- Agence de la Biomédecine -

DSI pôle SIAM unité Architecture

GUIDE DE DEVELOPPEMENT SPRING BATCH

Version 1.0



agence de la biomédecine	**GUIDE DE DEVELOPPEMENT**	*Version 1.0*

Description du document	
Sujet du Document	**Guide de développement**
Date du document	**17/01/2022**
Nom du Document	**Guide de développement**
Auteur	**André-Pierre ABELLARD**
Version	**2.4**
Statut	

Diffusion du document					
Version	**Nom**	**Direction / Service**	**Pour validation**	**Pour informatic**	**Date**
2.3	Lionel Corlobé				11/09/2020

Validité du document			
Version	**Nom**	**Direction / Service**	**Date**

Historique du document		
Version	**Date**	**Description**
2.3	11/09/2020	

Sommaire

1 PRESENTATION GENERALE 6
1.1 OBJECTIFS DU DOCUMENT 6
2 CONCEPT ET ARCHITECTURE GENERALE D'UNE APPLICATION SPRING BATCH 7
2.1 PRINCIPALES FONCTIONNALITES 7
2.2 ARCHITECTURE D'UNE APPLICATION SPRING BATCH 7
2.2.1 Représentation générale de l'application 7
2.2.2 Composants d'un Job 7
2.2.3 Cinématique d'un Job Spring Batch 8
2.3 NOTION SUR SPRING BATCH 8
2.3.1 Notions sur le Job Repository 8
2.3.2 Notions, création et configuration d'un Job 9
2.3.3 Notion de Listeners 9
2.3.4 Notions de Tasklets et chunk 9
2.4 EXECUTION D'UN JOB BATCH 10
2.4.1 Concept d'exécution d'un job 10
2.4.2 Planification d'un Job 10
2.5 LECTURE DE DONNEES 10
2.5.1 Principes 10
2.5.2 Lecture de Fichiers plats 11
2.5.3 Lecture en base de données 12
2.5.4 Implémentation de lecteur personnalisé 12
2.6 ÉCRITURE DE DONNEES 13
2.6.1 Principes 13
2.6.2 Écriture de Fichiers 13
2.6.3 Écriture en base de Données 15
2.6.4 Implémentation d'un Writer personnalisé 15
2.6.5 Chainage d'ItemWriter 15
2.7 LE TRAITEMENT DES DONNEES 16
2.7.1 Objectifs 16
2.7.2 Traitement métiers des données 16
2.7.3 Configuration de *l'ItemProcessor* 16
2.7.4 Implémentation de *l'ItemProcessor* 16

2.7.5 Modification de données par l'ItemProcessor 17
2.8 GESTION DES ERREURS 17
2.8.1 Principes 17
2.8.2 Détermination des scénarios d'erreurs 18
2.8.3 Les différents modes de gestion d'erreurs 18
3 DEVELOPPEMENT D'APPLICATION SPRING BATCH 19
3.1 AVANT PROPOS 19
3.2 CONFIGURATION DU PROJET 19
3.2.1 Pile technique 19
3.2.2 Dépendances requises pour les applications Spring Batch 19
3.2.3 Maven 20
3.2.4 Gestion des projets 25
3.2.5 Configuration de l'application 25
3.2.6 Variabilisation des développements pour la gestion des environnements 25
3.3 STRUCTURE DE L'APPLICATION 26
3.3.1 Les différents composants 26
3.3.2 Les packages 26
3.3.3 Les classes java 26
3.4 MISE EN PLACE DU LANCEUR DES BATCHES 27
3.4.1 Exécution des Batches 27
3.4.2 Planification et ordonnancement des Batches 28
3.4.3 Historisation du lancement des Batches 28
3.5 MISE EN PAGE ET DOCUMENTATION 29
3.5.1 Mise en page du code 29
3.5.2 Documentation et javadoc 29
3.6 GESTION DES ERREURS 29
3.6.1 Objectif 29
3.6.2 Les Exceptions 30
3.6.3 Batches et Exception 32
3.7 GESTION DES LOGS 32
3.7.1 Objectif du log 32
3.7.2 Logger Java 32
3.7.3 Définition des niveaux de log 34
3.7.4 Message de log 35

3.8 DEFINITION D'UN ITEMREADER 37
3.8.1 L'ItemReader 37
3.8.2 Les méthodes d'un ItemReader 37
3.9 DEFINITION D'UN ITEMPROCESSOR 37
3.9.1 L'ItemProcessor 37
3.9.2 Les méthodes d'un ItemProcessor 37
3.10 DEFINITION D'UN ITEMWRITER 37
3.10.1 L'ItemWriter 37
3.10.2 Les méthodes d'un ItemWriter 38
3.11 DEFINITION D'UN DATASERVICE 38
3.11.1 Le DataService 38
3.11.2 Méthode annotée @Transactional et accès base 38
3.12 DEFINITION D'UN REPOSITORY 39
3.12.1 Le Repository 39
3.12.2 Optimisation des recherches 39
3.13 DEFINITION D'UNE ENTITE 39
3.13.1 Entité JPA 39
3.13.2 Définition des propriétés 39

1 Présentation générale

1.1 OBJECTIFS DU DOCUMENT

L'objectif de ce document est de constituer un guide des bonnes pratiques de développement d'un batch de traitement de données basé sur le socle technique Spring Batch. Il fait suite et complète le guide de développement des applications Angular et des services Spring REST.

Il est constitué de plusieurs chapitres décrivant :

- La présentation de l'architecture générale nécessaire au fonctionnement d'un batch.
- Les recommandations de développement du batch de traitement de données.
- La gestion de la configuration décrivant la gestion des dépendances Spring Batch et l'externalisation de la configuration du batch de traitement de données.



Enrichissement du document

Ce guide sera mis à jour au fil de l'eau pendant les développements.

2 Concept et architecture générale d'une application Spring Batch

2.1 PRINCIPALES FONCTIONNALITES

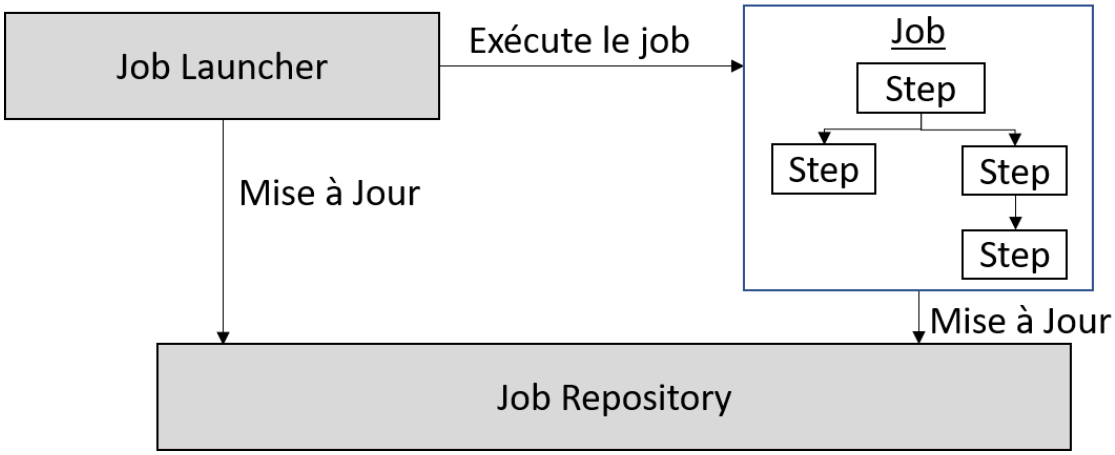
Une application Spring Batch a pour but d'apporter les fonctionnalités suivantes :

- Capacité à importer, exporter, traiter et analyser de larges volumes de données.
- Automatiser le fonctionnement de l'application sans intervention extérieure (sauf erreur importante nécessitante une intervention extérieure.)
- Solidité en fonctionnement, en cas de manipulation de données invalides, l'application ne doit pas crasher ou arrêter prématurément l'application.
- L'application doit effectuer ses « tâches » dans un délai raisonnable (à définir avec le client), afin d'éviter de perturber le fonctionnement d'applications tournant simultanément.
- Mise en place d'environnement de tests unitaires ou d'intégration.

2.2 ARCHITECTURE D'UNE APPLICATION SPRING BATCH

2.2.1 REPRESENTATION GENERALE DE L'APPLICATION

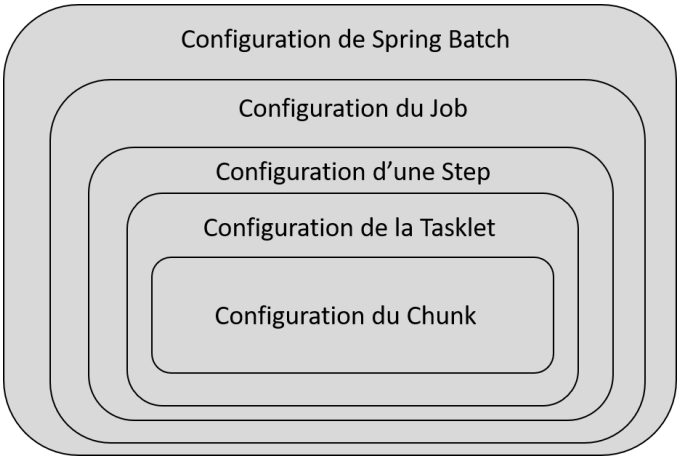
Une application Spring Batch est une application Java s'exécutant en tâche de fond ou à des intervalles de temps réguliers et prédéfinis ; elle et chargée de lancer le traitement principal (Job Launcher). Ce traitement a à sa charge l'ordonnancement des différentes travaux (Job) à mener, ainsi que leur suivi en traçant leur exécution dans un dépôt central (Job Repository). Les travaux sont chargés du traitement des données via un enchainement d'étapes (Step) élémentaires (lecture des données, transformation des données, écriture des données, etc.). On peut résumer l'ensemble par le schéma suivant :



A noter : Que le Job Launcher et le JobRepository sont gérés par Spring Batch.

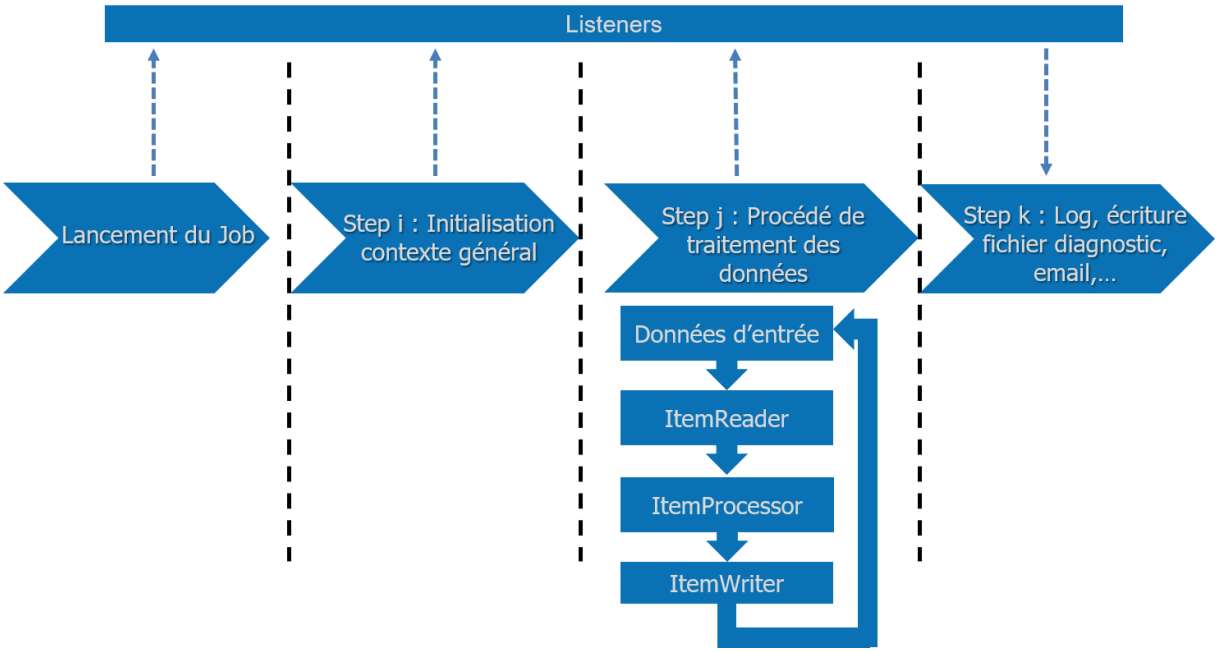
2.2.2 COMPOSANTS D'UN JOB

Un Job est composé de différentes steps et listeners permettant de décrire les séquences qui vont se dérouler durant l'exécution de celui-ci, ainsi que les entrées et sorties du job qui interagissent via les Listeners du Job. Une vision hiérarchique du processus de Batch peut alors être défini. Le schéma suivant permet de représenter l'encapsulation et la hiérarchisation de ces différents éléments :



Ainsi, on peut voir que la configuration de spring-batch contient la configuration du Job. Cette dernière encapsule la configuration des Steps. La configuration des steps intègre elle-même la configuration des tasklets qui incluent la configuration des chunk. Cette hiérarchie dans la configuration de Spring Batch est comme dit précédemment un point fort pour la maintenabilité et la recherche d'erreur des applications de type Batch.

2.2.3 CINEMATIQUE D'UN JOB SPRING BATCH



2.3 NOTION SUR SPRING BATCH

2.3.1 NOTIONS SUR LE JOB REPOSITORY

Dans Spring Batch, le Job Repository est le garant de l'intégrité des métadonnées lors du déroulement de notre Job. Il existe une seule Implémentation de l'interface JobRepository, il s'agit de la classe SimpleJobRepository. Cette classe est basé sur un ensemble de DAO utilisé pour l'interaction et la gestion des données. Spring Batch nous permet d'avoir à ce niveau deux types de DAO :

- En mémoire avec aucune persistance des métadonnées ;
- Ou en persistant les métadonnées en utilisant JDBC.

Dans le cadre d'un environnement de production, il est souvent déconseillé d'utiliser le premier type de DAO, car les données des batches sont perdus entre les exécutions des jobs. Cependant, pour le deuxième type de DAO persistante, il nous faudrait implémenter des informations additionnelles dans la configuration du Job, ce qui peut alourdir nos traitements de données, mais qui nous permet une plus grande robustesse de notre process de batch.

Il est donc ***fortement conseillé*** d'utiliser le deuxième type de DAO, c'est-à-dire de persister les métadonnées en base, afin d'éviter les pertes de celles-ci en serveur de production, lors des traitements des flux de données. De plus, cette deuxième option est utile dans le cadre de la gestion d'erreur et de l'arrêt d'un job en cours de fonctionnement pour sa reprise au moment de l'arrêt de celui-ci.

2.3.2 NOTIONS, CREATION ET CONFIGURATION D'UN JOB

Un job Spring Batch est une séquence de plusieurs étapes ou Step(s), configurée(s) dans le fichier de configuration XML de Spring Batch. Cette décomposition en steps distinctes du job permet d'obtenir une vision plus claire ; à la fois d'un point de vue modélisation du processus des jobs mais aussi du point de vue des tests et de la maintenabilité et des tests des différentes étapes du job.

2.3.3 NOTION DE LISTENERS

L'utilisation de Spring Batch nous donne la possibilité de spécifier et de contrôler, via des listeners, le déroulement des Jobs ou des steps. Cette fonctionnalité présente dans Spring Batch est particulièrement intéressante. En effet, les listeners peuvent être définis à chaque niveau du processus de Batch, où l'on souhaite ajouter ou modifier la séquence du batch.

On définit 3 types de Listeners :

- Le Job Listener : il sert à intervenir au niveau du Job du processus du batch ;
- Le Step Listener : il sert à intervenir au niveau Step du processus du batch ;
- L'Item Listener : Il sert à intervenir au niveau de l'item d'un chunk.

Cependant, il faut noter que cela dépend de la volonté de client sur le comportement à adopter suivant les cas abordés.

2.3.4 NOTIONS DE TASKLETS ET CHUNK

Spring Batch comme nous avons pu le voir précédemment, définit l'interface Step et fournit différentes implémentations de cette interface. La seule qui nous intéresse dans le cadre du développement d'une application est l'implémentation en utilisant TaskletStep qui permet de déléguer le traitement de données à un objet de type Tasklet. Il est aussi possible d'implémenter ses propres Tasklet lorsqu'il est nécessaire d'effectuer des opérations particulières (appels de procédure stockés, supprimer les fichiers temporaires à la fin d'un job, …).

Toutefois, si la Step suit le modèle classique Lecture-Traitement-écriture, il est alors recommandé d'utiliser la configuration XML de Spring Batch d'un élément Chunk et de configurer la Tasklet comme étant un traitement de type chunk. L'élément Chunk autorise alors la Step d'utiliser les Chunk de façon efficace afin de lire, traiter et écrire les données.

Il faut noter que l'élément Chunk de Spring Batch est « mappé » à une implémentation de Tasklet par la classe ChunkOrientedTasklet.

On peut donc résumer qu'une task ou tasklet (Tâche) est une simple commande ou itération de traitements sur des lots de données (ou chunk).

2.4 EXECUTION D'UN JOB BATCH

2.4.1 CONCEPT D'EXECUTION D'UN JOB

L'exécution du Job d'un batch repose principalement sur l'API Spring Batch Job Launcher. Cette API après implémentation pour votre Job de l'interface JobLauncher, nous permettra de lancer l'exécution du Job. Ensuite, suivant la méthode choisie pour exécuter celui-ci, l'implémentation, les paramètres et l'environnement d'exécution seront assez différents. Nous allons tout d'abord de voir comment le lancement des batch est géré actuellement pour les EDI. Ensuite nous verrons comment on gère la planification des Jobs, soit à intervalle de temps régulier, soit par une planification dite « agenda ».

Tout d'abord, nous allons voir les notions générales importantes et communes aux différentes méthodes d'exécution.

2.4.1.1 Notions de Job Parameters

La variable Job Parameters se comporte comme un mapper. C'est ainsi que l'on peut définir les ressources nécessaires, les dossiers et fichiers ciblés à l'exécution de notre Job.

Pour déclencher l'évaluation dynamique d'une propriété d'un Job Parameter, il faut impérativement utiliser la syntaxe suivante :

```
# {expression à évaluer}
```

2.4.1.2 Notions de Job Instance

Un « Job instance » correspond à l'exécution du job et les Job Parameters donnés. Par exemple quand il est planifié le 25 juillet 2018 à 15h, la date et l'heure sont des paramètres qui qualifient le « Job Instance » avec le Job lui-même. C'est une manière efficace et assez simple de définir une instance de Job. En effet, nous avons un contrôle sur les paramètres du job, comme la date Système de la base de données par exemple afin de définir l'instance de façon unique. On peut donc définir un « Job Instance » comme étant la somme du Job et des Jobs Parameters.

2.4.2 PLANIFICATION D'UN JOB

2.4.2.1 Objectifs

Bien que Spring Batch soit un excellent outil pour développer des traitements batch, les entreprises ont besoin de pouvoir planifier ceux-ci afin de pouvoir définir les moments opportuns pour lancer leur batch sans avoir à programmer ceux-ci. C'est pour cela que la planification de jobs est essentielle dans le cadre de batch process. On verra alors 3 moyens de planifications différents (Cron, Spring Scheduler, Quartz).

2.4.2.2 Utilisation de Cron

Le Cron est un planificateur de job qui repose sur les systèmes de type UNIX. Celui-ci permet d'exécuter des commandes ou des scripts de façon périodique, en utilisant les *expressions* « *cron* ». La configuration de cron est simple, on configure les commandes à lancer et quand les exécuter dans le fichier *crontab*.

2.4.2.3 Utilisation de Spring Scheduler

Le planificateur Spring Scheduler est un planificateur de lancement de tâches pouvant être utilisé dans Spring Batch.

2.4.2.4 Utilisation de Quartz

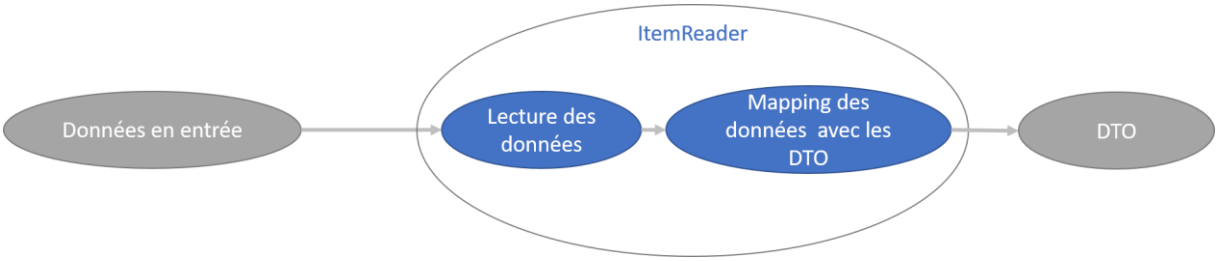
Le planificateur Quartz est un planificateur open source. Il est assez utilisé dans les environnements Java de toutes tailles. De plus, il est basé sur une communauté d'utilisateurs assez importantes.

2.5 LECTURE DE DONNEES

2.5.1 PRINCIPES

Dans cette partie, nous aborderons les types et principes de lecture de données en utilisant Spring Batch. Ces concepts sont la fondation même de la lecture de données via Spring Batch. Tout d'abord, on s'intéresse à l'interface ItemReader qui fournit le contrat concernant l'extraction lors de la lecture de données. Cette Interface est générique et contient une méthode « read » qui nous retournera le prochain élément qui sera lu par la méthode.

Cependant, il y a une autre interface disponible l'interface « ItemStream » qui permet de gérer les erreurs et de mettre à jour l'état du process.



2.5.2 LECTURE DE FICHIERS PLATS

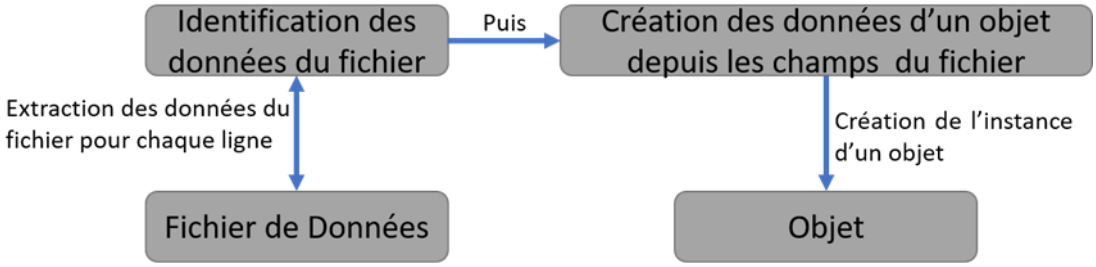
Les fichiers contiennent un ensemble de donnée à intégrer dans la structure de données de l'application qui va les utiliser. Chaque type de fichier est caractérisé par sa propre syntaxe et structure de données. Chaque structure permet d'identifier un élément de données différent. Pour configurer un type de fichier dans Spring Batch, le format de celui-ci doit être défini.

2.5.2.1 Utilisation de la classe FlatFileItemReader

Dans le cas des « Flat Files », il s'agit de fichiers de données pures qui contiennent peu ou pas du tout de métadonnées. Par exemple dans le cas des fichiers (TXT), les données sont séparées par une tabulation et la première ligne de celui-ci contient l'en-tête des colonnes. Dans la majorité des cas, le fournisseur du fichier définit le format de celui-ci. Cette information peut consister en une longueur de champs de données prédéfini ou correspondre à un séparateur de champs de données. Lors de la configuration de Spring Batch, pour la manipulation de « flat files » correspond à la définition du format de fichiers afin de permettre la corrélation entre données des champs du fichiers et objets de données.

L'ItemReader correspondant au « flat files » est responsable d'identifier les champs de données du fichier, puis de créer les objets de données correspondant à celles-ci.

On s'intéresse dans un premier temps à l'ItemReader pour « Flat Files », qui est résumé par le schéma suivant :



La gestion de la lecture du fichier se fait grâce à la classe FlatFileItemReader. En effet, cela nous permet d'éviter la gestion des entrées et sorties du fichier qui peuvent engendrer fréquemment des erreurs. Cette classe va donc nous fournir pour chaque ligne du fichier de données, une représentation objet (de ce que l'on attend comme type de données) qui pourra être persisté en base plus tard. La classe « FlatFileItemReader » s'appuie sur l'interface « LineMapper ». Cette Interface étant implémentée par la classe « DefaultLineMapper ». Cette classe utilise aussi deux interfaces « LineTokenizer » et « FieldSetMapper » permettant l'extraction des données d'une ligne d'un fichier. L'interface « LineTokenizer » est responsable de la séparation des données d'une ligne en tokens et l'interface « FieldSetMapper » permet de créer des objets à partir de tokens.

Donc pour résumer comment la méthode de lecture d'un fichier de données est effectuée, la classe FlatFileItemReader nous autorise à paramétrer un certain nombre d'attributs concernant le type de fichier lu dans le cadre des batches.

2.5.2.2 Configuration de la classe FlatFileItemReader

Avec l'interface LineMapper de Spring Batch, une chaîne est fournie, représentant un enregistrement unique à partir d'un fichier. Avec la chaîne brute du fichier, il existe un processus en deux étapes pour l'obtenir dans l'instance

de l'objet avec lequel vous travaillerez ultérieurement. Ces deux étapes sont gérées par LineTokenizer et FieldSetMapper.

L'implémentation de LineTokenizer analyse la ligne. La chaîne fournie représente la ligne entière du fichier. Pour pouvoir mapper les champs individuels de chaque enregistrement sur votre instance de l'objet, vous devez analyser la ligne dans une collection de champs. Le FieldSet dans Spring Batch représente cette collection de champs pour une seule ligne.

L'implémentation FieldSetMapper mappe le FieldSet à une instance de l'objet. Avec la ligne étant divisée en champs individuels, vous pouvez maintenant mapper chaque champ de saisie sur le champ de votre objet de domaine, comme un RowMapper mapperait une ligne de ResultSet sur une instance de l'objet.

2.5.3 LECTURE EN BASE DE DONNEES

Un autre standard d'entrée de données pour un batch spring Batch est une base de données relationnelles. Dans ce cas précis, les données à importer proviennent des tables de la base de données. Spring Batch permet deux approches différentes en ce qui concernent l'approche de la lecture d'une base de données. La première est l'utilisation de JDBC. La deuxième est faite par ORM (en anglais Object-Relational Mapping) via Hibernate.

2.5.3.1 Utilisation de JDBC

Dans cette approche, Spring Batch laisse la responsabilité de la lecture des données au JDBC. L'Interface *ResultSet* est la représentation objet d'un curseur de la base de données, qui permet de parcourir le résultat des données d'une instruction SELECT. Dans ce cas, le jeu de résultats intègre des mécanismes pour diffuser des données. Avec cette approche, Spring Batch n'exécute qu'une requête et récupère progressivement les données de résultat en utilisant JDBC avec des lots de données. Spring Batch s'appuie sur la configuration et les optimisations JDBC pour fonctionner efficacement. La classe Spring Batch *JdbcCursorItemReader* implémente cette interface. Les attributs requis par la classe sont *dataSource*, *SQL* et *rowMapper*. Ces propriétés servent respectivement à renseigner les informations d'accès à la base de données, la requête SQL servant à exécuter le SELECT afin d'obtenir les données et la classe qui nous permettra de mapper nos données en objets.

2.5.3.2 Utilisation d'un ORM

Le but d'un ORM est généralement de s'occuper de la gestion des incompatibilités entre le modèle de la base de données et modèle orientée objet. Les outils d'ORM gère efficacement les conversions entre les différents modèles et retire aussi la nécessité de spécifier les requêtes SQL en les générant automatiquement à votre place. Comme cela est le cas avec JDBC, Spring fournit un support à l'utilisation d'un ORM.

La lecture avec les curseurs ORM implique que le code responsable de la gestion des classes de domaine n'utilise pas de cache de premier niveau. Seul Hibernate prend en charge cette fonctionnalité via une interface appelée *StatelessSession* qui fournit les mêmes méthodes que la session classique, mais sans la mise en cache et la vérification de l'état modifié. Vous définissez d'abord un mappage de classe de modèle sur une entité de base de données relationnelle.

L'utilisation des curseurs ORM est similaire à JDBC, comme indiqué précédemment. La seule différence est que ORM est une couche supplémentaire au-dessus de JDBC. Pour ORM, Spring Batch prend uniquement en charge Hibernate via la classe *HibernateCursorItemReader*.

2.5.4 IMPLEMENTATION DE LECTEUR PERSONNALISE

Si les implémentations de l'interface ItemReader ne conviennent pas au besoin auquel nous devons faire face, il est possible de proposer notre propre implémentation. Comme par exemple, si l'on souhaite prendre en charge l'ensemble des fichiers dans un dossier. La liste des fichiers dans le dossier est alors chargée quand l'ItemReader est instancié.

En tant que *ItemReader* personnalisé, cette classe implémente l'interface *ItemReader*. Etant donné que l'interface *ItemReader* prend en charge les génériques, vous spécifiez le type associé à la classe (File). Dans le constructeur, vous initialisez la liste des fichiers du répertoire donné. L'implémentation de l'interface *ItemReader* nécessite la définition de la méthode de lecture. Spring Batch appelle cette méthode jusqu'à ce que la méthode retourne « null », indiquant que la méthode a renvoyé tous les fichiers de la liste, un à la fois. Lorsque vous créez un lecteur personnalisé dans Spring Batch, vous implémentez l'interface *ItemReader*. La méthode read effectue tout le traitement de la lecture, qui renvoie les éléments un par un.

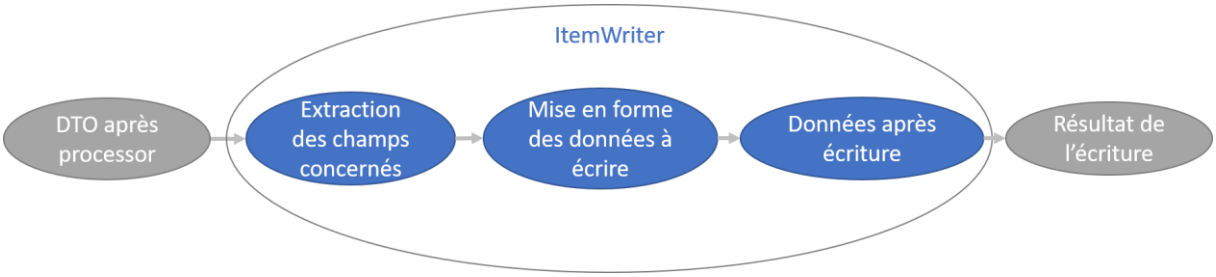
2.6 ÉCRITURE DE DONNEES

Spring Batch fournit diverses implémentations prêtes à l'emploi pour l'écriture dans des fichiers, base de données, flux … Mais il peut être nécessaire de créer nos propres implémentations. Si vous disposez déjà de services hérités produisant ou sauvegardant des données, Spring Batch peut déléguer l'écriture à ces services et les réutiliser.

2.6.1 PRINCIPES

Dans un tasklet, un *ItemReader* lit les données d'entrée, un *ItemProcessor* (éventuellement) les traite et un *ItemWriter* les écrit. Spring Batch fournit la tuyauterie nécessaire pour regrouper la lecture et la transmission des données pour l'écriture. L'interface *ItemWriter* représente le Writer, qui est l'homologue de l'interface *ItemReader*.

L'interface *ItemWriter* définit une seule méthode appelée write, qui enregistre les données de sortie. La plupart des Writer ont la possibilité d'écrire simultanément un ensemble d'éléments. C'est pourquoi il faut prendre une liste d'éléments en tant que paramètre. Après avoir écrit les éléments, le Writer peut vider avant que le processus continue.



Pour le cas de l'écriture dans des fichiers, le Writer vide le flux sous-jacent pour garantir que celui transmet toutes les données fournit au système. Il se passe exactement la même chose dans le cas d'Hibernate, *HibernateItemWriter* vide la session Hibernate après avoir enregistré ou mis à jour des objets à la fin d'un *chunk* par défaut.

Il est de la responsabilité de chaque Writer de s'occuper de la mise en œuvre du vidage de la mémoire le cas échéant. Une fois tous les éléments traités, Spring Batch valide la transaction en cours. Grâce aux Writer de JDBC intégrés, Spring Batch utilise des mises à jour par lot pour envoyer toutes les instructions SQL en une seule opération, afin d'optimiser les performances lors du traitement.

L'attribut commit-interval de l'élément chunk définit la taille de ce dernier. Spring Batch est suffisamment performant pour éviter de charger toutes les données en mémoire. Il traite les éléments en morceaux pour réduire la consommation de mémoire. Comme on le verra plus tard, il est également possible de créer nos propres implémentations d'ItemWriter. Maintenant que les concepts sont clairs, nous allons commencer à voir l'implémentation et la configuration de l'écriture dans des fichiers.

2.6.2 ÉCRITURE DE FICHIERS

Spring Batch fournit des ItemWriter qui écrivent des fichiers dans différents formats : texte délimité, largeurs de champ fixe, et XML.

2.6.2.1 Cas d'écriture de « Fichiers Plats »

Le format de « Fichiers plats » est l'un des premiers formats de fichier utilisés afin de stoker des données. Il s'agit d'un fichier texte contenant des enregistrements. Généralement, ceux-ci sont stocker ligne par ligne. Les champs de données qui composent un enregistrement sont délimités par un séparateur. En règle générale, le séparateur d'un fichier est défini comme étant une virgule ou un caractère de tabulation, pour permettre la délimitation de chaque champ. Cependant, comme on a pu le voir dans le cas de la lecture de fichiers, il existe une variante reposant sur le fait que chaque champ peut avoir une longueur fixe prédéterminée, où les espaces permettent d'appliquer des valeurs de remplissage pour obtenir chaque champ à la longueur souhaitée.

De plus, ce type de fichiers peut inclure un en-tête ou un pied de page facultatifs, permettant de définir des métadonnées supplémentaires ou bien encore d'ajouter des commentaires.

2.6.2.1.1 Notion générale sur FlatFileItemWriter

Nous utilisons donc la classe FlatFileItemWriter pour écrire dans des fichiers de type « Flat Files ». FlatFileItemWriter implémente l'interface ItemWriter et des interfaces complémentaires afin de pouvoir effectuer l'écriture du fichier. L'interface ItemStream est aussi implémentée par la classe FlatFileItemWriter et permet de suivre le cycle de vie du flux de données à écrire. Lorsque Spring Batch appelle la méthode correspondant à l'ouverture du flux, alors FlatFileItemWriter écrit l'en-tête grâce à un FlatFileHeaderCallback. De plus, lorsque la méthode correspondant à la fermeture du flux est appelée alors FlatFileItemWriter écrit le pied de page par l'intermédiaire de FlatFileFooterCallback. Il faut noter que l'utilisation d'un en-tête et/ou d'un pied de page est facultative dans l'écriture des « Flat Files ».

FlatFileItemWriter utilise LineAggregator pour transformer chaque élément en chaîne de caractères. L'implémentation la plus simple de l'interface LineAggregator est la classe PassThroughLineAggregator, qui appelle la méthode toString () sur chaque objet d'élément.

Si des classes spécifiques à un domaine sont utilisées, vous devrez peut-être convertir chaque objet en une chaîne de caractères plus complexe. Dans ce cas, Spring Batch définit une classe abstraite appelée ExtractorLineAggregator, dans laquelle les sous-classes extraient des champs d'un élément donné dans un tableau à l'aide d'un FieldExtractor, puis les agrègent pour le format de fichier donné. Ces interfaces et classes jouent toutes un rôle dans l'écriture de « Flat Files », indépendamment du format de sortie utilisé.

2.6.2.1.2 Configuration d'un FlatFileItemWriter

La classe *FlatFileItemWriter* est le point de départ de l'écriture de fichiers plats. Vous définissez un *FlatFileItemWriter* en tant que Bean dans un contexte XML Spring.

La propriété resource pointe dynamiquement l'URL vers le fichier d'écriture. La classe *FlatFileItemWriter*. Cette classe a aussi une propriété obligatoire nommée *LineAggregator*. Le but de l'interface *LineAggregator* est de créer une chaîne de caractères représentant un Item.

2.6.2.1.3 L'interface LineAggregator et ces différentes implémentations possibles

Spring Batch fournit les implémentations *LineAggregator* suivantes :

- *PassThroughLineAggregator*
- *ExtractorLineAggregator*
- *DelimitedLineAggregator*
- *FormatterLineAggregator*

Dans des cas d'utilisation plus avancés, la classe *FlatFileItemWriter* peut être amener à utiliser un *ExtractorLineAggregator* et un *FieldExtractor* pour formater des données récupérées et écrire le fichier de sortie.

L'interface *LineAggregator* est simple et se concentre sur la conversion d'un objet en une chaîne de caractères. L'implémentation *PassThroughLineAggregator* de l'interface *LineAggregator* appelle la méthode *toString()* sur chaque élément.

L'implémentation *ExtractorLineAggregator* est plus intéressante que la précédente car elle sert de classe de base pour des algorithmes d'agrégation plus élaborés des données en écriture. En effet, la méthode d'agrégation employée permet d'extraire les champs de l'élément donné à l'aide d'un *FieldExtractor*, puis les agrège dans une chaîne de caractères. L'utilité de l'interface *FieldExtractor* consiste à convertir un objet en un tableau de ses parties. Enfin, *FlatFileItemWriter* écrit la chaîne dans un fichier.

La classe *DelimitedLineAggregator* implémente l'interface *LineAggregator* pour créer des fichiers délimités. Un caractère, généralement une virgule ou une tabulation, sépare chaque valeur dans un champ de la valeur suivante. La classe *DelimitedLineAggregator* est chargée de transformer un élément en chaîne de caractères. La liste suivante configure un Bean *DelimitedLineAggregator* dans un contexte Spring. Il faut noter que le délimiteur par défaut est une virgule.

2.6.2.1.4 Ajout et configuration d'un en-tête ou d'un pied de page

Un en-tête est un enregistrement de texte que Spring Batch insère avant le traitement de l'article. Un pied de page est un enregistrement de texte ajouté à la fin du fichier après le traitement de l'élément. Spring Batch fournit les interfaces de méthode unique *FlatFileHeaderCallback* et *FlatFileFooterCallback* pour implémenter ces fonctionnalités.

Spring Batch transmet un enregistreur à la méthode writeHeader pour générer l'en-tête. La classe *FooterCallBack* étend *StepExecutionListenerSupport* pour avoir accès au nombre d'éléments, aux heures de début et de fin à partir de la méthode writeFooter.

Pour configurer votre pied de page et en-tête, des beans de rappel d'en-tête et de pied de page sont à créer. Le point important ici est de configurer le pied de page en tant que ***Listener***, qui implémente l'interface *StepExecutionListenerSupport* afin d'accéder à *StepExecution*. Le pied de page et l'en-tête sont à définir en tant que propriétés de *FlatFileItemWriter*. Par exemple, si l'on souhaite écrire un message dans le fichier de diagnostic du job comme Fin des données.

2.6.2.2 Cas d'écriture de fichiers XML

Spring Batch inclut une implémentation de *ItemWriter* pour les fichiers XML appelée *StaxEventItemWriter*. Ce Writer utilise un marshaller pour écrire des fichiers XML. Un *Marshaller* est une interface générique fournie par le module Spring Object / XML Mapping pour convertir des objets en XML, et inversement. Nous ne développerons pas plus sur l'écriture de fichiers XML.

2.6.2.3 Écriture de lots de fichiers

Spring Batch fournit un mécanisme pour écrire des ensembles de fichiers au lieu d'un seul fichier. C'est utile si vous voulez créer des fichiers avec un maximum d'éléments. La classe *MultiResourceItemWriter* est une implémentation *ItemWriter* pour plusieurs fichiers de sortie.

Par exemple, un flux de données avec 5100 objets à récupérer d'une base de données où d'un fichier et que le commit-interval est fixée à 100 ; Il faut néanmoins faire attention au nombre défini dans le commit-interval pouvant fortement influencer le nombre d'objets retranscrits dans les différents fichiers. De plus il est possible de définir dans le *Bean* du fichier XML, une propriété intitulée *itemCountLimitPerResource*. Si cette propriété est définie à 1000, la classe MultiResourceItemWriter créera alors six fichiers, le premier avec 1000 objets et le dernier avec 100 objets. Par défaut, ce Writer permet de définir un suffixe les noms de fichiers de sortie avec un index.

2.6.3 ÉCRITURE EN BASE DE DONNEES

Comme pour la lecture de données, l'accès à une base de données relationnels est possible à la fois avec JDBC ou un outil ORM. Des Writer sont fournis pour les deux par Spring Batch. Nous verrons dans un premier temps l'écriture en base de données par l'utilisation d'un JDBC ItemWriter.

2.6.3.1 Écriture en utilisant JDBC

Pour implémenter l'interface ItemWriter dans le cas de l'écriture dans une base de données en utilisant JDBC, on utilise la classe JdbcBatchItemWriter qui s'appuie sur Spring JDBC (gérant la plupart des interactions avec la base de données). Cette classe utilise une des propriétés de JDBC pour exécuter tout un groupe de requête SQL pour l'ensemble d'un élément.

La taille du lot de requête SQL est égale à la valeur de commit-interval qui est configuré dans au niveau de la tasklet orienté chunk.

Les principales propriétés de la classe JdbcBatchItemWriter sont sql et la possibilité de choisir entre itemPreparedStatementSetter et itemSqlParameterSourceProvider. La classe ItemPreparedStatementSetter permet d'exécuter les requêtes SLQ avec des marqueurs ? désignant des paramètres donnés dans le fichier de configuration et/ou la classe java. La classe ItemSqlParameterSourceProvider quant à elle, exécute des instructions SQL avec des paramètres nommés. Enfin, la propriété assertUpdates est définie à true tel que si l'exécution d'une requête n'effectue aucune update d'une ligne, alors Spring Batch retourne une EmptyResultDataAccessException.

2.6.4 IMPLEMENTATION D'UN WRITER PERSONNALISE

Dans le cas où aucune des implémentations de ItemWriter ne correspond à ce qui est nécessaire pour réaliser notre Batch, il est alors possible de définir une implémentation d'ItemWriter personnalisé.

Afin de créer des ItemWriter dit « personnalisés », il suffit juste de déclarer dans la classe Java de celui-ci qu'il implémente l'interface ItemWriter. Dans ce cas-là, il est alors possible de définir un ItemWriter répondant à nos besoins. Cependant, il faudra alors faire attention si l'on définit la possibilité au Job dans le cadre de la gestion d'erreur de reprendre à l'endroit où celui-ci a échoué. Cela est géré de base pour les ItemWriter fournit par Spring Batch. Il faudra alors faire attention à ce que notre ItemWriter puisse en faire de même dans ce cas précis.

2.6.5 CHAINAGE D'ITEMWRITER

Lors d'une Step de type chunk, la configuration de Spring permet seulement d'utiliser un seul ItemWriter. Pour éviter cela, lorsque l'on doit effectuer des mises à jour sur des tables différentes ou écrire des données sur plusieurs fichiers, il faut alors utiliser un CompositeItemWriter. Cet ItemWriter permet en effet de chaîner plusieurs ItemWriter.

2.7 LE TRAITEMENT DES DONNEES

2.7.1 OBJECTIFS

Les applications de de type Batch ne sont pas simplement limitées à des entrées ou sorties, elles ont aussi une logique métier définie qui est implémentée. En effet, cette logique métier (spécifications fonctionnelles) sert principalement à appliquer des règles avant d'envoyer les données à une base de données, de transformer les données d'une source en lecture en un format de données attendu à la destination ciblée, …

Dans les applications Spring Batch, on intègre cette phase de logique métier à la phase de traitement à la suite de la lecture de jeu de données, mais aussi avant l'écriture de ces données.

Grâce à l'architecture sous forme de blocs que nous fournit Spring Batch, le Framework fournit un support très pratique dans le cadre du traitement de ce type, l'ItemProcessor. Celui-ci est étant situé entre l'ItemReader et l'ItemWriter. En effet, cet emplacement est idéal pour intégrer la/les règles métiers de l'application et de mélanger ces règles dans les phases de lecture et d'écriture de notre Step.

Quand une Step de type Chunk ne contient aucun traitement de déclarer, les éléments lus lors de la Step sont envoyés tel quel à *l'ItemWriter* et Spring se charge d'agréger les différents éléments. Cependant, il est fortement déconseillé de ne pas effectuer de traitement en premier lieu avant la persistance des données afin de s'assurer de la cohérence de celles-ci. Pour cela, il faut effectuer ces contrôles dans la partie traitement de notre Step.

2.7.2 TRAITEMENT METIERS DES DONNEES

2.7.2.1 Type de traitement métier supporté

La phase de traitement de notre Step est le lieu privilégié pour effectuer les contrôles ou règles dit « métiers ». Il existe deux types de traitement ou règles métiers supportés dans la phase de traitement ; Il s'agit des traitements dits de « Transformation » et des traitements dits de « filtrage ».

Le traitement de type transformation permet d'effectuer les modifications nécessaires sur les données lues par l'ItemReader avant de les envoyer à l'ItemWriter. L'ItemProcessor a la possibilité dans ce cas de changer l'état de l'objet lu, de créer un nouvel objet, ou bien encore de changer le type de données qui sera inséré en base (Par exemple : Chaînes de caractères devenues des entiers, …).

Le traitement de type Filtrage quant à lui, permet de décider d'envoyer ou non chaque donnée lue de l'ItemReader à l'ItemWriter.

2.7.3 CONFIGURATION DE *L'ITEMPROCESSOR*

L'interface ItemProcessor est caractérisée par son utilisation de deux types d'arguments I et O. Avec l'objet de type I qui est transmis à l'ItemProcessor et pour lequel l'élément de type doit être compatible avec l'ItemReader. Et la méthode liée à ce processeur renvoie un objet de type O.

Les type I et O sont définis lors de l'implémentation de l'interface ItemProcessor. Si la méthode de traitement renvoie la valeur null, Spring Batch n'enverra pas celui-ci à l'ItemWriter, comme on le définit dans le contrat de filtrage (le filtrage est différent de la gestion d'erreur).

L'ajout d'un ItemProcessor est simple dans Spring Batch et le fichier de configuration XML, on écrit un Bean qui implémente l'interface ItemProcessor, puis on y fait référence dans la configuration des Steps de type chunk via l'attribut processor.

2.7.4 IMPLEMENTATION DE *L'ITEMPROCESSOR*

Il est assez simple d'implémenter des *ItemProcessor* pour définir les règles métiers de l'application. Néanmoins, Spring Batch fournit plusieurs implémentations de *ItemProcessor* qui peuvent être pratiques à utiliser :

- L'ItemProcessorAdapter qui permet d'utiliser une méthode customisée sur un objet Java pour lequel il n'est pas requis d'implémenter un ItemProcessor.
- Le ValidatingItemProcessor qui permet de déléguer le filtrage logique/métier afin de valider l'objet.
- Le CompositeItemProcessor qui permet de déléguer le traitement des données à une liste ou un ensemble d'ItemProcessor.

2.7.5 MODIFICATION DE DONNEES PAR L'ITEMPROCESSOR

La transformation d'objets lues via un *ItemReader* et leur écriture ensuite via un *ItemWriter* est le cas typique d'utilisation d'un *ItemProcessor*. Spring Batch nous permet de différencier deux types de transformation, le changement de l'état de l'élément lu et la production d'un nouvel objet basé sur l'élément lu. Dans ce dernier cas, l'objet renvoyé par *l'ItemProcessor* peut être d'un type différent de celui de l'objet entrant.

2.7.5.1 Changement d'état d'un objet lu par l'ItemReader

Le client peut dans certains cas avoir besoin d'importer venant d'une tierce personne afin de mettre à jour certaines données de sa base à partir d'un fichier de données fourni. Cependant, ces données ont besoin de subir un traitement dans certains cas afin de traiter les problèmes pouvant être rencontrés. Ces traitements nécessitent de créer des règles et des traitements customisés à son application afin d'adapter celle-ci, dans une méthode dédiée de *l'ItemProcessor*. Par exemple, dans le cas de notre exemple, lors de la lecture d'un fichier de données si un champ a été renseignée par un espace, alors celui-ci sera considérée comme « nulle » et est valorisé à la valeur « 0 ».

Assez souvent un composant de traitement ou règle métier différent légèrement d'une interface Spring Batch comme celle de *l'ItemProcessor* mais comme l'interface n'implémente pas directement l'interface, le Framework ne peut pas l'appeler directement. Cependant Spring nous permet d'utiliser une classe *ItemProcessorAdapter* nous permettant d'éliminer le besoin d'implémenter une interface dédiée. De plus la classe nous permet de connecter la méthode recherchée dans le cadre du traitement en ayant besoin d'effectuer quelques configurations spécifiques dans le fichier de configuration XML.

Par exemple, le *Bean* processor avec la classe *ItemProcessorAdapter* doit contenir la propriété targetObject avec une référence « *ref* » pointant sur le *Bean* sur lequel on pointe afin de récupérer la méthode qui est nécessaire au traitement. Et on récupère la méthode désirée en utilisant la propriété du Bean *targetMethod*.

Cela nous permet de diminuer de manière drastique les erreurs liées aux lancements des batches lors de la récupération du contexte de l'application ainsi que lors de l'exécution du Job.

2.7.5.2 Création de nouveaux Objets à partir de l'objet lu par l'ItemReader

Dans le cadre du traitement de données du client, les fichiers envoyées par les partenaires peuvent être caractérisés par des différences qu'il faut traiter. Par exemple, on a une ligne d'un fichier qui est lue par l'ItemReader et on transforme ce tableau de chaînes de caractères en un Objet Line. Cette transformation de ce tableau de chaînes de caractères en objet Line a lieu dans une classe de Mapper.

Pour cela on va donc utiliser un ItemProcessor personnalisé. *L'ItemProcessor* utilisé est donc spécifique car il implémente un ItemProcessor qui délègue le traitement à une classe de traitement de données dit « métier ».

2.7.5.3 Implémentation de motif de requête conduite avec un ItemProcessor

Il existe enfin une dernière manière de transformer les objets lus utilisés par un modèle fourni dans le cadre des bases de données. Ce modèle se déroule en deux parties :

- Exécution d'une requête pour charger les IDs des objets avec lesquels on veut effectuer des traitements sur ces données. Cette première requête est la requête dite de « routage » et nous fournit N IDs.
- Puis, il y a exécution de plusieurs requêtes que l'on récupère pour chaque ligne de la base de données. Au total, on effectuera N requêtes supplémentaires afin de charger l'objet requis.

Dans le cadre de fichier de données à traiter présentant un nombre conséquent de données brutes, ce modèle de traitement peut être plus performant car celui-ci limite le nombre de requête nécessaires à effectuer afin de traiter mais, il permet aussi de limiter les effets de bords sur les bases de données dans le cadre de résultats de requête large si les tables où s'applique notre Batch et sont les mêmes que celles utilisées par d'autres applications.

2.8 GESTION DES ERREURS

2.8.1 PRINCIPES

Un Job est robuste et fiable se définit par la priorité aux différents scénarios d'erreurs lors du fonctionnement. En effet, lors de l'exécution d'un Job de traitement de données, il est difficile d'observer lorsqu'un problème est apparu lors de l'exécution de ce dernier. Il faut donc prévoir un maximum d'erreurs afin de limiter les effets sur le traitement des données.

2.8.2 DETERMINATION DES SCENARIOS D'ERREURS

Afin de rendre optimale la gestion des erreurs d'un batch, il faut effectuer un recensement de différents éléments tels que :

- La liste définissant les différents Reader, Processor et Writer de nos Jobs.
- La détermination des erreurs pouvant arriver dans chacune de ces parties.
- La méthode de résolution à appliquer dans chaque type d'erreur anticipé.
- La gestion des erreurs dans les classe Java mais aussi au niveau du fichier de configuration XML doit être prévue afin de minimiser le nombre d'erreur non traitée.
- Les listeners pour chaque type d'erreurs et leur méthode de gestion à implémenter afin de connaître l'erreur qui s'est produite

2.8.3 LES DIFFERENTS MODES DE GESTION D'ERREURS

Il existe trois modes de gestion des erreurs dans une application Spring Batch :

- Le mode « skip » permet de gérer des exceptions de type « Fatal » ou de type « métier », et de continuer les traitements en cours pour un élément incorrect générant une erreur ou exception. En général, elle est utilisée dans les Step de type Chunk.
- Le mode « retry » permet de gérer des exceptions de type « Transcient », et de gérer un nouvel essai de traitement. En général, elle utilisée dans les Step de type Chunk et dans le code de l'application dont dépend le Batch (accès à la base de données, …).
- La mode « restart » permet de gérer l'échec de l'exécution d'un Job, et de relancer l'instance d'un Job ayant échouée. En général, elle est utilisée pour l'exécution d'un Job.

Les principaux types d'exception dans une application Spring Batch sont les suivants :

- Exceptions de type « Fatal » : Quand l'exception ne peut pas être géré par l'application et que celle-ci ne peut plus fonctionner.
- Exceptions de type « Transcient » : Par exemple, quand l'enregistrement en base a échoué à la suite d'une interruption de réseau.
- Exceptions de type « Métier » : Une exception remontée à la suite d'un contrôle ou traitement.

3 Développement d'application Spring Batch

3.1 AVANT PROPOS



Langage utilisé

Le français sera utilisé pour nommer les objets, leurs attributs ou leurs méthodes. Le but est d'éviter toute interprétation.

3.2 CONFIGURATION DU PROJET

3.2.1 PILE TECHNIQUE

Pour l'Agence de la biomédecine, la pile technique pour l'application Spring Batch est la suivante :

- Spring Core 5.2
- JPA + Hibernate 5.4
- Spring Batch 4.2



Versions de la pile technique

La version de la pile peut être amenée à évoluée selon les travaux de détermination du socle technique en cours à l'Agence.

3.2.2 DEPENDANCES REQUISES POUR LES APPLICATIONS SPRING BATCH

Les dépendances Spring nécessaires sont les suivantes :

- spring-batch-core,
- spring-batch-infrastructure,
- spring-batch-test,
- spring-core,
- spring-context,
- spring-jdbc,
- spring-beans.

Les dépendances Spring facultatives sont les suivantes :

- spring-orm,
- spring-orx,
- spring-expression,
- spring-aop,
- spring-context-support,
- spring-boot (permet l'utilisation des annotations dans les traitement des batches).

Les dépendances Hibernate sont les suivantes :

- hibernate-core

- hibernate-envers (utile pour les tests Spring batch).

3.2.3 MAVEN

Le pom pour le batch de relance de green est défini ci-dessous :

```
<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>com.abm.batches</groupId>
    <artifactId>batch-grn-rel</artifactId>
    <version>1.00.00</version>
    <name>batch-grn-rel</name>
    <packaging>jar</packaging>
    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <springframework.core.version>5.2.5.RELEASE</springframework.core.version>
        <spring.batch.version>4.2.4.RELEASE</spring.batch.version>
        <hibernate.core.version>5.4.6.Final</hibernate.core.version>
        <log4j.core.version>2.0.2</log4j.core.version>
        <appName>batch-engine</appName>
        <env>natsys-dev</env>
        <appType>jar</appType>
    </properties>
    <build>
     <resources>
            <resource>
                <directory>packaging/${env}</directory>
                <includes>
                    <include>emailContext.xml</include>
                </includes>
                <targetPath>${project.build.directory}/classes/com/abm/batches/config</ta
rgetPath>

                <filtering>false</filtering>
            </resource>
            <resource>
                <directory>src/main/resources</directory>
                <filtering>true</filtering>
            </resource>
        </resources>
        <pluginManagement>
        <plugins>

            <!-- overrides the clean configuration from parent -->
            <plugin>
                <artifactId>maven-clean-plugin</artifactId>
                <version>2.5</version>
                <executions>
                    <execution>
                        <id>default-clean</id>
                        <phase>clean</phase>
                        <goals>
                            <goal>clean</goal>
                        </goals>
```

```
                </execution>
            </executions>
        </plugin>

        <!-- Maven Shade Plugin -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
            <version>2.4.3</version>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>shade</goal>
                    </goals>
                    <configuration>
                        <transformers>
                            <transformer
                                implementation="com.github.edwgiz.mavenShadePlugin.log4j2CacheTransformer.PluginsCacheFileTransformer">
                            </transformer>
                            <transformer
                                implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                                <mainClass>com.abm.batches.laneur.MainBatches</mainClass>
                            </transformer>
                            <transformer
                                implementation="org.apache.maven.plugins.shade.resource.AppendingTransformer">
                                <resource>META-INF/spring.handlers</resource>
                            </transformer>
                            <transformer
                                implementation="org.apache.maven.plugins.shade.resource.AppendingTransformer">
                                <resource>META-INF/spring.schemas</resource>
                            </transformer>
                        </transformers>
                    </configuration>
                </execution>
            </executions>
            <dependencies>
                <dependency>
                    <groupId>com.github.edwgiz</groupId>
                    <artifactId>maven-shade-plugin.log4j2-cachefile-transformer</artifactId>
                    <version>2.8.1</version>
                </dependency>
            </dependencies>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <executions>
                <!-- never skip test compilation -->
                <execution>
                    <id>default-testCompile</id>
                    <goals>
                        <goal>testCompile</goal>
                    </goals>
```

```
                    <configuration>
                        <skip>false</skip>
                    </configuration>
                </execution>

            </executions>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-source-plugin</artifactId>
            <version>3.2.1</version>
            <executions>
                <execution>
                    <id>attach-sources</id>
                    <goals>
                        <goal>jar</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-javadoc-plugin</artifactId>
            <version>3.2.0</version>
            <configuration>
                <javadocExecutable>${java.home}/bin/javadoc</javadocExecutable>
            </configuration>
            <executions>
                <execution>
                    <id>attach-javadoc</id>
                    <goals>
                        <goal>jar</goal>
                    </goals>
                    <configuration>
                        <additionalJOption>-Xdoclint:none</additionalJOption>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
    </pluginManagement>
</build>
    <dependencies>
        <dependency>
            <groupId>javax.xml</groupId>
            <artifactId>jaxrpc</artifactId>
            <version>1.1</version>
        </dependency>
        <!-- Spring Batch unit test -->
        <dependency>
            <groupId>org.springframework.batch</groupId>
            <artifactId>spring-batch-test</artifactId>
            <version>${spring.batch.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
```

```
        </dependency>
        <dependency>
            <groupId>org.springframework.batch</groupId>
            <artifactId>spring-batch-core</artifactId>
            <version>${spring.batch.version}</version>
            <scope>compile</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-batch</artifactId>
            <version>1.5.3.RELEASE</version>
            <scope>compile</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-expression</artifactId>
            <version>${springframework.core.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>${springframework.core.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aop</artifactId>
            <version>${springframework.core.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>${springframework.core.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-beans</artifactId>
            <version>${springframework.core.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${springframework.core.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-oxm</artifactId>
            <version>${springframework.core.version}</version>
            <scope>compile</scope>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-core</artifactId>
            <version>${hibernate.core.version}</version>
            <scope>compile</scope>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-envers</artifactId>
            <version>${hibernate.core.version}</version>
            <scope>compile</scope>
```

```
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-orm</artifactId>
            <version>${springframework.core.version}</version>
            <scope>compile</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context-support</artifactId>
            <version>${springframework.core.version}</version>
        </dependency>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjrt</artifactId>
            <version>1.8.11</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
            <version>1.8.11</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>com.oracle</groupId>
            <artifactId>ojdbc8</artifactId>
            <version>12.2.0.1.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>javax.mail</groupId>
            <artifactId>mail</artifactId>
            <version>1.4.7</version>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>3.1.0</version>
        </dependency>
        <dependency>
            <groupId>efg.co.services</groupId>
            <artifactId>co-logger</artifactId>
            <version>1.0.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-slf4j-impl</artifactId>
            <version>${log4j.core.version}</version>
        </dependency>
      <!-- https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-core -->
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-core</artifactId>
            <version>${log4j.core.version}</version>
        </dependency>
       <!-- https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-jcl -->
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-jcl</artifactId>
```

```
            <version>${log4j.core.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-web</artifactId>
            <version>${log4j.core.version}</version>
        </dependency>
        <dependency>
            <groupId>javax.inject</groupId>
            <artifactId>javax.inject</artifactId>
            <version>1</version>
        </dependency>
        <dependency>
            <groupId>org.jasypt</groupId>
            <artifactId>jasypt</artifactId>
            <version>1.9.2</version>
        </dependency>
    </dependencies>
</project>
```

3.2.4 GESTION DES PROJETS

La gestion des versions est effectuée via SVN.

- Un projet est créé pour batch : son nom reflète le domaine fonctionnel qu'il couvre, comme **« batch-green-relance »** par exemple. Le build de ce projet génère un fichier JAR.

3.2.5 CONFIGURATION DE L'APPLICATION

La connexion à la base de données doit être déclarée comme des propriétés du batch qui seront récupéré dans un fichier env. properties. Si plusieurs batches sont déployées au sein d'un même serveur, un fichier env. properties unique doit être déclaré pour chaque batch.

Pour les logs :

- Il faut prévoir une rotation par date et par taille (idéalement une taille inférieure à 20 mo) avec un maximum de 10 fichiers.
- Le répertoire où se trouve ce fichier est configuré dans le fichier de properties de l'application.



Le logger sera configuré en production pour afficher les niveaux INFO et supérieurs.

3.2.6 VARIABILISATION DES DEVELOPPEMENTS POUR LA GESTION DES ENVIRONNEMENTS

Il est important de ne pas mettre les valeurs des variables dans le code mais de les variabiliser dans des fichiers annexe afin de rendre le code plus lisible.

Il faut utiliser une stratégie de variables dans le properties :

- Utilisation des fichiers de properties, qui seront chargés au démarrage du serveur d'application (voir fichier de configuration XML),

3.3 STRUCTURE DE L'APPLICATION

3.3.1 LES DIFFERENTS COMPOSANTS

Le batch est composée d'un projet Java qui produit un jar.

3.3.2 LES PACKAGES

Ils suivent le regroupement par fonctionnalité et par couche de composants.



Le nom des packages doivent être :

- **En minuscules**
- **Au singulier**

```
com.abm.batches.nomdomaineapplicatif.data.mondomainefonctionnel.*
```

Pour les fonctionnalités génériques :

```
com.abm.batches.nomdomaineapplicatif.generic.mondomainefonctionnel.
```

Le paramètre de package nomdomaineapplicatif correspond au domaine applicatif dont lequel dépend le batch. Par exemple, pour le batch de relance de mail aux équipes de Green, le nom de domaine applicatif sera green.

3.3.3 LES CLASSES JAVA

Les noms des classes et interfaces java doivent respecter les conventions suivantes :

- Le nom des classes, méthode et variables doivent être en français pour le domaine fonctionnel afin d'éviter des écarts de compréhension avec le métier
- Les noms des classes doivent utiliser la notation « UpperCamelCase » (NomDeClasse).
- Les noms des classes qui sont des itemReaders doivent être suffixés par « ItemReader ». Exemple : UtilisateurItemReader.java
- Les noms des classes qui sont des itemProcessors doivent être suffixés par « ItemProcessor ». Exemple : UtilisateurItemProcessor.java
- Les noms des classes qui sont des itemWriters doivent être suffixés par « ItemWriter ». Exemple : UtilisateurItemWriter.java
- Les noms des classes qui sont des DataTransferObject doivent être suffixés par « DTO ». Exemple : UtilisateurDTO.java
- Les noms des classes qui sont des data services doivent être suffixés par « DataService ». Exemple : UtilisateurDataService.java
- Les noms des interfaces qui héritent de JPARepository doivent être suffixés par « Repository ». Exemple : UtilisateurRepository.java
- Les noms des classes qui sont utilisées comme EmbeddedId par les entités JPA pour définir les clés composites doivent être suffixés par « PK ». Exemple : UtilisateurPK.java
- Les noms des classes qui implémentent des interfaces doivent être suffixés par « Impl ». Exemple : UtilisateurServiceImpl.java

- Les noms des classes qui sont des Enums doivent être suffixés par « Enum ». Exemple : CodeEnum.java
- Les noms des classes qui sont des exceptions doivent être suffixés par « Exception ». Exemple : UserNotFoundException.java
- Les noms des classes utilitaires doivent être suffixés par « Utils ». Exemple : DateUtils.java



Les noms des méthodes et propriétés doivent utiliser la notation « lowerCamelCase »

On commence par une minuscule, ensuite chaque nouveau mot constituant le nom commence par une majuscule : nomDeMethode.



Les noms des constantes doivent être en majuscule et les mots séparés par des underscores (SREAMING_SNAKE_CASE)

3.4 MISE EN PLACE DU LANCEUR DES BATCHES

3.4.1 EXECUTION DES BATCHES

Une application Spring Batch doit pouvoir être exécuté de plusieurs moyens différents, la plupart du temps, ces derniers sont exécutés automatiquement à heure fixe. Le lancement est effectué par un script Shell. Ce script shell doit être fournit dans les ressources du livrable. Voici le cas du lanceur du batch EDI 065 :

```
#!/usr/bin/sh

JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.171-2.6.13.0.el7_4.x86_64
echo $JAVA_HOME
$JAVA_HOME/jre/bin/java -Dkey=Zjc4NXNkMWdzZA== -Dlog4j.configurationFile=batch-edi065-
log4j2.xml -DenvProperties=application-batches.properties -jar batch-edi065.jar --
job=T2C065 --filename=$1 --codeLabo=$2
```

Il faut faire attention à plusieurs points ici :

- `JAVA_HOME` correspondant à la localisation de la JVM qui sera utilisé par le batch. Il faut donc bien définir la version de la JDK qui sera utilisé pour faire fonctionner le batch ;
- `-Dlog4j.configurationFile=batch-edi065-log4j2.xml` correspondant à la déclaration de la configuration des logs du batch.
- `-DenvProperties=application-batches.properties` correspondant à la déclaration du fichier de propriétés du batch.
- `--job=T2C065` correspondant à l'identifiant du job.

L'exécution des batches s'effectue en lançant un ou des jobs décomposés en plusieurs étapes, dont on doit contrôler l'enchaînement.

Le scénario d'exécution le plus simple est l'exécution séquentielle, dans laquelle les étapes s'exécutent les unes à la suite des autres. L'attribut `next` de la balise `step` permet de préciser l'étape suivante. Spring Batch propose aussi une exécution conditionnelle, afin de choisir l'étape suivante selon le statut de sortie de l'étape courante. Cette fonctionnalité permet de composer des enchaînements complexes d'étapes, pour, par exemple, remédier au

problème d'arrêt brutal du job en cas d'erreur dans une étape. Elle se combine particulièrement bien avec l'utilisation d'écouteurs d'étape, capable de changer le statut de sortie selon certaines conditions. Dans le batch EDI 065, on a le cas suivant :

```
    <!-- Description du job T2C065 -->
    <job id="T2C065" xmlns="http://www.springframework.org/schema/batch">
        <listeners>
            <listener ref="IDR.JobListener" />
        </listeners>

        <step id="IDR.step0InitJobContext">
            <tasklet ref="IDR.InitJobContextTasklet" />
            <next on="*" to="IDR.step1ProcessFileLabo" />
            <next on="FAILED" to="IDR.step2Log" />
        </step>
        <step id="IDR.step1ProcessFileLabo">
            <tasklet>
                <chunk reader="IDR.fileLaboReader" processor="IDR.lineCompositeProcessor"
                    writer="IDR.lineDbWriter" commit-interval="1" />
            </tasklet>
            <next on="*" to="IDR.step2Log" />
            <next on="FAILED" to="IDR.step2Log" />
            <listeners>
                <listener ref="IDR.lineStructureProcessor" />
                <listener ref="IDR.lineBusinessProcessor" />
            </listeners>
        </step>
        <step id="IDR.step2Log">
            <tasklet>
                <chunk
reader="IDR.logReader"                                                    process
or="IDR.logProcessor"
                    writer="IDR.logWriter" commit-interval="1" />
            </tasklet>
        </step>
    </job>
```

3.4.2 Planification et ordonnancement des Batches

La planification des batches pour une application Spring Batch peut être supporté dans la configuration du batch lui-même. Cependant, cette dernière se ferait en marge de l'environnement actuellement en place. De plus, cela permet de planifier de manière globale sur les environnements techniques l'ordonnancement de ces derniers.

La planification doit donc être paramétrée au niveau de la crontab Unix du serveur de l'Agence de Biomédecine.

3.4.3 Historisation du lancement des Batches

Spring Batch permet l'historisation des batch grâce à sa notion de JobRepository. L'historisation a deux utilités principales. Elle permet, d'une part, le monitoring des batch, particulièrement important pour les équipes de production, et, d'autre part, la reprise des batch, principalement quand une erreur est survenue.

L'historisation ne gère pas seulement les dates de lancement d'un job, mais aussi le contexte complet d'une exécution. Ce contexte peut être reconstitué pour l'exécution suivante du job, dans le cas où il aurait échoué précédemment. Il est dès lors possible de reprendre le batch exactement là où une erreur est survenue.

La configuration de l'historisation est possible grâce à l'unique implémentation qui est proposée par Spring Batch. Cette implémentation de JobRepository délègue la persistance des différentes entités d'une entité de batch à des DAO.

Le cas le plus courant lors du développement, les tests ou bien encore quand l'historisation des batches n'est pas nécessaire se fait pour les DAO via le stockage en mémoire des données.

Néanmoins, il reste possible de persister de configurer cette historique des batches par l'intermédiaire d'un mécanisme de persistance en base de données. Il faut donc dans un premier définir une datasource possiblement spécifique pour l'historisation des batches dans le contexte de ce dernier. Il faut aussi noter que comme le batch et lancé en ligne de commande par le script Shell, l'exécution des étapes de nos traitements est donc séquentielle et n'a lieu que dans le cadre de notre fil d'exécution. Cela implique qu'il n'est pas nécessaire de définir de pool de connexions pouvant créer des connexions inutiles.

Il faut ensuite définir le gestionnaire de transactions de Spring, ce dernier nécessite d'effectuer la référence à notre dataSource. La définition du *JobRepository* vient compléter la configuration. Cette balise de notre fichier xml nécessite dans ce cas-là, une référence à notre dataSource et au gestionnaire de transactions.

Le script SQL pour générer les différentes tables utilisées par Spring Batch est intégré à la racine de l'archive Java de la librairie Spring Batch Core. Suivant la base de données, Spring Batch fournit un script SQL différent.

Les données contenues dans l'historisation des batches est la suivante :

- Les instances de Job ;
- Les paramètres d'instances de Job ;
- Les exécutions des instances de Job ;
- Le contexte dans lequel s'est déroulé chacune des exécutions de Job ;
- Les différentes étapes des exécutions des Jobs ;
- Le contexte dans lequel s'est déroulé chacune des étapes.

Pour chaque exécution d'un job, Spring Batch persiste l'ensemble de ces éléments de façon transparente. Nous avons déjà vu que les paramètres d'un job définissent l'identité de l'instance de ce job. En lançant un batch en ligne de commande, les paramètres de l'instance peuvent être précisés à la fin de la ligne. L'ordonnanceur chargé de lancer les batches gère généralement l'aspect dynamique de ces paramètres.

3.5 MISE EN PAGE ET DOCUMENTATION

3.5.1 MISE EN PAGE DU CODE

Les développeurs doivent s'assurer que le code source est formaté conformément au fichier de formatage de l'IDE, avant de commiter.

Le fichier de formatage de l'IDE doit contenir les règles suivantes :

- 120 caractères par ligne
- Espaces pour indenter, plutôt qu'une tabulation
- Style d'indentation : K&R, variante 1TBS

3.5.2 DOCUMENTATION ET JAVADOC

Toutes classes et méthodes publiques doit disposer d'une documentation au format Javadoc.

3.6 GESTION DES ERREURS

3.6.1 OBJECTIF

Une application doit faire face à différents types d'événements qui interrompent son fonctionnement standard :

- Les actions utilisateurs incorrectes comme le format de fichier incorrect, la saisie d'une valeur incorrecte dans le fichier ou la non saisie d'une valeur obligatoire dans ce dernier,
- Les manques de ressource du système (OutOfMemoryError…) ou une configuration incorrecte
- Les erreurs de programmation comme un NullPointerException…

Le programme doit :

- Être capable d'informer correctement l'utilisateur sans dévoiler des informations techniques qui pourrait aiguiller les attaques d'un hacker,
- Permettre à l'exploitation et au développeur de corriger l'erreur s'il s'agit d'un problème de ressource, configuration ou de développement.

3.6.2 Les Exceptions

Le mécanisme mis à disposition par Java est l'exception.

Lorsque le système rencontre une situation anormale qui nécessite l'interruption du traitement en cours, il lance une exception. Cette exception va remonter la pile des appels en les interrompant jusqu'à être « catchée ».

Si personne ne « catch » l'exception celle-ci est affichée par le système en interrompant l'application.

Le mécanisme d'exception permet :

- L'interruption du traitement,
- Le nommage de l'exception,
- Et l'affichage de sa pile.

Dans l'objectif de faciliter l'identification de l'origine du problème.

En conséquence,

- Si on lance une exception, il faut utiliser un nom d'exception qui soit significatif.
- Il ne faut surtout pas « avaler » une exception : catcher une exception sans renvoyer d'exception et sans tracer sa pile d'appel.

```
// Code à proscrire
try {

} catch (final NumberFormatException he) {
                // rien : cela ne devrait jamais arriver
}
```

A minima, il faut tracer l'exception :

```
// Code à préférer mais déconseillé
try {

} catch (final NumberFormatException he) {
                log.error("cela ne devrait jamais arriver",he);
}
```

Dans la pratique, il est déconseillé de « catcher » des runtime exception et encore plus `Exception` et, en aucun cas `Throwable` (`Throwable` est l'interface mère que partagent `Exception` et `Error`, « catcher » une Error peut potentiellement déstabiliser l'ensemble du serveur d'application)

Une runtime exception doit pouvoir être évitée par un code de vérification en amont comme la vérification qu'un objet n'est pas nul avant l'exécution de l'appel d'une méthode.

Les seules runtime exceptions qu'il est possible de catcher sont celle qui concernent les `IOException` : il n'est pas possible de savoir si un fichier est présent, lisible ou modifiable avant d'y accéder.

« `Catcher` » Throwable est formellement déconseillé, cela correspond à catcher des exceptions de la JVM comme le OutOfMemoryError pour laquelle rien ne peut être fait.

```
// Code à proscrire
try {

} catch (final Throwable he) {
                log.error("cela ne devrait jamais arriver",he);
}
```

Nous recommandons la création d'une exception mère pour les exceptions applicatives : on l'appellera `BusinessLogicException`. Elle héritera de la classe `Exception`.

L'exception doit aussi permettre de véhiculer les raisons de l'erreur vers l'appelant ou l'utilisateur. Il faut donc attacher une certaine importance aux propriétés de l'exception.

On recommandera de définir les propriétés suivantes pour la `BusinessLogicException` :

- errorCode : le code erreur, un code technique permettant d'identifier la cause de l'erreur
- origin : doit permettre de retrouver dans le code, l'erreur dans le code
- detail : une explication à destination de l'utilisateur.

Nous vous conseillons également de définir des exceptions plus précises qui héritent de la BusinessLogicException comme :

- ForbiddenException : si l'utilisateur n'est pas autorisé
- NotAllowedException : si l'utilisateur n'est pas authentifié
- NotFoundException : par exemple si on ne trouve pas un objet

Cette liste n'est pas exhaustive et peut être complétée : plus le nom de l'exception est précis plus le travail d'analyse est simplifié.

Si vous catchez une exception pour en envoyer une nouvelle, il faut faire en sorte que la pile de la première exception ne soit tracée qu'une et une seule fois.

On ne trace pas pour renvoyer :

```
// Code à proscrire : on risque de voir apparaître deux fois la pile de l'erreur
try {

} catch (final NumberFormatException he) {
                log.error("message",he);
                throw he;
}
```

On peut intercepter pour lancer sa propre exception mais dans ce cas, il faut passer l'exception interceptée en cause pour que le destinataire dispose de la pile complète.

```java
// Code recommandé si on lance sa propre exception
try {

} catch (final NumberFormatException he) {
                throw new BusinessLogicException("mon message",he);
}

// Code à proscrire : on perd la pile de la première exception
try {

} catch (final NumberFormatException he) {
                throw new BusinessLogicException("mon message");
}
```

3.6.3 Batches et Exception

Comme vu précédemment, les principaux types d'exception dans une application Spring Batch sont les suivants :

- Exceptions de type « Fatal » : Quand l'exception ne peut pas être géré par l'application et que celle-ci ne peut plus fonctionner.
- Exceptions de type « Transcient » : Par exemple, quand l'enregistrement en base a échoué à la suite d'une interruption de réseau.
- Exceptions de type « Métier » : Une exception remontée à la suite d'un contrôle ou traitement.

Suivant le type d'exception généré, il est possible de définir différents cas de gestion de nos erreurs :

- Le redémarrage sur erreur quand notre batch a dû s'arrêter et que l'on doit recommencer le traitement à la suite de cette erreur ;
- Le saut sur erreur (*skip*) consiste à sauter l'élément nous posant un problème et passer au suivant. Il est aussi possible de mettre une limite au nombre d'éléments qui peuvent être passé en positionnant l'attribut *skip-limit* ;
- Le dernier cas consiste à effectuer une nouvelle tentative sur erreur (*retry*).

Le cas de gestion à utiliser doit dépendre de la criticité du traitement et de l'importance des données qui seront traitées.

3.7 GESTION DES LOGS

3.7.1 Objectif du log

Le log permet de tracer les actions du système d'information lors de son exécution : les solutions de log Java permettent de garder une trace du fonctionnement en production en enregistrement cette trace dans des fichiers.

L'exploitation peut utiliser ces informations pour surveiller le bon fonctionnement de l'application.

Le log est particulièrement utile dans les phases d'installation et configuration d'une application : les informations qu'elle affiche doivent permettre de résoudre les problèmes de configuration.

Le log permet également aux équipes de développement d'analyser les disfonctionnements détectés en production. Pour ce point, il est important que la pile d'une erreur soit présente dans le log mais que l'on est une idée des actions précédentes de l'utilisateur.

3.7.2 Logger Java

Les loggers Java permettent sur le principe de log4j2, de définir des niveaux de log que l'on pourra activer par configuration externe de la log.

Ils permettent d'avoir :

- Un log horodaté,
- D'identifier le thread qui a émis le log : utile en production pour rattacher les différentes traces à un même utilisateur
- De définir un logger par classe : cette information peut être présente dans le fichier de log, ce qui permet de savoir qui a émis le log
- Un mécanisme d'affichage d'une pile d'erreur.

Tous ces éléments vont permettre à partir du fichier produit de retrouver le disfonctionnement et de reconstituer ce qui s'est passé.

Pour que cela soit possible, il ne faut pas casser le mécanisme en utilisant des mécanismes non gérer par le logger pour tracer des informations.



Ne jamais utiliser e.printStackTrace() pour tracer une exception
Ce mécanisme n'est pas géré par le logger, et l'erreur peut ne pas être retrouvée dans le fichier de log.



Utiliser logger.error(« message », e) pour tracer l'erreur et la pile d'erreur.



Ne jamais utiliser System.out.println() même pour debugguer
Le logger dispose d'une méthode pour tracer en mode debug :
logger.debug(« message »). Vous n'aurez pas besoin de supprimer la ligne avant de livrer et vous ne risquez pas de l'oublier.



Utiliser un logger par classe

```
private static final Logger logger =
LoggerFactory.getLogger(this.getClass());
```

En définissant un logger par classe, on facilite l'identification de la ligne émettrice du log.



Ne pas faire de traitement conditionnel en fonction du niveau de log

Il faut à tout prix éviter de faire des actions uniquement sur condition du niveau de log. Cela crée de la complexité cyclomatique et rend le comportement de l'application différent en fonction du niveau de trace (heisenbug).



Utiliser des lambdas pour tracer les niveaux « debug » et « trace », ou dans le cas de traces structures complexes pour les niveaux moins fins.

La trace d'informations peut être coûteuse en temps. Malheureusement, l'évaluation de ce qui est à tracer se fait dans tous les cas, même si le niveau de trace ne permet pas la sortie du message, on fait donc du traitement pour rien.

Depuis Java 8, il est possible de passer des callbacks aux services de log, ce qui permet de différer l'évaluation du contenu à tracer, au moment où le framework de trace sait s'il doit laisser sortir le message ou non.

L'appel se fait de cette manière :

LOGGER.debug(() -> « ma trace avec cet objet coûteux à évaluer : {} », obj) ;

3.7.3 DEFINITION DES NIVEAUX DE LOG

Les loggers Java permettent la définition de 6 niveaux hiérarchisés de log :

- Trace : le niveau le plus bas
- Debug : niveau utilisé pour le debugging.
- Info
- Warn
- Error
- Fatal

La configuration du logger permet de définir à partir de quel niveau on trace l'information dans le fichier, réduisant le coût des logs de niveau inférieur. Si on définit qu'en production on trace à partir du niveau info, le coût des logs de niveau Debug ou Trace sera négligeable et peut rester dans le code si on respecte les recommandations précédentes.

Le choix d'un niveau de log dépend principalement du choix de leur présence en production ou non. Pour celle présente en production, cela dépend également du mode de travail de l'équipe d'exploitation et des conventions définies entre le développement et l'exploitation.

De manière générale, les équipes d'exploitation sont toutes habituées à suivre les ERROR et les FATAL : elles considèrent que ce qui est en error ou fatal nécessite une action de leur part. On évitera l'utilisation de ces niveaux de log pour tracer une erreur de l'utilisateur comme une saisie invalide.

Pour les WARN cela dépend des équipes d'exploitation. Certaines sont attentives à son occurrence d'autre non. Dans le cadre de l'ABM, il est convenu que ce niveau sera utilisé pour tracer les erreurs utilisateurs.

Si nous nous limitons en production à ces 3 niveaux (WARN, ERROR, FATAL), nous n'aurons qu'une trace de la survenue d'un incident. Il ne sera pas possible de le recontextualiser dans un flux d'actions. La possibilité de retrouver dans quel contexte, quelle série d'actions à amener l'incident, améliore grandement le suivi de l'exploitation.

Le trafic des applications de gestion comme celles de l'Agence et l'efficacité du système de log Java permet d'envisager de tracer de 1 à 3 lignes par actions utilisateur en production.

Nous conseillons de positionner le seuil de log pour la production au niveau INFO et d'utiliser ce niveau pour tracer au moins une ligne INFO par action utilisateur.

La ligne tracée en INFO doit être si possible, une synthèse du traitement réalisé par le service REST concerné : idéalement s'il s'agit d'une création ou mise à jour en base, il faudrait tracer le login de l'utilisateur et l'identifiant de l'objet créé ou mis à jour.

On n'utilisera pas le niveau INFO pour tracer l'entrée dans le service REST (à ce niveau, il est rarement possible de dire si tout s'est bien passé). Si on veut garder une trace de l'entrée dans un service REST, nous conseillons le niveau DEBUG.

Le niveau DEBUG est généralement utilisé comme son nom l'indique pour le DEBUG.

En synthèse, on peut retenir les règles suivantes :



On utilisera les 4 niveaux de log suivant : DEBUG, INFO, WARN, ERROR

DEBUG sera utilisé pour le debug ou garder une trace d'entrée dans un service.

INFO sera utilisé pour garder une trace de succès des actions utilisateur importantes. On essaiera de tracer le login de l'utilisateur et l'identifiant de l'objet impacté.

WARN sera utilisé pour tracer les erreurs utilisateurs qu'il doit être en capacité de résoudre lui-même.

ERROR sera utilisé pour toutes les erreurs techniques



Le logger sera configuré en production pour afficher les niveaux INFO et supérieurs.



Un service REST devra tracer en cas de succès, au minimum une ligne en INFO. Cette ligne devra être une synthèse de l'opération réalisée.

La ligne pourra contenir le login de l'utilisateur et l'identifiant de l'objet impacté

3.7.4 MESSAGE DE LOG

L'analyse d'un log doit permettre de retrouver le cheminent du code : c'est pour cette raison qu'il est utile d'avoir dans certain cas, une trace des traitements réussis en amont.

Toutefois, pour être en mesure de reconstituer ce cheminement, il est impératif de pouvoir lier la ligne de log avec une ligne de code.

Pour que cela soit simple et facile, nous recommandons que chaque ligne de log commence par le nom de la méthode qui contient l'instruction logger. Cette règle doit être appliquée dans tous les cas du DEBUG à ERROR.



Une ligne de log doit toujours commencer par le nom de la méthode qui l'a émise.

Cette règle doit être respectée y compris pour le DEBUG et ERROR.

Un message de log doit être unique : l'utilisation du nom de la méthode aide à l'unicité, mais il faut éviter les messages qui n'affichent qu'une valeur, il faut au moins préciser la variable concernée.

Les messages de niveau INFO, WARN et ERROR sont destinés à une exploitation en production, ils doivent être clairs et significatifs en indiquant l'objet principale (par son identifiant par exemple) et son contexte en précisant par exemple la valeur erronée.

Pour les messages en INFO, des valeurs significatives pour le métier peuvent être ajoutées à l'identifiant. Attention à toutefois ne pas divulguer d'informations sensibles dans le log.

Pensez qu'une des façons principales d'utilisation du log est la recherche d'une chaine de caractères : essayez de faciliter cette opération.



Nous vous déconseillons l'utilisation des mots ERROR et FATAL dans vos messages
Ces mots peuvent être utilisés par l'équipe d'exploitation pour détecter les éventuels disfonctionnements.

Le message de log sera structuré d'une telle manière qu'il sera facile à lire et à retrouver :

- Le message commencera toujours par le nom de la méthode
- Une valeur sera toujours préfixée par le nom de la variable la contenant ou du nom du libellé dans l'écran : il est plus facile et pertinent de rechercher « Donneur:200900 » que « 200900 ». La seconde peut ramener également un Utilisateur 200900, un patient 200900…



Toujours préfixer une valeur par son nom.

Cela permet de retrouver plus facilement les lignes où l'objet a été manipulé et de reconstituer son évolution.

Il est important pour le suivi d'exploitation de garder une trace des cas les plus imprévisibles. En conséquence, dans tous les blocs catch, il est nécessaire a minima de tracer l'erreur détectée.



Tout bloc catch d'une RuntimeException doit tracer une ERROR.

Les seuls blocs catch qui ne sont pas soumis à cette règle sont ceux qui intercepte une exception fonctionnelle héritant de BusinessLogicException.

```
Attention ! si on fait dans le catch un throw, il ne faut pas
logger la pile, uniquement le message.
```

Pour la reconstitution du cheminent d'un traitement, il est important de conserver une trace du cas fonctionnel dans lequel se range une action particulièrement si une partie du traitement est spécifique (par exemple si le patient est mineur). Dans ce cas, il faudra une ligne de log de niveau INFO précisant le passage dans la branche concernée.

Si en debug, vous souhaitez tracer un ensemble d'informations couteuses, il existe un test « if(logger.isDebugEnabled()) » avant le bloc contenant logger.debug().



L'utilisation du logger.isDebugEnabled est à proscrire
Ce type de bloc conditionnel en fonction du niveau de log est dangereux : il peut masquer en développement des comportements erronés en production : souvent si le bloc est couteux c'est qu'il modifie l'état du système (initialise une valeur, charge un enregistrement…).

3.8 DEFINITION D'UN ITEMREADER

3.8.1 L'ITEMREADER

L'**ItemReader** va stocker la logique de récupération de données. L'ItemReader est une interface Spring de Spring Batch. Cette dernière est caractérisée par une unique méthode read () permettant de récupérer les données et ensuite les mapper via un DataObject (DTO)

3.8.2 LES METHODES D'UN ITEMREADER

L'ItemReader fournit par Spring Batch permet de ne pas avoir à écrire une classe ItemReader spécifique dans les cas simples. Cependant, dans certains cas particuliers, il est possible d'utiliser le CustomItemReader, afin de répondre à un besoin spécifique.

Néanmoins, l'ItemReader sera toujours constitué de la méthode *read ()*.

3.9 DEFINITION D'UN ITEMPROCESSOR

3.9.1 L'ITEMPROCESSOR

L'**ItemProcessor** va contenir la logique du traitement avec comme données en entrée ceux de l'ItemReader.

3.9.2 LES METHODES D'UN ITEMPROCESSOR

L'ItemProcessor est caractérisé par la méthode *process ()* permettant d'effectuer les traitements et contrôles métiers sur les données provenant de l'ItemReader. Il est possible de chaîner plusieurs traitements à un ItemProcessor en utilisant un CustomItemProcessor.

3.10 DEFINITION D'UN ITEMWRITER

3.10.1 L'ITEMWRITER

L'ItemWriter se charge d'écrire les données en sortie de l'ItemProcessor.

3.10.2 LES METHODES D'UN ITEMWRITER

L'ItemWriter est caractérisé par la méthode *write ()* permettant d'effectuer la logique d'écriture de l'ItemWriter.

3.11 DEFINITION D'UN DATASERVICE

3.11.1 LE DATASERVICE

Le DataService est une classe annotée par @Service. Cela permet d'indiquer qu'il s'agit d'une classe qui a la charge de coder les règles métier.

Il est aussi possible de ne pas utiliser les annotation sur la classe Java et de créer un DataService abstrait dont hériterait les autres dataservices.

3.11.2 METHODE ANNOTEE @TRANSACTIONAL ET ACCES BASE

Le DataService est une classe qui utilise les interfaces Repository pour l'implémentation des règles de gestion métier.

La première méthode du DataService appelée par le controller doit ouvrir la transaction base afin que tous les accès base réalisés par la suite soit dans la même session hibernate. Pour cela, il faut annoter la méthode avec `@Transactional.`

Il est important que tous les échanges avec la base soient dans une même et unique session hibernate, ce qui est assuré par l'ouverture d'une transaction :

- cela permet de récupérer un objet et de l'utiliser dans un autre accès, par exemple, lors d'une mise à jour : dans un premier temps on récupère l'objet et dans un second temps on le met à jour.
- Si on dans une session, il est possible de parcourir les liens ManyToOne entre deux instances.



La première méthode d'un DataService appelé par le controller doit être annotée par @Transactional.

`Il s'agit de l'annotation Transaction de Spring. Le mode de propagation de la transaction doit être REQUIRED qui est le mode par défaut.`

La méthode du DataService doit s'assurer que toutes les entités liées qui doivent être renvoyées par le DataService, aient été chargées par la méthode : une fois la méthode terminée, la transaction sera fermée et il ne sera plus possible de charger une entité lorsqu'on parcourt un lien entre deux entités JPA.

Un DataService ne doit accéder à la base qu'à travers l'utilisation de Repository. Cela garantit que seule une syntaxe portable de l'accès à la base de données sera présente dans le code.



Un DataService ne doit utiliser que les interfaces Repository pour accéder à la base.



L'usage de jdbc pour accéder à la base de données et à proscrire.

3.12 DEFINITION D'UN REPOSITORY

3.12.1 LE REPOSITORY

Le Repository est une interface annotée par @Repository qui hérite de la classe `JpaRepository`.

Cette interface permet de définir les accès à la base de données et dispose de méthodes prêtes à l'emploi comme des méthodes permettant de gérer la pagination.

3.12.2 OPTIMISATION DES RECHERCHES

Le Repository sert principalement à définir des requêtes de recherche d'informations.

Dans la grande majorité des cas, un objet est lié à d'autres objets et plusieurs d'entre eux sont nécessaires au front : il va falloir les charger.

Pour optimiser les temps de réponse, il est souhaitable de charger en une seule requête les différents objets. Pour cela, il est possible d'annoter la méthode avec `EntityGraph`.

Cette annotation permet grâce à son attribut attributePath, la définition de la liste des entités liées à charger.

3.13 DEFINITION D'UNE ENTITE

3.13.1 ENTITE JPA

Une entité JPA est une classe Java permettant de faire le lien entre la classe Java et le modèle physique de la base.

Une classe Java doit posséder une et une seule propriété annotée Id ou EmbeddedId pour une clé composée. Cette propriété doit impérativement correspondre à un index unique comme une clé primaire. JPA/Hibernate ne fonctionne pas s'il récupère deux enregistrements ou plus pour une recherche avec l'id.



Une entité JPA doit posséder un Id ou EmbeddedId correspond à un index unique.

Une entité JPA doit être annoté avec

- Entity
- Table afin de préciser le nom de la table ou de la vue.

3.13.2 DEFINITION DES PROPRIETES

Les propriétés simples d'une entité peuvent être annotées avec Column afin de définir le nom de la colonne.



Nous conseillons d'annoter toutes les propriétés avec Column.

Cela permet l'utilisation de nom de propriétés correspondant à un concept métier et non pas au nom de la colonne.