Spring Batch en Kotlin

Sylvain Debras
ACII by Audensiel

audensiel.com

Généralités Kotlin

https://kotlinlang.org/docs/home.html

- Langage moderne qui aide les dev. à gagner en productivité
- Être plus concis, moins de code
- 100 % compatible avec le code Java
- Statiquement typé (les erreurs de type apparaissent à la compilation)
- Inférence de type (pas besoin de déclarer le type d'une variable à chaque fois)
- POO et programmation fonctionnelle (first-class function, immuabilité, fonctions "pures")

Kotlin

Exemples Kotlin (1)

```kotlin
fun main() {
    val a: Int = 1  // immediate assignment
    val b = 2   // `Int` type is inferred
    val c: Int  // Type required when no initializer is provided
    c = 3       // deferred assignment
    println("a = $a, b = $b, c = $c")
}
```

```
a = 1, b = 2, c = 3
```

```kotlin
fun main() {
    var a = 1
    // simple name in template:
    val s1 = "a is $a"

    a = 2
    // arbitrary expression in template:
    val s2 = "${s1.replace("is", "was")}, but now is $a"
    println(s2)
}
```

```
a was 1, but now is 2
```

```kotlin
data class Customer(val name: String, val email: String)
```

Exemples Kotlin (2)

```kotlin
fun double(x: Int): Int {
    return 2 * x
}
```

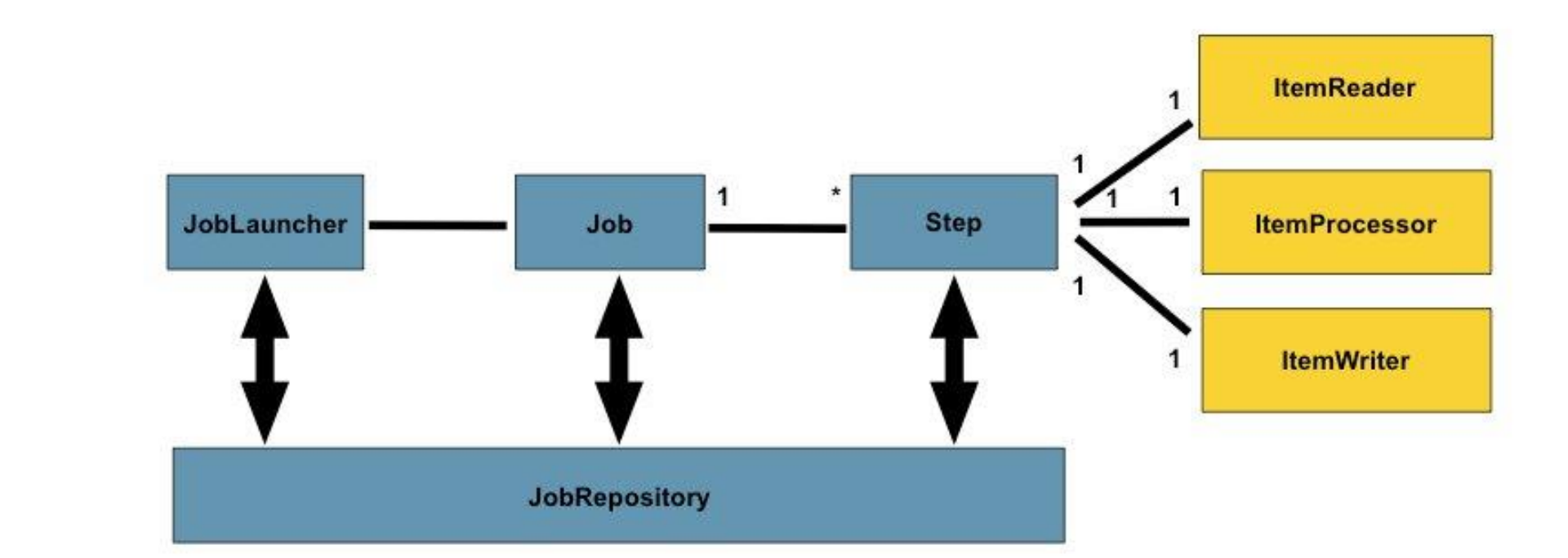
```kotlin
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> {
        print("x is neither 1 nor 2")
    }
}
```

```kotlin
val max = if (a > b) {
    print("Choose a")
    a
} else {
    print("Choose b")
    b
}
```

Généralités Spring Batch

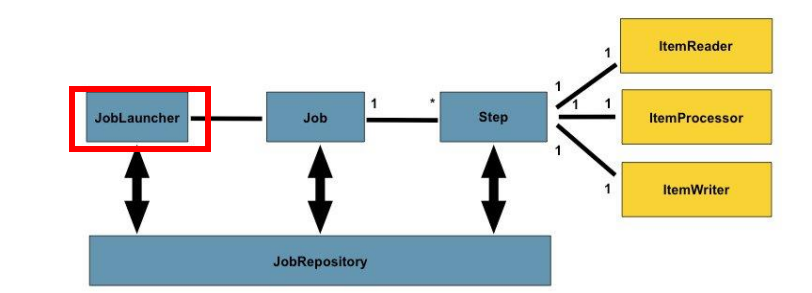
- Un batch est un programme qui va traiter un gros volume de données
- Spring Batch est un framework Java basé sur Spring et qui a pour but de faciliter la création d'un batch
- Grande communauté (Quelqu'un a souvent déjà fait ce que vous voulez faire :))
- Spring Batch permet de créer des batchs Java facilement en répondant à certains critères indispensables :
  - Intégrer le batch dans une architecture utilisant le framework Spring
  - Division du code bien définie permettant une meilleure maintenabilité et une logique commune à la création de batch
  - Traiter de gros volumes de données par lots de façon qualitative

Schéma global



Job Launcher

- Permet de lancer un job
- Possibilité de lancer un job de façon synchrone ou asynchrone (via une appli. Web ou une API Rest)



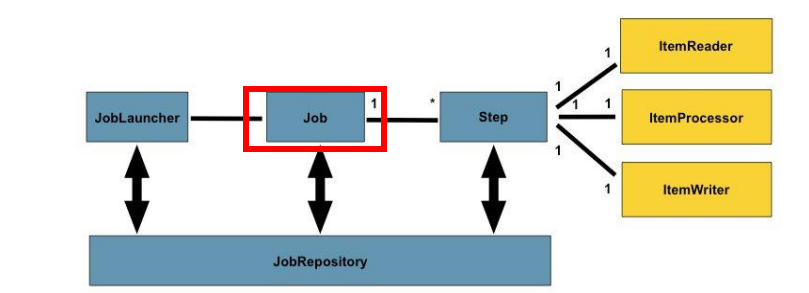

```kotlin
@SpringBootApplication
class BatchExempleKotlinApplication(
    val jobLauncher: JobLauncher,
    val job: Job
): CommandLineRunner {
    override fun run(vararg args: String?) {
        val execution: JobExecution = jobLauncher.run(job, JobParameters())
        log.info { "STATUS :: ${execution.status}" }

        val exitCode :Int  = when (execution.status) {
            BatchStatus.COMPLETED -> 0
            BatchStatus.ABANDONED -> 4
            BatchStatus.FAILED -> 5
            else -> 1
        }
        exitProcess(exitCode)
    }

}
```

Le Job

- C'est la représentation du batch à travers le Framework.
- Dans cette classe, on va pouvoir définir différentes Steps au sein de l'exécution du batch

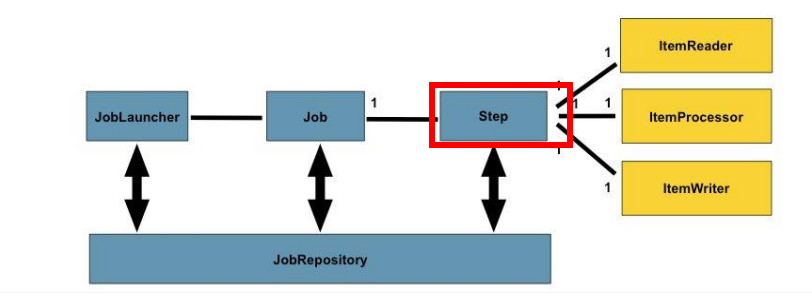



```kotlin
@EnableBatchProcessing
@Configuration
class JobConfiguration  (
    private val jobRepository: JobRepository
) {

    @Bean
    fun job(
        @Qualifier("stepExampleUn") stepExampleUn: Step,
        @Qualifier("stepExampleDeux") stepExampleDeux: Step): Job {
        return JobBuilder( name: "myJob", jobRepository)
            .start(stepExampleUn)
            .next(stepExampleDeux)
            .build()
    }
}
```

La Step

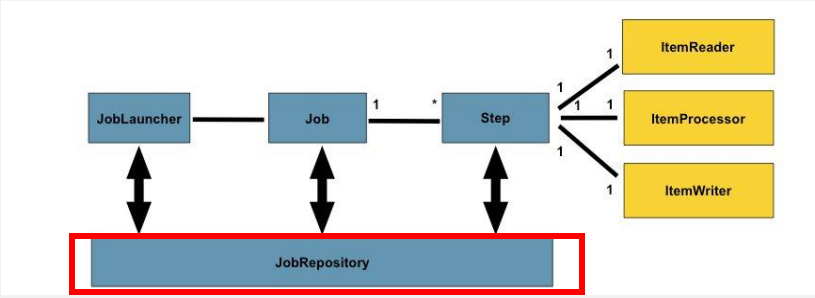
- Une Step est un objet qui encapsule indépendamment les différentes phases d'un Job.
- Chaque Job est composé d'une ou plusieurs Step.
- Une Step contient les infos. Nécessaires pour définir et contrôler le batch




```kotlin
@Bean
fun stepExampleOne(
    jobRepository: JobRepository,
    transactionManager: PlatformTransactionManager,
    @Qualifier("itemReaderCustom") itemReaderCustom: CompositeCursorItemReader<ContractInvoice>,
    compositeItemProcessor: CompositeItemProcessor<ContractInvoice, ContractInvoice>,
    @Qualifier("itemWriterStepOne") itemWriterStepOne: FlatFileItemWriter<ContractInvoice>,
    stepListener: StepExampleListener,
) : Step {
    return StepBuilder("stepExampleOne", jobRepository)
        .chunk<ContractInvoice, ContractInvoice> (2, transactionManager)
        .listener(stepListener)
        .reader(itemReaderCustom)
        .processor(compositeItemProcessor)
        .writer(itemWriterStepOne)
        .faultTolerant()
        .skipLimit(5)
        .skip(ValidationException::class.java)
        .build()
}
```

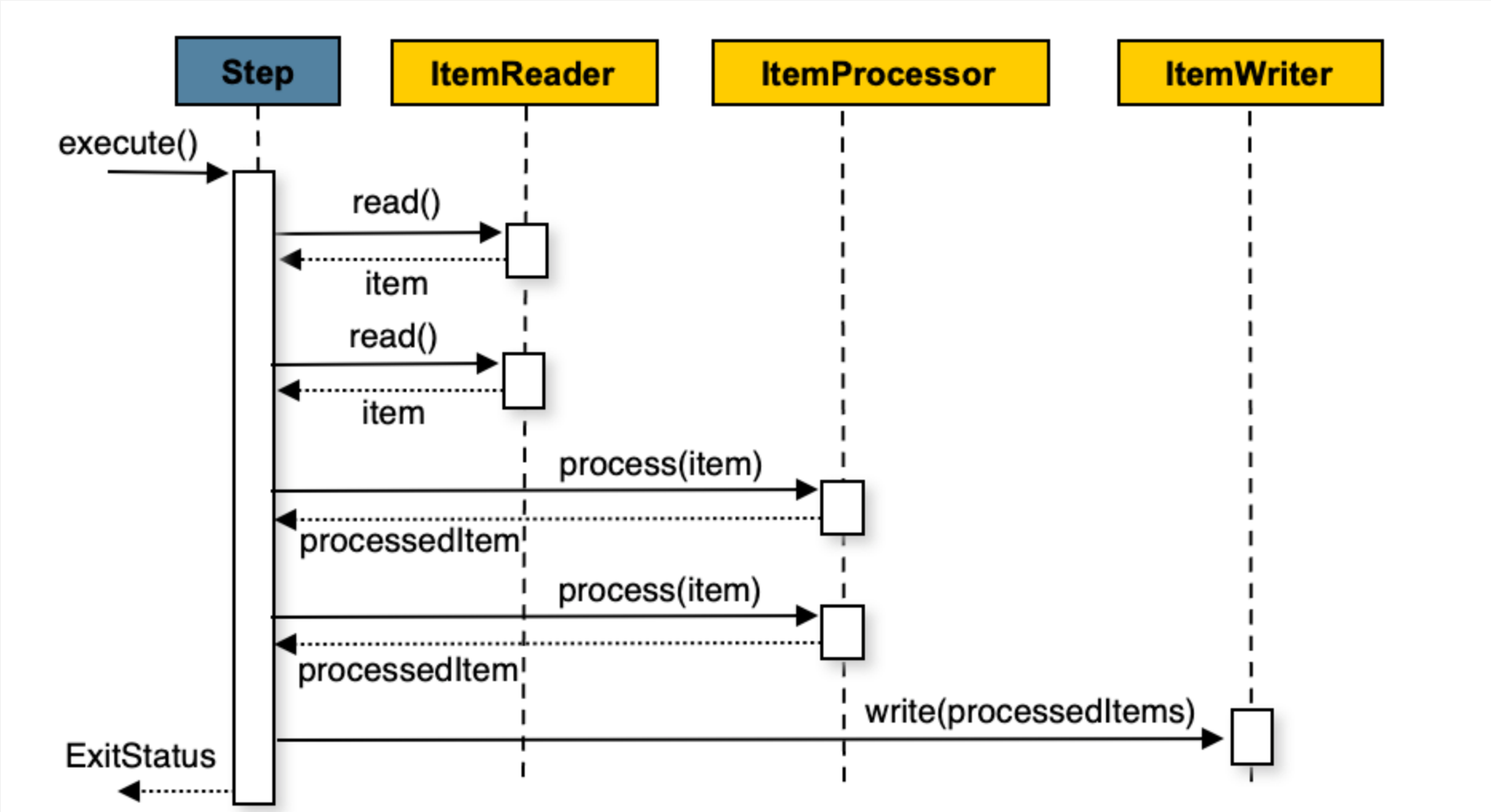
Le Job Repository

- La classe qui va stocker un grand nombre de données autour du job
- On peut récupérer l'historique des différents jobs et beaucoup d'autres données pertinentes
- Permet d'effectuer de la reprise sur erreur
- Permet d'effectuer des pauses dans le traitement



Le chunk

- Notion de transaction
- Les éléments sont lus un à un dans les limites de la transaction
- Ecriture des éléments en une seule transaction à la fin
- Et on recommence ..



Le reader

- Interface ItemReader
- Classes : JdbcCursorItemReader, FlatFileItemReader, JsonItemReader, etc.

```
Strategy interface for providing the data.
Implementations are expected to be stateful and will be called multiple times for each batch, with each
call to read() returning a different value and finally returning null when all input data is exhausted.
Implementations need not be thread-safe and clients of a ItemReader need to be aware that this is the
case.
A richer interface (e.g. with a look ahead or peek) is not feasible because we need to support
transactions in an asynchronous batch.
Since: 1.0
Author: Rob Harrop, Dave Syer, Lucas Ward, Mahmoud Ben Hassine, Taeik Lim
@FunctionalInterface
public interface ItemReader<T> {

    Reads a piece of input data and advance to the next one. Implementations must return null at
    the end of the input data set. In a transactional setting, caller might get the same item twice from
    successive calls (or otherwise), if the first call was in a transaction that rolled back.
    Returns: T the item to be processed or null if the data source is exhausted
    Throws: ParseException – if there is a problem parsing the current record (but the next one may
            still be valid)
            NonTransientResourceException – if there is a fatal exception in the underlying
            resource. After throwing this exception implementations should endeavour to return null
            from subsequent calls to read.
            UnexpectedInputException – if there is an uncategorised problem with the input data.
            Assume potentially transient, so subsequent calls to read might succeed.
            Exception – if an there is a non-specific error.
    @Nullable
    T read() throws Exception, UnexpectedInputException, ParseException, NonTransientResourceException;

}
```

Le processor

- Interface ItemProcessor
- Classes : CompositeItemProcessor, FunctionItemProcessor, ValidatingItemProcessor, etc.

```
Interface for item transformation. Given an item as input, this interface provides an extension point
which allows for the application of business logic in an item oriented processing scenario. It should be
noted that while it's possible to return a different type than the one provided, it's not strictly
necessary. Furthermore, returning null indicates that the item should not be continued to be
processed.
Author:          Robert Kasanicky, Dave Syer, Mahmoud Ben Hassine, Taeik Lim
Type parameters: <I> – type of input item
                 <O> – type of output item
@FunctionalInterface
public interface ItemProcessor<I, O> {

    Process the provided item, returning a potentially modified or new item for continued processing.
    If the returned result is null, it is assumed that processing of the item should not continue.
    A null item will never reach this method because the only possible sources are:
    • an ItemReader (which indicates no more items)
    • a previous ItemProcessor in a composite processor (which indicates a filtered item)
    Params: item – to be processed, never null.
    Returns: potentially modified or new item for continued processing, null if processing of the
             provided item should not continue.
    Throws: Exception – thrown if exception occurs during processing.
    @Nullable
    O process(@NonNull I item) throws Exception;

}
```

Le writer

- Interface ItemWriter
- Classes FlatFileItemWriter, JdbcCursorItemWriter, CompositeItemWriter etc.

```
Basic interface for generic output operations. Class implementing this interface will be responsible for
serializing objects as necessary. Generally, it is responsibility of implementing class to decide which
technology to use for mapping and how it should be configured.

The write method is responsible for making sure that any internal buffers are flushed. If a transaction
is active it will also usually be necessary to discard the output on a subsequent rollback. The resource
to which the writer is sending data should normally be able to handle this itself.

Author: Dave Syer, Lucas Ward, Taeik Lim, Mahmoud Ben Hassine

@FunctionalInterface
public interface ItemWriter<T> {

    Process the supplied data element. Will not be called with any null items in normal operation.
    Params: chunk – of items to be written. Must not be null.
    Throws: Exception – if there are errors. The framework will catch the exception and convert or
            rethrow it as appropriate.

    void write(@NonNull Chunk<? extends T> chunk) throws Exception;

}
```

La tasklet

- Pour effectuer des traitements plus simples et théoriquement sans lecture ni écriture de données.
- Exemple : Génération d'un rapport via des données collectées pendant le traitement batch, exécution d'une procédure stockée

```
Strategy for processing in a step.
Author: Dave Syer, Mahmoud Ben Hassine, Taeik Lim
@FunctionalInterface
public interface Tasklet {

    Given the current context in the form of a step contribution, do whatever is necessary to process
    this unit inside a transaction. Implementations return RepeatStatus.FINISHED if finished. If not
    they return RepeatStatus.CONTINUABLE. On failure throws an exception.
    Params: contribution – mutable state to be passed back to update the current step execution
            chunkContext – attributes shared between invocations but not between restarts
    Returns: an RepeatStatus indicating whether processing is continuable. Returning null is
             interpreted as RepeatStatus.FINISHED
    Throws: Exception – thrown if error occurs during execution.

    @Nullable
    RepeatStatus execute(StepContribution contribution, ChunkContext chunkContext) throws Exception;

}
```

Les listeners

- Possiblités de poser des listeners sur l'ensemble des composants existants avant et après traitement
- StepExecutionListener
- ChunkListener
- ItemReadListener
- ItemProcessListener
- ItemWriteListener
- SkipListener

Des exemples

Merci pour votre attention

- N'hésitez pas d'abuser de la doc. Spring Batch
- Elle est vraiment bien faite !!!
- https://docs.spring.io/spring-batch/reference/index.html
- En cas de problèmes ou questions
  - sdebras@audensiel.fr

audensiel.com