Spring Batch au quotidien

1. Comprendre le traitement des données en lots
2. Implémenter et configurer des batchs
3. Lancer des batchs
4. Tester les différents élements d’un batch

1. Introduction

- Le traitement par lots, caractérisé par une exécution en arrière-plan:
  - orientée vers le volume,
  - non interactive
  - souvent longue,

- Il s'applique à un large éventail de tâches:
  - Traitement des données journalières dans une mutuelle
  - Traitement journalier des données d’avion pour les vols

- Le traitement par lots peut nécessiter beaucoup de données ou d’informatique, s’exécuter séquentiellement ou en parallèle,
  - Il peut être lancé à l’aide de divers modèles d’appel, notamment ad hoc, planifié et à la demande.

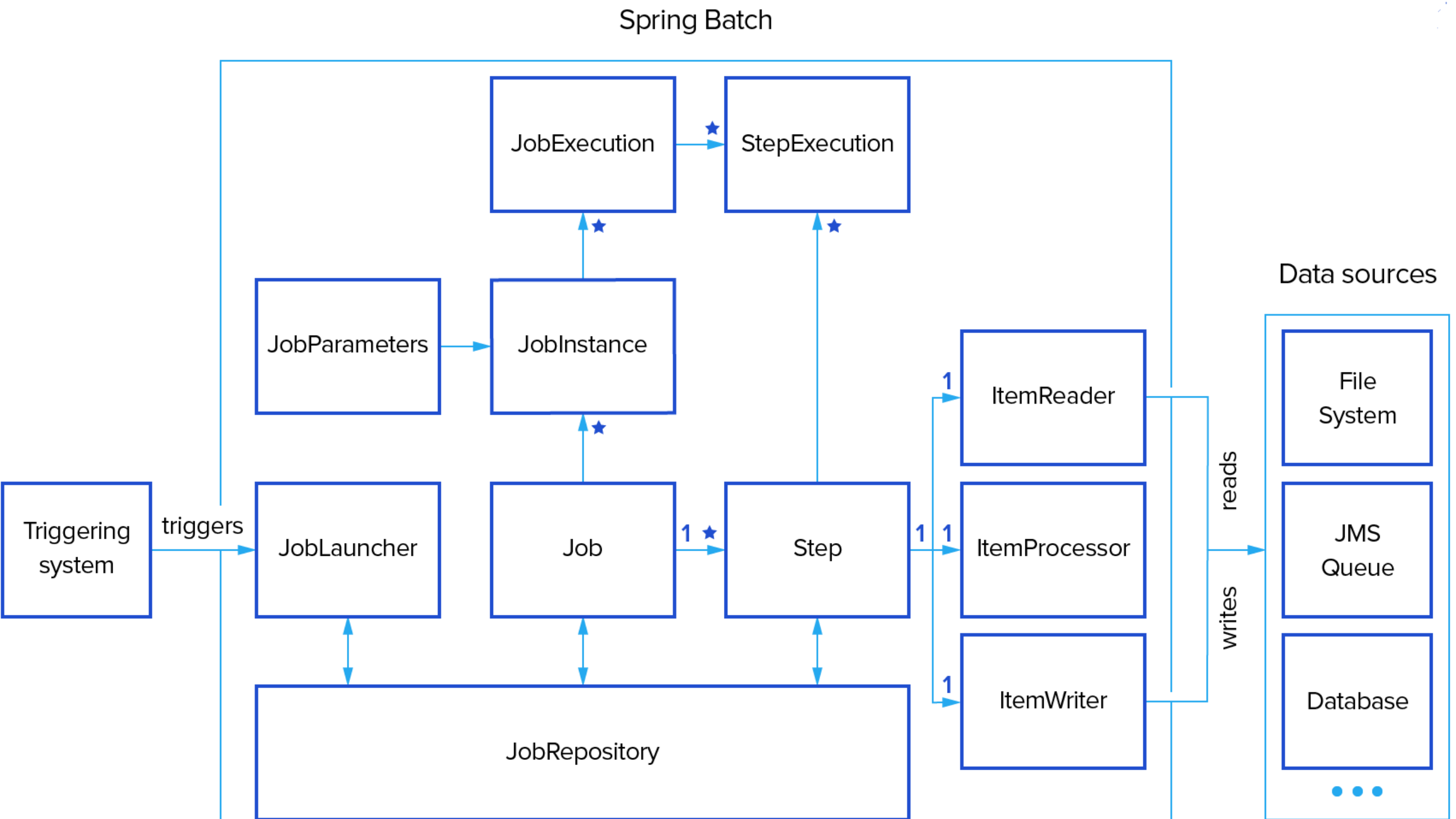
1. Introduction

- Spring Batch est un framework léger et complet conçu pour faciliter le développement d’applications par lots robustes.
- Il propose des fonctionnalités et des services techniques plus avancés prenant en charge des travaux par lots extrêmement volumineux et hautes performances grâce à ses techniques d'optimisation et de partitionnement.
- Spring Batch s'appuie sur l'approche de développement Spring Framework basée sur POJO, familière à tous les développeurs Spring:
  - version Spring Batch 3.0.7

1. Concepts clés et terminologie

- Un **batch** est encapsulé par un **job** comportant plusieurs **steps**.
  - Chaque step a généralement un seul **ItemReader**, **ItemProcessor** et **ItemWriter**.
  - Un Job est exécuté par un **JobLauncher** et les métadonnées relatives aux jobs configurés et exécutés sont stockées dans un **JobRepository**.

- Chaque job peut être associé à plusieurs **JobInstances**,
  - Chacune étant définie de manière unique par ses **JobParameters** particuliers utilisés pour démarrer un batch.
  - Chaque exécution d'un JobInstance est appelée **JobExecution**.
    Chaque JobExecution suit généralement ce qui s'est passé lors d'une exécution, tels que les statuts courant et de sortie, les heures de début et de fin, etc.


Spring Batch
JobExecution
StepExecution
JobParameters
JobInstance
Triggering system
triggers
JobLauncher
Job
Step
JobRepository
ItemReader
ItemProcessor
ItemWriter
reads
writes
Data sources
File System
JMS Queue
Database
1
1
1
1
1
1

1. Concepts clés et terminologie

- Un **step** est une phase spécifique et indépendante d'un batch, de sorte que chaque job est composé d'un ou plusieurs steps.
  - Semblable à un job, un step a une **StepExecution** individuel qui représente une seule tentative d'exécution d'un step.
  - Une StepExecution stocke les informations sur les statuts courants et de sortie, les heures de début et de fin, etc., ainsi que des références aux instances de **Step** et **JobExecution** correspondantes.

- Un **ExecutionContext** est un ensemble de paires clé-valeur contenant des informations qui sont étendues à StepExecution ou JobExecution.
  - Spring Batch conserve un ExecutionContext, ce qui est utile pour redémarrer un batch (par exemple, lorsqu'une erreur fatale s'est produite, etc.).
  - On peut placer objet à partager entre des steps dans le contexte et le framework s'occupera du reste.
  - Après le redémarrage, les valeurs du précédent ExecutionContext sont restaurées à partir de la base de données et appliquées.

1. Concepts clés et terminologie

- Un **JobRepository** est le mécanisme de Spring Batch qui autorise le principe de persistance.
  - Il fournit des opérations CRUD pour les instanciations de JobLauncher, Job et Step.
  - Une fois qu'un Job est lancé, une **JobExecution** est obtenue à partir du référentiel et, au cours de l'exécution, les instances StepExecution et JobExecution sont conservées dans le référentiel.

- Exemple:
  - Prendre en charge un fichier client au format XML, filtrer les clients selon divers attributs et génère les entrées filtrées dans un fichier texte.

2. Premiers pas avec Spring Batch Framework

- L'un des avantages de Spring Batch est la gestion des dépendances de projet, ce qui facilite la mise en route rapide.
- Les quelques dépendances existantes sont spécifiées dans le fichier pom.xml du projet,

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>fr.formation.spring</groupId>
    <artifactId>exemple-batch</artifactId>
    <version>1,0,0</version>
    <packaging>jar</packaging>

    <name>exemple-batch</name>
    <description>Demo project for Spring Batch</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.2.RELEASE</version>
        <relativePath/>
    </parent>
```

2. Premiers pas avec Spring Batch Framework

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-batch</artifactId>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.16.16</version>
    </dependency>

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.batch</groupId>
        <artifactId>spring-batch-test</artifactId>
    </dependency>
</dependencies>
```

```
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

2. Premiers pas avec Spring Batch Framework

- Le démarrage réel de l'application se produit dans une classe ayant l'aspect suivant:

```
@EnableBatchProcessing
@SpringBootApplication
public class BatchApplication {
    public static void main(String[] args) {
        prepareTestData(1000);
        SpringApplication.run(BatchApplication.class, args);
    }
}
```

- L'annotation **@EnableBatchProcessing** active les fonctionnalités Spring Batch et fournit une configuration de base pour la configuration du batch.
- L'annotation **@SpringBootApplication** provient du projet Spring Boot, qui fournit des applications autonomes, prêtes pour la production, basées sur Spring.
  - Il spécifie une classe de configuration qui déclare un ou plusieurs beans Spring et déclenche également la configuration automatique et l'analyse des composants de Spring.

2. Premiers pas avec Spring Batch Framework

- Cet exemple de projet ne contient qu'un seul job configuré par **CustomerReportJobConfig** avec un **JobBuilderFactory** et un **StepBuilderFactory** injectés.
- La configuration du job minimale peut être définie dans **CustomerReportJobConfig** comme suit:

- Il y a 2 approches principales pour construire un step.
  - Une approche basée sur des tasklets: ci contre
  - Un Tasklet prend en charge une interface simple comportant une seule méthode, execute (), qui est appelée à plusieurs reprises jusqu'à ce qu'elle retourne **RepeatStatus.FINISHED** ou lève une exception pour signaler un échec.
  - Chaque appel au Tasklet est encapsulé dans une transaction.

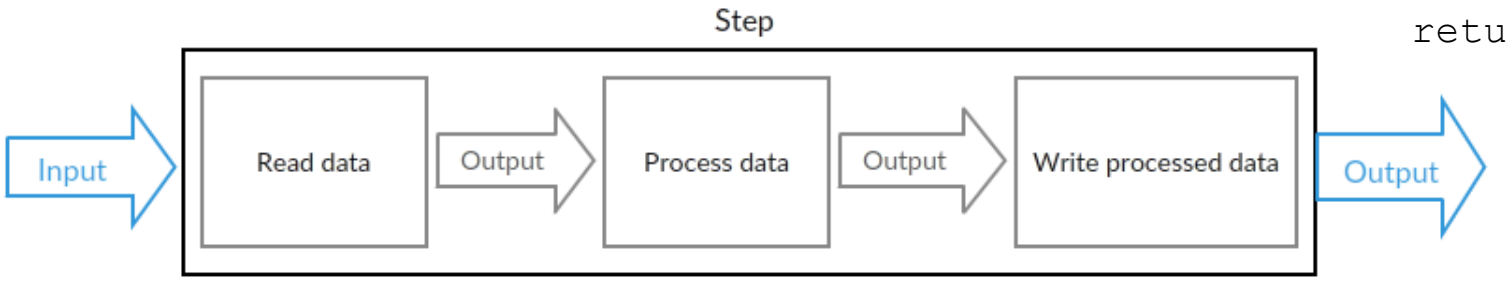
  - Une autre approche, le traitement orienté par blocs: prochain slide
  - Ell consiste à lire les données séquentiellement et à créer des «morceaux» qui seront écrits dans une limite de transaction.
  - Chaque élément individuel est lu à partir d'un **ItemReader**, remis à un **ItemProcessor** et agrégé.
  - Une fois que le nombre d'éléments lus est égal à l'intervalle de validation, le bloc entier est écrit via **ItemWriter**, puis la transaction est validée.

```
@Configuration
public class CustomerReportJobConfig {
    @Autowired
    private JobBuilderFactory jobBuilders
    @Autowired
    private StepBuilderFactory stepBuilders;

    @Bean
    public Job customerReportJob() {
        return jobBuilders.get("customerReportJob")
            .start(taskletStep())
            .next(chunkStep())
            .build();
    }
    @Bean
    public Step taskletStep() {
        return stepBuilders.get("taskletStep")
            .tasklet(tasklet())
            .build();
    }
    @Bean
    public Tasklet tasklet() {
        return (contribution, chunkContext) -> {
            return RepeatStatus.FINISHED;
        };
    }
}
```

2. Premiers pas avec Spring Batch Framework

- La méthode **chunk()** crée un **Step** qui traite les éléments en morceaux de la taille fournie, chaque morceau étant ensuite transmis au lecteur, au processeur et à l'écrivain spécifiés.




```
@Bean
public Job customerReportJob() {
    return jobBuilders.get("customerReportJob")
        .start(taskletStep())
        .next(chunkStep())
        .build();
}

@Bean
public Step chunkStep() {
    return stepBuilders.get("chunkStep")
        .<Customer, Customer>chunk(20)
        .reader(reader())
        .processor(processor())
        .writer(writer())
        .build();
}
```

3. Reader personnalisé

- Pour l'exemple Spring Batch, et lire une liste de clients à partir d'un fichier XML, nous devons fournir une implémentation de l'interface **org.springframework.batch.item.ItemReader**:

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

- **ItemReader** fournit les données au step et devrait être dynamique.
  - Il est généralement appelé plusieurs fois pour chaque lot, chaque appel de read() renvoyant la valeur suivante et renvoyant la valeur **null** lorsque toutes les données en entrée ont été épuisées.

3. Reader personnalisé

- Spring Batch fournit des implémentations prêtes à l'emploi de ItemReader, qui peuvent être utilisées à diverses fins, telles que la lecture de collections, de fichiers, l'intégration de JMS et de JDBC, ainsi que de multiples sources, etc.

- Dans l'exemple, la classe **CustomerItemReader** délègue les appels **read()** réels à une instance initialisée (lazzy) de la classe **IteratorItemReader**:

```
public class CustomerItemReader implements ItemReader<Customer> {

    private final String filename;
    private ItemReader<Customer> delegate;

    public CustomerItemReader(final String filename) {
        this.filename = filename;
    }

    @Override
    public Customer read() throws Exception {
        if (delegate == null) {
            delegate = new IteratorItemReader<>(customers());
        }
        return delegate.read();
    }

    private List<Customer> customers() throws FileNotFoundException {
        try (XMLDecoder decoder = new XMLDecoder(new
FileInputStream(filename))) {
            return (List<Customer>) decoder.readObject();
        }
    }
}
```

3. Reader personnalisé

- Un bean Spring pour cette implémentation est créé avec les annotations **@Component** et **@StepScope**, ce qui permet à Spring de savoir que cette classe est un composant Spring à portée Step et sera créé une fois par étape, comme suit:

```
@StepScope
@Bean
public ItemReader<Customer> reader() {
    return new CustomerItemReader(XML_FILE);
}
```

4. Processeur personnalisé

- **ItemProcessors** transforme les éléments en entrée et introduit la logique métier dans un scénario de traitement batch.
- Ils doivent fournir une implémentation de l'interface org.springframework.batch.item.ItemProcessor:

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

- La méthode **process()** accepte une instance de la classe **I** et peut renvoyer ou non une instance du classe **O**.
- Le renvoi de null indique que l'élément ne doit pas continuer à être traité
- Spring fournit des processeurs standards, tels que **CompositeItemProcessor** qui transmet l'élément à travers une séquence de **ItemProcessors** injectés et un **ValidatingItemProcessor** qui valide l'entrée.

- Dans l'exemple, les processeurs sont utilisés pour filtrer les clients selon les exigences suivantes:
  - Un client doit être né dans le mois en cours (par exemple, pour marquer des promotions d'anniversaire, etc.)
  - Un client doit avoir moins de 5 transactions terminées (par exemple, pour identifier de nouveaux clients).

- L'exigence du «mois en cours» est implémentée via un **ItemProcessor** personnalisé:

```
public class BirthdayFilterProcessor implements
ItemProcessor<Customer, Customer> {
    @Override
    public Customer process(final Customer item) throws
Exception {
        if (new GregorianCalendar().get(Calendar.MONTH)
            == item.getBirthday().get(Calendar.MONTH)) {
            return item;
        }
        return null;
    }
}
```

4. Processeur personnalisé

- L'exigence de «nombre limité de transactions» est implémentée en tant que **ValidatingItemProcessor**:

```
public class TransactionValidatingProcessor extends ValidatingItemProcessor<Customer> {
    public TransactionValidatingProcessor(final int limit) {
        super(
            item -> {
                if (item.getTransactions() >= limit) {
                    throw new ValidationException("Customer has less than " + limit + " transactions");
                }
            }
        );
        setFilter(true);
    }
}
```

- Cette paire de processeurs est ensuite encapsulée dans un **CompositeItemProcessor** qui implémente le modèle de délégation:

```
@StepScope
@Bean
public ItemProcessor<Customer, Customer> processor() {
    final CompositeItemProcessor<Customer, Customer> processor = new CompositeItemProcessor<>();
    processor.setDelegates(Arrays.asList(new BirthdayFilterProcessor(), new TransactionValidatingProcessor(5)));
    return processor;
}
```

5. Writer personnalisé

- Spring Batch fournit l'interface pour la sortie des données

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

- La méthode **write()** est chargée de s'assurer que tous les buffers internes sont vidés.
- Si une transaction est active, il sera également nécessaire de supprimer la sortie lors d'une restauration ultérieure.
- La ressource à laquelle le writer envoie des données devrait normalement être capable de gérer cela elle-même.
- Il existe des implémentations standard telles que **CompositeItemWriter, JdbcBatchItemWriter, JmsItemWriter, JpaItemWriter, SimpleMailMessageItemWriter** et autres.

```
public class CustomerItemWriter implements ItemWriter<Customer>,
Closeable {
    private final PrintWriter writer;

    public CustomerItemWriter() {
        OutputStream out;
        try {
            out = new FileOutputStream("output.txt");
        } catch (FileNotFoundException e) {
            out = System.out;
        }
        this.writer = new PrintWriter(out);
    }

    @Override
    public void write(final List<? extends Customer> items)
         throws Exception {
        for (Customer item : items) {
            writer.println(item.toString());
        }
    }

    @PreDestroy
    @Override
    public void close() throws IOException {
        writer.close();
    }
}
```

6. Séquencement de Job Spring Batch

- Par défaut, Spring Batch exécute tous les jobs qu'il peut trouver (c'est-à-dire qui sont configurés comme dans CustomerReportJobConfig) au démarrage.
- Pour modifier ce comportement, désactivez l'exécution des jobs au démarrage en ajoutant la propriété suivante à application.properties:

```
spring.batch.job.enabled = false
```

- La planification réelle est ensuite réalisée en ajoutant l'annotation **@EnableScheduling** à une classe de configuration et l'annotation **@Scheduled** à la méthode qui exécute le job lui-même.
- La planification peut être configurée avec des expressions de délai, de débit ou cron:

```
// run every 5000 msec (i.e., every 5 secs)
@Scheduled(fixedRate = 5000)
public void run() throws Exception {
    JobExecution execution = jobLauncher.run(
        customerReportJob(),
        new JobParametersBuilder().toJobParameters()
    );
}
```

- Il y a cependant un problème avec l'exemple.
- Au moment de l'exécution, le travail ne réussira que la 1ère fois. Lors du 2ème lancement (après 5 secondes), les messages suivants seront générés dans les logs (notez que dans les versions précédentes de Spring Batch, une exception **JobInstanceAlreadyCompleteException** aurait été levée):

```
INFO 36988 --- [pool-2-thread-1]
o.s.b.c.l.support.SimpleJobLauncher      : Job:
[SimpleJob: [name=customerReportJob]] launched with the
following parameters: [{}]
INFO 36988 --- [pool-2-thread-1]
o.s.batch.core.job.SimpleStepHandler     : Step already
complete or not restartable, so no action to execute:
StepExecution: id=1, version=3, name=taskletStep,
status=COMPLETED, exitStatus=COMPLETED, readCount=0,
filterCount=0, writeCount=0 readSkipCount=0,
writeSkipCount=0, processSkipCount=0, commitCount=1,
rollbackCount=0, exitDescription=
INFO 36988 --- [pool-2-thread-1]
o.s.batch.core.job.SimpleStepHandler     : Step already
complete or not restartable, so no action to execute:
StepExecution: id=2, version=53, name=chunkStep,
status=COMPLETED, exitStatus=COMPLETED, readCount=1000,
filterCount=982, writeCount=18 readSkipCount=0,
writeSkipCount=0, processSkipCount=0, commitCount=51,
rollbackCount=0, exitDescription=
```

6. Séquencement de Job Spring Batch

- Cela est dû au fait que seules des JobInstances uniques peuvent être créées et exécutées et que Spring Batch ne dispose d'aucun moyen de faire la distinction entre le 1èr et le 2ème JobInstance.

- Il existe 2 manières d'éviter ce problème lorsque l'on planifie un job batch.

- L'une consiste à être sûr d'introduire un ou plusieurs paramètres uniques (par exemple, l'heure de début réelle en nanosecondes) pour chaque travail:

```
@Scheduled(fixedRate = 5000)
public void run() throws Exception {
    jobLauncher.run(
        customerReportJob(),
        new JobParametersBuilder()
      .addLong("uniqueness", System.nanoTime())
      .toJobParameters()
    );
}
```

- On peut aussi lancer le job suivant dans une séquence de JobInstances déterminée par le JobParametersIncrementer lié au job spécifié avec SimpleJobOperator.startNextInstance():

```
@Autowired
private JobOperator operator;

@Autowired
private JobExplorer jobs;

@Scheduled(fixedRate = 5000)
public void run() throws Exception {
    List<JobInstance> lastInstances
        = jobs.getJobInstances(JOB_NAME, 0, 1);
    if (lastInstances.isEmpty()) {
        jobLauncher.run(customerReportJob(),
        new JobParameters());
    } else {
        operator.startNextInstance(JOB_NAME);
    }
}
```

7. Test unitaire

- Généralement, pour exécuter des tests unitaires dans une application Spring Boot, le framework doit charger un **ApplicationContext** correspondant. 2 annotations sont utilisées à cet effet:

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = {...})
```

- Il existe une classe d'utilitaires **org.springframework.batch.test.JobLauncherTestUtils** pour tester les jobs batchs.
- Il fournit des méthodes pour lancer un job complet et permet de tester de bout en bout des **steps** individuels sans avoir à exécuter chaque step du travail. Il doit être déclaré comme un bean Spring:

```
@Configuration
public class BatchTestConfiguration {
    @Bean
    public JobLauncherTestUtils jobLauncherTestUtils() {
        return new JobLauncherTestUtils();
    }
}
```

- Un test typique pour un job et un step peut utiliser tous les frameworks mocks:

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = {BatchApplication.class,
BatchTestConfiguration.class})
public class CustomerReportJobConfigTest {

    @Autowired
    private JobLauncherTestUtils testUtils;
    @Autowired
    private CustomerReportJobConfig config;

    @Test
    public void testEntireJob() throws Exception {
        final JobExecution result = testUtils.getJobLauncher()
                    .run(config.customerReportJob(),
                         testUtils.getUniqueJobParameters());
        Assert.assertNotNull(result);
        Assert.assertEquals(BatchStatus.COMPLETED,
                            result.getStatus());
    }

    @Test
    public void testSpecificStep() {
        Assert.assertEquals(BatchStatus.COMPLETED,
          testUtils.launchStep("taskletStep").getStatus());
    }
}
```

7. Test unitaire

- Spring Batch introduit des scopes supplémentaires pour les contextes de step et de job.
- Les objets de ces scopes utilisent le conteneur Spring en tant que fabrique d'objets.
  - Il n'y a donc qu'une seule instance de chaque bean de ce type par exécution de step ou de job. En outre, l'association lazzy des références accessibles à partir de **StepContext** ou de **JobContext** est prise en charge.
- Les composants configurés au moment de l'exécution pour être définis par step ou par job sont difficiles à tester en tant que composants autonomes, sauf si on a le moyen de définir le contexte comme s'ils étaient dans un step ou dans l'exécution d'un job.
- C'est l'objectif des composants **org.springframework.batch.test.StepScopeTestExecutionListener** et **org.springframework.batch.test.StepScopeTestUtils** dans Spring Batch, ainsi que de **JobScopeTestExecutionListener** et **JobScopeTestUtils**.
- Les **TestExecutionListeners** sont déclarés au niveau de la classe et son travail consiste à créer un contexte d'exécution d'étape pour chaque méthode de test. Par exemple:

```
@RunWith(SpringRunner.class)
@TestExecutionListeners({DependencyInjectionTestExecutionListener.class, StepScopeTestExecutionListener.class})
@ContextConfiguration(classes = {BatchApplication.class, BatchTestConfiguration.class})
public class BirthdayFilterProcessorTest {

    @Autowired
    private BirthdayFilterProcessor processor;

    public StepExecution getStepExecution() {
        return MetaDataInstanceFactory.createStepExecution();
    }

    @Test
    public void filter() throws Exception {
        final Customer customer = new Customer();
        customer.setId(1);
        customer.setName("name");
        customer.setBirthday(new GregorianCalendar());
        Assert.assertNotNull(processor.process(customer));
    }

}
```

7. Test unitaire

- Il existe 2 TestExecutionListeners.
- L'une provient du framework **Spring Test** habituel et gère l'injection de dépendance à partir du contexte d'application configuré.
- L'autre est **SpringBatch StepScopeTestExecutionListener** qui configure le contexte d'étape pour l'injection de dépendance dans les tests unitaires.
- Un **StepContext** est créé pour la durée d'une méthode de test et mis à la disposition de toutes les dépendances injectées.
- Le comportement par défaut consiste simplement à créer une **StepExecution** avec des propriétés fixes.
- Alternativement, le StepContext peut être fourni par le scénario de test en tant que Factory method renvoyant le type correct.

- Une autre approche est basée sur la classe d'utilitaires **StepScopeTestUtils**.
- Cette classe est utilisée pour créer et manipuler **StepScope** dans des tests unitaires de manière plus flexible sans utiliser d'injection de dépendance.
- Par exemple, la lecture de l'ID du client filtré par le processeur peut être effectuée:

```
@Test
public void filterId() throws Exception {
    final Customer customer = new Customer();
    customer.setId(1);
    customer.setName("name");
    customer.setBirthday(new GregorianCalendar());
    final int id = StepScopeTestUtils.doInStepScope(
        getStepExecution(),
        () -> processor.process(customer).getId()
    );
    Assert.assertEquals(1, id);
}
```

8. Parallel processing

- Exemple: évolution
  - Nous voulons convertir les informations client de 5 fichiers XML en fichiers texte correspondants, en utilisant une implémentation multithread.

- Nous pouvons y parvenir en utilisant un seul **Partitioner** de Job et Step.
- Nous aurons 5 threads, un pour chacun des fichiers XML.
  - Tout d'abord, créer un job:

```
@Bean(name = "partitionerJob")
public Job partitionerJob()
  throws UnexpectedInputException,
  MalformedURLException, ParseException {
    return jobs.get("partitioningJob")
      .start(partitionStep())
      .build();
}
```

- On note que, ce job commence par **PartitioningStep**.
- C'est notre étape principale qui sera divisée en différentes étapes esclaves:

- Ici, nous allons créer le PartitioningStep à l'aide de StepBuilderFactory.
- Pour cela, nous devons donner les informations sur SlaveSteps et le partitionneur.

```
@Bean
public Step partitionStep()
  throws UnexpectedInputException,
  MalformedURLException, ParseException {
    return steps.get("partitionStep")
      .partitioner("slaveStep", partitioner())
      .step(slaveStep())
      .taskExecutor(taskExecutor())
      .build();
}
```

- Le partitionner est une interface qui permet de définir un ensemble de valeurs d'entrée pour chacun des esclaves.
- En d'autres termes, la logique pour diviser les tâches en threads respectifs va ici.

8. Parallel processing

- Nous allons en créer une implémentation, appelée **CustomMultiResourcePartitioner**, dans laquelle nous placerons les noms des fichiers d'entrée et de sortie dans ExecutionContext pour les transmettre à chaque step de l'esclave:

```
public class CustomMultiResourcePartitioner implements Partitioner {

    @Override
    public Map<String, ExecutionContext> partition(int gridSize) {
        Map<String, ExecutionContext> map = new HashMap<>(gridSize);
        int i = 0, k = 1;
        for (Resource resource : resources) {
            ExecutionContext context = new ExecutionContext();
            Assert.state(resource.exists(), "Resource does not exist: "
              + resource);
            context.putString(keyName, resource.getFilename());
            context.putString("opFileName", "output"+k+++".xml");
            map.put(PARTITION_KEY + i, context);
            i++;
        }
        return map;
    }
}
```

8. Parallel processing

- Création d’un classe Bean, qui reçoit le nom du répertoire source pour les fichiers d’entrée:

```
@Bean
public CustomMultiResourcePartitioner partitioner() {
    CustomMultiResourcePartitioner partitioner = new CustomMultiResourcePartitioner();
    Resource[] resources;
    try {
        resources = resoursePatternResolver.getResources("file:src/main/resources/input/*.xml");
    } catch (IOException e) {
        throw new RuntimeException("I/O problems when resolving the input file pattern.", e);
    }
    partitioner.setResources(resources);
    return partitioner;
}
```

- Nous définirons l’étape esclave, comme toute étape avec le reader et le writer. Le lecteur et le rédacteur seront identiques à ceux décrits dans notre exemple d'introduction, à ceci près qu'ils recevront le paramètre filename du StepExecutionContext.
- Notez que ces beans doivent être étendus pour pouvoir recevoir les paramètres stepExecutionContext à chaque étape. S'ils ne sont pas soumis à l'étape, leurs beans seront créés initialement et n'accepteront pas les noms de fichiers au niveau de l'étape:

8. Parallel processing: reader slave

```
@StepScope
@Bean
public FlatFileItemReader<Transaction> itemReader(
  @Value("#{stepExecutionContext[fileName]}") String filename)
  throws UnexpectedInputException, ParseException {

    FlatFileItemReader<Transaction> reader
      = new FlatFileItemReader<>();
    DelimitedLineTokenizer tokenizer = new DelimitedLineTokenizer();
    String[] tokens  = {"username", "userid", "transactiondate", "amount"};
    tokenizer.setNames(tokens);
    reader.setResource(new ClassPathResource("input/" + filename));
    DefaultLineMapper<Transaction> lineMapper = new DefaultLineMapper<>();
    lineMapper.setLineTokenizer(tokenizer);
    lineMapper.setFieldSetMapper(new RecordFieldSetMapper());
    reader.setLinesToSkip(1);
    reader.setLineMapper(lineMapper);
    return reader;
}
```

8. Parallel processing: writer slave

```
@Bean
@StepScope
public ItemWriter<Transaction> itemWriter(Marshaller marshaller,
  @Value("#{stepExecutionContext[opFileName]}") String filename)
  throws MalformedURLException {
    StaxEventItemWriter<Transaction> itemWriter = new StaxEventItemWriter<Transaction>();
    itemWriter.setMarshaller(marshaller);
    itemWriter.setRootTagName("transactionRecord");
    itemWriter.setResource(new ClassPathResource("txt/" + filename));
    return itemWriter;
}
```

- Tout en mentionnant le lecteur et le rédacteur dans l'étape slave, nous pouvons passer les arguments comme null, car ces noms de fichiers ne seront pas utilisés, car ils recevront les noms de fichiers de stepExecutionContext:

```
@Bean
public Step slaveStep()
  throws UnexpectedInputException, MalformedURLException, ParseException {
    return steps.get("slaveStep").<Transaction, Transaction>chunk(1)
      .reader(itemReader(null))
      .writer(itemWriter(marshaller(), null))
      .build();
}
```

9. Spring Batch Listener

- Il y a 6 «listeners» pour intercepter l'exécution pas à pas. Le nom des classes est explicite.
  - StepExecutionListener
  - ItemReadListener
  - ItemProcessListener
  - ItemWriteListener
  - ChunkListener
  - SkipListener

```
public class CustomStepListener implements StepExecutionListener {

    @Override
    public void beforeStep(StepExecution stepExecution) {
        System.out.println("StepExecutionListener - beforeStep");
    }

    @Override
    public ExitStatus afterStep(StepExecution stepExecution) {
        System.out.println("StepExecutionListener - afterStep");
        return null;
    }

}
```

9. Spring Batch Listener

```
public class CustomItemReaderListener implements
ItemReadListener<Domain> {

    @Override
    public void beforeRead() {
        System.out.println("ItemReadListener -
                            beforeRead");
    }

    @Override
    public void afterRead(Domain item) {
        System.out.println("ItemReadListener -
                            afterRead");
    }

    @Override
    public void onReadError(Exception ex) {
        System.out.println("ItemReadListener -
                            onReadError");
    }

}
```

```
public class CustomItemWriterListener implements
ItemWriteListener<Domain> {

    @Override
    public void beforeWrite(List<? extends Domain> items)
{
        System.out.println("ItemWriteListener -
                            beforeWrite");
    }

    @Override
    public void afterWrite(List<? extends Domain> items) {
        System.out.println("ItemWriteListener -
                            afterWrite");
    }

    @Override
    public void onWriteError(Exception exception, List<?
extends Domain> items) {
        System.out.println("ItemWriteListener -
                            onWriteError");
    }

}
```

10. Référence

- https://docs.spring.io/spring-batch/trunk/reference/html/

- https://spring.io/projects/spring-boot