ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μελέτη Σύγχρονων Προκλήσεων στο Πεδίο του Batch Processing και Σχεδιασμός με Ανάπτυξη Εφαρμογής Data Migration για Μεγάλο Ηλεκτρονικό Κατάστημα

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

της

ΣΚΙΑΔΑ ΕΥΓΕΝΙΑΣ

Επιβλέπων : Θεοδώρα Βαρβαρίγου

Καθηγήτρια Ε.Μ.Π

Αθήνα, Μάιος 2017

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ

ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μελέτη Σύγχρονων Προκλήσεων στο Πεδίο του Batch Processing και Σχεδιασμός με Ανάπτυξη Εφαρμογής Data Migration για Μεγάλο Ηλεκτρονικό Κατάστημα

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

της

ΣΚΙΑΔΑ ΕΥΓΕΝΙΑΣ

Επιβλέπων : Θεοδώρα Βαρβαρίγου

Καθηγήτρια Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 16η Μαίου 2017.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Βαρβαρίγου Θεοδώρα	Βαρβαρίγος Εμμανουήλ	Λούμος Βασίλειος
Καθηγήτρια Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Μάιος 2017

.....

Ευγενία Σκιαδά

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη Στα Ελληνικά

Αντικείμενο της παρούσας διπλωματικής αποτελεί σε πρώτο στάδιο η περιγραφή των σύγχρονων προκλήσεων στο πεδίο των Βάσεων Δεδομένων ενώ σε δεύτερο στάδιο υλοποιείται μια Batch εφαρμογή για μεταφορά μεγάλου όγκου δεδομένων μεταξύ δύο σχεσιακών βάσεων, διαδικασία που απαντάται συχνά στα σύγχρονα επιχειρησιακά περιβάλλοντα και αποτελεί πρόκληση λόγω του όγκου και της σημαντικότητας των δεδομένων για κάθε επιχείρηση. Στο σημερινό τοπίο ολοένα και περισσότερες ανθρώπινες δραστηριότητες πραγματοποιούνται μέσω διαδικτύου (ηλεκτρονικό εμπόριο, ηλεκτρονικές δημόσιες υπηρεσίες κλπ), ενώ η εισβολή των κοινωνικών δικτύων σηματοδοτεί την πρόσβαση σε μια νέα εποχή οπού κάθε άτομο διατηρεί μια επιπλέον «online ζωή και προσωπικότητα». Είναι λοιπόν αντιληπτό, ότι σε αυτό το διαμορφούμενο πλαίσιο ζωής παράγονται καθημερινά τεράστιοι όγκοι δεδομένων, οι οποίοι γεννούν αυξανόμενες προκλήσεις όσον αφορά στον τρόπο διαχείρισης, μεταφοράς και αποθήκευσης τους, δεδομένου ότι μετά την ολοκλήρωση οποιασδήποτε ενέργειας, τα δεδομένα πρέπει να παραμένουν πλήρη και σε συνεπή κατάσταση.

Ειδικότερα, η μεταφορά μεγάλου όγκου δεδομένων μεταξύ δυο σχεσιακών βάσεων είναι μια επιχειρησιακή ανάγκη η οποία προκύπτει συχνά, για λόγους συντήρησης ή αναβάθμισης των υποδομών, αλλαγής τεχνολογίας κλπ, και παρουσιάζει δυσκολίες και προκλήσεις που αφορούν την ταχύτητα επεξεργασίας, την διατήρηση της ορθότητας των δεδομένων, των μεταξύ τους εξαρτήσεων ακόμα και την διαχείριση απρόοπτων διακοπών της διαδικασίας. Η εφαρμογή που υλοποιήθηκε στα πλαίσια της εργασίας αυτής, προέκυψε από πραγματική επιχειρησιακή απαίτηση που αφορούσε την μεταφορά της βάσης δεδομένων ενός διαδικτυακού βιβλιοπωλείου, από MySQL σύστημα διαχείρισης σε Oracle. Η εφαρμογή είναι γραμμένη σε Java και για την εξαγωγή, επεξεργασία και εγγραφή των δεδομένων κατά δεσμίδες χρησιμοποιεί το περιβάλλον Spring Batch. Περιλαμβάνει ροές εξαγωγής δεδομένων από βάση αλλά και από αρχεία, καθώς είναι σύνηθες κάποιοι πίνακες των επιχειρησιακών βάσεων να ενημερώνονται με δεδομένα που αντλούν από αρχεία. Επιπλέον περιλαμβάνει μια γραφική διεπαφή χρήστη βασισμένη στο Spring MVC περιβάλλον ώστε να μπορούν οι εμπορικοί χρήστες να παραμετροποιούν τις εργασίες, να ελέγχουν τη διαδικασία και να ενημερώνονται για λάθη και τυχόν αποτυχίες καθώς, όπως όλες οι batch διαδικασίες, οι εργασίες τρέχουν στο παρασκήνιο χωρίς παρέμβαση χρήστη. Για την έγκαιρη ενημέρωση των διαχειριστών κατά το σενάριο αποτυχίας έχει υλοποιηθεί και μηχανισμός αποστολής email μόλις κάποια εργασία αποτύχει. Για το σενάριο αυτό υλοποιείται και μηχανισμός επαναλήψεων της εργασίας από το σημείο στο οποίο διακόπηκε, ώστε να μη χρειαστεί να επαναληφθεί από την αρχή κάτι που θα είχε μεγάλο χρονικό κόστος. Τέλος, για τη αυτού ακριβώς του κόστους, δίνεται η επιλογή να τρέξει παράλληλα η διαδικασία με χρήση partitioning και παρατίθεται σύγκριση χρόνου της σειριακής και της παράλληλης εκτέλεσης.

Λέξεις Κλειδιά: Βάσεις Δεδομένων, RDBMS, Batch Processing, Spring Batch, Spring MVC, Java, Γραφική Διεπαφή Χρήστη

Περίληψη στα Αγγλικά (Abstract)

The objective of this Diploma Thesis is, in the first place the description of modern challenges in the field of Databases, and secondly the design and implementation of a Batch application, for the migration of large data sets between two Relational Databases, which is a common business demand and shows several challenges because of the volume and importance of the data for a modern company. Nowadays, more and more human activities are performed via the Internet (e-commerce, electronic public services etc), not to mention our everyday occupation with the various social media platforms. In this context, it is obvious that, large amounts of data are produced in a daily basis, which in turn means augmented challenges concerning the monitoring, migration, storing of such great volumes of Information, in order for that Information to remain consistent.

Data Migration for large amounts of data is especially considered an important and challenging business need, that occurs whenever a company or organization plans to upgrade or change its infrastructure or move to another technology, which is a frequent case. After the migration is completed data need to be left in a consistent state avoiding any data loss, while the time consumed for the process plays an crucial role. The application developed as part of this Diploma Thesis, responds to a real-life business demand, which was the migration of the Relational Database of an online bookstore from the existing MySQL management System to an Oracle Management System. The application is written in Java, and employs Spring Batch Framework for the extracting, transforming and loading of the data, splitted in batches. It includes Jobs for extracting data from the Database as well as from files, for in many modern companies, databases include tables that are populated with information extracted from files. It also offers a Graphical User Interface based in Spring MVC Framework, aimed to help business users configure and monitor the whole process- that runs in the background without human intervention- and get informed for the status of jobs, that is whether they finish succeeding or failing. In order for the users to be immediately informed of a failed job, a mechanism for sending emails on job failure is provided. For this exact job failure scenario, we also implemented skip and retry mechanisms. Finally, aiming to reduce time consumation of the process, we provided alternative parallel execution of the jobs using partitioners, and compared the time consummation results of sequential and parallel execution.

Keywords: Databases, RDBMS, Batch Processing, Spring Batch, Spring MVC, Java, Web Interface

Ευχαριστίες

Θα ήθελα να ευχαριστήσω την καθηγήτρια μου κ. Θεοδώρα Βαρβαρίγου, επιβλέπουσα της παρούσης διπλωματικής εργασίας, καθώς και τον υποψήφιο διδάκτορα κ. Παύλο Κρανά, για την άψογη συνεργασία που είχαμε, την καθοδήγηση, την ενθάρρυνση και την εμπιστοσύνη που μου έδειξαν σε όλη τη διάρκεια της ενασχόλησής μου και ακόμα επειδή μου έδωσαν την ευκαιρία να ασχοληθώ με ένα τόσο ενδιαφέρον θέμα δίνοντάς μου παράλληλα μεγάλη ευελιξία.

Έπειτα, θα ήθελα να ευχαριστήσω τον συνάδελφο κ. Νίκο Παπαδόπουλο, διπλωματούχο Η.Μ.Μ.Υ., για την αμέριστη και έμπρακτη βοήθεια που μου παρείχε κατά τη διάρκεια συγγραφής της παρούσας εργασίας.

Επίσης, ευχαριστώ πολύ την οικογένεια μου και τους φίλους μου για την στήριξη και συμπαράσταση που μου παρείχαν και εξακολουθούν να μου παρέχουν όλα αυτά τα χρόνια των σπουδών μου στο Εθνικό Μετσόβιο Πολυτεχνείο.

Τέλος, θα ήθελα να εκφράσω την ευγνωμοσύνη μου στους συμφοιτητές μου και σε όλο το διδακτικό, ερευνητικό και διοικητικό προσωπικό της Σχολής Η.Μ.Μ.Υ. και του Ε.Μ.Π.

Περιεχόμενα

ΚΕΦΑΛΑΙΟ 1
Εισαγωγή

1.1 Βάσεις Δεδομένων

Ένα σύστημα διαχείρισης βάσεων δεδομένων (Database Management System, DBMS) είναι ένα σύνολο από σχετιζόμενα δεδομένα και προγράμματα που χρησιμοποιούνται για πρόσβαση σε αυτά τα δεδομένα. Η συλλογή των δεδομένων, που συνήθως αναφέρεται ως βάση δεδομένων, περιέχει πληροφορίες σχετικά με μια επιχείρηση ή έναν οργανισμό. Στόχος ενός συστήματος διαχείρισης βάσεων δεδομένων είναι να παρέχει έναν βολικό και αποτελεσματικό τρόπο να αποθηκεύονται και να ανακαλούνται οι πληροφορίες αυτές από τις βάσεις δεδομένων.

Τα συστήματα διαχείρισης βάσεων δεδομένων προέκυψαν σαν απόκριση στις αρχικές μεθόδους διαχείρισης δεδομένων. Στη δεκαετία του ’60 οι απαιτούμενες πληροφορίες αποθηκεύονταν στα αρχεία του λειτουργικού συστήματος ηλεκτρονικών υπολογιστών και άρα για την εκτέλεση ενεργειών όπως εισαγωγή, ενημέρωση ή διαγραφή αυτών των πληροφοριών χρειαζόταν να γραφεί ανά περίπτωση και ένα κατάλληλο πρόγραμμα. Πέρα από αυτή την ανάγκη, που προφανώς δυσχέραινε τη λειτουργία των επιχειρήσεων, μια τέτοιου είδους προσέγγιση είχε και άλλα μειονεκτήματα όπως προβλήματα ταυτόχρονης πρόσβασης, ασυνέπειας των δεδομένων, ασφάλειας κλπ.

Σήμερα τα συστήματα βάσεων δεδομένων βρίσκονται παντού και οι περισσότεροι άνθρωποι συνδιαλέγονται άμεσα ή έμμεσα μαζί τους πολλές φορές την ημέρα. Όταν μέσω web περιηγούμαστε σε ένα online βιβλιοπωλείο και υποβάλλουμε μια online παραγγελία, όταν αναζητούμε τον υπολοιπόμενο χρόνο ομιλίας στο συμβόλαιο που έχουμε με έναν τηλεπικοινωνιακό πάροχο ή όταν συνδεόμαστε στην ιστοσελίδα μιας τράπεζας και ανακτούμε πληροφορίες σχετικές με το λογαριασμό μας, στην πραγματικότητα λαμβάνει χώρα κάποιας μορφής επικοινωνία με μια βάση. Αυτή η επικοινωνία, σε άλλες περιπτώσεις, μπορεί να μην είναι καν προφανής για τους χρήστες. Μπορεί για παράδειγμα απλά να περιηγούμαστε σε μια ιστοσελίδα –χωρίς να εκτελούμε κάποια ενέργεια εγγραφής ή αναζήτησης δεδομένων– η ιστοσελίδα όμως στο παρασκήνιο να καταγράφει σε μια βάση τα κλικ που κάναμε ώστε να μπορεί σε δεύτερο χρόνο να αναλύσει τις προτιμήσεις μας και να μας προτείνει σχετικό περιεχόμενο την επόμενη φορά που θα την επισκεφτούμε.

Τα συστήματα διαχείρισης βάσεων δεδομένων συνεχίζουν να εξελίσσονται και να προσφέρουν όλο και πιο προηγμένα συστήματα αποθήκευσης, ανάκτησης και διανομής δεδομένων. Η μετεξέλιξή τους έχει

τροφοδοτήσει την ανάπτυξη πληθώρας προηγμένων τεχνολογιών όπως των συστημάτων πελάτη / διακομιστή (client / server), του Data Warehousing και του Online Analytical Processing (OLAP) που αποτελούν τον πυρήνα των κορυφαίων σύγχρονων πληροφοριακών συστημάτων. Περισσότερα για όλα αυτά αφήνεται να αναφερθούν στα επόμενα.

1.1.1 Γλώσσες Βάσεων Δεδομένων

Ένα σύστημα βάσης δεδομένων παρέχει μια γλώσσα ορισμού των δεδομένων (**Data Definition Language, DDL**) με την οποία καθορίζεται το σχήμα της βάσης και μια γλώσσα χειρισμού των δεδομένων (**Data Manipulation Language, DML**) για να εκφράζονται τα ερωτήματα και να εκτελούνται οι ενημερώσεις των δεδομένων. Αυτές οι δυο δεν είναι δυο ξεχωριστές γλώσσες αλλά πρακτικά μέρη μιας γλώσσας, κατά κανόνα της γλώσσας **SQL** η οποία είναι η πλέον χρησιμοποιούμενη γλώσσα βάσεων δεδομένων.

Μέσω της **DDL** καθορίζουμε το σχήμα της βάσης δεδομένων το οποίο αποτελείται από ένα σύνολο ορισμών και πρόσθετων ιδιοτήτων των δεδομένων. Οι τιμές που αποθηκεύονται στη βάση πρέπει να ικανοποιούν κάποιους περιορισμούς συνέπειας και ακεραιότητας οι οποίοι καθορίζονται με τη βοήθεια της DDL. Ένας περιορισμός μπορεί να είναι ένα αυθαίρετο κατηγόρημα που αναφέρεται στη βάση δεδομένων και σε κάποιες περιπτώσεις μπορεί είναι χρονοβόρο. Για αυτό το λόγο τα συστήματα βάσεων δεδομένων εφαρμόζουν περιορισμούς ακεραιότητας που μπορούν να ελέγχονται με ελάχιστο κόστος (περιορισμοί πεδίου τιμών, ακεραιότητα αναφορών-κλειδιών, εξουσιοδότηση και άλλες διασφαλίσεις). Η έξοδος της DDL τοποθετείται στο λεξικό δεδομένων (**Data Dictionary**), το οποίο περιέχει μετα-δεδομένα, δεδομένα δηλαδή σχετικά με τα δεδομένα. Το λεξικό είναι ένας ειδικού τύπου πίνακας που μπορεί να προσπελαστεί και να ενημερωθεί μόνο απο το ίδιο το σύστημα της βάσης δεδομένων.

Μέσω της **DML** οι χρήστες μπορούν να ανακτούν πρόσβαση στα δεδομένα της βάσης μέσω της υποβολής ερωτημάτων (queries). Ένα query είναι μια πρόταση σε γλώσσα SQL που ανακαλεί πληροφορίες. Πέρα από την ανάκληση πληροφοριών (λειτουργία read), οι άλλοι τύποι πρόσβασης στα δεδομένα που μπορούμε να έχουμε μέσω της DML είναι η εισαγωγή των πληροφοριών (λειτουργία create), η διαγραφή πληροφοριών (λειτουργία delete) και η τροποποίηση υπάρχουσων πληροφοριών (λειτουργία update). Οι τέσσερις αυτές λειτουργίες αναφέρονται συχνά με το ακρωνύμιο **CRUD** (Create-Read-Update-Delete).

1.1.2 Μοντέλα Βάσεων Δεδομένων

Τα μοντέλα βάσεων δεδομένων είναι ο τρόπος να καθοριστεί η λογική δομή μιας βάσης δεδομένων και ο τρόπος με τον οποίο τα δεδομένα θα αποθηκεύονται, οργανώνονται και διαχειρίζονται. Τα μοντέλα

δεδομένων μπορούν να χωριστούν σε τέσσερις κατηγορίες: το σχεσιακό μοντέλο (**Relational Model**), το μοντέλο οντοτήτων-συσχετίσεων (**Entity-Relationship ή E-R Model**), το αντικειμενοστραφές μοντέλο (**Object-Based Data Model**) και το ημιδομημένο μοντέλο (**Semi-structured Data Model**). Παρουσιάζουμε εδώ εν συντομία καθένα από αυτά, στεκόμενοι περισσότερο στο με διαφορά δημοφιλέστερο και ευρύτερα χρησιμοποιούμενο –το σχεσιακό μοντέλο, ενώ θα πρέπει να αναφερθεί πως ιστορικά υπήρξαν και άλλα μοντέλα όπως το δικτυακό (**Network Model**) ή το ιεραρχικό (**Hierarchical Model**) τα οποία όμως ήταν αρκετά περίπλοκα και δυσνόητα με αποτέλεσμα να έχουν στο μεγαλύτερο μέρος τους αντικατασταθεί απο το σχεσιακό.

1.1.3 Το Σχεσιακό Μοντέλο (Relational Model)

Το σχεσιακό μοντέλο χρησιμοποιεί ένα σύνολο απο πίνακες που αντιπροσωπεύουν τα δεδομένα και τις σχέσεις μεταξύ αυτών των δεδομένων. Κάθε πίνακας (αναφέρεται και ως σχέση) έχει πολλές στήλες και κάθε στήλη έχει ένα μοναδικό όνομα. Το σχεσιακό μοντέλο είναι παράδειγμα μοντέλου βασισμένου στις **εγγραφές**. Η βάση δεδομένων είναι δομημένη σε εγγραφές (αναφέρονται και ως γραμμές) σταθερής μορφής, διάφορων τύπων. Κάθε πίνακας περιέχει εγγραφές ενός συγκεκριμένου τύπου που ορίζεται από ένα σταθερό αριθμό πεδίων ή / και ιδιοτήτων. Οι στήλες του πίνακα αντιστοιχούν στις ιδιότητες του τύπου της εγγραφής.
Στο σχεσιακό μοντέλο πρέπει να έχουμε ένα τρόπο να καθορίζουμε πως θα διαφοροποιούνται οι εγγραφές μιας δεδομένης σχέσης. Αυτό εκφράζεται σε σχέση με τις ιδιότητές τους. Οι τιμές των ιδιοτήτων μιας εγγραφής πρέπει να είναι τέτοιες που να μπορούν να προσδιορίζουν μοναδικά την εγγραφή. Με άλλα λόγια, δεν επιτρέπεται δυο εγγραφές σε μια σχέση (πίνακα) να έχουν ακριβώς την ίδια τιμή σε όλες τις ιδιότητές τους, για αυτό χρησιμοποιούμε την έννοια των κλειδιών. Ορίζουμε ότι ένα υπερ-κλειδί είναι ένα σύνολο από μια ή περισσότερες ιδιότητες που, αν ληφθούν συλλογικά, μας επιτρέπουν να προσδιορίζουμε μοναδικά την εγγραφή μιας σχέσης. Ένα υπέρ-κλειδί μπορεί να περιέχει πολλές ιδιότητες. Συνήθως ενδιαφερόμαστε για υπέρ-κλειδιά για τα οποία δεν υπάρχει υποσύνολο των ιδιοτήτων τους που να αποτελεί επίσης υπέρ-κλειδί. Αυτά τα ελάχιστα υπέρ-κλειδιά ονομάζονται υποψήφια κλειδιά.

Με τον όρο **πρωτεύον κλειδί** (primary key) δηλώνουμε ένα υποψήφιο κλειδί που έχει επιλεχθεί από τον σχεδιαστή της βάσης δεδομένων ως το βασικό μέσο προσδιορισμού των εγγραφών της βάσης. Ως εκ τούτου, οποιεσδήποτε δυο εγγραφές στο σύνολο οντοτήτων απαγορεύεται να έχουν ταυτόχρονα την ίδια τιμή στις ιδιότητες του κλειδιού. Το κλειδί αντιπροσωπεύει έναν περιορισμό της κατάστασης η οποία μοντελοποιείται.

Τέλος, μια σχέση (σχέση αναφοράς) μπορεί να περιλαμβάνει στις ιδιότητές της το πρωτεύον κλειδί μιας άλλης σχέσης (αναφερόμενη σχέση). Αυτή η ιδιότητα ονομάζεται **ξένο κλειδί** (foreign key). Τα ξένα κλειδιά ορίζουν τους περιορισμούς ακεραιότητας αναφορών, σύμφωνα με τους οποίους οι τιμές που εμφανίζονται σε καθορισμένες ιδιότητες οποιασδήποτε εγγραφής στη σχέση αναφοράς πρέπει να εμφανίζονται επίσης σε καθορισμένες ιδιότητες τουλάχιστον μιας εγγραφής στην αναφερόμενη σχέση.

1.1.4 Μοντέλο Οντοτήτων-Συσχετίσεων (Entity-Relationship ή E-R Model)

Βάση για το μοντέλο οντοτήτων-συσχετίσεων είναι η κατηγοριοποίηση αντικειμένων και των σχέσεων μεταξύ τους. Οντότητα είναι ένα αντικείμενο ενδιαφέροντος του πραγματικού κόσμου που ξεχωρίζει από τα υπόλοιπα (άνθρωποι, μέρη, αντικείμενα, γεγονότα κλπ). Συσχέτιση είναι η σύνδεση δυο ή περισσότερων οντοτήτων με τρόπο που να παρουσιάζει ενδιαφέρον για το σχεδιασμό. Το μοντέλο οντοτήτων-συσχετίσεων χρησιμοποιείται στο πρώτο στάδιο σχεδίασης ενός συστήματος πληροφοριών, κατά την ανάλυση των απαιτήσεών του.

1.1.5 Αντικειμενοστραφές Μοντέλο Δεδομένων (Object-Based Data Model)

Ο αντικειμενοστραφής προγραμματισμός (Java, C++ κ.α.) έχει γίνει η κυρίαρχη μεθοδολογία ανάπτυξης λογισμικού. Αυτό οδήγησε στην ανάπτυξη ενός αντικειμενοστραφούς μοντέλου δεδομένων που μπορεί να θεωρηθεί επέκταση του μοντέλου οντοτήτων-συσχετίσεων και περιγράφει όχι μόνο τα χαρακτηριστικά εκείνα που ορίζουν την κατάσταση ενός αντικειμένου αλλά και τις ενέργειες που μπορεί να εκτελούνται από / σε αυτό. Τα αντικείμενα δηλαδή σε ένα τέτοιο μοντέλο έχουν εκτός από ταυτότητα και συμπεριφορά.

1.1.6 Ημιδομημένο Μοντέλο Δεδομένων (Semi-structured Data Model)

Το ημιδομημένο μοντέλο δεδομένων επιτρέπει να υπάρχουν προδιαγραφές των δεδομένων όταν διαφορετικά στοιχεία δεδομένων του ίδιου τύπου μπορεί να έχουν διαφορετικά σύνολα ιδιοτήτων. Τα μοντέλα αυτά είναι πολύ ευέλικτα καθώς δεν περιορίζονται από αυστηρούς ορισμούς σχημάτων αλλά δεν είναι τόσο αποδοτικά κατά την εκτέλεση ερωτημάτων. Τυπικά οι εγγραφές μιας ημιδομημένης βάσης (που μπορεί για παράδειγμα να είναι XML αρχεία) αποθηκεύονται με μια αναφορά για τη θέση τους στο file system κάτι που επιτρέπει την εκτέλεση path-based ερωτημάτων αποδοτικά, χωρίς όμως να συμβαίνει το ίδιο για περισσότερο σύνθετα ερωτήματα.

Προς απόδειξη του ισχυρισμού οτι το σχεσιακό μοντέλο είναι με διαφορά το ευρύτερα χρησιμοποιούμενο μοντέλο βάσεων δεδομένων μπορούμε να δώσουμε το Σχήμα 1.1 στο οποίο απεικονίζονται τα 10

δημοφιλέστερα DBMs τον Απρίλιο του 2017[1], μαζί με το μοντέλο στο οποίο βασίζονται και την μεταβολή της δημοτικότητάς τους τα τελευταία δυο χρόνια. Τις 7 απο τις 10 θέσεις καταλαμβάνουν σχεσιακά συστήματα διαχείρισης, με την MongoDB (ημιδομημένο μοντέλο, το πρώτο μη σχεσιακό μοντέλο της λίστας) να χάνει μάλιστα το 2017 την θέση που είχε εντός της πεντάδας. Στις τρεις πρώτες θέσεις, με μεγάλη διαφορά από όλα τα υπόλοιπα, βρίσκονται η Oracle, η MySQL και ο Microsoft SQL Server.

Rank			DBMS	Database Model	Score		
Apr 2017	Mar 2017	Apr 2016			Apr 2017	Mar 2017	Apr 2016
1.	1.	1.	Oracle	Relational DBMS	1402.00	+2.50	-65.54
2.	2.	2.	MySQL	Relational DBMS	1364.62	-11.46	-5.49
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1204.77	-2.72	+69.72
4.	4.	↑ 5.	PostgreSQL	Relational DBMS	361.77	+4.14	+58.05
5.	5.	↓ 4.	MongoDB	Document store	325.43	-1.51	+12.98
6.	6.	6.	DB2	Relational DBMS	186.66	+1.74	+2.57
7.	7.	7.	Microsoft Access	Relational DBMS	128.18	-4.76	-3.79
8.	8.	8.	Cassandra	Wide column store	126.18	-3.01	-3.49
9.	↑ 10.	9.	Redis	Key-value store	114.36	+1.35	+3.12
10.	↓ 9.	10.	SQLite	Relational DBMS	113.80	-2.39	+5.83

Σχήμα 1.1 Τα 10 δημοφιλέστερα DBMs

1.2 Συναλλαγές

Οι συναλλαγές παίζουν σημαντικό ρόλο στη διαδικασία ανάπτυξης λογισμικού, εξασφαλίζοντας ότι τα δεδομένα και οι πόροι μιας εφαρμογής βρίσκονται πάντα σε συνεπή κατάσταση. Χωρίς τις συναλλαγές υπάρχει κίνδυνος για data corruption ή ασυνέπεια μεταξύ των δεδομένων και των εμπορικών απαιτήσεων της εφαρμογής.

Κλασικό παράδειγμα για να γίνουν τα παραπάνω αντιληπτά, αποτελεί το σενάριο μεταφοράς κεφαλαίου μεταξύ δυο τραπεζικών λογαριασμών, στο οποίο χρεώνεται ένας λογαριασμός A και πιστώνεται ένας λογαριασμός B. Σαφώς είναι απαραίτητο είτε να συμβεί και η πίστωση και η χρέωση, είτε να μη συμβεί τίποτα. Η μεταφορά κεφαλαίου δηλαδή θα πρέπει να συμβεί συνολικά ή καθόλου. Αυτή η απαίτηση ονομάζεται **ατομικότητα** (atomicity). Επιπλέον, είναι ουσιαστικό η εκτέλεση της μεταφοράς να διατηρήσει τη **συνέπεια** (consistency) της βάσης δεδομένων. Στο παράδειγμά μας δηλαδή, πρέπει να διατηρηθεί η αξία του αθροίσματος των υπολοίπων των λογαριασμών A και B. Έπειτα το σύστημα της βάσης πρέπει να λάβει ειδικά μέτρα ώστε να εξασφαλίσει ότι η συναλλαγή θα λειτουργήσει σωστά χωρίς παρέμβαση από ταυτόχρονα εκτελούμενες προτάσεις. Αυτή η απαίτηση ονομάζεται **απομόνωση** (isolation). Τέλος, μετά την επιτυχή εκτέλεση της μεταφοράς κεφαλαίων, οι νέες τιμές των υπολοίπων των λογαριασμών A και B πρέπει

[1] Δεδομένα απο το site http://db-engines.com/en/ranking που σε μηνιαία βάση καταγράφει τις τάσεις του χώρου.

να διατηρηθούν έστω και αν συμβεί κάποιο απρόοπτο περιστατικό όπως η διακοπή του συστήματος. Αυτή η απαίτηση ονομάζεται **ανθεκτικότητα** (durability).

Σε αυτό το πλαίσιο, ορίζουμε μια συναλλαγή (**transaction**) ως μια συλλογή λειτουργιών που εκτελούν μια μόνο λογική λειτουργία σε μια βάση δεδομένων, σχηματίζουν δηλαδή μια λογική μονάδα εργασίας. Κάθε συναλλαγή είναι μια μονάδα ατομικότητας και συνέπειας. Απαιτούμε οι συναλλαγές να μην παραβιάζουν οποιονδήποτε από τους περιορισμούς συνέπειας της βάσης δεδομένων. Αν η βάση ήταν σε συνεπή κατάσταση όταν ξεκίνησε μια συναλλαγή, πρέπει να είναι συνεπής και όταν τερματίσει η συναλλαγή. Είναι ευθύνη των προγραμματιστών να ορίσουν σωστά τις διάφορες συναλλαγές έτσι ώστε η κάθε μία να διατηρεί τη συνέπεια της βάσης δεδομένων. Στο παράδειγμα της μεταφοράς, θα μπορούσαν εντός συναλλαγής να εκτελεστούν δυο διαφορετικές ενέργειες: μια για χρέωση του λογαριασμού Α και μια για πίστωση του λογαριασμού Β. Η εκτέλεση των δυο ενεργειών σε σειρά θα διατηρήσει τη συνέπεια. Εντούτοις, κάθε ενέργεια από μόνη της δεν μετασχηματίζει τη βάση από μια συνεπή κατάσταση σε μια νέα συνεπή κατάσταση, άρα αυτές οι ενέργειες είναι προγραμματιστικά λάθος να θεωρηθούν ξεχωριστές συναλλαγές.

Παραδοσιακά, οι τέσσερις ιδιότητες των συναλλαγών που μόλις αναλύθηκαν αναφέρονται στην επιστήμη των υπολογιστών με το ακρωνύμιο **ACID** (Atomicity, Consistency, Isolation, Durability).

1.2.1 Επίπεδα Απομόνωσης

Αξίζει να σταθούμε λίγο παραπάνω στην ιδιότητα της απομόνωσης (isolation) η οποία καθορίζει το πως και πότε οι αλλαγές που γίνονται από μια συναλλαγή σε κάποιες εγγραφές είναι ορατές στις υπόλοιπες συναλλαγές. Στα συστήματα διαχείρισης βάσεων δεδομένων, μηχανισμοί ελέγχου συγχρονισμού ενεργοποιούνται από το database engine όταν προκύπτουν ταυτόχρονες συναλλαγές στα ίδια δεδομένα που, με βάση το ορισμένο επίπεδο απομόνωσης, αποφασίζουν το πως θα συμπεριφερθεί το σύστημα. Ένα χαμηλό επίπεδο απομόνωσης αυξάνει την πιθανότητα ταυτόχρονης πρόσβασης πολλών χρηστών στα ίδια δεδομένα, αυξάνει όμως και τον κίνδυνο εμφάνισης προβλημάτων συγχρονισμού. Από την άλλη, ένα υψηλότερο επίπεδο απομόνωσης μειώνει τον κίνδυνο προβλημάτων συγχρονισμού, αλλά απαιτεί την χρήση περισσότερων συστημικών πόρων ενώ αυξάνει και την πιθανότητα κάποιες συναλλαγές να μπλοκάρουν την εκτέλεση άλλων.

Είναι αντιληπτό ότι η επιλογή του επιπέδου απομόνωσης εμπεριέχει κινδύνους για την συμπεριφορά και την απόδοση του συστήματος. Το πρότυπο ANSI/ISO SQL ορίζει τρία φαινόμενα που μπορεί να προκύψουν όταν μια συναλλαγή επιχειρήσει να διαβάσει δεδομένα που μια άλλη έχει τροποποιήσει και τα οποία

πρέπει να λαμβάνονται καλά υπόψη κατά τη σχεδίαση της εκάστοτε λύσης ώστε να βρίσκεται η κατά το δυνατό χρυσή τομή μεταξύ απόδοσης και αξιοπιστίας. Τα φαινόμενα είναι τα ακόλουθα:

- **Dirty reads:** Μια συναλλαγή Α διαβάζει δεδομένα που μόλις τροποποιήθηκαν από κάποια άλλη συναλλαγή Β, χωρίς όμως η Β να έχει ακόμα ολοκληρωθεί. Αν η Β κάνει roll back ή τροποποιήσει εκ νέου τα δεδομένα η συναλλαγή Α θα έχει διαβάσει λανθασμένες τιμές.
- **Non repeatable reads:** Αυτό το φαινόμενο προκύπτει όταν κατά τη διάρκεια μιας συναλλαγής δεδομένα διαβάζονται περισσότερες από μια φορές με την κάθε φορά να επιστρέφει διαφορετικά αποτελέσματα. Σε ένα lock-based σύστημα ελέγχου συγχρονισμού αυτό μπορεί να συμβεί αν δεν απαιτείται κλείδωμα για την ανάγνωση ή αν το κλείδωμα απελευθερώνεται απευθείας μετά την εκτέλεση της ανάγνωσης.
- **Phantom reads:** Το φαινόμενο αυτό προκύπτει όταν κατά τη διάρκεια μιας συναλλαγής εκτελούνται δυο πανομοιότυπα ερωτήματα με τα δεδομένα που επιστρέφονται στο καθένα να είναι διαφορετικό. Αυτό μπορεί να συμβεί όταν δεν χρησιμοποιούνται range locks για την εκτέλεση των ερωτημάτων και μια δεύτερη συναλλαγή δημιουργήσει νέες εγγραφές ή τροποποιήσει κάποια από τις υπάρχουσες εντός αυτού του range. Τα phantom reads είναι ειδική περίπτωση non repeatable reads.

Έχοντας παρουσιάσει αυτά τα φαινόμενα μπορούμε τώρα να παρουσιάσουμε και τα επίπεδα απομόνωσης που μπορούν να οριστούν –είτε σε επίπεδο βάσης είτε προγραμματιστικά– και που σε σειρά προτεραιότητας από το υψηλότερο στο χαμηλότερο είναι τα ακόλουθα:

- **Serializable:** Το υψηλότερο επίπεδο απομόνωσης που εξασφαλίζει ότι τυχόν ταυτόχρονες συναλλαγές θα είναι σειριοποιημένες, θα έχουν δηλαδή πάνω στα δεδομένα το ίδιο αποτέλεσμα που θα είχαν αν εκτελούνταν και ολοκληρώνονταν σειριακά η μια μετά την άλλη. Σε lock-based συστήματα για την επίτευξη αυτού του είδους απομόνωσης απαιτούνται read και write κλειδώματα τα οποία ελευθερώνονται με την ολοκλήρωση της συναλλαγής αλλά και range κλειδώματα για την αποφυγή των phantom reads. Σε non lock-based συστήματα (π.χ. snapshot isolation) δεν απαιτούνται κλειδώματα αλλά αν προκύψει σύγκρουση από την προσπάθεια ταυτόχρονης εγγραφής δυο συναλλαγών, μόνο η μια επιτρέπεται να ολοκληρωθεί.
- **Repeatable reads:** Σε αυτό το επίπεδο απομόνωσης απαιτούνται read και write κλειδώματα όπως και πριν αλλά δεν υλοποιούνται range κλειδώματα και άρα είναι πιθανό να προκύψουν phantom reads.
- **Read committed:** Σε αυτό το επίπεδο απομόνωσης τα write κλειδώματα ελευθερώνονται με την ολοκλήρωση της συναλλαγής τα read όμως ελευθερώνονται αμέσως μόλις ολοκληρωθεί η ανάγνωση με αποτέλεσμα να καθίσταται πιθανή η εμφάνιση non-repeatable reads. Πιο απλά, το

επίπεδο αυτό αποτρέπει τον χρήστη από το να κάνει dirty reads, δεν εγγυάται όμως ότι αν η συναλλαγή αμέσως μετά ξαναδιαβάσει τις ίδιες εγγραφές θα τις βρει ίδιες.

- **Read uncommitted:** Τέλος, στο χαμηλότερο επίπεδο απομόνωσης είναι πιθανό να προκύψουν και dirty reads, μια συναλλαγή δηλαδή να διαβάσει αλλαγές που έχουν γίνει από κάποια άλλη που δεν έχει ακόμα ολοκληρωθεί.

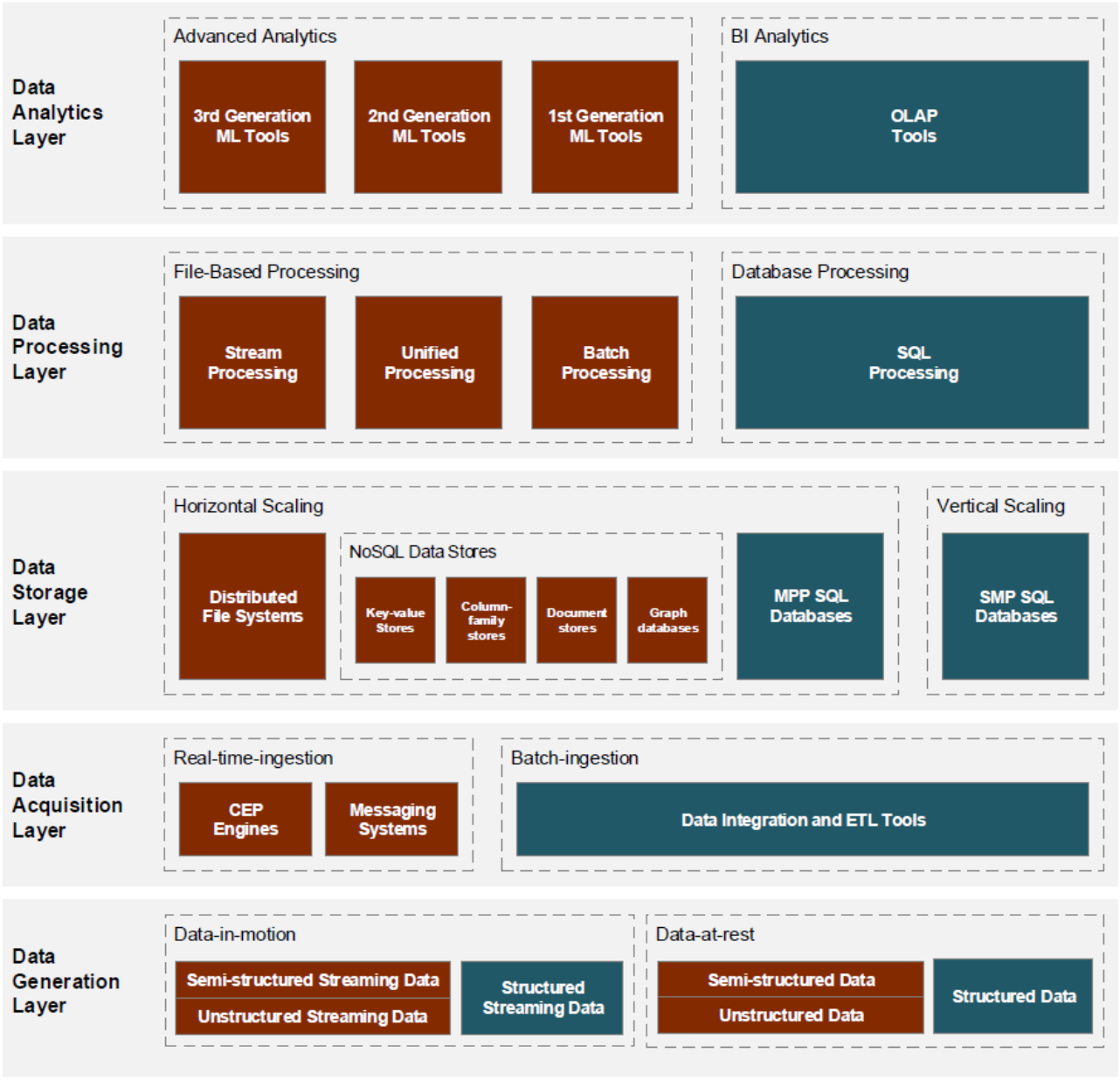
1.3 Big Data

Βρισκόμαστε αναμφίβολα στην εποχή των Big Data. Ο όρος εμφανίστηκε τα τελευταία χρόνια στο επιστημονικό πεδίο των βάσεων δεδομένων και αποκτά ολοένα και περισσότερη δημοφιλία. Καθημερινά οι χρήστες του διαδικτύου παράγουν έναν τεράστιο όγκο δεδομένων με τη συμμετοχή τους στα μέσα κοινωνικής δικτύωσης (Facebook, Twitter, Instagram κ.α.), οι επιχειρήσεις αποθηκεύουν μεγάλο πλήθος δεδομένων που αφορούν τις online δραστηριότητες και προτιμήσεις των πελατών τους ώστε να αναλύουν και να προβλέπουν τις ανάγκες τους και η ακαδημαϊκή κοινότητα χειρίζεται λεπτομερείς πληροφορίες μεγάλου όγκου απαραίτητες για την έρευνα. Η ανάγκη για χειρισμό και ανάλυση των δεδομένων είναι επιτακτική, το επιστημονικό πεδίο όμως βρίσκεται ακόμα στη βρεφική του ηλικία καθώς έχει πολλά προβλήματα και προκλήσεις να αντιμετωπίσει. Η σημαντικότερη από αυτές είναι το ότι ο χώρος είναι αρκετά ετερογενής με αποτέλεσμα να μην υπάρχει μια ενοποιημένη πλατφόρμα που να ανταποκρίνεται ικανοποιητικά σε όλα τα σενάρια. Οι τεχνολογίες που διατίθενται αντιμετωπίζουν επαρκώς ένα μόνο τμήμα του σύμπαντος των Big Data και παρουσιάζονται ανεπαρκείς στα υπόλοιπα (φαινόμενο που συχνά περιγράφεται απο την κοινότητα και ως «No One Size Fits All»). Ο χώρος εξελίσσεται ταχύτατα, οι υπάρχουσες τεχνολογίες διαρκώς αλλάζουν, νέες εμφανίζονται διεκδικώντας μερίδιο της πίτας και κάπως έτσι οι εταιρίες –ιδιαίτερα οι μικρού ή μεσαίου μεγέθους– αδυνατούν να παρακολουθήσουν τις εξελίξεις και δυσκολεύονται να επιλέξουν τα απαραίτητα εργαλεία για την επίλυση των προβλημάτων που τους απασχολούν.

Για να παρουσιαστεί το πλήθος αποφάσεων που πρέπει να ληφθούν, αλλά και για να καταστεί σαφές το πόσο δύσκολο μπορεί να είναι ανά περίπτωση να επιλεχθούν οι κατάλληλες τεχνολογίες, μπορούμε να ανατρέξουμε στο Σχήμα 1.2 στο οποίο απεικονίζεται το σύνολο των πιθανών αποφάσεων που ο σχεδιαστής ενός τέτοιου συστήματος έχει να πάρει χωρισμένες σε επίπεδα[2]. Από κάτω προς τα πάνω έχουμε το επίπεδο παραγωγής των δεδομένων (**Data Generation Layer**), το επίπεδο απόκτησης (**Data Acquisition Layer**), το επίπεδο αποθήκευσης (**Data Storage Layer**), το επίπεδο επεξεργασίας (**Data Processing Layer**) και

[2] Technology Selection for Big Data and Analytical Applications των Denis Lehmann, David Fekete, Gottfried Vossen

το επίπεδο ανάλυσης (**Data Analytics Layer**). Η κατηγοριοποίηση αυτή θα μας βοηθήσει στις ακόλουθες υποενότητες να παρουσιάσουμε εν συντομία το επιστημονικό πεδίο των Big Data.



Σχήμα 1.2 Κατηγοριοποίηση Big Data τεχνολογιών

1.3.1 Παραγωγή των δεδομένων

Το επίπεδο αυτό χωρίζει τις διαθέσιμες επιλογές με βάση τη δομή των πηγών δεδομένων και την ταχύτητα με την οποία χρειάζεται να γίνεται η ανάλυσή τους.

Ξεκινώντας από τη δομή, οι πηγές δεδομένων μπορούν να χωριστούν σε τρεις τύπους: τις δομημένες (κλασικότερο παράδειγμα οι βάσεις δεδομένων), τις ημι-δομημένες (για παράδειγμα XML ή JSON αρχεία, emails κ.α.) και τις αδόμητες (για παράδειγμα κείμενο, εικόνες ή βίντεο, log αρχεία κ.α.).

Ο παράγοντας της ταχύτητας από την άλλη χωρίζει τις επιλογές σε δυο τρόπους: ανάλυση δεδομένων εν κινήσει (data-in-motion) και ανάλυση δεδομένων σε ανάπαυση (data-in-rest). Στην πρώτη περίπτωση έχουμε εφαρμογές που χρειάζεται να αναλύουν τα γεγονότα με το που συμβαίνουν (π.χ. οι ροές των μέσων κοινωνικής δικτύωσης που αναφέρθηκαν νωρίτερα ή οι χρηματιστηριακές συναλλαγές), ενώ στη δεύτερη εφαρμογές που λόγω της φύσης τους επιτρέπεται να δουλεύουν με ελαφρώς παρωχημένα δεδομένα και άρα μπορούν να τα αποθηκεύουν ανά τακτά διαστήματα (π.χ. κάθε βράδυ που η κίνηση σε ένα ηλεκτρονικό κατάστημα είναι χαμηλή) σε κάποια βοηθητική, δευτερεύουσα πηγή ώστε να κάνουν την ανάλυση τους σε αυτή χωρίς να επιβαρύνουν την πρωτογενή. Οι επιχειρήσεις καθορίζουν το πόσο γρήγορα πρέπει να αναλύονται τα δεδομένα και αυτή τους η απόφαση προφανώς καθορίζει την στρατηγική που πρέπει να ακολουθηθεί. Αν για παράδειγμα μιλάμε για ένα σύστημα προειδοποίησης επερχόμενων σεισμών η ανάλυση πρέπει να γίνεται με τη μικρότερη δυνατή καθυστέρηση. Αντιθέτως, αν έχουμε ένα ηλεκτρονικό κατάστημα και χρειάζεται να αναλύουμε τις πωλήσεις του ώστε να παράγουμε αναφορές σχετικά με τα ευπώλητα ή τις προτιμήσεις των χρηστών, αυτό μπορεί να γίνει και με offline δεδομένα σε ώρες χαμηλής κίνησης ακολουθώντας την τακτική που αναφέρθηκε προηγουμένως.

1.3.2 Απόκτηση των δεδομένων

Στο επίπεδο αυτό μας απασχολεί ο τρόπος με τον οποίο θα γίνει η έγχυση των δεδομένων στην υποδομή μιας Big Data εφαρμογής, με την ταχύτητα να είναι και πάλι ο παράγοντας που καθορίζει την επιλογή. Τα δεδομένα μπορεί να εγχύονται είτε με χρήση παραδοσιακών **Extract Transform Load (ETL)** τεχνικών, είτε σε πραγματικό χρόνο με χρήση τεχνικών **Complex Event Processing (CEP)** και / ή messaging συστημάτων.

Η μέθοδος **Extract Transform Load (ETL)** είναι μια ειδική περίπτωση μαζικής διαδικασίας που χρησιμοποιείται εδώ και δεκαετίες στο πεδίο του Data Warehousing. Όπως ορίζει και η ονομασία αποτελείται από τρία στάδια: **Extract** (εξαγωγή), **Transform** (μετατροπή) και **Load** (φόρτωμα). Τα δεδομένα λαμβάνονται από διάφορες ομογενείς ή ετερογενείς πηγές κατά το στάδιο της εξαγωγής, μετασχηματίζονται σε δομές (φορμάτ) κατάλληλες για να υποστούν ανάλυση κατά το στάδιο της μετατροπής και αποθηκεύονται τελικά στην αποθήκη δεδομένων κατά το στάδιο του φορτώματος. Επειδή η εξαγωγή των δεδομένων είναι μια χρονοβόρα διαδικασία, είναι σύνηθες οι τρεις φάσεις να εκτελούνται παράλληλα: τα δεδομένα εξάγονται σε δεσμίδες (batches), οι δεσμίδες προχωράνε στη φάση της επεξεργασίας και μόλις κάθε μια από αυτές επεξεργαστεί προχωράει στη φάση του φορτώματος στην

αποθήκη δεδομένων. Με αυτό τον τρόπο καμιά από τις τρεις φάσεις δεν μένει ανενεργή περιμένοντας κάποια άλλη, κάτι που συνεπάγεται τεράστιο κέρδος στην ταχύτητα με την οποία ολοκληρώνεται το σύνολο της διαδικασίας.

Τα ETL συστήματα είναι σχεδιασμένα ώστε να μπορούν να ενοποιούν δεδομένα προερχόμενα απο πολλές ετερογενείς πηγές, παρόχους και συστήματα που μπορούν να βρίσκονται ακόμα και σε διαφορετικά φυσικά μηχανήματα. Εργαλεία όπως το **Microsoft SQL Server Integration Services (SSIS)** και το **Pentaho Data Integration (PDI)**[3] επιτρέπουν για παράδειγμα την ενοποίηση δομημένων και αδόμητων πηγών (π.χ. βάσεις δεδομένων και αρχείων σε κάποιο file system), ενώ άλλα, όπως το **Apache Sqoop**[4], επιτρέπουν τη σύνδεση με κατανεμημένες αποθήκες δεδομένων όπως τα Hadoop Distributed File System (HDFS) και HBase.

Αντίθετα με τη μαζική επεξεργασία των ETL συστημάτων, οι **Complex Event Processing (CEP)** τεχνικές έχουν σαν στόχο την ανάλυση των δεδομένων σε πραγματικό χρόνο ώστε να παράγονται αποτελέσματα για τα δεδομένα με το που αυτά εμφανίζονται. Προκαθορισμένα events αναζητούνται στη ροή των δεδομένων, και το σύστημα χωρίς να απαιτεί την παρέμβαση χρήστη αποφασίζει αν αυτά τα δεδομένα πρέπει να ληφθούν υπόψη ή να φιλτραριστούν, αν πρέπει να αναλυθούν, αν πρέπει να εκτελέσει κάποια ενέργεια που καθορίζεται από την πληροφορία που φέρουν και τελικώς αν θα τα αποθηκεύσει στην τελική υποδομή. Αυτές οι τεχνικές εφαρμόζονται συνήθως για την ανίχνευση απάτης σε τραπεζικά συστήματα (π.χ. υπέρογκη απόσυρση χρημάτων από τραπεζικούς λογαριασμούς), την ανάλυση των κλικ ιστοσελίδων, την αξιολόγηση των posts σε μέσα κοινωνικής δικτύωσης κ.α. Εργαλεία που εφαρμόζουν CEP τεχνικές είναι, μεταξύ άλλων, τα **Apache Flume**[5] και **Apache Storm**[6].

Στο Σχήμα 1.2, ανάμεσα στα CEP συστήματα και τα παραδοσιακά ETL εργαλεία βρίσκονται τα messaging συστήματα. Αυτά δεν παρέχουν δυνατότητες για επεξεργασία ροών δεδομένων σε πραγματικό χρόνο· παίζουν απλά το ρόλο ουρών για events και μηνύματα που θα επεξεργαστούν από κάποιο CEP και εξασφαλίζουν πως δεν θα χαθούν δεδομένα κατά την επικοινωνία με εξωτερικά συστήματα.

1.3.3 Αποθήκευση των δεδομένων

Στο επίπεδο αυτό οι επιλογή αφορά τις τεχνολογίες που θα χρησιμοποιηθούν για να αποθηκευτούν τα δεδομένα στην Big Data υποδομή. Η διάκριση, όπως φαίνεται και στο Σχήμα 1.2, μπορεί να γίνει σε τρεις τύπους: κατανεμημένα συστήματα αρχείων, Not-Only SQL (NoSQL) αποθήκες δεδομένων και σχεσιακές

[3] Πληροφορίες για το Pentaho Data Integration εδώ: http://www.pentaho.com/product/data-integration
[4] Πληροφορίες για το Apache Sqoop εδώ: http://sqoop.apache.org/
[5] Πληροφορίες για το Apache Flume εδώ: https://flume.apache.org/
[6] Πληροφορίες για το Apache Storm εδώ: http://storm.apache.org/

βάσεις δεδομένων (RDBMs). Η κλιμάκωση, αν λόγοι απόδοσης απαιτούν κάτι τέτοιο, μπορεί να γίνει για τους δυο πρώτους τύπους οριζόντια (προσθήκη περισσότερων κόμβων) και για τον τρίτο οριζόντια αν μιλάμε για **Massively Parallel Processing (MPP) RDBMs** ή κάθετα (προσθήκη CPU ή / και RAM) αν μιλάμε **για Symmetric Multi Processing (SMP) RDBMs**.

Στα Symmetric Multi Processing RDBMs ανήκουν, μεταξύ άλλων, ο **Microsoft SQL Server** ή η **MySQL**, ενώ στα Massively Parallel Processing (MPP) RDBMs τα **Teradata**, **Netezza**, **Greenplum**, **Vertica** και **SAP Hana**. Η δυνατότητα για οριζόντια κλιμάκωση καθιστά τα τελευταία ιδανική επιλογή για Data Warehousing μεγάλου μεγέθους και ανάλυση μεγάλου όγκου δεδομένων, το κόστος τους όμως είναι συνήθως απαγορευτικό λόγω του εξειδικευμένου hardware και των αδειών χρήσης που απαιτούν. Όταν λοιπόν ο όγκος είναι πολύ μεγάλος, πολλά υποσχόμενες εναλλακτικές επιλογές είναι τα κατανεμημένα συστήματα αρχείων (π.χ. το **Hadoop Distributed File System, εφεξής HDFS**) και οι NoSQL τεχνολογίες (**Riak**, **MongoDB**, **HBase**, **Neo4J** κ.α.). Αυτές είναι δυνατό να υποστηρίξουν την προσθήκη περισσότερων κόμβων αν αυτό απαιτηθεί, υποστηρίζουν διαφορετικούς τύπους δομών δεδομένων, υπόσχονται υψηλές ταχύτητες και χαμηλό κόστος.

Οι John King και Roger Magoulas[7] το 2014 έκαναν μια έρευνα η οποία έδειξε πως το 42% των αναλυτών δεδομένων που συμμετείχαν χρησιμοποιούσαν SQL για την αποθήκευση των Big Data τους, ενώ μόνο το 23% HDFS. Παρόμοια έρευνα της Jaspersoft[8] έδειξε πως τα RDBMs χρησιμοποιούνται σε ποσοστό 56%, η MongoDB σε ποσοστό 23%, τα MPP RDBMs σε ποσοστό 14% και το HDFS σε ποσοστό 12%. Συμπεραίνουμε λοιπόν πως, ακόμα και στην εποχή των Big Data, οι σχεσιακές βάσεις δεδομένων παίζουν πρωταρχικό ρόλο στην ανάλυση και δεν έχουν αντικατασταθεί απο άλλες τεχνολογίες έστω και αν οι εναλλακτικές επιλογές, όπως μόλις παρουσιάστηκε, είναι αρκετές και πολλά υποσχόμενες.

1.3.4 Επεξεργασία των δεδομένων

Στο επίπεδο αυτό ανήκουν οι τεχνολογίες που είναι υπεύθυνες για την εκτέλεση ενεργειών όπως η ανάγνωση, η εγγραφή και η διαγραφή δεδομένων. Η επεξεργασία μπορεί να γίνεται σε δομημένα δεδομένα αποθηκευμένα σε μια βάση (database processing) ή σε ημιδομημένα / αδόμητα δεδομένα που έχουν τη μορφή αρχείων και επεξεργάζονται από κατανεμημένα συστήματα αρχείων ή NoSQL αποθήκες δεδομένων (file-based processing). Η δεύτερη περίπτωση χωρίζεται περαιτέρω σε τεχνικές μαζικής επεξεργασίας, επεξεργασίας ροής δεδομένων και συνδυασμούς αυτών. Σε αντίθεση με την επεξεργασία δεδομένων από βάση –η λογική της οποίας είναι πολύ οικεία σε όλους μας και δε χρήζει ανάλυσης, η

[7] John King and Roger Magoulas. 2013 Data Science Salary Survey: Tools, Trends, WhatPays (and What Doesn't) for Data Professionals. O'Reilly Strata, 2014.

8 Olov Schelen, Ahmed Elragel, and Moutaz Haddara. A Roadmap for Big-Data Research and Education. Technical report, 2015.

επεξεργασία αρχείων δεν μπορεί εύκολα να κατηγοριοποιηθεί καθώς η δομή και η μορφή των αρχείων προφανώς ποικίλλει και συνήθως απαιτεί την υλοποίηση κώδικα απο όποιον επιθυμεί να αναλύσει τέτοιου είδους πληροφορία.

Η μαζική επεξεργασία (**Batch Processing**) και κατά συνέπεια οι σχετιζόμενες μηχανές μαζικής επεξεργασίας (**Batch Processing Engines, BPEs**) δρουν εξ ορισμού σε δεδομένα σε ανάπαυση (data-in-rest). Αλγόριθμοι χωρίζουν τον αρχικό όγκο σε δεσμίδες (chunks) που επεξεργάζονται παράλληλα από πολλούς κόμβους οι οποίοι παράγουν τα «ενδιάμεσα» αποτελέσματα που θα ενοποιηθούν στα τελικά, πριν φορτωθούν στην Big Data υποδομή. Ο τρόπος που τα δεδομένα θα χωριστούν σε δεσμίδες, το πως θα κατανεμηθούν στους κόμβους και το πως θα παραχθεί το τελικό αποτέλεσμα είναι δουλειά του εκάστοτε προγραμματιστή. Δημοφιλές παράδειγμα είναι το μοντέλο **MapReduce**: ο προγραμματιστής καθορίζει μια συνάρτηση «map» η οποία επεξεργάζεται ένα ζεύγος κλειδιού-τιμής (key-value pair) για να δημιουργηθεί το σύνολο ενδιάμεσων key-value ζευγών που θα μοιραστούν προς επεξεργασία στους κόμβους του συστήματος, και μια συνάρτηση «reduce» που συγχωνεύει όλες τις ενδιάμεσες τιμές που μοιράζονται το ίδιο κλειδί.

Η επεξεργασία ροών (**Stream Processing**) δρα σε δεδομένα εν κινήσει (data-in-motion) και χρησιμοποιείται όταν απαιτείται η άμεση παραγωγή αποτελεσμάτων. Το να υλοποιηθεί ένα τέτοιο σύστημα είναι αρκετά σύνθετο, η ανάγκη όμως για αξιολόγηση των posts ενός μέσου κοινωνικής δικτύωσης σε πραγματικό χρόνο χωρίς την παρέμβαση χρήστη ή για πρόβλεψη σεισμών, που αναφέρθηκαν και στα προηγούμενα ως παραδείγματα, το καθιστά συχνά μονόδρομο. Εργαλείο που μπορεί να χρησιμοποιηθεί ως **Stream Processing Engine (SPE)** είναι το Apache Storm.

Τέλος, ο συνδυασμός των θετικών στοιχείων τόσο της μαζικής επεξεργασίας όσο και της επεξεργασίας ροών οδήγησε στα λεγόμενα **Unified Processing Engines (UPEs)** που σε –σχεδόν– πραγματικό χρόνο επεξεργάζονται τόσο δεδομένα σε ανάπαυση όσο και δεδομένα εν κινήσει. Ένα πολύ διαδεδομένο εργαλείο ενοποιημένης επεξεργασίας είναι το Apache Spark[9].

1.3.5 Ανάλυση των δεδομένων

Στο τελευταίο επίπεδο βρίσκεται όλη η ουσία των Big Data. Τα δεδομένα αναλύονται με στόχο την εύρεση άγνωστων συσχετίσεων και κρυφών patterns που εφόσον κατανοηθούν θα οδηγήσουν σε επιχειρηματικά οφέλη όπως νέες ευκαιρίες εσόδων, αποτελεσματικότερο μάρκετινγκ, καλύτερη εξυπηρέτηση πελατών, αποδοτικότερη λειτουργία κ.α. Οι διαθέσιμες τεχνολογίες χωρίζονται εδώ στην περιγραφική Business Intelligence ανάλυση (π.χ. OLAP) και στις προχωρημένες τεχνικές για πρόβλεψη μέσω machine learning.

[9] Πληροφορίες για το Apache Spark εδώ: http://spark.apache.org/

Ένα σύστημα online αναλυτικής επεξεργασίας **(Online Analytical Processing, OLAP)** είναι ένα διαδραστικό σύστημα που επιτρέπει στον αναλυτή να βλέπει διαφορετικά σύνολα πολυδιάστατων δεδομένων online· άμεσα δηλαδή, χωρίς να χρειάζεται να περιμένει πολύ για να δεί τα αποτελέσματα του ερωτήματός του. Οι αρχικές εκδόσεις πολλών εργαλείων OLAP υπέθεταν οτι τα δεδομένα βρίσκονται στη μνήμη, κάτι λογικό όταν μιλάμε για μικρές ποσότητες δεδομένων αφού η ανάλυσή τους σε μια τέτοια περίπτωση θα μπορούσε να γίνει μέχρι και από εφαρμογές λογιστικών φύλλων σαν το Excel. Ωστόσο, σε πολύ μεγάλο όγκο δεδομένων το OLAP απαιτεί την ύπαρξη μιας βάσης δεδομένων και υποστήριξη απο αυτή ώστε να επιτευχθεί αποτελεσματικότερη προεπεξεργασία των πληροφοριών και online επεξεργασία των ερωτημάτων. Έτσι λοιπόν πλέον, υπάρχουν διαθέσιμα πολλά OLAP προϊόντα, κάποια ως συμπληρωματικά εργαλεία σε προϊόντα βάσεων δεδομένων όπως το **Microsoft SQL Server Analysis Services (SSAS)** ή το **Oracle OLAP** και άλλα ως αυτόνομα εργαλεία όπως το **Pentaho Mondrian**.

Σήμερα, το παραγωγικό περιβάλλον των επιχειρήσεων είναι τυπικά οργανωμένο γύρω από ένα OLTP σύστημα. Με τον όρο online επεξεργασία συναλλαγών (**Online Transaction Processing, OLTP**) περιγράφουμε την κλάση υπολογιστικών συστημάτων που προσανατολίζονται στην ευκολότερη και ταχύτερη διαχείριση συναλλαγών. Πληροφορία διαβάζεται απο κάποια πηγή δεδομένων (τυπικά ένα DBMS), επεξεργάζεται και οι αλλαγές αποθηκεύονται και πάλι στη βάση δεδομένων. Παραδείγματα OLTP συστημάτων αποτελούν τα ATM των τραπεζών, τα συστήματα εισαγωγής παραγγελιών, λιανικής πώλησης και οικονομικών συναλλαγών. Τα OLTP συστήματα όμως δεν είναι κατάλληλα για την εκτέλεση ανάλυσης. Αυτό υποχρεώνει τις επιχειρήσεις να εφαρμόζουν τεχνικές ETL ώστε τα δεδομένα να μεταφέρονται από τις παραγωγικές βάσεις δεδομένων σε άλλες που θα παίξουν το ρόλο του Data Warehouse επί του οποίου θα γίνει η ανάλυση. Αυτό είναι εξαιρετικά μη αποδοτικό καθώς απαιτεί περιοδικό φόρτωμα από ένα DBMS σε κάποιο άλλο, μετατρέπει την ανάλυση από διαδικασία πραγματικού χρόνου σε batch διαδικασία και γίνεται ακόμα χειρότερο όταν μιλάμε για σχεσιακές βάσεις δεδομένων, οι ACID ιδιότητες των συναλλαγών των οποίων συνεπάγονται έξτρα καθυστερήσεις λόγω των κλειδωμάτων που απαιτούνται σε επίπεδο πινάκων ή εγγραφών. Αναγκαστικά η μεταφορά προγραμματίζεται να εκτελεστεί κάποια στιγμή που η κίνηση είναι χαμηλή (π.χ. τις βραδινές ώρες) και άρα, εκτός όλων των άλλων, η πλατφόρμα ανάλυσης καταλήγει να ενεργεί σε παρωχημένα αντίγραφα των δεδομένων. Τα παραπάνω είναι ένα από τα μεγαλύτερα προβλήματα των OLAP τεχνικών και μια από τις μεγαλύτερες προκλήσεις που έχει να αντιμετωπίσει η κοινότητα των Big Data.

Οι προχωρημένες τεχνικές τώρα, μπορούν κατά τον Vijay Srinivas Agneeswaran[10] να διακριθούν σε τρεις γενιές:

[10] Why Look Beyond Hadoop Map-Reduce: http://www.ftpress.com/articles/article.aspx?p=2215090&seqNum=2

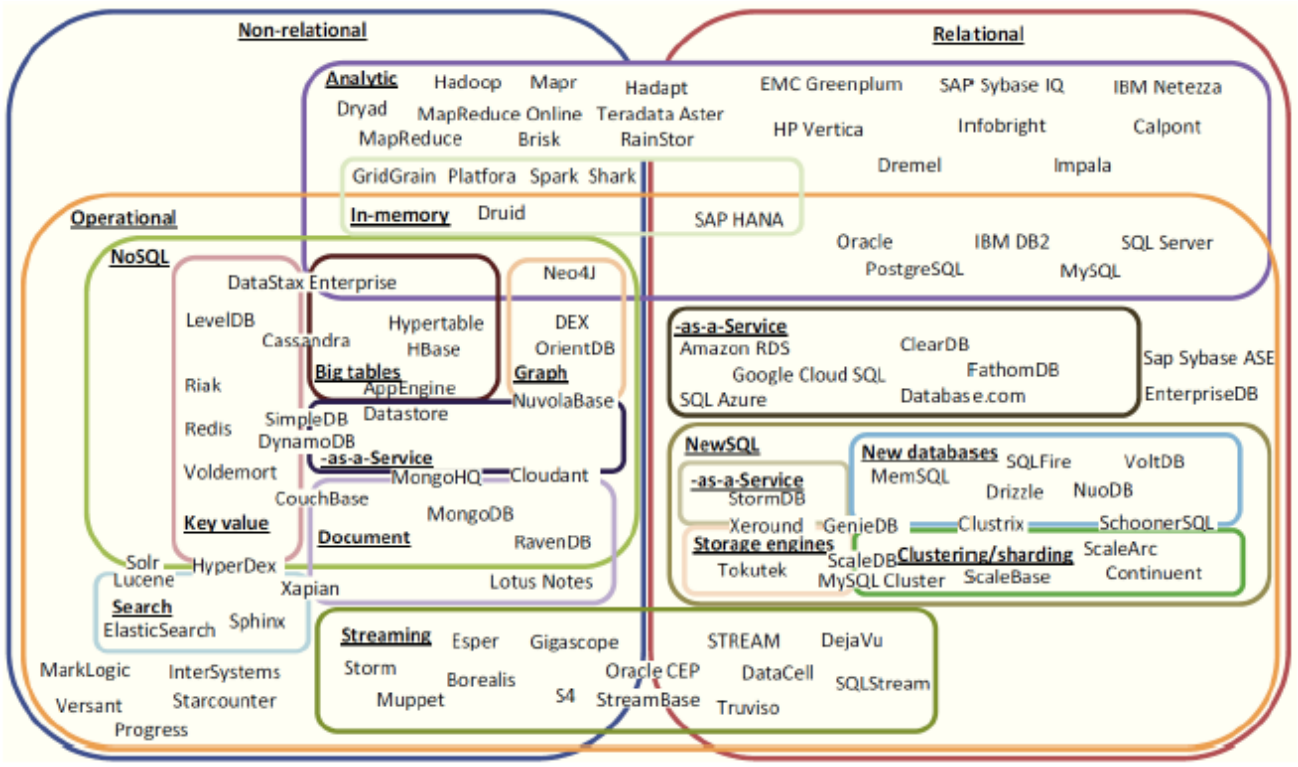
- **Machine learning πρώτης γενιάς (1GML):** Τα δεδομένα βρίσκονται στη μνήμη ενός μόνο κόμβου κάτι που συνεπάγεται δυνατότητα μόνο για κάθετη κλιμάκωση και άρα καθιστά την επέκταση σε Big Data σενάρια πρακτικά αδύνατη. Τα εργαλεία αυτής της γενιάς (**R**, **RapidMiner**, **Knime**, **WEKA** κ.α.) αναπτύχθηκαν πριν το Hadoop και συχνά αναφέρονται ως «παραδοσιακά εργαλεία ανάλυσης». Συνήθως, για βελτίωση της απόδοσης, παρέχονται connectors για την ανάγνωση και την εγγραφή σε HDFS με τα εργαλεία να εκτελούν μόνο την ανάλυση: τα δεδομένα διαβάζονται από το file system, αναλύονται και φορτώνονται εκ νέου σε αυτό.
- **Machine learning δεύτερης γενιάς (2GML):** Τα εργαλεία αυτής της γενιάς (π.χ. το **Apache Mahout (MapReduce)**[11]) επεκτείνουν τη λογική των 1GML και επιτρέπουν την κατανεμημένη επεξεργασία των δεδομένων από Hadoop clusters. Συχνά αναφέρονται ως εργαλεία **«over Hadoop»**, το γεγονός όμως ότι δεν είναι εύκολο να υλοποιηθούν MapReduce αλγόριθμοι για machine learning τα καθιστούν συνήθως μη ανταγωνιστικά.
- **Machine learning τρίτης γενιάς (3GML):** Αυτό ακριβώς το πρόβλημα προσπαθούν να λύσουν τα εργαλεία τρίτης γενιάς (**«beyond Hadoop»**), που εφαρμόζουν σύνθετες τεχνικές κατανεμημένης επεξεργασίας ή επεξεργασία σε βάσεις δεδομένων ώστε να ξεπεράσουν τους περιορισμούς του MapReduce. Παραδείγματα τέτοιων εργαλείων είναι τα Mahout (Spark / H2O / Flink), MLlib, SAMOA, MADlib κ.α.

Η έρευνα των John King και Roger Magoulas που αναφέρθηκε νωρίτερα, έδειξε πως το 71% όσων συμμετείχαν χρησιμοποιούσαν (και) SQL βάσεις για την ανάλυση των δεδομένων τους, ενώ το αμέσως επόμενο δημοφιλέστερο εργαλείο ήταν το R με ποσοστό 43%. Το Mahout βρισκόταν λίγο κάτω από το 7%. Μπορεί δηλαδή το Hadoop και οι NoSQL βάσεις να έλυναν τα προβλήματα αποθήκευσης μεγάλου όγκου δεδομένων, δεν μπορούσαν όμως να καλύψουν τις ανάγκες των χρηστών όσον αφορά στην ανάλυσή τους.

Καταλαβαίνουμε, τώρα που έχουν παρουσιαστεί και τα πέντε επίπεδα, το πόσο σύνθετος μπορεί να είναι ο σχεδιασμός ενός Big Data συστήματος. Τα επίπεδα και τα στοιχεία του κάθε επιπέδου τα οποία θα εμπλακούν στην τελική λύση καθορίζονται από το είδος της ανάλυσης που θέλουμε να υποστηρίξουμε, τη δομή των δεδομένων εισόδου, τους επιθυμητούς χρόνους απόκρισης, το αν η ανάλυση πρέπει να γίνει σε πραγματικό χρόνο ή offline και πολλά άλλα. Οι απαντήσεις σε καθένα από αυτά τα ερωτήματα καθορίζουν ένα σύνολο από τεχνολογίες που μπορούν να υποστηρίξουν τις ανάγκες μας (αναφερθήκαμε ήδη σε πολλές, μπορούμε όμως να ανατρέξουμε και στο Σχήμα 1.3 που τις παρουσιάζει σε ομάδες), και από αυτές πρέπει να επιλέξουμε αυτές που τελικά θα χρησιμοποιηθούν λαμβάνοντας υπόψη το κατά πόσο είναι

[11] Περισσότερα για το Apache Mahout εδώ: http://mahout.apache.org/. Το εργαλείο πρόσφατα πέρασε απο την 2GML έκδοσή του στην 3GML επιτρέποντας πλέον την επεξεργασία με μηχανές όπως τα Spark, Flink και H2O και για αυτό παρουσιάζεται ως δυνατή επιλογή και στις δυο γενιές εργαλείων.

συμβατές μεταξύ τους, την πολυπλοκότητά τους, το κόστος που μπορεί να έχουν λόγω απαιτήσεων σε υλικό ή άδειες χρήσης και πολλά άλλα.



Σχήμα 1.3 Το πολύπλοκο και πολυποίκιλο σύμπαν των Big Data τεχνολογιών

1.3.6 Το project LeanBigData

Το πεδίο των Big Data μπορεί να βρίσκεται ακόμα στη βρεφική του ηλικία, δεν λείπουν όμως οι ενδιαφέρουσες και φιλόδοξες προσπάθειες για υλοποίηση λύσεων που θα ξεπεράσουν το «No One Size Fits All» φαινόμενο –που αναφέρθηκε στα προηγούμενα, θα καλύψουν μεγαλύτερο εύρος του φάσματος και λύνοντας τα προβλήματα που ο χώρος αντιμετωπίζει θα προχωρήσουν την έρευνα αλλά και την τεχνολογία ένα βήμα μπροστά. Μια από αυτές τις πολύ ενδιαφέρουσες προσπάθειες είναι το project **LeanBigData**[12].

Οι πλατφόρμες που κυκλοφορούν σήμερα είναι ανομοιόμορφες, περιορίζονται σε ένα υποσύνολο μόνο του σύμπαντος των Big Data αδυνατώντας να αντιμετωπίσουν οτιδήποτε άλλο, αναγκάζουν σε χρήση ETL τεχνικών ώστε τα διαφορετικής φύσεως (δομής) δεδομένα να επεξεργαστούν και να φορτωθούν σε ένα Data Warehouse από όπου τελικά θα αναλυθούν –με το κόστος που είδαμε πως αυτές οι τεχνικές έχουν σε απόδοση αλλά και ακρίβεια, δεν μπορούν να εξασφαλίσουν ατομικότητα ή συνάφεια κατά την εκτέλεση

[12] Το επίσημο site του LeanBigData εδώ: http://leanbigdata.eu/

ενεργειών επί των εμπλεκόμενων πηγών και είναι αναποτελεσματικές όσον αφορά στην κατανάλωση πόρων.

Το LeanBigData προσανατολίζεται στην υλοποίηση μιας εύκολα κλιμακούμενης και αποδοτικής λύσης, ικανής για επεξεργασία τόσο δεδομένων σε ανάπαυση όσο και δεδομένων εν κινήσει, που θα λύνει όλα τα παραπάνω προβλήματα. Συνδυάζοντας **NoSQL**, **SQL** και **Complex Event Processing (CEP)** τεχνικές, με τη δυνατότητα για παράλληλη και κατανεμημένη επεξεργασία ερωτημάτων από ανεξάρτητα μεταξύ τους περιβάλλοντα, το σύστημα θα μπορεί να κλιμακώνεται εύκολα προς εξυπηρέτηση πολύ μεγάλου όγκου. Η πλατφόρμα θα είναι ιδιαίτερα αποτελεσματική μέσω της υιοθέτησης πληθώρας καινοτόμων πρακτικών που θα βελτιστοποιούν τη χρήση hardware, δικτύου, μέσων αποθήκευσης, διαχείρισης συναλλαγών, garbage collection κ.α. Οι τελικοί χρήστες θα έχουν τη δυνατότητα να αντιλαμβάνονται εγκαίρως αστοχίες στη σχεδίαση της ανάλυσης και να επεμβαίνουν διορθωτικά χωρίς να χρειάζεται να περιμένουν ώρες ή και μέρες για να ολοκληρωθεί ο εκτελούμενος κύκλος πριν ξεκινήσουν από την αρχή.

Στα πλαίσια του LeanBigData θα δημιουργηθεί ένα καινοτόμο key-value, column-oriented data store που θα λειτουργεί ως η υποκείμενη πλατφόρμα διαχείρισης. Ένα SQL επίπεδο θα λειτουργεί σαν ξεχωριστό επίπεδο που θα επιτρέπει να εκτελούνται ερωτήματα αποθηκευμένα στο NoSQL επίπεδο. Ένα CEP επίπεδο θα δίνει τη δυνατότητα επεξεργασίας δεδομένων ροής. Οι τελεστές πινάκων στο επίπεδο CEP θα έχουν πρόσβαση στα αποθηκευμένα δεδομένα ώστε να τα ενημερώνουν ή να εισάγουν νέα, ενώ θα μπορούν να τα συσχετίζουν και με τα δεδομένα ροής. Οι εφαρμογές που θα χρησιμοποιούν την πλατφόρμα ανάλογα με τις ανάγκες τους θα έχουν πρόσβαση σε οποιοδήποτε απο τα παρεχόμενα APIs επιθυμούν (key-value, SQL, CEP). Προκειμένου να υπάρξει συνοχή ανάμεσα σε όλα τα επίπεδα διαχείρισης θα ενσωματώνεται κλιμακούμενη διαχείριση συναλλαγών στο key-value επίπεδο αποθήκευσης. Η πλατφόρμα θα εξασφαλίζει πως όλα τα επίπεδα θα έχουν πρόσβαση σε μια συνεπή εικόνα των δεδομένων διαμέσω ACID συναλλαγών.

Σήμερα, δυο είναι οι κυρίαρχες τάσεις στο πεδίο των scalable analytics και των Data Warehouses. Από τη μια υπάρχουν projects σαν τα **BigQuery**, **Tenzing** και **Hive** που ακολουθούν την προσέγγιση MapReduce και προσπαθούν να την βελτιώσουν ώστε να έρθει εγγύτερα στην SQL. Η προσέγγιση αυτή υποχρεώνει σε χρήση ETL τεχνικών, καθώς δεν μπορούν να εκτελεστούν σε ένα OLTP σύστημα, που προκαλούν όπως προαναφέρθηκε μείωση της απόδοσης και άσκοπη δημιουργία διπλότυπων των δεδομένων. Η δεύτερη τάση, με εμπνευστές τις εταιρίες **Greenplum** και **Vertica**, προσπαθεί να επωφεληθεί των αποτελεσμάτων της έρευνας στο πεδίο των παράλληλων και κατανεμημένων συστημάτων διαχείρισης βάσεων δεδομένων για την εκτέλεση ερωτημάτων παράλληλα πάνω από ένα κατανεμημένο shared-nothing σύστημα. Και αυτές οι λύσεις όμως αντιμετωπίζουν προβλήματα καθώς δεν υποστηρίζουν ενημερώσεις (updates) ή τις υποστηρίζουν με πολύ χαμηλό rate, απαιτώντας μάλιστα εξειδικευμένο data store παραμετροποιημένο ειδικά για ανάλυση γεγονός που υπαινίσσεται επίσης την ανάγκη για χρήση ETL. Το LeanBigData θα

υιοθετήσει μια υβριδική προσέγγιση παρέχοντας παράλληλα και κατανεμημένα ερωτήματα απευθείας στην παραγωγική βάση, αποφεύγοντας το κόστος της ETL μεθόδου. Η SQL engine θα έχει παραλληλισμό και σε intra-query και σε intra-operation επίπεδο. Για να καταστεί δυνατή η διαχείριση αδόμητων δεδομένων, θα αναπτυχθεί ένα framework συλλογής με δηλωτική προσέγγιση βασισμένο στην τεχνολογία compiler-compiler (για παράδειγμα JavaCC). Αυτό το framework θα δίνει τη δυνατότητα καθορισμού του τρόπου σύνταξης των αδόμητων δεδομένων και του πως θα γίνεται η μετατροπή τους σε δομημένα προτού αποθηκευτούν στο key-value store για να επεξεργαστούν είτε από το ίδιο το key-value store είτε απο την SQL engine (η τελευταία θα βασιστεί στην **Apache Derby**, μια υπάρχουσα, αρκετά δημοφιλή, κεντρικοποιημένη λύση).

Με χρήση πληθώρας μοντέρνων τεχνικών και τεχνολογιών το project LeanBigData φιλοδοξεί λοιπόν να καλύψει όλα τα παραπάνω και να καινοτομήσει σε τρεις κατευθύνσεις: 1) την αποδοτική κλιμάκωση, 2) την ενοποίηση της OLTP επεξεργασίας με την OLAP επεξεργασία και 3) την end-to-end υποστήριξη αναλυτών και λοιπών χρηστών της πλατφόρμας. Αυτές οι κατευθύνσεις αναλύονται στα ακόλουθα.

1.3.7 Αποδοτική Κλιμάκωση

Οι τεχνικές που σήμερα εφαρμόζονται για την επεξεργασία μεγάλου όγκου δεδομένων είναι εξαιρετικά μη αποδοτικές καθώς καταναλώνουν τεράστιες ποσότητες πόρων και οδηγούν σε μεγάλα κόστη ιδιοκτησίας (Total Cost of Ownership, TOC). Για να ποσοτικοποιηθεί αυτό αρκεί να αναφερθεί πως σύμφωνα με έρευνες τα δημόσια cloud συστήματα σήμερα καταναλώνουν το 2% της ηλεκτρικής ενέργειας που παράγεται στις ΗΠΑ και το 1,3% παγκοσμίως. Ένας από τους βασικούς στόχους του LeanBigData λοιπόν, είναι να βελτιωθεί αυτό το πρόβλημα μέσω της υιοθέτησης καινοτόμων πρακτικών που, εν συντομία, είναι οι εξής:

- **Αποδοτική παράλληλη επεξεργασία στο NoSQL επίπεδο:** Οι πόροι που καταναλώνονται από τους data managers κατά την προσπάθειά τους να ελέγξουν τον τρόπο που πολλά threads επεξεργάζονται δεδομένα είναι ένα από τα προβλήματα που αντιμετωπίζουν τα σημερινά συστήματα. Για παράδειγμα, το HBase (που χρησιμοποιείται κατά κόρον από το Facebook και το Yahoo) όταν λειτουργεί ταυτόχρονα για ανάγνωση και επεξεργασία δεδομένων καταναλώνει το 50-80% της διαθέσιμης CPU μόνο και μόνο για τον έλεγχο συγχρονισμού των threads. Το LeanBigData θα αντιμετωπίσει αυτό το πρόβλημα μέσω δυναμικού partitioning των πόρων –buffers, δομών δεδομένων και μετα-δεδομένων– που ο κάθε manager απαιτεί.
- **Αποδοτική διαχείριση δικτυακών πόρων:** Άλλη μια πρόκληση που έχουν να αντιμετωπίσουν τα σύγχρονα συστήματα είναι η υψηλή κατανάλωση δικτυακών πόρων. Και εδώ, παίρνοντας σαν παράδειγμα την επεξεργασία που εκτελεί ένας CEP data manager, 50-80% της συνολικής CPU

καταναλώνεται για την διαχείριση των αλληλεπιδράσεων με το δίκτυο. Τα IP πακέτα που σχετίζονται με κάθε υπό επεξεργασία εγγραφή φτάνουν στο CEP επίπεδο, γίνονται deserialize, επεξεργάζονται (το κόστος επεξεργασίας είναι χαμηλό) και στέλνονται στο επόμενο CEP βήμα επιβαρύνοντας εκ νέου τους δικτυακούς πόρους. Το LeanBigData υιοθετώντας τις τάσεις στο υψηλής ταχύτητας Ethernet (π.χ. 10-40 GigE) θα υλοποιήσει μια καινοτόμα λύση με τα πακέτα να αντιγράφονται κατευθείαν από τη μνήμη ενός κόμβου στη μνήμη του επόμενου, ενώ παράλληλα μέσω διαχείρισης των connections θα επιτευχθεί ένα είδος pooling για τον κάθε κόμβο ώστε να μειωθεί η επιβάρυνση που δημιουργεί η ανάγκη για δημιουργία τεράστιου αριθμού συνδέσεων.

- **Αποδοτικό Garbage Collection:** Πολλά Big Data συστήματα είναι υλοποιημένα σε Java ή άλλες γλώσσες που γίνονται compile σε bytecode και εκτελούνται από κάποιο JVM. Το JVM περιοδικά τρέχει αλγόριθμους για να αναγνωρίσει μη χρησιμοποιούμενα κομμάτια μνήμης και να τα ελευθερώσει. Η διαδικασία αυτή (Garbage Collection, GC) είναι ιδιαίτερα επιβαρυντική για ένα σύστημα που, επεξεργαζόμενο μεγάλο όγκο δεδομένων, δημιουργεί και καταστρέφει διαρκώς αντικείμενα στη μνήμη. Μάλιστα, κατά τη διάρκεια του garbage collection το σύστημα σταματά να εκτελεί οποιαδήποτε άλλη ενέργεια (stop the world event) με ότι αυτό μπορεί να συνεπάγεται όταν η ανάγκη για άμεση απόκριση είναι κάτι παραπάνω από επιτακτική. Το LeanBigData προσανατολίζεται στο να επιλύσει τα παραπάνω προβλήματα είτε μέσω διαρκούς garbage collection (και άρα πολύ μικρότερο αριθμό αντικειμένων προς έλεγχο), είτε μέσω της επαναχρησιμοποίησης των αντικειμένων σε επίπεδο server που θα έχει σαν αποτέλεσμα την κατάργηση του garbage collection.
- **Αποδοτική Είσοδος/Έξοδος (Input/Output, I/O):** Τα σύγχρονα λειτουργικά συστήματα είναι σχεδιασμένα έτσι ώστε να υποστηρίζουν μικρό αριθμό πυρήνων και I/O συσκευών. Το περιορισμένο αυτό I/O bandwidth αποτελεί προφανώς πρόβλημα για ένα Big Data σύστημα που καλείται να γράψει μέσω του λειτουργικού συστήματος στις αποθήκες δεδομένων τεράστιο όγκο εγγραφών. Το LeanBigData θα περιορίσει κατά το δυνατό τις I/O εγγραφές –περιορίζοντας για παράδειγμα τις αλληλεπιδράσεις με το τοπικό file system– ώστε η τελική λύση να μπορεί χωρίς πρόβλημα να κλιμακώνεται όσο ο αριθμός των πυρήνων αυξάνεται.

1.3.8 Ενοποίηση της OLTP επεξεργασίας με την OLAP επεξεργασία

Για να αποφευχθούν οι ανεπάρκειες και να ξεπεραστούν οι καθυστερήσεις που επιφέρει η χρήση παραδοσιακών ETL τεχνικών, το LeanBigData θα ενσωματώσει στις παραγωγικές OLTP βάσεις δεδομένων λειτουργίες ανάλυσης. Με αυτό τον τρόπο θα αποφευχθεί το τωρινό κόστος που επιφέρει η ETL προσέγγισης για την αντιγραφή της παραγωγικής βάσης δεδομένων σε μια βάση για ανάλυση. Η

καινοτομία αυτή εκμεταλλεύεται τα αποτελέσματα του project **CumuloNimbo FP7** στο οποία αναπτύχθηκαν εξαιρετικά κλιμακούμενες ACID συναλλαγές, με μεγάλη σημασία στην ενσωμάτωση ενός OLAP συστήματος σε μια παραγωγική OLTP βάση. Το LeanBigData θα βελτιώσει την αποτελεσματικότητα αυτού του μηχανισμού συναλλαγών.

Επιπλέον, το LeanBigData θα χρησιμοποιήσει το κλιμακούμενο key-value επίπεδο για την επεξεργασία ad-hoc (και όχι περιοδικών) SQL ερωτημάτων. Θα εκμεταλλευτεί την διαχείριση συγχρονισμού της key-value βάσης για την εκτέλεση intra-query και intra-operator παραλληλισμού, τρέχοντας ένα ερώτημα σε πολλαπλούς κόμβους. Αυτό θα επιταχύνει την επεξεργασία μεγάλων ερωτημάτων, με τρόπο παρόμοιο με εκείνο των σύγχρονων Data Warehouses, όπως το **EMC Greenplum**. Σε αντίθεση όμως με αυτά τα συστήματα, τα ερωτήματα θα εκτελούνται πάνω στα δεδομένα της παραγωγικής βάσης και όχι σε ξεχωριστό, παρωχημένο αντίγραφο των δεδομένων. Όλα αυτά καθίστανται εφικτά χάρις στην καινοτομία της υποστήριξης multi-versioning στο επίπεδο των συναλλαγών, κάτι που θα εξασφαλίσει ότι τα ερωτήματα της ανάλυσης δεν θα παρεμβαίνουν στις OLTP εργασίες, σε αντίθεση με ότι συμβαίνει στα υπάρχοντα συστήματα.

Τέλος, το LeanBigData θα επιτρέψει την εύκολη ανάπτυξη κατανεμημένων CEP ερωτημάτων πάνω στο ίδιο σύνολο δεδομένων κατά την ώρα που τα δεδομένα ενημερώνονται. Σαν παράδειγμα μπορούμε να πάρουμε τις εταιρίες Τηλεπικοινωνιών οι οποίες πραγματοποιούν χρεωστικές λειτουργίες πάνω στις εγγραφές στοιχείων κλήσεων (Call Detail Records, CDRs). Για να μπορούν να συγκεντρώνονται και να συσχετίζονται εκατομμύρια τέτοιες εγγραφές ανά δευτερόλεπτο, είναι απαραίτητο να υπάρχουν υψηλά επίπεδα παραλληλισμού στο CEP επίπεδο με χρήση intra-queries και intra-operators. Αυτό επετεύχθη στο project **Stream FP7** που επέφερε την ανάπτυξη του **StreamCloud CEP** και στα project **S4** και **Storm** που ακολούθησαν. Το LeanBigData θα επιτρέψει διαρκείς υπολογισμούς αποτελεσμάτων καθώς τα δεδομένα ενημερώνονται, αντί να εκτελεί ερωτήματα ανάλυσης ανά σταθερά διαστήματα (π.χ. κάθε μήνα). Αυτό φυσικά θα βελτιώσει το χρόνο απόκρισης, αφού επί της ουσίας θα μετατρέψει μια batch διαδικασία σε διαδικασία πραγματικού χρόνου.

Στο CEP επίπεδο θα υπάρχει ένα ευρύ σύνολο αισθητήρων και συλλεκτών δεδομένων ώστε να παρέχεται ένα πλήρες framework συλλογής χρήσιμων για την πλατφόρμα πληροφοριών. Οι αισθητήρες θα εκμεταλλεύονται τους καταχωρητές stat της INTEL CPU και θα παρέχουν λεπτομερείς πληροφορίες σχετικά με την απόδοση και τη χρήση των πόρων σε όλα τα επίπεδα (εφαρμογή, λειτουργικό σύστημα, εικονικά μηχανήματα, server logs κ.α.). Θα αναπτυχθεί επίσης ένα σύνολο καινοτόμων ενεργειακών αισθητήρων οι οποίοι θα παρέχουν λεπτομερείς πληροφορίες για την απόδοση των ερωτημάτων κάτι που θα βοηθήσει στη βελτιστοποίηση των κέντρων δεδομένων. Αυτή η υποδομή πέρα από το monitoring προς τον τελικό

χρήστη θα χρησιμοποιηθεί και από την ίδια την πλατφόρμα του LeanBigData με στόχο την ελαστικότητα και την εξισορρόπηση φορτίου.

1.3.9 End-to-End Υποστήριξη Χρηστών

Το LeanBigData θα υποστηρίζει την γραφική απεικόνιση των αποτελεσμάτων της ανάλυσης κατά τη διάρκεια εκτέλεσης των analytical ερωτημάτων, χωρίς να χρειάζεται να τα περιμένει να ολοκληρωθούν. Οι χρήστες θα μπορούν με αυτό τον τρόπο να βγάλουν γρήγορα συμπεράσματα σχετικά με το αν οι αρχικές τους υποθέσεις ήταν προς την σωστή κατεύθυνση ή όχι και να δράσουν ανάλογα αφήνοντας την ανάλυση – που μπορεί να απαιτεί ώρες ή ακόμα και μέρες– να συνεχιστεί ή επεμβαίνοντας διορθωτικά. Αυτό, συγκρινόμενο με την εναλλακτική του να περιμένουν την ανάλυση να ολοκληρωθεί για να δουν πως δεν είχε σχεδιαστεί σωστά και άρα πρέπει να διορθωθεί και να εκτελεστεί εκ νέου, θα εξοικονομήσει τόσο χρόνο όσο και πόρους.

Η επιλογή του τρόπου γραφικής απεικόνισης και του συνόλου των δεδομένων τα οποία αυτή θα περιλαμβάνει θα γίνεται με τρόπο δυναμικό ώστε να μη δένει το χρήστη με προκαθορισμένες επιλογές που μπορεί να μην ταιριάζουν σε αυτά που θέλει να δει. Σαν παράδειγμα μπορούμε να δώσουμε την περίπτωση ενός Big Data συστήματος για την παρακολούθηση ενός cloud data-centre. Το πρώτο πράγμα που θα ζητήσει να βλέπει ο χρήστης μια τέτοιας λύσης θα είναι ανάλυση κάποιων βασικών δεικτών απόδοσης (Key Performance Indicators, KPIs) των κόμβων του συστήματος. Οι κόμβοι σε ένα τέτοιο σύστημα όμως μπορεί να είναι χιλιάδες. Το να δοθούν στατικές όψεις με τα KPIs του καθενός είναι προφανώς αναποτελεσματικό. Το LeanBigData μέσω μιας γραφικής γλώσσας χειρισμού θα επιτρέψει τον καθορισμό των πηγών δεδομένων (π.χ. μέσω καθορισμού ερωτημάτων), της μεταξύ τους διασύνδεσης και του τρόπου απεικόνισης. Χρήση μοντέρνων και πολλά υποσχόμενων τεχνικών (fisheye απεικονίσεις, Kinect volumetric κάμερες, multi-touch οθόνες κ.α.) θα επιταχύνουν την ανάλυση, δίνοντας στο χρήστη πρόσβαση σε 3D/holographic απεικονίσεις τις οποίες θα μπορεί εύκολα να τροποποιεί με κινήσεις του χεριού ώστε να βλέπει αυτό ακριβώς που θέλει.

Κλείνοντας αυτό το κεφάλαιο, αναφέρεται πως υλοποιώντας όλα τα παραπάνω, η LeanBigData πλατφόρμα φιλοδοξεί να επιτύχει επεξεργασία σε πραγματικό χρόνο ενός εκατομμύριου events το δευτερόλεπτο και τον ίδιο αριθμό SQL ερωτημάτων σε δέκατα του δευτερολέπτου. Αυτό μάλιστα με άνευ προηγουμένου απόδοση όσον αφορά στους πόρους που θα καταναλώνει καθώς υπολογίζεται πως θα δρα 5-10 φορές αποδοτικότερα από τις υπάρχουσες Big Data τεχνολογίες. Για να αποδειχθεί πως όλα αυτά μπορούν να επιτευχθούν, θα υλοποιηθεί ένα cluster με περισσότερους από 1000 πυρήνες και τουλάχιστον 100 χρήστες που θα αναλύσει δεδομένα που θα αντιπροσωπεύουν πραγματικά επιχειρηματικά σενάρια (monitoring

cloud data centers, ανίχνευση απάτης σε τραπεζικά συστήματα, ανάλυση trends σε μέσα κοινωνικής δικτύωσης, στοχευμένη διαφήμιση κ.α.) και θα ξεπερνούν σε όγκο τα 23TB. Επιτυγχάνοντας τους στόχους που έχει θέσει σε σενάρια σαν αυτά, θα καταστήσει σαφές το ότι ο χώρος των Big Data είναι έτοιμος να κάνει ένα βήμα μπροστά ξεπερνώντας τις παιδικές ασθένειες που αυτή τη στιγμή αντιμετωπίζει. Η έρευνα έχει αναγνωρίσει τα υπάρχοντα προβλήματα και είναι σε θέση να προτείνει λύσεις, η τεχνολογία έχει προχωρήσει και άρα δεν υπάρχει λόγος να μένουμε προσκολλημένοι σε δύστροπες και αναποτελεσματικές πρακτικές.

ΚΕΦΑΛΑΙΟ 2
Το Spring Batch Framework

2.1 Περιγραφή προβλήματος

Μεγάλο ηλεκτρονικό κατάστημα πώλησης βιβλίων, χαρτικών, παιχνιδιών και άλλων ειδών αποφάσισε την ανακατασκευή της ιστοσελίδας του. Το έργο δόθηκε σε νέα εταιρία που ανάμεσα σε αυτά που χρειαζόταν να κάνει ήταν να διαχειριστεί τη μεταφορά των δεδομένων της παλιάς ιστοσελίδας στις νέες υποδομές. Η εταιρία που είχε υλοποιήσει την παλιά εφαρμογή είχε χτίσει γέφυρες για να ενημερώνει τη βάση της με δεδομένα προερχόμενα από πολλές εξωτερικές πηγές όπως η **βιβλιονet**[13], η **Nielsen**[14], η **Gardners**[15] κ.α. Η βάση δεδομένων είχε πάρει μεγάλες διαστάσεις καθώς περιείχε περισσότερα από 7 εκατομμύρια προϊόντα, μαζί με πλήθος άλλων οντοτήτων απαραίτητων για την εμπορική διαχείριση και παρουσίαση στην ιστοσελίδα αυτών των προϊόντων (κατηγορίες, προμηθευτές, συγγραφείς, τιμές, αποθέματα κ.α.). Η νέα εταιρία χώρισε το έργο σε δυο φάσεις. Κατά την πρώτη φάση έπρεπε να υλοποιηθεί η εφαρμογή που παρουσιάζεται σε αυτή την εργασία, και η οποία χωριζόταν με τη σειρά της σε δυο κομμάτια: α) το αρχικό import του συνόλου της πληροφορίας στη νέα βάση δεδομένων και β) την υλοποίηση μηχανισμού ώστε σε καθημερινή βάση να διαβάζονται αλλαγές (δέλτα) των δεδομένων και να ενημερώνεται η νέα υποδομή. Τα δεδομένα από την αρχή φάνηκε πως είχαν πολλά προβλήματα και άρα δεν αρκούσε η αντιγραφή από το ένα data store στο άλλο. Έπρεπε να αναπτυχθεί λογική ώστε οι εγγραφές να περνάνε από πληθώρα ελέγχων και validations ώστε να διορθώνεται ότι είναι δυνατό ή / και να φιλτράρονται τα λάθη ώστε να μην μεταφέρονται σκουπίδια στη νέα βάση. Λόγω συμβολαίων με τους παρόχους που προαναφέρθηκαν, η εταιρία που είχε υλοποιήσει την παλιά ιστοσελίδα θα συνέχιζε για ένα σεβαστό χρονικό διάστημα να συντηρεί τη βάση δεδομένων της ώστε μην κοπούν οι γέφυρες με αυτούς. Η νέα εταιρία θα βασιζόταν στον μηχανισμό ανάγνωσης του δέλτα από την παλιά υποδομή μέχρι να της ανατεθεί η υλοποίηση της δεύτερης φάσης κατά την οποία θα υλοποιούνταν από τη νέα εφαρμογή οι γέφυρες ενημέρωσης και θα κοβόταν εντελώς η σχέση με την παλιά εταιρία.

Το πρόβλημα, αν και δεν ήταν εξ ορισμού Big Data είχε αρκετά κοινά σημεία με μια τέτοια εφαρμογή. Έχει ενδιαφέρον να δούμε πως τοποθετούνται οι απαιτήσεις του στα επίπεδα απόφασης που παρουσιάστηκαν στο προηγούμενο κεφάλαιο (εξαιρώντας προφανώς το επίπεδο της ανάλυσης που ήταν εντελώς έξω από τις ανάγκες μας). Τα δεδομένα ήταν ξεκάθαρα data-in-rest: ο παλιός πάροχος μέσω replication σε

[13] Η ιστοσελίδα της βιβλιονet βρίσκεται εδώ: http://www.biblionet.gr/
[14] Η ιστοσελίδα της Nielsen βρίσκεται εδώ: http://www.nielsen.com
[15] Η ιστοσελίδα της Gardners βρίσκεται εδώ: http://www.gardners.com

καθημερινή βάση θα έδινε αντίγραφο της παραγωγικής του βάσης, η οποία έτσι και αλλιώς δεν θα είχε traffic εκτός αυτού που θα δεχόταν από την εφαρμογή μας (με εξαίρεση ένα σύντομο χρονικό διάστημα κατά το οποίο, για λόγους testing, παλιά και νέα ιστοσελίδα θα έπαιζαν ταυτόχρονα). Ο μεγαλύτερος όγκος των δεδομένων διαβαζόταν από μια σχεσιακή βάση δεδομένων (δομημένα), αλλά δεν έλειπαν και κάποιες εξαιρέσεις που έπρεπε να χειριστούν μέσω αρχείων (αδόμητων δεδομένων), όπως η ανάγκη για ανάγνωση exports από το ERP του πελάτη για ενημέρωση των αποθεμάτων στην οποία θα αναφερθούμε με λεπτομέρειες σε επόμενο κεφάλαιο. Η ανάγκη για επεξεργασία των δεδομένων πριν αυτά εισαχθούν στη νέα βάση, αν και δεν είναι ξεκάθαρα ETL διαδικασία μοιάζει πολύ με τέτοια. Ετερογενής πληροφορία (από βάση δεδομένων, αρχεία κ.α.) εξαγάγεται, επεξεργάζεται (φιλτράρισμα, validations κ.α.), και τελικά μετασχηματίζεται στη δομημένη πληροφορία που φορτώνεται στην νέα σχεσιακή βάση δεδομένων. Η διαδικασία, ειδικά κατά το αρχικό import ήταν μια batch διαδικασία. Μεγάλος όγκος data-in-rest σε περιορισμένο χρονικό παράθυρο (ένα βράδυ) έπρεπε να μεταφερθεί στη νέα υποδομή. Ο χωρισμός τους σε δεσμίδες (chunks) και η παράλληλη επεξεργασία αυτών απο πολλά threads ήταν ο τρόπος για να επιτευχθεί ο στόχος. Οι ανάγκες αυτές οδήγησαν στη χρήση του Spring Batch framework μια εκτενής παρουσίαση των δυνατοτήτων του οποίου γίνεται στα ακόλουθα.

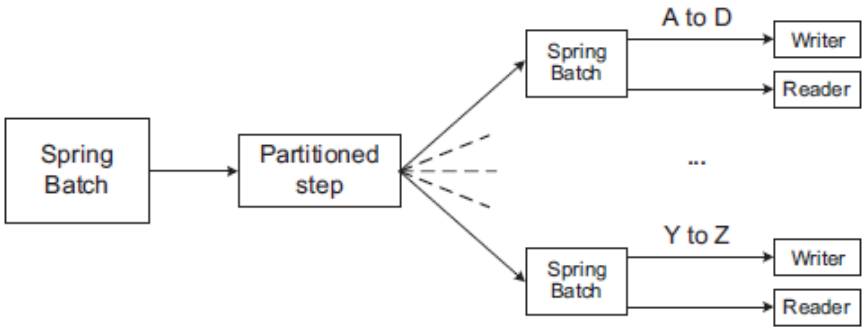
2.2 Εισαγωγή στο Spring Batch

Το Spring Batch είναι ένα open-source batch-oriented framework που δημιουργήθηκε το 2007 απο την SpringSource σε συνεργασία με την Accenture και σαν στόχο έχει να αντιμετωπίσει τις συνηθέστερες απαιτήσεις των batch εφαρμογών. Μια batch εφαρμογή πρέπει γρήγορα και αξιόπιστα να μπορεί να επεξεργαστεί πολύ μεγάλο όγκο δεδομένων χωρίς να απαιτεί την παρέμβαση χρήστη για να διαχειριστεί ή ακόμα και να ανακάμψει από σφάλματα που είναι αναμενόμενο να προκύψουν κατά την εκτέλεση.

Για παράδειγμα, και για να γίνει κατανοητό τι εννοούμε με το «αξιόπιστα», αν διαβάζουμε ένα αρχείο και σε αυτό υπάρχουν μια-δυο εγγραφές με λάθος φορμάτ, πρέπει ολόκληρο το batch job να αποτύχει; Αν η απάντηση είναι όχι, το Spring Batch μπορεί να χειριστεί αυτή την περίπτωση και το μόνο που χρειάζεται είναι λίγο configuration. Από την άλλη τι θέλουμε να συμβεί αν επανεκκινήσουμε ένα αποτυχημένο job; Θέλουμε το job να αρχίσει από το μηδέν και να επεξεργαστεί αντικείμενα που τα είχε ήδη επεξεργαστεί το προηγούμενο execution ή θέλουμε να συνεχίσει από εκεί που το προηγούμενο σταμάτησε; Αν θέλουμε το δεύτερο, το Spring Batch μπορεί πάλι να το κάνει αφού κατά την εκτέλεση των jobs του καταγράφει οτιδήποτε συμβαίνει και έτσι εύκολα μπορεί να συνεχίσει από το σωστό σημείο.

Για να επιτευχθεί τώρα το «γρήγορα», το framework ορίζει την έννοια του chunk processing. Αντί του να διαβάζει το σύνολο των δεδομένων ταυτόχρονα (με ότι αρνητικό αυτό μπορεί να συνεπάγεται για την

απόδοση της όλης διαδικασίας), τα διαβάζει και τα επεξεργάζεται σε μικρότερα chunks. Επιτυγχάνει έτσι streaming των δεδομένων, κρατώντας όλες τις φάσεις της διαδικασίας εκτέλεσης ενός job διαρκώς απασχολημένες, γεγονός που συνεπάγεται καλή απόδοση ακόμα και στην περίπτωση που το job εκτελείται από μόνο ένα thread. Αν παρόλα αυτά διαπιστώσουμε καθυστερήσεις και θέλουμε βελτίωση στην ταχύτητα, μπορούμε να εφαρμόσουμε κάποια από τις διαθέσιμες επιλογές για scaling. Μια από αυτές είναι το partitioning· τα steps χωρίζονται σε sub-steps με καθένα από αυτά να χειρίζεται ένα μόνο μέρος των δεδομένων και το κάθε sub-step να εκτελείται από διαφορετικό thread. Κάτι τέτοιο προϋποθέτει να γνωρίζουμε τη δομή των input δεδομένων ώστε αυτά να μπορούν να χωριστούν –κατά το δυνατόν– ισομερώς πριν διανεμηθούν στα διαθέσιμα threads. Ένα παράδειγμα φαίνεται στο Σχήμα 2.1. Σε αυτό, ένα σετ αρχείων χωρίζεται σε partitions με βάση το filename τους (ξεχωριστό partition για τα αρχεία με όνομα από A έως D, ξεχωριστό για τα αρχεία με όνομα από E έως H κ.ο.κ.) και αυτά επεξεργάζονται παράλληλα από κάποιο multi-threaded περιβάλλον.



Σχήμα 2.1. Partitioning με βάση το filename ενός σετ αρχείων

Για να κάνουμε μια εισαγωγή στα βασικά συστατικά ενός Spring Batch job είναι απαραίτητο να δώσουμε σαν παράδειγμα το configuration ενός τέτοιου στην Listing 2.1. Όλα όσα δίνονται σε αυτή στην παρούσα φάση είναι εντελώς άγνωστα, θα αναλυθούν όμως με λεπτομέρειες στις επόμενες ενότητες και όλα τα κομμάτια θα μπουν στη θέση τους. Προς το παρόν μας ενδιαφέρει μόνο η γενική εικόνα. Το **job** element (το γονικό στο configuration της Listing 2.1) αποτελείται από δυο **step** elements: το πρώτο βήμα (decompressStep) δέχεται σαν είσοδο ένα συμπιεσμένο αρχείο (input.zip), το αποσυμπιέζει και καλεί το επόμενο βήμα (readWriteStep) που διαβάζει τις εγγραφές του αποσυμπιεσμένου αρχείου (output.txt) και τις γράφει σε μια βάση δεδομένων. Εδώ, η εκτέλεση των δυο βημάτων είναι γραμμική. Το πρώτο δείχνει με το **next** attribute ποιο θα είναι το επόμενο. Αυτό προφανώς δεν είναι η μόνη παρεχόμενη δυνατότητα. Στα επόμενα θα αναφερθούμε στον τρόπο με τον οποίο μπορούμε να ορίσουμε περισσότερο σύνθετες ροές εκτέλεσης.

Τόσο για την ανάγνωση όσο και για την εγγραφή δεδομένων το framework παρέχει έτοιμους readers και writers που καλύπτουν τα συνηθέστερα batch σενάρια (ανάγνωση/εγγραφή από/σε flat files, XML, JSON, βάσεις δεδομένων, JMS κ.α.). Στο παράδειγμά μας χρησιμοποιούμε έναν **FlatFileItemReader** για ανάγνωση

από αρχείο, οι εγγραφές του οποίου έχουν fields που διαχωρίζονται από τον χαρακτήρα ‘,’. Οι εγγραφές περνάνε από τον **DelimitedLineTokenizer** που με βάση αυτό το διαχωριστικό τις «σπάει» στα properties ID και NAME ώστε να μπορέσει στη συνέχεια ο **FieldSetMapper** από τα fields αυτά να φτιάξει domain αντικείμενα (εδώ Authors). Για λόγους συντομίας, η υλοποίηση του mapper δεν δίνεται εδώ. Παραδείγματα θα δωθούν σε επόμενες ενότητες. Για τον ίδιο λόγο δεν δίνεται εδώ ούτε η υλοποίηση του writer. Αρκεί να αναφερθεί πως είναι ένας custom item writer που με τη βοήθεια του Spring JDBC template κάνει INSERT ή UPDATE σε μια βάση δεδομένων τα Author αντικείμενα.

```
<job id="importJob" xmlns="http://www.springframework.org/schema/batch">
  <step id="decompressStep" next="readWriteStep">
    <tasklet ref="decompressTasklet" />
  </step>
  <step id="readWriteStep">
    <tasklet>
      <chunk reader="reader" writer="writer"
        commit-interval="1000" skip-limit="5">
        <skippable-exception-classes>
          <include class="org.springframework.batch.item.file.FlatFileParseException" />
        </skippable-exception-classes>
      </chunk>
    </tasklet>
  </step>
</job>
<!-- Decompress Tasklet -->
<bean id="decompressTasklet" class="gr.booksAdmin.batch.DecompressTasklet">
  <property name="inputResource" value="file:/var/data/input/input.zip" />
  <property name="targetFile" value="file:/var/data/output/output.txt" />
</bean>
<!-- Read From A File -->
<bean id="reader" class="org.springframework.batch.item.file.FlatFileItemReader">
  <property name="resource" value="file:/var/data/output/output.txt" />
  <property name="linesToSkip" value="1" />
  <property name="lineMapper">
    <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
      <property name="lineTokenizer">
        <bean class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer">
          <property name="delimiter" value=","/>
          <property name="names" value="ID,NAME" />
        </bean>
      </property>
      <property name="fieldSetMapper">
        <bean class="gr.booksAdmin.batch.AuthorsFieldSetMapper" />
      </property>
    </bean>
  </property>
</bean>
<!-- Write To A Database -->
<bean id="writer"
   class="gr.booksAdmin.batch.AuthorsJdbcItemWriter">
  <constructor-arg ref="dataSource" />
</bean>
```

Listing 2.1 Configuration ενός Spring Batch job

Το μεγαλύτερο ενδιαφέρον απο τα στοιχεία της Listing 2.1 το έχει το readWriteStep που είναι ένα **chunk-oriented βήμα**. Εντός ενός **step** element ορίζεται το **chunk** element που καθορίζει τον **reader** και τον **writer** που θα χρησιμοποιηθούν καθώς και το **commit-interval** (το επιθυμητό μέγεθος για τα chunks). Θέτοντας το τελευταίο ίσο με 1000 καθορίζουμε πως το streaming των δεδομένων θα γίνεται σε σετ αντικειμένων αυτού του μεγέθους. Σωστή τιμή για το commit-interval δεν μπορεί να καθοριστεί θεωρητικά καθώς εξαρτάται από τη φύση του εκάστοτε job (ποια είναι τα εμπλεκόμενα data sources, ποιος ο όγκος των δεδομένων κ.α.). Πολύ μικρές τιμές θα οδηγήσουν σε άσκοπα μεγάλο αριθμό transactions (θα αναλυθεί το γιατί στα επόμενα) ενώ πολύ μεγάλη τιμή θα οδηγήσει σε άσκοπα μεγάλη χρήση πόρων. Στο βήμα αυτό, με τα **skippable-exception-classes** και **skip-limit** ορίζουμε και το ότι μέχρι 5 εγγραφές του αρχείου επιτρέπεται να μην έχουν το αναμενόμενο φορμάτ και άρα τα εγειρόμενα **FlatFileParseExceptions** (εφόσον δεν ξεπεράσουν αυτό τον αριθμό) δεν θα πρέπει να οδηγήσουν σε αποτυχία του job. Αυτό είναι ένα μόνο απο τα χαρακτηριστικά που διαθέτει το Spring Batch στη φαρέτρα του με σκοπό την αυτόματη διαχείριση πιθανών σφαλμάτων.

Κλείνοντας αυτή την εισαγωγή, πρέπει να αναφερθούμε και σε αυτό που το Spring Batch ονομάζει **tasklet**. Tasklet είναι κάθε ενέργεια που πρέπει να εκτελέσει ένα job η οποία δεν μπορεί να υλοποιηθεί με τη μορφή ενός chunk-oriented βήματος. Στο παράδειγμα που μελετάμε, μια τέτοια περίπτωση είναι η αποσυμπίεση του input αρχείου πριν αυτό διαβαστεί από τον FlatFileItemReader. Το tasklet πρέπει να υλοποιεί την **execute** μέθοδο του **Tasklet** interface και να επιστρέφει ένα **RepeatStatus** (π.χ. RepeatStatus.FINISHED) ώστε να γνωρίζει το job αν πρέπει να συνεχίσει στο επόμενο βήμα ή όχι. Ο σκελετός μιας τέτοιας υλοποίησης δίνεται ως παράδειγμα στην Listing 2.2 ενώ το configuration του tasklet έχει ήδη δοθεί στην Listing 2.1. Το μόνο που χρειάζεται είναι η κλάση μας να δηλωθεί ως ένα Spring bean και να γίνει injected σε ένα βήμα.

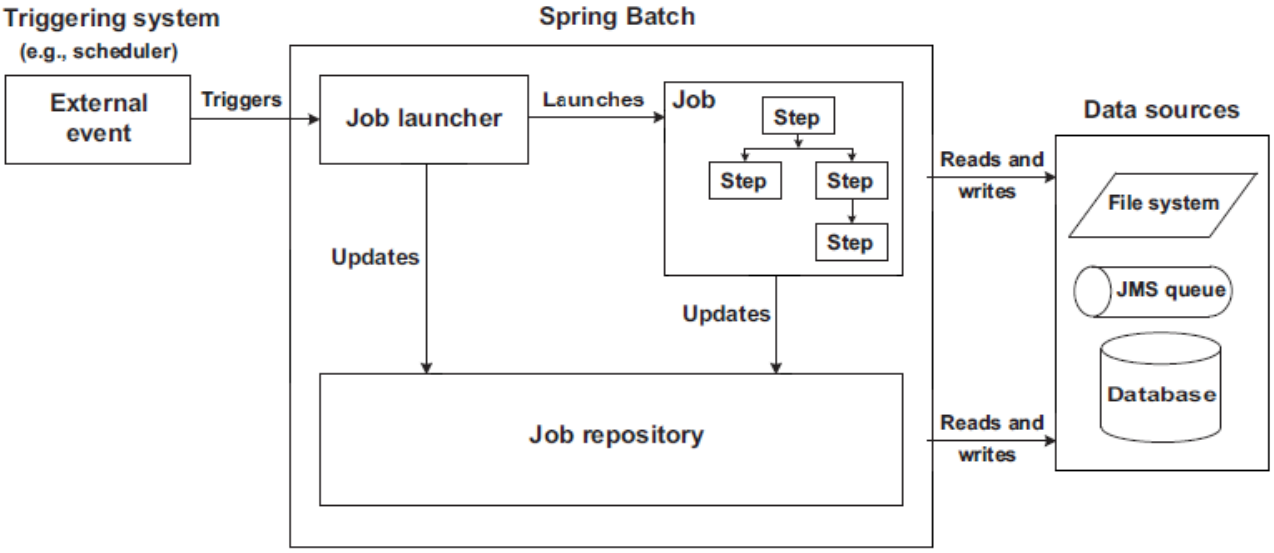
```
public class DecompressTasklet implements Tasklet {

  private Resource inputResource;

  private String targetFile;

  @Override
  public RepeatStatus execute(StepContribution stepContribution, ChunkContext chunkContext) throws Exception {
    // Decompress Implementation Skipped For Brevity
    return RepeatStatus.FINISHED;
  }

  // Setters and  Getters Skipped For Brevity
}
```

Listing 2.2 Παράδειγμα implementation ενός tasklet

2.2.1. Υποδομή του Spring Batch

Αν σε συνέχεια της γενικής εικόνας που δώσαμε για το configuration ενός Spring Batch job, θέλουμε να δούμε την γενική εικόνα του ίδιου του framework τότε μπορούμε να ανατρέξουμε στο Σχήμα 2.2. Το πρώτο πράγμα στο οποίο πρέπει να σταθούμε είναι στα δομικά συστατικά του framework: το **JobRepository** στο οποίο το Spring Batch κρατάει τα metadata των jobs που εκτελούνται (για λόγους monitoring αλλά και διαχείρισης των restarts εφόσον αυτό απαιτηθεί) και τον **JobLauncher** ο οποίος είναι υπεύθυνος για την εκτέλεση των jobs. Το triggering του JobLauncher είναι αποτέλεσμα ενός εξωτερικού event (cron job, HTTP call κ.α.) το οποίο πιθανώς περνά και τις απαραίτητες παραμέτρους στον JobLauncher με τη βοήθεια της **JobParameters** κλάσης. Έξω από το ίδιο το framework βρίσκονται και οι πηγές δεδομένων με τις οποίες τελικά τα jobs αλληλεπιδρούν. Όπως έχει ήδη αναφερεθεί, αλλά φαίνεται και στο Σχήμα 2.2, αυτές μπορεί να είναι αρχεία, βάσεις δεδομένων, Java Message Services (JMS) κ.α.

Για τον JobLauncher παρέχεται μόνο ένα implementation, το **SimpleJobLauncher**, το οποίο έχει μόνο μια μέθοδο, την **run**, που δέχεται σαν ορίσματα το **Job** που πρέπει να εκτελεστεί (ένα configuration σαν αυτό της Listing 2.1) και τις **JobParameters** που το job χρειάζεται και συνήθως δημιουργούνται δυναμικά κατά την εκκίνησή του. Πρέπει να ξεκαθαριστεί εδώ πως ο JobLauncher δεν δημιουργεί τα jobs, απλά ζητά την σύγχρονη ή ασύγχρονη εκτέλεσή τους από το JobRepository.



Σχήμα 2.2 Η γενική εικόνα του Spring Batch framework

Το JobRepository είναι αυτό που θα δημιουργήσει το job και θα αναλάβει κατά την εκτέλεσή του να ενημερώνει το repository με metadata σχετικά με τη λίστα των βημάτων που εκτελέστηκαν, τον αριθμό των αντικειμένων που διαβάστηκαν, έγιναν filtered ή skipped, το χρόνο που το κάθε βήμα χρειάστηκε κ.α. Για το

JobRepository παρέχεται πάλι μόνο ένα implementation, το **SimpleJobRepository**, μέσω του οποίου όμως έχουμε τη δυνατότητα να επιλέξουμε αν θέλουμε να κρατάμε τα metadata στη μνήμη (π.χ. κατά τη διάρκεια του development ή για λόγους testing) ή σε κάποια σχεσιακή βάση δεδομένων. Στην πρώτη περίπτωση δεν θα αναφερθούμε καθόλου. Χρειάζεται όμως να αναφερθούμε στη δεύτερη και το κάνουμε στην επόμενη υποενότητα.

2.2.1.1 JobRepository σε σχεσιακή βάση δεδομένων

Για την περίπτωση λοιπόν που θέλουμε το τα job metadata να διατηρούνται σε βάση δεδομένων, το Spring Batch παρέχει τόσο τα SQL scripts[16] για την δημιουργία των απαραίτητων πινάκων (βλ. Σχήμα 2.4), όσο και τα απαιτούμενα Data Access Objects (DAOs) ώστε τα jobs να μπορούν να εκτελούν τις απαραίτητες ενέργειες στους πίνακες αυτούς. Το μόνο δηλαδή που χρειάζεται, αν θεωρήσουμε πως έχουμε εκτελέσει το SQL script για το database engine της επιλογής μας, είναι το XML configuration του repository. Αυτό φαίνεται στην Listing 2.3.

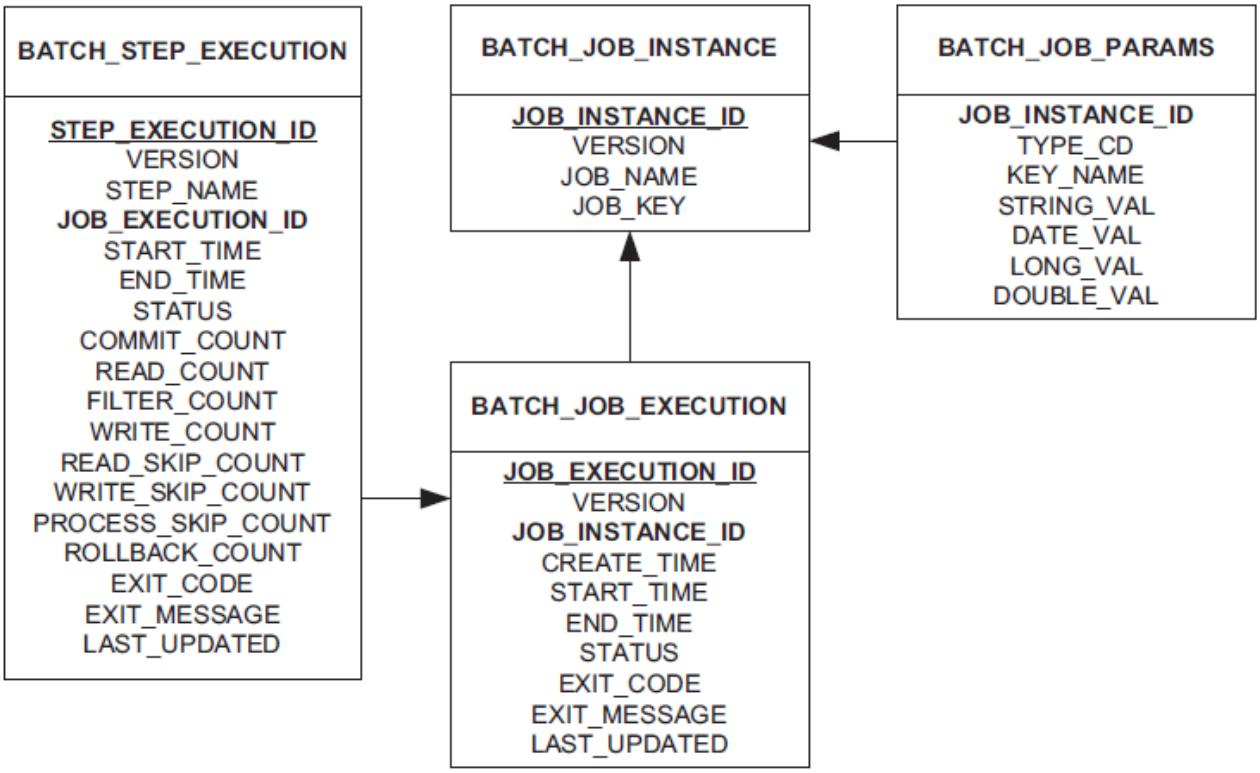
```
<batch:job-repository
   id="jobRepository"
   data-source="dataSource"
   transaction-manager="transactionManager"
   table-prefix="BATCH_"/>
<bean id="jobLauncher" class="org.springframework.batch.core.launch.support.SimpleJobLauncher">
   <property name="jobRepository" ref="jobRepository" />
</bean>
<bean id="dataSource" class="org.springframework.jdbc.datasource.SingleConnectionDataSource">
   <!-- Properties Skipped For Brevity -->
</bean>
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
   <property name="dataSource" ref="dataSource" />
</bean>
<!-- Example Of Job Using This JobRepository -->
<batch:job id="importJob" job-repository="jobRepository">
   (...)
</batch:job>
```

Listing 2.3 Configuration JobRepository σε βάση δεδομένων

Το **job-repository** element χρειάζεται ένα **data-source** (εδώ το **SingleConnectionDataSource**, ένα μόνο δηλαδή connection με τη βάση, αλλά θα μπορούσε να είναι και κάποιο connection pool σαν το Apache DBCP [17] ή το c3p0 [18]) και έναν **transaction-manager** για αυτό το data-source (εδώ τον **DataSourceTransactionManager**). Άλλα properties που μπορούν να οριστούν είναι τα **isolation-level-for-create** (default το SERIALIAZABLE), το **table-prefix** για να καθορίσουμε το πρόθεμα των πινάκων του Spring

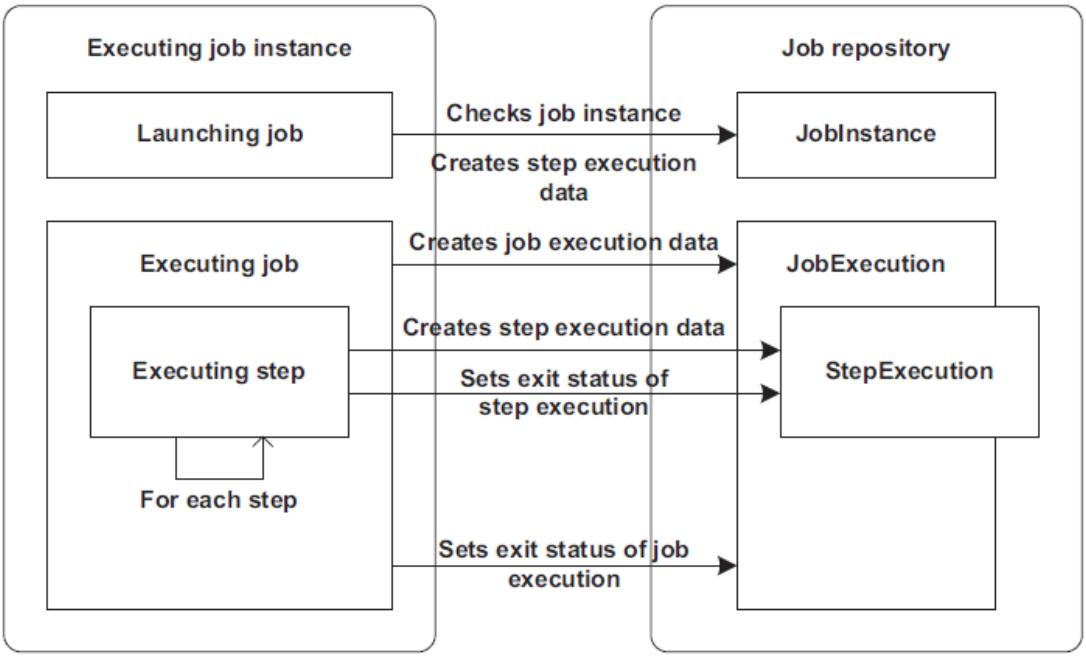
[16] Τα database engines που υποστηρίζονται είναι τα: DB2, Derby, H2. HSQLDB, MySQL, Oracle, PostgreSQL, SQL Server και Sybase
[17] Περισσότερα για το Apache DBCP εδώ: https://commons.apache.org/proper/commons-dbcp/
[18] Περισσότερα για το c3p0 εδώ: http://www.mchange.com/projects/c3p0/

Batch (default το BATCH_), το **max-varchar-length** για το μέγιστο μήκος των VARCHAR πεδίων στη βάση που χρησιμοποιούμε κ.α.



Σχήμα 2.3 Σχήμα της βάσης δεδομένων του JobRepository

Οι πίνακες του Σχήματος 2.4, μπορούν να μας βοηθήσουν και στο να καταλάβουμε το lifecycle ενός Spring Batch job και την αλληλεπίδραση που αυτό έχει με το job repository. Όταν λοιπόν ζητήσουμε από το Spring Batch την εκκίνηση ενός job, το framework θα εξετάσει με βάση τις παραμέτρους του το κατά πόσο το instance αυτό υπάρχει ήδη ή όχι (ένα job και οι παράμετροί του ορίζουν μοναδικά ένα job instance). Αν δεν υπάρχει μπορεί να δημιουργηθεί και να εκτελεστεί το πρώτο του execution, διαφορετικά θα πρέπει να ξεκινήσουμε ένα νέο execution για το ήδη υπάρχον instance και μόνο εφόσον αυτό δεν έχει ολοκληρωθεί επιτυχώς (μπορούμε δηλαδή να κάνουμε restart). Το execution κρατάει στο repository πληροφορίες για το πότε ξεκίνησε, το πότε τελείωσε, το instance στο οποίο ανήκει κ.α. Επόμενα στην αλυσίδα είναι τα βήματα του job που αρχίζουν να εκτελούνται και να ενημερώνουν και αυτά το repository με τα metadata που τα αφορούν. Κάθε βήμα με το που ολοκληρωθεί ενημερώνει με το exit status του. Αφού όλα τα βήματα ολοκληρωθούν (επιτυχώς ή όχι), ενημερώνεται και το exit status του job execution. Τα παραπάνω απεικονίζονται στο διάγραμμα του Σχήματος 2.5 με το οποίο ολοκληρώνεται αυτή η ενότητα.



Σχήμα 2.4 Αλληλεπίδραση ενός job με το job repository κατά τη διάρκεια εκτέλεσής του

2.2.2 Το step scope

Το Spring Batch παρέχει ένα ειδικό scope για τα beans, το step scope. Με χρήση αυτού και της **Spring Expression Language** (SpEL[19]) γίνεται δυνατή η αρχικοποίηση των beans ενός batch job κατά την εκκίνηση του εκάστοτε βήματος ώστε να είναι παραδείγματος χάριν δυνατό να καθορίσουμε παραμέτρους για το βήμα που είναι άγνωστες κατά την εκκίνηση του job. Οι οντότητες στις οποίες μέσω SpEL έχουμε πρόσβαση από το step scope είναι οι **jobParameters**, **jobExecutionContext** και **stepExecutionContext**. Σαν παράδειγμα χρήσης τους μπορούμε να δούμε πως βελτιώνεται το decompressTasklet της Listing 2.1 ώστε τα inputResource και targetFile properties να μην ορίζονται στατικά στο XML configuration. Θέτοντας το scope attribute του tasklet ίσο με step και χρησιμοποιώντας τα σύμβολα #{ και } αποκτάμε πρόσβαση στο jobParameters map οι τιμές του οποίου τίθενται από τον JobLauncher κατά την εκκίνηση του job. Με τον τρόπο αυτό, τα properties του tasklet ορίζονται πλέον δυναμικά (βλ. Listing 2.4). Παραδείγματα των όσων επιτυγχάνονται με χρήση των άλλων δυο οντοτήτων (jobExecutionContext και stepExecutionContext) θα δούμε σε επόμενες ενότητες.

```
<bean id="decompressTasklet" class="gr.booksAdmin.batch.DecompressTasklet" scope="step">
  <property name="inputResource" value="#{jobParameters['inputResource']}" />
  <property name="targetFile" value="#{jobParameters['targetFile']}" />
</bean>
```

Listing 2.4 Χρήση του step scope

[19] Περισσότερα για την Spring Expression Language εδώ: https://docs.spring.io/spring/docs/current/spring-framework-reference/html/expressions.html

2.2.3 Listeners

Το Spring Batch παρέχει τη δυνατότητα για χρήση listeners στο επίπεδο τόσο του job όσο και του step ενός batch. Με χρήση τους μπορούμε να παρακολουθούμε συγκεκριμένα events στο lifecycle ενός batch job και να προσθέτουμε λογική ανάλογα με τις απαιτήσεις τις εφαρμογής μας. Ένα από τα κλασικότερα παραδείγματα είναι η χρήση listener ώστε να ειδοποιείται ο διαχειριστής της εφαρμογής (π.χ. με email) όταν ένα job αποτυγχάνει. Το πως μπορεί να γίνει αυτό αναλύεται ευθύς αμέσως.

Για να παρακολουθούμε τα **beforeJob** και **afterJob** events ενός job παρέχεται το **JobExecutionListener** interface. Υλοποιούμε το interface αυτό κάνοντας override της δυο αυτές μεθόδους ή χρησιμοποιούμε τα **@BeforeJob** και **@AfterJJob** annotations σε μια απλή κλάση όπως αυτή της Listing 2.5. Σε αυτή, με το που ολοκληρώνεται το job ελέγχουμε αν το ExitStatus του είναι FAILED και αν ναι στέλνουμε ένα notification email στον διαχειριστή (το πως, παραλείπεται εδώ για λόγους συντομίας).

```
public class JobFailureListener {

   @BeforeJob
  public void beforeJob(JobExecution jobExecution) {
            //Notifying When Job Starts
  }

  @AfterJob
  public void afterJob(JobExecution jobExecution) {
    if(ExitStatus.FAILED.equals(jobExecution.getExitStatus())) {
            //Implementation Skipped For Brevity
    }
  }
}
```

Listing 2.5 Παράδειγμα χρήσης των @BeforeJob και @AfterJob annotations

Η Listing 2.6 δείχνει το πως αυτός ο listener ορίζεται στο configuration του job μας με χρήση του **listeners** element ως παιδί του element job. Το listeners δέχεται μια λίστα από **listener** elements πράγμα που σημαίνει πως μπορούμε να χρησιμοποιήσουμε όσους listener θέλουμε. Εκτός του JobExecutionListener παρέχεται πληθώρα άλλων listeners τόσο σε επίπεδο step όσο και σε επίπεδο item (**StepExecutionListener**, **ChunkListener**, **SkipListener**, **RepeatListener**, **RetryListener** κ.α.) ώστε με παρόμοια λογική να παρακολουθούμε events όπως τα beforeStep, afterStep, beforeChunk, afterChunk, onSkipInRead, onSkipInWrite και πολλά άλλα. Παραδείγματα χρήσης τους θα δούμε σε επόμενες ενότητες.

```
<batch:job id="importJob">
  <batch:listeners>
    <batch:listener ref="jobFailureListener"/>
  </batch:listeners>
</batch:job>
<bean id="jobFailureListener" class="gr.booksAdmin.batch.JobFailureListener"/>
```

Listing 2.6 Configuration ενός JobExecutionListener

2.2.3.1 Monitoring listener με χρήση του BatchMonitoringNotifier

Στην περίπτωση που στόχος μας είναι όντως το monitoring, και θέλουμε να υλοποιήσουμε περισσότερους από έναν μηχανισμούς για ειδοποιήσεις σε περίπτωση αποτυχίας των jobs (π.χ. και μέσω email και μέσω Java messaging), για να κρατήσουμε την υλοποίηση του listener γενική και παραμετροποιήσιμη μπορούμε να χρησιμοποιήσουμε το **BatchMonitoringNotifier** interface όπως φαίνεται στην Listing 2.7.

```
public class MonitoringExecutionListener {
  private BatchMonitoringNotifier monitoringNotifier;

  @BeforeJob
  public void executeBeforeJob(JobExecution jobExecution) {
    //Do Nothing
  }
  @AfterJob
  public void executeAfterJob(JobExecution jobExecution) {
    if(BatchStatus.FAILED.equals(jobExecution.getStatus())) {
      monitoringNotifier.notify(jobExecution);
    }
  }
  public void setMonitoringNotifier(BatchMonitoringNotifier monitoringNotifier) {
    this.monitoringNotifier = monitoringNotifier;
  }
}
```

Listing 2.7 Χρήση του BatchMonitoringNotifier απο listerner για monitoring

Στο **afterJob** event ελέγχουμε το ExitStatus του execution αλλά τώρα, αντί του να καλούμε κάποιο συγκεκριμένο service για να στείλουμε την ειδοποίηση (π.χ. μέσω email όπως κάναμε στην Listing 2.5), καλούμε την **notify** μέθοδο του BatchMonitoringNotifier η οποία δέχεται ως παράμετρό της το JobExecution για να έχει πρόσβαση στο instance του job, τα exception που πιθανώς παρουσιάστηκαν κτλ. Μπορούμε λοιπόν να έχουμε περισσότερες από μια υλοποιήσεις του εν λόγω interface που θα παίζουν το ρόλο του notification service και περισσότερους του ενός listeners που θα χρησιμοποιούν αυτά τα services χωρίς όμως να χρειάζεται κάτι παραπάνω από λίγο XML configuration σαν αυτό της Listing 2.8.

```
<bean id="emailNotifierListener" class="gr.booksAdmin.batch.MonitoringExecutionListener">
  <property name="monitoringNotifier" ref="emailNotifier"/>
</bean>
<bean id="messagingNotifierListener" class="gr.booksAdmin.batch.MonitoringExecutionListener">
<property name="monitoringNotifier" ref="messagingNotifier"/>
</bean>
<bean id="emailNotifier" class="(...)"><!-- BatchMonitoringNotifier Implementations -->
  (...)
</bean>
<bean id="messagingNotifier" class="(...)">
  (...)
</bean>
```

Listing 2.8 Configuration των παραμετροποιήσιμων listeners

2.2.4 Κληρονομικότητα

Χρησιμοποιώντας τη δυνατότητα για κληρονομικότητα στα XML configurations που το Spring framework παρέχει, το Spring Batch επιτρέπει τον ορισμό **abstract** jobs ή βήματα ώστε κάποια κοινά χαρακτηριστικά να μπορούν να οριστούν σε αυτά και να κληρονομούνται από όποιον τα χρειάζεται χωρίς να επαναλαμβάνονται άσκοπα τα configurations. Βλέπουμε για παράδειγμα στη Listing 2.9 πως ορίζουμε το abstract parentJob που χρησιμοποιεί τον jobFailureListener της προηγούμενης ενότητας ή το abstract parentStep που χρησιμοποιεί έναν StepExecutionListener και ορίζει πως την διαχείριση των transactions του την κάνει το transcactionManager bean που ορίστηκε στην Listing 2.3. Το authorsJob τέλος, ορίζει με το **parent** attribute πως κληρονομεί τον listener του parentJob, και το authorsStep με το ίδιο attribute πως κληρονομεί τον listener και τον transaction-manager του parentStep.

```
<batch:job id="parentJob" abstract="true">
  <batch:listeners>
    <batch:listener ref="jobFailureListener"/>
  </batch:listeners>
</batch:job>
<batch:step id="parentStep" abstract="true">
  <batch:tasklet transaction-manager="transactionManager">
    <batch:listeners>
      <batch:listener ref="stepStatusListener"/>
    </batch:listeners>
  </batch:tasklet>
</batch:step>
<!-- Job Example -->
<batch:job id="authorsJob" parent="parentJob">
  <batch:step id="authorsStep" parent="parentStep" next="(...)">
    (...)
  </batch:step>
</batch:job>
```

Listing 2.9 Abstract Job και Step με παράδειγμα χρήσης τους

Ενδιαφέρον έχει και η δυνατότητα για merge των list ορισμάτων ενός abstract job ή βήματος με αυτά των παιδιών που τα κληρονομούν. Αν για παράδειγμα το abstract job ορίζει κάποιους listeners και θέλουμε το job μας να προσθέσει και κάποιον (κάποιους) ακόμα, μπορούμε να το κάνουμε θέτοντας το **merge** attribute σε true όπως φαίνεται στην Listing 2.10.

```
<batch:job id="parentJob" abstract="true">
  <batch:listeners>
    <batch:listener ref="listener1"/>
    <batch:listener ref="listener2"/>
  </batch:listeners>
</batch:job>
<batch:job id="authorsJob" parent="parentJob">
  (...)
  <batch:listeners merge="true">
    <batch:listener ref="listener3"/>
```

```
  <batch:listeners>
</batch:job>
```

Listing 2.10 Merge ορισμάτων ενός abstract job με τα ορίσματα του child job

2.2.5 Εκκίνηση Spring Batch jobs

Αναφέρθηκε στα προηγούμενα πως υπεύθυνος για την εκκίνηση των Spring Batch jobs είναι ο **JobLauncher**. Το interface του παρέχει την μέθοδο **run** που δέχεται σαν όρισμα το **Job** που πρέπει να τρέξει και τις **JobParameters** αυτού. Παράδειγμα μιας τέτοιας κλήσης δίνεται στην Listing 2.11 στην οποία χρησιμοποιούμε την κλάση **JobParametersBuilder** για να δημιουργήσουμε ένα JobParameters instance με δυο παραμέτρους. Υπενθυμίζεται πως για το Spring Batch το job και οι παράμετροί του ορίζουν μοναδικά ένα job instance και εφόσον αυτό έχει ολοκληρωθεί δεν μπορεί να ξανατρέξει (παρά μόνο αν μιλάμε για restart ενός αποτυχημένου instance). Για το λόγο αυτό είναι που στον JobParametersBuilder περνάμε και την τρέχουσα ημερομηνία ως παράμετρο: για να έχουμε σε κάθε εκτέλεση της run μεθόδου διαφορετικό job instance.

```
ApplicationContext context = (...)
JobLauncher jobLauncher = context.getBean(JobLauncher.class);
Job job = context.getBean(Job.class);
jobLauncher.run(
   job,
   new JobParametersBuilder()
      .addString("inputFile", "file:./input.txt")
      .addDate("date", new Date())
      .toJobParameters()
);
```

Listing 2.11 Παράδειγμα εκκίνησης ενός job

Για τα jobs, επειδή είναι batch διαδικασίες που μπορεί να χρειάζονται πολύ ώρα για να εκτελεστούν παρέχεται η δυνατότητα για σύγχρονη (το default) και ασύγχρονη εκτέλεση. Στην περίπτωση που επιθυμούμε το δεύτερο το μόνο που χρειάζεται να κάνουμε είναι να δώσουμε σαν όρισμα στον JobLauncher έναν **TaskExecutor**. Παράδειγμα αυτού του configuration με ένα pool από 10 threads δίνεται στην Listing 2.12.

```
<task:executor id="executor" pool-size="10" />
<bean id="jobLauncher" class="org.springframework.batch.core.launch.support.SimpleJobLauncher">
   <property name="jobRepository" ref="jobRepository" />
   <property name="taskExecutor" ref="executor" />
</bean>
```

Listing 2.12 Configuration ασύγχρονου JobLauncher

Έχοντας ορίσει τον JobLauncher και το αν αυτός θα τρέχει σύγχρονα ή ασύγχρονα τα jobs, το μόνο που μένει είναι ο τρόπος με τον οποίο θα καλέσουμε την run μεθοδό του. Διάφορες είναι οι εναλλακτικές εδώ.

Μπορούμε να χρησιμοποιήσουμε τον command line launcher που παρέχει το ίδιο το framework, ένα cron job εφόσον δουλεύουμε σε UNIX-like σύστημα, Spring scheduled tasks, HTTP κλήσεις σε έναν controller που θα αναλάβει την επικοινωνία με τον JobLauncher κ.α. Στα επόμενα, εν συντομία, θα παρουσιαστούν οι δυο τελευταίες επιλογές.

2.2.5.1 Εκκίνηση με χρήση του Spring Scheduler

Το Spring framework παρέχει τη δυνατότητα για χρονοπρογραμματισμό εργασιών είτε μέσω Java είτε μέσω XML configuration. Στην δεύτερη περίπτωση, στην οποία θα σταθούμε εδώ, χρησιμοποιώντας τα elements που ορίζονται στο **task** namespace (www.springframework.org/schema/task), καθορίζουμε (βλ. σαν παράδειγμα την Listing 2.13) τον task **scheduler** που με ένα pool απο 10 threads θα εκτελεί τα tasks, ένα bean που περνάει σε μια custom κλάση (SpringSchedulingLauncher) το προς εκτέλεση job (ήδη ορισμένο σε XML) και τον jobLauncher, και το ίδιο το **scheduled-task** με το οποίο διαλέγουμε την μέθοδο της κλάσης μας που θα εκτελείται (εδώ launch) και το κάθε πότε. Το τελευταίο μπορεί να είναι ένα **cron** expression σαν το "**0 0 15 * * ***" (κάθε μέρα στις 15:00) ή τιμές για τα fixed-rate, fixed-delay attributes ώστε τα tasks να εκτελούνται περιοδικά με βάση την ώρα εκκίνησης ή ολοκλήρωσης του προηγούμενου task (π.χ. κάθε μια ώρα ή μια ώρα από την ολοκλήρωση του προηγούμενου).

Η υλοποίηση της κλάσης SpringSchedulingLauncher δεν έχει κάτι άξιο αναφοράς. Γνωρίζοντας το Job που πρέπει να τρέξει και τον JobLauncher της εφαρμογής, το μόνο που έχει να κάνει είναι να εκκινήσει το job με λογική παρόμοια αυτής της Listing 2.11.

```
<task:scheduler id="scheduler" pool-size="10"/>
<!-- Scheduled Task -->
<bean id="scheduledTask" class="gr.booksAdmin.task.SpringSchedulingLauncher">
   <property name="job" ref="authorsJob"/>
   <property name="jobLauncher" ref="jobLauncher"/>
</bean>
<task:scheduled-tasks scheduler="scheduler">
   <task:scheduled
      ref="scheduledTask"
      method="launch"
      cron="0 0 15 * * *" />
</task:scheduled-tasks>
```

Listing 2.13 Παράδειγμα Spring scheduler για προγραμματισμό Spring Batch job

2.2.5.2 Εκκίνηση μέσω Web εφαρμογής

Για να παρουσιάσουμε την περίπτωση αυτή θεωρούμε πως εκτός της Spring Batch εφαρμογής υπάρχει και μια Spring MVC εφαρμογή που θα αναλάβει την διαχείριση των HTTP κλήσεων προς κάποιο controller (βλ. τον JobLauncherController της Listing 2.14), ο οποίος με τη σειρά του θα αναλάβει την επικοινωνία με τον

JobLauncher. Δεν θα αναφέρουμε τίποτα εδώ σχετικά με το πως μπορεί να υλοποιηθεί μια Spring MVC εφαρμογή. Θεωρούμε πως υπάρχει έτοιμο το infrastructure ώστε μια κλήση προς το URL:

/joblauncher?job=authorsJob&param1=value1&param2=value2

να διαχειριστεί από τον JobLauncherController ο οποίος, όπως φαίνεται στον κώδικα, αναλαμβάνει να μετατρέψει τα URL parameters (param1, param2 κοκ) σε ένα JobParameters instance και να καλέσει την run μέθοδο του JobLauncher.

```
@Controller
public class JobLauncherController {

   private static final String JOB_PARAM = "job";

   private JobLauncher jobLauncher;
   private JobRegistry jobRegistry;

   @RequestMapping(value="joblauncher", method = RequestMethod.GET)
   @ResponseStatus(HttpStatus.ACCEPTED)
   public void launch(@RequestParam String job, HttpServletRequest request)
            throws Exception {
      JobParametersBuilder builder = extractParameters(request);
      jobLauncher.run(
         jobRegistry.getJob(request.getParameter(JOB_PARAM)),
         builder.toJobParameters()
      );
   }

   private JobParametersBuilder extractParameters(HttpServletRequest request) {
      JobParametersBuilder builder = new JobParametersBuilder();
      Enumeration<String> paramNames = request.getParameterNames();
      while(paramNames.hasMoreElements()) {
         String paramName = paramNames.nextElement();
         if(!JOB_PARAM.equals(paramName)) {
            builder.addString(paramName,request.getParameter(paramName));
         }
      }
      return builder;
   }

   //Setters Skipped For Brevity
}
```

Listing 2.14 Spring MVC Controller για την εκκίνηση Spring Batch job

Ιδιαίτερη αναφορά χρειάζεται πριν κλείσουμε αυτή την ενότητα στο **JobRegistry** αντικείμενο της Listing 2.14. Με δεδομένο το ότι το όνομα του job που θέλουμε να εκτελέσουμε καθορίζεται ως μια request παράμετρος τύπου String, χρειαζόμαστε ένα τρόπο ώστε αυτή να μετατραπεί στο Job αντικείμενο που περιμένει σαν παράμετρο ο JobLauncher. Αυτό γίνεται με τη βοήθεια του JobRegistry interface (μέθοδος **getJob**) που ορίζεται μαζί με τα υπόλοιπα δομικά στοιχεία μιας Spring Batch εφαρμογής όπως φαίνεται στην Listing 2.15.

```
<bean id="jobRegistry" class="org.springframework.batch.core.configuration.support.MapJobRegistry" />
<bean class="org.springframework.batch.core.configuration.support.JobRegistryBeanPostProcessor">
  <property name="jobRegistry" ref="jobRegistry" />
</bean>
```

Listing 2.15 Configuration του JobRegistry

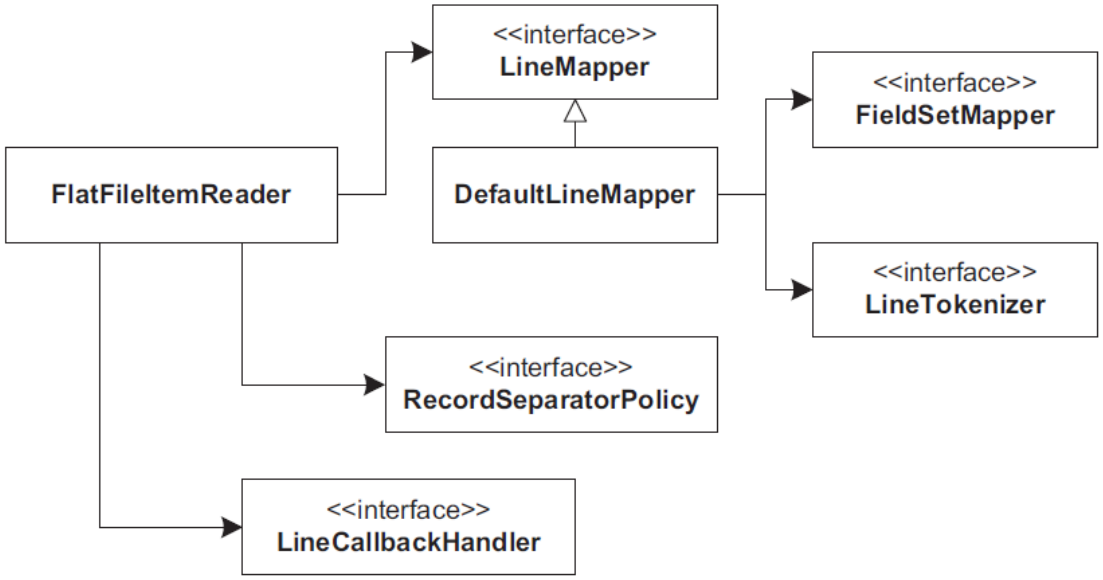
2.3 Ανάγνωση Δεδομένων

Όπως έχει ήδη αναφερθεί, το chunk-processing χωρίζεται σε 3 φάσεις: ανάγνωση, επεξεργασία και εγγραφή. Ξεκινάμε με την πρώτη από αυτές τις φάσεις, την ανάγνωση, για την οποία το Spring Batch παρέχει έτοιμες κλάσεις και implementations που μπορούν να καλύψουν τα συνηθέστερα σενάρια. Μεταξύ άλλων λοιπόν, υπάρχουν readers για ανάγνωση αρχείων (flat files, XML και JSON), βάσεων δεδομένων, Java Message Services (JMS), ενώ παρέχεται και η δυνατότητα για επέκταση των core reading interfaces (**ItemReader** και **ItemStream**) ώστε να υλοποιηθούν readers που θα διαβάζουν πηγές δεδομένων εκτός των προαναφερθέντων. Εδώ θα σταθούμε στα όσα χρησιμοποιήθηκαν στα πλαίσια αυτής της εργασίας και είναι η ανάγνωση από flat αρχεία και βάσεις δεδομένων.

2.3.1 Ανάγνωση Αρχείων

Τα πεδία ενός flat αρχείου μπορεί να έχουν σταθερό μήκος ή να διαχωρίζονται από κάποιον προκαθορισμένο separator (π.χ. comma-separated CSV), η πρώτη του γραμμή μπορεί να είναι επικεφαλίδα, είναι πιθανό να περιέχει εγγραφές σχόλια που δεν πρέπει να ληφθούν υπόψη από τον reader κ.α. Καθώς λοιπόν ο πάροχος του αρχείου είναι αυτός που καθορίζει το ακριβές format του, το default implementation του Spring Batch για ανάγνωση flat αρχείων (ο FlatFileItemReader) δουλεύει σε συνεργασία με διάφορα άλλα interfaces ώστε να καθοριστεί επακριβώς ο τρόπος με τον οποίο πρέπει να διαβαστούν τα δεδομένα. Τα interfaces αυτά παρουσιάζονται στο Σχήμα 2.6.

Το **RecordSeparatorPolicy** interface είναι υπεύθυνο για το καθορισμό του που αρχίζει και που τελειώνει κάθε εγγραφή του αρχείου, το **LineMapper** interface είναι υπεύθυνο για την εξαγωγή δεδομένων από τις γραμμές και το **LineCallbackHandler** interface χειρίζεται ειδικές περιπτώσεις (π.χ. τι πρέπει να γίνει με τις εγγραφές που γίνονται skip κατά την ανάγνωση). Το πιο συχνά χρησιμοποιούμενο implementation του LineMapper είναι το **DefaultLineMapper** που χρησιμοποιεί το **LineTokenizer** interface για να διαχωρίζει τις γραμμές που διαβάζονται σε tokens, και το **FieldSetMapper** interface που από αυτά τα tokens δημιουργεί αντικείμενα.



Σχήμα 2.5 Κλάσεις και interfaces που εμπλέκονται στην ανάγνωση flat αρχείων

Στην Listing 2.16 δίνεται ένα τυπικό configuration του FlatFileItemReader. Καθορίζουμε ποιο είναι το αρχείο που θα διαβάζει με το property **resource**, το ότι η πρώτη γραμμή του αρχείου είναι επικεφαλίδα που δεν πρέπει να ληφθεί υπόψη (property **linesToSkip**), πως διαχωρίζονται οι εγγραφές του αρχείου (property **recordSeparatorPolicy**) και ποιος θα είναι ο **lineMapper** που θα δημιουργεί από τις γραμμές που διαβάζονται τα αντικείμενα (περισσότερα για τα δυο τελευταία στα ακόλουθα). Διατίθενται και άλλα properties πέρα απο αυτά, όπως το **encoding** του αρχείου, το αν κάποιες γραμμές του είναι **comments** που πρέπει να γίνουν skip (String[] value με προθέματα που καθορίζουν τις γραμμές σχόλια), το αν ο reader πρέπει να οδηγεί σε exception όταν δεν βρει αρχείο στο καθορισμένο path ή όχι (property **strict**) κ.α.

```
<bean id="inventoryItemReader" class="org.springframework.batch.item.file.FlatFileItemReader">
   <property name="resource" value="inventory.csv"/>
   <property name="linesToSkip" value="1"/>
   <property name="recordSeparatorPolicy" ref="inventoryRecordSeperatorPolicy"/>
   <property name="lineMapper" ref="inventoryLineMapper"/>
</bean>
<bean id="inventoryRecordSeperatorPolicy" class="(...)">
   (...)
</bean>
<bean id="inventoryLineMapper" class="(...)">
   (...)
</bean>
```

Listing 2.16 Configuration του FlatFileItemReader

Το **RecordSeparatorPolicy** interface καθορίζει που τελειώνει κάθε εγγραφή του αρχείου και παρέχει μεθόδους για το pre- ή post- process καθεμιάς απο αυτές. Το Spring Batch παρέχει 4 implementations αυτού του interface: το **SimpleRecordSeparatorPolicy** (κάθε γραμμή του αρχείου είναι και μια εγγραφή), το **DefaultRecordSeparatorPolicy** (οι εγγραφές περικλείονται σε quotes), το **JsonRecordSeparatorPolicy** (οι εγγραφές είναι JSON strings που πιθανώς εκτείνονται σε περισσότερες από μια γραμμές) και το

SuffixRecordSeparatorPolicy (οι εγγραφές τερματίζουν με κάποιο suffix, το default είναι το ";"). Το default configuration αυτών των policies καλύπτει τις περισσότερες περιπτώσεις που συναντώνται στην πράξη και άρα το μόνο που συνήθως απαιτείται είναι απλά ένα reference στην κλάση που επιθυμούμε να χρησιμοποιήσουμε.

Το **LineMapper** interface, όπως προαναφέρθηκε, είναι υπεύθυνο για την δημιουργία αντικειμένων από τις εγγραφές του αρχείου. Τα implementations που παρέχονται είναι τα **PassThroughLineMapper** (επιστρέφεται το string που διαβάζεται χωρίς να μετασχηματιστεί σε αντικείμενο), **DefaultLineMapper** (η εγγραφή σπάει σε tokens και με βάση αυτά δημιουργούνται αντικείμενα), **JsonLineMapper** (η εγγραφή που αναπαριστάται ως JSON μετατρέπεται σε αντικείμενο με τη βοήθεια του Jackson[20] mapper) και **PatternMatchingCompositeLineMapper** (για την υποστήριξη ετερογενών εγγραφών με κάθε τύπο να πρέπει να διαχειριστεί απο διαφορετικό tokenizer και/ή field-set mapper).

Σαν παράδειγμα εδώ μπορούμε να δώσουμε το configuration του DefaultLineMapper (βλ. Listing 2.17) που χειρίζεται κάθε εγγραφή σε δυο φάσεις: α) με τη βοήθεια ενός **LineTokenizer** την σπάει σε tokens (fields), β) με τη βοήθεια ενός **FieldSetMapper** μετατρέπει τα tokens σε αντικείμενα. Ο LineTokenizer ορίζει πως το **delimiter** για τα tokens κάθε εγγραφής είναι το "!*" και πως τα ονόματα των fields είναι τα sapid, productId και stock (παράδειγμα μιας τέτοιας εγγραφής είναι το 200000001!*PR100!*3). Αντί του **DelimitedLinetokenizer**, αν τα tokens των εγγραφών μας είχαν σταθερό μήκος θα μπορούσαμε να χρησιμοποιήσουμε τον **FixedLengthTokenizer** ορίζοντας την αρχή και το τέλος του κάθε token (π.χ. 1-9,10-15,16-18).

```
<bean id="inventoryLineMapper" class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
  <property name="lineTokenizer" ref="inventoryLineTokenizer"/>
  <property name="fieldSetMapper" ref="inventoryFieldSetMapper"/>
</bean>
<bean id="inventoryLineTokenizer" class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer">
  <property name="delimiter" value="!*"/>
  <property name="names" value="sapId,productId,stock"/>
</bean>
<bean id="inventoryFieldSetMapper" class="(...)"/>
```

Listing 2.17 Configuration του DefaultLineMapper

Έχοντας τα fields της κάθε εγγραφής το μόνο που μένει είναι να καθορίσουμε πως θα δημιουργήσουμε τα αντικείμενα. Μπορούμε να χρησιμοποιήσουμε τον **PassThroughFieldSetMapper** (στέλνει το FieldSet στον processor ή τον writer χωρίς να το μετασχηματίσει σε αντικείμενο), τον **BeanWrapperFieldSetMapper** (χρησιμοποιεί τα ονόματα των fields που καθορίστηκαν παραπάνω ώστε να γίνει ένα είδους mapping με τα attribute names ενός plain old Java Object (POJO)) ή να υλοποιήσουμε έναν δικό μας mapper που θα

[20] Περισσότερα για το Jackson εδώ: https://github.com/codehaus/jackson

μετατρέπει τα FieldSet σε αντικείμενα κάνοντας implement το **FieldSetMapper** interface (περισσότερα σε επόμενο κεφάλαιο).

Κλείνοντας αξίζει να αναφερθεί πως στην περίπτωση που απαιτείται να διαβάσουμε σετ αρχείων (που έρχονται π.χ. σε ένα FTP ή SCP directory με μόνο γνωστό το pattern του ονόματός τους) μπορούμε να χρησιμοποιήσουμε τον **MultiResourceItemReader** που διαχειρίζεται σειριακά όλα τα resources αναθέτοντας το processing σε έναν FlatFileItemReader σαν αυτόν που αναφέρθηκε στην Listing 2.16. Το configuration ενός τέτοιου reader είναι αυτό που φαίνεται στην Listing 2.18. Το μόνο που χρειάζεται είναι να καθορίσουμε μέσω του property **resources** ποια θα είναι τα αρχεία που θα λαμβάνονται υπόψη (εδώ όσα CSV βρεθούν στον φάκελο /var/data και το όνομά τους ξεκινάει με "inventory-") και ποιος θα είναι ο FlatFileItemReader που θα τα διαχειρίζεται (property **delegate**).

```
<bean id="inventoryMultiResourcesItemReader"
class="org.springframework.batch.item.file.MultiResourceItemReader">
   <property name="resources" value="file:/var/data/inventory-*.csv"/>
   <property name="delegate" ref="inventoryItemReader"/>
</bean>
<bean id="inventoryItemReader" class="org.springframework.batch.item.file.FlatFileItemReader" />
```

Listing 2.18 Configuration του MultiResourceItemReader

2.3.2 Ανάγνωση απο Βάσεις Δεδομένων

Μαζί με την ανάγνωση από αρχεία, το συνηθέστερο σενάριο ανάγνωσης είναι η ανάγνωση από βάσεις δεδομένων. Εδώ δυο είναι οι προσεγγίσεις: χρήση JDBC ή Object-Relational Mapping (ORM).

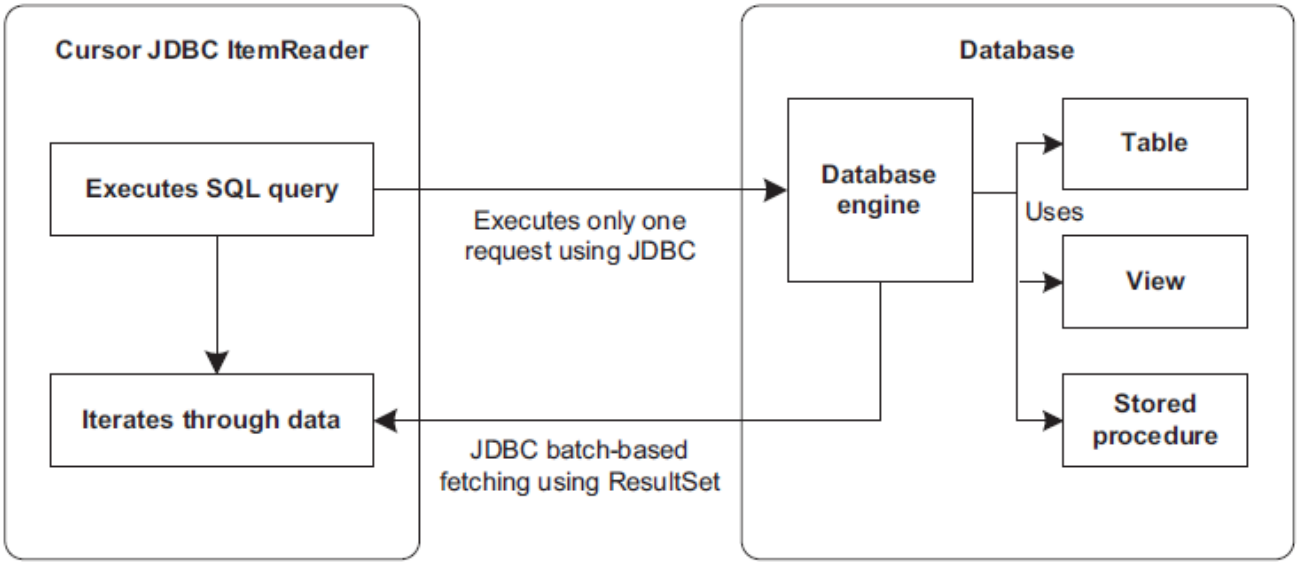
2.3.2.1 JDBC Item Readers

Το Spring Batch βασίζεται στο Spring JDBC layer για να διαβάζει δεδομένα από βάσεις δεδομένων και να μένει ανεξάρτητο από το πλήθος των διαθέσιμων database engines. Χρησιμοποιεί τα **RowMapper** και **PreparedStatement** interfaces και «κρύβει» απο τον χρήστη τον τρόπο που χειρίζεται τα requests προς τη βάση ή τα transactions με αποτέλεσμα το μόνο που συνήθως χρειάζεται να είναι να καθορίσουμε τα queries με τα οποία θα διαβάσουμε τα δεδομένα και τον τρόπο με τον οποίο θα χειριστούμε τα αποτελέσματα.

Ανάγνωση με Cursors

Εδώ, το Spring Batch αφήνει την ευθύνη για την ανάγνωση των δεδομένων στο JDBC **ResultSet** interface. Αυτό το interface είναι στην ουσία ένας database cursor υπεύθυνος για το browsing των αποτελεσμάτων ενός SELECT statement. Όπως φαίνεται και στο Σχήμα 2.7, το Spring Batch εκτελεί μόνο ένα request και

ανακτά τα αποτελέσματα σε batches καθορισμένου μεγέθους ώστε να μπορεί να κρατά διαρκώς απασχολημένες όλες τις φάσεις της διαδικασίας αναγνωση/επεξεργασία/εγγραφή και να επιτυγχάνει την επιθυμητή απόδοση.



Σχήμα 2.6 Ανάγνωση δεδομένων με τον JdbcCursorItemReader

Στην Listing 2.19 δίνεται ένα τυπικό configuration του **JdbcCursorItemReader**. Κατ' ελάχιστο, τα properties που χρειάζεται να ορίσουμε είναι τα **dataSource**, **sql** και **rowMapper** που ορίζουν τη βάση δεδομένων από την οποία διαβάζουμε, τις SQL εντολές που εκτελούμε και την κλάση που χρησιμοποιούμε για να δημιουργήσουμε αντικείμενα από τα αποτελέσματα. Παράδειγμα ενός τέτοιου mapper είναι αυτό της Listing 2.20 που αναλαμβάνει να δημιουργήσει Author instances από το παρεχόμενο ResultSet. Άλλα properties που μπορούν να καθοριστούν εκτός από τα προηγούμενα είναι τα **maxRows** για να περιοριστεί ο αριθμός των αποτελεσμάτων σε ένα μέγιστο αριθμό, **fetchSize** για να ανακτώνται τα αποτελέσματα σε groups καθορισμένου μεγέθους, **queryTimeout** για να καθορίσουμε το μέγιστο χρόνο αναμονής για απόκριση από τη βάση, **preparedStatementSetter** για να ορίσουμε μια κλάση που θα θέτει παραμέτρους του SQL statement κ.α.

```
<bean id="authorsItemReader" class="org.springframework.batch.item.database.JdbcCursorItemReader">
  <property name="dataSource" ref="inputDataSource"/>
  <property name="sql" value="SELECT * FROM ebooks_virtuemart_authors authors"/>
  <property name="rowMapper" ref="authorsRowMapper"/>
</bean>
<bean id="authorsRowMapper" class="(...)"/>
```

Listing 2.19 Configuration του JdbcCursorItemReader

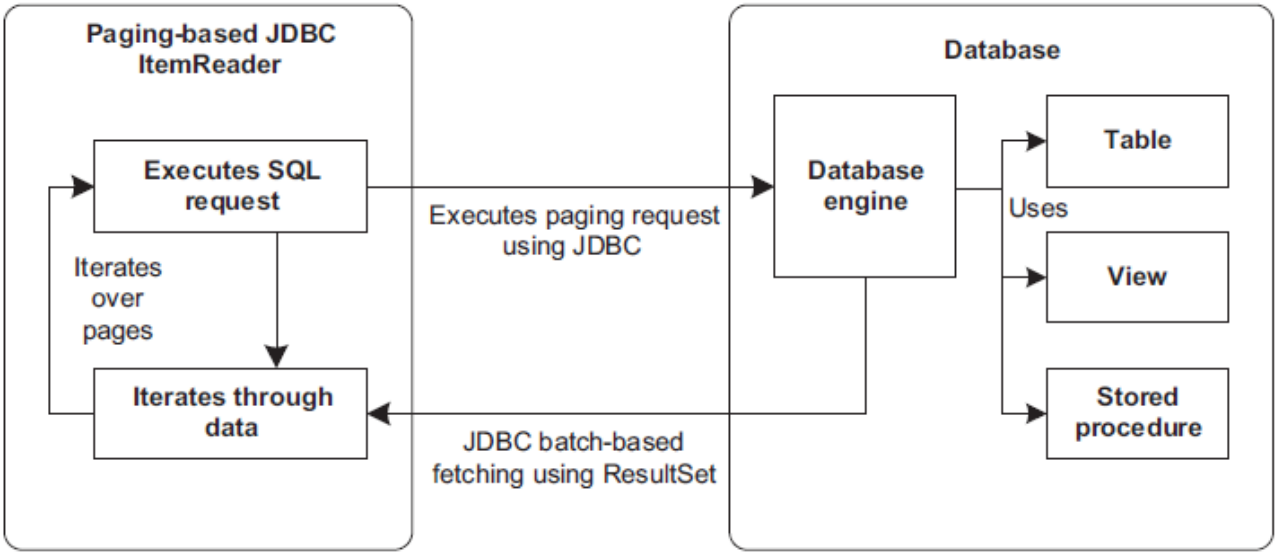
```
public class AuthorsRowMapper implements RowMapper<Author> {
  @Override
  public Author mapRow(ResultSet resultSet, int i) throws SQLException {
    Author author = new Author();
    author.setId(resultSet.getInt("id"));
```

```
        author.setName(resultSet.getString("name"));
        author.setDescription(resultSet.getString("description"));
        return author;
    }
}
```

Listing 2.20 Authors RowMapper

Ανάγνωση με Paging

Εναλλακτικά το Spring Batch μπορεί να χειριστεί την όλη διαδικασία με χρήση paging (βλ. Σχήμα 2.8). Σε αυτή την περίπτωση, εκτελούνται πολλά διαδοχικά requests και τα δεδομένα διαβάζονται σε σελίδες με βάση κάποιο sort key ώστε να είναι δυνατό το να οριστούν τα δεδομένα της κάθε σελίδας. Συγκρινόμενη με τον cursor-based reader αυτή η προσέγγιση είναι περισσότερο απαιτητική καθώς εκτελεί περισσότερα requests προς τη βάση και καταναλώνει περισσότερη μνήμη αλλά κάποιοι JDBC drivers υποστηρίζουν ελλιπώς ή και καθόλου τους cursors. Σε μια τέτοια περίπτωση ο page-based reader είναι μονόδρομος. Πάντως, είτε λόγω ανάγκης, είτε λόγω προτίμησης η επιλογή του reader που θα χρησιμοποιήσουμε δεν επηρεάζει τον κώδικα της εφαρμογής μας και το μόνο που χρειάζεται από τη μεριά μας είναι ελαφρώς διαφορετικό XML configuration.



Σχήμα 2.7 Ανάγνωση δεδομένων με τον JdbcPagingItemReader

Το configuration για τον JdbcPagingItemReader φαίνεται στην Listing 2.21. Ορίζουμε το **dataSource** από το οποίο διαβάζουμε δεδομένα, το **pageSize** και τον **rowMapper** που θα δημιουργεί αντικείμενα από τα αποτελέσματα (δεν χρειάζεται να αλλάξει τίποτα σε σχέση με τον mapper της Listing 2.20). Το query που θα εκτελέσουμε ορίζεται με τη βοήθεια του **SqlPagingQueryProviderFactoryBean** για το οποίο χρειάζεται να ορίσουμε τα **selectClause**, **fromClause**, **whereClause**, **sortKey** κ.α. Ο reader του παραδείγματός μας θα εκτελεί queries σαν το:

SELECT * FROM ebooks_virtuemart_authors WHERE id>? ORDER BY id LIMIT 1000;

μέχρι να εξαντλήσει τα διαθέσιμα αποτελέσματα. Το μέγεθος του **pageSize** δεν πρέπει να είναι ούτε πολύ μικρό (για να μην εκτελούνται πολλά queries), ούτε πολύ μεγάλο (για να μην καταναλώνεται πολύ μνήμη). Ένα μέγεθος γύρω στο 1000 μοιάζει λογικό για τις περισσότερες περιπτώσεις χωρίς όμως αυτό να είναι κανόνας ή να είναι κάτι που μπορεί να καθοριστεί θεωρητικά.

```
<bean id="authorsReader" class="org.springframework.batch.item.database.JdbcPagingItemReader" scope="step">
  <property name="dataSource" ref="inputDataSource"/>
  <property name="queryProvider">
    <bean class="org.springframework.batch.item.database.support.SqlPagingQueryProviderFactoryBean">
      <property name="dataSource" ref="inputDataSource"/>
      <property name="selectClause">
        <value>SELECT *</value>
      </property>
      <property name="fromClause">
        <value>FROM ebooks_virtuemart_authors</value>
      </property>
      <property name="sortKey" value="virtuemart_author_id"/>
    </bean>
  </property>
  <property name="pageSize" value="1000"/>
  <property name="rowMapper" ref="authorsRowMapper"/>
</bean>
```

Listing 2.21 Configuration του JdbcPagingItemReader

2.3.2.2 ORM Item Readers

Τα Object Relational Mapping (ORM) implementations σαν το Hibernate, το Java Persistence API (JPA) ή το iBatis χρησιμοποιούνται συχνά ως τρόπος επικοινωνίας των σύγχρονων Java και Java EE εφαρμογών με τις σχεσιακές βάσεις δεδομένων. Αν και δεν χρησιμοποιήθηκαν ORM readers στα πλαίσια αυτής της εργασίας, αξίζει μια πολύ σύντομη αναφορά στα όσα προσφέρει το Spring Batch.

Όπως και με τους JDBC readers υπάρχουν δυο προσεγγίσεις: η ανάγνωση με cursors και η ανάγνωση σε σελίδες. Η πρώτη περίπτωση υποστηρίζεται μόνο απο το Hibernate καθώς η χρήση cursors προϋποθέτει πως ο κώδικας που είναι υπεύθυνος για την διαχείριση των domain classes δεν χρησιμοποιεί first-level cache. Αυτή η δυνατότητα υποστηρίζεται μόνο από το Hibernate που παρέχει το **StatelessSession** interface (ίδια λειτουργικότητα με το **Session** αλλά χωρίς caching και έλεγχο για dirty state). Σε περίπτωση που χρησιμοποιείται κάποιο άλλο ORM οι page-based readers είναι και εδώ μονόδρομος. Το configuration ενός τέτοιου reader δίνεται ως παράδειγμα στην Listing 2.22. Ο reader διαβάζει εγγραφές από τη βάση και τις μετατρέπει σε αντικείμενα (εδώ Authors) με βάση το κατά Hibernate mapping αυτής της κλάσης.

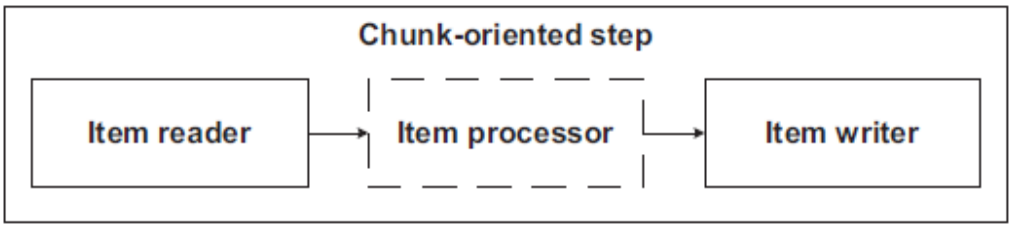
```
<bean id="authorsItemReader" class="org.springframework.batch.item.database.HibernatePagingItemReader">
  <property name="sessionFactory" ref="sessionFactory"/>
  <property name="queryString" value="FROM Author"/>
</bean>
```

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
   (...)
</bean>
```

Listing 2.22 Configuration του HibernatePagingItemReader

2.4 Επεξεργασία Δεδομένων

Τα chunk-oriented βήματα του Spring Batch παρέχουν έναν πολύ βολικό και αποδοτικό τρόπο για την ανάγνωση και την εγγραφή μεγάλου όγκου δεδομένων. Εκτός όμως από την ανάγνωση και την εγγραφή υπάρχει και το business logic της κάθε εφαρμογής που θα πρέπει να ενσωματωθεί στην όλη διαδικασία. Το ιδανικό σημείο για να γίνει αυτό είναι στο στάδιο της επεξεργασίας. Κατά το Spring Batch το στάδιο αυτό είναι ένα προαιρετικό στάδιο μεταξύ ενός item reader και ενός item writer (Σχήμα 2.9). Στόχος του είναι είτε να μετασχηματίσει / εμπλουτίσει τα αντικείμενα που διαβάζει ο item reader, είτε να ελέγξει το αν αυτά πληρούν κάποιες προϋποθέσεις ώστε να φιλτράρει όσα δεν πρέπει να φτάσουν στον item writer.



Σχήμα 2.8 Το προαιρετικό στάδιο επεξεργασίας σε ένα chunk-oriented βήμα

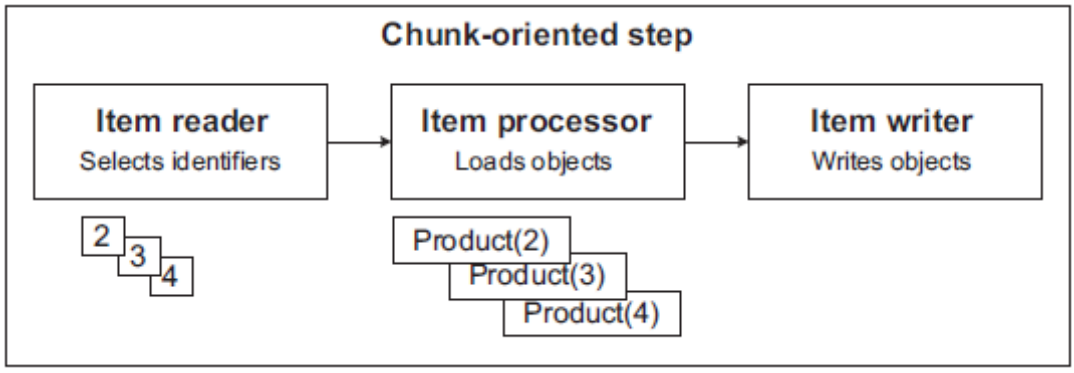
Το interface που παρέχεται είναι το **ItemProcessor** interface και το μόνο που χρειάζεται είναι implementation της **process** μεθόδου του (βλ. σαν παράδειγμα την Listing 2.23). Ο ItemProcessor στέλνει ένα αντικείμενο στην process (ο τύπος του πρέπει να είναι συμβατός με τα αντικείμενα του reader) και η μέθοδος αυτή το επεξεργάζεται επιστρέφοντας είτε το ίδιο είτε κάποιο άλλο αντικείμενο εφόσον η λογική της εφαρμογής μας απαιτεί κάτι τέτοιο. Αν το αντικείμενο δεν πρέπει να σταλθεί στον item writer (επειδή π.χ. δεν είναι valid) το μόνο που χρειάζεται είναι να επιστραφεί null απο την μέθοδο process.

```
public class AuthorsProcessor implements ItemProcessor<Author, Author> {
   @Override
   public Author process(Author author) throws Exception {
      return validAuthor(author) ? author : null;
   }

   private boolean validAuthor(Author author) {
      //Implementation Skipped For Brevity
   }
}
```

Listing 2.23 Παράδειγμα χρήσης του ItemProcessor interface

2.4.1 Μετασχηματισμός και Validation

Ένα ωραίο παράδειγμα για να παρουσιαστούν περισσότερες λεπτομέρειες για την φάση επεξεργασίας είναι η παρουσίαση ενός pattern που χρησιμοποιείται συχνά σε batch εφαρμογές· του driving query pattern. Κάποια database engines τείνουν να χρησιμοποιούν pessimistic locking όταν διαβάζουν μεγάλα σετ δεδομένων χρησιμοποιώντας cursors. Αυτό μπορεί να οδηγήσει σε καθυστερήσεις ή ακόμα και deadlocks αν πέρα απο την Spring Batch εφαρμογή υπάρχουν και άλλοι που διαβάζουν τους ίδιους πίνακες. Το τρικ σε αυτές τις περιπτώσεις είναι, ακολουθώντας το driving query pattern, να διαβάσουμε με τον item reader μόνο τα IDs των αντικειμένων που μας ενδιαφέρουν και κατά το processing να ανακτούμε ένα-ένα τα αντικείμενα για να τα στείλουμε στον writer (βλ. Σχήμα 2.10). Αποφεύγουμε έτσι να κάνουμε ένα μόνο –απαιτητικό– query για να ανακτήσουμε το σύνολο της πληροφορίας και αντ' αυτού εκτελούμε πολλά συντομότερα queries κατά τη φάση της επεξεργασίας.



Σχήμα 2.9 Χρήση του driving-query pattern σε μια Spring Batch εφαρμογή

Το πρώτο που έχουμε να κάνουμε για να εφαρμόσουμε το pattern είναι να ρυθμίσουμε κατάλληλα τον **JdbcCursorItemReader** (Listing 2.24). Το query που θα εκτελεί πρέπει να διαβάζει μόνο τα IDs των αντικειμένων που μας ενδιαφέρουν (εδώ Authors) και να τα επιστρέφει με χρήση του out of the box παρεχόμενου **SingleColumnRowMapper** (που στο παράδειγμά μας ρυθμίζεται με constructor-arg το java.lang.Integer που είναι ο τύπος των IDs).

```
<bean id="authorsItemReader" class="org.springframework.batch.item.database.JdbcCursorItemReader">
  <property name="dataSource" ref="inputDataSource"/>
  <property name="sql"
            value="SELECT virtuemart_author_id FROM ebooks_virtuemart_authors"/>
  <property name="rowMapper">
    <bean class="org.springframework.jdbc.core.SingleColumnRowMapper">
      <constructor-arg value="java.lang.Integer"/>
    </bean>
  </property>
</bean>
```

Listing 2.24 JdbcCursorItemReader για το driving-query pattern

Ο processor θα πρέπει για καθένα από τα Integer IDs που δέχεται ως input να εκτελεί ένα query για να διαβάσει το σύνολο της απαιτούμενης πληροφορίας για αυτόν τον Author, με χρήση ενός mapper να μετασχηματίζει τα αποτελέσματα σε αντικείμενα Author (η υλοποίηση του mapper δεν δίνεται εδώ για λόγους συντομίας) και να τα επιστρέφει (εφόσον πιθανώς τα περάσει από κάποιο validation) ώστε στην επόμενη φάση της διαδικασίας ο writer να τα γράψει στην πηγή εξόδου. Η λογική είναι αυτή που φαίνεται στην Listing 2.25.

```
public class AuthorsProcessor implements ItemProcessor<Integer, Author> {
  private JdbcTemplate jbdcTemplate;
  private static final String SELECT_AUTHOR
          = "SELECT * FROM ebooks_virtuemart_authors WHERE virtuemart_author_id = ?";
  private RowMapper<Author> rowMapper = new AuthorRowMapper();

  @Override
  public Author process(Integer authorId) throws Exception {
    Author author = jdbcTemplate.queryForObject(
      SELECT_AUTHOR, rowMapper, author
    );
    return validAuthor(author) ? author : null;
  }

  //Author's Validation Method and Setters/Getters Skipped For Brevity
}
```

Listing 2.25 Μετασχηματισμός των Integer Author IDs σε Authors κατα τη φάση επεξεργασίας

Το μόνο που πλέον μένει για να ολοκληρωθεί το παράδειγμα είναι να ορίσουμε κατά τα γνωστά στο configuration του job πως το chunk-oriented βήμα θα χρησιμοποιεί για processor τον **AuthorsProcessor** που μόλις παρουσιάστηκε.

Αξίζει να αναφερθεί, πως στην περίπτωση που δεν είναι επιθυμητό το να δημιουργηθεί μια νέα κλάση σαν την παραπάνω για την επεξεργασία των αντικειμένων, διατίθεται ο **ItemProcessorAdapter** σαν ένας τρόπος να ανατεθεί το processing σε κάποια μέθοδο ενός από τα ήδη υπάρχοντα POJOs της εφαρμογής. Το μόνο που χρειάζεται είναι το configuration ενός bean που θα χρησιμοποιεί αυτόν τον adapter και θα ορίζει σαν properties του τα **targetObject** και **targetMethod** (το POJO και τη μέθοδο που επιθυμούμε να καλείται για την επεξεργασία).

Τέλος, για την περίπτωση που κατά το processing μας ενδιαφέρει το validation των δεδομένων παρέχεται και το **ValidatingItemProcessor** interface που έχει δυο ενδιαφέροντα χαρακτηριστικά: α) αναθέτει το validation στο Spring Batch **Validator** interface ώστε εύκολα, αν δεν θέλουμε να γράψουμε custom λογική, να μπορεί να γεφυρωθεί το processing με κάποια dedicated validation language (π.χ. Spring Valang[21]) και β) μέσω του property **filter** διακρίνει εννοιολογικά το αν ο processor πρέπει να κάνει filter ή skip τα

[21] Πληροφορίες για την Valang εδώ: http://www.springbyexample.org/examples/spring-modules-validation-module.html

αντικείμενα που δεν πληρούν τις απαραίτητες προϋποθέσεις. Χρησιμοποιώντας το default **filter = false** ορίζουμε πως ένα λάθος αντικείμενο θα πρέπει να εγείρει κάποιο validation exception που το XML configuration του chunk-oriented βήματος θα αποφασίζει αν θα είναι μέσα σε αυτά που δικαιολογείται να συναντώνται ή όχι (**skippable-exception-classes**), διαφορετικά χρησιμοποιώντας το filter = true ορίζουμε πως τα λάθος αντικείμενα απλά δεν στέλνονται ποτέ στον writer (ο processor επιστρέφει null). Εύκολα λοιπόν, καθορίζουμε πότε ένα αντικείμενο θεωρείται skipped και πότε filtered, και, χωρίς να χρειαστεί να κάνουμε τίποτα άλλο, η πληροφορία για το step execution καταγράφεται στο job repository ώστε να έχουμε εικόνα για το τι έχει συμβεί.

2.4.2 Chaining Processors

Στην περίπτωση που η λογική που πρέπει να υλοποιηθεί στην φάση της επεξεργασίας είναι πολύ σύνθετη (υπάρχουν για παράδειγμα πολλοί business κανόνες που πρέπει να εφαρμοστούν ή πρέπει να γίνουν τόσο μετασχηματισμοί όσο και validations των δεδομένων) παρέχεται η δυνατότητα για chaining των item processors με χρήση του **CompositeItemProcessor**. Ορίζουμε μια λίστα από processors οι οποίοι καλούνται ο ένας μετά τον άλλο (ο επιστρεφόμενος τύπος κάθε κρίκου της αλυσίδας πρέπει να είναι συμβατός με τον τύπο που περιμένει σαν είσοδο ο επόμενος) και έτσι μπορούμε να υλοποιήσουμε καλύτερα δομημένη λογική ή ακόμα και να επαναχρησιμοποιήσουμε κομμάτια κώδικα (processors) που ήδη έχουν υλοποιηθεί και χρησιμοποιούνται και κάπου αλλού. Παράδειγμα για το configuration ενός CompositeItemProcessor φαίνεται στη Listing 2.26.

```
<bean id="processor" class="org.springframework.batch.item.support.CompositeItemProcessor">
  <property name="delegates">
    <list>
      <ref bean="authorTransformProcessor" />
      <ref bean="authorValidationProcessor" />
    </list>
  </property>
</bean>
<bean id="authorTransformProcessor" class="(...)"/>
<bean id="authorValidationProcessor" class="(...)"/>
```

Listing 2.26 Configuration του CompositeItemProcessor

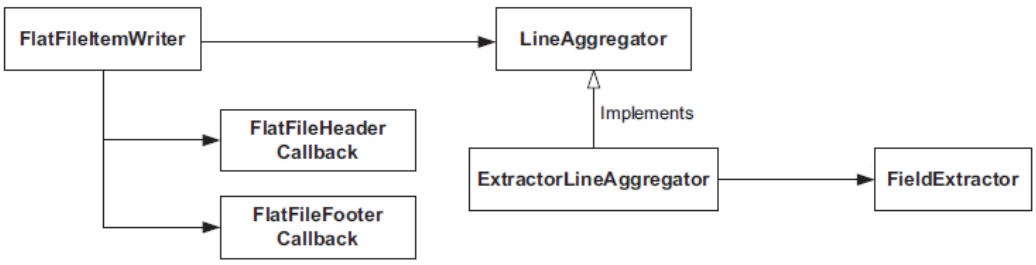
2.5 Εγγραφή Δεδομένων

Ο τελευταίος κρίκος στην αλυσίδα ενός chunk-oriented βήματος είναι η φάση της εγγραφής. Όπως και για την φάση της ανάγνωσης παρέχονται έτοιμες κλάσεις και interfaces που καλύπτουν τα συνηθέστερα σενάρια: εγγραφή σε αρχεία, βάσεις δεδομένων, Java Message Services (JMS), ενώ εύκολα δίνεται και η

δυνατότητα επέκτασης του core writing interface (**ItemWriter**) στην περίπτωση που οι απαιτήσεις μας δεν καλύπτονται από τα προηγούμενα. Ο ItemWriter έχει μια μόνο μέθοδο, την **write**, που στις περισσότερες των περιπτώσεων παίρνει μια λίστα από αντικείμενα και τα χειρίζεται ως ένα ώστε να επιτύχει καλύτερη απόδοση. Αν για παράδειγμα μιλάμε για writer σε βάση δεδομένων, με τον τρόπο αυτό το Spring Batch χειρίζεται σε κοινό transaction ένα σετ αντικειμένων, αποφεύγοντας τα άσκοπα round-trips στη βάση. Το μέγεθος αυτού του σετ ορίζεται όπως έχει ήδη αναφερθεί από το **commit-interval** του chunk-oriented βήματος.

2.5.1 Εγγραφή σε Αρχεία

Οι writers που παρέχονται για την εγγραφή σε αρχεία καλύπτουν (σε αντιστοιχία με τους readers αρχείων) flat αρχεία (fixed μήκος για τα fields ή delimited text), XML κ.α. Τα αρχεία μπορεί να απαιτείται να έχουν headers ή/και footers, οι εγγραφές τους να πρέπει να καταλαμβάνουν περισσότερες από μια γραμμές, να υπάρχουν εγγραφές σχόλια κτλ. Για την εγγραφή σε flat αρχεία χρησιμοποιείται ο **FlatFileItemWriter** που ακολουθεί το lifecycle του **ItemStream** interface το οποίο και υλοποιεί. Μόλις το Spring Batch ανοίγει το stream καλεί την **FlatFileHeaderCallback** για να γράψει (προαιρετικά) το header του αρχείου, και πριν το κλείσει καλεί την **FlatFileFooterCallback** για να γράψει (πάλι προαιρετικά) το footer. Για την μετατροπή των αντικειμένων σε strings ο writer χρησιμοποιεί το **LineAggregator** interface. Ένα συχνά χρησιμοποιούμενο implementation του είναι το **PassThroughLineAggregator** που απλά καλεί την toString() μέθοδο των αντικειμένων προς εγγραφή. Αν απαιτείται κάτι περισσότερο από την toString(), υπάρχουν και άλλα implementations όπως το **ExtractorLineAggregator** (βλ. το Σχήμα 2.11), το οποίο με χρήση ενός **FieldExtractor** αναλαμβάνει να δημιουργήσει ένα πίνακα αντικειμένων (π.χ. Strings) δοθέντος ενός domain αντικειμένου προς εγγραφή.



Σχήμα 2.10 Κλάσεις και interfaces που εμπλέκονται στην εγγραφή flat αρχείων

Στην Listing 2.27 δίνεται ένα παράδειγμα για το configuration του FlatFileItemWriter. Με το property **resource** ορίζουμε το path για το αρχείο που θα παραχθεί και με το **lineAggregator** τον LineAggregator που θα χρησιμοποιηθεί (εδώ ο PassThroughLineAggregator). Άλλα properties μπορούν να καθορίσουν το **encoding** του αρχείου, τα **footerCallback** και **headerCallback** που είδαμε στα προηγούμενα, το αν θα πρέπει

να διαγραφεί τυχόν αρχείο που υπάρχει με το ίδιο όνομα όταν ξεκινήσει η διαδικασία (**shouldDeleteIfExists**), το αν θα πρέπει να δημιουργηθεί –άδειο– αρχείο στην περίπτωση που δεν υπάρχουν δεδομένα για εγγραφή (**shouldDeleteIfEmpty**) κ.α.

```
<bean id="authorsItemWriter" class="org.springframework.batch.item.file.FlatFileItemWriter">
  <property name="resource" value="file:/var/data/authors.txt"/>
  <property name="lineAggregator">
    <bean class="org.springframework.batch.item.file.transform.PassThroughLineAggregator" />
  </property>
</bean>
```

Listing 2.27 Configuration του FlatFileItemWriter

Περισσότερο ενδιαφέρον έχει εδώ να παρουσιαστεί σαν παράδειγμα το πως θα μπορούσε να υλοποιηθεί ένας FlatFileItemWriter που γράφει ένα delimited-field αρχείο με τη χρήση του DelimitedLineAggregator interface και ενός custom FieldExtractor (έστω AuthorFieldExtractor). Το configuration του writer είναι αυτό που φαίνεται στην Listing 2.28. Εδω, πέρα από τα όσα παρουσιάστηκαν μόλις στην Listing 2.27, για τον DelimitedLineAggregator ορίζουμε τον χαρακτήρα που θα χρησιμοποιηθεί για διαχωριστικό των fields (χαρακτήρας '|') και την custom κλάση που θα αναλάβει το ρόλο του FieldExtractor (βλ. Listing 2.29).

```
<bean id="authorsItemWriter" class="org.springframework.batch.item.file.FlatFileItemWriter">
  <property name="resource" value="file:/var/data/authors.txt"/>
  <property name="lineAggregator">
    <bean class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
      <property name="delimiter" value="|">
      <property name="fieldExtractor">
        <bean class="gr.booksAdmin.batch.AuthorFieldExtractor" />
      </property>
    </bean>
  </property>
</bean>
```

Listing 2.28 Configuration FlatFileItemWriter με χρήση DelimitedLineAggregator

Ο custom FieldExtractor πρέπει απλά να υλοποιεί την **extract** μέθοδο του FieldExtractor interface και να επιστρέφει ένα πίνακα αντικειμένων με τα fields που τελικά μας ενδιαφέρει να εγγραφούν στο αρχείο. Η έξοδος αυτού του writer, αν θεωρήσουμε πως μας ενδιαφέρουν μόνο τα Id και Name του κάθε Author θα είναι εγγραφές σαν την "100|Kurt Vonnegut".

```
public class AuthorFieldExtractor implements FieldExtractor<Author> {
  @Override
  public Object[] extract(Author author) {
    return new Object [] {
      author.getId(),
      author.getName()
    };
  }
}
```

Listing 2.29 AuthorFieldExtractor για τον καθορισμό των fields προς εγγραφή

Αξίζει να αναφερθεί πως σε αντιστοιχία με τον ItemProcessorAdapter που μπορεί να χρησιμοποιηθεί κατά το processing αν δεν θέλουμε να δημιουργήσουμε μια νέα κλάση μόνο για αυτό το σκοπό, υπάρχει και για την φάση της εγγραφής ένας παρόμοιος adapter, ο **ItemWriterAdapter**. Και εδώ, τα properties που πρέπει να οριστούν είναι τα **targetObject** και **targetMethod** για το POJO και την μέθοδο αυτού αντίστοιχα που θα αναλάβουν την εγγραφή.

Κλείνοντας , στην περίπτωση που θέλουμε να γράψουμε σε περισσότερα από ένα αρχεία (επειδή για παράδειγμα θέλουμε ένα μέγιστο αριθμό εγγραφών ανα αρχείο) μπορούμε να χρησιμοποιήσουμε τον **MultiResourceItemWriter** (βλ. Listing 2.30) καθορίζοντας τον μέγιστο αριθμό εγγραφών ανά αρχείο (**itemCountLimitPerResource**), το **resource** των αρχείων (δημιουργείται ένα index ως suffix, για παράδειγμα εδώ θα δημιουργηθούν τα authors.txt.1, authors.txt.2 κοκ) και τον writer στον οποίο θέλουμε να κάνουμε **delegate** την εγγραφή.

```
<bean id="multiResourcesItemWriter" class="org.springframework.batch.item.file.MultiResourceItemWriter">
   <property name="resource" value="file:/var/data/authors.txt"/>
   <property name="itemCountLimitPerResource" value="10000" />
   <property name="delegate" ref="authorsItemWriter" />
</bean>
<bean id="authorsItemWriter" class="(...)">
   (...)
</bean>
```

Listing 2.30 Configuration του MultiResourceItemWriter

2.5.2 Εγγραφή σε Βάσεις Δεδομένων

Η εγγραφή σε βάσεις δεδομένων μπορεί να χωριστεί και αυτή σε δυο κατηγορίες: εγγραφή με JDBC ή εγγραφή με κάποιο ORM εργαλείο. Παρουσιάζουμε εν συντομία τις δυο προσεγγίσεις καθώς και το πως θα μπορούσαμε να υλοποιήσουμε τον δικό μας, custom item writer.

2.5.2.1 JDBC Item Writers

Σε αυτή την προσέγγιση χρησιμοποιείται η **JdbcBatchItemWriter** που είναι implementation του ItemWriter για JDBC το οποίο βασίζεται στο Spring JDBC layer «κρύβοντας» απο τον χρήστη οτιδήποτε περίπλοκο θα είχε να αντιμετωπίσει αν χρησιμοποιούσε μόνος του το συγκεκριμένο API. Ένα τυπικό παράδειγμα για το configuration αυτού του writer είναι αυτό που φαίνεται στην Listing 2.31. Θέτουμε το property **assertUpdates** σε true ώστε το Spring Batch να εγείρει EmptyResultDataAccessException στην περίπτωση που το sql statement δεν κάνει πραγματικά update κάποιο row, καθορίζουμε το **dataSource** και το **sql** statement που θέλουμε να εκτελέσουμε και ορίζουμε τον τρόπο με τον οποίο θα τεθούν οι παράμετροι

αυτού του query. Ο **BeanPropertyItemSqlParameterSourceProvider** που έχουμε χρησιμοποιήσει στην Listing 2.31, κατά την εκτέλεση του θα κάνει mapping των παραμέτρων (:id, :name) με τα attribute names του προς εγγραφή αντικειμένου Author.

```
<bean id="authorsItemWriter" class="org.springframework.batch.item.database.JdbcBatchItemWriter">
  <property name="assertUpdates" value="true" />
  <property name="itemSqlParameterSourceProvider">
    <bean class="org.springframework.batch.item.database.BeanPropertyItemSqlParameterSourceProvider" />
  </property>
  <property name="sql"
    value="INSERT INTO CB_AUTHOR (ID, NAME) VALUES(:id, :name)" />
  <property name="dataSource" ref="outputDataSource" />
</bean>
```

Listing 2.31 Configuration του JdbcBatchItemWriter

Αντί του itemSqlParameterSourceProvider, στην περίπτωση που είχαμε positional query parameters, θα μπορούσε να χρησιμοποιηθεί το property **itemPreparedStatementSetter**. Θα υλοποιούσαμε δηλαδή έναν **ItemPreparedStatementSetter** σαν αυτόν της Listing 2.32 και στο configuration του writer θα ορίζαμε πως αυτός θα αναλάμβανε να θέσει τις παραμέτρους του query.

```
public class AuthorItemPreparedStatementSetter implements ItemPreparedStatementSetter<Author> {

  @Override
  public void setValues(Author author, PreparedStatement ps) throws SQLException {
    ps.setString(1, author.getId());
    ps.setString(2, author.getName());
  }
}
```

Listing 2.32 Παράδειγμα implementation του ItemPreparedStatementSetter

2.5.2.2 ORM Item Writers

Εδώ παρέχονται οι **HibernateItemWriter**, **JpaItemWriter** και **IbatisBatchItemWriter**. Θεωρώντας πως το ORM mapping των domain αντικειμένων υπάρχει, το να χρησιμοποιήσουμε κάποιον από αυτούς τους writers είναι τόσο απλό όσο αυτό που παρουσιάζεται στην Listing 2.33 για την περίπτωση του Hibernate. Το Hibernate αναλαμβάνει να διαβάσει το υπό εγγραφή αντικείμενο από την βάση εφόσον δεν βρίσκεται ήδη στο session (SELECT) και ανάλογα με το αν αυτό υπάρχει ή όχι να εκτελέσει INSERT ή UPDATE εντολές (το implementation χρησιμοποιεί την **saveOrUpdate**). Προφανώς όλη η λογική ενός ORM εργαλείου (fetch στρατηγικές για να αποφασιστεί αν πρέπει να διαβαστεί ή όχι ένα αντικείμενο, loading των related οντοτήτων κτλ) κάνουν την προσέγγιση των ORM writers περισσότερο απαιτητική από άποψη πόρων σε σχέση με τους απλούστερους JDBC writers.

```
<bean id="authorsItemWriter" class="org.springframework.batch.item.database.HibernateItemWriter">
  <property name="hibernateTemplate" ref="hibernateTemplate" />
</bean>
```

Listing 2.33 Configuration του HibernateItemWriter

2.5.2.3 Custom Item Writers

Τέλος, υπάρχει και η δυνατότητα για έναν custom writer στην περίπτωση που καμία από τις δυο προσεγγίσεις δεν ταιριάζει στις απαιτήσεις μας. Παράδειγμα ενός τέτοιου JDBC writer φαίνεται στην Listing 2.34. Αρκεί να κάνουμε implement την μέθοδο **write** του ItemWriter interface που να δέχεται μια λίστα απο αντικείμενα σαν όρισμα και να εκτελεί την custom λογική μας για καθένα από αυτά (π.χ. προσπάθεια για UPDATE μιας εγγραφής και αν δεν υπάρχουν affected εγγραφές εκτέλεση μιας INSERT εντολής).

```
public class AuthorsWriter implements ItemWriter<Author> {
  private JdbcTemplate jdbcTemplate;
  private static final String UPDATE = "UPDATE CB_AUTHOR SET NAME=? WHERE ID=?";
  private static final String INSERT = "INSERT INTO CB_AUTHOR(ID,NAME) VALUES(?,?)";

  @Override
  public void write(List<? extends Author> authors) throws Exception {
    for(Author author : authors){
      int affected = jdbcTemplate.update(
        UPDATE, author.getName(), author.getId()
      );
      if(affected==0) {
        jdbcTemplate.update(
          INSERT, author.getId(), author.getName()
        );
      }
    }
  }
  //Setters and Getters Skipped For Brevity
}
```

Listing 2.34 Παράδειγμα implementation του ItemWriter

2.5.3 Composite Item Writer

Υπάρχουν περιπτώσεις που η δυνατότητα για μόνο έναν writer σε ένα chunk-oriented βήμα είναι περιοριστική. Τυπικό παράδειγμα η περίπτωση του να θέλουμε να γράψουμε τα ίδια δεδομένα και σε fixed-length και σε delimited-field αρχεία, ή το να θέλουμε να γράψουμε και σε αρχείο και σε βάση ταυτόχρονα. Για τέτοιες περιπτώσεις έχει προβλεφθεί ο **CompositeItemWriter**. Αυτός αναθέτει την εγγραφή σε μια λίστα από writers που ορίζονται με μια προκαθορισμένη σειρά (η σειρά με την οποία θα καλείται ο κάθε writer). Παράδειγμα τέτοιου configuration δίνεται στην Listing 2.35.

```
<bean id="authorsItemWriter" class="org.springframework.batch.item.support.CompositeItemWriter">
  <property name="delegates">
    <list>
      <ref local="delimitedProductItemWriter"/>
      <ref local="fixedWidthProductItemWriter"/>
    </list>
  </property>
</bean>
<bean id="delimitedProductItemWriter" class="(...)">
  (...)
</bean>
<bean id="fixedWidthProductItemWriter" class="(...)">
  (...)
</bean>
```

Listing 2.35 Configuration του CompositeItemWriter

2.6. Bulletproof Jobs

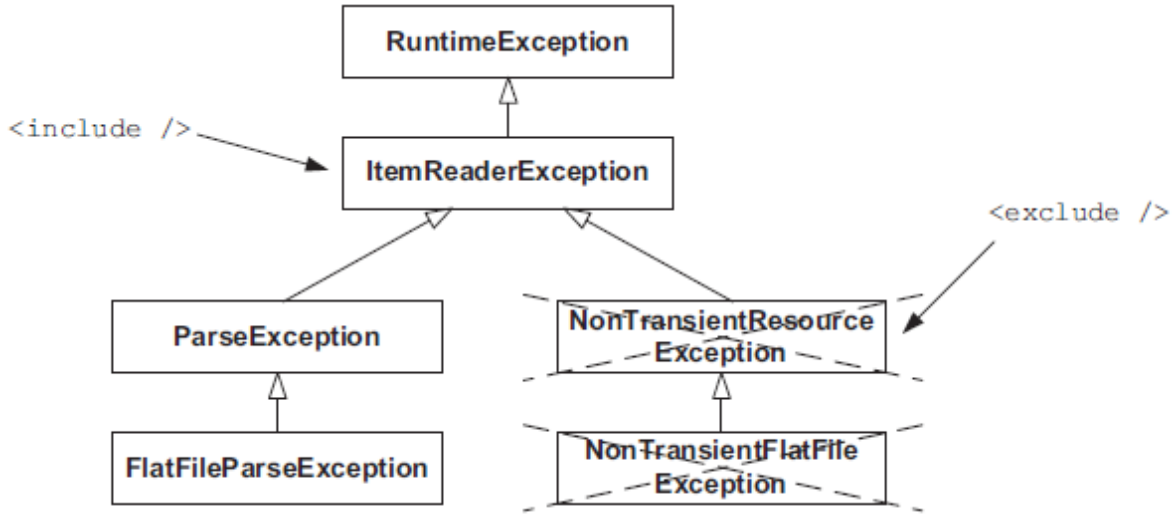
Ένα απο τα χαρακτηριστικά που κάνουν το Spring Batch να ξεχωρίζει είναι το ότι παρέχει μηχανισμούς ώστε να αντιμετωπίζει αυτόματα τα λάθη που μπορεί να προκύψουν κατά την εκτέλεση ενός job. Τα exceptions που προκύπτουν δεν είναι απαραίτητα καίρια· κάποια μπορεί να έχουν μικρή σημασία και άρα να θέλουμε να τα κάνουμε skip χωρίς το job να αποτύχει (π.χ. αν 1-2 εγγραφές ενός αρχείου δεν έχουν το αναμενόμενο φορμάτ δεν χρειάζεται να αποτύχει όλο το αρχείο), ενώ άλλα μπορεί να είναι παροδικά και να θέλουμε να ξαναπροσπαθήσουμε πριν ακυρώσουμε το job (π.χ. ένα deadlock κατά την προσπάθεια εγγραφής σε βάση). Επειδή όμως κάποια στιγμή αναπόφευκτα θα παρουσιαστεί και κάποιο fatal exception που θα κάνει το job μας να αποτύχει, θα πρέπει να υπάρχει επιπλέον η δυνατότητα για restart και μάλιστα χωρίς απαραίτητα να ξαναεκτελέσουμε steps που η αποτυχημένη εκτέλεση είχε ολοκληρώσει επιτυχώς. Στα ακόλουθα παρουσιάζεται το τι παρέχει το Spring Batch σχετικά με όλα αυτά.

2.6.1 Skip

Ο συνηθέστερος τρόπος για να ορίσουμε τα exceptions που πρέπει να γίνονται skip είναι το configuration που φαίνεται στην Listing 2.36. Ορίζουμε με το **include** του **skippable-exception-classes** element τα exceptions που θέλουμε να γίνονται skip (και όλα τα subclasses τους, π.χ. εδώ όλα τα subclasses του FlatFileParseException) και με το attribute **skip-limit** τον μέγιστο αριθμό αντικειμένων που μπορούν να εγείρουν αυτό το σφάλμα πριν το job αποτύχει (π.χ. εδώ μέχρι 10 εγγραφές του αρχείου). Στην περίπτωση που θέλουμε κάποιο subclass στην ιεραρχία των exceptions που γίνονται skip να θεωρηθεί fatal, μπορούμε να το δηλώσουμε στο skippable-exception-classes με το element **exclude** (π.χ. skip τα ItemReaderException αλλά όχι τα NonTransientResourceException, βλ. Σχήμα 2.12).

```
<job id="inventoryJob">
  <step id="inventoryStep">
    <tasklet>
      <chunk reader="inventoryReader" writer="writer" commit-interval="1000" skip-limit="10">
        <skippable-exception-classes>
          <include class="org.springframework.batch.item.file.FlatFileParseException" />
        </skippable-exception-classes>
      </chunk>
    </tasklet>
  </step>
</job>
```

Listing 2.36 Skippable exceptions σε ένα chunk-oriented βήμα



Σχήμα 2.11 <include /> και <exclude /> elements του skippable-exception-classes

Τα παραπάνω είναι αυτό που το Spring Batch ονομάζει **LimitCheckingItemSkipPolicy**, ενώ έτοιμα διατίθενται και άλλα skip policies όπως τα **AlwaysSkipItemSkipPolicy** (χωρίς skip-limit και άσχετα με τον τύπο του exception), **NeverSkipItemSkipPolicy** και **ExceptionClassifierSkipPolicy** (συνδυάζει πολλά policies και ο τύπος του exception είναι αυτός που καθορίζει ποιο θα χρησιμοποιηθεί). Σε κάθε περίπτωση, αν οι απαιτήσεις μας δεν καλύπτονται απο τα προηγούμενα μπορούμε εύκολα να κάνουμε implement την **shouldSkip** μέθοδο του **SkipPolicy** interface.

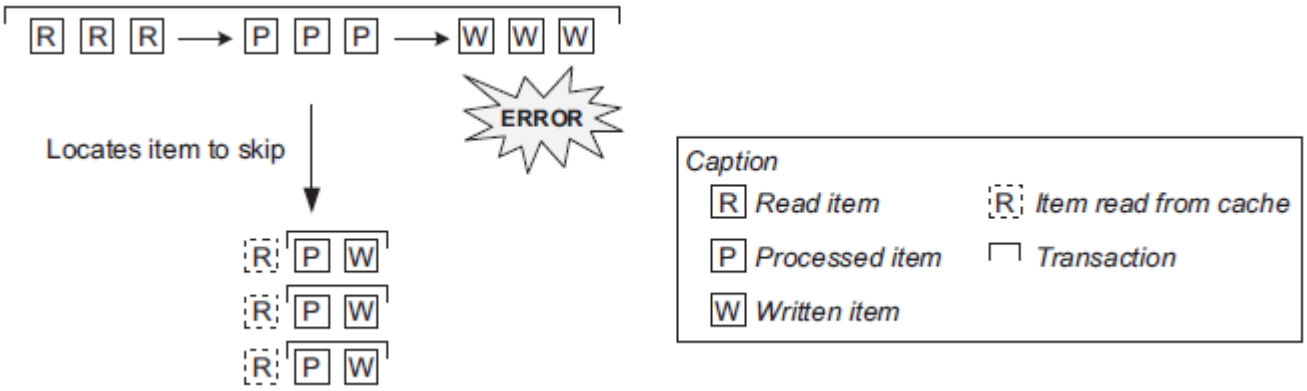
Επειδή τα αντικείμενα που γίνονται skip θα πρέπει με κάποιο τρόπο να καταγράφονται ώστε να είναι δυνατόν ο χρήστης να βλέπει τι έχει συμβεί και να μπορεί να επέμβει διορθωτικά, μπορούμε να υλοποιήσουμε τις μεθόδους του **StepListener** interface (**onSkipInRead**, **onSkipInProcess**, **onSkipInWrite**) ή να χρησιμοποιήσουμε τα annotations @OnSkipInRead, @OnSkipInProcess, @OnSkipInWrite σε ένα απλό POJO (βλ. Listing 2.37) που θα το προσθέσουμε στους listeners του chunk-oriented βήματος.

```
public class SkipListener {
  @OnSkipInWrite
  public void logSkips(Object item, Throwable t) {
```

```
    if(t instanceof FlatFileParseException) {
      //Log To A File Or A Database
    }
  }
}
```

Listing 2.37 Παράδειγμα χρήσης του @OnSkipInWrite annotation

Κλείνοντας, πρέπει να αναφερθεί ο διαφορετικός τρόπος με τον οποίο το Spring Batch χειρίζεται ένα exception σε καθεμία από τις τρεις φάσεις ενός chunk-oriented βήματος. Αν ένα skippable exception εγερθεί στην φάση της ανάγνωσης ο reader θα προσπαθήσει απλά να διαβάσει το επόμενο αντικείμενο. Δεν υπάρχει rollback στο transaction. Αντιθέτως, αν το exception εγερθεί στη φάση της επεξεργασίας ο processor θα κάνει rollback το transaction, θα αφαιρέσει από το chunk το προβληματικό αντικείμενο και θα στείλει τα υπόλοιπα εκ νέου για επεξεργασία. Τα πράγματα είναι περισσότερο σύνθετα στην περίπτωση που το exception παρουσιαστεί κατά την εγγραφή. Επειδή το framework δεν έχει τρόπο να γνωρίζει ποιο αντικείμενο από την λίστα που παίρνει ο writer ως όρισμα ήταν αυτό που οδήγησε στο exception, κάνει rollback, και ξεκινά να επεξεργάζεται κάθε ένα από τα αντικείμενα μεμονωμένα στο δικό του transaction. Τα αντικείμενα δεν χρειάζεται να διαβαστούν ξανά καθώς υπάρχει caching για το chunk στην οποία ο writer ανατρέχει σε αυτή την περίπτωση. Η απεικόνιση αυτής της διαδικασίας φαίνεται στο Σχήμα 2.13.



Σχήμα 2.12 Χειρισμός των exceptions κατά την φάση της εγγραφής

2.6.2 Retry

Σε αντίθεση με το παράδειγμα των λάθος εγγραφών ενός αρχείου προς ανάγνωση που οδηγούν σε ντετερμινιστικά exceptions, υπάρχουν περιπτώσεις όπου το exception μπορεί να είναι παροδικό. Κλασικό παράδειγμα είναι το lock σε κάποια εγγραφή της βάσης στην οποία γράφουμε ή το πρόβλημα δικτύου κατά την προσπάθεια επικοινωνίας με ένα web service. Σε τέτοιες περιπτώσεις είναι εξαιρετικά χρήσιμη η δυνατότητα για retry που παρέχεται από το Spring Batch. Σε αντιστοιχία με το skippable-exception-classes ορίζεται το element **retryable-exception-classes** με το configuration του να είναι σαν αυτό που φαίνεται στην Listing 2.38. Και εδώ ορίζουμε με το **include** τα exceptions που θέλουμε να γίνονται retry (μαζί με τα

subclasses τους, αν θέλουμε να εξαιρέσουμε κάποιο μπορούμε να χρησιμοποιήσουμε το **exclude**) και με το **retry-limit** το πόσες φορές θα πρέπει να ξαναγίνει προσπάθεια. Αυτό το policy είναι το **SimpleRetryPolicy**, το default από τα τρία που παρέχονται. Τα άλλα δυο είναι το **TimeoutRetryPolicy** και το **ExceptionClassifierRetryPolicy** (συνδυάζει πολλά policies και ο τύπος του εγειρόμενου exception καθορίζει ποιο θα χρησιμοποιηθεί).

Τέλος, υπάρχουν και εδώ interfaces για το listening των retries ώστε να μπορούμε, για παράδειγμα, να καταγράφουμε τα αντικείμενα για τα οποία χρειάστηκε εκ νέου προσπάθεια. Αυτά είναι τα **RetryListener** (με τις μεθόδους **open** και **close**) και ο adapter αυτού με όνομα **RetryListenerSupport** (με την μέθοδο onError).

```
<job id="authorsJob">
  <step id="authorsStep">
    <tasklet>
      <chunk reader="reader" writer="writer" commit-interval="1000" retry-limit="3">
        <retryable-exception-classes>
          <include class="org.springframework.dao.OptimisticLockingFailureException" />
        </retryable-exception-classes>
      </chunk>
    </tasklet>
  </step>
</job>
```

Listing 2.38 Retryable exceptions σε ένα chunk-oriented βήμα

2.6.3 Restart

Αναπόφευκτα, κάποια στιγμή, κάποιο job θα αποτύχει. Το Spring Batch παρέχει την δυνατότητα για restart και συνέχιση από το σημείο στο οποίο το αποτυχημένο job σταμάτησε. Η συμπεριφορά του framework για αυτή την περίπτωση καθορίζεται από τρία configurations. Το **restartable** attribute που ορίζεται σε επίπεδο job και καθορίζει αν το job μπορεί να επανεκκινηθεί ή όχι (το default είναι true) και τα **allow-start-if-complete** και **start-limit** που ορίζονται σε επίπεδο tasklet και καθορίζουν το αν το tasklet πρέπει να επαναληφθεί ακόμα και αν ολοκληρώθηκε επιτυχώς (παρά την αποτυχία του job) (το default είναι το false) και τον μέγιστο αριθμό restarts που μπορούν να γίνουν για ένα step (το default είναι το Integer.MAX_VALUE).

Για να γίνουν κατανοητά τα παραπάνω, ας θεωρήσουμε το παράδειγμα ενός job με δυο βήματα: το πρώτο λαμβάνει ένα συμπιεσμένο αρχείο που περιέχει το CSV αρχείο που θέλουμε να διαβάσουμε και το δεύτερο είναι ένα κλασικό chunk-oriented βήμα που διαβάζει το CSV αρχείο και το γράφει σε μια βάση δεδομένων. Έστω ότι το πρώτο step επιτυγχάνει αλλά αποτυγχάνει το δεύτερο μετά από κάποια chunks. Το Spring Batch μπορεί με το restart να συνεχίσει το δεύτερο βήμα από το chunk στο οποίο σταμάτησε και μετά (όλη η

απαραίτητη πληροφορία υπάρχει στα metadata του job). Το πρώτο βήμα όμως πρέπει να ξαναεκτελεστεί; Αν θεωρήσουμε πως χρήστης θα επέμβει διορθωτικά στο αρχείο και θα κάνει restart το job για να συνεχίσει, το πρώτο βήμα δεν πρέπει να εκτελεστεί (και by default δεν θα εκτελεστεί, το allow-start-if-complete είναι false). Υπάρχει όμως περίπτωση ο χρήστης να ζητήσει από τον πάροχο του αρχείου να το διορθώσει και να το ξαναστείλει. Σε αυτή την περίπτωση θα πρέπει και το πρώτο βήμα να ξαναεκτελεστεί. Η απάντηση δηλαδή είναι θέμα business απόφασης. Κλείνουμε την ενότητα αυτή δίνοντας το configuration της δεύτερης από αυτές τις περιπτώσεις (Listing 2.39). Σε αυτό έχει τεθεί και το attribute start-limit ίσο με 3 στο δεύτερο step ώστε να καθορίσουμε τον μέγιστο επιτρεπτό αριθμό επανεκκινήσεων.

```
<job id="inventoryJob">
  <step id="decompressStep" next="readWriteStep">
    <tasklet allow-start-if-complete="true">
      (...)
    </tasklet>
  </step>
  <step id="readWriteStep">
    <tasklet start-limit="3">
      (...)
    </tasklet>
  </step>
</job>
```

Listing 2.39 Επανεκτέλεση ενός επιτυχημένου step

2.6.4. Διαχείριση Transactions

Η δυνατότητα για skip κάποιων exceptions ή restart ενός αποτυχημένου job είναι προφανώς ένα απο τα πολύ δυνατά χαρακτηριστικά του Spring Batch. Η συζήτηση όμως δεν μπορεί να ολοκληρωθεί αν δεν αναφερθούμε στον τρόπο που το framework έχει για να διαχειρίζεται τα transactions. Transactional κώδικας μπορεί να εκτελείται κατά την εκτέλεση ενός chunk-oriented βήματος, από τον κώδικα ενός tasklet ή ακόμα και από έναν listener που μπορεί να χρησιμοποιείται ώστε να καταγράφουμε σε μια βάση δεδομένων τα αντικείμενα που γίνονται skip κατά το processing. Το Spring Batch έχει διαφορετικά defaults για κάθε μια από αυτές τις περιπτώσεις.

Ένα tasklet για παράδειγμα υλοποιεί όπως έχουμε δει την **execute** του **Tasklet** interface (βλ. Listing 2.40) και επαναλαμβάνεται όσο η μέθοδος αυτή επιστρέφει RepeatStatus.CONTINUABLE. Μόλις επιστραφεί null ή RepeatStatus.FINISHED το tasklet ολοκληρώνεται και το job συνεχίζει με το επόμενο βήμα. Κάθε κλήση της execute εκτελείται by default μέσα στο δικό της transaction. Αν η χρήση transactions είναι περιττή για το tasklet (επειδή π.χ. το μόνο που κάνει είναι αποσυμπίεση ενός αρχείου όπως στο παράδειγμα που μόλις αναφέρθηκε) μπορούμε να θέσουμε το propagation level του σε PROPAGATION_NEVER.

```
public class MyTasklet implements Tasklet {

   @Override
   public RepeatStatus execute(
         StepContribution contribution, ChunkContext chunkContext) throws Exception {
      //Custom Processing Here
      return RepeatStatus.FINISHED;
   }
}
```

Listing 2.40 Παράδειγμα implementation του Tasklet interface

Αντίθετα, για ένα chunk-oriented βήμα που μπορεί να κάνει rollback λόγω κάποιου exception στη φάση της επεξεργασίας ή της εγγραφής, το transaction δημιουργείται γύρω από κάθε chunk ώστε να επιτευχθεί μια ισορροπία ως προς τον αριθμό τον αντικειμένων που το κάθε transaction χειρίζεται. Δεν θέλουμε, για λόγους απόδοσης, ούτε να χειριστούμε όλα τα αντικείμενα σε ένα μόνο transaction, ούτε καθένα απο αυτά (που μπορεί να είναι χιλιάδες) στο δικό του transaction.

Τέλος, για την διαχείριση σε επίπεδο listener η απάντηση εξαρτάται απο το interface που υλοποιούμε. Ο **ChunkListener** για παράδειγμα εκτελεί την **beforeChunk** μέθοδό του εντός του transaction του chunk, αλλά την **afterChunk** χωρίς καθόλου transaction. Άρα αν ο κώδικάς μας πρέπει και για την δεύτερη μέθοδο να είναι transactional αυτό είναι κάτι που θα πρέπει να το χειριστούμε μόνοι μας.

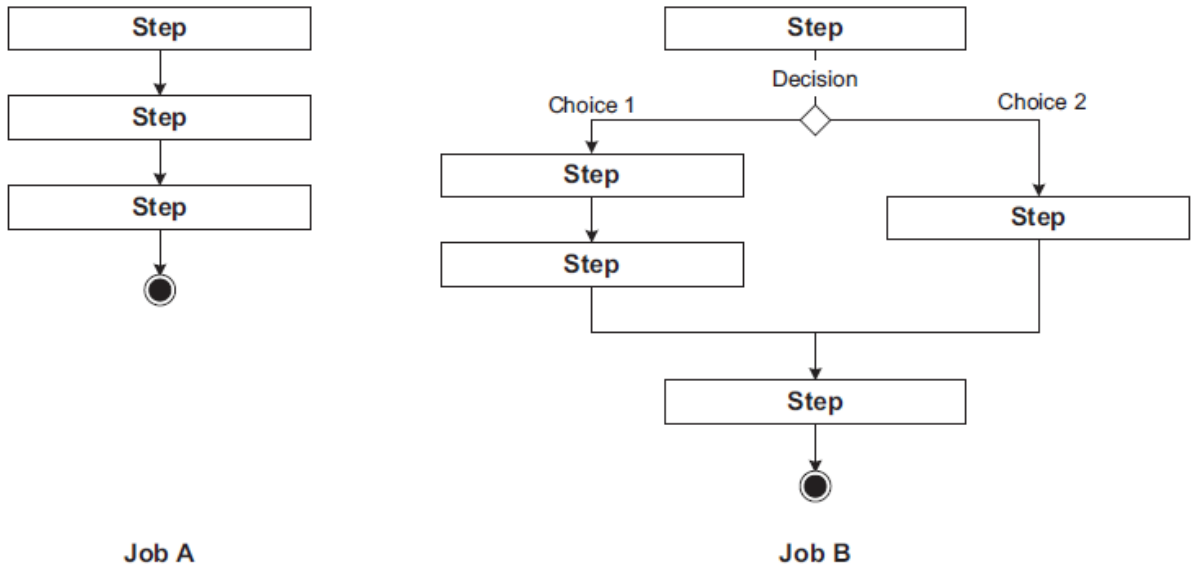
Τα default transactional χαρακτηριστικά ενός tasklet μπορούν, όπως έχει ήδη αναφερθεί, να παρακαμφθούν με τη βοήθεια του **transactional-attributes** element (στην περίπτωση για παράδειγμα που δεν υπάρχει άλλη εφαρμογή που παράλληλα με το job μας διαβάζει τους ίδιους πίνακες στη βάση δεδομένων μπορούμε να επιτύχουμε καλύτερο performance κατεβάζοντας το isolation level σε READ_UNCOMMITED), ενώ παρέχεται και το **no-rollback-exception-classes** element που μπορεί να χρησιμοποιηθεί αν γνωρίζουμε πως κάποιο exception δεν χρειάζεται να οδηγήσει σε rollback (βλ. το configuration του στη Listing 2.41).

```
<job id="authorsJob">
   <step id="authorsStep">
      <tasklet>
         <chunk reader="reader" writer="writer" commit-interval="1000" skip-limit="5">
            <skippable-exception-classes>
               <include class="org.springframework.batch.item.validator.ValidationException" />
            </skippable-exception-classes>
         </chunk>
         <no-rollback-exception-classes>
            <include class="org.springframework.batch.item.validator.ValidationException"/>
         </no-rollback-exception-classes>
      </tasklet>
   </step>
</job>
```

Listing 2.41 Αποφυγή του rollback για κάποιο exception με το no-rollback-exception-classes

2.7 Σύνθετες ροές εκτέλεσης

Τα job στα οποία έχουμε αναφερθεί μέχρι στιγμής είναι όλα γραμμικά· τα βήματα τους εκτελούνται το ένα μετά το άλλο σε σειρά (παράδειγμα το Job A του Σχήματος 2.14), με το **step** element να καθορίζει κάθε φορά το επόμενο βήμα μέσω του **next** attribute. Αν και αυτή είναι η συνηθέστερα συναντώμενη περίπτωση, δεν αποκλείεται οι προδιαγραφές της εφαρμογής μας να απαιτούν και κάποια μη-γραμμική, περισσότερο σύνθετη ροή εκτέλεσης. Το διάγραμμα ενός τέτοιου job δίνεται στο δεύτερο μέρος του Σχήματος 2.14 στο οποίο βλέπουμε το πρώτο βήμα να καθορίζει ποιο από τα δυο παρακλάδια πρέπει να ακολουθηθεί στη συνέχεια. Η απόφαση καθορίζεται από το **ExitStatus** του βήματος και το configuration ορίζει αυτή τη φορά το next ως εμφωλευμένο element στο step element.



Σχήμα 2.13 Γραμμικές και μη-γραμμικές ροές εκτέλεσης

Για να το δούμε αυτό με τη βοήθεια ενός παραδείγματος ας θεωρήσουμε πως για το job της Listing 2.1 υπάρχει μια νέα απαίτηση: αν το readWriteStep αποτύχει θέλουμε να παράγεται ένα report πριν προχωρήσουμε στο τελευταίο βήμα που θα είναι για όλες τις περιπτώσεις ένα cleanupStep (για διαγραφή π.χ. του συμπιεσμένου και του αποσυμπιεσμένου αρχείου ή για μεταφορά τους σε κάποιο άλλο φάκελο). Η απαίτηση αυτή θα μπορούσε να μοντελοποιηθεί με το configuration της Listing 2.42. Τα next elements του readWriteStep βλέπουμε πως καθορίζουν το ότι αν το ExitStatus είναι FAILED το job θα συνεχίζει με το generateReportStep, ενώ αν είναι οτιδήποτε άλλο (χαρακτήρας '*') με το cleanupStep.

```
<job id="importJob">
  <step id="decompressStep" next="readWriteStep">
    <tasklet>(...)</tasklet>
  </step>
  <step id="readWriteStep">
    <tasklet>(...)</tasklet>
```

```
    <next on="FAILED" to="generateReportStep"/>
    <next on="*" to="cleanStep" />
  </step>
  <step id="generateReportStep" next="cleanStep">
    <tasklet>(...)</tasklet>
  </step>
  <step id="cleanStep">
    <tasklet>(...)</tasklet>
  </step>
</job>
```

Listing 2.42 Configuration μιας μη-γραμμικής ροής

Θα είχε ενδιαφέρον, αντί του FAILED status μπορούσαμε να ορίσουμε ένα custom ExitStatus που θα έδινε με το όνομά του και μόνο περισσότερες πληροφορίες για το τι ακριβώς συνέβη κατά την εκτέλεση του βήματος. Η απάντηση είναι πως μπορούμε και μάλιστα με διάφορους τρόπους. Ο πιο όμορφος από αυτούς είναι με τη βοήθεια ενός StepExecutionListener σαν αυτόν της Listing 2.43. Ο listener στο impelementation της **afterStep** μεθόδου εφόσον το ExitStatus δεν είναι FAILED αλλά υπάρχουν skipped items (εύκολα παρέχεται η πληροφορία αυτή αν καλέσουμε την **getSkipCount** στο StepExecution) επιστρέφει ένα custom status με όνομα "COMPLETED WITH SKIPS", διαφορετικά επιστρέφει το πραγματικό (COMPLETED, FAILED, STOPPED ή οτιδήποτε άλλο από αυτά που το framework παρέχει). Με τον τρόπο αυτό, εφόσον προσθέσουμε τον listener στους listeners του βήματος, μπορούμε να αλλάξουμε τον έλεγχο της Listing 2.42 σε:

<next on="COMPLETED WITH SKIPS" to="generateReportStep" />

και η απόφαση να παίρνεται με βάση το δικό μας ExitStatus. Η τιμή του θα καταγράφεται μαζί με τα υπόλοιπα metadata στο job repository (εφόσον έχουμε skipped αντικείμενα), και άρα μόνο με μια ματιά σε αυτά θα μπορούμε να καταλάβουμε τι ακριβώς συνέβη.

```
public class SkippedItemsListener implements StepExecutionListener {

  @Override
  public void beforeStep(StepExecution stepExecution) {
    //Do Nothing
  }

  @Override
  public ExitStatus afterStep(StepExecution stepExecution) {
    if(!ExitStatus.FAILED.equals(stepExecution.getExitStatus()) && stepExecution.getSkipCount() > 0) {
      return new ExitStatus("COMPLETED WITH SKIPS");
    } else {
      return stepExecution.getExitStatus();
    }
  }
}
```

Listing 2.43 Καθορισμός τους ExitStatus με τη βοήθεια ενός StepExecutionListener

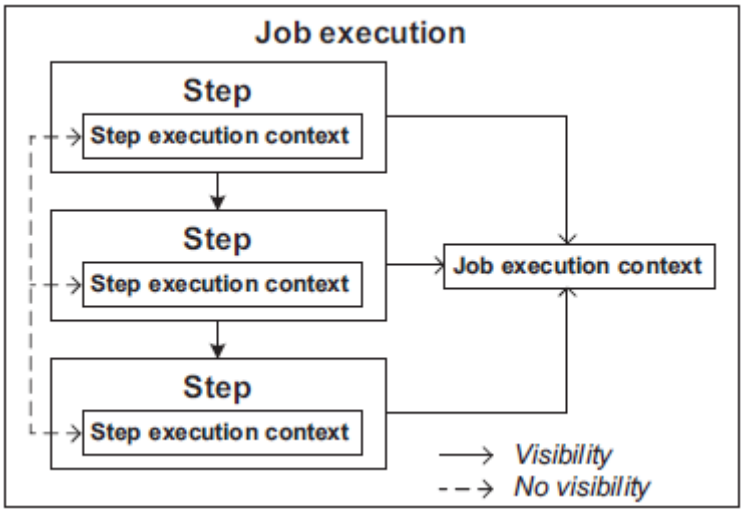
Τέλος, με λογική παρόμοια αυτής των εμφωλευμένων next elements σε ένα step element, ορίζονται και τα **end**, **fail** και **stop** elements. Αυτά, αντί του να καθορίζουν το επόμενο βήμα, με βάση το ExitStatus του στοιχείου που τα χρησιμοποιεί τερματίζουν, αποτυγχάνουν ή σταματούν ένα job. Μπορούμε να σκεφτούμε σαν παράδειγμα ένα έξτρα βήμα πριν το decompressStep που σκοπό έχει να κάνει download το ZIP αρχείο από κάποια εξωτερική πηγή. Το βήμα αυτό ελέγχει με τη βοήθεια ενός listener το αν υπάρχει αρχείο προς download ή όχι και ανάλογα με τα custom ExitStatuses “NO FILE ”, “FILE EXISTS” επιλέγει να τερματίσει το job ή να προχωρήσει στο επόμενο βήμα (βλ. Listing 2.44).

```
<job id="importJob">
  <step id="downloadStep">
    <tasklet ref="downloadTasklet">
      <listeners>
        <listener ref="fileExistsStepListener" />
      </listeners>
    </tasklet>
    <end on="NO FILE" />
    <next on="FILE EXISTS" to="decompressStep"/>
    <fail on="*" />
  </step>
  <step id="decompressStep" next="readWriteStep">
    (...)
  </step>
  (...)
</job>
```

Listing 2.44 Τερματισμός job με βάση το “NO FILE” ExitStatus

2.8 Ανταλλαγή δεδομένων μεταξύ βημάτων

Τα βήματα ενός job πρέπει θεωρητικά να είναι όσο το δυνατόν περισσότερο ανεξάρτητα μεταξύ τους. Το καθένα πρέπει να εκτελεί μια πολύ συγκεκριμένη εργασία και με το που ολοκληρώνεται, η εκτέλεση να ανατίθεται στο επόμενο βήμα. Υπάρχουν περιπτώσεις όμως που αυτό δεν είναι εφικτό και κάποια steps πρέπει να μοιράζονται μεταξύ τους δεδομένα. Κάτι τέτοιο επιτυγχάνεται με χρήση του **ExecutionContext**. Το Spring Batch παρέχει δυο ειδών execution contexts με τον ίδιο τύπο αλλά διαφορετικό scope: το job execution context και το step execution context (βλ. Σχήμα 2.15). Αν το tasklet, ο reader, ο processor, ο writer ή ο listener έχει πρόσβαση στο ExecutionContext μπορεί να το χρησιμοποιήσει για να μοιραστεί με άλλα artifacts τα δεδομένα που θέλει. Υπάρχουν διάφορες προσεγγίσεις για να επιτευχθεί κάτι τέτοιο. Παρουσιάζουμε εδώ δυο απο τις δυο ενδιαφέρουσες.



Σχήμα 2.14 Job και step execution contexts

2.8.1 Χρήση του Job ExecutionContext για ανταλλαγή δεδομένων

Σύμφωνα με την πρώτη προσέγγιση τα βήματα ανταλλάσουν δεδομένα μέσω του job ExecutionContext. Το πρώτο βήμα γράφει την τιμή που θέλει στο execution context και το δεύτερο βήμα την διαβάζει από αυτό. Για να δούμε ένα παράδειγμα μπορούμε να θεωρήσουμε οτι το πρώτο βήμα είναι το tasklet της Listing 2.45 (WritingTasklet), το δεύτερο το tasklet της Listing 2.46 (ReadingTasklet) και πως το μόνο που χρειάζεται να μοιραστούν είναι μια String τιμή (shared). Όπως φαίνεται στον κώδικα, τα tasklets έχουν πρόσβαση στο job ExecutionContext (key/value pair map) μέσω του ChunkContext που είναι το ένα από τα ορίσματα της **execute** μεθόδου. Αυτό που θέλει να γράψει την τιμή στο context καλεί την **putString**, αυτό που θέλει να την διαβάσει καλεί την **getString** και έτσι επιτυγχάνεται η επιθυμητή επικοινωνία.

```
public class WritingTasklet implements Tasklet {
  @Override
  public RepeatStatus execute(StepContribution contribution, ChunkContext chunkContext) throws Exception {
    ExecutionContext jobExecutionContext =
      chunkContext
        .getStepContext()
        .getStepExecution()
        .getJobExecution()
        .getExecutionContext();
    jobExecutionContext.putString("shared", "The value to share");
    return RepeatStatus.FINISHED;
  }
}
```

Listing 2.45 Tasklet για εγγραφή τιμής στο Job execution context

```
public class ReadingTasklet implements Tasklet {
  @Override
  public RepeatStatus execute(StepContribution contribution, ChunkContext chunkContext) throws Exception {
```

```
    ExecutionContext jobExecutionContext =
      chunkContext
        .getStepContext()
        .getStepExecution()
        .getJobExecution()
        .getExecutionContext();

    String shared = jobExecutionContext.getString("shared");

    return RepeatStatus.FINISHED;
  }
}
```

Listing 2.46 Tasklet για ανάγνωση τιμής απο το Job execution context

2.8.2 Εγγραφή στο Job execution context και ανάγνωση με late binding

Η δεύτερη προσέγγιση γράφει στο job ExecutionContext με τον ίδιο ακριβώς τρόπο όπως και η πρώτη (το WritingTasklet παραμένει δηλαδή αυτό της Listing 2.45), αλλά διαβάζει τις τιμές με late binding που καθορίζεται σε XML configuration. Παράδειγμα του ReadingTasklet δίνεται στη Listing 2.47.

```
public class ReadingTasklet implements Tasklet {

  private String shared;

  @Override
  public RepeatStatus execute(StepContribution contribution, ChunkContext chunkContext) throws Exception {
    //Do Something With The shared Value
    return RepeatStatus.FINISHED;
  }

  public void setShared(String shared) {
    this.shared = shared;
  }
}
```

Listing 2.47 Tasklet με την shared τιμή ως property

Το tasklet είναι ευκολότερο να τεσταριστεί καθώς δεν εξαρτάται πλέον από το Spring Batch runtime. Αυτή είναι μια βελτίωση σε σχέση με την υλοποίηση που προτάθηκε με την πρώτη προσέγγιση. Το configuration του σε XML επίπεδο δίνεται στην Listing 2.48. Με χρήση SpEL θέτουμε το shared property ίσο με την τιμή που θα βρεθεί στο αντικείμενο jobExecutionContext. Το ότι το scope τίθεται ίσο με step έχει όπως γνωρίζουμε ως αποτέλεσμα τιμή σε αυτό το property να δίνεται την τελευταία στιγμή, μόλις ξεκινά η εκτέλεση του βήματος (**late binding**).

```
<bean id="readingTasklet" class="gr.booksAdmin.batch.ReadingTasklet" scope="step">
  <property name="shared" value="#{jobExecutionContext['shared']}" />
</bean>
```

Listing 2.48 Configuration του ReadingTasklet

2.9 Monitoring με χρήση των JobExplorer και JobOperator

Το Spring Batch παρέχει δυο interfaces που επιτρέπουν εύκολα την πρόσβαση στα metadata του job repository αλλά και την εκτέλεση ενεργειών όπως το σταμάτημα ενός job που εκτελείται ή το restart ενός που απέτυχε. Τα interfaces αυτά είναι τα **JobExplorer** και **JobOperator** και με χρήση τους μπορούμε να παρακολουθούμε τα όσα συμβαίνουν στο background κατά την εκτέλεση των Spring Batch jobs. Σε επόμενο κεφάλαιο θα παρουσιαστεί με λεπτομέρειες η υλοποίηση ενός πλήρους user interface που σκοπό είχε αυτό ακριβώς το monitoring και βασίστηκε στο Spring MVC και τα JobExplorer, JobOperator interfaces. Εδώ αξίζει να παρουσιάσουμε εν συντομία τις δυνατότητες που αυτά παρέχουν.

Ξεκινάμε με την Listing 2.49 στην οποία παρουσιάζεται το configuration των δυο interfaces. Ο JobExplorer εκτελεί μόνο ενέργειες ανάγνωσης του job repository και για αυτό το μόνο property που χρειάζεται είναι το **dataSource** στο οποίο βρίσκονται οι πίνακες του Spring Batch. Αντίθετα ο JobOperator μπορεί να εκτελέσει και ενέργειες όπως τα stop ή restart που προαναφέρθηκαν και για αυτό χρειάζεται πρόσβαση και σε άλλα αντικείμενα όπως το JobRegistry ή τον JobLauncher (βλ. τα αντίστοιχα properties).

```
<!-- Job Explorer Configuration -->
<bean id="jobExplorer" class="org.springframework.batch.core.explore.support.JobExplorerFactoryBean">
   <property name="dataSource" ref="dataSource"/>
</bean>
<!-- Job Operator Configuration -->
<bean id="jobOperator" class="org.springframework.batch.core.launch.support.SimpleJobOperator">
   <property name="jobRepository" ref="jobRepository"/>
   <property name="jobExplorer" ref="jobExplorer"/>
   <property name="jobRegistry" ref="jobRegistry"/>
   <property name="jobLauncher" ref="jobLauncher"/>
</bean>
```

Listing 2.49 Configuration των JobExplorer και JobOperator

Έχοντας πλέον διαθέσιμα τα δυο αντικείμενα μπορούμε να καλέσουμε τις μεθόδους που το καθένα παρέχει για να εκτελέσουμε τις επιθυμητές ενέργειες. Με τον **JobExplorer** για παράδειγμα μπορούμε να έχουμε πρόσβαση στα ονόματα όλων των job της εφαρμογής μας (μέθοδος **getJobNames**) ή στα executions που εκτελούνται δοθέντος του ονόματος ενός job (μέθοδος **findRunningJobExecutions**) (βλ. Listing 2.50). Διαθέσιμες υπάρχουν και άλλες μέθοδοι όπως η **getJobInstances** δοθέντος του ονόματος ενός job, **getJobInstance** δοθέντος του instance ID ενός job, **getJobExecutions**, **getJobExecution**, **getStepExecution** κ.α.

```
List<JobExecution> runningJobInstances = new ArrayList<JobExecution>();
List<String> jobNames = jobExplorer.getJobNames();
for(String jobName : jobNames) {
   Set<JobExecution> jobExecutions = jobExplorer.findRunningJobExecutions(jobName);
   runningJobInstances.addAll(jobExecutions);
}
```

Listing 2.50 Παράδειγμα χρήσης του JobExplorer

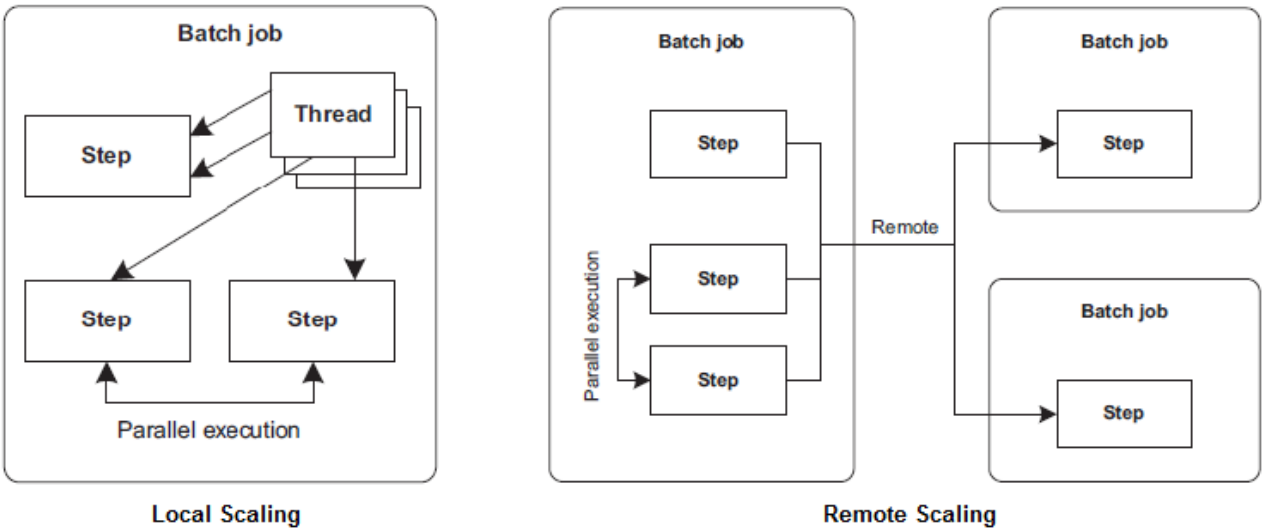
Ο **JobOperator** παρέχει πολλές μεθόδους παρόμοιας λογικής με τον JobOperator χρησιμοποιώντας όμως απλούστερους επιστρεφόμενους τύπους. Η **getRunningExecutions** μέθοδός του επιστρέφει, για παράδειγμα, ένα σετ από Long (instance IDs) όταν η **findRunningExecutions** του JobExplorer, όπως είδαμε, επιστρέφει ένα σετ απο JobExecution αντικείμενα. Μέθοδοι που αντίστοιχες τους δεν παρέχονται απο τον JobExplorer είναι οι **getParameters** για να διαβάσουμε τις παραμέτρους ενός execution, **getSummary** και **getStepExecutionSummaries** για να μας επιστραφούν πληροφορίες σε επίπεδο execution ή βήματος δοθέντος ενός execution ID, καθώς και οι **start**, **startNewInstence**, **stop** και **restart** με τις οποίες μπορούμε να εκτελέσουμε ενέργειες εκκίνησης, επανεκκίνησης κτλ. Δεν χρειάζεται να παρουσιαστεί καμία απο αυτές εδώ καθώς, όπως προαναφέρθηκε, θα δούμε παραδείγματα χρήσης τους στα επόμενα.

2.10 Scaling και Partitioning

Το steaming των δεδομένων που επιτυγχάνεται με τα chunk-oriented βήματα συνεπάγεται συνήθως την εκτέλεση των Spring Batch jobs γρήγορα και αποδοτικά από άποψη χρήσης πόρων. Δεν αποκλείεται όμως, αν για παράδειγμα ο όγκος των δεδομένων είναι τεράστιος, η απόδοση να μην είναι καλή. Σε μια τέτοια περίπτωση μπορούμε να εφαρμόσουμε κάποια από τις διαθέσιμες επιλογές για scaling και partitioning. Μπορούμε να επεξεργαστούμε ένα βήμα με πολλά threads, να τρέξουμε βήματα του job παράλληλα, να χωρίσουμε το σύνολο των δεδομένων σε υποσύνολα που θα επεξεργαστούν παράλληλα τοπικά (πολλά threads) ή απομακρυσμένα (πολλοί κόμβοι) ή και να συνδυάσουμε κάποιες από αυτές τις επιλογές (βλ. παραδείγματα στο Σχήμα 2.16).

Πριν παρουσιάσουμε κάποιες από αυτές τις επιλογές[22], και παρά το ότι το scaling ήταν μια από τις πρώτες δυνατότητες του Spring Batch framework στην οποία αναφερθήκαμε ξεκινώντας την παρουσίασή του σε αυτό το κεφάλαιο (καθώς είναι πράγματι ένα πολύ δυνατό και ενδιαφέρον χαρακτηριστικό) θα πρέπει να ξεκαθαριστεί πως η χρήση του δεν είναι απαραίτητη ή συνιστώμενη για όλες τις περιπτώσεις. Τα jobs μας μπορεί να εκτελούνται αποδοτικά και χωρίς να εφαρμόσουμε τεχνικές scaling. Μόνο αν δούμε πως υπάρχει όντως πρόβλημα ή πως το διαθέσιμο χρονικό παράθυρο για ολοκλήρωση ενός job (που π.χ. πρέπει να τρέχει βραδινές ώρες γιατί τότε το traffic μιας εμπλεκόμενης πηγής δεδομένων είναι χαμηλό) δεν επαρκεί θα πρέπει να αναζητήσουμε τη λύση σε κάποια από αυτές.

[22] Δεν θα σταθούμε καθόλου σε remote τεχνικές παραλληλισμού με χρήση περισσότερων του ενός κόμβων (μηχανημάτων). Το Spring Batch παρέχει hooks για επέκταση της λειτουργικότητάς του προς αυτή την κατεύθυνση αλλά όχι κάποια υλοποίηση και άρα η παρουσίαση μιας τέτοιας λύσης ξεφεύγει απο τα πλαίσια αυτής της εργασίας.



Σχήμα 2.15 Απεικόνιση παραδειγματων τοπικού και απομακρυσμένου scaling

2.10.1 Επεξεργασία βήματος από πολλά threads

Η default συμπεριφορά του Spring Batch είναι να εκτελεί ένα job χρησιμοποιώντας ένα μόνο thread που σειριακά καλεί όλα τα βήματά του. Εύκολα όμως μπορούμε να τρέξουμε ένα βήμα με χρήση περισσότερων του ενός threads. Το μόνο που χρειάζεται είναι να ορίσουμε έναν task-executor στο tasklet element του βήματος που μας ενδιαφέρει όπως γίνεται, για παράδειγμα, στην Listing 2.51.

```
<batch:job id="importJob">
  <batch:step id="importStep">
    <batch:tasklet task-executor="taskExecutor">
      <batch:chunk reader="reader" writer="writer" commit-interval="10"/>
    </batch:tasklet>
  </batch:step>
</batch:job>
<bean id="taskExecutor" class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
  <property name="corePoolSize" value="5"/>
  <property name="maxPoolSize" value="5"/>
</bean>
```

Listing 2.51 Configuration ενός multithreaded βήματος

Για να καταλάβουμε τι συμβαίνει σε αυτή την περίπτωση, αν ενεργοποιήσουμε trace logs τόσο στον reader όσο και στον writer, το import ενός batch θα δώσει αποτελέσματα σαν αυτά της Listing 2.52. Διαφορετικά threads (#3, #5 κτλ) διαβάζουν και γράφουν τα δεδομένα μας με τη σειρά εγγραφής και ανάγνωσης να είναι τυχαία. Δεν υπάρχει τρόπος να εξασφαλιστεί δηλαδή πως τα αντικείμενα θα επεξεργαστούν με μια προκαθορισμένη σειρά. Αυτός είναι ένας περιορισμός της συγκεκριμένης τεχνικής που πρέπει να έχουμε υπόψη μας εφόσον θέλουμε να την χρησιμοποιήσουμε.

```
(...)
thread #5 – Read Item with ID #59
thread #3 – Read Item with ID #60
thread #5 – Read Item with ID #61
thread #3 – Read Item with ID #62
thread #5 – Read Item with ID #63
thread #3 – Read Item with ID #64
thread #5 – Read Item with ID #65
thread #3 – Read Item with ID #66
thread #5 – Read Item with ID #67
thread #3 – Read Item with ID #68
thread #3 – Read Item with ID #69
thread #5 – Write Items with IDs #51, #52, #53, #55, #57, #59, #61, #63, #65, #67
thread #3 – Read Item with ID #70
thread #3 – Write Items with IDs #54, #56, #58, #60, #62, #64, #66, #68, #69, #70
(...)
```

Listing 2.52 Logging κατά την εκτέλεση ενός multithreaded βήματος

Δεν είναι όμως ο μόνος. Οι περισσότεροι readers και writers του Spring Batch θέλοντας να επιτύχουν χαρακτηριστικά όπως το restart διατηρούν state και άρα δεν είναι thread-safe[23]. Επομένως δεν μπορούν να χρησιμοποιηθούν σε multi-threaded σενάρια όπως αυτό που μόλις παρουσιάστηκε.

Ένας τρόπος για να ξεπεράσουμε αυτό το πρόβλημα είναι η υλοποίηση ενός thread-safe reader σαν αυτόν της Listing 2.53. Μαρκάρουμε την read μέθοδο με το **synchronized** keyword και αναθέτουμε την επεξεργασία στον delegate item reader. Επειδή αυτός πιθανότατα υλοποιεί το ItemStream interface για να διατηρεί το state, πρέπει και εμείς να το υλοποιήσουμε κάνοντας override της μεθόδους του (**open**, **update**, **close**). Η ανάγνωση συνήθως είναι γρηγορότερη από την εγγραφή κάτι που καθιστά μια προσέγγιση σαν και αυτή εφικτή. Ο reader διαβάζει (γρήγορα) δεδομένα και τα στέλνει στις επόμενες φάσεις –επεξεργασία και εγγραφή– που εξ ορισμού είναι περισσότερο απαιτητικές από άποψη χρόνου. Τα thread λοιπόν δεν θα συγκρούονται για να διαβάσουν δεδομένα, αφού θα είναι απασχολημένα με την εγγραφή τους, και άρα το επιθυμητό αποτέλεσμα θα έχει επιτευχθεί.

```
public SynchronizingItemReader implements ItemReader<Author>, ItemStream {
  private ItemReader<Author> delegate;

  public synchronized Author read() throws Exception {
    return delegate.read();
  }

  public void close() throws ItemStreamException {
    if(this.delegate instanceof ItemStream) {
      ((ItemStream)this.delegate).close();
    }
  }
```

[23] Μια κλάση είναι thread-safe όταν μπορεί να χρησιμοποιηθεί σε multi-threaded περιβάλλοντα δίνοντας σωστά αποτελέσματα. Προβλήματα προκύπτουν απο τη χρήση των static και instance variables της κλάσης ή των μεθόδων της που τα καλούν. Για υποστήριξη του thread-safety η Java παρέχει το synchronized keyword, την ThreadLocal κλάση και το .util.concurrent πακέτο.

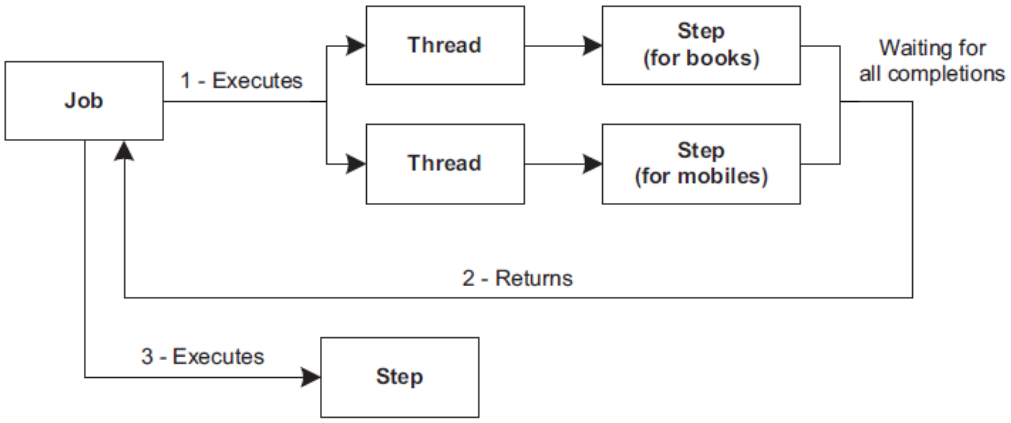
```
  public void open(ExecutionContext context) throws ItemStreamException {
    if(this.delegate instanceof ItemStream) {
      ((ItemStream)this.delegate).open(context);
    }
  }

  public void update(ExecutionContext context) throws ItemStreamException {
    if(this.delegate instanceof ItemStream) {
      ((ItemStream)this.delegate).update(context);
    }
  }
  //The Rest Of The Implementation Skipped For Brevity
}
```

Listing 2.53 Παράδειγμα υλοποίησης thread-safe ItemReader

Ένας άλλος τρόπος (γνωστός και ως indicator pattern), εφόσον αναφερόμαστε σε ανάγνωση απο βάση δεδομένων (π.χ. με τον JdbcCursorItemReader) και έχουμε πρόσβαση σε αυτή ώστε να προσθέσουμε μια στήλη (έστω processed) που θα παίζει το ρόλο του flag για το αν μια εγγραφή διαβάστηκε ή όχι, θα ήταν να απενεργοποιήσουμε την διαχείριση του state στον reader (property **saveState="false"**), να διαβάζουμε εγγραφές με βάση αυτό το flag (π.χ. "WHERE processed=false") και ο writer μετά την εγγραφή να ενημερώνει και το flag ως processed. Έτσι μπορούμε αφενός να εκτελέσουμε το βήμα με χρήση πολλών threads και αφετέρου να μην ανησυχούμε για το τι θα συμβεί αν αυτό αποτύχει και χρειαστεί restart. Το flag είναι αυτό που ορίζει το τι απομένει να διαβαστεί και άρα μπορούμε να επανεκκινήσουμε το job χωρίς κανένα πρόβλημα.

2.10.2 Παραλληλισμός επεξεργασίας

Το Spring Batch παρέχει τη δυνατότητα να οργανώσουμε την εκτέλεση των βημάτων ενός job ώστε κάποια να εκτελούνται παράλληλα με κάποια άλλα. Θα μπορούσαμε να σκεφτούμε ένα job που διαβάζει δυο αρχεία (π.χ. ένα για αρχείο για βιβλία και ένα αρχείο για κινητά τηλέφωνα) (βλ. Σχήμα 2.17), τα γράφει σε μια βάση δεδομένων και πριν τελειώσει εκτελεί ένα βήμα για μεταφορά των επεξεργασμένων αρχείων σε κάποιο φάκελο ώστε να διατηρούνται σαν ιστορικό. Τα βήματα για τα αρχεία μπορούν να εκτελεστούν σειριακά το ένα μετά το άλλο (το default), πολύ εύκολα όμως, και με το μόνο που χρειάζεται να είναι XML configuration, θα μπορούσαν να εκτελεστούν και παράλληλα (βλ. Listing 2.54). Προφανώς μια τέτοια προσέγγιση δεν έχει τα προβλήματα που παρουσιάστηκαν στην προηγούμενη ενότητα (όπου με περισσότερα του ενός thead εκτελούσαμε ένα μόνο βήμα) και άρα, εφόσον το job μας αποτελείται από βήματα που μπορούν να παραλληλιστούν, είναι σαφώς προτιμότερη.



Σχήμα 2.16 Εκτέλεση βημάτων παράλληλα με ξεχωριστό thread για κάθε βήμα

Με το **split** element ορίζουμε τα **flows** που επιθυμούμε να τρέξουν παράλληλα. Αυτά τα flows μπορεί να είναι ένα μόνο βήμα ή μια σειρά βημάτων με καθορισμένη σειρά. Το split element είναι και αυτό ένα βήμα και άρα έχει ένα **id** attribute ώστε να μπορεί κάποιο άλλο βήμα να αναφέρεται σε αυτό (το decompressStep στο παράδειγμα μας), ενώ μπορεί να ορίζει και το βήμα που πρέπει να εκτελεστεί με το που ολοκληρωθούν όλα τα flows που περιέχει (**next** attribute). Ο **task-executor** που είναι υπεύθυνος για τη δημιουργία των threads που θα αναλάβουν την επεξεργασία του κάθε flow είναι προαιρετικός. Στην περίπτωση που δεν τον ορίσουμε ρητά όπως εδώ, το framework θα χρησιμοποιήσει τον default **SyncTaskExecutor**.

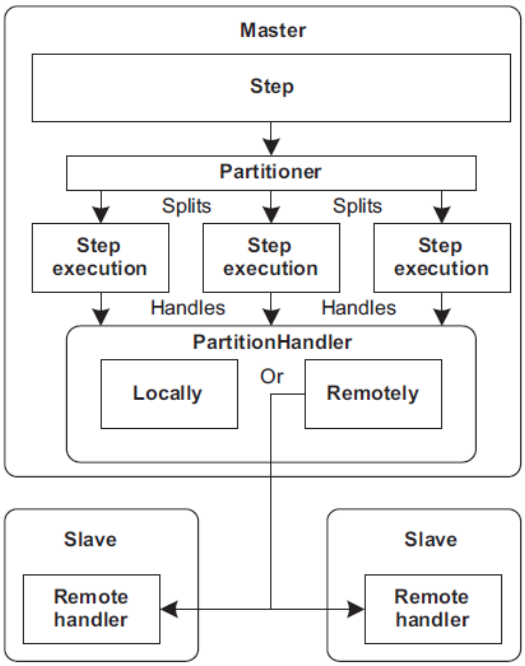
```
<batch:job id="importJob">
  <batch:step id="decompressStep" next="readWriteStep">
    <batch:tasklet ref="decompressTasklet"/>
  </batch:step>
  <batch:split id="readWriteStep" task-executor="taskExecutor" next="moveProcessedFilesStep">
    <batch:flow>
      <batch:step id="importBookProducts"/>
    </batch:flow>
    <batch:flow>
      <batch:step id="importMobileProducts"/>
    </batch:flow>
  </batch:split>
  <batch:step id="moveProcessedFilesStep">
    <batch:tasklet ref="moveProcessedFilesTasklet" />
  </batch:step>
</batch:job>
```

Listing 2.54 Configuration των παράλληλων βημάτων

2.10.3 Partitioning

Η τελευταία –και πολύ δημοφιλής– τεχνική για scaling που παρουσιάζεται εδώ είναι το partitioning. Το partitioning λαμβάνει χώρα σε επίπεδο βήματος, το configuration του είναι σχετικά απλό και υποστηρίζει out of the box τη δυνατότητα για restart αποτυχημένων jobs. Το πως θα χωριστούν τα δεδομένα του

βήματος σε partitions εξαρτάται από τη φύση τους. Μπορούμε για παράδειγμα να χωρίσουμε τον πίνακα μιας βάσης δεδομένων σε partitions με βάση το primary key του ή τα προς επεξεργασία αρχεία με βάση το πρώτο γράμμα του ονόματός τους. Αυτό είναι πιθανό να χρειαστεί custom υλοποίηση με τη βοήθεια των όσων το **Partitioner** interface προσφέρει. Τέλος, το πως θα διαχειριστούν τα παραγόμενα step executions είναι και αυτό θέμα σχεδίασης. Μπορούν να διαχειριστούν τοπικά απο περισσότερα του ενός threads ή και απομακρυσμένα με χρήση τεχνολογιών όπως το Spring Integration [24](και οι δυο δυνατότητες μπορούν να υποστηριχθούν με τη βοήθεια του **PartitionHandler** interface).



Σχήμα 2.17 Partitioning βήματος με τα partitions να επεξεργάζονται τοπικά ή απομακρυσμένα

Σαν παράδειγμα μπορούμε να δούμε το configuration ενός partitioned job στην Listing 2.55. Σε αυτό, αντί του tasklet element εντός του βήματος χρησιμοποιούμε το **partition** element το οποίο αναφέρεται στο προς εκτέλεση βήμα με το **step** element. Ορίζονται επίσης ο **partitioner** (δεν δίνεται καθόλου η υλοποίησή του) και ο **handler** που εφόσον δεν καθοριστεί ρητώς να είναι κάποια custom υλοποίηση (όπως δεν έχει καθοριστεί εδώ) έχει τον default τύπο που είναι ο **TaskExecutorPartitionHandler**. Ο handler είναι αυτός που κάνει όλη τη δουλειά. Με τη βοήθεια ενός **StepExecutionSplitter** (συνήθως χρησιμοποιείται ο **SimpleStepExecutionSplitter**) αναθέτει τη δημιουργία των partitions (step executions) σε έναν Partitioner. Μόλις αυτά δημιουργηθούν, ο handler αναλαμβάνει με βάση μια προκαθορισμένη στρατηγική (είτε τοπικά σε multithreaded περιβάλλον, είτε remotely σε περισσότερους του ενός κόμβους) να εκτελέσει καθένα από αυτά τα step executions. Τελικώς λοιπόν εμείς, ως Spring Batch developers, το μόνο που έχουμε να κάνουμε

[24] Περισσότερα για το Spring Integration εδώ: https://projects.spring.io/spring-integration/

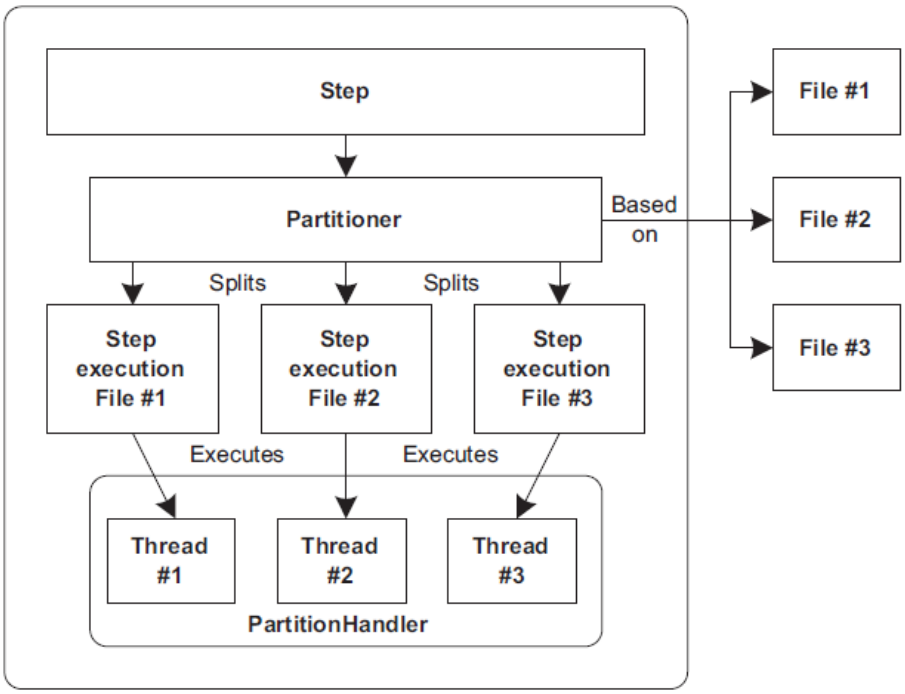
–πέρα από ένα configuration σαν αυτό της Listing 2.55– είναι να υλοποιήσουμε τον Partitioner που θα χωρίσει τα δεδομένα μας σε κομμάτια (εφόσον δεν μας καλύπτει κάποια έτοιμη υλοποίηση).

```
<batch:job id="importProducts">
  <batch:step id="importStep">
    <batch:partition step="partitionedImportStep" partitioner="partitioner">
      <batch:handler grid-size="5" task-executor="taskExecutor"/>
    </batch:partition>
  </batch:step>
</batch:job>
<batch:step id="partitionedImportStep">
  <batch:tasklet>
    <batch:chunk reader="reader" writer="writer" commit-interval="1000"/>
  </batch:tasklet>
</batch:step>
<bean id="partitioner" class="(...)" />
```

Listing 2.55 Configuration ενός partitioned job

2.10.3.1 Χρήση του default PartionHandler

Κλείνοντας αυτό το κεφάλαιο, και για να γίνουν τα παραπάνω καλύτερα κατανοητά, δίνουμε ένα πλήρες παράδειγμα χρήσης του default partition handler (**PartitionHandler** interface) που διαβάζει πολλά flat αρχεία ταυτόχρονα[25]. Παράδειγμα custom υλοποίησης για partitioning δεδομένων που διαβάζονται από βάση αφήνεται να παρουσιαστεί λεπτομερώς σε επόμενο κεφάλαιο.



Σχήμα 2.18 Partitioning με τον MultiResourcePartitioner

[25] Υπενθυμίζεται πως ο MultiResourceItemReader που παρουσιάστηκε στην Ενότητα S.2.1 ναι μεν διαβάζει περισσότερα του ενός resources αλλά, by default, σειριακά.

Το Spring Batch παρέχει την κλάση **MultiResourcePartitioner** που δημιουργεί ένα step execution για καθένα από τα προς επεξεργασία αρχεία (3 στο παράδειγμα του Σχήματος 2.19). Το απαιτούμενο configuration είναι αυτό της Listing 2.56. Κατά τα γνωστά, το **partition** element καθορίζει το βήμα που θα χωριστεί σε partitions (παραλείπεται εδώ) και τον partitioner, ενώ ο **handler** (TaskExecutorPartitionHandler) ορίζει τα task-executor και grid-size. Ο partitioner αναφέρεται στην κλάση MultiResourcePartitioner, καθορίζει το pattern για τα αρχεία που θα διαβαστούν (property **resources**) καθώς και το ότι το όνομα του κάθε αρχείου θα είναι αυτό που θα χρησιμοποιηθεί ως κλειδί όταν ο partitioner δημιουργήσει το step execution αυτού του αρχείου (property **keyName**).

```
<batch:job id="importJob">
  <batch:step id="importStep">
    <batch:partition step="partitionedImportStep" partitioner="partitioner">
      <batch:handler grid-size="3" task-executor="taskExecutor"/>
    </batch:partition>
  </batch:step>
</batch:job>
<bean id="partitioner" class="org.springframework.batch.core.partition.support.MultiResourcePartitioner">
  <property name="keyName" value="fileName"/>
  <property name="resources" value="file:./resources/partition/input/*.txt"/>
</bean>
(...)
<bean id="taskExecutor" class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
  <property name="corePoolSize" value="5"/>
  <property name="maxPoolSize" value="5"/>
</bean>
```

Listing 2.56 Configuration για partitioning με τον MultiResourcePartitioner

Το μόνο που μένει να γίνει για να ολοκληρωθεί το παράδειγμά μας, είναι με late binding (**scope="step"**) σε επίπεδο reader να καθοριστεί το **resource** που θα διαβάζεται και το οποίο είναι το fileName που σαν κλειδί μόλις χρησιμοποιήσαμε στον partitioner (βλ. Listing 2.57).

```
<bean id="itemReader" scope="step" class="org.springframework.batch.item.file.FlatFileItemReader">
  <property name="resource" value="#{stepExecutionContext[fileName]}"/>
  (...)
</bean>
```

Listing 2.57 Property resource για τον FlatFileItemReader

ΚΕΦΑΛΑΙΟ 3
Ανάλυση Προβλήματος – High Level Requirements

3.1 Ανάλυση Προβλήματος και Απαιτήσεων

Όπως ήδη περιγράψαμε στο Κεφάλαιο 2, η εφαρμογή που υλοποιείται στα πλαίσια της παρούσας διπλωματικής προκύπτει από την ανάγκη μεταφοράς ολόκληρης της παραγωγικής υποδομής αποθήκευσης δεδομένων ενός μεγάλου ηλεκτρονικού βιβλιοπωλείου. Το ηλεκτρονικό βιβλιοπωλείο επανακατασκευάστηκε και η εταιρία που ανέλαβε την υλοποίηση της νέας εφαρμογής είχε να αντιμετωπίσει δυο προκλήσεις: το πώς θα γίνει το αρχικό import των δεδομένων στις νέες υποδομές, και το πώς σε καθημερινή βάση θα ενημερώνεται η νέα εφαρμογή με αλλαγές που για ένα μεταβατικό διάστημα θα γίνονται στην παλαιά βάση δεδομένων (εφεξής η λειτουργία αυτή θα αναφέρεται και ως δέλτα). Το νέο ηλεκτρονικό κατάστημα υλοποιήθηκε με χρήση του ATG Web Commerce platform της Oracle[26] που λειτουργεί κάνοντας χρήση 4 βάσεων δεδομένων: της PUBLISHING βάσης που συντηρεί versions των δεδομένων, των SWITCHING_A και SWITCHING_B που φέρουν μόνο την πιο πρόσφατη έκδοση κάθε εγγραφής και πρέπει να είναι πανομοιότυπες καθώς ανάλογα με τα λειτουργίες που εκτελεί η πλατφόρμα χρησιμοποιείται πότε η μια και πότε η άλλη, και της PRODUCTION βάσης στην οποία αποθηκεύεται οποιαδήποτε πληροφορία δεν μας ενδιαφέρει να έχει versioning (κλασικότερο παράδειγμα οι παραγγελίες των πελατών ή τα αποθέματα). Οι βάσεις αυτές, αν εξαιρέσουμε το αρχικό import για το οποίο θα χρησιμοποιηθεί το Spring Batch, δεν θα ενημερώνονται από την εφαρμογή που παρουσιάζεται σε αυτή την εργασία. Το ATG έχει έτοιμους μηχανισμούς για εισαγωγή πληροφορίας από μια βάση δεδομένων εφόσον αυτή πληροί κάποιες προϋποθέσεις. Με αυτή την «ενδιάμεση» βάση, που εφεξής θα αναφέρεται και ως Consolidation, θα ασχοληθούμε κατά κύριο λόγο εδώ.

3.1.1 Αρχικό Import

Η παραγωγική βάση δεδομένων της εταιρίας που ήταν υπεύθυνη για το παλιό ηλεκτρονικό κατάστημα, περιείχε πάνω από 7.000.000 προϊόντα με όλες τις σχετιζόμενες οντότητες που ένα προϊόν χρειάζεται για να μπορεί να εμφανίζεται ολοκληρωμένα σε ένα κατάστημα (συγγραφείς, κατηγορίες, προμηθευτές, τιμές, αποθέματα κ.α.). Τα περισσότερα από τα δεδομένα εξάγονται από σχεσιακούς πίνακες της προηγούμενης βάσης, υπάρχουν όμως και περιπτώσεις όπως για παράδειγμα αυτή των αποθεμάτων, όπου τα δεδομένα

[26] Περισσότερα για το ATG εδώ: http://www.oracle.com/us/products/applications/atg/web-commerce/index.html

εξάγονται από το σύστημα ενδοεπιχειρησιακού σχεδιασμού της εταιρίας (ERP) σε αρχείο το οποίο εν συνεχεία διαβάζεται, επεξεργάζεται και εισάγεται στη βάση δεδομένων.

Η απαίτηση για αρχικό import των δεδομένων τόσο στην Consolidation βάση όσο και στις βάσεις του ATG μέσα στο περιορισμένο χρονικό παράθυρο μερικών ωρών (δυο βράδια) κατέστησε επιτακτική την χρήση ενός framework σαν το Spring Batch ώστε να μπορεί να γίνει αξιόπιστα και ταχύτερα η διαδικασία.

Η απαίτηση για αξιόπιστη εκτέλεση της όλης διαδικασίας, ορίζει την ανάγκη για ενημέρωση των χρηστών με το exit status (επιτυχές ή αποτυχημένο) των εργασιών, τόσο σε επίπεδο Step όσο και σε επίπεδο Job. Επειδή, μάλιστα η αποτυχία ενός ολόκληρου Job, είναι ιδιαίτερα σοβαρή και επηρεάζει την ολοκλήρωση του import, κρίνεται σκόπιμο να ενημερώνονται οι διαχειριστές άμεσα, μέσω αποστολής email, ώστε να μπορούν να ελέγξουν τα σφάλματα που προέκυψαν, να επέμβουν πιθανώς διορθωτικά (αν για παράδειγμα το αρχείο των αποθεμάτων έχει κάποιες εγγραφές με λάθος φορμάτ που πρέπει να διορθωθούν ή να αφαιρεθούν) και να επανεκκινήσουν το Job.

Η αξιόπιστη εκτέλεση είναι κάτι που απαιτείται τόσο για το αρχικό import όσο και για το δέλτα. Το ίδιο προφανώς ισχύει και για την ταχύτητα, με το αρχικό import όμως είναι αναμφίβολα μεγαλύτερη πρόκληση λόγω του όγκου των προς μεταφορά δεδομένων. Η διαδικασία κρίθηκε πως για να ολοκληρωθεί στον απαιτούμενο χρόνο έπρεπε να χωριστεί σε partitions που θα ανατεθούν σε περισσότερα του ενός thread και θα επεξεργαστούν παράλληλα με χρήση partitioners σαν και αυτούς που παρουσιάστηκαν θεωρητικά στην ενότητα 2.10.3. Διαφορετικά το να μεταφερθεί μέσα σε δυο βράδια η πληροφορία 7.000.000 προϊόντων από την αρχική βάση δεδομένων στην Consolidation και από εκεί στις βάσεις του ATG δεν θα ήταν εφικτό. Περισσότερα για αυτή την υλοποίηση και για την βελτίωση από άποψη χρόνου που επέφερε θα αναφερθούν στα όσα ακολουθούν.

3.1.2 Άντληση των καθημερινών αλλαγών (Δέλτα).

Όπως ήδη αναφέρθηκε, η προηγούμενη εταιρία, κατά το διάστημα που διαχειριζόταν το ηλεκτρονικό βιβλιοπωλείο είχε δημιουργήσει και συντηρούσε γέφυρες άντλησης περιεχομένου από αρκετούς παρόχους (βιβλιονet, Gardners, Nielsen κ.α.), τα συμβόλαια με τους οποίους επεκτείνονταν χρονικά και μετά το go-live της νέας εφαρμογής. Αυτό σημαίνει ότι, μετά το αρχικό import των δεδομένων στη νέα υποδομή και για κάποιο σεβαστό χρονικό διάστημα η παλιά βάση θα συνέχιζε να ενημερώνεται με υλικό από αυτούς τους παρόχους. Η διαφορά του instance των δεδομένων που είχε μια συγκεκριμένη μέρα η παλιά βάση με το instance που έχει την επόμενη λόγω αλλαγών που προήλθαν από τους εξωτερικούς παρόχους αναφέρεται ως δέλτα. Η εφαρμογή μας αντλεί καθημερινά τις αλλαγές από την παλιά βάση δεδομένων και ενημερώνει την Consolidation βάση, όχι όμως και τις βάσεις του ATG. Πρέπει να καταστεί σαφές πως από

το go-live και μετά απαγορεύεται οποιαδήποτε ενέργεια στις βάσεις του ATG. Το δέλτα αντλείται από την αρχική βάση, μεταφέρεται στην Consolidation και από εκεί και πέρα το ATG αναλαμβάνει με τους δικούς του μηχανισμούς να ενημερώσει τις βάσεις του.

Με το πέρας αυτού του μεταβατικού διαστήματος, η παλιά βάση δεδομένων αχρηστεύεται και η νέα εταιρία διαχείρισης του ηλεκτρονικού καταστήματος αναλαμβάνει να υλοποιήσει εκ νέου γέφυρες άντλησης περιεχομένου προς τις νέες υποδομές. Η επέκταση όμως της εφαρμογής για την υποστήριξη αυτής της λειτουργίας, αν και μελλοντικός στόχος, είναι έξω από τα πλαίσια της παρούσας εργασίας.

3.2 High Level Requirements

Τα περισσότερα δεδομένα διαβάζονται από και γράφονται σε σχεσιακούς πίνακες. Για την περίπτωση αυτή, του διαβάσματος δηλαδή από βάση και εγγραφής σε βάση παρουσιάζουμε εδώ δύο εργασίες που αφορούν δύο κύριες οντότητες: τα προϊόντα (Products) και τους συγγραφείς (Authors). Στην πραγματικότητα, υλοποιούνται πολλές περισσότερες αντίστοιχες, μια για κάθε οντότητα-πίνακα της παλιάς βάσης δεδομένων, οι οποίες όμως δεν έχουν καμία ουσιαστική διαφορά με τις δυο εργασίες που προαναφέρθηκαν, εξού και παραλείπονται.

Για τον δεύτερο τύπο εργασίας, την περίπτωση διαβάσματος από αρχείο και εγγραφής σε βάση, παρουσιάζουμε την εργασία που αφορά το απόθεμα (inventory), το οποίο διαβάζεται από αρχείο CSV που έχει εξαχθεί από το ERP της εταιρίας στην οποία ανήκει το ηλεκτρονικό βιβλιοπωλείο.

Για κάθε οντότητα (product, author, inventory κ.α.) ο αντίστοιχος πίνακας στη νέα βάση εκτός από τις στήλες που θα συντηρούν την πραγματική πληροφορία θα περιέχει και flags που θα δείχνουν το ποιο Batch job instance ήταν αυτό που τροποποίησε την κάθε εγγραφή (απαραίτητο ώστε σε συνδυασμό με τους πίνακες του Spring Batch να μπορούμε να αντλούμε πληροφορία για το πότε αυτή τροποποιήθηκε ή το πόσες εγγραφές τροποποίησε το κάθε job), καθώς και το αν μια εγγραφή έχει τροποποιηθεί ή / και καταναλωθεί από το ATG (με δεδομένο το ότι καθημερινά το ATG θα αντλεί το δέλτα από την ενδιάμεση βάση δεδομένων το flag αυτό είναι απαραίτητο ώστε να ξέρει ποιες εγγραφές είναι αυτές που έχουν τροποποιηθεί αλλά δεν έχει καταναλώσει).

3.2.1 Αρχικό Import

Για τη λειτουργία του αρχικού import, σε ένα υψηλότερο, αφαιρετικό επίπεδο έχουμε τις εξής απαιτήσεις:

Οντότητα Authors (Συγγραφείς)

- Διάβασμα από τον παλιά βάση (input database) δεσμίδων οντοτήτων. Το μέγεθος της δεσμίδας, δηλαδή ο αριθμός των εγγραφών του αντίστοιχου πίνακα που επιστρέφονται ανά ερώτημα, ορίζεται στις 1000, θα μπορεί όμως να μεταβάλλεται με προγραμματιστικό τρόπο. Η εξαγωγή της πληροφορίας πραγματοποιείται με συνδυαστικά ερωτήματα σε τρείς πίνακες καθώς η παλιά βάση δεδομένων κρατά όσες πληροφορίες πρέπει να μεταφράζονται στα αγγλικά σε δυο ξεχωριστούς πίνακες (ebooks_virtuemart_product_authors είναι ο βασικός πίνακας και ebooks_virtuemart_product_authors_el_gr, ebooks_virtuemart_product_authors_en_gb είναι οι πίνακες στους οποίους συντηρούνται πληροφορίες όπως τα ελληνικά και αγγλικά ονοματεπώνυμα, τα ελληνικά και αγγλικά βιογραφικά κ.α.).
- Επεξεργασία κάθε μέλους της δεσμίδας. Κάθε οντότητα Author χρειάζεται να έχει το ονοματεπώνυμο του συγγραφέα και στα ελληνικά και στα αγγλικά. Εάν κάποιο από τα δύο πεδία λείπει, η εμπορική απαίτηση ορίζει να συμπληρώνεται με το πεδίο που υπάρχει (π.χ. και αγγλικό και ελληνικό ονοματεπώνυμο ίδιο με το ελληνικό που βρέθηκε να υπάρχει στη βάση). Στη περίπτωση που κάποια οντότητα έχει κενά και τα δύο πεδία φιλτράρεται και δεν εγγράφεται στη νέα βάση καθώς είναι άχρηστη χωρίς ονοματεπώνυμο.
- Εγγραφή στην νέα βάση δεδομένων (output database) όσων οντοτήτων της δεσμίδας επεξεργάστηκαν επιτυχώς στο προηγούμενο βήμα. Για κάθε συγγραφέα που εγγράφεται στον πίνακα, εκτός από τα πεδία με τις σχετικές με αυτών πληροφορίες (ονοματεπώνυμο, βιογραφικό κτλ), συμπληρώνονται και τα πεδία που αφορούν το job το οποίο πραγματοποίησε την εγγραφή και τα flags για το ότι αυτή έχει τροποποιηθεί αλλά δεν έχει ακόμα καταναλωθεί από το ATG.

Οντότητα Products (Προϊόντα)

- Διάβασμα από τον παλιά βάση (input database) δεσμίδων οντοτήτων. Η εξαγωγή της πληροφορίας γίνεται και εδώ με συνδυαστικά ερωτήματα και το μέγεθος της δεσμίδας ορίζεται σε 1000 εγγραφές (μπορεί να αλλάζει προγραμματιστικά).
- Επεξεργασία κάθε μέλους της δεσμίδας. Κάθε οντότητα Product χρειάζεται να περιλαμβάνει τον τίτλο του προϊόντος και στα ελληνικά και στα αγγλικά (διαφορετικά είτε θα χρησιμοποιηθεί και εδώ η τιμή που βρέθηκε και στα δυο πεδία είτε θα φιλτραριστεί), και να ανήκει σε ένα από τα αποδεκτά είδη προϊόντων (Kinds, βλέπε τον Πίνακα 3.1). Όσα προϊόντα πληρούν αυτές τις προϋποθέσεις εγκυρότητας, πρέπει στη συνέχεια να εμπλουτιστούν με τιμές στα πεδία: Κατηγορία (**Category**), Συγγραφέας (**Author**), Κατασκευαστής (**Manufacturer**), Θέμα (**Subject**), Ηλικίες στις οποίες το προϊόν απευθύνεται (**Ages**), είδος εξωφύλλου (**Cover**) κ.α. Οι τιμές αυτές εξάγονται, δοθέντος του ID ενός προϊόντος, με ένα συνδυαστικό ερώτημα από τους αντίστοιχους πίνακες οντοτήτων της input database. Το μεγάλο πλήθος των εμπλεκόμενων πινάκων καθιστούσε αναποτελεσματική την προσέγγιση ανάγνωσης όλης της

πληροφορίας με μια κατά τη φάση της ανάγνωσης και για αυτό, ακολουθώντας κατά κάποιο τρόπο την λογική του driving query pattern που περιγράφηκε στην Ενότητα 2.4.1, μετακινήθηκε ο εμπλουτισμός της βασικής πληροφορίας με όλα τα παραπάνω στο στάδιο της επεξεργασίας.

- Εγγραφή στην νέα βάση δεδομένων (output database) όσων οντοτήτων της δεσμίδας επεξεργάστηκαν επιτυχώς στο προηγούμενο βήμα. Για κάθε προϊόν που εγγράφεται στον πίνακα, εκτός από τα πεδία με τις σχετικές με αυτών πληροφορίες (τίτλος, είδος, περιγραφή κτλ), συμπληρώνονται και τα πεδία που αφορούν το job το οποίο πραγματοποίησε την εγγραφή και τα flags για το ότι αυτή έχει τροποποιηθεί αλλά δεν έχει ακόμα καταναλωθεί από το ATG.

Κωδικός Είδους	Είδος Προϊόντος
4	Ελληνικό eBook
15	Ελληνικό βιβλίο
16	Ξενόγλωσσο eBook
18	Ταινία
19	Είδος γραφικής ύλης
20	Παιχνίδι
21	Ξενόγλωσσο βιβλίο

Πίνακας 3.1. Πίνακας αποδεκτών ειδών προϊόντων

Οντότητα Inventory (Αποθέματα)

- Διάβασμα των δεδομένων αποθέματος από αρχείο CSV το οποίο έχει εξαχθεί από το ERP της εταιρίας. Το όνομα, η τοποθεσία του αρχείου καθώς και η τοποθεσία όσων αρχείων έχουν ήδη επεξεργαστεί, αποτελούν παραμέτρους που εξάγονται από configuration πίνακα της νέας εφαρμογής, και μέσω του πίνακα αυτού μπορούν να αλλαχθούν από τους διαχειριστές της. Δοθέντων των παραμέτρων αυτών, χρειάζεται να γίνει έλεγχος ύπαρξης της τοποθεσίας και του αρχείου που οι παράμετροι υποδεικνύουν. Αν το αρχείο δεν υπάρχει, ολόκληρο το job αποτυγχάνει. Διαφορετικά, διαβάζονται τα δεδομένα του αρχείου, κάνοντας την υπόθεση ότι κάθε γραμμή του αρχείου αποτελεί μια ξεχωριστή εγγραφή inventory, υπόθεση που μπορεί να αλλάξει με προγραμματιστικό τρόπο.
- Κάθε εγγραφή του αρχείου αναμένεται να είναι της μορφής: ***SapId*!**productId*!**stockQuantity***. Περιέχει δηλαδή τρία πεδία, τα οποία χωρίζονται μεταξύ τους με τους διαχωριστικούς χαρακτήρες **!*** . Το πρώτο από τα πεδία αφορά το ID του προϊόντος κατά το ERP της εταιρίας (στην υλοποίησή μας αγνοείται), το δεύτερο είναι ο κωδικός του προϊόντος κατά το site και το τρίτο η τρέχουσα τιμή του αποθέματος. Τα προϊόντα διαβάζονται από το αρχείο και πάλι σε δεσμίδες των 1000.

- Επεξεργασία κάθε μέλους της δεσμίδας. Στη φάση του αρχικού import δεν απαιτείται κάποια επεξεργασία των εγγραφών inventory πριν την εισαγωγή τους πέρα από το να είναι το απόθεμα θετικός ακέραιος αριθμός ώστε να μην φιλτραριστεί.
- Εγγραφή κάθε μέλους τις δεσμίδας στη νέα βάση. Όπως και στις προηγούμενες περιπτώσεις, για κάθε εγγραφή αποθέματος, εκτός από τα πεδία productId και stockQuantity, συμπληρώνονται και τα πεδία που αφορούν το job το οποίο πραγματοποίησε την αλλαγή και τα flags για το ότι αυτή έχει τροποποιηθεί αλλά δεν έχει ακόμα καταναλωθεί από το ATG.
- Τέλος, μετά την εξαγωγή όλων των δεδομένων και την εισαγωγή τους στη νέα βάση, το αρχείο μεταφέρεται σε ένα φάκελο επεξεργασμένων αρχείων στο file system ώστε να υπάρχει ιστορικό. Ο φάκελος αυτός καθορίζεται στον configuration πίνακα που προαναφέρθηκε.

3.2.2 Δέλτα

Μετά την ολοκλήρωση του αρχικού import, η νέα βάση θα πρέπει να ενημερώνεται καθημερινά με τις αλλαγές που εισήχθησαν από τους παρόχους περιεχομένου στην παλιά βάση και αφορούν τις συγκεκριμένες οντότητες. Η ακριβής ώρα της έναρξης της διαδικασίας αυτής, θα είναι σταθερή καθημερινά, και οριζόμενη προγραμματιστικά μέσω schedulers.

Τα βήματα που περιγράφηκαν στην προηγούμενη ενότητα, θα ακολουθούνται και εδώ, με τη διαφορά ότι θα διαβάζονται μόνο νέες για την βάση εγγραφές ή όσες εγγραφές υπήρχαν αλλά τροποποιήθηκαν. Το ποιες εγγραφές είναι αυτές που τροποποιήθηκαν, καθορίζεται για κάθε οντότητα σε μια timestamp στήλη της παλιάς βάσης. Για να γνωρίζει η εφαρμογή μας μέχρι ποια ημερομηνία έτρεξε το προηγούμενο job θα υπάρχει βοηθητικός πίνακας που θα κρατά για κάθε Batch job instance και κάθε οντότητα την ημερομηνία από την οποία ξεκίνησε και την ημερομηνία στην οποία σταμάτησε (με βάση αυτή τη στήλη). Αν ένα job ολοκληρωθεί επιτυχώς και για τα προϊόντα (παραδείγματος χάριν) καθορίσει πως διάβασε το χρονικό διάστημα Α έως Β, το επόμενο job θα συνεχίσει από το Β μέχρι το μέγιστο διαθέσιμο timestamp την στιγμή που θα ξεκινήσει να εκτελείται. Αν αντίθετα αποτύχει, το επόμενο job για να διαβάσει τις εγγραφές που το προηγούμενο δεν μπόρεσε, θα ξεκινήσει και αυτό από το Α μέχρι το μέγιστο διαθέσιμο timestamp την στιγμή που θα ξεκινήσει να εκτελείται.

Κάθε οντότητα υφίσταται κατά το στάδιο της επεξεργασίας τα validity checks που περιγράφηκαν παραπάνω. Η μόνη διαφορά έχει να κάνει με το inventory όπου για κάθε εγγραφή που διαβάζουμε από το αρχείο ελέγχουμε εάν υπάρχει μια πανομοιότυπη εγγραφή στη βάση, και αν ναι, τη φιλτράρουμε μιας και δεν έχει νόημα να επανεγγραφεί, ούτε να ενημερωθεί, εφόσον δεν έχει μεταβληθεί το απόθεμα του συγκεκριμένου προϊόντος.

3.2.3 Ενημέρωση των Διαχειριστών Σε Περίπτωση Αποτυχίας

Οι δυο ροές εργασιών που αναμένεται να εκτελεί η εφαρμογή (αρχικό import και δέλτα αλλαγών), θα εξάγουν, επεξεργάζονται και επανεισάγουν δεδομένα σε μεγάλες ποσότητες, χωρισμένα σε δεσμίδες, και, όπως όλες οι Batch εργασίες, θα τρέχουν στο παρασκήνιο, χωρίς παρέμβαση του χρήστη του συστήματος. Είναι κρίσιμο, μιας και μιλάμε για επεξεργασία μεγάλου όγκου δεδομένων που πιθανώς χρειάζεται να ολοκληρωθεί σε περιορισμένο χρονικό παράθυρο, να ενημερώνονται άμεσα οι διαχειριστές σε περίπτωση αποτυχίας ενός job, καθώς αυτό το σενάριο απαιτεί τη λήψη άμεσων ενεργειών για διόρθωση των σφαλμάτων και πιθανώς επανεκκίνηση της διαδικασίας.

Για το σκοπό αυτό, χρειάζεται να υλοποιηθεί αυτόματος μηχανισμός αποστολής email στους χρήστες-δαιαχειριστές της εφαρμογής με τα στοιχεία του job που απέτυχε, αμέσως μόλις το job αποτύχει. Η λίστα των εγγεγραμμένων email, θα μπορεί να ενημερωθεί με προγραμματιστικό τρόπο.

3.2.4 Web Interface

Σημαντικό τμήμα της εφαρμογής θα αποτελεί η υλοποίηση διαδικτυακής γραφικής διεπαφής χρήστη, αποτελούμενης από σελίδες ενημέρωσης των χρηστών σχετικά με την εκτέλεση των jobs. Οι χρήστες θα μπορούν να ενημερώνονται με οπτικοποιημένο τρόπο, μέσω web interface για το πλήθος των εκτελέσεων κάθε job, τις λεπτομέρειες της εκτέλεσής του καθώς και για το exit status του (επιτυχημένο ή αποτυχημένο).

Στην κεντρική σελίδα, θα υπάρχει η λίστα όλων των jobs, όπως αυτά έχουν καταγραφεί στη βάση. Από εκεί ο χρήστης θα μπορεί να μεταβεί στη σελίδα ενός συγκεκριμένου job instance, όπου θα βλέπει τη λίστα με όλες τις εκτελέσεις (Job Executions) του συγκεκριμένου instance, και φυσικά λεπτομέρειες για την κάθε εκτέλεση (το exit status, τους χρόνους εκκίνησης, τερματισμού κ.α.). Θα μπορεί επίσης να βλέπει τη λίστα των παραμέτρων της κάθε εκτέλεσης. Για λόγους καλύτερης εποπτείας, θα παρέχεται ξεχωριστή σελίδα με συγκεντρωτική λίστα των αποτυχημένων jobs.

Επιπλέον, ξεχωριστό view θα παρέχεται και για τα βήματα ενός job, ώστε να δει ο χρήστης αν ολοκληρώθηκαν όλα με επιτυχία, καθώς επίσης και συγκεντρωτικές πληροφορίες για βήμα: πόσες διαβάστηκαν από τη βάση ή το αρχείο, πόσες φιλτραρίστηκαν κατά την επεξεργασία, πόσες εγράφησαν στη νέα βάση καθώς και πόσες έγιναν skip από το framework και σε ποίο στάδιο (read, process and write skip counts). Θα παρουσιάζεται επίσης το πλήθος των εγγραφών που έγιναν rollback λόγω κάποιου exception στην επεξεργασία ή την εγγραφή. Έτσι ο διαχειριστής της εφαρμογής θα έχει πλήρη εικόνα της όλης διαδικασίας.

Μια σημαντικότατη λειτουργία του γραφικού περιβάλλοντος είναι η παροχή στους χρήστες, της δυνατότητας να εκκινούν οι ίδιοι κάποιο job. Πιο συγκεκριμένα, πατώντας ένα link στην αρχική σελίδα, ο χρήστης θα μπορεί να δώσει την εντολή άμεσης εκκίνησης κάποιου job, με παραμέτρους που ο ίδιος θα καθορίζει. Αυτό είναι ιδιαίτερα χρήσιμο σε περιπτώσεις αποτυχημένων job, καθώς σε αυτή την περίπτωση χρειάζεται παρέμβαση από χρήστη για να ληφθεί απόφαση για την τύχη του job. Είναι επίσης χρήσιμο σε περιπτώσεις που για εμπορικούς λόγους αποφασίζεται ότι το import ή το δέλτα των αλλαγών κάποιας οντότητας πρέπει να εκτελεστεί άμεσα ή με προτεραιότητα. Δυνατότητα για εκκίνηση των jobs δεν θα έχουν όλοι οι χρήστες. Θα υλοποιηθεί μηχανισμός security ώστε μόνο authorized χρήστες με κάποιο συγκεκριμένο ρόλο (π.χ. ROLE_ADMIN) να μπορούν να τις εκκινούν. Χρήστες και ρόλοι θα μπορούν να προστεθούν με προγραμματιστικό τρόπο.

Τέλος, μέσω του γραφικού περιβάλλοντος, θα πρέπει ο χρήστης να μπορεί κατά βούληση να ενεργοποιεί ή να απενεργοποιεί την εκτέλεση των χρονοπρογραμματισμένων εργασιών. Αυτό είναι ιδιαίτερα χρήσιμο στην περίπτωση που εκτελούνται εργασίες συντήρησης του συστήματος κατά τη διάρκεια των οποίων δεν θέλουμε να έχουμε δραστηριότητα στη βάση δεδομένων λόγω των Spring Batch jobs.

ΚΕΦΑΛΑΙΟ 4
Σχεδίαση Της Εφαρμογής

4.1 Αρχιτεκτονική Σχεδίαση

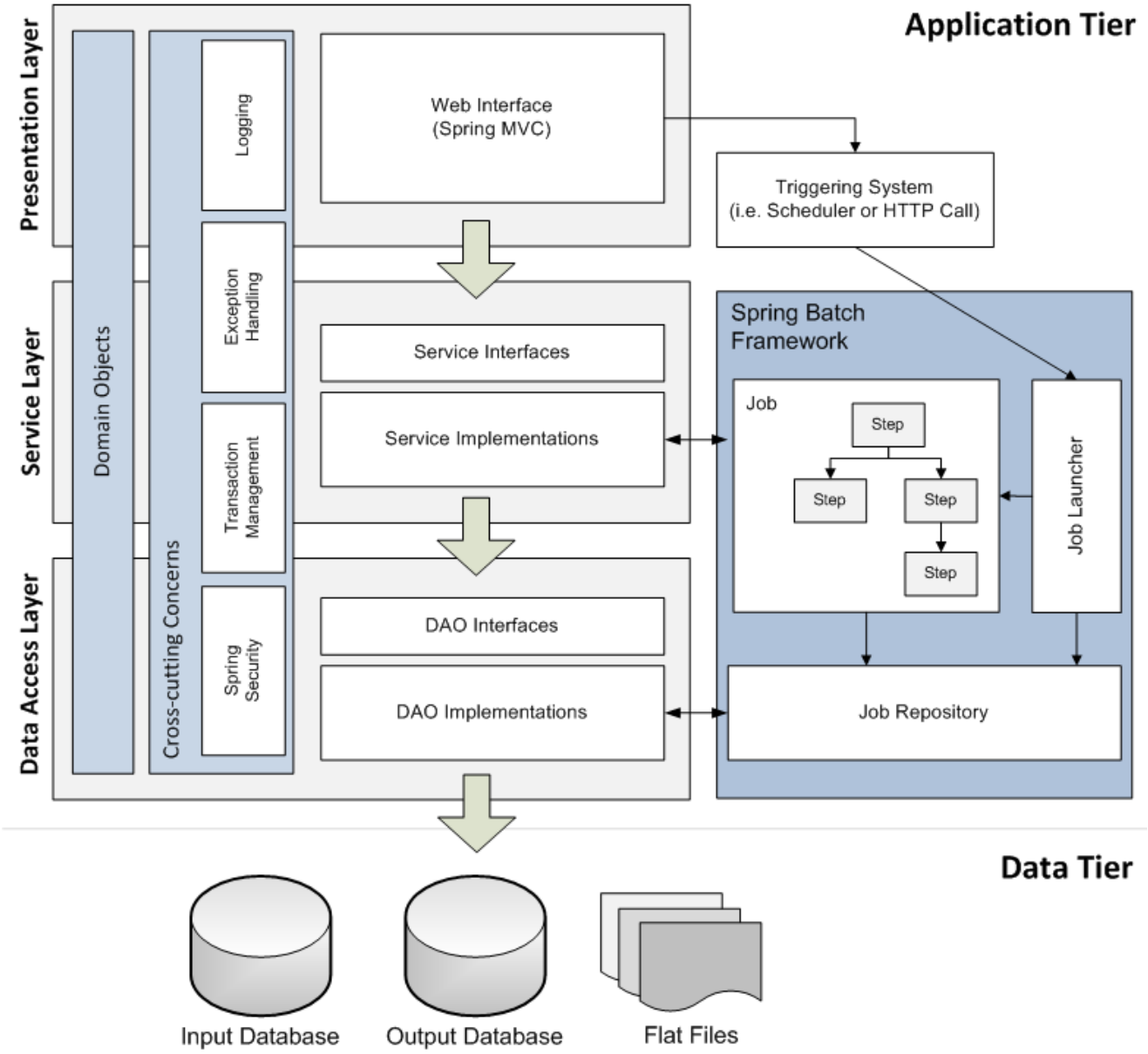
Παράλληλα με το Spring Batch, και με δεδομένο του ότι, όπως περιγράφηκε στο Κεφάλαιο 3, πέρα από την εκτέλεση batch εργασιών απαιτείται και η ύπαρξη ενός graphical user interface που θα ενημερώνει το χρήστη για τα όσα εκτελούνται ή έχουν εκτελεστεί, η εφαρμογή που υλοποιήθηκε έκανε εκτενή χρήση και του Spring Framework. Το Spring Framework αποτελείται από έναν container βασισμένο στο Inversion of Control (IoC) ή κατά άλλους Dependency Injection. Πρόκειται για μια τεχνική όπου υποδεικνύει σε ένα κομμάτι εφαρμογής ποια άλλα κομμάτια μπορεί να χρησιμοποιεί. Για όλες τις εφαρμογές που χτίζονται με το Spring, ο container αποτελεί την καρδιά του συστήματος και όλα τα Java Bean που περιέχει αρχικοποιούνται, παραμετροποιούνται και συναρμολογούνται από τον container. Το framework είναι προϊόν της εταιρίας Springsource και αποτελείται από μια πληθώρα sub-projects[27] που αναπτύσσονται διαρκώς εξελίσσοντας τη λειτουργικότητά του και αυξάνοντας το ήδη τεράστιο μερίδιο της πίτας που πλέον κατέχει στο χώρο των Java EE εφαρμογών.

Για να σχηματοποιήσουμε την γενική αρχιτεκτονική του συστήματος μπορούμε να ανατρέξουμε στο σχήμα που ακολουθεί (Σχήμα 4.1). Η εφαρμογή, αν αφήσουμε προσωρινά εκτός το Spring Batch, χωρίζεται σε τρία επίπεδα (layers): το Data Access Layer (DAO) που βρίσκεται στο χαμηλότερο επίπεδο και αναλαμβάνει την επικοινωνία με τις πηγές δεδομένων (για εμάς βάσεις δεδομένων και flat files), το Service Layer το οποίο περιέχει όλη την business λογική της εφαρμογής και το Presentation Layer που αναλαμβάνει την διασύνδεση του χρήστη με τα πιο κάτω επίπεδα.

Στο Data Access Layer, για την επικοινωνία με τις βάσεις δεδομένων έγινε χρήση του Spring JDBC Template. Το template αυτό απλουστεύει την χρήση JDBC κλήσεων και βοηθά στην αποφυγή συνηθισμένων σφαλμάτων που αφορούν στη διαχείριση των συνδέσεων, των prepared statements κτλ. Μέσω χρήσης παρεχόμενων interfaces ο χρήστης επικεντρώνεται στη λογική της εφαρμογής του αφήνοντας όλα τα υπόλοιπα στο template. Το μόνο που χρειάζεται, όπως θα αναλυθεί και στο επόμενο Κεφάλαιο είναι το configuration του template ώστε να καθοριστεί η πηγή με την οποία θα συνδιαλέγεται και το πώς θα γίνεται αυτό (π.χ. χρήση connection pool).

[27] Περισσότερα για τα projects του Spring Framework εδώ: https://spring.io/projects

Στο Service Layer βρίσκεται όλη η ουσία της εφαρμογής. Το επίπεδο αυτό αναλαμβάνει τη διαχείριση των αιτημάτων που λαμβάνει από το χρήστη μέσω του Presentation Layer και εφόσον χρειάζεται πρόσβαση σε κάποια πηγή δεδομένων ζητά τη βοήθεια του Data Access Layer.



Σχήμα 4.1 Άποψη της γενικής αρχιτεκτονικής του συστήματος

Στο Presentation Layer έγινε χρήση του Spring MVC, ενός sub-project του Spring, που μέσω της Model-View-Controller αρχιτεκτονικής επιτρέπει τη δημιουργία ευέλικτων loosely coupled web εφαρμογών. Το Model είναι τα domain αντικείμενα της εφαρμογής (συνήθως POJO αντικείμενα) διαθέσιμα σε οποιοδήποτε επίπεδο τα χρειάζεται (αυτό αναπαρίσταται και στο Σχήμα 4.1), το View είναι υπεύθυνο για την μετάφραση του Model σε HTML ή κάποιο άλλο τρόπο αναπαράστασης κατανοητό από τον client και ο Controller είναι

αυτός που παίζει τον ρόλο του ενορχηστρωτή μετατρέποντας τα αιτήματα που λαμβάνοντας από το χρήστη σε Model που τελικώς θα στείλει για εμφάνιση σε κάποιο View.

Πέρα των domain αντικειμένων, υπάρχουν και άλλα cross-cutting concerns όπως το transaction management, το exception handling, το logging (η εφαρμογή μας χρησιμοποιεί το Logback[28] της Apache) και φυσικά το Spring Security. Με το τελευταίο, το οποίο αποτελεί επίσης ένα sub-project του Spring, επιτυγχάνεται η εμπορική απαίτηση για authentication και authorization των χρηστών της εφαρμογής. Πρόσβαση στην εφαρμογή θα δίνεται μόνο σε authenticated χρήστες και ο ρόλος τους θα καθορίζει τις ενέργειες που μπορούν να εκτελούν σε αυτή. Αναφέρθηκε και στο προηγούμενο κεφάλαιο η κρισιμότερη από τις εμπορικές απαιτήσεις: μέσω του web interface ο χρήστης θα μπορεί να ξεκινά, σταματά ή επανεκκινεί Spring Batch jobs αλλά μόνο εφόσον ανήκει σε κάποιο ρόλο που να του το επιτρέπει (π.χ. ROLE_ADMIN). Πληροφορίες για το πώς μπορεί να γίνει αυτό θα δοθούν στα ακόλουθα.

Έχοντας την γενική άποψη, μπορούμε τώρα για να κλείσουμε αυτή την ενότητα να δούμε πως συνδέονται όλα αυτά με το Spring Batch και τις ήδη γνωστές έννοιες των Job, Job Launcher και Job Repository που φαίνονται στο Σχήμα 4.1. Το Presentation Layer (άρα ο χρήστης) μέσω ενός Controller (στην εφαρμογή μας JobsController) ζητά από τον Job Launcher –παραδείγματος χάριν– την εκκίνηση κάποιου job. Αυτό την λογική που χρειάζεται να εκτελέσει την ζητά από το Service Layer (π.χ. για τις μεθόδους εκείνες που θα ελέγξουν το validity των δεδομένων πριν αυτά σταλούν στον Batch writer). Το Job, αλλά και το Job Repository, εφόσον χρειάζονται πρόσβαση στο επίπεδο των δεδομένων ζητάνε τη βοήθεια του Data Access Layer.

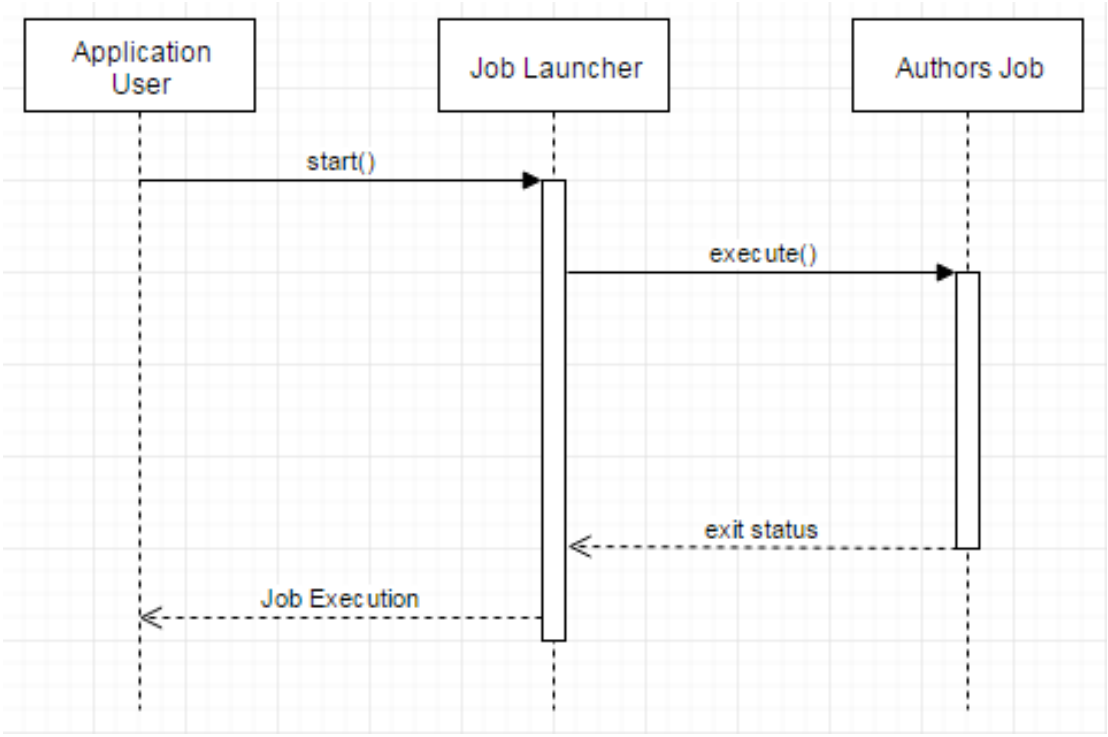
Ακολουθώντας την παραπάνω λογική και χωρίζοντας την εφαρμογή σε επίπεδα μπορούμε να διακρίνουμε τις λειτουργίες όλων των εμπλεκομένων και να έχουμε σαν αποτέλεσμα μια καθαρά γραμμένη και ευκολότερα κατανοητή υλοποίηση. Περισσότερα όμως για όλα αυτά θα δοθούν στο επόμενο κεφάλαιο όπου και θα δίνονται λεπτομέρειες αυτής της υλοποίησης.

4.2 Ακολουθιακά διαγράμματα

Η κλήση του Job Launcher που μόλις περιγράφηκε μπορεί να απεικονιστεί και με το ακολουθιακό διάγραμμα του Σχήματος 4.2. Χρήστης ζητά από τον Job Launcher την εκκίνηση μιας εργασίας (π.χ. του Authors job στο οποίο θα αναφερθούμε και στη συνέχεια) και αυτός ξεκινά την εκτέλεσή του. Με το που ολοκληρωθεί το job, το exit status (επιτυχές ή όχι) επιστρέφεται στον Job Launcher που καταγράφει στο

[28] Περισσότερα για το Logback εδώ: https://logback.qos.ch/

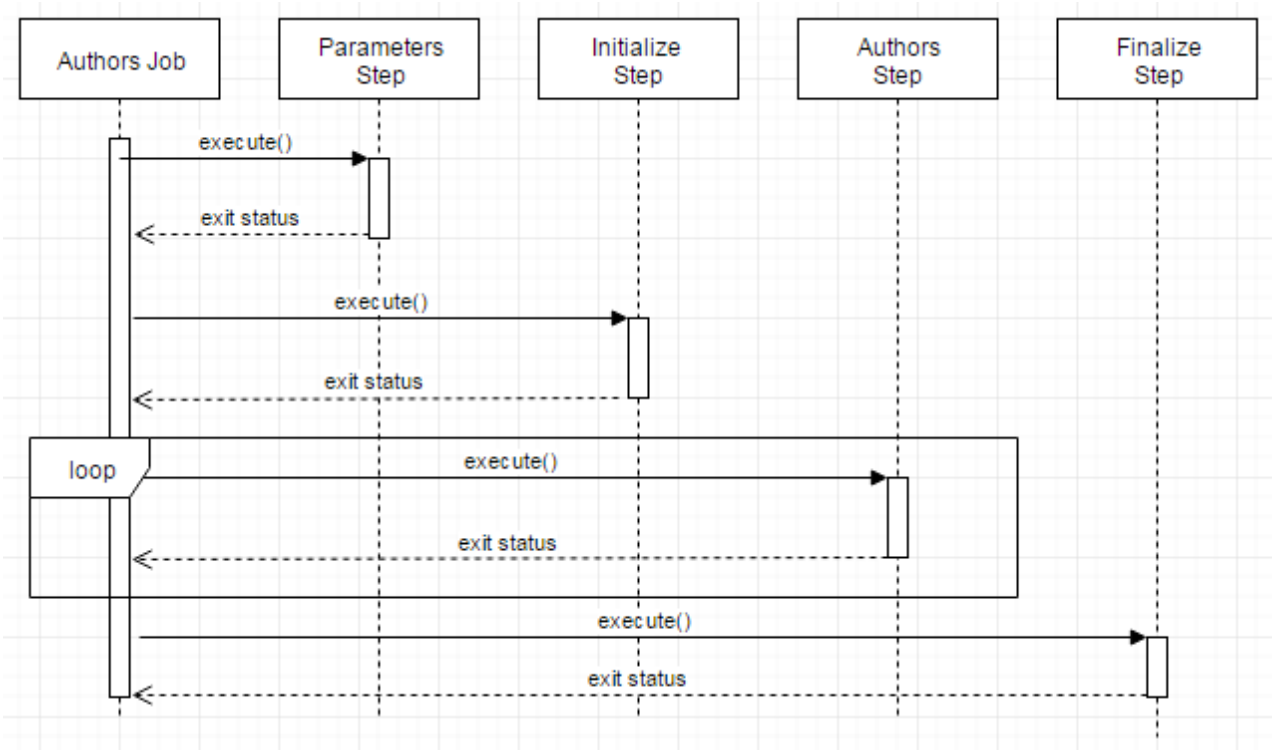
repository τις απαραίτητες πληροφορίες και επιστρέφει (πιθανώς, εφόσον η κλήση είναι σύγχρονη) το status στον χρήστη.



Σχήμα 4.2 Εκτέλεση Spring Batch Job

Το Authors Job αναλύεται περαιτέρω με το ακολουθιακό διάγραμμα του Σχήματος 4.3. Πρώτο βήμα της διαδικασίας είναι η εκτέλεση του Parameters Step: το job εφόσον δεν είναι το initial import χρειάζεται να γνωρίζει για ποιο χρονικό διάστημα θα τρέξει. Ένα tasklet (JobParametersTasklet) διαβάζει την ημερομηνία από και έως για την οποία έτρεξε το προηγούμενο job καθώς και το status του ώστε να αποφασίσει για ποιο διάστημα θα τρέξει. Αν το προηγούμενο job ήταν επιτυχημένο συνεχίζει από εκεί που αυτό σταμάτησε, διαφορετικά ξεκινά από το ίδιο σημείο. Αν είναι το initial import θα τρέξει για το σύνολο των Authors.

Επόμενο βήμα στη διαδικασία είναι το Initialize Step. Μέσω ενός tasklet (JobInfoTasklet) οι ημερομηνίες από και έως που διαβάστηκαν στο προηγούμενο βήμα και τέθηκαν στο job execution context πρέπει να κρατηθούν σε κάποιον βοηθητικό πίνακα της βάσης δεδομένων ώστε να είναι διαθέσιμες στα επόμενα executions. Στον ίδιο πίνακα συντηρείται πληροφορία σχετικά με το status του job (ολοκληρωμένο ή όχι) ώστε να μπορεί να παρθεί η απόφαση για την ημερομηνία από την οποία θα ξεκινήσει το επόμενο execution.

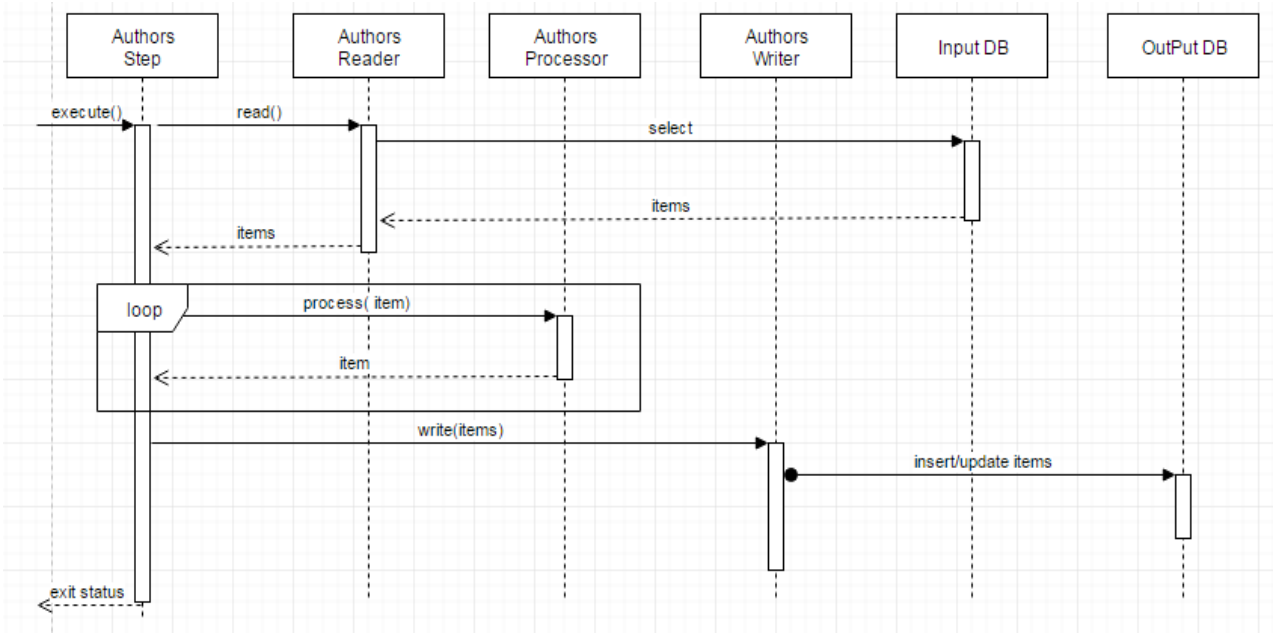


Σχήμα 4.3 Authors Job

Το ακολουθιακό διάγραμμα του Authors Step που ακολουθεί δίνεται στο Σχήμα 4.4. Εδώ έχουμε να κάνουμε με μια παραδοσιακή chunk-oriented Spring Batch εργασία. Ο Authors Reader διαβάζει από την InputDB δεδομένα σχετικά με τους συγγραφείς σε δεσμίδες των χιλίων, και τις στέλνει μια-μια στον Authors Processor που έχει να εκτελέσει τα validations που αναφέρθηκαν στο προηγούμενο κεφάλαιο. Όσες εγγραφές δεν είναι έγκυρες φιλτράρονται, ενώ οι υπόλοιπες στέλνονται στον Authors Writer (πάλι σε δεσμίδες) που αναλαμβάνει την εγγραφή τους στην OutputDB (INSERT ή UPDATE).

Τελευταίο βήμα στη διαδικασία, αφού ολοκληρωθεί το Authors Step, είναι το Finalize Step. Εδώ εκτελείται και πάλι το JobInfoTasklet μόνο που τώρα απλά ενημερώνει το status του job ως ολοκληρωμένο (αν κάποιο σφάλμα παρουσιαστεί στα προηγούμενα βήματα το update αυτό δεν θα εκτελεστεί ποτέ και άρα το execution από τα επόμενα executions θα θεωρηθεί ως αποτυχημένο) και γίνεται υπολογισμός των εγγραφών που τροποποιήθηκαν ανά οντότητα εν είδη στατιστικών στοιχείων.

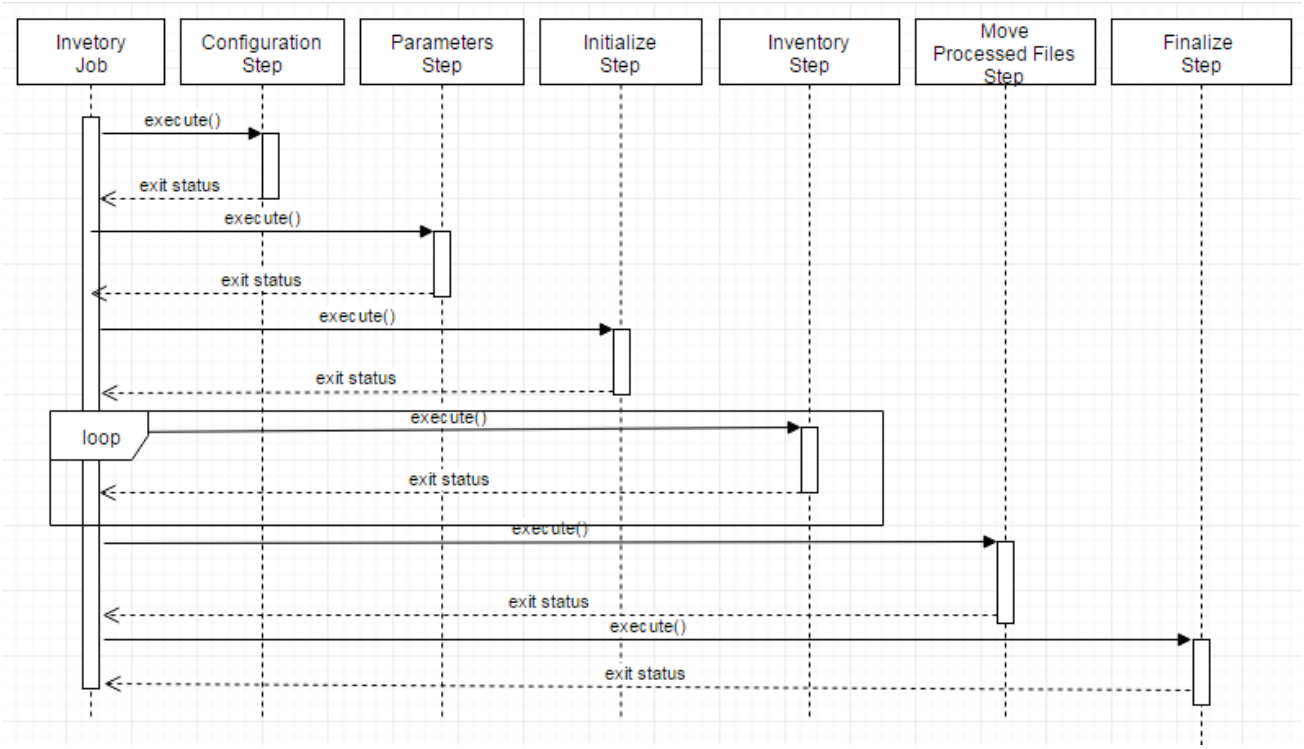
Η ροή αυτή ακολουθείται για οποιοδήποτε job διαβάζει από βάση δεδομένων και γράφει σε βάση δεδομένων. Άρα και το Products job που θα παρουσιαστεί στο επόμενο κεφάλαιο είναι ακριβώς το ίδιο. Μόνη διαφορά πως στο στάδιο της επεξεργασίας περιέχει αρκετά περισσότερη λογική ώστε να εμπλουτιστεί, όπως αναφέρθηκε, το Product με πληροφορίες που αφορούν τις κατηγορίες του, τους συγγραφείς του κλπ.



Σχήμα 4.4 Authors Step

4.2.1 Inventory Job

Κάποιες τροποποιήσεις στο ακολουθιακό διάγραμμα υπάρχουν εφόσον μιλάμε για job που διαβάζει από αρχείο και γράφει σε βάση δεδομένων (π.χ. το Inventory Job). Πριν από το Parameters Step, όπως φαίνεται και στο Σχήμα 4.5, εκτελείται το Configuration Step το οποίο διαβάζει από τη βάση βοηθητικό πίνακα της βάσης δεδομένων πληροφορίες σχετικά με το path στο οποίο αναμένεται να βρεθεί το αρχείο, τον τύπο του, το pattern του ονόματός του κτλ. Οι πληροφορίες αυτές αποθηκεύονται στο execution context του job ώστε να είναι διαθέσιμες από όποιο επόμενο βήμα τις χρειάζεται. Επιπρόσθετα, πριν ολοκληρωθεί το job (με την εκτέλεση του Finalize Step) υπάρχει ένα ακόμα βήμα (Move Processed Files Step) που αναλαμβάνει να μεταφέρει σε έναν άλλο φάκελο του file system το επεξεργασμένο αρχείο ώστε αφενός να υπάρχει σαν ιστορικό και αφετέρου να μην επεξεργαστεί εκ νέου όταν ξεκινήσει το επόμενο job execution. Σε αυτό τον τύπο των jobs, το Initialize Step ναι μεν γράφει πληροφορίες σχετικά με τα αρχεία που διαβάστηκαν αλλά, επειδή τα αρχεία μπορεί να έρχονται πάντα με το ίδιο όνομα, δεν βασιζόμαστε σε αυτές ώστε να αποφασίζουμε αν ένα αρχείο που βρέθηκε στον καθορισμένο φάκελο πρέπει να επεξεργαστεί ή όχι. Κάθε αρχείο που επεξεργάζεται μεταφέρεται στο φάκελο με τα επεξεργασμένα αρχεία και αυτή είναι μια παραδοχή που προς το παρόν καλύπτει πλήρως τις εμπορικές απαιτήσεις.



Σχήμα 4.5 Inventory Job

ΚΕΦΑΛΑΙΟ 5
Υλοποίηση

5.1 Περιβάλλοντα Εκτέλεσης της Εφαρμογής

Όπως όλες οι εφαρμογές που προορίζονται για παραγωγική χρήση, η εφαρμογή μας είναι απαραίτητο να παραμετροποιείται με τρόπο ώστε να μπορεί να γίνει deploy σε διαφορετικά περιβάλλοντα εκτέλεσης. Ένα περιβάλλον εκτέλεσης τυπικά περιλαμβάνει έναν ή περισσότερους web servers, καθώς και μια ή περισσότερες βάσεις δεδομένων. Τα συνηθέστερα χρησιμοποιούμενα περιβάλλοντα είναι το development, στο οποίο γίνεται η ανάπτυξη της εφαρμογής και το παραγωγικό, το οποίο αποτελεί το τελικό περιβάλλον χρήσης. Ανάμεσα στο development και το παραγωγικό, μπορεί να υπάρχουν και άλλα περιβάλλοντα (staging, user acceptance testing κλπ), όμως στην παρούσα εργασία η εφαρμογή χρησιμοποιεί μόνο τα δυο προαναφερθέντα.

Καθένα από τα περιβάλλοντα έχει στην περίπτωσή μας ξεχωριστή παραμετροποίηση για τις συνδέσεις με τις βάσεις του και για το server directory μέσα στο οποίο θα γράφονται τα log files της εφαρμογής. Οι παράμετροι κάθε περιβάλλοντος, ενσωματώνονται στην εφαρμογή κατά το build του WAR file με τη βοήθεια του εργαλείου Maven.

5.1.1 Το εργαλείο Maven

Το Maven αποτελεί ένα εργαλείο αυτοματοποίησης της διαδικασίας του build μιας Java εφαρμογής. Σχετικά με το build, παρέχει δύο κύριες ευκολίες: περιγράφει τα βήματα της διαδικασίας και χειρίζεται τα dependencies του project (κατεβάζει δηλαδή δυναμικά τα libraries που αποτελούν εξαρτήσεις του). Το configuration του Maven ορίζεται by default στο αρχείο **pom.xml** που πρέπει να βρίσκεται στο root directory του project**.** Σε αυτό, μπορούμε για παράδειγμα να ζητήσουμε το download της jstl βιβλιοθήκης στην έκδοση 1.2 από τα Maven repositories στο classpath της εφαρμογής μας με configuration σαν και αυτό που δίνεται στην Listing 5.1.

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>jstl</artifactId>
  <version>1.2</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
```

```
  <version>${spring-version}</version>
  <exclusions>
    <exclusion>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Listing 5.1 Maven dependencies

Όπως φαίνεται στην ίδια Listing, είναι δυνατό να οριστούν και exclusions για τις εξαρτήσεις που μπορεί με τη σειρά του να φέρει κάποιο dependency. Στα προηγούμενα ορίζουμε ότι θέλουμε να έρθει στο classpath η βιβλιοθήκη spring-context του Spring Framework, χωρίς όμως τη βιβλιοθήκη commons-logging (που κανονικά αποτελεί τμήμα της), καθώς εμείς για logging θα χρησιμοποιήσουμε το εργαλείο Logback της Apache (περισσότερα στα επόμενα).

Επιπλέον, μέσω του Maven χειριζόμαστε την παραμετροποίηση ανά περιβάλλον που αναφέρθηκε παραπάνω. Για παράδειγμα στο configuration που ακολουθεί, το dev περιβάλλον ορίζεται ως το **activeByDefault** και καθορίζεται πως τα parent configuration αρχεία που βρίσκονται στο /src/main/resources, με χρήση ενός φίλτρου θα ενημερώνονται κατά το build με τις τιμές που για το περιβάλλον υπάρχουν στο /dev/configuration.properties αρχείο. Τοποθετώντας δηλαδή σε αυτό το αρχείο όσα properties διαφέρουν για το dev περιβάλλον και εκτελώντας το build με μια εντολή σαν την **mvn clean install -Pdev** επιτυγχάνουμε την ζητούμενη παραμετροποίηση. Εντελώς ανάλογα δημιουργείται και το production profile. Αρκεί να καθορίσουμε ένα διαφορετικό properties αρχείο (/prod/configuration.properties) και να εκτελέσουμε το build με την παράμετρο **-Pprod**. Αν η παράμετρος παραληφθεί το WAR θα χτιστεί με τις τιμές του activeByDefault περιβάλλοντος.

```
<profile>
  <id>dev</id>
  <activation>
    <activeByDefault>true</activeByDefault>
  </activation>
  <build>
    <filters>
      <filter>src/main/resources/profiles/dev/configuration.properties</filter>
    </filters>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
        <filtering>true</filtering>
      </resource>
    </resources>
  </build>
</profile>
```

Listing 5.2 Configuration Maven profile

Το configuration.properties αρχείο για το dev περιβάλλον είναι αυτό που φαίνεται στην Listing 5.2. Σε αυτό ορίζονται το path στο οποίο θα γράφει τα logs η εφαρμογή, και τα στοιχεία που χρειάζονται για να συνδεθούμε στις input και output βάσεις δεδομένων. Το configuration.properties της παραγωγής περιέχει προφανώς ίδια properties (κλειδιά) με διαφορετικές τιμές.

```
#Log Path
profile.log.path=/dev/apache-tomcat-7.0.69/logs/BooksAdmin

#MySql  Database Properties
profile.batch.jdbc.driver=com.mysql.jdbc.Driver
profile.batch.jdbc.url=jdbc:mysql://127.0.0.1:3306/digibooks?autoReconnect=true&characterEncoding=UTF-
8&zeroDateTimeBehavior=convertToNull
profile.batch.jdbc.user=username
profile.batch.jdbc.password=password

#Oracle Database Properties
profile.batch.ojdbc.driver=oracle.jdbc.OracleDriver
profile.batch.ojdbc.dataSourceName=DS
profile.batch.ojdbc.url=
profile.batch.ojdbc.user=username
profile.batch.ojdbc.password=password
```

Listing 5.3 configuration.properties του dev περιβάλλοντος

Τα parent configuration αρχεία (στο παράδειγμά μας μόνο ένα, το configuration.properties) ορίζουν τα properties που το παραγόμενο WAR θα χρησιμοποιεί, μόνο που αντί για τιμές έχουν placeholders που αναφέρονται στα κλειδιά των child configuration αρχεία. Το να αντικατασταθούν τα placeholders με τις πραγματικές τιμές για το περιβάλλον είναι κάτι που το αναλαμβάνει το Maven κατά το build με τον τρόπο που ήδη περιγράφηκε.

```
#Log Path
log.path=${profile.log.path}

#MySql  Database Properties
batch.jdbc.driver=${profile.batch.jdbc.driver}
batch.jdbc.url=${profile.batch.jdbc.url}
batch.jdbc.user=${profile.batch.jdbc.user}
batch.jdbc.password=${profile.batch.jdbc.password}

#Oracle  Database Properties
batch.ojdbc.driver=${profile.batch.ojdbc.driver}
batch.ojdbc.dataSourceName=${profile.batch.ojdbc.dataSourceName}
batch.ojdbc.url=${profile.batch.ojdbc.url}
batch.ojdbc.user=${profile.batch.ojdbc.user}
batch.ojdbc.password=${profile.batch.ojdbc.password}
```

Listing 5.4 Γονικό configuration.properties αρχείο

Τέλος, το Maven είναι χτισμένο με μια plug-in based αρχιτεκτονική, κάτι που επιτρέπει να επεκταθούν οι κύριες λειτουργίες του μέσω της χρήσης plug-ins. Εμείς χρησιμοποιούμε το **compiler plug-in** για το compile της εφαρμογής, το **surefire plug-in** για παραγωγή στατιστικών σχετικά με τα JUnit tests, καθώς και το

Liquibase[29] plug-in για τη δημιουργία και την ενημέρωση της output database με αλλαγές που πιθανώς γίνονται κατά το development ή προκύπτουν ως change requests.

Το Liquibase plug-in ορίζεται στο pom.xml με τον τρόπο που παρουσιάζεται στην Listing 5.5. Το configuration file του ορίζεται πως βρίσκεται στον /target/classes/liquibase/ φάκελο και περιέχει properties (π.χ. URL, username, password της βάσης) που μέσω placeholders όπως περιγράφηκε στα προηγούμενα γίνονται generate κατά το build του εκάστοτε περιβάλλοντος. Με δεδομένο το ότι το εργαλείο επικοινωνεί με μια βάση δεδομένων, πρέπει να οριστεί ως dependency του plug-in και ο driver που θα χρησιμοποιηθεί για τη σύνδεση με αυτή (για εμάς ο ojdbc6 της Oracle). Κάποιες επιπλέον πληροφορίες για το συγκεκριμένο εργαλείο δίνονται εν συντομία στην υποενότητα που ακολουθεί.

```
<plugin>
  <groupId>org.liquibase</groupId>
  <artifactId>liquibase-maven-plugin</artifactId>
  <version>3.3.2</version>
  <configuration>
     <propertyFile>/target/classes/liquibase/liquibase.properties</propertyFile>
    <propertyFileWillOverride>true</propertyFileWillOverride>
    <promptOnNonLocalDatabase>false</promptOnNonLocalDatabase>
  </configuration>
  <dependencies>
    <dependency>
      <groupId>com.oracle</groupId>
      <artifactId>ojdbc6</artifactId>
      <version>${oracle-version}</version>
    </dependency>
  </dependencies>
</plugin>
```

Listing 5.5 Configuration του Liquibase plug-in

5.1.2 Το εργαλείο Liquibase

Το Liquibase είναι μια database-independent βιβλιοθήκη ανοιχτού κώδικα, η οποία χρησιμοποιείται για την καταγραφή, διαχείριση και εφαρμογή αλλαγών στο σχήμα και τα δεδομένα μιας βάσης δεδομένων. Το μεγαλύτερο πλεονεκτήματα της χρήσης του, είναι η εύκολη καταγραφή οποιασδήποτε αλλαγής συμβαίνει σε μια βάση δεδομένων, πράγμα ιδιαίτερα χρήσιμο σε σύγχρονα software projects μεγάλης κλίμακας όπου πολλοί developers δουλεύουν ταυτόχρονα και πιθανότατα σε περισσότερα του ενός branches.

Όλες οι αλλαγές που αφορούν τη βάση αποθηκεύονται σε XML αρχεία που καλούνται changelog files, με την κάθε μια να προσδιορίζεται μοναδικά από έναν συνδυασμό **id** και **author** (όνομα του developer που έγραψε την αλλαγή). Το Liquibase αυτόματα δημιουργεί στη βάση δύο πίνακες τους **DATABASECHANGELOG** και **DATABASECHANGELOGLOCK** την πρώτη φορά που θα εκτελεστεί. Η λίστα με

[29] Περισσότερα για το liquibase εδώ: http://www.liquibase.org/

όλες τις αλλαγές που διενεργούνται, αποθηκεύονται στον πρώτο πίνακα και το εργαλείο τον συμβουλεύεται κάθε φορά που πρόκειται να κάνει update ώστε να καθορίσει ποιες αλλαγές έχουν ήδη εφαρμοστεί και ποιες όχι. Για να αποφευχθεί η περίπτωση περισσότεροι του ενός clients να προσπαθήσουν να εκτελέσουν ταυτόχρονα τα ίδια changelogs, χρησιμοποιείται ο δεύτερος πίνακας στον οποίο αποθηκεύεται μια εγγραφή κλειδώματος με το που η διαδικασία ξεκινά, και διαγράφεται (το κλείδωμα ελευθερώνεται) με το που αυτή ολοκληρωθεί.

Για λόγους καλύτερης εποπτείας, προτιμάται να γράφονται σε ξεχωριστά changelog αρχεία οι αλλαγές που αφορούν δεδομένα (dml), από αυτές που αφορούν το σχήμα της βάσης (ddl). Εμείς χρησιμοποιήσαμε ένα αρχείο **changelog-ddl.xml** για τις αλλαγές στο σχήμα, και ένα **changelog-dml.xml** για τις αλλαγές που αφορούν τα δεδομένα. Τα δύο αυτά αρχεία ενσωματώνονται σε ένα parent changelog, το **changelog-master.xml** (βλ. Listing 5.6). Εκτελώντας την εντολή **liquibase update**, το Liquibase διαβάζει τις αλλαγές ξεκινώντας από το changelog-master.xml και στη συνέχεια συμβουλεύεται τους δύο πίνακες του ώστε να βρει και να εκτελέσει όσες από αυτές είναι νέες. Για να εκτελεστεί οποιαδήποτε ενέργεια πρέπει να ικανοποιούνται μια σειρά από προαπαιτήσεις (**<preconditions/>**). Εδώ για παράδειγμα ορίζουμε πως η βάση στην οποία το εργαλείο θα γράφει πρέπει να είναι **oracle**.

```
<databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
    http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.1.xsd">

  <preConditions>
    <dbms type="oracle" />
  </preConditions>
  <!-- Included Files -->
  <include file="src/main/resources/liquibase/changelog-ddl.xml" />
  <include file="src/main/resources/liquibase/changelog-dml.xml" />
</databaseChangeLog>
```

Listing 5.6 Το αρχείο changelog-master.xml

Οι προς εκτέλεση SQL εντολές, όπως φαίνεται και στις ακόλουθες Listings, μπορούν είτε να ενσωματώνονται απευθείας στο changeset (**<sql/>** block), είτε να παρέχονται σε ξεχωριστό αρχείο .sql εντός του classpath, με το changeset να περιλαμβάνει το path προς το αρχείο (**<sqlFile/>** block). Τα πλήρη .sql αρχεία που εκτελέστηκαν στην output database, παραθέτονται στο Παράρτημα. Αξίζει να αναφερθεί πως το Liquibase παρέχει και την δυνατότητα σύνταξης των επιθυμητών αλλαγών με χρήση XML ή JSON σύνταξης ώστε να μην δένεται η υλοποίηση σε κάποιο συγκεκριμένο DBMS. Αυτή η δυνατότητα, αν και πολύ ενδιαφέρουσα, δεν χρησιμοποιήθηκε στα πλαίσια της παρούσας εργασίας.

```
<changeSet author="eskiada" id="ddl-12">
  <comment>ADD COLUMNS TO CB_PRODUCT</comment>
  <sql>
    ALTER TABLE CB_PRODUCT ADD ("MANUFACTURERS" VARCHAR2(255 CHAR) DEFAULT NULL);
```

```
    ALTER TABLE CB_PRODUCT ADD ("CATEGORIES" CLOB DEFAULT NULL);
    ALTER TABLE CB_PRODUCT ADD ("SUBJECTS" VARCHAR2(255 CHAR) DEFAULT NULL);
    ALTER TABLE CB_PRODUCT ADD ("COVERS" VARCHAR2(255 CHAR) DEFAULT NULL);
    ALTER TABLE CB_PRODUCT ADD ("BOOK_AGES" VARCHAR2(255 CHAR) DEFAULT NULL);
  </sql>
</changeSet>
```

Listing 5.7 Παράδειγμα changeset με χρήση του <sql/> block

```
<changeSet author="eskiada" id="ddl-1">
  <comment>CREATE SPRING BATCH TABLES</comment>
  <sqlFile path="/src/main/resources/liquibase/oracle/spring_batch.sql"/>
</changeSet>
```

Listing 5.8 Παράδειγμα changeset με χρήση του <sqlFile/> block

5.1.3 Το εργαλείο Logback

Σε όλα το σύγχρονα software projects, το logging, η καταγραφή δηλαδή μηνυμάτων κατά τη διάρκεια εκτέλεσης του κώδικα, θεωρείται απαραίτητη τεχνική για την καθοδήγηση των προγραμματιστών στη διαδικασία απασφαλμάτωσης (debugging) του κώδικα. Τα μηνύματα (logs) γράφονται σε ένα ή περισσότερα αρχεία και υποδεικνύουν λάθη και exceptions στον κώδικα κατά την εκτέλεση της εφαρμογής. Για το σκοπό αυτό εμείς χρησιμοποιήσαμε το εργαλείο Logback, το οποίο είναι ένα από τα ευρύτερα χρησιμοποιούμενα εργαλεία logging για Java projects, και αποτελεί διάδοχο του πασίγνωστου Log4j.

Το Logback καθορίζει 5 ιεραρχικά επίπεδα σοβαρότητας του log (TRACE, DEBUG, INFO, WARN, ERROR), και κάθε Log ορίζεται να ανήκει σε ένα καθορισμένο επίπεδο. Στο configuration του (αρχείο logback.xml) ορίσαμε ότι θα γράφονται στην έξοδο τα log που ανήκουν στο επίπεδο DEBUG και όσα είναι ιεραρχικά ανώτερα του (δεν θα γράφονται δηλαδή τα TRACE logs).

Τα σημεία (έξοδοι) που γράφονται τα logs δηλώνονται ως appenders (βλ. και την Listing 5.9). Για εμάς appenders είναι η CONSOLE (standard output) και ένα FILE, το path του οποίου καθορίζεται κατά το build ανάλογα με το profile που χρησιμοποιείται. Για να μην δημιουργούνται πολύ μεγάλα αρχεία, ορίζουμε ότι μόλις ένα logging αρχείο φτάσει σε μέγεθος τα 50MB, θα γίνεται rolled, και τα logs θα γράφονται σε νέο αρχείο. Ο μέγιστος αριθμός αρχείων που για λόγους ιστορικού θα κρατιόνται είναι 5. Τα παλιότερα αυτόματα διαγράφονται. Τέλος, με τον encoder **[%d{ISO8601}] %5p [%c{0}]: %m%n** καθορίζουμε το φορμάτ του μηνύματος στο αρχείο, ώστε αν για παράδειγμα ο κώδικάς μας στην κλάση ProductsDaoImpl χρησιμοποιήσει μια κλήση σαν την:

```
logger.debug("Inserting/Updating Product With ID = {}.", product.getId());
```

το μήνυμα που θα καταγράφεται στα logs να είναι το ακόλουθο:

[2017-02-02 13:19:18,332] DEBUG [ProductsDaoImpl]: Inserting/Updating Product With ID = 7704404.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <contextName>booksAdmin</contextName>
  <property resource="configuration.properties"/>

  <!-- Console Appender -->
  <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
    <target>System.out</target>
    <encoder>
      <pattern>[%d{ISO8601}] %5p [%c{0}]: %m%n</pattern>
    </encoder>
  </appender>
  <!-- File Appender -->
  <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>${log.path}/BooksAdmin.log</file>
    <!--  File Size Rolling Policy-->
    <rollingPolicy class="ch.qos.logback.core.rolling.FixedWindowRollingPolicy">
      <fileNamePattern>${log.path}/BooksAdmin.log.%i</fileNamePattern>
      <minIndex>1</minIndex>
      <maxIndex>5</maxIndex>
    </rollingPolicy>
    <triggeringPolicy class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
      <maxFileSize>50MB</maxFileSize>
    </triggeringPolicy>
    <encoder>
      <pattern>[%d{ISO8601}] %5p [%c{0}]: %m%n</pattern>
    </encoder>
  </appender>
  <!-- Loggers -->
  <logger name="gr.booksAdmin" level="DEBUG" />
  <root level="INFO">
    <appender-ref ref="CONSOLE" />
    <appender-ref ref="FILE" />
  </root>
</configuration>
```

Listing 5.9 Το logback.xml αρχείο

5.1.4 Web Server, JDK και Περιβάλλον ανάπτυξης κώδικα

Για το deploy του παραγόμενου από το Maven WAR της εφαρμογής, θα χρησιμοποιήσουμε τον **Apache Tomcat**[30] στην έκδοση 7.0.69, τον οποίο εγκαταστήσαμε τοπικά ως μέρος του dev περιβάλλοντος. Για το prod περιβάλλον, θα εγκατασταθεί η ίδια έκδοση του Apache Tomcat στα παραγωγικά μηχανήματα της εταιρίας.

[30] Περισσότερα για τον Apache Tomcat εδώ: http://tomcat.apache.org/

Για την εκτέλεση της εφαρμογής χρησιμοποιήθηκε το περιβάλλον **Java SDK 1.7,** ενώ για την ανάπτυξη του κώδικα το ενοποιημένο περιβάλλον ανάπτυξης κώδικα (IDE) **IntelliJ IDEA**[31] της Jetbrains, στην έκδοση 14.0.3.

5.2 Παραμετροποίηση του project

Η παραμετροποίηση του τρόπου deployment του project στον server, καθώς και όλων των modules της εφαρμογής (Spring Batch infrastructure, Spring MVC, Spring Security, των JDBC templates που θα επιτρέψουν τη σύνδεση με τις βάσεις δεδομένων κλπ.), γίνεται μέσω XML configuration.

5.2.1 web.xml

Το αρχείο web.xml αποτελεί το κεντρικό αρχείο μιας web εφαρμογής. Εκεί δίνονται οι οδηγίες για τον τρόπο που αυτή θα γίνεται deploy στον web server. Στην περίπτωση χρήσης ενός web framework όπως το Spring, στο web.xml προσδιορίζεται το mapping των URL διαμέσου του Dispatcher Servlet ο οποίος είναι υπεύθυνος για την δρομολόγηση όλων των HTTP αιτήσεων που λαμβάνει ο server προς τις κατάλληλες μεθόδους χειρισμού. Το configuration του Dispatcher Servlet έχει ως εξής:

```
<servlet>
  <servlet-name>mvc-dispatcher</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/spring/dispatcher-context.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>mvc-dispatcher</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

Listing 5.10 Configuration του Spring Dispatcher Servlet

Επειδή χρησιμοποιούμε Spring Security για authentication και authorization της πρόσβασης των χρηστών στη γραφική διεπαφή χρήστη, σε κάθε εισερχόμενο HTTP request παρεμβάλλονται τα φίλτρα του Spring Security (DelegatingFilterProxy) ώστε να ταυτοποιείται ο χρήστης που αιτείται την προβολή της κάθε σελίδας ή resource. Τα φίλτρα αυτά, καθώς παρεμβάλλονται όλων των HTTP κλήσεων, ορίζονται επίσης στο web.xml με τον τρόπο που φαίνεται στην ακόλουθη Listing.

[31] Πληροφορίες για το IntelliJ IDEA εδώ: https://www.jetbrains.com/idea/

```
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Listing 5.11 Configuration του DelegatingFilterProxy

Τέλος, στο ίδιο αρχείο δηλώνονται όλα τα parent configuration αρχεία που χρησιμοποιούνται από τα modules της εφαρμογής, ώστε να μπορούν να γίνουν οι κατάλληλες αρχικοποιήσεις κατά την εκκίνηση της. Ο τρόπος για να δηλωθούν είναι ο ακόλουθος:

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    /WEB-INF/spring/application-context.xml
    /WEB-INF/spring/jobs-context.xml
    /WEB-INF/spring/tasks-context.xml
    /WEB-INF/spring/security-context.xml
  </param-value>
</context-param>
```

Listing 5.12 Δήλωση των configuration αρχείων της εφαρμογής

Μηδενός εξαιρουμένου, όλα τα παραπάνω αρχεία έχουν ιδιαίτερο ενδιαφέρον και για αυτό παρουσιάζονται αναλυτικά στη συνέχεια μαζί με τα αντίστοιχα modules όπου αυτό κρίνεται αναγκαίο.

5.2.2 application-context.xml

Το application-context.xml αποτελεί το κεντρικό configuration file σε επίπεδο application. Σε αυτό το αρχείο ορίζουμε το root package, ώστε, κατά την εκκίνηση της εφαρμογής, να ανιχνεύσει ο Spring Context Loader Listener τις annotated κλάσεις (@Component, @Repository, @Service, @Controller κ.α.), και να τις αρχικοποιήσει ως beans στον Spring Container. Από το scanning εξαιρούμε τις κλάσεις των Controllers, καθώς το framework απαιτεί αυτές να δηλώνονται προς scanning στο configuration αρχείο του Spring MVC module, το οποίο είναι το dispatcher-context.xml και θα παρουσιαστεί παρακάτω.

```
<!-- Register All Annotations Except @Controller -->
<context:component-scan base-package="gr.booksAdmin">
  <context:exclude-filter expression="org.springframework.stereotype.Controller" type="annotation"/>
</context:component-scan>
```

Στη συνέχεια δηλώνονται data sources για τις input και output databases και συνδέονται με τα αντίστοιχα Jdbc Templates, τα οποία είναι interfaces του Spring framework υπεύθυνα για την υλοποίηση σύνδεσης με μια σχεσιακή βάση δεδομένων, και την εκτέλεση ερωτημάτων δια μέσου αυτής της σύνδεσης χωρίς να

χρειάζεται ο χρήστης να ασχολείται με τον boilerplate κώδικα που συναντάται κατά κόρον σε παλαιότερες εφαρμογές και έχει να κάνει με το άνοιγμα και το κλείσιμο connections και λοιπών resources.

```
<!-- MySQL Database To Read From -->
 <bean id="inputDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="${batch.jdbc.driver}" />
     <property name="url" value="${batch.jdbc.url}" />
    <property name="username" value="${batch.jdbc.user}" />
    <property name="password" value="${batch.jdbc.password}" />
 </bean>
  <!--Oracle Database To Write To -->
 <bean id="outputDataSource" class="oracle.jdbc.pool.OracleDataSource">
    <property name="dataSourceName" value="${batch.ojdbc.dataSourceName}"/>
    <property name="URL" value="${batch.ojdbc.url}"/>
    <property name="user" value="${batch.ojdbc.user}"/>
    <property name="password" value="${batch.ojdbc.password}"/>
</bean>
<!--JDBC Templates-->
<bean id="inputJdbcTemplate"
class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">
   <constructor-arg ref="inputDataSource"/>
</bean>
<bean id="outputJdbcTemplate"
class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">
   <constructor-arg ref="outputDataSource"/>
</bean>
```

Listing 5.13 Configuration για τα data sources και τα Jbdc Templates

Όπως φαίνεται παραπάνω, οι τιμές για τα properties των data sources λαμβάνονται από placeholders κατά το build του project, αφού όπως εξηγήσαμε οι βάσεις που χρησιμοποιούνται είναι profile-specific, και άρα το configuration της Listing 5.13 πρέπει να είναι σε θέση να λαμβάνει διαφορετικές τιμές για τα dev και prod περιβάλλοντα.

Στο Κεφάλαιο 2, αναφέραμε ότι το Spring Batch Framework, χρειάζεται ένα transaction manager, για τον χειρισμό των transactions που αφορούν τις εγγραφές στην output database, ώστε να εξασφαλίζει την ορθή ACID λειτουργία των συνδιαλλαγών με αυτή. Το transaction manager bean δηλώνεται στο application-context.xml και σαν property όρισμά του δέχεται το data source επί του οποίου θα ενεργήσει (εδώ outputDataSource).

```
<!--Transaction Manager for Consolidation Data Source-->
<bean id="transactionManager" lazy-init="true"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
   <property name="dataSource" ref="outputDataSource" />
</bean>
```

Listing 5.14 Configuration του Transaction Manager

Τέλος, εδώ δηλώνονται τα configuration αρχεία που θα χρησιμοποιεί η εφαρμογή και είναι: το **configuration.properties** (το parent configuration αρχείο για το οποίο μιλήσαμε παραπάνω) και το

email.properties αρχείο που περιέχει τιμές σχετικές με την υλοποίηση αποστολής email την οποία παρουσιάζουμε αναλυτικά στα επόμενα. Για την διαχείριση του property placeholding χρησιμοποιούμε την παρεχόμενη από το Spring **PropertyPlaceholderConfigurer** κλάση ως ένα bean που δηλώνεται με τον εξής τρόπο:

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="locations">
    <list>
      <value>classpath:/configuration.properties</value>
      <value>classpath:/email.properties</value>
    </list>
  </property>
</bean>
```

Listing 5.15 Configuration του PropertyPlaceholderConfigurer

5.2.3 dispatcher-context.xml

Όπως είδαμε στο web.xml ο Dispatcher Servlet δέχεται ως παράμετρο αρχικοποίησης (init param) το αρχείο dispatcher-context.xml. Σε αυτό το αρχείο δηλώνουμε το scanning των Controller κλάσεων, που είχε εξαιρεθεί από το component-scanning που δηλώσαμε στο application-context.xml. Έπειτα, εδώ ορίζεται ένα από τα κυριότερα συστατικά μέρη του Spring MVC module, ο view resolver, το bean δηλαδή που χειρίζεται το resolving των views στο presentation layer. Ο view resolver λαμβάνει τα λογικά view names που επιστρέφονται από τις μεθόδους των Controllers που χειρίστηκαν μια HTTP αίτηση, και τα αντιστοιχίζει σε πραγματικά view names (JSP ή HTML σελίδες), που επιστρέφονται στο χρήστη. Όπως φαίνεται και παρακάτω, εμείς χρησιμοποιήσαμε την υλοποίηση InternalResourceViewResolver που παρέχει το Spring. Στα λογικά view names που επιστρέφονται από τους Controllers αυτού του τύπου ο Resolver προσθέτει τα prefix και suffix ώστε ένα input σαν το “somepage” να αντιστοιχιστεί στην /WEB-INF/pages/somepage.jsp.

```
<!-- Enable Annotations-->
<annotation-driven />
<!-- Register @Controller Annotations -->
<context:component-scan base-package="gr.booksAdmin" use-default-filters="false">
  <context:include-filter expression="org.springframework.stereotype.Controller" type="annotation" />
</context:component-scan>
<!-- Resources + View Resolver -->
<resources mapping="/resources/**" location="/resources/" />
<beans:bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <beans:property name="prefix" value="/WEB-INF/pages/"/>
  <beans:property name="suffix" value=".jsp"/>
</beans:bean>
```

Listing 5.16 dispatcher-context.xml Configuration

5.2.4 security-context.xml

Αντίστοιχα με όσα έχουμε αναφέρει μέχρι τώρα, το security-context.xml, αναμενόμενα αποτελεί το configuration αρχείο του Spring Security module της εφαρμογής. Το Spring Security είναι υπεύθυνο για το authentication και το authorization των χρηστών της εφαρμογής.

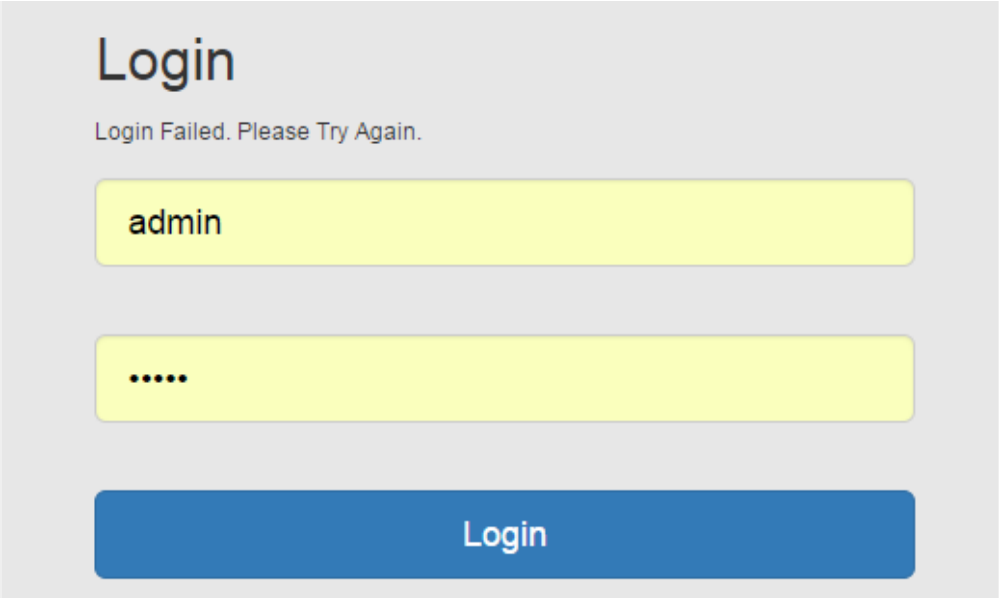
Ορίζουμε στο security-context.xml χρήστες, δηλώνοντας για τον καθένα το username και το password του. Κάνουμε χρήση ενός password encoder και συγκεκριμένα του **BCryptPasswordEncoder** που μας παρέχει out-of-the-box το framework, ώστε οι κωδικοί των χρηστών να αποθηκεύονται στο XML κρυπτογραφημένοι για λόγους τήρησης της ιδιωτικότητας. Στη συνέχεια, ορίζουμε ρόλους χρηστών (ROLE_ANONYMOUS, ROLE_USER κλπ) και αποδίδουμε στον κάθε χρήστη που έχουμε φτιάξει έναν ή περισσότερους από αυτούς. Έπειτα, δηλώνουμε σε ποία URLs θα έχει πρόσβαση κάθε ρόλος. Όπως φαίνεται και από το αρχείο που παρατίθεται παρακάτω, ορίζουμε ότι στη login θα έχει πρόσβαση οποιοσδήποτε, ώστε να μπορεί να πραγματοποιηθεί η σύνδεση των εγγεγραμμένων χρηστών. Όπως δηλώσαμε και στο web.xml, Spring Security φίλτρα θα παρεμβάλλονται σε κάθε HTTP αίτηση προς την εφαρμογή, εμφανίζοντας στο χρήστη τη σελίδα login, και επιτρέποντας τη μετάβαση στη σελίδα που πραγματικά ζητήθηκε, μόνο εάν ο χρήστης κάνει επιτυχώς login και μόνο εάν ο ρόλος του του παρέχει δικαιώματα σε αυτό το URL. Για παράδειγμα, ορίζουμε στο XML μέσω του element **intercept-url** ότι πρόσβαση στο URL /admin/jobLaunch έχει μόνο ο ρόλος ROLE_ADMIN μιας και η σελίδα αφορά την έκδοση, από τον χρήστη, εντολής εκκίνησης κάποιου Spring Batch job, και αυτή είναι μια κρίσιμη λειτουργία που δεν θέλουμε να επιτρέπεται σε οποιονδήποτε. Εάν ένας απλός χρήστης (ROLE_USER), ζητήσει πρόσβαση σε κάποιο URL στο οποίο δεν έχει δικαιώματα, τότε επιστρέφεται η denied.jsp.

Το γεγονός ότι οι χρήστες της εφαρμογής δημιουργούνται με προγραμματιστικό τρόπο στο security-context.xml, έχει σαφείς περιορισμούς: για κάθε νέο χρήστη πρέπει ο developer να πραγματοποιήσει προσθήκες στο XML αρχείο καθώς δεν υπάρχει λειτουργία sign up νέου χρήστη, παρά μόνο login. Η εφαρμογή θα μπορούσε να επεκταθεί ώστε να παρέχει αυτή τη λειτουργία ή/και να αντλεί τα αναγνωριστικά των χρηστών από βάση δεδομένων, όμως αυτή η επέκταση ξεφεύγει από τα όρια της παρούσας εργασίας.

```
<http pattern="/resources/**" security="none" />
<http auto-config="true" use-expressions="true" disable-url-rewriting="true">
  <form-login login-page="/login.html"
        authentication-failure-url="/login.html?failed=true"
        default-target-url="/home.html" />
  <logout logout-success-url="/login.html" />
  <access-denied-handler error-page="/denied.html" />
```

```
    <!-- Intercept URL Patterns -->
    <intercept-url pattern="/favicon.ico" access="ROLE_ANONYMOUS" />
    <intercept-url pattern="/login*" access="permitAll" />
    <intercept-url pattern="/admin/jobLaunch*" access="hasRole('ROLE_ADMIN')" />
    <intercept-url pattern="/admin/stopJobExecution*" access="hasRole('ROLE_ADMIN')" />
    <intercept-url pattern="/**" access="hasAnyRole('ROLE_ADMIN,ROLE_USER')" />
</http>

<authentication-manager>
    <authentication-provider>
        <password-encoder ref="passwordEncoder"/>
        <user-service>
            <user name="admin" password="$2a$10$mGKnse1eqjBKEuAs6kGq60OZ6S"   authorities="ROLE_ADMIN"/>
            <user name="viewer" password="$5a$20$nGFnse5I0L.Vt.MK1qs4kGq70OZ6S" authorities="ROLE_USER" />
        </user-service>
    </authentication-provider>
</authentication-manager>

<beans:bean id="passwordEncoder" class="org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder" />
```

Listing 5.17 Configuration του spring-security.xml



Σχήμα 5.1 Η σελίδα login.jsp μετά από αποτυχημένη προσπάθεια σύνδεσης

5.2.5 jobs-context.xml

Το jobs-context.xml είναι το κεντρικό configuration αρχείο του Spring Batch module. Περιλαμβάνει δηλώσεις των beans που αποτελούν την υποδομή του Spring Batch (Job Launcher, Job Registry, Job Explorer, Job Operator, Job Repository κ.α.), και περιγράφηκαν εκτενώς στο Κεφάλαιο 2. Δεν χρειάζεται λοιπόν κάτι παραπάνω εδώ πέρα από το να παραθέσουμε, για λόγους πληρότητας, τα εν λόγω configurations (Listing 5.18).

```
<batch:job-repository id="jobRepository" data-source="outputDataSource"
  transaction-manager="transactionManager" isolation-level-for-create="DEFAULT" />

<bean id="jobLauncher" class="org.springframework.batch.core.launch.support.SimpleJobLauncher">
  <property name="jobRepository" ref="jobRepository" />
  <property name="taskExecutor" ref="executor"/>
</bean>

<bean id="jobRegistry" class="org.springframework.batch.core.configuration.support.MapJobRegistry" />
<bean class="org.springframework.batch.core.configuration.support.JobRegistryBeanPostProcessor">
  <property name="jobRegistry" ref="jobRegistry" />
</bean>

<bean id="jobExplorer" class="org.springframework.batch.core.explore.support.JobExplorerFactoryBean">
  <property name="dataSource" ref="outputDataSource"/>
</bean>

<bean id="jobOperator" class="org.springframework.batch.core.launch.support.SimpleJobOperator">
  <property name="jobRepository" ref="jobRepository"/>
  <property name="jobExplorer" ref="jobExplorer"/>
  <property name="jobRegistry" ref="jobRegistry"/>
  <property name="jobLauncher" ref="jobLauncher"/>
</bean>
```

Listing 5.18 Configuration του Spring Batch Infrastructure

Ορίζεται επίσης εδώ ο task executor ο οποίος όπως έχουμε ήδη αναφέρει είναι υπεύθυνος για τη δημιουργία ενός pool από threads που jobs με βήματα σε παράλληλη επεξεργασία (<split/> ή partitioned) χρησιμοποιούν ώστε να δημιουργούν τα απαραίτητα threads. Ορίζεται πως κατά μέγιστο το pool αυτό μπορεί να έχει 50 threads και ο executor (όπως φαίνεται στην προηγούμενη Listing) δίνεται ως property κατά το initialization στον JobLauncher.

```
<task:executor id="executor" pool-size="50"/>
```

Στο jobs-context.xml δημιουργούμε επίσης ένα abstract parent job και ένα abstract parent step. Ορίζονται ως abstract καθώς δεν προορίζονται να χρησιμοποιηθούν αυτούσια, αλλά να κληρονομηθούν από όλα τα jobs και όλα τα chunk-oriented steps (read/process/write) αντίστοιχα. Το parent job ενσωματώνει σε όλα τα jobs έναν **InstanceStatusListener** για την αποστολή email σε περίπτωση που κάποιο job αποτύχει. Το parent step ενσωματώνει στο core step κάθε job έναν **StepStatusListener** ο οποίος ελέγχει αν διαβάστηκαν δεδομένα από τον Reader του βήματος και εάν υπάρχουν φιλτραρισμένα δεδομένα, μεταβάλλοντας κατάλληλα το ExitStatus. Τα beans των δυο παραπάνω listeners δηλώνονται επίσης στο jobs-context.xml. Για τις κλάσεις που υλοποιούν τους listeners μιλάμε σε επόμενη ενότητα.

```
<!-- Abstract Job And Step For Configuring Common Parameters And Listeners -->
<batch:job id="parentJob" abstract="true">
  <batch:listeners>
    <batch:listener ref="instanceStatusListener"/>
  </batch:listeners>
</batch:job>
```

```
<batch:step id="parentStep" abstract="true">
  <batch:tasklet transaction-manager="transactionManager">
    <batch:listeners>
      <batch:listener ref="stepStatusListener"/>
    </batch:listeners>
  </batch:tasklet>
</batch:step>
<!-- Listeners -->
<bean id="stepStatusListener" class="gr.booksAdmin.batch.listeners.StepStatusListener"/>
<bean id="instanceStatusListener" class="gr.booksAdmin.batch.listeners.InstanceStatusListener"/>
```

Listing 5.19 Abstract Job και Step, Spring Batch Listeners

Τέλος, για λόγους καλύτερης οργάνωσης, το jobs-context.xml αποτελεί το γονικό αρχείο όλων των batch XML configuration, στα οποία περιγράφονται τα batch jobs, επομένως πρέπει αυτά να γίνουν import σε αυτό. Κάθε ένα από τα παρακάτω αρχεία θα παρουσιαστεί εκτενώς στα επόμενα.

```
<!--Import Job Configuration Files-->
<import resource="batch/inventoryJob.xml"/>
<import resource="batch/multipleJobs.xml"/>
<import resource="batch/productsJob.xml"/>
<import resource="batch/authorsJob.xml"/>
```

5.2.6 JobParametersTasklet

Όπως έχει ήδη αναφερθεί, το JobParametersTasklet είναι αυτό που αναλαμβάνει να διαβάσει από βοηθητικό πίνακα της βάσης δεδομένων (**BATCH_JOB_INFO**) πληροφορίες σχετικά με την εξέλιξη (επιτυχή ή όχι) του προηγούμενου job execution με το ίδιο όνομα, και τις παραμέτρους που αυτό χρησιμοποίησε, ώστε να θέσει στο job execution context τις παραμέτρους που χρειάζονται τα επόμενα βήματα του τρέχοντος execution για να εκτελέσουν τις ενέργειές τους.

Ορίζονται δύο τύποι jobs (enumeration JobType): τα **FILE_INPUT** jobs και τα **DATABASE_INPUT** jobs. Τα πρώτα είναι κατά πολύ απλούστερα καθώς το μόνο που κάνουν είναι να φτιάχνουν ένα Map με τις μόνες τιμές του να είναι το **JOB_NAME**, το **JOB_TYPE** και το **JOB_INSTANCE_ID** του τρέχοντος Batch job, που θα χρησιμοποιηθεί για να κρατήσουμε σε κάθε εγγραφή της βάσης δεδομένων που τροποποιείται (INSERT ή UPDATE) πληροφορία σχετικά με το job που την τροποποίησε. Η πληροφορία αυτή θα χρησιμοποιηθεί όπως θα αναλυθεί σε επόμενη ενότητα για την παραγωγή κάποιων βασικών στατιστικών στοιχείων. Τα JOB_NAME, JOB_TYPE και JOB_INSTANCE_ID είναι επίσης μέλη ενός enumeration (JobParameter) που περιέχει τα ονόματα όλων των αποδεκτών παραμέτρων. Οι μέθοδοι που καλούνται σε αυτή την περίπτωση, όπως φαίνεται και στην Listing 5.20 είναι οι **setInventoryParameters** και **setCommonParameters**. Το αποτέλεσμα τους είναι να μπει στο job execution context η παράμετρος **PARAMETERS_MAP** με τιμή το Map που περιέχει όλα τα προηγούμενα.

Η περίπτωση των DATABASE_INPUT jobs είναι αρκετά πιο σύνθετη καθώς το Map που δημιουργείται για την PARAMETERS_MAP πρέπει να περιέχει τις πληροφορίες εκείνες που θα επιτρέψουν στο τρέχον job execution να συνεχίσει από εκεί που το προηγούμενο σταμάτησε. Η **setInputDatabaseParameters** καλεί και αυτή την **setCommonParameters** για να θέσει τα JOB_NAME, JOB_TYPE και JOB_INSTANCE_ID αλλά επιπρόσθετα εκτελεί με τη βοήθεια του **@Autowired BatchDao** τα queries εκείνα που θα καθορίσουν τις ημερομηνίες «από» και «έως» που χρειάζονται οι Batch item readers για κάθε οντότητα. Πιο αναλυτικά, σαν πρώτο βήμα ελέγχεται αν job με το ίδιο όνομα έχει ξανατρέξει ή όχι. Αν όχι θεωρείται πως εκτελείται αρχικό import και άρα οι τιμές «από» είναι η μέγιστη καταχωρημένη ημερομηνία στην output database για αυτή την οντότητα (σαν ένα δίχτυ ασφαλείας για πιθανό misconfiguration της εφαρμογής, καθώς στην περίπτωση του initial import η ημερομηνία αυτή θα είναι κανονικά η ελάχιστη ημερομηνία στην input database), και οι τιμές «έως» είναι η μέγιστη καταχωρημένη ημερομηνία στην input database. Στην περίπτωση που έχει ξανατρέξει job με το ίδιο όνομα (σενάριο δέλτα job), οι τιμές «έως» καθορίζονται με τον ίδιο ακριβώς τρόπο αλλά για τις τιμές «από» πρέπει να ελέγξουμε την πληροφορία που υπάρχει στον BATCH_JOB_INFO για το προηγούμενο execution. Αν το προηγούμενο execution ολοκληρώθηκε επιτυχώς το τρέχον θα πρέπει να συνεχίσει από εκεί που αυτό σταμάτησε και άρα οι τιμή «από» είναι η τιμή «έως» του προηγούμενου. Αν το προηγούμενο execution απέτυχε, το τρέχον πρέπει να ξεκινήσει από εκεί που αυτό προσπάθησε να ξεκινήσει και άρα η ημερομηνία «από» και των δυο είναι η ίδια.

Σε επίπεδο XML configuration το JobParametersTasklet το μόνο που χρειάζεται είναι ο καθορισμός των jobName και jobType. Το πρώτο γίνεται με late binding και χρησιμοποιεί τιμή που τίθεται στον JobLauncher κατά το αίτημα εκκίνησης ενός job ενώ το δεύτερο είναι το value που αντιστοιχεί στο code του enumeration JobType (databaseInput ή fileInput).

```
public class JobParametersTasklet implements Tasklet {

  private static final Logger logger = LoggerFactory.getLogger(JobParametersTasklet.class);

  @Autowired
  private BatchDao batchDao;

  private String jobName;
  private String jobType;
  private ExecutionContext executionContext;

  @Override
  public RepeatStatus execute(
      StepContribution stepContribution,
      ChunkContext chunkContext)
      throws Exception {

    JobExecution jobExecution =
      chunkContext
        .getStepContext()
```

```
            .getStepExecution()
            .getJobExecution();
    this.executionContext = jobExecution.getExecutionContext();

    /*
     * The Parameters Map May Be Already Created If A Nested Job Is Running
     * (i.e. 'authorsJob' Is Nested In 'multipleJob' When 'multipleJob'
     * Is Running) Or When The Requester Is The Web Controller Which
     * Generates The Map Based On The HttpRequest Parameters. In Both
     * Cases We Don't Want To Recreate The Parameters Map
     */
    Map<String, String> parametersMap =
        ProcessingUtils.convertStringToMap(
            jobExecution.getJobParameters().getString(JobParameter.PARAMETERS_MAP.getCode())
        );
    if(parametersMap.size() == 0) { //A Nested Job Should Not Recreate The ParametersMap
        logger.info("Creating Parameters Map [JobName={}, StepName={}].",
            new Object[]{jobName, chunkContext.getStepContext().getStepName()});
        //The Job Should Create Info Records
        executionContext.put("executeInfoTaskletFlag", true);
        JobType type = JobType.get(jobType);
        switch (type) {
            case DATABASE_INPUT:
                setInputDatabaseParameters(chunkContext);
                break;
            case FILE_INPUT:
                setInventoryParameters(chunkContext);
                break;
            default:
                logger.error("The Given Job Type [{}] Is not An Allowed One.", jobType);
                stepContribution.setExitStatus(new ExitStatus("PARAMETERS ERROR"));
        }
    } else {
        logger.info("Parameters Map Already Created. [Job Name {}, Step Name {}].",
            new Object[]{jobName, chunkContext.getStepContext().getStepName()});
        String executeInfoTasklet = executionContext.getString("executeInfoTaskletFlag");
        //In Any Other Case The JobInfoTasklet Should Not Be Executed
        this.executionContext.put("executeInfoTaskletFlag", executeInfoTasklet == null);
        //Add Parameters Map To The Context
        this.executionContext.put(JobParameter.PARAMETERS_MAP.getCode(), parametersMap);
    }
    return RepeatStatus.FINISHED;
}

/**
 * Add All Required Parameters Of A JobType.DATABASE_INPUT
 * Job To A Map And Inject It To The Execution Context
 * @param chunkContext ChunkContext
 */
private void setInputDatabaseParameters(ChunkContext chunkContext) {
    Map<String, String> parametersMap = new LinkedHashMap<String, String>();
    setCommonParameters(parametersMap,chunkContext);
    //Job Parameters Depend On The Previous Execution
    Map<String, String> previousExecutionsParameters = new LinkedHashMap<String, String>();
    List<Map<String, Object>> rows =
        batchDao.queryBatchJobInfo(
            "PREVIOUS_JOB_PARAMETERS",
```

```
        new MapSqlParameterSource("jobName", jobName)
    );
boolean completed = false;
if(rows.size()> 0) { //There Is A Previous Execution
    for(Map<String, Object> row : rows){
        previousExecutionsParameters =
            ProcessingUtils.convertStringToMap((String) row.get("PARAMETERS"));
        completed = "1".equals((String) row.get("COMPLETED"));
    }
    if(!completed){ //Start From Where The Last Execution Started
        parametersMap.put(
            JobParameter.AUTHOR_FROM.getCode(),
            previousExecutionsParameters.get(JobParameter.AUTHOR_FROM.getCode())
        );
        parametersMap.put(
            JobParameter.PRODUCT_FROM.getCode(),
            previousExecutionsParameters.get(JobParameter.PRODUCT_FROM.getCode())
        );
    } else { //Start From Where The Last Execution Stopped
        parametersMap.put(
            JobParameter.AUTHOR_FROM.getCode(),
            previousExecutionsParameters.get(JobParameter.AUTHOR_TO.getCode())
        );
        parametersMap.put(
            JobParameter.PRODUCT_FROM.getCode(),
            previousExecutionsParameters.get(JobParameter.PRODUCT_TO.getCode())
        );
    }
} else {
    //When A Differential Job Is Executed For The First
    //Time We Start From The Maximum Persisted Values
    parametersMap.put(
        JobParameter.AUTHOR_FROM.getCode(),
        batchDao.queryForMaximum(
            "MAXIMUM_PERSISTED_AUTHOR",
            new MapSqlParameterSource(), false
        )
    );
    parametersMap.put(
        JobParameter.PRODUCT_FROM.getCode(),
        batchDao.queryForMaximum(
            "MAXIMUM_PERSISTED_PRODUCT",
            new MapSqlParameterSource(), false
        )
    );
}
//Get The Maximum Author From Input DB
parametersMap.put(
    JobParameter.AUTHOR_TO.getCode(),
    batchDao.queryForMaximum(
        "MAXIMUM_AUTHOR_TO_UPDATE",
        new MapSqlParameterSource(), true
    )
);
parametersMap.put(
    JobParameter.PRODUCT_TO.getCode(),
    batchDao.queryForMaximum(
```

```
            "MAXIMUM_PRODUCT_TO_UPDATE",
            new MapSqlParameterSource(), true
          )
      );
      this.executionContext.put(JobParameter.PARAMETERS_MAP.getCode(), parametersMap);
    }

    /**
     * Add All Required Parameters Of A File Reading Job To A Map And
     * Inject It To The Execution Context. The Processed Files Is
     * The Parameters Map Of The Last Job Successfully Executed
     * (So That We Can Be Sure That The Mentioned Files Were Actually Processed)
     * @param chunkContext ChunkContext
     */
    private void setInventoryParameters(ChunkContext chunkContext) {
      Map<String, String> parameters = new LinkedHashMap<String, String>();
      setCommonParameters(parameters, chunkContext);
      this.executionContext.put(JobParameter.PARAMETERS_MAP.getCode(), parameters);
    }

    /**
     * Common Parameters Needed By All Job Types
     * @param parameters The Parameters Map
     * @param chunkContext ChunkContext Object
     */
    private void setCommonParameters(Map<String,String> parameters, ChunkContext chunkContext) {
      parameters.put(JobParameter.JOB_NAME.getCode(), jobName);
      parameters.put(JobParameter.JOB_TYPE.getCode(), jobType);
      //Parent Job's Instance Id For CHANGED_BY, CONSUMED_BY Flags
      parameters.put(
        JobParameter.JOB_INSTANCE_ID.getCode(),
        String.valueOf(
          chunkContext
            .getStepContext()
            .getStepExecution()
            .getJobExecution()
            .getJobInstance()
            .getId()
        )
      );
    }

    public void setJobName(String jobName) { this.jobName = jobName; }

    public void setJobType(String jobType) {  this.jobType = jobType; }
}
```

Listing 5.20 Υλοποίηση του JobParametersTasklet

5.2.7 BatchDao Interface και Implementation

Το interface και το implementation του BatchDao που είδαμε στα παραπάνω πως χρησιμοποιεί το JobParametersTasklet είναι αυτά που φαίνονται στις Listing 5.21 και 5.22 (στην δεύτερη δίνονται προς το παρόν μόνο οι μέθοδοι που το συγκεκριμένο tasklet χρησιμοποιεί). Η **queryBatchJobInfo** απλά εκτελεί ένα

query που δέχεται ως String παράμετρο μαζί με τις παραμέτρους που αυτό χρειάζεται. Η **queryForMaximum** πέρα από αυτές τις παραμέτρους δέχεται και μια boolean τιμή που καθορίζει το αν το ερώτημα για την maximum persisted ημερομηνία θα γίνει στην input ή στην output database (είδαμε πως ανά περίπτωση μπορεί να χρειαζόμαστε αυτή την τιμή είτε από την μία είτε από την άλλη βάση).

```
public interface BatchDao {

   List<Map<String, Object>> queryBatchJobInfo(
      String key, MapSqlParameterSource parameterSource
   );

   void insertOrUpdateBatchJobInfo(
      String key, MapSqlParameterSource mapSqlParameterSource
   );

   String queryForMaximum(
      String key, MapSqlParameterSource mapSqlParameterSource, boolean inputDatabase
   );
}
```

Listing 5.21 Interface του BatchDao

```
@Repository
public class BatchDaoImpl extends AbstractDao implements BatchDao {

   @Override
   public List<Map<String, Object>> queryBatchJobInfo(
         String key, MapSqlParameterSource parameters) {
      return outputJdbcTemplate.queryForList(getSQLQuery(key), parameters);
   }

   @Override
   public String queryForMaximum(
         String key,
         MapSqlParameterSource parameters,
         boolean inputDatabase) {
      SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
      String result = "0000-00-00 00:00:00";
      Timestamp timestampResult;
      try {
          timestampResult =
             inputDatabase ?
               inputJdbcTemplate.queryForObject(
                  getSQLQuery(key), parameters, Timestamp.class)
                  : outputJdbcTemplate.queryForObject(getSQLQuery(key), parameters , Timestamp.class);
         if(timestampResult!=null) {
            Date dateValue = new java.util.Date(timestampResult.getTime());
            result = dateFormat.format(dateValue);
         }
      } catch(EmptyResultDataAccessException exception) {  //Do Nothing
      }
      return result;
   }
}
```

Listing 5.22 Implementation του BatchDao

5.2.8 AbstractDao

Όλα τα DAO objects της εφαρμογής μας κάνουν extend το AbstractDao που παρουσιάζεται στην Listing 5.21 και επιτρέπει την ανάκτηση ενός προς εκτέλεση query με βάση το key του από ένα resource bundle αρχείο (resourceBundles/sql). Εφόσον γίνει το configuration στο application-context.xml του **ResourceBundle** με τον τρόπο που φαίνεται στην Listing 5.24 το μόνο που μένει στο AbstractDao είναι η κλήση της **getString** με όρισμα το key του query.

```
public abstract class AbstractDao {

   private final ResourceBundle SQL_BUNDLE = ResourceBundle.getBundle("resourceBundles/sql");

   @Autowired
   @Qualifier("outputJdbcTemplate")
   protected NamedParameterJdbcTemplate outputJdbcTemplate;

   @Autowired
   @Qualifier("inputJdbcTemplate")
   protected NamedParameterJdbcTemplate inputJdbcTemplate;

   @PostConstruct
   public void init() {
      Assert.notNull(inputJdbcTemplate, "inputJdbcTemplate must not be null.");
      Assert.notNull(outputJdbcTemplate, "outputJdbcTemplate must not be null.");
   }

   /**
    * Retrieves a sql query by it's key from corresponding properties file.
    * @param key The Key
    * @return The SQL Query
    */
   protected String getSQLQuery(String key){
      return SQL_BUNDLE.containsKey(key)? SQL_BUNDLE.getString(key) : null;
   }
}
```

Listing 5.23 AbstractDao

```
<bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
   <property name="defaultEncoding" value="UTF-8" />
</bean>
```

Listing 5.24 Configuration του ResourceBundle

5.2.9 JobInfoTasklet

Επόμενο βήμα στη ροή όλων των jobs είναι το JobInfoTasklet (βλ. Listing 5.25). Η λογική του είναι αρκετά απλούστερη από αυτή του προηγούμενου tasklet. Το JobInfoTasklet έχει ένα boolean flag (**initializeFlag**) που καθορίζει το αν εκτελείται κατά την εκκίνηση ή την ολοκλήρωση ενός job. Στην πρώτη περίπτωση αυτό

που χρειάζεται να γίνει είναι μια εγγραφή στον BATCH_JOB_INFO πίνακα για το τρέχον job ούτως ώστε η πληροφορία για τις παραμέτρους με τις οποίες έτρεξε να είναι διαθέσιμη όπως περιγράφηκε στην προηγούμενη ενότητα σε επόμενα executions. Μαζί με τις παραμέτρους (που μετατρέπονται σε ένα comma separated String με χρήση της **convertMapToString** και του Joiner της βιβλιοθήκης **Guava**[32] από την Google), στον ίδιο πίνακα κρατάμε το timestamp κατά το οποίο δημιουργήθηκε η εγγραφή και θέτουμε την στήλη COMPLETED ίση με 0. Το job δηλαδή ξεκίνησε αλλά δεν έχει ολοκληρωθεί. Το να γίνει η τιμή αυτή 1 (και άρα να θεωρηθεί το job ολοκληρωμένο), είναι δουλειά του ίδιου tasklet όταν εκτελείται σαν τελευταίο βήμα στη ροή ενός Batch job με το initializeFlag ίσο με false.

Το δεύτερο flag, με όνομα **executeTaskletFlag**, καθορίζει το αν το tasklet πρέπει να εκτελεστεί ή όχι. Επειδή ορίζουμε, εντός των άλλων, splits από jobs που τρέχουν παράλληλα άλλα jobs (π.χ. ένα γενικό job configuration που τρέχει παράλληλα τους Authors, τα Products και το Inventory με χρήση των <split/> και <flow/> elements), πρέπει να προσέξουμε ώστε το JobInfoTasklet να μην δημιουργήσει εγγραφές στον BATCH_JOB_INFO πίνακα περισσότερες από μια φορές (επειδή και το γονικό job και τα παιδιά αυτού χρησιμοποιούν το tasklet). Το tasklet λοιπόν θα εκτελέσει όσα περιγράφηκαν παραπάνω μόνο εφόσον το executeTaskletFlag είναι true. Το set του flag γίνεται από το JobParametersTasklet που όπως φαίνεται στην Listing 5.25 ελέγχει κάθε φορά που εκτελείται το αν στο execution context υπάρχει ήδη το PARAMETERS_MAP και αν το βρει θέτει το executeTaskletFlag ίσο με false (κάποιος άλλος έχει κάνει ήδη την αρχικοποίηση).

```
public class JobInfoTasklet implements Tasklet {

  private static final Logger logger = LoggerFactory.getLogger(JobInfoTasklet.class);

  @Autowired
  private BatchDao batchDao;

  @Autowired
  private DeltaDao deltaDao;

  private boolean executeTaskletFlag;
  private boolean initializeFlag;

  @Override
  public RepeatStatus execute(
      StepContribution stepContribution,
      ChunkContext chunkContext) throws Exception {
    StepExecution stepExecution = chunkContext.getStepContext().getStepExecution();
    JobExecution jobExecution = stepExecution.getJobExecution();
    ExecutionContext executionContext = jobExecution.getExecutionContext();
    Long instanceId = jobExecution.getJobInstance().getId();
    Long executionId = stepExecution.getJobExecutionId();
```

[32] Περισσότερα για την Guava εδώ: https://github.com/google/guava

```
    String jobName = chunkContext.getStepContext().getJobName();
    Map<String, String> parametersMap =
            (Map<String, String>) executionContext.get(JobParameter.PARAMETERS_MAP.getCode());
    String filesList = (String) executionContext.get(JobParameter.FILES_LIST.getCode());
    String requester = jobExecution.getJobParameters().getString(JobParameter.JOB_REQUESTER.getCode());

    if (BooleanUtils.isNotFalse(executeTaskletFlag)) {
      //A Nested Job Should Not Run The Info Tasklet (i.e. 'authorsJob'
      //Is Nested In 'multipleJob' When 'multipleJob' Is Running)
      if (BooleanUtils.isNotFalse(initializeFlag)) {
        //InputDatabase Jobs Set BATCH_JOB_INFO.PARAMETERS At Initialization
        MapSqlParameterSource parameters = prepareParameters(
          instanceId,
          executionId,
          jobName,
          initialize(parametersMap) ? ProcessingUtils.convertMapToString(parametersMap) : null, requester
        );
         batchDao.insertOrUpdateBatchJobInfo("CREATE_ENTRY", parameters);
      } else {
        if (filesList != null) { //Update Status And Parameters
          MapSqlParameterSource parameters =
           prepareParameters(instanceId, executionId, jobName, filesList, requester);
          batchDao.insertOrUpdateBatchJobInfo(
            "UPDATE_STATUS_AND_PARAMETERS", parameters
          );
        } else {//Update Status
          MapSqlParameterSource parameters =
            prepareParameters(instanceId, executionId, jobName, null, requester);
          batchDao.insertOrUpdateBatchJobInfo(
            "UPDATE_STATUS", parameters
          );
        }
        //BATCH_JOB_DELTA Record Only At Finalization
        calculateDelta(instanceId, executionId, jobName, parametersMap);
      }
    } else {
      logger.info("A Parent Job Already Initialized / Finalized The Info Records [JobName={}, StepName={}].",
          parametersMap.get(JobParameter.JOB_NAME.getCode()), chunkContext.getStepContext().getStepName());
    }
    return RepeatStatus.FINISHED;
}

/**
 * It's Critical For A JobType.DIGIBOOKS Job To Set The Parameters Map
 * During Initialization Since A Failed Job Execution Holds Info Data
 * That The Next Execution Should Read.
 * @param parametersMap The Parameters Map
 * @return boolean Indicating If The BATCH_JOB_INFO.PARAMETER Value Should Be Persisted
 */
private boolean initialize(Map<String, String> parametersMap) {
    if (parametersMap != null) {
      String type = parametersMap.get(JobParameter.JOB_TYPE.getCode());
      JobType jobType = JobType.get(type);
      return jobType.equals(JobType.DATABASE_INPUT);
    }
    return false;
}
```

```
  /**
   * Calculate The Given Job Execution's Delta And Create A Record In BATCH_JOB_DELTA Table
   * @param jobInstanceId Job's Instance Id
   * @param jobExecutionId Job's Execution Id
   * @param jobName Job's Name
   * @param parametersMap The Parameters Map
   */
  private void calculateDelta(
        Long jobInstanceId,
        Long jobExecutionId,
        String jobName,
        Map<String, String> parametersMap) {
    if (parametersMap != null) {
      String type = parametersMap.get(JobParameter.JOB_TYPE.getCode());
      JobType jobType = JobType.get(type);
      MapSqlParameterSource parameters =
        prepareParameters(jobInstanceId, jobExecutionId, jobName, null, null);
      switch (jobType) {
        case DATABASE_INPUT:
          Integer productsChanged = deltaDao.getDelta("PRODUCTS_CHANGED", jobInstanceId);
          Integer authorsChanged = deltaDao.getDelta("AUTHORS_CHANGED", jobInstanceId);
          //All Other Entities Skipped For Brevity
          parameters.addValue("deltaType", "changed");
          parameters.addValue("productsChanged", productsChanged);
          parameters.addValue("authorsChanged", authorsChanged);
          parameters.addValue("inventoryChanged", 0);
          deltaDao.updateDelta("UPDATE_DELTA", parameters);
          break;
        case FILE_INPUT:
          Integer inventoryChanged =
            deltaDao.getDelta("INVENTORY_CHANGED", jobInstanceId);
          parameters.addValue("deltaType", "changed");
          parameters.addValue("productsChanged", 0);
          parameters.addValue("authorsChanged", 0);
          parameters.addValue("inventoryChanged", inventoryChanged);
          deltaDao.updateDelta("UPDATE_DELTA", parameters);
          break;
        default:
      }
    }
  }

private MapSqlParameterSource prepareParameters(
        Long jobInstanceId,
        Long jobExecutionId,
        String jobName,
        String parameters,
        String requester) {
    MapSqlParameterSource result = new MapSqlParameterSource();
    result.addValue("jobInstanceId", jobInstanceId);
    result.addValue("jobExecutionId", jobExecutionId);
    result.addValue("jobName", jobName);
    result.addValue("parameters", parameters);
    result.addValue("requester", requester);
    result.addValue("changedOn", new Date());
    return result;
  }
```

```
    public void setExecuteTaskletFlag(boolean executeTaskletFlag) {
        this.executeTaskletFlag = executeTaskletFlag;
    }

    public void setInitializeFlag(boolean initializeFlag) {
        this.initializeFlag = initializeFlag;
    }
}
```

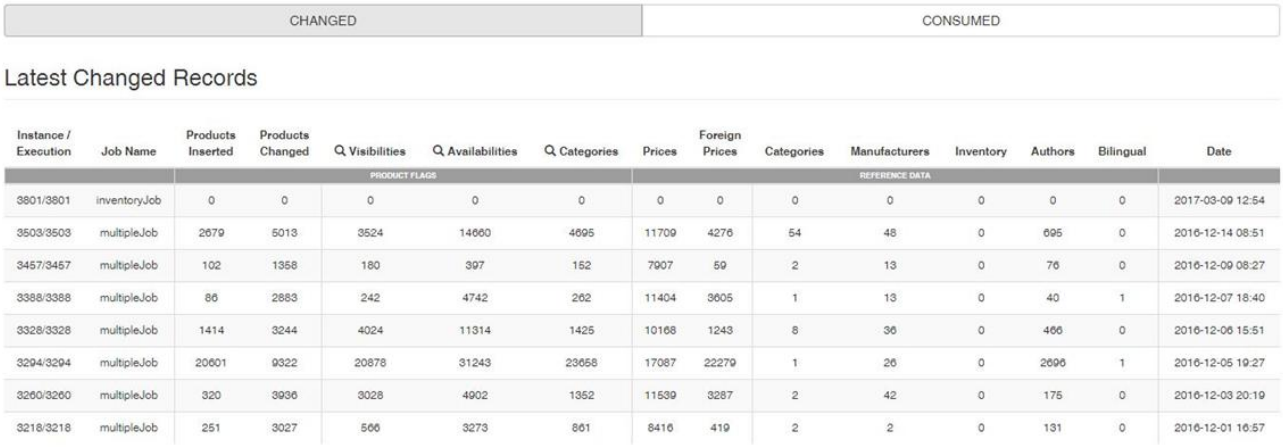
Listing 5.25 Υλοποίηση του JobInfoTasklet

Στο XML configuration λοιπόν (jobs-context.xml), ορίζουμε δυο tasklets που χρησιμοποιούν την παραπάνω κλάση: το **initializeInfoTasklet** (initializeFlag=true) και το **finalizeInfoTasklet** (initializeFlag=false) τα οποία πρέπει να χρησιμοποιούνται στην αρχή και το τέλος οποιουδήποτε job επιθυμεί να κρατήσει πληροφορία στον BATCH_JOB_INFO. Το ακριβές σημείο εντός της ακολουθίας των βημάτων στο οποίο πρέπει να γίνει αυτό έχει ήδη παρουσιαστεί στο ακολουθιακό διάγραμμα του Σχήματος 4.3. Βλέπουμε επίσης στο configuration των tasklets πως η τιμή για το executeInfoTaskletFlag γίνεται με late binding με βάση την τιμή που θα βρεθεί στο Job Execution Context κατά το initialization του step.

```
<bean id="initializeInfoTasklet" class="gr.booksAdmin.batch.tasklets.JobInfoTasklet" scope="step">
    <property name="initializeFlag" value="true"/>
    <property name="executeTaskletFlag" value="#{jobExecutionContext['executeInfoTaskletFlag']}"/>
</bean>
<bean id="finalizeInfoTasklet" class="gr.booksAdmin.batch.tasklets.JobInfoTasklet" scope="step">
    <property name="initializeFlag" value="false"/>
    <property name="executeTaskletFlag" value="#{jobExecutionContext['executeInfoTaskletFlag']}"/>
</bean>
```

Listing 5.26 Configuration των JobInfoTasklet

Κλείνουμε την ενότητα αυτή με μια αναφορά στα στατιστικά που παράγονται από το JobInfoTasklet κατά το finalization. Με δεδομένο πως κάθε εγγραφή που γίνεται INSERT ή UPDATE στη βάση δεδομένων κρατάει και το Instance ID του job που την τροποποίησε (ή την δημιούργησε), μπορούμε πριν ολοκληρώσουμε το job να υπολογίσουμε για κάθε οντότητα που μας ενδιαφέρει το σύνολο των εγγραφών που τροποποιήθηκαν (με απλά COUNT(*) ερωτήματα επί της στήλης **CHANGED_BY** που υπάρχει σε κάθε πίνακα της βάσης που σχετίζεται με κάποια οντότητα). Κρατώντας για κάθε execution τις τιμές αυτές στον βοηθητικό πίνακα **BATCH_JOB_DELTA** μπορούμε εύκολα να οδηγηθούμε σε ένα view σαν αυτό του Σχήματος 5.2 που μας δίνει εύκολα και γρήγορα πληροφορίες για το πλήθος των αλλαγών που το κάθε job πραγματοποίησε. Η υλοποίηση αυτού του view δεν χρήζει αναφοράς. Ούτε χρειάζεται να αναφερθούμε σε όλες τις άλλες οντότητες που εμφανίζονται στο view αυτό και δεν είναι γνωστές από τα όσα έχουν συζητηθεί ως τώρα. Με τον ίδιο ακριβώς τρόπο που εκτελούνται τα jobs για τα Products ή τους Authors και υπολογίζονται τα στατιστικά τους, εκτελούνται τα jobs και υπολογίζονται τα στατιστικά όλων των οντοτήτων που παρουσιάζονται στο Σχήμα 5.2.



CHANGED | CONSUMED

Latest Changed Records

Instance / Execution	Job Name	Products Inserted	Products Changed	Visibilities	Availabilities	Categories	Prices	Foreign Prices	Categories	Manufacturers	Inventory	Authors	Bilingual	Date
		PRODUCT FLAGS							REFERENCE DATA					
3801/3801	inventoryJob	0	0	0	0	0	0	0	0	0	0	0	0	2017-03-09 12:54
3503/3503	multipleJob	2679	5013	3524	14660	4695	11709	4276	54	48	0	695	0	2016-12-14 08:51
3457/3457	multipleJob	102	1358	180	397	152	7907	59	2	13	0	76	0	2016-12-09 08:27
3388/3388	multipleJob	86	2883	242	4742	262	11404	3605	1	13	0	40	1	2016-12-07 18:40
3328/3328	multipleJob	1414	3244	4024	11314	1425	10168	1243	8	36	0	466	0	2016-12-06 15:51
3294/3294	multipleJob	20601	9322	20878	31243	23658	17087	22279	1	26	0	2696	1	2016-12-05 19:27
3260/3260	multipleJob	320	3936	3028	4902	1352	11539	3287	2	42	0	175	0	2016-12-03 20:19
3218/3218	multipleJob	251	3027	566	3273	861	8416	419	2	2	0	131	0	2016-12-01 16:57

Σχήμα 5.2 Εγγραφές που το κάθε Job τροποποίησε ανά τύπο οντότητας

5.2.10 Listeners

Για τους δύο listeners που μόλις περιγράψαμε, δημιουργούμε δικές μας custom κλάσεις καθώς ενσωματώνουν business λογική. Ο **InstanceStatusListener** που ενσωματώνεται στο parentJob, υλοποιεί το interface **JobExecutionListener** του Spring Batch και καλείται από το framework με την ολοκλήρωση του κάθε job ελέγχοντας το ExitStatus του. Εάν το job δώσει **ExitStatus=FAILED**, τότε ο listener, χρησιμοποιώντας την @Autowired κλάση BatchMonitoringNotifier, εκκινεί τη διαδικασία αποστολής email στους διαχειριστές της εφαρμογής. Με τον BatchMonitoringNotifier και την υλοποίηση του μηχανισμού αποστολής email, ασχολούμαστε στην επόμενη ενότητα.

```
public class InstanceStatusListener implements JobExecutionListener {

   private static final Logger logger = LoggerFactory.getLogger(InstanceStatusListener.class);

   @Autowired
   private BatchMonitoringNotifier batchMonitoringNotifier;

   @Override
   public void beforeJob(JobExecution jobExecution) { //Do nothing
   }

   @Override
   public void afterJob(JobExecution jobExecution) {
      if(ExitStatus.FAILED.equals(jobExecution.getExitStatus())){
         try {
            batchMonitoringNotifier.notify(jobExecution);
         } catch (InterruptedException e) {
            logger.debug("Email Notification For Failed Job, Failed. Job Id: ", jobExecution.getJobId());
         }
      }
   }
}
```

Listing 5.27 Η κλάση InstanceStatusListener

Ο **StepStatusListener** χρησιμοποιείται στο abstract parentStep, το οποίο κάνουν extend τα chunk steps κάθε job, δηλαδή το authorsStep, το productsStep και το inventoryStep. Εφόσον πρόκειται για listener στο επίπεδο του βήματος, το Spring Batch ορίζει ότι θα πρέπει να υλοποιεί το **StepExecutionListener** interface. Ο StepStatusListener καλείται από το framework με το που ολοκληρωθεί κάποιο step για να ελέγξει το ExitStatus του. Αν το βήμα ολοκληρώθηκε επιτυχώς χωρίς ο ItemReader να έχει διαβάσει δεδομένα ή αν κάποιες εγγραφές φιλτραρίστηκαν επειδή δεν πληρούσαν τις προϋποθέσεις εγκυρότητας, ο StepStatusListener μεταβάλλει ανάλογα το ExitStatus, δίνοντας έξτρα πληροφορία σχετικά με την έκβαση του βήματος στο JobRepository και συνεπακόλουθα στη γραφική διεπαφή χρήστη.

```
public class StepStatusListener implements StepExecutionListener {

   @Override
   public void beforeStep(StepExecution stepExecution) { }//Do nothing

   @Override
   public ExitStatus afterStep(StepExecution stepExecution) {
      if(ExitStatus.COMPLETED.equals(stepExecution.getExitStatus()) && stepExecution.getReadCount() == 0){
         return new ExitStatus("NO MODIFIED DATA");
      } else if(ExitStatus.COMPLETED.equals(stepExecution.getExitStatus()) && stepExecution.getFilterCount() > 0){
         return new ExitStatus("FILTERED DATA");
      } else{
         return stepExecution.getExitStatus();
      }
   }
}
```

Listing 5.28 Η κλάση StepStatusListener

Πέρα από τους δύο παραπάνω listeners, που δηλώνονται στο jobs-context.xml και εφαρμόζονται σε όλα τα job flows, υπάρχει ένας ακόμα listener, ο **SkipListener**, ο οποίος αφορά μόνο το inventory job, και για αυτό αφήνεται να παρουσιαστεί στην αντίστοιχη ενότητα.

5.3 Ο μηχανισμός αποστολής email

Στο Κεφάλαιο 3 αναφέραμε πως μια από τις βασικές εμπορικές απαιτήσεις, είναι η δυνατότητα αποστολής email στους διαχειριστές της εφαρμογής αμέσως μόλις κάποιο job τερματίσει αποτυχημένα, ώστε να μπορούν να το επανεκκινήσουν εγκαίρως και να μην καθυστερήσει χρονικά η όλη διαδικασία.

Για το σκοπό αυτό, αρχικά χρειάζεται να δηλώσουμε και να παραμετροποιήσουμε μια κλάση **JavaMailSender** στο application-context.xml, δίνοντας της τα στοιχεία του email server μέσω του οποίου θα αποστέλλονται τα emails. Στα πλαίσια της εργασίας θα χρησιμοποιήσουμε τον SMTP email server που παρέχεται από την Google. Στο πραγματικό περιβάλλον θα χρησιμοποιηθούν emails servers της εταιρίας. Η παραμετροποίηση, γίνεται όπως βλέπουμε στην Listing 5.29, όπως και σε κάθε άλλη περίπτωση μέχρι τώρα

με χρήση property placeholders που αντικαθίστανται κατά το build από τις τιμές που υπάρχουν στο email.properties.

Για το email template, δηλαδή το στάνταρ φορμάτ του περιεχομένου του email, χρησιμοποιούμε ένα implementation του Velocity Engine, το VelocityEngineFactoryBean που μας παρέχει το Spring, και ορίζεται επίσης στο application-context.xml.

```
<!--Java Mail Sender-->
<bean id="javaMailSender" class="org.springframework.mail.javamail.JavaMailSenderImpl">
   <property name="host" value="${email.host}"/>
   <property name="port" value="${email.port}"/>
   <property name="username" value="${email.username}"/>
   <property name="password" value="${email.password}"/>
   <property name="javaMailProperties">
      <props>
         <prop key="mail.smtp.auth">${email.smtp.starttls.enable}</prop>
         <prop key="mail.smtp.starttls.enable">${email.smtp.auth}</prop>
      </props>
   </property>
</bean>
<!--Velocity Engine-->
<bean id="velocityEngine" class="org.springframework.ui.velocity.VelocityEngineFactoryBean">
   <property name="resourceLoaderPath" value="classpath:/velocity/"/>
</bean>
```

Listing 5.29 Configuration JavaMailSender και VelocityEngine

Παρακάτω παρατίθεται η κλάση EmailServiceImpl μαζί με το αντίστοιχο interface που υλοποιήθηκε για λόγους σχεδιαστικής καθαρότητας. Με χρήση των JavaMailSender και Velocity Engine, και εφόσον το flag **errorNotificationsOn** είναι true (ορίζεται σαν δυνατότητα απενεργοποίησης του όλου functionality και η τιμή του διαβάζεται επίσης από .properties αρχείο), η κλάση μας φορτώνει από τα application resources το **failedJob.vm** template αρχείο, χτίζει από αυτό το body του προς αποστολή email προσθέτοντας πληροφορίες σχετικά με το job που απέτυχε και τελικώς καλεί την **sendEmail** μέθοδο που αποστέλλει το email.

```
public interface EmailService {
    void sendJobFailureEmail(Long jobId, String jobName);
}
```

Listing 5.30 To EmailService Interface

```
public class EmailServiceImpl implements EmailService{

   private static final Logger logger = LoggerFactory.getLogger(EmailServiceImpl.class);

   private JavaMailSender javaMailSender;
   private VelocityEngine velocityEngine;

   private String emailTo;
   private String emailFrom;
```

```
    private String emailCc;
    private String applicationDomain;
    private boolean errorNotificationsOn;

    @Override
    public void sendJobFailureEmail(Long jobId, String jobName) {
        if(errorNotificationsOn) {
            Map<String, Object> model = new HashMap<String, Object>();
            model.put("jobId", jobId);
            model.put("jobName", jobName);
            model.put("jobUrl", applicationDomain + "/admin/jobExecutions/" + jobId + "?jobName=" + jobName);
            String emailText =
                    VelocityEngineUtils.mergeTemplateIntoString(velocityEngine, "failedJob.vm", "UTF-8", model);
            sendEmail("Cosmobooks Batch Job Failure", emailText);
            logger.debug("Mail For Failed Job With Name: {} and Id : {} Delivered.", new Object[]{jobName, jobId});
        } else {
            logger.warn("Email Error Notifications Are Turned Off.");
        }
    }

    private void sendEmail(String emailSubject, String emailText){
        try {
            MimeMessage mimeMessage = javaMailSender.createMimeMessage();
            MimeMessageHelper mimeMessageHelper = new MimeMessageHelper(mimeMessage, false);
            mimeMessageHelper.setFrom(emailFrom);
            mimeMessageHelper.setTo(emailTo);
            mimeMessageHelper.setCc(emailCc);
            mimeMessageHelper.setSubject(emailSubject);
            mimeMessageHelper.setText(emailText, true);
            javaMailSender.send(mimeMessage);
        } catch (Exception exception) {
            logger.warn("Exception While Trying To Send Email", exception);
        }
    }
    //Setters and Getters Skipped For Brevity
}
```

Listing 5.31 Υλοποίηση του EmailService Interface

Με αυτά, έχουμε ολοκληρώσει την υλοποίηση του κώδικα που αποστέλλει τα emails. Μένει η ενσωμάτωση του στον InstanceStatusListener, ώστε να καλείται όταν αποτύχει ένα job. Για το σκοπό αυτό δημιουργούμε την κλάση **BatchMonitoringNotifierImpl** μαζί με το αντίστοιχο interface. Η κλάση αυτή, γίνεται @Autowired στην κλάση του listener όπως είδαμε στα προηγούμενα, ενώ στον BatchMonitoringNotifier φέρνουμε με @Autowired το instance της EmailService κλάσης. Έτσι μόλις ένα job αποτύχει, το Spring Batch ενημερώνει τον listener, ο οποίος μέσω του Notifier καλεί την EmailService κλάση.

Είναι σημαντικό να παρατηρήσουμε το annotation **@Async** πάνω από την υλοποίηση της μεθόδου **notify** στην Listing 5.33. Η διαδικασία της αποστολής των email θα πρέπει να μην μπλοκάρει την υπόλοιπη εφαρμογή, γι' αυτό και εκτελείται ασύγχρονα.

```
public interface BatchMonitoringNotifier {
   void notify(JobExecution execution) throws InterruptedException;
}
```

Listing 5.32 Το BatchMonitoringNotifier Interface

```
@Component
public class BatchMonitoringNotifierImpl implements BatchMonitoringNotifier {

   private static final Logger logger = LoggerFactory.getLogger(BatchMonitoringNotifierImpl.class);

   @Autowired
   private EmailService emailService;

   @Override
   @Async
   public void notify(JobExecution jobExecution)  {
      String jobName = jobExecution.getJobInstance().getJobName();
       Long jobId = jobExecution.getJobId();
      emailService.sendJobFailureEmail(jobId, jobName);
   }
}
```

Listing 5.33 Υλοποίηση του BatchMonitoringNotifier Interface

5.4 Authors-job.xml

Το αρχείο authors-job.xml περιλαμβάνει όλο το configuration που αφορά το Authors Job (βλ. Listing 5.34). Όπως δείξαμε και στο αντίστοιχο ακολουθιακό διάγραμμα του Κεφαλαίου 4, το job αποτελείται κατά σειρά εκτέλεσης από τα βήματα: authorsParametersStep, authorsInitializeStep, authorsStep και authorsFinalizeStep. Τα authorsParametersStep, authorsInitializeStep και authorsFinalizeStep αποτελούν κοινά βήματα στη ροή όλων των jobs της εφαρμογής και χρησιμοποιούν τα JobParametersTasklet και JobInfoTasklet που αναλύθηκαν στις αντίστοιχες ενότητες. Το **authorsStep** είναι το κύριο βήμα του συγκεκριμένου job, εκτελεί το chunk processing με το μέγεθος του chunk να ορίζεται στα 1000 αντικείμενα και να χρησιμοποιεί τους ItemReader, ItemProcessor και ItemWriter που δηλώνονται στο XML configuration ως Spring beans και αναλύονται στις επόμενες παραγράφους.

```
<batch:job id="authorsJob" parent="parentJob">
   <!--Generic Parameters Step-->
   <batch:step id="authorsParametersStep">
      <batch:tasklet>
         <bean class="gr.booksAdmin.batch.tasklets.JobParametersTasklet" scope="step">
            <property name="jobName" value="#{jobParameters['jobName']}"/>
            <property name="jobType" value="inputDatabase"/>
         </bean>
      </batch:tasklet>
      <batch:fail on="PARAMETERS ERROR"/>
      <batch:next on="*" to="authorsInitializeStep"/>
   </batch:step>
   <batch:step id="authorsInitializeStep" next="authorsStep">
      <batch:tasklet ref="initializeInfoTasklet"/>
```

```
  </batch:step>
  <batch:step id="authorsStep" parent="parentStep" next="authorsFinalizeStep">
    <batch:tasklet>
      <batch:chunk reader="authorsReader"
          processor="authorsProcessor"
          writer="authorsWriter"
        commit-interval="1000" />
    </batch:tasklet>
  </batch:step>
  <batch:step id="authorsFinalizeStep">
    <batch:tasklet ref="finalizeInfoTasklet"/>
  </batch:step>
</batch:job>
```

Listing 5.34 Configuration του Authors Job

5.4.1 Authors Reader

Για το διάβασμα των authors από την input database, χρησιμοποιήσαμε τον out-of the-box παρεχόμενο JdbcPagingItemReader του Spring Batch, για τον οποίο μιλήσαμε εκτενώς στο Κεφάλαιο 2. Ο JdbcPagingItemReader χτίζει το SQL ερώτημα τμηματικά, μέσω της συμπλήρωσης των selectClause, fromClause και whereClause (βλ. Listing 5.35) και επιστρέφει τα δεδομένα ανά σελίδες των 1000 εγγραφών. Για την whereClause, λαμβάνονται δυναμικά από το JobExecutionContext οι παράμετροι authorFrom, authorTo ώστε να διαβάσουμε δεδομένα από εκεί που σταμάτησε το προηγούμενο authorsJob (ή από την αρχή εάν πρόκειται για το αρχικό import). Οι τιμές αυτές έχουν τεθεί στο context από το JobParametersTasklet μαζί με παρόμοιας λογικής τιμές για όλες τις οντότητες που απαιτούν το ίδιο functionality.

Αφού επιστραφούν τα δεδομένα, κάθε εγγραφή του result set πρέπει να μετασχηματιστεί σε Author domain object για περαιτέρω επεξεργασία από την εφαρμογή. Αυτή την αναλαμβάνει μια custom RowMapper κλάση, η AuthorsRowMapper, η οποία υλοποιεί το RowMapper interface με τον τρόπο που φαίνεται στην Listing 5.36.

```
<bean id="authorsReader" class="org.springframework.batch.item.database.JdbcPagingItemReader" scope="step">
  <property name="dataSource" ref="inputDataSource"/>
  <property name="queryProvider">
    <bean class="org.springframework.batch.item.database.support.SqlPagingQueryProviderFactoryBean">
      <property name="dataSource" ref="inputDataSource"/>
      <property name="selectClause">
        <value>
          SELECT authors.virtuemart_author_id, authors.LastMod, authors.created_on,
            authorsEl.au_name, authorsEl.au_desc, authorsEng.au_name, authorsEng.au_desc
        </value>
      </property>
      <property name="fromClause">
        <value>
          FROM ebooks_virtuemart_authors authors
```

```
            LEFT JOIN ebooks_virtuemart_authors_el_gr authorsEl
            ON authors.virtuemart_author_id = authorsEl.virtuemart_author_id
            LEFT JOIN ebooks_virtuemart_authors_en_gb authorsEng
            ON authors.virtuemart_author_id = authorsEng.virtuemart_author_id
          </value>
        </property>
        <property name="whereClause">
          <value>
            WHERE authors.LastMod &gt;= '#{jobExecutionContext['parametersMap']['authorFrom']}'
            AND authors.LastMod &lt;= '#{jobExecutionContext['parametersMap']['authorTo']}'
          </value>
        </property>
        <property name="sortKey" value="authors.virtuemart_author_id"/>
      </bean>
    </property>
    <property name="pageSize" value="1000"/>
    <property name="rowMapper" ref="authorsRowMapper"/>
</bean>
```

Listing 5.35 Configuration του Authors job στο authors-job.xml

```
public class AuthorsRowMapper implements RowMapper<Author> {
    @Override
    public Author mapRow(ResultSet resultSet, int i) throws SQLException {
        Author author = new Author();
        author.setId(resultSet.getInt("authors.virtuemart_author_id"));
     author.setCreatedOn(ProcessingUtils.convertTimestampToDate(resultSet.getTimestamp("authors.created_on")));
         author.setLastMod(ProcessingUtils.convertTimestampToDate(resultSet.getTimestamp("authors.LastMod")));
        author.setNameEll(ProcessingUtils.trimToNull(resultSet.getString("authorsEl.au_name")));
        author.setNameEng(ProcessingUtils.trimToNull(resultSet.getString("authorsEng.au_name")));
        author.setDescriptionEll(ProcessingUtils.trimToNull(resultSet.getString("authorsEl.au_desc")));
        author.setDescriptionEng(ProcessingUtils.trimToNull(resultSet.getString("authorsEng.au_desc")));
        return author;
    }
}
```

Listing 5.36 Η κλάση AuthorsRowMapper

5.4.2 Authors Processor

Για την επεξεργασία κάθε Author domain object υλοποιούμε μια custom AuthorsProcessor κλάση, η οποία υλοποιεί το ItemProcessor interface που παρέχει το Batch Batch. Ο processor δέχεται ένα αντικείμενο τη φορά και αν είναι valid το στέλνει στον ItemWriter (αφού συμπληρωθεί το καθορισμένο chunk size). Όπως ορίσαμε σε προηγούμενο Κεφάλαιο, ένα αντικείμενο Author είναι invalid εάν έχει κενά τόσο το αγγλικό όσο και το ελληνικό όνομα του συγγραφέα. Σε αυτή την περίπτωση ο AuthorsProcessor το φιλτράρει (επιστρέφει null). Αν είναι κενό μόνο το ένα από τα δύο ονόματα, τότε με κλήση της μεθόδου **validateAuthorNames**, θέτουμε και τα δύο ίσα με την τιμή που υπάρχει.

```
public class AuthorsProcessor implements ItemProcessor<Author, Author> {

    private static final Logger logger = LoggerFactory.getLogger(AuthorsProcessor.class);
```

```
    @Override
    public Author process(Author author) throws Exception {
        if (author.getNameEng() == null && author.getNameEll() == null){
            logger.debug("Author with Id: {} has Neither Greek Nor English Name. Shall Be Filtered Therefore",
                    new Object[]{author.getId()});
            return null; //Filter Author Objects That Have No Name Attributes All
        } else {
            validateAuthorNames(author);
            return author;
        }
    }

    //If One Of The Author Names Is Null, Set It Equal To The Other. Author Names Must Not Be Null In The Db.
    private void validateAuthorNames(Author author) {
        if (author.getNameEng() == null) {
            author.setNameEng(author.getNameEll());
        } else if(author.getNameEll() == null){
            author.setNameEll(author.getNameEng());
        }
    }
}
```

Listing 5.37 Η κλάση AuthorsProcessor

5.4.3 Authors Writer

Αφού ο Processor επεξεργαστεί ένα-ένα σειριακά τα items του chunk, η λίστα με όσα από τα items πέρασαν επιτυχώς τα validity checks, στέλνεται από το framework στον AuthorsWriter με σκοπό να εγγραφούν στην output βάση δεδομένων. Ο AuthorsWriter είναι υλοποίηση του ItemWriter interface του Spring Batch και όπως όλες οι κλάσεις μέχρι τώρα, δηλώνεται ως bean στο XML configuration της εφαρμογής. Επειδή ο item writer χρειάζεται πληροφορία για το instance id του τρέχοντος job ώστε να κρατά στη βάση το ποια εργασία εισήγαγε / τροποποίησε την κάθε εγγραφή, βλέπουμε στη Listing 5.38 πως σαν property του bean θέτουμε μέσω late binding (χρήση του step scope) αυτό ακριβώς το id που έχει θέσει για εμάς στο execution context το JobParametersTasklet.

```
<bean id="authorsWriter" class="gr.booksAdmin.batch.writers.AuthorsWriter" scope="step">
    <property name="jobInstanceId" value="#{jobExecutionContext['parametersMap']['jobInstanceId']}"/>
</bean>
```

Listing 5.38 Δήλωση του authorsWriter στο authors-job.xml

Για καθένα από τη λίστα των αντικειμένων που ο writer λαμβάνει ως είσοδο καλείται το AuthorsDaoImpl (βλ. Listing 5.40) που αναλαμβάνει να το εγγράψει στη βάση δεδομένων.

Η κλάση AuthorsDaoImpl είναι μια επέκταση της κλάσης AbstractDao που περιγράψαμε παραπάνω, προσαρμοσμένη στο Authors job. Λαμβάνει τα SQL ερωτήματα από το resource bundle αρχείο,

συμπληρώνει τις κατάλληλες παραμέτρους διαβάζοντας τα attributes του Author αντικειμένου, και με τη βοήθεια του Spring JDBC Template εκτελεί τα ερωτήματα στη βάση (INSERT ή UPDATE).

```
public class AuthorsWriter implements ItemWriter<Author> {

   @Autowired
   private AuthorsDao authorsDao;

   private Long jobInstanceId;

   @Override
   public void write(List<? extends Author> authors) throws Exception {
      for(Author author : authors){
         authorsDao.insertOrUpdateAuthor(author, jobInstanceId);
      }
   }

   public void setJobInstanceId(Long jobInstanceId) {
      this.jobInstanceId = jobInstanceId;
   }
}
```

Listing 5.39 Η κλάση AuthorsWriter

```
@Repository
public class AuthorsDaoImpl extends AbstractDao implements AuthorsDao {

   private static final Logger logger = LoggerFactory.getLogger(AuthorsDaoImpl.class);

   @Override
   public void insertOrUpdateAuthor(Author author, Long jobInstanceId) {
         logger.debug("Inserting/Updating Author With ID = {}.", author.getId());
         DatabaseUtils.insertOrUpdate(outputJdbcTemplate,
            getSQLQueries("AUTHOR_INSERT"), getSQLQueries("AUTHOR_UPDATE"),
            prepareSqlParameters(author, jobInstanceId)
         );
   }

   private MapSqlParameterSource prepareSqlParameters(Author author, Long jobInstanceId){
      MapSqlParameterSource parameters = new MapSqlParameterSource()
         .addValue("id", author.getId())
         .addValue("nameEll", author.getNameEll())
         .addValue("nameEng", author.getNameEng())
         .addValue("descriptionEll", author.getDescriptionEll())
         .addValue("descriptionEng", author.getDescriptionEng())
         .addValue("createdOn", author.getCreatedOn())
         .addValue("lastMod", author.getLastMod())
         .addValue("changed", 1)
         .addValue("changedBy", jobInstanceId)
         .addValue("consumed", 0)
         .addValue("consumedBy", null);
      return parameters;
   }
}
```

Listing 5.40 Υλοποίηση του AuthorsDao Interface

5.5 Products-job.xml

Το products-job.xml της Listing 5.41 αναμενόμενα περιέχει το configuration που αφορά το Products Job. Όπως και το Authors Job, το Products Job αποτελείται κατά σειρά εκτέλεσης από τα βήματα: productsParametersStep, productsInitializeStep, productsStep και productsFinalizeStep. Τα productsParametersStep, productsInitializeStep και productsFinalizeStep αποτελούν κοινά βήματα στη ροή όλων των jobs της εφαρμογής και χρησιμοποιούν τα JobParametersTasklet και JobInfoTasklet που αναλύθηκαν στις αντίστοιχες ενότητες. Το productsStep το οποίο εκτελεί το chunk processing με το μέγεθος του chunk να ορίζεται στα 1000 αντικείμενα και να χρησιμοποιεί τις ProductsReader, ProductsProcessor και ProductsWriter που θα παρουσιαστούν στα επόμενα. Οι ProductsReader και ProductsWriter, αν και αποτελούν ξεχωριστές υλοποιήσεις, έχουν εντελώς ανάλογη λειτουργικότητα με τις αντίστοιχες κλάσεις του Authors job και για αυτό η παρουσίαση τους εδώ παραλείπεται. Θα παρουσιάσουμε στην επόμενη παράγραφο τον ProductsProcessor, ο οποίος υλοποιεί τα απαραίτητα validations επί των Products.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:batch="http://www.springframework.org/schema/batch"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/batch http://www.springframework.org/schema/batch/spring-
batch.xsd">
  <batch:job id="productsJob" parent="parentJob">
    <batch:step id="productsParametersStep">
      <batch:tasklet>
        <bean class="gr.booksAdmin.batch.tasklets.JobParametersTasklet" scope="step">
          <property name="jobName" value="#{jobParameters['jobName']}"/>
          <property name="jobType" value="inputDatabase"/>
        </bean>
      </batch:tasklet>
      <batch:fail on="PARAMETERS ERROR"/>
      <batch:next on="*" to="productsInitializeStep"/>
    </batch:step>
    <batch:step id="productsInitializeStep" next="productsStep">
      <batch:tasklet ref="initializeInfoTasklet"/>
    </batch:step>
    <batch:step id="productsStep" parent="parentStep" next="productsFinalizeStep">
      <batch:tasklet>
        <batch:chunk
              reader="productsReader"
            processor="productsProcessor"
            writer="productsWriter"
            commit-interval="1000"/>
      </batch:tasklet>
    </batch:step>
    <batch:step id="productsFinalizeStep">
      <batch:tasklet ref="finalizeInfoTasklet"/>
    </batch:step>
  </batch:job>
```

Listing 5.41 Configuration του productsJob στο products-job.xml

5.5.1 ProductsProcessor

Η ProductsProcessor λαμβάνει ένα Product domain object, και ελέγχει εάν έχει συμπληρωμένο τόσο τον ελληνικό όσο και τον αγγλικό του τίτλο (δεν μας αρκεί να παρέχεται μόνο ο ένας από τους δύο) και εάν το είδος του (Kind) ανήκει στη λίστα με τα αποδεκτά είδη προϊόντων (παρουσιάστηκαν στο Κεφάλαιο 3). Αν το προϊόν πληροί και τις τρείς παραπάνω προϋποθέσεις, δίνεται στη μέθοδο **enrichProductAttributes** προς περαιτέρω επεξεργασία, διαφορετικά φιλτράρεται (επιστροφή null).

Η enrichProductAttributes, αντλεί από τη βάση δεδομένα για την κατηγορία, τον κατασκευαστή, τους συγγραφείς, το θέμα (αν πρόκειται για βιβλίο), την ηλικία (αν πρόκειται για παιχνίδι) και πολλά άλλα, με σκοπό να εμπλουτίσει με αυτές τα αντίστοιχα attributes του Product αντικειμένου που έχει δημιουργηθεί από τον item reader. Για να αντληθεί όλη απαιτούμενη πληροφορία ενός προϊόντος απαιτείται η εκτέλεση συνδυαστικών queries σε όλους τους πίνακες με τους οποίους σχετίζεται αυτό που στην input βάση είναι περισσότεροι από είκοσι. Το να εκτελεστούν paging ερωτήματα που θα διαβάζουν όλα αυτά τα δεδομένα με τη μία, κρίθηκε στην πράξη ιδιαίτερα αναποτελεσματικό: ο item reader χρειαζόταν πολλή περισσότερη ώρα για να διαβάσει chunks αντικειμένων από όση χρειάζονταν οι επόμενες φάσεις για να τα επεξεργαστούν και να τα γράψουν στην output βάση, με αποτέλεσμα να παρακωλύει την όλη διαδικασία. Κρίθηκε λοιπόν, και αποδείχτηκε και στην πράξη, πως ακολουθώντας λογική παρόμοια της driving query pattern ο χρόνος θα μπορούσε να βελτιωθεί εντυπωσιακά. Κατά τη φάση ανάγνωσης διαβάζονται οι βασικές πληροφορίες των προϊόντων (από τους ebooks_virtuemart_products, ebooks_virtuemart_products_el_gr και ebooks_virtuemart_products_en_gb), μετασχηματίζονται σε Product αντικείμενα και στέλνονται στον processor που εκτελώντας για κάθε προϊόν ένα UNION query σε όλους τους related πίνακες εμπλουτίζει το προϊόν με τις πληροφορίες που του λείπουν. Τα κοστοβόρα UNION queries για αντικείμενα σε chunks δηλαδή, μετατρέπονται σε πολλά μικρότερα και κατά πολύ γρηγορότερα UNION queries με ανάθεση μέρους της δουλειάς στην **enrichProductAttributes** του ItemProcessor.

```
public class ProductsProcessor implements ItemProcessor<Product, Product> {

  private static final Logger logger = LoggerFactory.getLogger(ProductsProcessor.class);

  @Autowired
  private ProductsDao productsDao;

  @Override
  public Product process(Product product) throws Exception {
    //A Product Should Always Have nameEll, nameEng And An Known Kind Id
    if(product.getNameEll() == null || product.getNameEng() == null
      || product.getKind().equals(0) || !knownKind(product.getKind(), product.getId())) {
      logger.info("Filtering Product With Id = {} [NameEll = {}, NameEng = {}, Kind = {}].",
        new Object[]{product.getId(), product.getNameEll(), product.getNameEng(), product.getKind()});
      return null; //Filter it
```

```
        } else {
            enrichProductAttributes(product);
            return product;
        }
    }

    private void enrichProductAttributes(Product product) {
        Map<Integer, String> attributesMap = productsDao.getProductAttributes(product.getId());
        Set<Map.Entry<Integer, String>> attributesSet = attributesMap.entrySet();
        for(Map.Entry<Integer, String> entry : attributesSet){
            String value = ProcessingUtils.trimToNull(entry.getValue());
            switch(entry.getKey()) {
                case -1:
                    product.setCategories(value);
                    break;
                case -2:
                    product.setAuthors(value);
                    break;
                case -3:
                    product.setManufacturers(value);
                    break;
                //Remaining Relationships Skipped For Brevity
            }
        }
    }

    private boolean knownKind(Integer kindId, Integer productId){
        boolean knownKind = MiscellaneousUtils.productKindsMap.get(kindId) != null;
        if(!knownKind) {
            logger.info("Product With ID = {} Has A Not Existing Kind With ID = {}.", productId, kindId);
        }
        return  knownKind;
    }
}
```

Listing 5.42 Η κλάση ProductsProcessor

5.6 Inventory-job.xml

Το configuration του Inventory Job δίνεται στην Listing 5.43. Η ροή του διαφέρει ελαφρώς από αυτή των jobs που διαβάζουν από βάση δεδομένων και γράφουν σε βάση δεδομένων. Κατά σειρά εκτελούνται τα inventoryConfigurationStep, inventoryParametersStep (το tasklet για τον καθορισμό των παραμέτρων του job που έχει ήδη παρουσιαστεί), inventoryInitializeStep (το κοινό για όλους τους τύπους jobs tasklet για initialize του BATCH_JOB_INFO), checkFileExistenceStep, inventoryStep (το chunk-oriented step), moveProcessedFilesStep και inventoryFinalizeStep (το κοινό για όλους τους τύπους jobs tasklet για finalize του BATCH_JOB_INFO). Αναφορά στα steps (συνεπακόλουθα και tasklets) που αφορούν τα jobs από αρχείο σε βάση, και δεν έχουν παρουσιαστεί στα μέχρι τώρα, κρίνεται απαραίτητο να γίνει στις ενότητες που ακολουθούν.

```
<batch:job id="inventoryJob" parent="parentJob" job-repository="jobRepository">
  <!-- Configuration Step-->
  <batch:step id="inventoryConfigurationStep">
    <batch:tasklet>
      <bean class="gr.booksAdmin.batch.tasklets.InventoryJobConfigurationTasklet">
        <property name="configurationKey" value="inventory.job"/>
      </bean>
    </batch:tasklet>
    <batch:fail on="CONFIGURATION ERROR"/>
    <batch:next on="*" to="inventoryParametersStep"/>
  </batch:step>
  <batch:step id="inventoryParametersStep">
    <batch:tasklet>
      <bean class="gr.booksAdmin.batch.tasklets.JobParametersTasklet" scope="step">
        <property name="jobName" value="#{jobParameters['jobName']}"/>
        <property name="jobType" value="inventory"/>
      </bean>
    </batch:tasklet>
    <batch:fail on="PARAMETERS ERROR"/>
    <batch:next on="*" to="inventoryInitializeStep"/>
  </batch:step>
  <batch:step id="inventoryInitializeStep" next="checkFileExistenceStep">
    <batch:tasklet ref="initializeInfoTasklet"/>
  </batch:step>
  <!-- File Existence Step-->
  <batch:step id="checkFileExistenceStep">
    <batch:tasklet>
      <bean class="gr.booksAdmin.batch.tasklets.FileExistenceTasklet" scope="step">
        <property name="filesPath" value="#{jobExecutionContext['inventoryFilePath']}"/>
        <property name="filePattern" value="#{jobExecutionContext['inventoryFilePattern']}"/>
      </bean>
    </batch:tasklet>
    <batch:fail on="FAILED"/>
    <batch:next on="NO FILES" to="inventoryFinalizeStep"/>
    <batch:next on="*" to="inventoryStep"/>
  </batch:step>
  <batch:step id="inventoryStep" parent="parentStep">
    <batch:tasklet>
      <batch:chunk
        reader="inventoryMultiResourcesItemReader"
        processor="inventoryProcessor"
        writer="inventoryWriter"
        skip-policy="exceptionSkipPolicy"
        commit-interval="100" />
      <batch:listeners>
        <batch:listener ref="skipListener"/>
      </batch:listeners>
    </batch:tasklet>
    <batch:fail on="FAILED"/>
    <batch:next on="*" to="moveProcessedFilesStep"/>
  </batch:step>
  <!-- Moved Processed Files To Processed Folder Step -->
  <batch:step id="moveProcessedFilesStep">
    <batch:tasklet>
      <bean class="gr.booksAdmin.batch.tasklets.FilesProcessedTasklet" scope="step">
        <property name="filesPath" value="#{jobExecutionContext['inventoryFilePath']}"/>
      </bean>
```

```
    </batch:tasklet>
     <batch:fail on="FILES ERROR"/>
    <batch:next on="*" to="inventoryFinalizeStep"/>
  </batch:step>
  <batch:step id="inventoryFinalizeStep">
    <batch:tasklet ref="finalizeInfoTasklet"/>
  </batch:step>
</batch:job>
```

Listing 5.43 Configuration του Inventory Job στο inventory-job.xml

5.6.1 inventoryJobConfigurationStep

Συνήθης πρακτική όταν μας ενδιαφέρει να συντηρούμε κάποια properties διαφορετικά ανά περιβάλλον ή όταν θέλουμε αυτά να μπορούν να αλλάζουν δυναμικά από διαχειριστές της εκάστοτε εφαρμογής είναι το να κρατούνται σε βάση δεδομένων. Αυτοί ακριβώς ήταν οι λόγοι που οδήγησαν στην ανάπτυξη του **InventoryJobConfigurationTasklet** (βλ. Listing 5.44). Το tasklet εκτελείται σαν πρώτο βήμα στη ροή του Inventory Job (ή όποιου άλλου job χρειάζεται να διαβάσει configuration από βάση), αναζητά σε ένα βοηθητικό πίνακα (**BATCH_JOB_CONFIGURATION**) τιμές με βάση το κλειδί **configurationKey** που δέχεται ως όρισμα κατά την αρχικοποίησή του, και τις βάζει στο execution context ώστε οποιοδήποτε βήμα ακολουθεί να μπορεί εφόσον τις χρειάζεται να τις χρησιμοποιήσει. Όπως φαίνεται και στον κώδικα, οι τιμές που το Inventory Job κάνει expose στο context είναι οι inventoryFilePattern , inventoryFilePath και inventoryProcessedFolder που δηλώνουν το pattern του ονόματος των προς ανάγνωση αρχείων, το path του file system στο οποίο αναμένεται να βρεθούν και το path στο οποίο με το που ολοκληρωθεί το job τα επεξεργασμένα αρχεία θα μεταφερθούν.

Η αναζήτηση των τιμών στη βάση γίνεται με τη βοήθεια του **ConfigurationDao** που δεν παρουσιάζεται εδώ καθώς δεν έχει κάτι άξιο αναφοράς. Το μόνο που πρέπει να σημειωθεί είναι πως καθώς το ίδιο configurationKey μπορεί να έχει περισσότερες από μια τιμές στη βάση, ο BATCH_JOB_CONFIGURATION θα πρέπει να κρατάει τις τιμές με κάποια καθορισμένη σειρά ώστε να είναι δυνατή η αντιστοίχιση των μελών τις επιστρεφόμενης λίστας σε συγκεκριμένα attributes. Επιπρόσθετα, στην περίπτωση που το κλειδί δεν επιστρέψει τιμές από τη βάση, το ExitStatus του tasklet τίθεται ίσο με “CONFIGURATION ERROR” ώστε να μπορεί το Spring Batch να τερματίσει τη ροή του job χωρίς να προχωρήσει στα επόμενα βήματα (element **<fail/>**).

```
@Repository
public class InventoryJobConfigurationTasklet implements Tasklet {

  private static final org.slf4j.Logger logger = LoggerFactory.getLogger(InventoryJobConfigurationTasklet.class);

  @Autowired
  private ConfigurationDao configurationDao;
```

```
  private String configurationKey;

  @Override
  public RepeatStatus execute(StepContribution stepContribution, ChunkContext chunkContext) throws Exception {
    ExecutionContext executionContext =
            chunkContext.getStepContext().getStepExecution().getJobExecution().getExecutionContext();
    try {
      configureInventoryJobValues(executionContext);
    } catch(NoConfigurationValueException e) {
      logger.error("Could Not Configure Inventory Job Values" , e);
      stepContribution.setExitStatus(new ExitStatus("CONFIGURATION ERROR"));
    }
    return RepeatStatus.FINISHED;
  }

  private void configureInventoryJobValues(ExecutionContext executionContext)
   throws NoConfigurationValueException {
    List<String> configurationValues = configurationDao.getConfigurationValues(configurationKey);
    if(configurationValues.size()<3){
      throw new NoConfigurationValueException();
    }
    executionContext.put("inventoryFilePattern",  configurationValues.get(0));
    executionContext.put("inventoryFilePath", configurationValues.get(1));
    executionContext.put("inventoryProcessedFolder", configurationValues.get(2));
  }

  public void setConfigurationKey(String configurationKey) {this.configurationKey = configurationKey;}

}
```

Listing 5.44 Η κλάση InventoryJobConfigurationTasklet

5.6.2 checkFileExistenceStep

Το δεύτερο νέο tasklet για την περίπτωση των jobs από αρχείο σε βάση είναι το FileExistenceTasklet της Listing 5.45. Στόχος του να ελέγχει αν στο προκαθορισμένο path υπάρχουν όντως αρχεία με το ζητούμενο filename pattern και αν ναι, να βάζει στο execution context την απαραίτητη πληροφορία ώστε αφενός να μπορούμε να κρατήσουμε στον BATCH_JOB_INFO σαν ιστορικό το όνομα των αρχείων που επεξεργάστηκαν και αφετέρου να καθορίσουμε το input του MultiResourceItemReader (που θα παρουσιαστεί παρακάτω), ώστε αυτός να μπορεί δυναμικά να επεξεργαστεί διαφορετικούς τύπους αρχείων ανάλογα με το τι θα βρεθεί κάθε φορά στον input folder.

Μια πρόσθετη απαίτηση που υλοποιείται με χρήση των μεθόδων **processFiles** και **unzipFile** είναι το να ελέγχεται αν τα αρχεία που θα βρεθούν στον input folder είναι συμπιεσμένα (ZIP αρχεία) και αν ναι να αποσυμπιέζονται (στο ίδιο path) και να τίθεται στο execution context λίστα με τα ονόματα των αποσυμπιεσμένων αρχείων και όχι του ZIP (που έτσι και αλλιώς είναι άχρηστο για τα επόμενα βήματα).

Το tasklet πέρα από τη διαχείριση εξαιρέσεων που θα οδηγήσουν σε ExitStatus σαν τα “UNZIP ERROR” και “JOB_ERROR” (που με τη σειρά τους θα οδηγήσουν σε αποτυχία του job), χειρίζεται και την περίπτωση του να μην βρεθεί κανένα αρχείο στο προκαθορισμένο directory, επιστρέφοντας το ExitStatus “NO FILES”. Όπως φαίνεται και στη Listing 5.45, αυτό το ExitStatus οδηγεί στο inventoryFinalizeStep χωρίς την εκτέλεση κάποιου άλλου από τα ενδιάμεσα βήματα. Δεν υπάρχει λόγος να εκτελεστεί κανένα από αυτά.

```
public class FileExistenceTasklet implements Tasklet {

  private static final Logger logger = LoggerFactory.getLogger(FileExistenceTasklet.class);

  private String filesPath;
  private String filePattern;

  private static final String FILE_PATTERN = "inventoryFilePattern";

  @Override
  public RepeatStatus execute(StepContribution stepContribution, ChunkContext chunkContext) throws Exception {
    ExecutionContext executionContext =
          chunkContext.getStepContext().getStepExecution().getJobExecution().getExecutionContext();
    List<String> filesList;
    try{
      filesList = processFiles(executionContext);
       executionContext.put(JobParameter.FILES_LIST.getCode(), ProcessingUtils.convertListToString(filesList));
       if(filesList.size()==0) {
         logger.info("No File With Pattern {} Found In {}, Job Is Not Continuing.",
            new Object[] { filePattern, filesPath });
         stepContribution.setExitStatus(new ExitStatus("NO FILES"));
       }
    } catch(FileUnzipException exception) {
      stepContribution.setExitStatus(new ExitStatus("UNZIP ERROR"));
    } catch(JobConfigurationException exception) {
      stepContribution.setExitStatus(new ExitStatus("JOB ERROR"));
    }
    return RepeatStatus.FINISHED;
  }

  private List<String> processFiles(ExecutionContext executionContext)
    throws FileUnzipException, JobConfigurationException {
    if(filesPath == null || filesPath.split(",").length>1){ //Sanity Check
      throw new JobConfigurationException(
        MessageFormat.format("A Job Reading Local File System Can Not Have " +
          "More Than One Path As Argument [filePaths = {0}]", filesPath));
    }
    List<String> fileNamesList = new ArrayList<String>();
    File fileDirectory = new File(filesPath);
    FileFilter fileFilter = new WildcardFileFilter(filePattern);
    File[] fileList = fileDirectory.listFiles(fileFilter);
    if(filePattern.toLowerCase().contains(".zip")) { //Extract .zip Contents
      for(File file : fileList) {
        List<String> unzippedFileFileNames = unzipFile(file, filesPath, filesPath);
        if (unzippedFileFileNames.size() == 0) {
          throw new FileUnzipException(
            MessageFormat.format("Tried To Unzip {0} But The File Is Empty.", file.getName()));
```

```
            } else {
               Iterator<String> iterator = unzippedFileFileNames.iterator();
               while (iterator.hasNext()) { //Copy Unzipped Files Names To Files Names List
                  fileNamesList.add(iterator.next());
               }
            }
            //Override File Pattern So That The Reading Step Can Read The Unzipped File
            executionContext.put(FILE_PATTERN, "*." + FilenameUtils.getExtension(unzippedFileFileNames.get(0)));
         }
      } else {
         for(File file : fileList) {
            logger.info("File {} Matching Pattern {} Found In Path {}",
                new Object[]{file.getName(), filePattern, filesPath});
            fileNamesList.add(file.getName());
         }
      }
      return fileNamesList;
   }

   private List<String> unzipFile(File file, String sourcePath, String destinationPath) throws FileUnzipException {
      logger.debug("Invoking unzipFile() For File {}, Source Path {}, Destination Path {}.",
         new Object[] { file.getName(), sourcePath, destinationPath });
      try{
         List<String> result = new ArrayList<String>();
         byte[] buffer = new byte[1024];
         ZipInputStream zipInputStream =
             new ZipInputStream(new FileInputStream(ProcessingUtils.prependFilePath(sourcePath, file.getName())));
          ZipEntry zipEntry = zipInputStream.getNextEntry();
         while(zipEntry != null){
            String fileName = zipEntry.getName();
            logger.debug("Decompressing File With Name {}.", fileName);
            File newFile = new File(ProcessingUtils.prependFilePath(destinationPath,fileName));
            FileOutputStream fileOutputStream = new FileOutputStream(newFile);
            int len;
            while((len = zipInputStream.read(buffer))>0){
               fileOutputStream.write(buffer, 0 ,len);
            }
            result.add(fileName);
            fileOutputStream.close();
            zipEntry = zipInputStream.getNextEntry();
         }
         zipInputStream.closeEntry();
         zipInputStream.close();
         return result;
      } catch(Exception exception){
         logger.error("Exception", exception);
         throw new FileUnzipException();
      }
   }

   public void setFilesPath(String filesPath) { this.filesPath = filesPath; }

   public void setFilePattern(String filePattern) { this.filePattern = filePattern; }

}
```

Listing 5.45 Η κλάση FileExistenceTasklet

5.6.3 moveProcessedFilesStep

Με την επιτυχή ολοκλήρωση ενός job που διαβάζει αρχεία, θέλουμε τα αρχεία αυτά να μεταφέρονται σε κάποιον φάκελο του file system ώστε να τηρείται ένα είδους ιστορικό. Αυτός είναι ο ρόλος του **FileProcessedTasklet** που δίνεται στην Listing 5.46. Το path του φακέλου για τα processed αρχεία καθορίζεται από τα configuration values που διάβασε το InventoryConfigurationTasklet και τα αρχεία εφόσον δεν έχουν τύπο ZIP μεταφέρονται σε αυτόν με χρήση της **Apache Commons FileUtils**, αφού πρώτα προστεθεί στο όνομά τους το String representation του timestamp κατά το οποίο έγινε η μεταφορά (ώστε να μπορούμε να χειριστούμε την περίπτωση αρχείων με το ίδιο όνομα σε διαφορετικές ημερομηνίες).

```
public class FilesProcessedTasklet implements Tasklet {

  private static final Logger logger = LoggerFactory.getLogger(FilesProcessedTasklet.class);

  private String filesPath; //The Path We Read For Inventory Files

  private static final String PROCESSED_FOLDER_NAME = "inventoryProcessedFolder";

  @Override
  public RepeatStatus execute(StepContribution stepContribution, ChunkContext chunkContext) throws Exception {

    ExecutionContext executionContext =
      chunkContext.getStepContext().getStepExecution().getJobExecution().getExecutionContext();
    File inputFile = new File(this.filesPath);
    String targetPath = ProcessingUtils.prependFilePath(
      filesPath, executionContext.getString(PROCESSED_FOLDER_NAME)
    );
    try {
      createFilePathNameIfAbsent(targetPath);
      File[] files = inputFile.listFiles();
      for(File file : files) {
        if(!file.isDirectory()){
          //The .zip Files Are Not Needed, Delete Them
          if(file.getName().endsWith(".zip")) {
            FileUtils.deleteQuietly(file);
            continue;
          }
          String fileName = getNewFileNameWithDateTime(file.getName());
          FileUtils.moveFile(file, FileUtils.getFile(ProcessingUtils.prependFilePath(targetPath,fileName)));
          logger.info("File {} Renamed To {} And Moved To Directory {}.",
            new Object[]{file.getName(), fileName, targetPath});
        }
      }
    } catch(Exception exception){
      logger.error("Exception", exception);
      stepContribution.setExitStatus(new ExitStatus("FILES ERROR"));
    }
    return RepeatStatus.FINISHED;
  }

  private void createFilePathNameIfAbsent(String targetPath) throws IOException {
```

```
        File targetDirectory = new File (targetPath);
        if(!targetDirectory.exists()){
            FileUtils.forceMkdir(targetDirectory);
        }
    }

    private String getNewFileNameWithDateTime(String fileName){
        DateFormat dateFormat = new SimpleDateFormat("yyyyMMddHHmmss");
        return dateFormat.format(Calendar.getInstance().getTime()) + "_" + fileName;
    }

    public void setFilesPath(String filesPath) { this.filesPath = filesPath; }

}
```

Listing 5.46 Η κλάση FileProcessedTasklet

5.6.4 InventoryMultiResourcesItemReader και InventoryFieldSetMapper

Για τον **MultiResourceItemReader** που χρησιμοποιείται για την ανάγνωση των Inventory αρχείων (CSV) δεν υπάρχει κάτι να συζητηθεί εδώ καθώς έχει αναλυθεί λεπτομερέστατα στο Κεφάλαιο 2. Το μόνο πρέπει να αναφερθεί είναι το ότι με configuration σαν και αυτό της Listing 5.47 μπορούμε να διαβάσουμε περισσότερα του ενός αρχεία στο path **resources**, που περιέχουν delimited εγγραφές με διαχωριστικό το String "!*", και field names τα "sapid,productId,stock". Οι εγγραφές θα μετασχηματιστούν σε Inventory αντικείμενα με χρήση του inventoryLineMapper (property του **FlatFileItemReader** στον οποίο κάνει delegate ο MultiResourceItemReader για την σειριακή ανάγνωση των αρχείων, το ένα μετά το άλλο).

```
<bean id="inventoryMultiResourcesItemReader"
class="org.springframework.batch.item.file.MultiResourceItemReader" scope="step">
    <property name="resources"
     value="file:#{jobExecutionContext['inventoryFilePath']}#{jobExecutionContext['inventoryFilePattern']}"/>
    <property name="delegate" ref="inventoryItemReader"/>
</bean>
<!-- Flat File Reader -->
<bean id="inventoryItemReader" class="org.springframework.batch.item.file.FlatFileItemReader">
    <property name="lineMapper" ref="inventoryLineMapper"/>
    <property name="recordSeparatorPolicy" ref="inventoryRecordSeperatorPolicy"/>
</bean>
<!-- Line Mapper -->
<bean id="inventoryLineMapper" class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
    <property name="fieldSetMapper" ref="inventoryFieldSetMapper"/>
    <property name="lineTokenizer" ref="inventoryLineTokenizer"/>
</bean>
<!-- Separator Policy -->
<bean id="inventoryRecordSeperatorPolicy"
class="org.springframework.batch.item.file.separator.SimpleRecordSeparatorPolicy"/>
<!-- Field Set Mapper -->
<bean id="inventoryFieldSetMapper" class="gr.booksAdmin.batch.mappers.InventoryFieldSetMapper"/>
<!-- Line Tokenizer -->
<bean id="inventoryLineTokenizer" class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer">
    <property name="delimiter" value="!*"/>
```

```
  <property name="names" value="sapId,productId,stock"/>
</bean>
```

Listing 5.47 Configuration του InventoryMultiResourceItemReader

Ιδιαίτερη αναφορά δεν χρειάζεται ούτε στην υλοποίηση του FieldSetMapper (**InventoryFieldSetMapper** της Listing 5.48). Ο mapper δέχεται ένα FieldSet σαν είσοδο και απλά δημιουργεί αντικείμενα Inventory με βάση τα fields **productId** και **stock** (το **sapid** αγνοήθηκε στην υλοποίησή μας). Τα αντικείμενα επιστρέφονται, και επόμενος στη σειρά είναι ο InventoryProcessor που θα αναλάβει το validation και θα αποφασίσει αν πρέπει να προωθηθούν στον item writer ή όχι. Ο InventoryProcessor παρουσιάζεται στην επόμενη ενότητα.

```
public class InventoryFieldSetMapper implements FieldSetMapper<Inventory> {

  private static final Logger logger = LoggerFactory.getLogger(InventoryFieldSetMapper.class);

  @Override
  public Inventory mapFieldSet(FieldSet fieldSet) throws BindException {
    Inventory inventory = new Inventory();
    try {
      inventory.setId(Integer.parseInt(ProcessingUtils.trimToNull( (fieldSet.readString("productId"))));
      inventory.setStockQuantity(Integer.parseInt(ProcessingUtils.trimToNull(fieldSet.readString("stock"))));
    } catch (NumberFormatException exception){
      logger.info("Not Valid Inventory FieldSet [PRODUCT_ID = {}, STOCK = {}]",
          new Object[] { fieldSet.readString("productId"), fieldSet.readString("stock") });
    }
    return inventory ;
  }
}
```

Listing 5.48 Η κλάση InventoryFieldSetMapper

5.6.5 InventoryProcessor

Ο InventoryProcessor, πέρα από το να ελέγχει το ότι το αντικείμενο που επεξεργάζεται έχει valid τιμή για το απόθεμα (θετικό ακέραιο), εκτελεί και μια πρόσθετη εμπορική απαίτηση: επειδή το input στο inventory job είναι η πλήρης λίστα ειδών όπως αυτή εξάγεται από το ERP της εταιρίας, και επειδή δεν θέλουμε να χαρακτηρίζουμε ως δέλτα σε καθημερινή βάση το σύνολο αυτών, πρέπει τα Inventory domain objects να στέλνονται στον item writer μόνο εφόσον το απόθεμά τους δεν έχει τροποποιηθεί σε σχέση με αυτό που υπάρχει ήδη στη βάση δεδομένων. Αυτό ακριβώς είναι που εξασφαλίζει η μέθοδος **sameStock** της Listing 5.49. Με αυτή, κλείνει η παρουσίαση του Inventory Job, καθώς όλη η υπόλοιπη υλοποίηση που το αφορά είναι ακριβώς ίδια με αυτή των jobs από βάση σε βάση και άρα δεν υπάρχει λόγος να επαναληφθεί εδώ.

```
public class InventoryProcessor implements ItemProcessor<Inventory, Inventory>{

  @Autowired
  private InventoryDao inventoryDaoImpl;
```

```
  @Override
  public Inventory process(Inventory inventory) throws Exception {
    //Filter Inventories that have same stock as existing
    return (!validInventory(inventory) || sameStock(inventory)) ? null : inventory;
  }

  private boolean sameStock(Inventory inventory) {
    Integer existingStock;
    try {
      existingStock = inventoryDaoImpl.getExistingStock(inventory.getId());
      inventory.setExists(true);  //For The Writer
    } catch (EmptyResultDataAccessException exception){
      existingStock = null;
    }
    return existingStock != null && existingStock.equals(inventory.getStockQuantity());
  }

  private boolean validInventory(Inventory inventory){
    return (inventory.getId() != null && inventory.getStockQuantity() != null && inventory.getStockQuantity() >0);
  }
}
```

Listing 5.49 Η κλάση InventoryProcessor

5.7 JobsLauncher και tasks-context.xml

Ο πρώτος από τους δύο τρόπους που υλοποιήθηκε ώστε να μπορούμε να εκκινήσουμε Spring Batch jobs είναι ο BatchJobsLauncher της Listing 5.50. Ορίζουμε τη μέθοδο launch που, αφού δημιουργήσει το JobParameters αντικείμενο με τα JOB_NAME, JOB_REQUESTER ("scheduler" ή "controller" που είναι ο δεύτερος τρόπος για εκκίνηση που θα παρουσιαστεί στη συνέχεια) και LAUNCH_DATE, ξεκινά το Job με τη βοήθεια του Spring Batch JobLauncher.

Το αντικείμενο Job ορίζεται σαν property του BatchJobsLauncher και αναλαμβάνει να το θέσει ένα Spring Batch **scheduled-task** σαν και αυτό που δίνεται ως παράδειγμα στην Listing 5.51. Το task ορίζει μέσω ενός cron expression πως θα καλεί την μέθοδο launch του BatchJobsLauncher κάθε μέρα στις 15:00 και θέτει το property job αναφερόμενο στο bean authorsJob που έχει οριστεί στο authors-job.xml.

```
public class BatchJobsLauncher {

  private Job job;
  private JobLauncher jobLauncher;

  // Build The Job's Parameters And Launch It
  public void launch() throws Exception {
    jobLauncher.run(job, createParameters(job));
  }

  private JobParameters createParameters(Job job){
    JobParametersBuilder parametersBuilder = new JobParametersBuilder();
```

```
        parametersBuilder.addDate(JobParameter.LAUNCH_DATE.getCode(), new Date());
        parametersBuilder.addString(JobParameter.JOB_NAME.getCode(), job.getName());
        parametersBuilder.addString(JobParameter.JOB_REQUESTER.getCode(), JobRequester.SCHEDULER.getCode());
        return parametersBuilder.toJobParameters();
    }

    public void setJob(Job job) {
      this.job = job;
    }

    public void setJobLauncher(JobLauncher jobLauncher) {
      this.jobLauncher = jobLauncher;
    }
}
```

Listing 5.50 Η κλάση BatchJobsLauncher

```
<task:scheduler id="scheduler" pool-size="10"/>
<!-- An Example Scheduled Task -->
<bean id="authorsTask" class="gr.booksAdmin.task.BatchJobsLauncher">
  <property name="job" ref="authorsJob"/>
  <property name="jobLauncher" ref="jobLauncher"/>
</bean>
<task:scheduled-tasks scheduler="scheduler">
  <task:scheduled ref="authorsTask" method="launch" cron="0 0 15 * * *" />
</task:scheduled-tasks>
```

Listing 5.51 Configuration του tasks-context.xml

5.8 Partitioning

Στην Ενότητα 2.10.3.1 είδαμε πως με την χρήση του **MultiResourcePartitioner** που παρέχεται από το Spring Batch μπορούμε να χωρίσουμε ένα job ανάγνωσης από πολλά αρχεία σε partitions ώστε καθένα από αυτά (τα partitions ή αρχεία, έννοια ισοδύναμη σε αυτή την περίπτωση) να επεξεργαστεί από διαφορετικό thread, επιταχύνοντας συνεπακόλουθα την όλη διαδικασία. Εδώ, για να παρουσιάσουμε με περισσότερες λεπτομέρειες τα όσα το partitioning μπορεί να προσφέρει, θα δούμε το πώς μπορούμε να χωρίσουμε σε partitions ένα job ανάγνωσης από βάση δεδομένων. Για την περίπτωση αυτή δεν παρέχεται κάποια έτοιμη υλοποίηση και άρα χρειάζεται να γραφτεί custom λογική με χρήση όπως πάντα interfaces που το framework παρέχει για αυτό το σκοπό.

5.8.1 ColumnRangePartitioner

Το μόνο που έχουμε να κάνουμε είναι να υλοποιήσουμε την **partition** μέθοδο του **Partitioner** interface η οποία, όπως φαίνεται στη Listing 5.52, δέχεται ως παράμετρο το **gridSize** που καθορίζει τον αριθμό των partitions (threads) στα οποία θα χωριστεί η διαδικασία. Η partition, δεδομένου του gridSize, αλλά και των παραμέτρων **table** και **column** στις οποίες τίθεται το όνομα του πίνακα που μας ενδιαφέρει να χωρίσουμε

σε partitions και το όνομα της στήλης στην οποία θα βασιστούμε ώστε να γίνει αυτό, πρέπει να χωρίσει τις εγγραφές του input πίνακα σε ίσα μέρη και να δημιουργήσει ένα key-value Map από **ExecutionContext** αντικείμενα που θα περιέχουν οποιαδήποτε τιμή χρειαζόμαστε για το configuration του partitioned reader μας.

Για να γίνουν κατανοητά τα παραπάνω, έστω για παράδειγμα πως μας ενδιαφέρει να χωρίσουμε σε partitions τον πίνακα ebooks_virtuemart_authors με βάση την PK στήλη του virtuemart_author_id. Η μέθοδος partition βλέπουμε στην Listing 5.52 πως ξεκινά υπολογίζοντας τις **min**, **max** και **size** τιμές που είναι αντίστοιχα η μικρότερη τιμή που θα βρεθεί στην PK στήλη την ώρα που θα τρέξει ο Partitioner, η μέγιστη τιμή που θα βρεθεί στην PK στήλη, και το μέγεθος που υπολογίζεται για το κάθε partition με βάση τις min, max και gridSize (διαίρεση του συνολικού εύρους με το gridSize και FLOOR ώστε να πάρουμε ακέραιο αποτέλεσμα). Έχοντας αυτές τις τιμές, ο Partitioner μπαίνει σε ένα while loop και υπολογίζει τις **minValue** και **maxValue** που καθορίζουν την τιμή από και έως με βάση το column του κάθε partition. Πρέπει να σημειωθεί εδώ, πως αν το PK ξεκινά από την τιμή 1 (min) και τελειώνει στην τιμή 10.000 (max), και θέλουμε να χωρίσουμε τον πίνακα σε 5 κομμάτια, το size δεν θα είναι απαραίτητα 2.000. Δεν υπάρχει κάτι που να μας εξασφαλίζει πως οι τιμές του PK είναι auto increment ή πως δεν θα λείπουν κάποιες (επειδή π.χ. έχουν διαγραφεί). Θα μπορούσε το size να είναι μόνο 1.000 (επειδή ο πίνακας εντός αυτού του range έχει στην πραγματικότητα μόνο 5.000 εγγραφές) και για αυτό χρειάζεται κατά τον υπολογισμό των minValue και maxValue να είμαστε ιδιαίτερα προσεκτικοί ώστε να χειριστούμε σωστά μια τέτοια περίπτωση. Θέλουμε τα μέρη στα οποία θα χωρίσουμε τον πίνακα να είναι κατά το δυνατόν ίσα (εξαίρεση θα αποτελεί μόνο το τελευταίο που λόγω της FLOOR μπορεί να έχει περισσότερες εγγραφές, προφανώς λιγότερες του size) ώστε τα threads που θα αναλάβουν την επεξεργασία τους κατά το execution του job να είναι ισόποσα απασχολημένα.

Έχοντας τις τιμές minValue και maxValue για το κάθε partition το μόνο που έχουμε να κάνουμε είναι να τις βάλουμε στα ExecutionContext αντικείμενα που θα επιστραφούν ως αποτέλεσμα από την partition. Η μέθοδος χτίζει ένα key-value Map με το key να είναι ένα String με το όνομα που δίνουμε στο partition (Partition0, Partition1, Partition2 κ.ο.κ.) και value το ExecutionContext που περιέχει τις minValue και maxValue και το επιστρέφει ως αποτέλεσμα. Το πώς αυτό το αποτέλεσμα μπορούμε να το χρησιμοποιήσουμε φαίνεται στο configuration του job που δίνεται στην ακόλουθη ενότητα.

```
public class ColumnRangePartitioner implements Partitioner {

  private JdbcOperations jdbcTemplate;
  private String table; //The Name Of The SQL Table The Data Are In
  private String column; //The Name Of The Column To Partition

  @Override
  public Map<String, ExecutionContext> partition(int gridSize) {
```

```
    Map<String, ExecutionContext> result = new HashMap<String, ExecutionContext>();
    int min = jdbcTemplate.queryForObject(
      "SELECT COALESCE(MIN(" + column + "), 0) FROM " + table,  Integer.class
    );
    int max = jdbcTemplate.queryForObject(
      "SELECT COALESCE(MAX(" + column + "), 0) FROM " + table, Integer.class
    );
    int size = jdbcTemplate.queryForObject(
      "SELECT FLOOR(count(*) / " + gridSize + ") " +
      "FROM " + table + " " +
      "WHERE " + column +" >= " + min + " " +
      "AND " + column + " <= " + max,
      Integer.class
    );
    int start = min, end = min, upper = 0, number = 0;
     if(min!=0 || max!=0) { //Table Is Not Empty
       do {
          ExecutionContext value = new ExecutionContext();
          result.put("Partition" + number, value);
          end = jdbcTemplate.queryForObject(
            "SELECT MAX(T." + column + ") " +
            "FROM (" +
              "SELECT " + column + " " +
              "FROM " + table + " " +
              "WHERE " + column + " >= " + start + " " +
              "ORDER BY " + column + " ASC  " +
              "LIMIT " + (size > 0 ? size : 1) +
            ") T",
            Integer.class
          );
          upper = number==gridSize-1 ? max : end;
          value.put("minValue", start);
          value.put("maxValue", upper);
          start = end + 1;
          number++;
       } while(upper!=max);
     }
     return result;
  }
  //Setters and Getters Skipped For Brevity
}
```

Listing 5.52 Υλοποίηση του ColumnRangePartitioner

5.8.2 Configuration Job για χρήση του ColumnRangePartitioner

Το configuration του partitioned job φαίνεται, παίρνοντας σαν παράδειγμα το initial import των Authors, στην Listing 5.53. Το μόνο ιδιαίτερο που έχει ο **partitionedAuthorsReader** (JdbcPagingItemReader) είναι το ότι στο **whereClause** με late binding χρησιμοποιεί τις τιμές minValue και maxValue που έχουμε υπολογίσει στα προηγούμενα και τις οποίες βρίσκει στο Step Context. Το job ορίζει πως θα χωρίσει τον πίνακα σε 5 κομμάτια (**grid-size** property) και πως θα χρησιμοποιεί τον authorsPartitioner που είναι απλά ένα bean το οποίο χρησιμοποιεί την custom ColumnRangePartitioner και θέτει τις τιμές **jdbcTemplate**, **table** και **column**.

Το framework, βλέποντας πως στο step επίπεδο χρησιμοποιείται ένας partitioner (**partition** element), κατά την εκκίνηση του job θα εκτελέσει την custom υλοποίησή μας ώστε να βάλει στο Step Context την απαιτούμενη πληροφορία (minValue, maxValue) και αυτόματα θα χωρίσει την εργασία σε partitions που θα ανατεθούν, με χρήση του task-executor, σε τόσα threads όσα το grid-size.

Το μόνο που μένει να αναφερθεί είναι πως στο configuration του chunk του βήματός μας χαμηλώνουμε το isolation level των transactions σε READ_COMMITTED και ορίζουμε πως το DeadlockLoserDataAccessException θα είναι ένα retryable-exception με retry-limit ίσο με 10. Αυτό έγινε γιατί είδαμε πως αυξάνοντας την τιμή του grid-size σε τιμές μεγαλύτερες του 5 (που δίνουμε εδώ ως παράδειγμα), ο συνεπακόλουθος μεγαλύτερος όγκος ερωτημάτων εγγραφής / ενημέρωσης οδηγούσε σε κάποια deadlocks που όμως αντιμετωπίζονταν με αυτό τον τρόπο. Το ότι τα partitioned jobs χρησιμοποιήθηκαν μόνο για το αρχικό import της βάσης δεδομένων, μας επέτρεπε να χαμηλώσουμε το isolation level στην μικρότερη δυνατή τιμή, αφού το αρχικό import έγινε σε off-working χρονικό παράθυρο που ο μόνος που αλληλεπιδρούσε με τη βάση δεδομένων ήταν η Spring Batch εφαρμογή.

```
<batch:job id="partitionedAuthorsJob" parent="parentJob">
  <batch:step id="authorsPartitionedStep">
    <batch:partition step="authorsPartitioned" partitioner="authorsPartitioner">
      <batch:handler grid-size="5" task-executor="taskExecutor"/>
    </batch:partition>
  </batch:step>
</batch:job>
<batch:step id="authorsPartitioned" parent="parentStep">
  <batch:tasklet>
    <batch:chunk
      reader="partitionedAuthorsReader"
      processor="authorsProcessor"
      writer="authorsWriter"
      commit-interval="1000"
      retry-limit="10">
      <batch:retryable-exception-classes>
        <batch:include class="org.springframework.dao.DeadlockLoserDataAccessException" />
      </batch:retryable-exception-classes>
    </batch:chunk>
    <batch:transaction-attributes isolation="READ_COMMITTED" />
  </batch:tasklet>
</batch:step>
<!-- Partitioner -->
<bean id="authorsPartitioner" class="gr.booksAdmin.batch.partitioners.ColumnRangePartitioner">
  <property name="dataSource" ref="inputDataSource" />
  <property name="table" value="ebooks_virtuemart_authors" />
  <property name="column" value="virtuemart_author_id" />
</bean>
<!-- Reader -->
<bean id="partitionedAuthorsReader" class="org.springframework.batch.item.database.JdbcPagingItemReader"
scope="step">
  <property name="dataSource" ref="inputDataSource"/>
  <property name="queryProvider">
    <bean class="org.springframework.batch.item.database.support.SqlPagingQueryProviderFactoryBean">
```

```
        <property name="dataSource" ref="inputDataSource"/>
        <property name="selectClause">
          <value>
            SELECT authors.virtuemart_author_id, authors.published, authors.created_on,
            authors.modified_on, authors.LastMod, authorsEll.au_name, authorsEll.au_desc,
            authorsEng.au_name, authorsEng.au_desc
          </value>
        </property>
        <property name="fromClause">
          <value>
            FROM ebooks_virtuemart_authors authors
            LEFT JOIN ebooks_virtuemart_authors_el_gr authorsEll
            ON authorsEll.virtuemart_author_id = authors.virtuemart_author_id
            LEFT JOIN ebooks_virtuemart_authors_en_gb authorsEng
            ON authorsEng.virtuemart_author_id = authors.virtuemart_author_id
          </value>
        </property>
        <property name="whereClause">
          <value>
            WHERE authors.virtuemart_author_id &gt;= #{stepExecutionContext[minValue]}
            AND authors.virtuemart_author_id &lt;= #{stepExecutionContext[maxValue]}
          </value>
        </property>
        <property name="sortKey" value="authors.virtuemart_author_id" />
      </bean>
    </property>
    <property name="pageSize" value="1000" />
    <property name="rowMapper" ref="authorsRowMapper" />
</bean>
```

Listing 5.53 Configuration του ColumnRangePartitioner

5.8.3 JUnit Testing του ColumnRangePartitioner

Για τα JUnit tests του ColumnRangePartitioner χρησιμοποιήθηκε η in-memory **H2 Database Engine**[33] και υλοποιήθηκαν tests σαν και αυτά της Listing 5.54. Η H2 μας επιτρέπει να ορίσουμε σε resource αρχεία τα scripts για δημιουργία πινάκων και εισαγωγή δεδομένων σε αυτούς, τα οποία κατά το set up του testing μας (**@Before** annotated μέθοδος) δημιουργούν μια βάση δεδομένων στη μνήμη που μπορεί να χρησιμοποιηθεί από τον partitioner στα tests που ακολουθούν (**@Test** annotated μέθοδοι). Τα **create-tables.sql** και **insert-data.sql** που φαίνονται στην μέθοδο αυτή, παραλείπονται για λόγους συντομίας καθώς δεν περιέχουν κάτι άξιο αναφοράς. Απλά δημιουργούν τέσσερεις πίνακες με στήλες ID (PK) και TITLE και εισάγουν εγγραφές σε αυτούς. Το πλήθος των εγγραφών είναι τέτοιο ώστε να μπορέσουμε να τεστάρουμε όλα τα πιθανά σενάρια του partitioning: αριθμός εγγραφών τέτοιος που θα οδηγήσει σε ίσα partitions (**uniformlyDistributestTest**), αριθμός εγγραφών τέτοιος που θα οδηγήσει στη δημιουργία και ενός partition με περισσότερες εγγραφές λόγω του υπολοίπου στο οποίο οδηγεί η FLOOR κατά τον

[33] Περισσότερα για την H2 Database Engine εδώ: http://www.h2database.com/html/main.html

υπολογισμό του size (**notUniformlyDistributedTest**) κλπ. Η in-memory βάση που δημιουργείται πρέπει κατά το tear down να γίνει shutdown και αυτό ακριβώς κάνουμε στην **@After** annotated μέθοδο.

```
public class ColumnRangePartitionerTest {

   private ColumnRangePartitioner partitioner = new ColumnRangePartitioner();
   private EmbeddedDatabase database;

   @Before
   public void setUp() {
      database = new EmbeddedDatabaseBuilder()
         .setType(EmbeddedDatabaseType.H2)
         .addScripts(
            "classpath:/create-tables.sql",
            "classpath:/insert-data.sql"
         )
         .build();
      partitioner.setDataSource(database);
   }

   @After
   public void tearDown() {
      database.shutdown();
   }

   @Test
   public void uniformlyDistributedTest() {
      partitioner.setTable("CB_TABLE_1");
      partitioner.setColumn("ID");
      Map<String, ExecutionContext> partitions = partitioner.partition(4);
      Assert.assertEquals(4, partitions.size()); //CB_TABLE_1 Has 4 Equal Partitions
      for(ExecutionContext executionContext : partitions.values()) {
         Integer min = 0, max = 0;
         for(Map.Entry<String, Object> entry : executionContext.entrySet()) {
            if(entry.getKey().equals("minValue")) min = (Integer) entry.getValue();
            if(entry.getKey().equals("maxValue")) max = (Integer) entry.getValue();
         }
         Assert.assertEquals(2, max-min); //Each Of The Partitions Has 3 Items
      }
   }

   @Test
   public void notUniformlyDistributedTest() {
      partitioner.setTable("CB_TABLE_2");
      partitioner.setColumn("ID");
      Map<String, ExecutionContext> partitions = partitioner.partition(4);
      Assert.assertEquals(4, partitions.size()); //CB_TABLE_2 Has A Couple Of More Items In The Last Partition
   }

   @Test
   public void singleRowTest() {
      partitioner.setTable("CB_TABLE_3");
      partitioner.setColumn("ID");
      Map<String, ExecutionContext> partitions = partitioner.partition(4);
      //CB_TABLE_3 Has Only One Row, So We Get Only One Partition
      Assert.assertEquals(1, partitions.size());
```

```
    }

    @Test
    public void emptyTableTest() {
        partitioner.setTable("CB_TABLE_4");
        partitioner.setColumn("ID");
        Map<String, ExecutionContext> partitions = partitioner.partition(4);
        Assert.assertEquals(0, partitions.size()); //When The Table Is Empty The Partitioner Returns An Empty Map
    }
}
```

Listing 5.54 JUnit Testing του ColumnRangePartitioner

5.8.4 Αποτελέσματα χρήσης του ColumnRangePartitioner

Κλείνουμε αυτή την ενότητα δίνοντας τις πληροφορίες που μέσα από το GUI παίρνουμε για το Job Execution και τα Step Executions ενός partitionedAuthorsJob (Σχήματα 5.3 και 5.4). Βλέπουμε πως το job διαβάζει 2.737.988 εγγραφές και τις επεξεργάζεται με 5 threads μοιράζοντάς τες στα partitions 0 έως και 4. Καθένα από αυτά επεξεργάζεται 547.597 εγγραφές με εξαίρεση το τελευταίο (Partition4) που επεξεργάζεται λίγο περισσότερες. Το job ολοκληρώνεται επιτυχώς σε κάτι λιγότερο από 15 λεπτά, με τον ίδιο όγκο εγγραφών όταν τρέχει σε single threaded configuration να χρειάζεται περισσότερο από 1 ½ ώρα. Γίνεται λοιπόν αντιληπτό πως, τουλάχιστον για το αρχικό import της Consolidation βάσης και των τεσσάρων βάσεων του ATG (τα σενάρια στα οποία ο partitioner χρησιμοποιήθηκε), η προσέγγιση αυτή ήταν μονόδρομος ούτως ώστε να επιτευχθεί η ολοκλήρωση της διαδικασίας εντός του καθορισμένου χρονικού παραθύρου.

Job Executions: [partitionedAuthorsJob | Job Instance Id = 110]

Execution Id	Start Time	End Time	Duration	Status	Exit Code	Step Executions	Actions
110	2017-05-07 08:56	2017-05-07 09:11	15 minutes	COMPLETED	COMPLETED	View Step Executions	

Σχήμα 5.3 Job Execution του partitionedAuthorsJob

Step Executions [authorsJob | Job Instance Id = 110, Job Execution Id =110]

ID	Name	Status	Exit Code	Read Count	Filter Count	Write Count	Read Skip Count	Write Skip Count	Process Skip Count	Commit Count	Rollback Count
0	authorsParametersStep	COMPLETED	COMPLETED	0	0	0	0	0	0	1	0
1	authorsInitializeStep	COMPLETED	COMPLETED	0	0	0	0	0	0	1	0
2	authorsPartitionedStep	COMPLETED	COMPLETED	2737988	3	2737985	0	0	0	2740	0
3	authorsPartitioned:Partition0	COMPLETED	COMPLETED	547597	1	547596	0	0	0	548	0
4	authorsPartitioned:Partition1	COMPLETED	COMPLETED	547597	0	547597	0	0	0	548	0
5	authorsPartitioned:Partition2	COMPLETED	COMPLETED	547597	2	547595	0	0	0	548	0
6	authorsPartitioned:Partition3	COMPLETED	COMPLETED	547597	0	547597	0	0	0	548	0
7	authorsPartitioned:Partition4	COMPLETED	COMPLETED	547600	0	547600	0	0	0	548	0
8	authorsFinalizeStep	COMPLETED	COMPLETED	0	0	0	0	0	0	1	0

Σχήμα 5.4 Step Executions του partitionedAuthorsJob

5.9 Web Interface

Τις ενέργειες που μπορεί να εκτελέσει ο χρήστης μέσω του web user interface μπορούμε να τις χωρίσουμε σε δυο κατηγορίες: ενέργειες για εκκίνηση ή σταμάτημα ενός job και ενέργειες για απλή παρουσίαση των όσων έχουν ήδη εκτελεστεί. Όπως αναφέρθηκε και στο προηγούμενο Κεφάλαιο, για την υλοποίηση του GUI χρησιμοποιήθηκε το Spring MVC. Πρόσθετα σε αυτό, και με σκοπό το να υλοποιηθεί γρηγορότερα το View, χρησιμοποιήθηκε και το **Twitter Bootstrap**[34], ένα HTML, CSS και JavaScript framework που μέσω των ευκολιών που παρέχει επιτρέπει την δημιουργία πλήρως responsive projects χωρίς ο χρήστης να χρειάζεται απαραίτητα –εφόσον του αρκούν τα default templates και functionalities– να γράψει δική του CSS ή JavaScript.

5.9.1 JobsController

Στον JobsController, που είναι και ο βασικός Controller της εφαρμογής ορίζονται τα endpoints **/jobLaunch** και **/stopJobExecution** (βλ. Listing 5.55). Το πρώτο (που έχει και περισσότερο ενδιαφέρον), δέχεται σαν **@RequestParam** το όνομα του job προς εκκίνηση, με την βοήθεια του JobRegistry (που γίνεται @Autowired στην κλάση μας) το μεταφράζει σε Job object και το περνάει μαζί με τις JobParameters (που χτίζονται από τις URL παραμέτρους του request με χρήση της μεθόδου **extractParameters**) στον JobLauncher που θα αναλάβει την ασύγχρονη εκκίνηση του job. Εφόσον δεν παρουσιαστεί κάποιο exception ο χρήστης ανακατευθύνεται στην αρχική σελίδα όπου με χρήση RedirectAttributes εμφανίζεται μήνυμα που τον ενημερώνει ότι το job ξεκίνησε επιτυχώς. Για την εξέλιξή του μπορεί να ενημερωθεί μέσω των όσων αναλύονται στα ακόλουθα.

Η /stopJobExecution, πολύ απλούστερη σε λογική, απλά παίρνει σαν **@PathVariable** το job execution ID που ο χρήστης επιθυμεί να σταματήσει και ζητά την εκτέλεση αυτής της ενέργειας από το **JobsService** που παρουσιάζεται σε επόμενη ενότητα.

```
@Controller
@RequestMapping("/admin")
public class JobsController {

   private static final Logger logger = LoggerFactory.getLogger(JobsController.class);

   private static final String JOB_PARAM = "job";
   private static final String EXCEPTION_PAGE = "exception";
   private static final String SUCCESS_LABEL = "successMessage";
   private static final String HOME_PAGE = "home";
   private static final String DEFAULT_JOB_NAME = "multipleJob";
```

[34] Περισσότερα για το Bootstrap εδώ: http://getbootstrap.com/

```
private static final Integer PAGE_SIZE = 10;  //Default Value
private static final Integer START_FROM = 0;  //Default Value

@Autowired
private JobRegistry jobRegistry;

@Autowired
private JobLauncher asynchronousJobLauncher;

@Autowired
private JobsService jobsService;

@RequestMapping(value = "/jobLaunch", method = RequestMethod.GET)
@ResponseStatus(HttpStatus.ACCEPTED)
public String launch(@RequestParam String jobName,
      HttpServletRequest request, RedirectAttributes redirectAttributes) throws Exception {
   JobParametersBuilder builder = extractParameters(request);
   //To Get A Different Job Instance
   builder.addDate("launchDate", new Date());
   //Asynchronous Job Launcher In Order To Not Monopolize The Web Container's Thread
   asynchronousJobLauncher.run(
      jobRegistry.getJob(request.getParameter(JOB_PARAM)),
      builder.toJobParameters()
   );
   redirectAttributes.addFlashAttribute(
      SUCCESS_LABEL,"Job With Name '" + jobName + "' Launched Successfully."
   );
   return "redirect:/";
}

@RequestMapping(value = "/stopJobExecution/{jobExecutionId}", method = RequestMethod.GET)
public String stopJobExecution(
      @PathVariable(value = "jobExecutionId") Long jobExecutionId,
      @RequestParam(required = true) String jobName,
      RedirectAttributes redirectAttributes) throws Exception {
   jobsService.stopJobExecution(jobExecutionId);
   redirectAttributes.addFlashAttribute(
      SUCCESS_LABEL, "Stop Signal Successfully Sent, But This Doesn't Mean That The Job Has Stopped."
   );
   return "redirect: /admin/jobExecutions/" + jobExecutionId + "?jobName=" + jobName;
}

@RequestMapping(value="/runningExecutions/{jobName}", method = RequestMethod.GET)
public String getRunningJobExecutions(@PathVariable(value = "jobName") String jobName, Model model) {
   model.addAttribute("jobExecutions", jobsService.getRunningJobExecutions(jobName));
   return "admin/jobExecutions";
}

@RequestMapping(value = "/jobExecutions/{jobInstanceId}", method = RequestMethod.GET)
public String getJobExecutions(@PathVariable(value = "jobInstanceId") Long jobInstanceId, Model model) {
   model.addAttribute("jobExecutions", jobsService.getJobExecutions(jobInstanceId));
   return "admin/jobExecutions";
}

@RequestMapping(value="/failedJobExecutions", method= RequestMethod.GET)
public String getFailedJobExecutions(Model model){
```

```
        model.addAttribute("failedJobExecutions", jobsService.getFailedJobExecutions());
        return "admin/failedJobExecutions";
    }

    @RequestMapping(value = "/stepExecutions/{jobExecutionId}", method = RequestMethod.GET)
    public String getStepExecutions(
            @PathVariable(value = "jobExecutionId") Long jobExecutionId, Model model) {
        model.addAttribute("stepExecutions", jobsService.getStepExecutions(jobExecutionId));
        return "admin/stepExecutions";
    }

    private JobParametersBuilder extractParameters(HttpServletRequest request){
        JobParametersBuilder builder = new JobParametersBuilder();
        Enumeration<String> paramNames = request.getParameterNames();
        while(paramNames.hasMoreElements()) {
            String paramName = paramNames.nextElement();
            if(!JOB_PARAM.equals(paramName)) {
                builder.addString(paramName, request.getParameter(paramName));
            }
        }
        return builder;
    }

    @RequestMapping(value = "/home", method= RequestMethod.GET)
    public String jobs(
        @RequestParam(required = false) String allJobs,
        @RequestParam(required = false) String pageSize,
        @RequestParam(required = false) String startFrom,
         HttpServletRequest request, Model model) throws Exception {
        String jobName = ProcessingUtils.trimToNull(allJobs) != null ? allJobs : DEFAULT_JOB_NAME;
        Integer resultsPerPage = NumberUtils.toInt(pageSize, PAGE_SIZE);
        Integer resultsStartFrom = NumberUtils.toInt(startFrom, START_FROM);
        model.addAttribute("jobName", jobName);
         model.addAttribute("allJobNames", jobService.getJobNames());//Select Options
        model.addAttribute("jobInstances", jobService.getJobInstances(jobName, resultsStartFrom, resultsPerPage));
        List<JobInstance> latestJobInstance = jobService.getJobInstances(jobName, 0, 1);
        if(latestJobInstance.size()>0){ //Add Info About The Latest Job Execution
            model.addAttribute("latestExecution",
                    jobService.getJobExecutions(latestJobInstance.get(0).getInstanceId()).get(0));
        }
        model.addAttribute("pager",
                new Pager(request.getRequestURI(), resultsPerPage, resultsStartFrom,
                  jobService.getJobInstanceCount(jobName), MiscellaneousUtils.httpRequestParameters(request)
                )
        );
        return HOME_PAGE;
    }

    @ExceptionHandler(Exception.class)
    public ModelAndView handleException(Exception exception) {
        logger.error("exception", exception);
        ModelAndView modelAndView = new ModelAndView(EXCEPTION_PAGE);
        modelAndView.addObject("exceptionName", exception.getClass().getSimpleName());
        return modelAndView;
    }
}
```

Listing 5.55 Υλοποίηση του JobsController

Εκτός αυτών, άλλα endpoints που ορίζονται και έχουν να κάνουν με την παρουσίαση των όσων η εφαρμογή έχει εκτελέσει είναι τα **/home** που εμφανίζει τα πιο πρόσφατα instances ενός job δοθέντος του ονόματός του ώστε να μπορεί ο χρήστης να αναζητήσει πληροφορίες σχετικά με αυτά, **/runningExecutions** που επιστρέφει τα job executions που εκτελούνται δοθέντος ενός job name, **/jobExecutions** που επιστρέφει τα job executions ενός job instance, **/stepExecutions** που επιστρέφει τα step executions ενός job execution και **/failedJobExecutions** που επιστρέφει όλα τα αποτυχημένα jobs. Όλα αυτά τα endpoints το μόνο που κάνουν είναι να αναθέτουν στο JobsService την ανάκτηση των απαιτούμενων πληροφοριών από το JobRepository, να τις θέτουν στο Model, και να ανακατευθύνουν το χρήστη στο κατάλληλο View. Η υλοποίηση του JobsService δίνεται ευθύς αμέσως.

5.9.2 JobsService

Το implementation του JobsService δίνεται στην Listing 5.56. Εδώ χρησιμοποιούνται τα **JobExplorer** και **JobOperator** interfaces που παρουσιάστηκαν θεωρητικά στο Κεφάλαιο 2, ώστε να ανακτώνται οι απαιτούμενες κάθε φορά πληροφορίες από το JobRepository του Spring Batch ή πιθανώς να εκτελούνται ενέργειες επί αυτού. Δεν υπάρχει κάτι στο οποία να χρειάζεται να σταθούμε με λεπτομέρεια εδώ. Ανάλογα με το τι ζητά το Presentation Layer από το Service Layer, εκτελούνται οι απαιτούμενες ενέργειες ώστε να ανακτηθούν οι απαραίτητες για το Model πληροφορίες (παραδείγματος χάριν τα ονόματα των registered jobs, τα εκτελούμενα executions ενός job, τα step executions ενός instance κλπ).

```
@Service
public class JobServiceImpl implements JobService {

   private static final Logger logger = LoggerFactory.getLogger(JobServiceImpl.class);

   @Autowired
   private JobExplorer jobExplorer;

   @Autowired
   private JobOperator jobOperator;

   @Override
   public List<String> getJobNames() {
      return jobExplorer.getJobNames();
   }

   @Override
   public List<JobInstance> getJobInstances(String jobName, int start, int count) {
      return jobExplorer.getJobInstances(jobName, start, count);
   }

   @Override
   public int getJobInstanceCount(String jobName) {
      int result = 0;
```

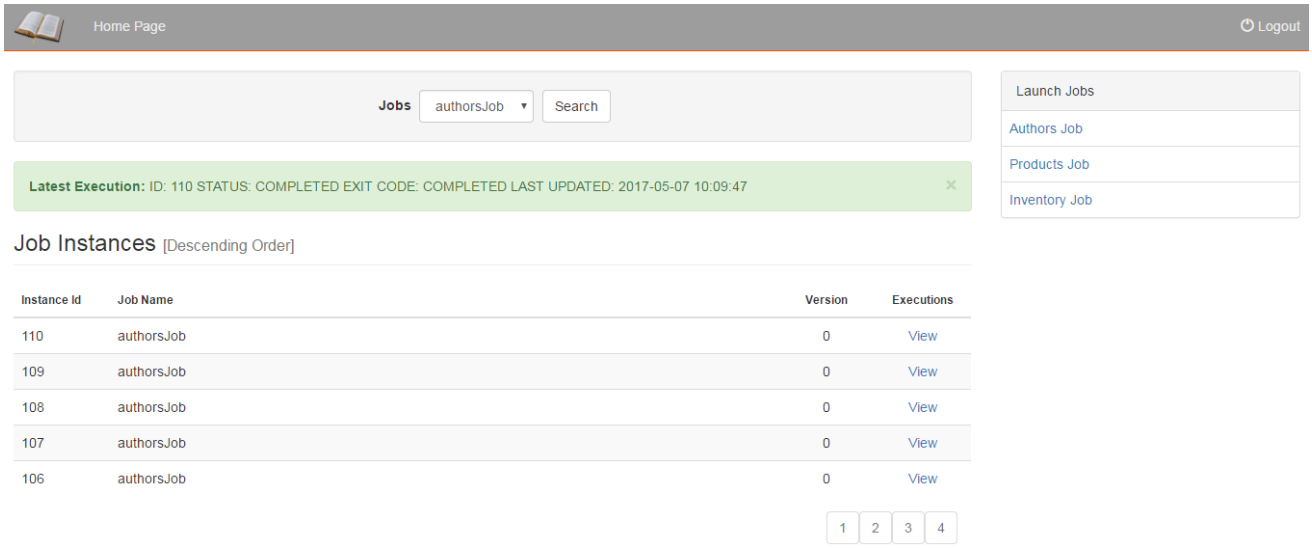
```
    try {
      result = jobExplorer.getJobInstanceCount(jobName);
    } catch (NoSuchJobException e) { //Do Nothing
      logger.debug("exception", e);
    }
    return result;
  }

  @Override
  public List<JobExecution> getJobExecutions(long jobInstanceId) {
    JobInstance jobInstance = jobExplorer.getJobInstance(jobInstanceId);
    return jobExplorer.getJobExecutions(jobInstance);
  }

  @Override
  public List<JobExecution> getFailedJobExecutions() {
    List<JobExecution> failedJobExecutions = new ArrayList<JobExecution>();
    List<String> jobNames = jobExplorer.getJobNames();
    for(String jobName : jobNames){
      int pageSize = 10;
      int currentPageSize = 10;
      int currentPage = 0;
      while(pageSize == currentPageSize){
        List<JobInstance> jobInstances = jobExplorer.getJobInstances(jobName, currentPage*pageSize, pageSize);
        currentPageSize = jobInstances.size();
        currentPage ++;
        for(JobInstance jobInstance : jobInstances){
          List<JobExecution> jobExecutions =
            jobExplorer.getJobExecutions(jobInstance);
          for(JobExecution jobExecution : jobExecutions){
            if(!jobExecution.getExitStatus().equals(ExitStatus.COMPLETED) &&
              !jobExecution.getExitStatus().equals(ExitStatus.STOPPED)) {
                failedJobExecutions.add(jobExecution);
            }
          }
        }
      }
    }
    return failedJobExecutions;
  }

  @Override
  public List<JobExecution> getRunningJobExecutions(String jobName) {
    List<JobExecution> result = new ArrayList<JobExecution>();
    try {
      Set<Long> runningJobExecutions = jobOperator.getRunningExecutions(jobName);
      for(Long runningJobExecution : runningJobExecutions){
        result.add(jobExplorer.getJobExecution(runningJobExecution));
      }
    } catch (NoSuchJobException e) {
      logger.debug("exception", e);
    }
    return result;
  }

  @Override
  public List<StepExecution> getStepExecutions(long jobExecutionId) {
```

```
        return (List<StepExecution>) jobExplorer
            .getJobExecution(jobExecutionId)
            .getStepExecutions();
    }

    @Override
    public boolean stopJobExecution(long jobExecutionId)
            throws NoSuchJobExecutionException, JobExecutionNotRunningException {
        return jobOperator.stop(jobExecutionId);
    }
}
```

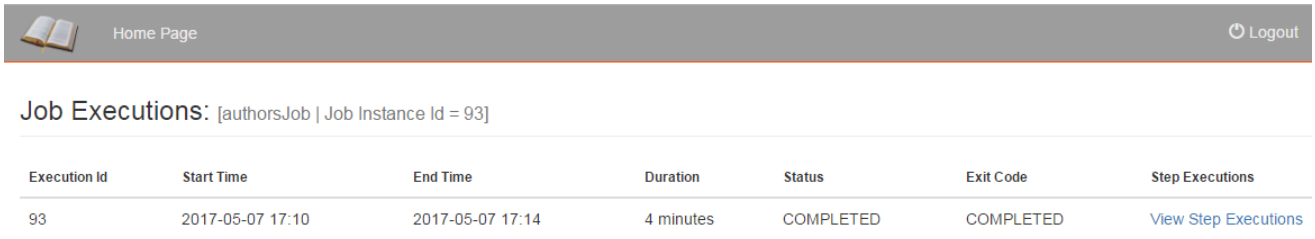
Listing 5.56 JobsService Implementation

5.9.3 Views

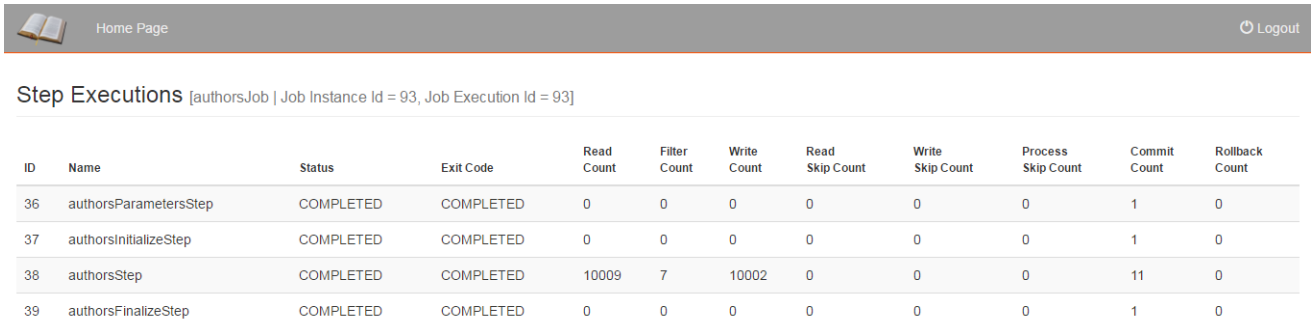
Όλα τα προηγούμενα έχουν τελικώς σαν παραγόμενο αποτέλεσμα views σαν και αυτά που δίνονται στα ακόλουθα σχήματα. Ο χρήστης μπορεί από την κεντρική σελίδα να αναζητήσει μέσω φίλτρων τα πιο πρόσφατα instances του job που τον ενδιαφέρει, να δει τα executions για κάποιο από αυτά καθώς και τα step executions ενός από αυτά τα job executions. Το interface δίνει πλήρεις πληροφορίες σχετικά με το πότε ξεκίνησε και πότε ολοκληρώθηκε ένα job, τα items που διάβασε, φίλτραρε και τελικά έγγραψε στην output database, τις παραμέτρους με τις οποίες έτρεξε το κάθε job, τα exceptions που πιθανώς εγέρθηκαν και πολλά άλλα. Ο χρήστης εφόσον έχει το ρόλο ROLE_ADMIN μπορεί να εκκινεί jobs μέσω του sidebar που φαίνεται στο πρώτο από τα παρακάτω σχήματα (διαφορετικά ούτε το βλέπει, ούτε μπορεί να καλέσει τις μεθόδους του JobsService καθώς το Spring Security μέσω intercept-url patterns του κόβει την πρόσβαση σε αυτές), να σταματά εφόσον το επιθυμεί jobs που τρέχουν κ.α.



Σχήμα 5.5 Home Page της εφαρμογής



Σχήμα 5.6 Job Executions για επιλεγμένο Job Instance



Σχήμα 5.7 Step Executions για επιλεγμένο Job Execution

5.9.3.1 Παράδειγμα JSP σελίδας με χρήση του Bootstrap

Σαν παράδειγμα του πως μπορούμε να χρησιμοποιήσουμε το Bootstrap για να χτίσουμε τις HTML σελίδες μας μπορούμε να δώσουμε τον JSP κώδικα της Home Page (Listing 5.57). Χρησιμοποιώντας τις CSS κλάσεις που το Bootstrap ορίζει για το grid-system του μπορούμε χωρίς ιδιαίτερο κόπο να δημιουργήσουμε responsive σελίδες, ενώ επιπρόσθετα, παρέχεται πληθώρα από έτοιμα CSS και JavaScript components που μπορούν να χρησιμοποιηθούν με το μόνο που να χρειάζεται συνήθως να είναι –και πάλι– το να δώσουμε στα elements της σελίδας μας τις κατάλληλες κλάσεις. Το framework ορίζει CSS breakpoints ώστε να χωρίσει τους clients σε 4 κατηγορίες με βάση τη διάστασή τους: Extra Small (xs, <768px), Small (sm, >=768px), Medium (md, >=992px) και Large (lg, >=1200px). Τα defaults μπορούν να αλλάξουν κατά βούληση, σε /admin υλοποιήσεις όμως σαν και αυτή που χρειαστήκαμε εμείς τα όσα έτοιμα παρέχονται είναι κάτι παραπάνω από αρκετά. Μεταξύ άλλων χρησιμοποιήθηκαν στην εφαρμογή μας fixed navbars, panels, pagination, dismissible alerts, popovers και πολλά άλλα.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<!DOCTYPE html>
<html lang="en">
  <head>
    <jsp:include page="../header.jsp"/>
  </head>
  <body>
```

```
    <jsp:include page="../navbar.jsp"/>
    <div class="container-fluid">
      <div class="row">
        <jsp:include page="../alerts.jsp"/>
        <div class="col-xs-12">
          <h3 class="page-header">
            Step Executions
          <small>
            [${param.jobName} | Job Instance Id = ${param.jobInstanceId},
            Job Execution Id = ${jobExecutionId}]
          </small>
          </h3>
          <div class="table-responsive">
            <table class="table table-striped" >
              <thead>
               <tr>
                  <th>ID</th>
                  <th>Name</th>
                  <th>Status</th>
                  <th>Exit Code</th>
                  <th>Read<br/>Count</th>
                  <th>Filter<br/>Count</th>
                  <th>Write<br/> Count</th>
                  <th>Read<br/>Skip Count</th>
                  <th>Write<br/>Skip Count</th>
                  <th>Process<br/>Skip Count</th>
                  <th>Commit<br/>Count</th>
                  <th>Rollback<br/>Count</th>
               </tr>
              </thead>
              <tbody>
                   <c:forEach var="stepExecution" items="${stepExecutions}">
                    <tr>
                      <td>${stepExecution.id}</td>
                        <td>${stepExecution.stepName}</td>
                      <td>${stepExecution.status}</td>
                      <td>${stepExecution.exitStatus.exitCode}</td>
                      <td>${stepExecution.readCount}</td>
                      <td>${stepExecution.filterCount}</td>
                      <td>${stepExecution.writeCount}</td>
                      <td>${stepExecution.readSkipCount}</td>
                      <td>${stepExecution.writeSkipCount}</td>
                      <td>${stepExecution.processSkipCount}</td>
                      <td>${stepExecution.commitCount}</td>
                      <td>${stepExecution.rollbackCount}</td>
                    </tr>
                  </c:forEach>
              </tbody>
            </table>
          </div>
        </div>
      </div>
    </div>
    <jsp:include page="../footer.jsp"/>
  </body>
</html>
```

Listing 5.57 JSP της Home Page

ΚΕΦΑΛΑΙΟ 6
Συμπεράσματα και Μελλοντικές Επεκτάσεις

Η εφαρμογή εδώ και κάποιους μήνες έχει βγει στην παραγωγή και χρησιμοποιείται με πλήρη επιτυχία. Οι στόχοι για το αρχικό import της Consolidation βάσης και των βάσεων του ATG εντός του στενού χρονικού παραθύρου που μας διατέθηκε επετεύχθησαν καθώς, όπως είδαμε στην σχετική με την υλοποίηση του partitioning ενότητα, με εκτέλεση σε multi-threaded configuration έγινε εφικτό το import κοντά 3 εκατομμυρίων εγγραφών σχετικών με τους Authors σε κάτι λιγότερο από 15 λεπτά. Η οντότητα αυτή δεν ήταν η μόνη με τόσο μεγάλο όγκο εγγραφών. Υπήρχαν περισσότερα από 7 εκατομμύρια προϊόντα (με το processing τους να είναι πολυπλοκότερο λόγω των οντοτήτων με τις οποίες κάθε προϊόν σχετίζεται), αντίστοιχος όγκος τιμών και αποθεμάτων κ.α. Παρόλα αυτά, με όλα τα jobs του initial import να τρέχουν παράλληλα από περισσότερα του ενός threads, ο διαθέσιμος χρόνος αποδείχτηκε παραπάνω από αρκετός ώστε να φορτωθούν τα δεδομένα σε όλα τα σχήματα. Μάλιστα, μέσω των validations και του filtering που κάναμε στην processing φάση όλων των οντοτήτων κατορθώσαμε να καθαρίσουμε σε μεγάλο βαθμό τα δεδομένα πετώντας εγγραφές σκουπίδια που συντηρούνταν στην παλιά υποδομή και δεν υπήρχε κανένας λόγος να μεταφερθούν και στις νέες.

Τα δέλτα jobs εκτελούνται επίσης ιδιαίτερα αποδοτικά σε καθημερινή βάση, με τα custom tasklets (JobInfoTasklet, JobParametersTasklet κλπ) να εκπληρώνουν όλες τις εμπορικές απαιτήσεις σχετικά με την άντληση των αλλαγών από την παλιά βάση, την δημιουργία στατιστικών στοιχείων σχετικά με τις αλλαγές που το κάθε job πραγματοποίησε κλπ. Το σύστημα καταγράφει και παρουσιάζει επαρκέστατες πληροφορίες σχετικά με το τι έχει εκτελεστεί (ή εκτελείται) στο background και είναι σε θέση να ενημερώνει τους διαχειριστές του συστήματος όταν προκύψει κάποιο σφάλμα. Μάλιστα, έχουν ήδη υλοποιηθεί πρόσθετα functionalities καθώς οι εμπορικοί χρήστες είδαν πως η υπάρχουσα υλοποίηση επιτρέπει στους developers να τους παραδίδουν σε σύντομο χρονικό διάστημα ότι έξτρα ζητηθεί. Και αυτό, σε μεγάλο βαθμό, οφείλεται στις ευκολίες που το Spring Batch παρέχει. Για παράδειγμα, τα notification emails που στέλνονταν στην περίπτωση σφάλματος με τον BatchMonitorNotifier έχουν πλέον επεκταθεί ώστε να μην ενημερώνονται οι χρήστες μόνο για τα σφάλματα αλλά και για τις επιτυχώς ολοκληρωμένες εργασίες και το πλήθος ανά οντότητα που αυτές τροποποίησαν (τα στατιστικά στοιχεία δηλαδή που ήδη κρατούνταν με τη βοήθεια του JobInfoTasklet στην βάση δεδομένων). Άλλη ενδιαφέρουσα αλλαγή είναι η on demand ενημέρωση της Consolidation βάσης με τις αλλαγές που έχουν προκύψει στην input database για συγκεκριμένο ή συγκεκριμένους τίτλους. Το ότι το δέλτα job είναι χρονοπρογραμματισμένο να τρέχει μια φορά τη μέρα σημαίνει πως αν στο ενδιάμεσο δημιουργηθεί ένα νέο είδος που επείγει να ανοιχτεί στο νέο site αυτό δεν θα μπορεί να γίνει. Υλοποιήθηκε λοιπόν η δυνατότητα αναζήτησης στην παλιά βάση

δεδομένων για είδη που δεν υπάρχουν στην νέα, μαρκάρισμα κάποιων από αυτά ως είδη προς import και εκκίνηση μέσω του Web Interface job που θα φέρνει στην νέα υποδομή μόνο αυτά τα μαρκαρισμένα είδη.

Η μεγαλύτερη πρόκληση όμως στην παρούσα φάση είναι η υλοποίηση γεφυρών διασύνδεσης με τους παρόχους των δεδομένων (βιβλιονet, Gardners, Nielsen κ.α.) που υποστήριζε η παλιά εφαρμογή. Τα συμβόλαια με τους παρόχους αυτούς σύντομα θα λήξουν και η εμπορική απαίτηση είναι να πάρει η νέα εφαρμογή όλο τον έλεγχο ώστε να μη χρειάζεται (άσκοπα όπως πιθανώς κάποιος θα μπορούσε να σκεφτεί), να συντηρούνται σχέσεις με την εταιρία που υλοποίησε την παλιά εφαρμογή. Κάθε είδους συνδιαλλαγής με την παλιά βάση δεδομένων δηλαδή θα κοπεί και η Consolidation βάση θα πρέπει να ενημερώνεται, και πάλι σε καθημερινή βάση, με τα δεδομένα που οι πραγματικοί πάροχοι του περιεχομένου παρέχουν. Επαφές με αυτούς έχουν ήδη γίνει. Κάποιοι δίνουν τα δεδομένα τους σε database views με κάποια στήλη να καθορίζει σε κάθε πίνακα την ημερομηνία τροποποίησης (άρα η υπάρχουσα υλοποίηση για ανάγνωση από βάση δεδομένων με κάποιες αλλαγές θα καλύψει τις όποιες ανάγκες), ενώ άλλοι τα παρέχουν σε αρχεία μεγάλου όγκου χωρίς όμως απαραίτητα να παρέχουν timestamps για το ποιες εγγραφές σε αυτά τα αρχεία είναι νέες και ποιες όχι. Εδώ, θα πρέπει οπωσδήποτε να εφαρμοστούν τεχνικές partitioning ώστε να μπορέσει η επεξεργασία να παραλληλιστεί κατά το δυνατόν, και έλεγχοι κατά το processing phase για το ποιες από τις εγγραφές των αρχείων έχουν πραγματικά αλλάξει. Η υπάρχουσα υλοποίησή έχει θέσει τις βάσεις, το σενάριο όμως μοιάζει αρκετά πιο σύνθετο και ίσως συνεπάγεται αρκετές αλλαγές. Το Spring Batch όμως δεν θα έχει πρόβλημα να αντιμετωπίσει την όποια απαίτηση. Το framework αποδείχτηκε επαρκέστατο, η κοινότητά του διαρκώς μεγαλώνει, η τεχνογνωσία από την μεριά μας αποκτήθηκε και άρα δεν τίθεται καν σαν θέμα το να αντικατασταθεί στην επόμενη φάση του έργου από κάποιο άλλο εργαλείο.

ΠΑΡΑΡΤΗΜΑ

A. Κατάλογος των SQL Scripts της Output Database

Τα παρακάτω SQL scripts για δημιουργία των πινάκων της output database εκτελέστηκαν μέσω του Liquibase με τον τρόπο που αναλύθηκε στην Ενότητα 5.1.2. Ορίζουν τους πίνακες που by default χρειάζεται το Spring Batch για να λειτουργήσει (BATCH_JOB_INSTANCE, BATCH_JOB_EXECUTION κλπ), τους βοηθητικούς πίνακες που δημιουργήθηκαν για να επιτευχθεί η έξτρα λογική που περιγράφηκε στο Κεφάλαιο 5 (BATCH_JOB_INFO, BATCH_JOB_CONFIGURATION κλπ) και όλους τους πίνακες που αφορούν στα domain objects της εφαρμογής.

```
-- AUTHORS TABLE

CREATE TABLE "CB_AUTHOR" (
 "ID" NUMBER NOT NULL PRIMARY KEY,
 "NAME_ELL" VARCHAR2 (500 CHAR) NOT NULL,
 "NAME_ENG" VARCHAR2 (500 CHAR) NOT NULL,
 "DESCRIPTION_ELL" VARCHAR2 (1000 CHAR) DEFAULT NULL,
 "DESCRIPTION_ENG" VARCHAR2 (1000 CHAR) DEFAULT NULL,
 "CREATED_ON" TIMESTAMP (6) DEFAULT NULL,
 "LAST_MOD" TIMESTAMP (6) DEFAULT NULL,
 "CHANGED" NUMBER(1,0) DEFAULT NULL,
 "CHANGED_BY" NUMBER(19,0) DEFAULT NULL,
 "CONSUMED" NUMBER(1,0) DEFAULT NULL,
 "CONSUMED_BY"  NUMBER(19,0) DEFAULT NULL
);

CREATE INDEX CB_AUTHOR_CHANGED ON CB_AUTHOR (CHANGED);

CREATE INDEX CB_AUTHOR_CHANGED_BY ON CB_AUTHOR (CHANGED_BY);

CREATE INDEX CB_AUTHOR_CONSUMED ON CB_AUTHOR (CONSUMED);

CREATE INDEX CB_AUTHOR_CONSUMED_BY ON CB_AUTHOR (CONSUMED_BY);

-- PRODUCTS TABLE

CREATE TABLE "CB_PRODUCT" (
  "ID" NUMBER NOT NULL PRIMARY KEY,
  "HIDDEN" NUMBER(1,0) DEFAULT 0 NOT NULL,
  "SKU" VARCHAR2(64 CHAR) DEFAULT NULL,
  "KIND" NUMBER NOT NULL,
   "AUTHORS" CLOB DEFAULT NULL,
  "NAME_ELL" VARCHAR2(180 CHAR) NOT NULL,
  "NAME_ENG" VARCHAR2(180 CHAR) NOT NULL,
  "DESCRIPTION_ELL" CLOB DEFAULT NULL,
  "DESCRIPTION_ENG" CLOB DEFAULT NULL,
  "ISBN" VARCHAR2(255 CHAR) DEFAULT NULL,
  "CREATED_ON" TIMESTAMP (6) DEFAULT NULL,
   "LAST_MOD" TIMESTAMP (6) DEFAULT NULL,
   "CHANGED" NUMBER(1,0) DEFAULT NULL,
   "CHANGED_BY" NUMBER(19,0) DEFAULT NULL,
   "CONSUMED" NUMBER(1,0) DEFAULT NULL,
```

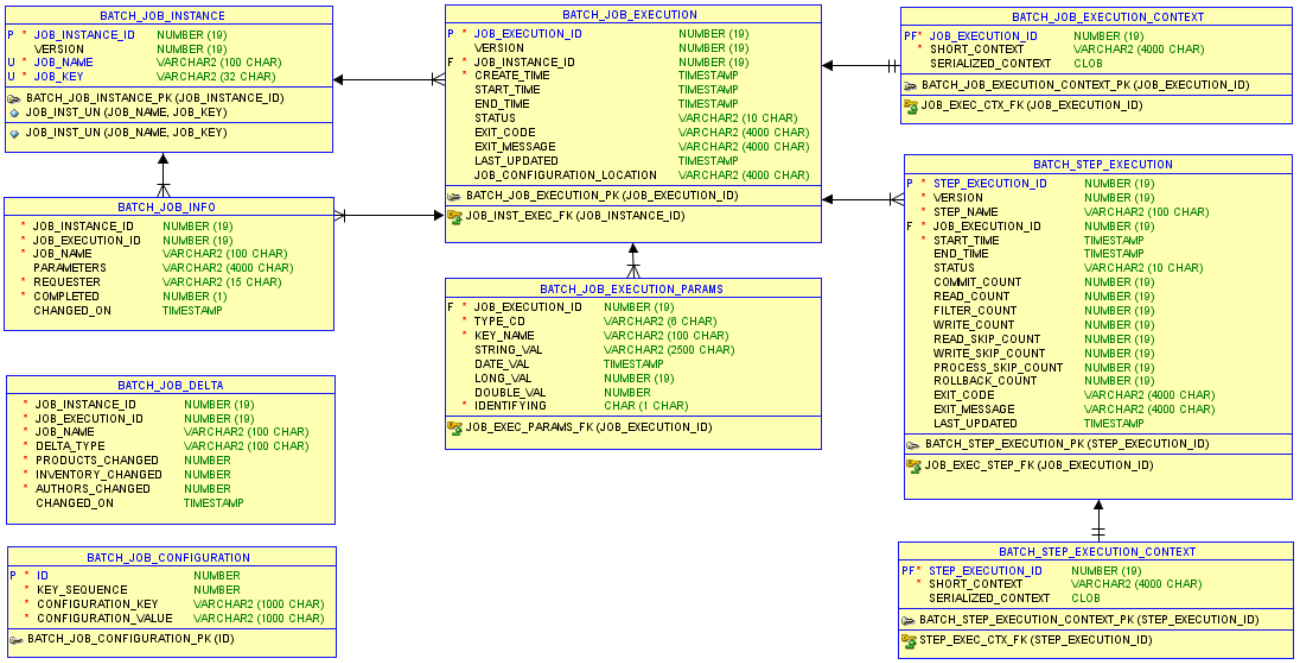
```
   "CONSUMED_BY"  NUMBER(19,0) DEFAULT NULL
);

CREATE INDEX CB_PRODUCT_CHANGED ON CB_PRODUCT (CHANGED);
CREATE INDEX CB_PRODUCT_CHANGED_BY ON CB_PRODUCT (CHANGED_BY);
CREATE INDEX CB_PRODUCT_CONSUMED ON CB_PRODUCT (CONSUMED);
CREATE INDEX CB_PRODUCT_CONSUMED_BY ON CB_PRODUCT (CONSUMED_BY);

-- INVENTORY TABLE

CREATE TABLE "CB_INVENTORY" (
 "ID" NUMBER NOT NULL PRIMARY KEY,
 "STOCK" NUMBER DEFAULT 0 NOT NULL,
 "CHANGED" NUMBER(1,0) DEFAULT NULL,
 "CHANGED_BY" NUMBER(19,0) DEFAULT NULL,
 "CONSUMED" NUMBER(1,0) DEFAULT NULL,
 "CONSUMED_BY"  NUMBER(19,0) DEFAULT NULL
);

CREATE INDEX CB_INVENTORY_CHANGED ON CB_INVENTORY (CHANGED);
CREATE INDEX CB_INVENTORY_CHANGED_BY ON CB_INVENTORY (CHANGED_BY);
CREATE INDEX CB_INVENTORY_CONSUMED ON CB_INVENTORY (CONSUMED);
CREATE INDEX CB_INVENTORY_CONSUMED_BY ON CB_INVENTORY (CONSUMED_BY);

-- PRODUCT RELATED ENTITITES TABLES

CREATE TABLE "CB_CATEGORY" (
 "ID" NUMBER NOT NULL PRIMARY KEY,
 "NAME_ELL" VARCHAR2(180 CHAR) NOT NULL,
 "NAME_ENG" VARCHAR2(180 CHAR) NOT NULL,
 "DESCRIPTION_ELL" CLOB DEFAULT NULL,
 "DESCRIPTION_ENG" CLOB DEFAULT NULL
);

CREATE TABLE "CB_MANUFACTURER" (
 "ID" NUMBER NOT NULL PRIMARY KEY,
 "NAME_ELL" VARCHAR2(180 CHAR) NOT NULL,
 "NAME_ENG" VARCHAR2(180 CHAR) NOT NULL
);

CREATE TABLE "CB_AGE" (
 "ID" NUMBER NOT NULL PRIMARY KEY,
 "NAME_ELL" VARCHAR2(255 CHAR) NOT NULL,
 "NAME_ENG" VARCHAR2(255 CHAR) NOT NULL
);

CREATE TABLE "CB_COVER" (
 "ID" NUMBER NOT NULL PRIMARY KEY,
 "NAME_ELL" VARCHAR2(255 CHAR) NOT NULL,
 "NAME_ENG" VARCHAR2(255 CHAR) NOT NULL
);

CREATE TABLE "CB_SUBJECT" (
 "ID" NUMBER NOT NULL PRIMARY KEY,
 "NAME_ELL" VARCHAR2(255 CHAR) NOT NULL,
 "NAME_ENG" VARCHAR2(255 CHAR) NOT NULL
);

CREATE SEQUENCE CB_SUBJECT_SEQUENCE START WITH 1 INCREMENT BY 1 CACHE 20;
CREATE SEQUENCE CB_AGE_SEQUENCE START WITH 1 INCREMENT BY 1 CACHE 20;
CREATE SEQUENCE CB_COVER_SEQUENCE START WITH 1 INCREMENT BY 1 CACHE 20;
```

```
-- SPRING BATCH TABLES

CREATE TABLE BATCH_JOB_INSTANCE  (
 JOB_INSTANCE_ID NUMBER(19,0)  NOT NULL PRIMARY KEY ,
 VERSION NUMBER(19,0) ,
 JOB_NAME VARCHAR2(100 CHAR) NOT NULL,
 JOB_KEY VARCHAR2(32 CHAR) NOT NULL,
 constraint JOB_INST_UN unique (JOB_NAME, JOB_KEY)
);

CREATE TABLE BATCH_JOB_EXECUTION  (
 JOB_EXECUTION_ID NUMBER(19,0)  NOT NULL PRIMARY KEY ,
 VERSION NUMBER(19,0)  ,
 JOB_INSTANCE_ID NUMBER(19,0) NOT NULL,
 CREATE_TIME TIMESTAMP NOT NULL,
 START_TIME TIMESTAMP DEFAULT NULL ,
 END_TIME TIMESTAMP DEFAULT NULL ,
 STATUS VARCHAR2(10 CHAR) ,
 EXIT_CODE VARCHAR2(4000 CHAR) ,
 EXIT_MESSAGE VARCHAR2(4000 CHAR) ,
 LAST_UPDATED TIMESTAMP,
 JOB_CONFIGURATION_LOCATION VARCHAR(4000 CHAR) NULL,
 constraint JOB_INST_EXEC_FK foreign key (JOB_INSTANCE_ID)
 references BATCH_JOB_INSTANCE(JOB_INSTANCE_ID)
);

CREATE TABLE BATCH_JOB_EXECUTION_PARAMS  (
 JOB_EXECUTION_ID NUMBER(19,0) NOT NULL ,
 TYPE_CD VARCHAR2(6 CHAR) NOT NULL ,
 KEY_NAME VARCHAR2(100 CHAR) NOT NULL ,
 STRING_VAL VARCHAR2(2500 CHAR) ,
 DATE_VAL TIMESTAMP DEFAULT NULL ,
 LONG_VAL NUMBER(19,0) ,
 DOUBLE_VAL NUMBER ,
 IDENTIFYING CHAR(1 CHAR) NOT NULL ,
 constraint JOB_EXEC_PARAMS_FK foreign key (JOB_EXECUTION_ID)
 references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
);

CREATE TABLE BATCH_STEP_EXECUTION  (
 STEP_EXECUTION_ID NUMBER(19,0)  NOT NULL PRIMARY KEY ,
 VERSION NUMBER(19,0) NOT NULL,
 STEP_NAME VARCHAR2(100 CHAR) NOT NULL,
 JOB_EXECUTION_ID NUMBER(19,0) NOT NULL,
 START_TIME TIMESTAMP NOT NULL ,
 END_TIME TIMESTAMP DEFAULT NULL ,
 STATUS VARCHAR2(10 CHAR) ,
 COMMIT_COUNT NUMBER(19,0) ,
 READ_COUNT NUMBER(19,0) ,
 FILTER_COUNT NUMBER(19,0) ,
 WRITE_COUNT NUMBER(19,0) ,
 READ_SKIP_COUNT NUMBER(19,0) ,
 WRITE_SKIP_COUNT NUMBER(19,0) ,
 PROCESS_SKIP_COUNT NUMBER(19,0) ,
 ROLLBACK_COUNT NUMBER(19,0) ,
 EXIT_CODE VARCHAR2(4000 CHAR) ,
 EXIT_MESSAGE VARCHAR2(4000 CHAR) ,
 LAST_UPDATED TIMESTAMP,
 constraint JOB_EXEC_STEP_FK foreign key (JOB_EXECUTION_ID)
 references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
);
```

```
CREATE TABLE BATCH_STEP_EXECUTION_CONTEXT  (
 STEP_EXECUTION_ID NUMBER(19,0) NOT NULL PRIMARY KEY,
 SHORT_CONTEXT VARCHAR2(4000 CHAR) NOT NULL,
 SERIALIZED_CONTEXT CLOB ,
 constraint STEP_EXEC_CTX_FK foreign key (STEP_EXECUTION_ID)
 references BATCH_STEP_EXECUTION(STEP_EXECUTION_ID)
);

CREATE TABLE BATCH_JOB_EXECUTION_CONTEXT  (
 JOB_EXECUTION_ID NUMBER(19,0) NOT NULL PRIMARY KEY,
 SHORT_CONTEXT VARCHAR2(4000 CHAR) NOT NULL,
 SERIALIZED_CONTEXT CLOB ,
 constraint JOB_EXEC_CTX_FK foreign key (JOB_EXECUTION_ID)
 references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
);

CREATE SEQUENCE BATCH_STEP_EXECUTION_SEQ START WITH 0 MINVALUE 0 MAXVALUE 9223372036854775807 NOCYCLE;

CREATE SEQUENCE BATCH_JOB_EXECUTION_SEQ START WITH 0 MINVALUE 0 MAXVALUE 9223372036854775807 NOCYCLE;

CREATE SEQUENCE BATCH_JOB_SEQ START WITH 0 MINVALUE 0 MAXVALUE 9223372036854775807 NOCYCLE;

-- TABLES EXTENDING SPRING BATCH INFRASTRUCTURE

-- TABLE FOR CONFIGURATION PER ENVIRONMENT
CREATE TABLE BATCH_JOB_CONFIGURATION (
  ID NUMBER NOT NULL PRIMARY KEY,
  KEY_SEQUENCE NUMBER  NOT NULL,
  CONFIGURATION_KEY VARCHAR2 (1000 CHAR) NOT NULL,
  CONFIGURATION_VALUE VARCHAR2 (1000 CHAR) NOT NULL
);

CREATE SEQUENCE BATCH_CONFIGURATION_SEQUENCE START WITH 1 INCREMENT BY 1 CACHE 20;

-- TABLE FOR STATISTICS PER EXECUTED JOB
CREATE TABLE BATCH_JOB_DELTA (
  JOB_INSTANCE_ID NUMBER(19,0) NOT NULL,
  JOB_EXECUTION_ID NUMBER (19,0) NOT NULL,
  JOB_NAME VARCHAR2 (100 CHAR) NOT NULL,
  DELTA_TYPE VARCHAR2 (100 CHAR) NOT NULL,
  PRODUCTS_CHANGED NUMBER DEFAULT 0 NOT NULL,
  INVENTORY_CHANGED NUMBER DEFAULT 0 NOT NULL,
  AUTHORS_CHANGED NUMBER DEFAULT 0 NOT NULL,
  CHANGED_ON TIMESTAMP (6) DEFAULT NULL
);

-- INFO ABOUT THE JOBS EXECUTED
CREATE TABLE BATCH_JOB_INFO (
  JOB_INSTANCE_ID NUMBER (19, 0) NOT NULL,
  JOB_EXECUTION_ID NUMBER (19, 0) NOT NULL,
  JOB_NAME VARCHAR2 (100 CHAR) NOT NULL,
  PARAMETERS VARCHAR2 (4000 CHAR) DEFAULT NULL,
  REQUESTER VARCHAR2 (15 CHAR) NOT NULL,
  COMPLETED NUMBER (1, 0) DEFAULT 0 NOT NULL,
  CHANGED_ON TIMESTAMP (6) DEFAULT NULL,
 constraint INFO_INSTANCE_FK foreign key (JOB_INSTANCE_ID)
 references BATCH_JOB_INSTANCE(JOB_INSTANCE_ID),
 constraint INFO_EXECUTION_FK foreign key (JOB_EXECUTION_ID)
 references BATCH_JOB_EXECUTION(JOB_EXECUTION_ID)
);
```

B. Batch Tables (Output Database)

Για λόγους πληρότητας, εκτός των SQL scripts δίνεται και το σχήμα των πινάκων του infrastructure της εφαρμογής το οποίο αποτελούν οι default Spring Batch πίνακες μαζί με τους βοηθητικούς πίνακες που προαναφέρθηκαν. Όλοι δημιουργούνται στην output database της εφαρμογής (Oracle) καθώς μόνο σε αυτή υπήρχαν δικαιώματα ερωτημάτων INSERT / UPDATE.



Βιβλιογραφία

J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In Communications of the ACM, 51 (1): 107-113, 2008.

Kimball, R. The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouse, 1996.

Michael T. Minella. Pro Spring Batch: Batch Processing With the Spring Batch Framework, 2011.

P. Raja Malleswara Rao. Spring Batch Essentials: Design Develop and Deliver Robust Bath Applications with the Power of The Spring Batch Framework, 2015.

Haerder T, Reuter A. Principles of Transaction-Oriented Database Recovery, 1983.

Jim Gray, Andreas Reuter. Distributed Transaction Processing: Concepts and Techniques, 1993.

Java Community Process. Batch Applications for The Java Platform, 2015.

Willie Willer, Joshua White. Spring In Practice, Manning 2013.

Craig Walls. Spring in Action, 4th Edition, Manning 2014.

Arnaud Cogoluegnes, Thierry Templer, Gary Gregory, Olivier Bazoud. Spring Batch in Action, 2012.

Gary Mak. Spring Recipes: A Problem-Solution Approach, 2010.

E.F. Codd. A Relational Model of Data for Large Shared Data Banks, 1970.

Jim Gray. TheTransaction Concept: Virtues And Limitations, 1981.

Collaborative Project LeanBigData: Ultra-Scalable and Ultra-Efficient Integrated and Visual Big Data Analytics, 2013.