo, se o Job Instance de janeiro executar corretamente, só existirá um Job Execution para janeiro, mas, se houver qualquer falha, como uma indisponibilidade do banco de dados, por exemplo, pode haver mais Job Executions para janeiro, de acordo com as configurações de reinício do Job.

2.4 Step

O Step é a unidade de trabalho do Spring Batch, que pode ser tão simples ou complexo de acordo com as necessidades da aplicação. É nos Steps que ficam as configurações de leitura, escrita, processamento e regras de negócio do processamento batch.

Os Steps podem ter múltiplas execuções da mesma maneira que os Jobs. O Step Execution armazena as informações do status do processamento e do tempo de execução, assim como as referências ao Step e ao Job Execution em que foram criados.

Os Steps podem ser do tipo Tasklet ou Chunk. Os Steps Tasklet são mais simples. Um caso de uso clássico para esses Steps são para setup, como mover ou deletar um arquivo em um processo. Já os Steps Chunk são a implementação mais comum e onde é possível tirar todo o valor do Spring Batch.

2.5 Configuração Spring Batch

A melhor forma de criar e iniciar um projeto Spring é por meio do Spring Initializr (<https://start.spring.io>).

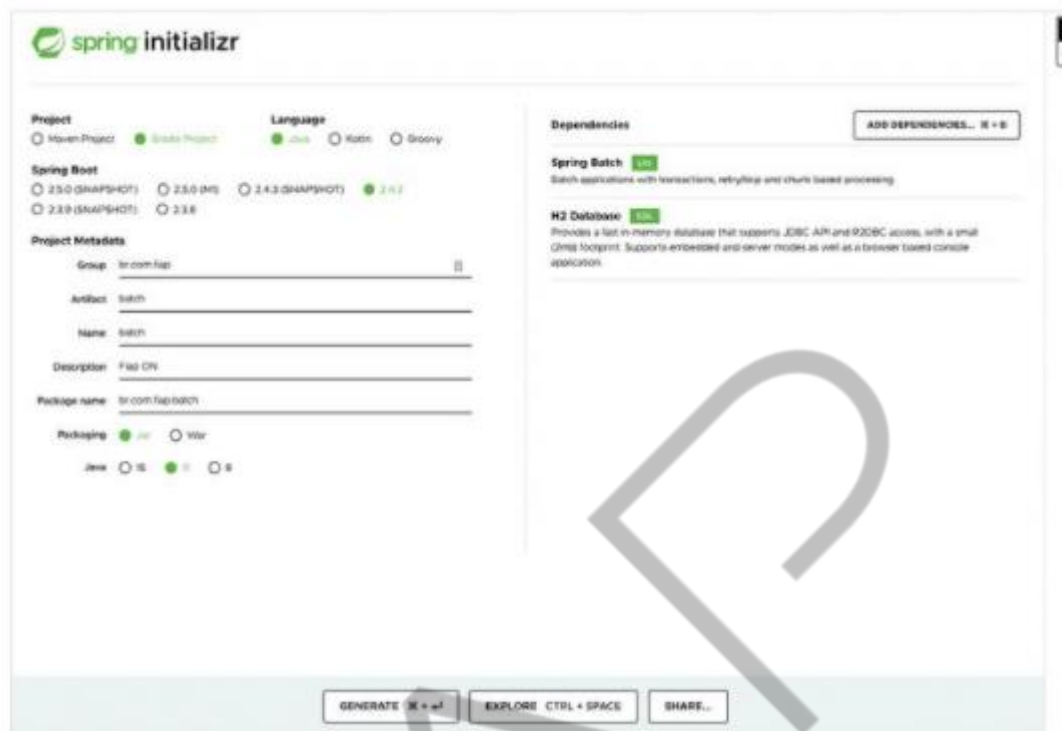


Figura 2.1 – Spring Initializr
Fonte: Spring Initializr (2021)

Vamos utilizar gradle project, java, a versão estável (2.4.2) e acrescentar as dependências Spring Batch e H2 Database.

Para configurar os dados de acesso ao banco, basta incluir essas propriedades no arquivo application.yml.

```
spring:
  datasource:
    url: jdbc:h2:~/fiapbatch1;DB_CLOSE_ON_EXIT=FALSE
    username: fiap
    password: fiap
  batch:
    initialize-schema: always
```

Código-fonte 2.1 – Configuração do acesso ao banco de dados H2
Fonte: Elaborado pelo autor (2021)

Essas propriedades configuram um banco de dados (H2) que será persistido em um arquivo no caminho especificado. O usuário e a senha são gerados no momento de criação da base e podem ser utilizados para acessar o banco.

O projeto criado pelo Spring Initializr gera a configuração baseada no subprojeto Spring Boot Starter Batch com uma estrutura familiar ao projeto criado no primeiro capítulo.

Na classe Application, além da anotação `@SpringBootApplication`, que já vem configurada, só é preciso colocar a anotação `@EnableBatchProcessing`.

2.6 Tasklet

Vamos criar um Job que possui um Step do tipo Tasklet para simular uma limpeza de arquivo. Antes de criar a Tasklet propriamente dita, é incluído o trecho de código para disponibilizar o objeto de tratamento de log.

```
Logger logger =
Factory.getLogger(BatchTaskletApplication.class);

@Bean

public Tasklet tasklet(@Value("${file.path}") String filepath){

    return (contribution, chunkContext) -> {

        File file = Paths.get(filepath).toFile();

        if(file.delete()){

            logger.info("File deleted");

        }else{

            logger.info("Couldn't delete file");

        }

        return RepeatStatus.FINISHED;

    };

}
```

Código-fonte 2.2 – Estrutura do Job
Fonte: Elaborado pelo autor (2021)

A construção do método Tasklet é feita utilizando uma implementação anônima (ou função) da interface Tasklet. Essa interface recebe dois parâmetros (StepContribution e ChunkContext) e retorna um RepeatStatus.

O método criado é bastante simples. Ele deleta um arquivo, caso o encontre, e imprime no log. O método foi anotado com a anotação @Bean, conceito apresentado no capítulo 1, para delegar o gerenciamento da instância dessa classe para o contexto do Spring. É necessário configurar o parâmetro file.path no application.yml (ou application.properties).

```
file:  
  
  path: /Users/fabiomiyasato/Documents/batch_file.txt
```

Código-fonte 2.3 – Configuração do caminho do arquivo a ser excluído no application.yml
Fonte: Elaborado pelo autor (2021)

Em seguida, cria-se o Step:

```
@Bean  
public Step step(StepBuilderFactory stepBuilderFactory,  
                 Tasklet tasklet){  
    return stepBuilderFactory.get("Delete file step")  
        .tasklet(tasklet)  
        .allowStartIfComplete(true)  
        .build();  
}
```

Código-fonte 2.4 – Estrutura do Step
Fonte: Elaborado pelo autor (2021)

Esse Step recebe dois parâmetros: StepBuilderFactory e Tasklet (a instância de Tasklet criada acima). Os Beans, assim como as classes de serviço, podem receber parâmetros que são gerenciados via injeção de dependência.

O `StepBuilderFactory` utiliza o conceito de `Builder` para gerar instâncias de classe `Step`. A primeira configuração é feita pelo método `get` e é o nome do `Step`. Em seguida, configuramos o `Tasklet`, utilizando a `Tasklet` recebida como parâmetro.

A configuração `allowStartIfComplete` permite que esse `Step` seja executado mais de uma vez com os mesmos parâmetros. Seguindo o conceito `Builder`, basta chamar o método `Build` e a instância da classe é gerada. Esse método também é anotado com `@Bean` e gerenciado pelo contexto do Spring.

O último `Bean` a ser configurado é o `Job`.

```
@Bean

public Job job(JobBuilderFactory jobBuilderFactory,

               Step step){

    return jobBuilderFactory.get("Delete file job")

        .start(step)

        .build();

}
```

Código-fonte 2.5 – Configuração do Job
Fonte: Elaborado pelo autor (2021)

Seguindo o mesmo padrão, esse método recebe o `JobBuilderFactory` e o `Step` configurado anteriormente. O `JobBuilderFactory` também utiliza o conceito de `Builder` e só precisamos configurar o nome do `Job` e definir que o `Step` recebido como parâmetro vai ser executado no início do `Job`.

Ao executar a classe, o arquivo é deletado (caso exista) ou será apresentado um log demonstrando que o arquivo não existe.

```
br.com.fiap.batch.BatchApplication : Started BatchApplication in 1.384 seconds (JVM running for 7.287)
o.s.b.a.b.JobLauncherApplicationRunner : Running default command line with: []
o.s.b.c.l.support.SimpleJobLauncher : Job: [SimpleJob: [name=Delete file job]] launched with the following parameters: [{}]
```

Figura 2.2 – Resultado da execução do Job
Fonte: Elaborado pelo autor (2021)

2.7 Chunk

Os Steps do tipo Chunk trazem as principais vantagens do Spring Batch. A principal é o processamento dividido em lotes menores, muito útil para manipulação de grandes volumes de registros e de dados. Esse tipo de Step divide a responsabilidade em outras classes relacionadas a seguir. Mais detalhes podem ser consultados em <https://docs.spring.io/spring-batch/docs/current/reference/html/step.html#configureStep>.

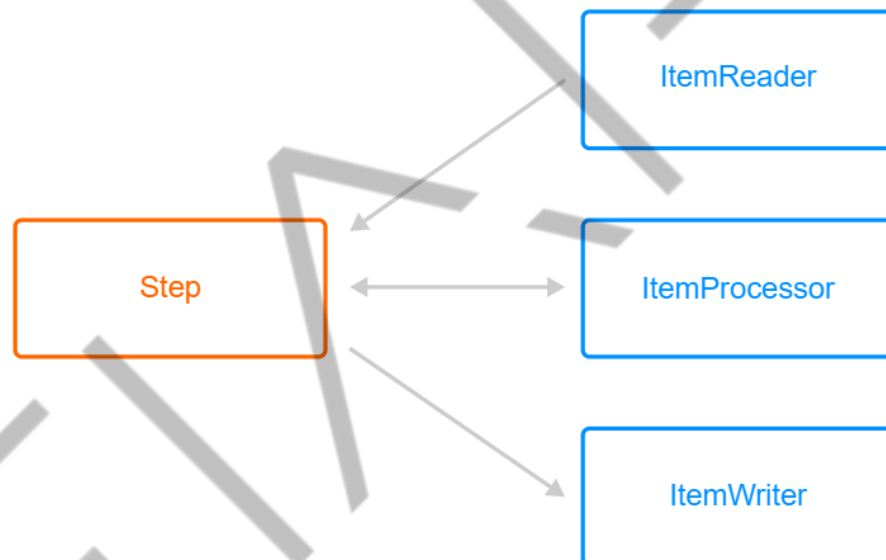


Figura 2.3 – Divisão de responsabilidade de steps do tipo “chunk”
Fonte: Spring Batch (2021)

Conforme os nomes, as interfaces ItemReader, ItemProcessor e ItemWriter são responsáveis respectivamente pela leitura, processamento e escrita dos dados.

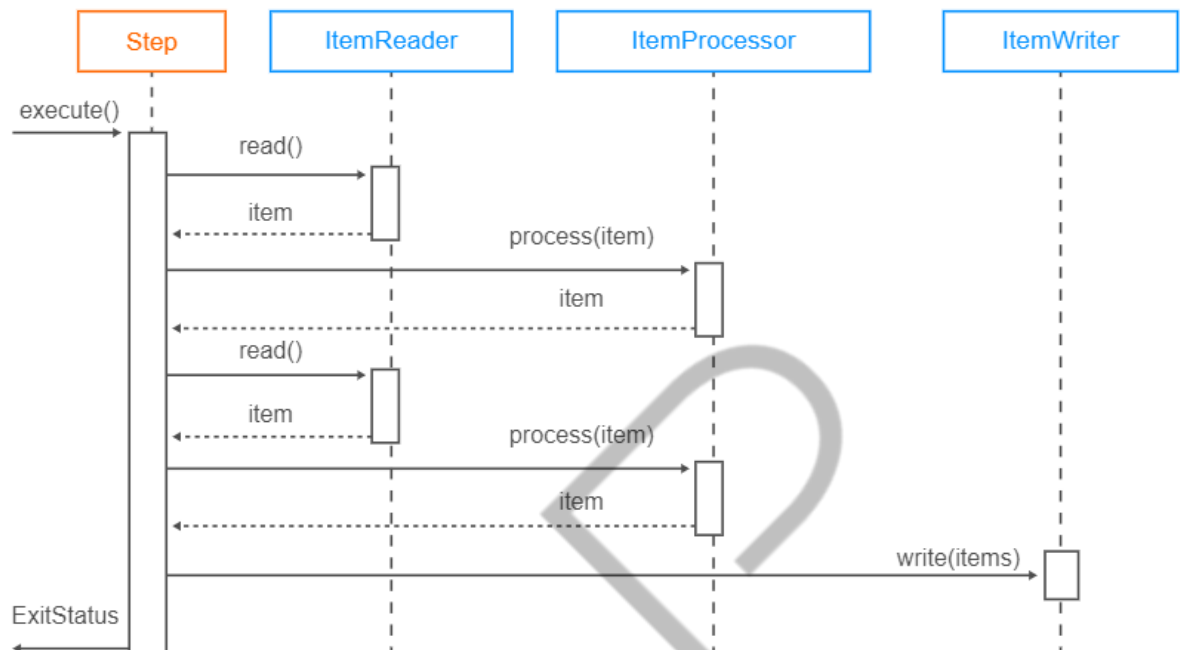


Figura 2.4 – Representação da sequência de execução de um step do tipo “chunk”
 Fonte: Spring Batch (2021)

Conforme a figura “Representação da sequência de execução de um step do tipo “chunk”, a interação entre as classes não é linear. O gerenciamento do processamento é de responsabilidade do Spring.

Para a criação desse processo, é necessário importar um arquivo (simulando uma integração com um ERP, por exemplo) e colocá-lo no banco de dados da aplicação. O primeiro passo é criar um pojo representando a entidade que será importada.

```

public class Livro {

    private String titulo;

    private int quantidade;

    // getters e setters

}
  
```

Código-fonte 2.6 – Representação da entidade Livro
 Fonte: Elaborado pelo autor (2021)

Dessa forma, a classe livro possui o título do livro e a quantidade vendida no período. O Spring Batch procura por scripts sql no classpath com o padrão de

nomenclatura “schema-PLATAFORMA_SQL.sql”, em que a palavra plataforma deve ser substituída pelo nome do banco de dados (schema-mysql.sql, por exemplo). O Spring Batch também pode carregar um arquivo genérico com o nome “schema-all.sql”. Basta criar esse arquivo no classpath (pasta resources) e usar o script a seguir.

```
drop table if exists TB_LIVRO;

create table TB_LIVRO (

    id long auto_increment not null primary key,

    titulo varchar(100) not null,

    quantidade int not null

)
```

Código-fonte 2.7 – Script para criação da tabela física que representa o objeto Livro
Fonte: Elaborado pelo autor (2021)

2.7.1 Item Reader

A primeira etapa a ser configurada é a leitura de dados. Item Reader é uma interface e existem diversas implementações fornecidas pelo Spring Batch: FlatFileItemReader, JsonItemReader, JdbcCursorItemReader, HibernateCursorItemReader, StaxEventItemReader e KafkaItemReader. Além disso, é possível criar uma implementação de leitura customizada por meio da interface CustomItemReader.

Para criar a aplicação Chunk, vamos gerar uma nova classe executável e incluir as anotações do Spring Boot e Spring Batch, além de configurar o método main e a propriedade de log.

```
@SpringBootApplication

@EnableBatchProcessing

public class BatchChunkApplication {

    Logger                                logger                                =
    LoggerFactory.getLogger(BatchChunkApplication.class);

    public static void main(String[] args) {
```

```
        SpringApplication.run(BatchChunkApplication.class, args);  
    }  
}
```

Código-fonte 2.8 – Estrutura da classe BatchChunkApplication
Fonte: Elaborado pelo autor (2021)

O primeiro Bean configurado é o Item Reader.

```
@Bean  
  
public FlatFileItemReader<Livro> itemReader(@Value("${file.chunk}")  
Resource resource){  
  
    return new FlatFileItemReaderBuilder<Livro>()  
        .name("Livro item reader")  
        .targetType(Livro.class)  
        .resource(resource)  
        .delimited().delimiter(";").names("titulo", "quantidade")  
        .build();  
}
```

Código-fonte 2.9 – Estrutura do método “itemReader”
Fonte: Elaborado pelo autor (2021)

A implementação FlatFileItemReader realiza a leitura do arquivo recebido com um parâmetro do tipo Resource. Logo após deve ser configurado o nome do arquivo na propriedade file.chunk no application.yml (ou application.properties):

```
file:  
  
    chunk: vendas_livros.txt
```

Código-fonte 2.10 – Configuração do nome do arquivo que será manipulado no “application.yml”
Fonte: Elaborado pelo autor (2021)

O FlatFileItemReader é uma classe parametrizada, no caso com a classe Livro (também é necessário definir o targetType). Além disso, basta configurar o nome do Item Reader, o resource (arquivo em questão) e o delimitador. O arquivo utilizado é

um arquivo separado por “;” e terá como nome de propriedades “título” e “quantidade”. Por fim, chamar o método Build, conforme os exemplos anteriores.

2.7.2 Item Processor

A etapa de Item Processor não é obrigatória, mas é nesse método que a regra de negócio pode ser aplicada. A interface ItemProcessor deve ser parametrizada com um objeto de entrada e um outro de saída; no exemplo aqui apresentado, a entrada e a saída são o mesmo objeto, Livro.

```
@Bean  
  
public ItemProcessor<Livro, Livro> itemProcessor() {  
    return (Livro) -> {  
        Livro.setTitulo(Livro.getTitulo().toUpperCase());  
        Livro.setQuantidade(Livro.getQuantidade());  
        return Livro;  
    };  
}
```

Código-fonte 2.11 – Parametrização do Item Processor
Fonte: Elaborado pelo autor (2021)

Uma implementação anônima (ou função) pode ser utilizada para evitar a criação de classes de maneira desnecessária. Esse método recebe um livro e deve retornar um livro. No escopo aberto é para onde vai a lógica de negócio, que, nessa aplicação, é apenas deixar o título do livro em caixa alta.

2.7.3 Item Writer

O Item Writer é responsável pela escrita dos dados e, conforme definido anteriormente, ele persiste os dados no banco. Assim como no Item Reader, existem

diversas implementações prontas, bem como a possibilidade de criar uma implementação customizada, como apresentado a seguir.

```
@Bean
public JdbcBatchItemWriter<Livro> itemWriter(DataSource dataSource) {
    return new JdbcBatchItemWriterBuilder<Livro>()
        .dataSource(dataSource)
        .sql("insert into TB_LIVRO (titulo, quantidade) values (:titulo,
dade)")
        .beanMapped()
        .build();
}
```

Código-fonte 2.12 – Estrutura do método “itemWriter”

Fonte: Elaborado pelo autor (2021)

A implementação `JdbcBatchItemWriter` deve ser parametrizada com a classe `Livro` e receber um `Datasource` como dependência/parâmetro. Após configurar o `Datasource`, basta criar a query de inserção no banco de dados.

2.7.4 Step

Com o `ItemReader`, `ItemProcessor` e `ItemWriter` prontos, podemos criar o `Step` `Chunk`.

```
@Bean
public Step step(StepBuilderFactory stepBuilderFactory,
                ItemReader<Livro> itemReader,
                ItemProcessor<Livro, Livro> itemProcessor,
                ItemWriter<Livro> itemWriter) {
    return stepBuilderFactory.get("Step chunk file -> jdbc")
        .<Livro, Livro>chunk(2)
        .reader(itemReader)
}
```

```
.processor(itemProcessor)

.writer(itemWriter)

.build();

}
```

Código-fonte 2.13 – Criação do step do tipo Chunk
Fonte: Elaborado pelo autor (2021)

Assim como no Step Tasklet, é utilizado o StepBuilderFactory. Configurar o ItemReader, ItemProcessor e ItemWriter como dependências. Após configurar o nome do Step, o método Chunk é chamado com a parametrização das classes de entrada e saída (Livro) e a quantidade de item que serão processados por lote (no exemplo acima, 2). Em seguida, configurar o reader, o processor e o writer com os parâmetros recebidos.

2.7.5 Job

```
@Bean
public Job job(JobBuilderFactory jobBuilderFactory,
               Step step){
    return jobBuilderFactory.get("Job chunk")
        .start(step)
        .build();
}
```

Código-fonte 2.14 – Estrutura de um Job
Fonte: Elaborado pelo autor (2021)

A configuração do Job é igual à configuração de um Job com Steps do tipo Tasklet. Ao executar a aplicação, o log a seguir será exibido.

```

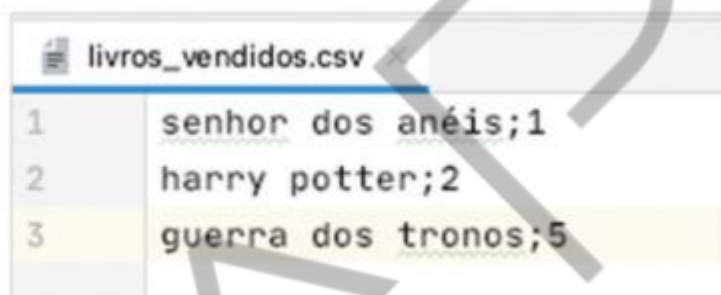
o.s.b.c.l.support.SimpleJobLauncher : No TaskExecutor has been set, defaulting to synchronous executor.
br.com.fiap.batch.BatchChunkApplication : Started BatchChunkApplication in 0.989 seconds (JVM running for 6.378)
o.s.b.s.b.JobLauncherApplicationRunner : Running default command line with: []
o.s.b.c.l.support.SimpleJobLauncher : Job: [SimpleJob: [name=Job chunk]] launched with the following parameters: [{]}
o.s.batch.core.job.SimpleStepHandler : Executing step: [Step chunk file -> jdbc]
o.s.batch.core.step.AbstractStep : Step: [Step chunk file -> jdbc] executed in 29ms
o.s.b.c.l.support.SimpleJobLauncher : Job: [SimpleJob: [name=Job chunk]] completed with the following parameters: [{]} and the

```

Figura 2.5 – Resultado da execução do Job

Fonte: Elaborado pelo autor (2021)

Dado o arquivo cujo conteúdo é exibido no exemplo a seguir.

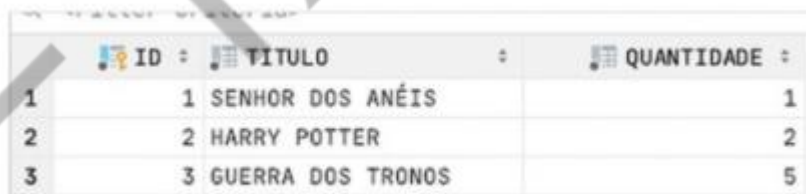


	livros_vendidos.csv
1	senhor dos anéis;1
2	harry potter;2
3	guerra dos tronos;5

Figura 2.6 – Conteúdo do arquivo “livros_vendidos.csv”

Fonte: Elaborado pelo autor (2021)

O banco de dados será preenchido com os registros.



	ID	TITULO	QUANTIDADE
1	1	SENHOR DOS ANÉIS	1
2	2	HARRY POTTER	2
3	3	GUERRA DOS TRONOS	5

Figura 2.7 – Representação da tabela física no banco de dados

Fonte: Elaborado pelo autor (2021)

2.8 Conclusão

Neste módulo foram criadas duas aplicações: um Job utilizando um Step Tasklet que deleta um arquivo e outro Job utilizando o Step Chunk para fazer a leitura de um arquivo csv e inserir o seu conteúdo no banco. Esses conceitos, apesar de básicos, permitem a criação de aplicações robustas, prontas para serem utilizadas em aplicações corporativas em produção.

O projeto criado está disponível em <https://github.com/fabiotadashi/spring-batch>. Para fechar o ciclo, no próximo capítulo serão apresentados alguns conceitos de segurança juntamente com o Spring Security.

REFERÊNCIAS

Spring Batch Documentation. Disponível em: <<https://spring.io/projects/spring-batch/>>. Acesso em: 15 jan. 2021.

EXEMPLO

GLOSSÁRIO

Datasource	Fonte de dados