ΠΑΝΕΠΙΣΤΗΜΙΟ ΜΑΚΕΔΟΝΙΑΣ
ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΤΜΗΜΑΤΟΣ ΕΦΑΡΜΟΣΜΕΝΗΣ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΤΕΧΝΙΚΕΣ ΑΝΑΛΥΣΗΣ ΚΑΙ ΑΞΙΟΛΟΓΗΣΗΣ ΠΛΑΙΣΙΩΝ ΛΟΓΙΣΜΙΚΟΥ

Διπλωματική Εργασία

Του

Κατσαΐτη Χρήστου

Θεσσαλονίκη, Νοέμβριος 2021

ΤΕΧΝΙΚΕΣ ΑΝΑΛΥΣΗΣ ΚΑΙ ΑΞΙΟΛΟΓΗΣΗΣ ΠΛΑΙΣΙΩΝ ΛΟΓΙΣΜΙΚΟΥ

Κατσαΐτης Χρήστος

Πτυχίο Εφαρμοσμένης Πληροφορικής, Πα.Μακ., 2021

Διπλωματική Εργασία

υποβαλλόμενη για τη μερική εκπλήρωση των απαιτήσεων του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΤΙΤΛΟΥ ΣΠΟΥΔΩΝ ΣΤΗΝ ΕΦΑΡΜΟΣΜΕΝΗ ΠΛΗΡΟΦΟΡΙΚΗ

Επιβλέπων Καθηγητής
Αλέξανδρος Χατζηγεωργίου

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 22/02/2022

Αλέξανδρος Χατζηγεωργίου	Στυλιανός Ξυνόγαλος	Απόστολος Αμπατζόγλου
.....	.....	.....

Κατσαΐτης Χρήστος

.....

Περίληψη

Η παρούσα εργασία αποτελεί μια απόπειρα μελέτης της συσχέτισης μεταξύ των λογικών αφαιρέσεων (abstractions) που παρέχονται από γνωστά πλαίσια λογισμικού, σε σχέση με την ωφέλεια που προκύπτει για τη διαδικασία ανάπτυξης και συντήρησης λογισμικού. Εξετάζεται το κατά πόσο είναι ωφέλιμη η χρήση ενός framework στην ανάπτυξη λογισμικού , όσον αφορά την ταχύτητα της ανάπτυξης, την επεκτασιμότητα, αλλά και την ευκολία συντήρησης, λαμβάνοντας υπ'όψιν και τα προβλήματα που πιθανώς δημιουργούνται, απο τους περιορισμούς που επιβάλλονται στους προγραμματιστές κατά την χρήση αυτών των πλαισίων. Πιο συγκεκριμένα γίνεται μια υλοποίηση διασύνδεσης μιας διαδικτυακής εφαρμογής με μια βάση δεδομένων, χρησιμοποιώντας τις παρεχόμενες αφαιρέσεις από το πλαίσιο που θα χρησιμοποιηθεί, και μετά γίνεται η ίδια διαδικασία, χωρίς τη χρήση πλαισίου. Η ίδια διαδικασία ακολουθείται και για την υλοποίηση ενός απλού συστήματος αυθεντικοποίησης (security configuration), και για την υλοποίηση διασύνδεσης μέσω http. Ως μετρικές χρησιμοποιούνται οι (Maintainability rating, Cyclomatic Complexity, Cognitive Complexity, και η σουίτα μετρικών από Chidamber & Kemerer). Επιπλέον, γίνεται και μια ανάλυση ποιοτικών μετρικών, όπως πχ χρόνος που χρειάστηκε για την ανάπτυξη, διαθέσιμη τεκμηρίωση, ευκολία ανάπτυξης. Οι παραπάνω μετρικές παρουσιάζονται και αξιολογούνται στο κεφάλαιο «Συμπεράσματα».

Λέξεις Κλειδιά: Java, MVC, software frameworks, Spring framework, code quality metrics, τεχνικό χρέος, Inversion of Control

Abstract

The work presented here is an attempt to study the correlation between the logical abstractions provided by known software frameworks, in relation to the benefit that arises from the software development and maintenance process.

We examine whether it is beneficial to use a framework in software development, in terms of development speed, scalability, and ease of maintenance, taking into account the problems that may arise from the constraints imposed on developers during use of these frameworks. Specifically, an implementation of linking a web application to a database is performed, using the provided abstractions from the framework to be used, and then the same process is performed, without the use of a framework.

The same procedure is followed for the implementation of a simple authentication system (security configuration), and for the implementation of an interface through http. After the development of said implementations, a number of qualitative and quantitative metrics are presented and evaluated in order to provide some insight in regards to the benefits and pitfalls of developing software with the aid of software frameworks.

Keywords: Java, MVC, software frameworks, Spring framework, code quality metrics, technical debt, Inversion of Control

Πρόλογος – Ευχαριστίες

Θα ήθελα πρωτίστως να ευχαριστήσω τον επιβλέποντα καθηγητή κ. Αλέξανδρο Χατζηγεωργίου, καθηγητή στο Πανεπιστήμιο Μακεδονίας, και κοσμήτορα της σχολής Επιστημών Πληροφορίας, για τη στήριξη, καθοδήγηση, και κατανόηση που μου παρείχε κατα τη διάρκεια της εκπόνησης αυτής της εργασίας.

Επίσης θα ήθελα να ευχαριστήσω την οικογένεια μου, για την αμέριστη στήριξη, συμπαράσταση, και υπομονή που επέδειξαν καθ'όλη τη διάρκεια εκπόνησης της εργασίας μου, αλλα και της φοίτησης μου στο πρόγραμμα μεταπτυχιακών σπουδών.

Περιεχόμενα

1 Εισαγωγή 1

1.1 Πρόβλημα – Σημαντικότητα του θέματος 1

1.2 Σκοπός – Στόχοι 2

1.3 Διάρθρωση της μελέτης 2

2 Βιβλιογραφική Επισκόπηση – Θεωρητικό Υπόβαθρο 4

2.1 Θεωρητικό Υπόβαθρο 4

2.1.1 Γενικοί όροι 4

2.1.2 Μετρικές 8

3 Μεθοδολογία 12

Γενικά 12

3.1.1 Οργάνωση της ΒΔ 13

3.1.2 Διαμόρφωση εφαρμογής με το Spring-Boot 14

3.1.3 Διαμόρφωση εφαρμογής ιδιότυπη υλοποίηση 17

3.2 Διασύνδεση με ΒΔ 19

3.2.1 Με τη χρήση του Spring boot 19

3.2.2 Με ιδιότυπη υλοποίηση 21

3.3 Υλοποίηση για διασύνδεση μέσω HTTP 25

3.3.1 Με τη χρήση του Spring boot 25

3.3.2 Με ιδιότυπη υλοποίηση 26

3.4 Υλοποίηση security 30

3.4.1 Με τη χρήση του Spring boot 30

3.4.2 Με ιδιότυπη υλοποίηση 31

4 Επίλογος 33

4.1 Σύνοψη και συμπεράσματα 33

4.1.1 Γενικά συμπεράσματα 33

4.1.2 Τεχνικό χρέος 36

4.1.3 Ποσοτικές μετρικές 38

4.1.4 Ποιοτικές μετρικές 46

5 Όρια και περιορισμοί της έρευνας 49

5.1 Μελλοντικές Επεκτάσεις 50

Βιβλιογραφία 51

1 Εισαγωγή

1.1 Πρόβλημα – Σημαντικότητα του θέματος

Από την αρχή και τη δημιουργία τους, τα πλαίσια λογισμικού έχουν υιοθετηθεί ευρέως στην ανάπτυξη λογισμικού. Σήμερα, ένα μεγάλο ποσοστό των προγραμμάτων υπολογιστών που δημιουργούνται χρησιμοποιούν τον κώδικα που παρέχεται από ένα πλαίσιο λογισμικού. Τα κύρια πλεονεκτήματα των πλαισίων που οδήγησαν σε μια τόσο ευρεία υιοθέτηση είναι η ταχύτητα ανάπτυξης, η μείωση του κώδικα αρχικοποιήσης της εφαρμογής και η αυτοματοποίηση των απανταχού χαρακτηριστικών. Αυτό, με τη σειρά του, επιτρέπει στους προγραμματιστές λογισμικού να αφιερώνουν περισσότερο χρόνο σε ζητήματα που προσανατολίζονται στην επιχειρηματική λογική, παρά σε αυστηρά τεχνικές πτυχές της εφαρμογής που έχουν αναπτυχθεί. Προκειμένου να υπάρχουν αυτά τα πλεονεκτήματα, τα πλαίσια λογισμικού βασίζονται σε μεγάλο βαθμό στις αφαιρέσεις. Ως εκ τούτου, υπάρχει μια πληθώρα αποφάσεων που πρέπει να ληφθούν για να διαμορφωθεί σωστά ένα πλαίσιο, ώστε να μην εκπλήσσονται οι προγραμματιστές που το χρησιμοποιούν Επειδή πολλές από αυτές τις αποφάσεις δεν είναι πάντα γνωστές στους προγραμματιστές (είτε λόγω απειρίας χρήσης του πλαισίου είτε λόγω ελλιπούς τεκμηρίωσης), μερικές φορές το τελικό αποτέλεσμα της λειτουργικότητας μιας εφαρμογής δεν συμπεριφέρεται όπως αναμένεται. Επιπλέον, επειδή το ίδιο το πλαίσιο αποκρύπτει τις λεπτομέρειες υλοποίησης διαφόρων λειτουργιών για να προσφέρει αφαιρέσεις στα "χαρακτηριστικά χαμηλότερου επιπέδου", λόγω της εν λόγω διαμόρφωσης ή απλώς λόγω της εγγενούς ασυμβατότητας των πραγμάτων που προσπαθεί να αφαιρέσει (ενα χαρακτηριστικό παράδειγμα αποτελεί το πρωτόκολλο TCP over IP), μερικές φορές το ίδιο το πλαίσιο μπορεί να είναι η αιτία των εκάστοτε προβλημάτων που μπορεί να προκύψουν. Το παραπάνω είναι γνωστό στην καθομιλουμένη ως "The law of leaky abstractions", όρος που επινοήθηκε από τον Joel Spolsky.[25] Η έννοια των leaky abstractions δεν είναι νέα: περιγράφηκε ήδη από το 2002 και δεν αναφερόταν αυστηρά στα πλαίσια λογισμικού, αλλά σε κάθε διαδικασία ανάπτυξης λογισμικού που εμφανίζεται η έννοια της αφαίρεσης, είτε αυτή εχει ως απώτερο σκοπό τη μοντελοποίηση, είτε την απλοποίηση πιο σύνθετων διαδικασιών. Με την ευρεία υιοθέτηση των πλαισίων, τέτοια θέματα αφαίρεσης έγιναν πιο εμφανή. Ως εκ τούτου, η ποσοτικοποίηση τέτοιων προβλημάτων μπορεί να αποδειχθεί εξαιρετικά

χρήσιμη κατά την αξιολόγηση πλαισίων λογισμικού και μπορεί να παρέχει πιο επιτυχημένες συγκρίσεις μεταξύ διαφορετικών προσεγγίσεων (πχ implicit versus explicit configuration).

1.2 Σκοπός – Στόχοι

Ο σκοπός της παρούσας εργασίας είναι να ελέγξει και να επιβεβαιώσει την υπόθεση πως η ανάπτυξη λογισμικού με frameworks προσφέρει σημαντική επιτάχυνση και διευκόλυνση στην ανάπτυξη λογισμικου. Επιπλέον, θα γίνει μια προσπάθεια να γίνει μια ποσοτικοποίηση της ωφέλειας που προκύπτει από την χρήση frameworks σε σχέση με τις πιθανές δυσκολίες που προκύπτουν από αυτό το φαινόμενο. Συγκεκριμένα, θα ακολουθηθούν τα εξής βήματα:

1. Σύντομη παρουσίαση βιβλιογραφίας σχετικής με το εξεταζόμενο ζήτημα, και επεξήγηση θεμελιωδών εννοιών απαραιτήτων για την κατανόηση του ζητήματος που εξετάζεται.
2. Υλοποίηση βασικών αφαιρέσεων μιας διαδικτυακής εφαρμογής (συνδεσιμότητα με ΒΔ, αυθεντικοποίηση) με τη βοήθεια ενός πλαισίου (για τον σκοπό αυτό επιλέχτηκε το Spring Framework που βασίζεται σε Java)
3. Υλοποίηση της ίδιας διαδικτυακής εφαρμογής χωρίς τις αφαιρέσεις που παρέχονται από τα πλαίσια λογισμικού
4. Εισαγωγή των δύο project σε κατάλληλο πρόγραμμα αξιολόγησης λογισμικού (συγκεκριμένα το Sonarqube[16]) και εξαγωγή συμπερασμάτων με βάση τις ποσοτικές μετρικές που παρέχονται από αυτό. Επιπλέον θα γίνει χρήση και ποιοτικών μετρικών (πχ. Χρόνος υλοποίησης κάθε εφαρμογής, ευκολία υλοποίησης, ύπαρξη τεκμηρίωσης, και φορές που χρειάστηκε να γίνει χρήση τεκμηρίωσης ή άλλων πηγών, πχ stackoverflow)

1.3 Διάρθρωση της μελέτης

Στο κεφάλαιο 2 παρουσιάζονται εργασίες που σχετίζονται με το ζήτημα που πραγματεύεται η παρούσα διπλωματική, και γίνεται μια σύντομη και περιεκτική αναφορά στο θεωρητικό υπόβαθρο που αποτελεί απαραίτητη γνώση για την βαθύτερη κατανόηση της εργασίας. Στο κεφάλαιο 3 γίνεται η μελέτη επί των υλοποιήσεων,

παρουσιάζεται εκτενώς η διαδικασία υλοποίησης του κάθε επιμέρους κομματιού της εφαρμογής, και στις δύο εκδοχές του (με χρησιμοποίηση πλαισίου λογισμικού, και χωρίς), και γίνεται αξιολόγηση με βάση τις μετρικές που χρησιμοποιούμε. Στο κεφάλαιο 4 γίνεται μια παρουσίαση και συγκριτικη μελέτη των μετρικών, και εξάγονται τα συμπεράσματα. Επίσης προσδιορίζονται πιθανά όρια και περιορισμοί της έρευνας.

2 Βιβλιογραφική Επισκόπηση – Θεωρητικό Υπόβαθρο

2.1 Θεωρητικό Υπόβαθρο

2.1.1 Γενικοί όροι

2.1.1.1 Software framework (Πλαίσιο λογισμικού)

Λογισμικό που παρέχει κάποιες θεμελιώδεις λειτουργίες, ώστε να διευκολύνει την δημιουργία εξειδικευμένων λογισμικών.[1] Ένα πλαίσιο λογισμικού συνήθως στοχεύει σε έναν συγκεκριμένο τομέα, π.χ. Πλαίσιο λογισμικού για ανάπτυξη εφαρμογών παγκόσμιου ιστού (web framework). Το πλαίσιο συμπληρώνεται / επεκτείνεται με λειτουργίες που υλοποιούν την λογική της συγκεκριμένης εφαρμογής.[5]

Τα πλαίσια διαφέρουν από τις κοινές βιβλιοθήκες λόγω της αρχής της αντιστροφής του ελέγχου (inversion of control). Σύμφωνα με αυτήν, ο έλεγχος της ροής του προγράμματος παραδίδεται στο πλαίσιο στην αρχή. Στη συνέχεια, το πλαίσιο καλεί κατάλληλα τις λειτουργίες που έχουν καθοριστεί (π.χ. Μετά από κάποιο συγκεκριμένο γεγονός). Η προσθήκη λειτουργιών σε ένα πλαίσιο υλοποιείται μέσω διεπαφών που ορίζει το ίδιο (πχ, σε πλαίσια λογισμικού που εφαρμόζουνε την αρχιτεκτονική MVC, η επιχειρηματική λογική περιγράφεται συνήθως στο επίπεδο του Service, ενώ η αναπαράσταση των δεδομένων που αντλούνται από τη ΒΔ γίνεται στο επίπεδο του Model)

2.1.1.2 Inversion of control

Πρόκειται για μία σχεδιαστική αρχή στην ανάπτυξη λογισμικού, σύμφωνα με την οποία η ροή της εκτέλεσης ενός προγράμματος διαφοροποιείται από τον παραδοσιακό διαδικαστικό προγραμματισμό: Ένα τυπικό διαδικαστικό πρόγραμμα αποτελείται από εξειδικευμένο κώδικα που καθορίζει την γενική ροή του προγράμματος. Η ροή ελέγχου καλεί υπορουτίνες οι οποίες υλοποιούν την προσδοκώμενη συμπεριφορά (υπολογισμοί, αποθήκευση δεδομένων κτλ) [9].

Η αντιστροφή του ελέγχου προϋποθέτει την ύπαρξη κώδικα γενικού πλαισίου (framework), ο οποίος συνήθως υπαγορεύει την ροή εκτέλεσης του προγράμματος και παρέχει βασικές επιθυμητές λειτουργίες. Ο κώδικας γενικού πλαισίου συμπληρώνεται με τις προσαρμοσμένες υπορουτίνες που υπηρετούν τον σκοπό της εφαρμογής (οι οποίες υλοποιούνται από τον προγραμματιστή)[8].

Οι εξειδικευμένες μονάδες επικοινωνούν με το framework (και επομένως μεταξύ τους) μέσω διεπαφών (contracts)[9]. Έτσι, προάγεται η ανάπτυξη σχετικά ανεξάρτητων μονάδων, οι οποίες μπορούν να αντικατασταθούν με εναλλακτικές υλοποιήσεις, και άρα διατηρείται χαμηλά η σύζευξη κλάσεων.

2.1.1.3 Dependency Injection

Η έννοια του dependency injection (DI) χρησιμοποιείται για να περιγράψει τη διαδκασία κατά την οποία οι εξαρτήσεις μιας οντότητας παρέχονται σε αυτό αντί να επαφίεται στο εκάστοτε αντικείμενο η κατασκευή τους. Συχνά, το αντικείμενο που λαμβάνει τις εξαρτήσεις ονομάζεται πελάτης και το μεταβιβασμένο ('injected') αντικείμενο ονομάζεται υπηρεσία. Ο κώδικας που είναι υπεύθυνος για την μεταβίσαση της υπηρεσίας στον πελάτη ονομάζεται injector.

Είναι μια πολύ χρήσιμη τεχνική με ευρεία εφαρμογή στα πλαίσια λογισμικού, αλλά και στις σουίτες testing, καθώς επιτρέπει δημιουργία ψευδοαντικειμένων (mock objects) .

Οι εξαρτήσεις μπορούν να εγχυθούν σε αντικείμενα με πολλούς τρόπους (όπως έγχυση μέσω κατασκευαστή ή έγχυση μέσω setter). Μπορεί κανείς να χρησιμοποιήσει ακόμη και εξειδικευμένα πλαίσια έγχυσης εξάρτησης (π.χ. Spring) για να το κάνει αυτό, αλλά σίγουρα δεν απαιτούνται. Η τεχνική του DI χρησιμοποιείται για να διατηρεί ται η σύζευξη σε χαμηλά επίπεδα, και να δίνεται η δυνατότητα πολλαπλών υλοποιήσεων μιας διεπαφής, αναλόγως με τις ανάγκες της εκάστοτε εφαρμογής.

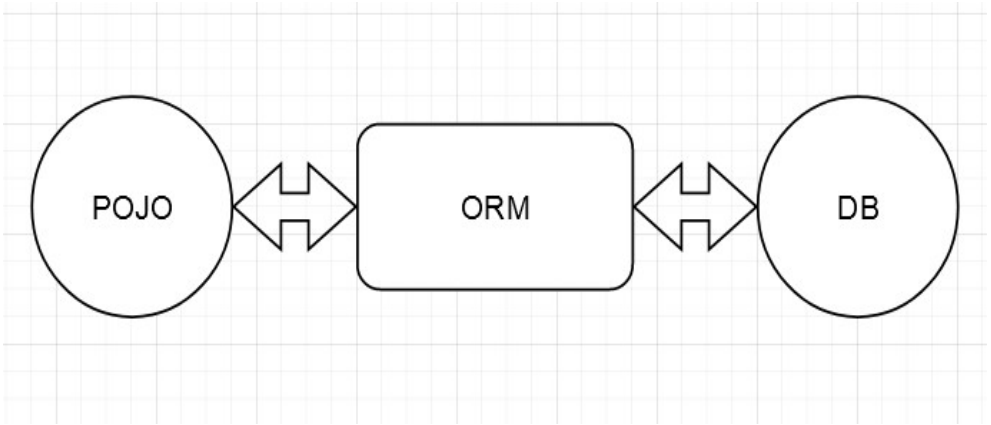
2.1.1.4 Database (ΒΔ)

Ηλεκτρονικός χώρος οργάνωσης, αποθήκευσης και αναζήτησης σχετιζόμενων (ή και όχι) δεδομένων. Χρησιμοποιείται για την αποτελεσματική αποθήκευση και διαχείριση ενός μεγάλου όγκου δεδομένων, τα οποία βρίσκονται συνήθως στην δευτερεύουσα μνήμη.[11] Επιπλέον, μπορεί να παρέχει λειτουργίες όπως ασφάλεια ευαίσθητων δεδομένων (π.χ. hashes κωδικών). Ο χρήστης μιας βάσης αλληλεπιδρά μαζί της μέσω ενός Συστήματος Διαχείρισης Βάσης Δεδομένων (DBMS). Ένα τέτοιο σύστημα παρέχει, μεταξύ άλλων, δυνατότητες προσθήκης, ανάκτησης, ενημέρωσης και διαγραφής δεδομένων (CRUD). Οι ενέργειες αυτές αποκαλούνται ερωτήματα (queries) και συνήθως συντάσσονται σε κάποια προσαρμοσμένη γλώσσα προγραμματισμού (DSL) όπως η SQL.

2.1.1.5 ORM (Object-relational Mapping)

Προγραμματιστική τεχνική που χρησιμοποιείται για να επιτρέπει την απεικόνιση σχεσιακών δεδομένων σε αντικείμενα γλωσσών προγραμματισμού. Στην ουσία, γίνεται αντιστοίχηση κάθε οντότητας μιας ΒΔ σε μια έννοια αντικειμενοστραφούς προγραμματισμού (πχ. Ένας πίνακας μιας ΒΔ αντιστοιχείται σε μια κλάση αντικειμένου, και κάθε εγγραφή ενός πίνακα σε ένα στιγμιότυπο μιας κλάσης).[23]

Το ORM είναι από τις βασικότερες λειτουργικότητες που προσφέρουν τα πλαίσια λογισμικού, καθώς δίνει την δυνατότητα διαχείρισης ΒΔ μέσα από το ίδιο το πλάισιο, και (στις περισσότερες των περιπτώσεων) χωρίς να δημιουργείται η ανάγκη συγγραφής SQL Επιπλέον, το επίπεδο αφαίρεσης που παρέχεται, επιτρέπει στον προγραμματιστή να χρησιμοποιεί τις «λογικές προεπιλογές» (sensible defaults) όσον αφορά τεχνικά ζητήματα που ενδεχομένως να απαιτούσαν τη συνδρομή διαχειριστή ΒΔ σε διαφορετική περίπτωση (π.χ. Isolation/Propagation levels, Indexing, execution plans).

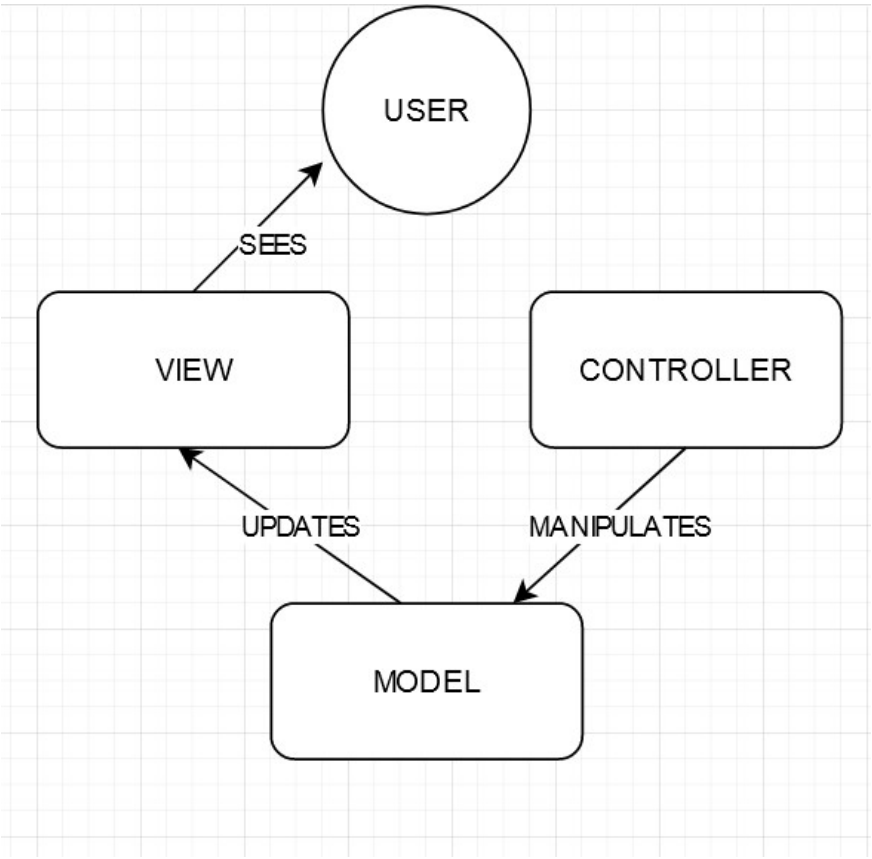


Εικόνα 2-: Λειτουργια ORM

2.1.1.6 MVC (Model-View-Controller)

Το MVC (Model-View-Controller) είναι μια αρχιτεκτονική που χρησιμοποιείται στο σχεδιασμό λογισμικού για την υλοποίηση διεπαφών χρήστη, δεδομένων και λογικής ελέγχου. Δίνει έμφαση στον διαχωρισμό της ανάπτυξης της εφαρμογής σε τρία επιμέρους στοιχεία (μοντέλο, ελεγκτής, όψη). Εκτός από τη διαίρεση της εφαρμογής, ο σχεδιασμός MVC ορίζει επίσης τις αλληλεπιδράσεις μεταξύ τους.[6]

Το MVC είναι ένα από τα πιο συχνά χρησιμοποιούμενα βιομηχανικά πρότυπα ανάπτυξης ιστοσελίδων για τη δημιουργία επεκτάσιμων και επεκτάσιμων έργων.



Εικόνα 2-: Δείγμα αρχιτεκτονικής MVC

2.1.1.7 Τεχνικό χρέος

Στα συστήματα λογισμικού, το τεχνικό χρέος είναι μια συλλογή υλοποιήσεων ή σχεδιασμών που είναι συμφέρουσες σε βραχυπρόθεσμα πλαίσια, αλλά δημιουργούν ένα που μπορεί να κάνει τις μελλοντικές αλλαγές πιο δαπανηρές ή αδύνατες. Το τεχνικό χρέος αντικατοπτρίζει μία πραγματική ή ενδεχόμενη υποχρέωση των προγραμματιστών απέναντι στην περάτωση ενός έργου λογισμικού, η οποία κοστίζει σε χρονικά και οικονομικά πλαίσια, και αποτελεί τον βασικότερο τρόπο συσχέτισης της ανάπτυξης λογισμικού με το οικονομικό χρέος.[13] Ως παράδειγμα, μπορούμε να θεωρήσουμε ότι κατα τη διάρκεια ανάπτυξης μιας εφαρμογής, προκύπτουνε δύο πιθανές υλοποιήσεις για κάποιο χαρακτηριστικό του έργου. Η πρώτη ενδεχομένως να μη χρειάζεται πολύ χρόνο στην υλοποίηση της σίγουρα (άρα προσφέρει βραχυπρόθεσμο κέρδος), αλλά σίγουρα στο μέλλον θα αποδειχθεί εξαιρετικά χρονοβόρα η διόρθωση της. Η δέυτερη έχει καλύτερο σχεδιασμό, και μακροπρόθεσμα θα είναι πιο αποδοτική, αλλά κοστίζει ιδιαιτέρως σε χρόνο. Επιλέγοντας την πρώτη από τις δύο προσεγγίσεις ενδεχομένως να επιφέρει βραχυπρόθεσμο κέρδος, καθώς συμβάλλει στην επίτευξη στόχων και στην

ικανοποίηση του πελάτη, αλλά ταυτόχρονα δημιουργεί μια υποχρέωση της ομάδας ανάπτυξης να ξαναεπισκεφτεί στο μέλλον την συγκεκριμένη υλοποίηση ώστε να τη βελτιώσει. Αυτή η υποχρέωση αποτελεί το λεγόμενο τεχνικό χρέος. Η έννοια αυτή δεν έχει αφ'εαυτής αρνητική σημασία, αλλά υπάρχει μόνο για να προσδιορίζει ποσοτικά τις υλοποιήσεις που γίνονταν με πιο βραχυπρόθεσμο στόχο. Κατα τα λεγόμενα του εμπνευστή του όρου Ward Cunningham : "[…] *A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt […]"*[13]

Ο όρος επινοήθηκε απο τον Ward Cunningham το 1992, και χρησιμοποιείται ευρέως ως μετρική στις διάφορες μεθόδους αξιολόγησης ποιότητας λογισμικού έκτοτε.

2.1.2 Μετρικές

2.1.2.1 Maintainability Rating

Γνωστή και ώς SQALE rating. Βαθμολογία που δίνεται σε ένα ολόκληρο έργο λογισμικού και σχετίζεται με την τιμή της αναλογίας τεχνικού χρέους.[18] Το προεπιλεγμένο πλέγμα καθορίζεται ως εξής :

A=0-0.05, B=0.06-0.1, C=0.11-0.20, D=0.21-0.5, E=0.51-1

Μια εναλλακτική διατύπωση για την κλίμακα αξιολόγησης συντηρησιμότητας ορίζεται βρίσκοντας το χρονικό κόστος που χρειάζεται να πληρωθεί σε σχέση με το συνολικό χρονικό κόστος που έχει πληρωθεί μέχρι τώρα. Συγκεκριμένα, εαν το κόστος που χρειάζεται να κατατεθεί είναι:

- <=5% του χρόνου που έχει ήδη διατεθεί στην εφαρμογή,τότε η βαθμολογία είναι A
- Ανάμεσα σε 6 και 10% τότε η βαθμολογία είναι B
- Ανάμεσα σε 11 και 20% τότε η βαθμολογία είναι C
- Ανάμεσα σε 21 και 50% τότε η βαθμολογία είναι D
- Πάνω από 50% τότε η βαθμολογία είναι E

2.1.2.2 Cyclomatic Complexity

Κυκλωματική πολυπλοκότητα (Cyclomatic Complexity) ονομάζεται η μετρική η οποία υποδεικνύει την σταθερότητα και το επίπεδο εμπιστοσύνης προς τον κώδικα με

την έννοια της επιτυχημένης λειτουργίας του προγράμματος. Υπολογίζει στα κατά τόπους κομμάτια του κώδικα τον αριθμό των γραμμικών και ανεξάρτητων διαδρομών που μπορούν να ακολουθηθούν.[7]

Μπορεί να αποτυπωθεί πάνω σε ένα διάγραμμα ελέγχου ροής του προγράμματος και υπολογίζεται με βάση τα σημεία ελέγχου ροής (control flow statements). Ο βαθμός πολυπλοκότητας καθορίζεται από το πλήθος των κορυφών που δημιουργούνται από τις ανεξάρτητες, γραμμικές διαδρομές που μπορούν να ακολουθηθούν, από το πλήθος των σημείων ελέγχου που χρειάζεται να προσπελάσει η εκάστοτε διαδρομή και από τον πλήθος των συνδεδεμένων συνιστωσών.

Βάσει αυτών, η φόρμουλα υπολογισμού είναι η εξής: M = E – N + 2P, όπου E ο αριθμός των κορυφών, N ο αριθμός των σημείων ελέγχου και P ο αριθμός των συνδεδεμένων συνιστωσών.

2.1.2.3 Cognitive Complexity

Η γνωστική πολυπλοκότητα (Cognitive Complexity) είναι μια μετρική που χρησιμοποιείται από την πλατφόρμα αξιολόγησης έργων λογισμικού Sonarqube.[24] Έχει αναπτυχθεί με σκοπό να παρέχει μια πιο ολοκληρωμένη εικόνα όσον αφορά την πολυπλοκότητα ενός έργου λογισμικού, αφ'ενός λαμβάνοντας υπ'όψιν περιπτώσεις που δεν καλύπτει η κυκλωματική πολυπλοκότητα (όπως π.χ. η δομή try/catch), και αφ'ετέρου παράγοντας τιμές και δεδομένα που έχουνε νόημα σε επίπεδα κλάσης και εφαρμογής (εν αντιθέσει με την κυκλωματική πολυπλοκότητα, που λόγω του τρόπου υπολογισμού της δεν έχει νόημα σε επίπεδο πέραν της μεθόδου).

Το πλέον σημαντικό της πλεονέκτημα όμως έγκειται στο γεγονός ότι λόγω του τρόπου υπολογισμού της, προσφέρει δυνατότητα εκτιμήσεων όσον αφορά την ευκολία κατανόησης μιας εφαρμογής.[24] Κατ'επέκταση, μπορεί να συμβάλλει στην ποσοτικοποίηση του χρόνου που χρειάζεται είτε για την αποσφαλμάτωση, είτε για την συγγραφή καινούργιων χαρακτηριστικών, είτε για την εισαγωγή καινούργιων μελών στην ομάδα συγγραφής κώδικα για το έργο.

2.1.2.4 Μετρικές Chidamber & Kemerer

Σουίτα από μετρικές που προτάθηκε πρώτη φορά από τους Chidamber & Kemerer, και χρησιμοποιείται ευρέως στην αξιολόγηση έργων λογισμικού. Οι έξι μετρικές που περιλαμβάνονται σε αυτήν είναι :

- WMC – Weighted Methods Per class (Μέθοδοι ανά κλάση)
- CBO – Coupling between object classes (Σύζευξη μεταξύ αντικειμένων κλασεων)
- RFC – Response for a class (Απόκριση για μια κλάση)
- NOC – Number of children (Αριθμός παιδιών)
- DIT – Depth of inheritance tree (Βάθος δέντρου κληρονομικότητας)
- LCOM – Lack of cohesion methods (Έλλειψη συνοχής μεθόδων)

Η μετρική WMC μετράει τον αριθμό των μεθόδων που είναι δηλωμένες σε μια κλάση.[23] Η μετρική αυτή εκτός του ότι βοηθάει στο να ποσοτικοποιεί μία έκφανση του μεγέθους μιας κλάσης, χρησιμεύει και για να βοηθήσει στην πρόβλεψη του πόσο χρόνο θα χρειαστεί στη συντήρηση και ανάπτυξη μιας κλάσης. Οι κλάσεις που έχουνε υψηλή τιμή σε αυτή τη μετρική είναι καλό να αναλύονται ώστε να μελετηθεί η πιθανότητα διαχωρισμού τους σε μικρότερες υποκλάσεις.

Η μετρική CBO χρησιμοποιείται σε επίπεδο κλάσεων, και δείχνει τον αριθμό κλάσεων που είναι συζευγμένη μια κλάση (είτε που τις χρησιμοποιεί, είτε χρησιμοποιείται από αυτές) .[22] Η μετρική αυτή πρέπει να διατηρείται όσο το δυνατόν περισσότερο σε χαμηλά επίπεδα, καθώς είναι προφανές πως εαν μια εφαρμογή έχει υψηλή τιμή σε αυτή τη μετρική, έχει και πολύ υψηλή πολυπλοκότητα (καθώς μια αλλαγή σε μια κλάση ενδεχομένως να επηρεάζει αρκετές άλλες), και καθίσταται δύσκολη η ανάπτυξη νέων χαρακτηριστικών σε αυτήν.

Η τιμή της μετρικής RFC καθορίζεται από τον αριθμό των μεθόδων που πιθανόν να εκτελεστούν, σε απάντηση ενός μηνύματος που μπορεί να λάβει ένα αντικείμενο μιας κλάσης.[22] Κλάσεις με υψηλή τιμή σε αυτή την μετρική είναι πιο πολύπλοκες στην κατανόηση τους, και κατ'επέκταση κάνουν το έργο της αποσφαλμάτωσης και της προσθήκης χαρακτηριστικών πιο δύσκολα.

Η μετρική NOC ορίζεται ως ο αριθμός των άμεσων υποκλάσεων μιας κλάσης. Χρησιμοποιείται συχνά σε συνδυασμό με την μετρική DIT, καθώς μελετούνε και οι δύο ζητήματα κληρονομικότητας, αλλά κάτω από διαφορετικό πρίσμα. Κλάσεις με υψηλή τιμή NOC θεωρούνται καλές, καθώς αυτό σημαίνει ότι έχουνε πάρα πολλές διαφορετικές

υποκλάσεις, και άρα αποτελούνε ένδειξη ότι ο κώδικας τους επαναχρησιμοποιείται σε μεγάλο βαθμό.[22]

Όσον αφορά την DIT, είναι μια μετρική που υποδηλώνει το βάθος κληρονομικότητας μιας κλάσης, δηλαδή τον αριθμό υπερκλάσεων από τις οποίες κληρονομεί.[22] Η λογική πίσω από αυτή την μετρική έγκειται στην υπόθεση ότι όσο πιο πολλές κλάσεις έχει ως προγόνους μια κλάση, τόσο μεγαλύτερος είναι ο αριθμός των μεθόδων που μπορεί να κληρονομεί, και άρα είναι πιο δύσκολο να προβλεφθεί η συμπεριφορά της. Επιπλέον, επειδή μεγάλος αριθμός DoI σημαίνει ότι εμπλέκονται πολλές κλάσεις, πρακτικά μπορούμε να συμπεράνουμε ότι μεγαλύτερος αριθμός στην μετρική αυτή συνεπάγεται και πιο πολύπλοκη εφαρμογή.

Η μετρική LCOM χρησιμοποιείται για να μετρήσει τη συνοχή μιας κλάσης. Υπολογίζεται παίρνοντας κάθε ζευγάρι μεθόδων σε μία κλάση , και αναλόγως με το αν υπάρχουν ή όχι κοινές μεταβλητές στις οποίες έχουνε πρόσβαση, τότε αυτος ο αριθμός αυξάνεται ή μειώνεται (με ελάχιστη τιμή το 0). Εάν μια κλάση έχει τιμή παραπάνω από 0, τότε θεωρητικά είναι καλό να σπάσει σε διαφορετικές κλάσεις (ακολουθώντας και την αρχή μοναδικής ευθύνης – Single responsibility principle) .[22]

3 Μεθοδολογία

Γενικά

Για τις ανάγκες της εργασίας υλοποιήθηκε μια εφαρμογή που χρησιμεύει ως API server ώστε να απαντάει σε Http αιτήματα. Από επιχειρηματικής σκοπιάς, επιλέχθηκε μια δοκιμαστική ΒΔ που έχει αναπτυχθεί κάτω από άδεια ανοιχτου λογισμικού, με σκοπό να χρησιμοποιείται για benchmarking. Η εφαρμογή λοιπόν υλοποιήθηκε δύο φορές, μια φορά με την χρήση ενός πλαισίου λογισμικού, συγκεκριμένα του spring framework, και μία φορά χρησιμοποιώντας μόνο τη βασικότερη εκδοχή ενός Http Server που παρέχεται από το πακέτο com.sun.net.httpserver. Η πρώτη εκδοχή μάλιστα χρησιμοποιεί ένα πακέτο που έχει φτιαχτεί για το Spring, ονόματι spring-boot, το οποίο θεωρείται μια προέκταση του spring, και ουσιαστικά εξαλείφει αρκετές από τις ανάγκες υλοποίησης που έχει από μόνη της η χρήση του Spring.

Πιο συγκεκριμένα, με τη χρήση του Spring-boot και των πακέτων που παρέχονται από αυτό, γίνονται κάποιες επιλογές που αφορούν διάφορα ζητήματα της εφαρμογή πχ ποιος Web server θα χρησιμοποιηθεί (η αρχική επιλογή είναι ο tomcat server), πως γίνεται η υλοποίηση της αυθεντικοποίησης (αναλύεται περαιτέρω στο κεφάλαιο 3.4), και ποιο templating engine χρησιμοποιείται (στην περίπτωση μας δεν έχει καμία πρακτική διαφορά αυτό, καθώς η εφαρμογή δεν προορίζεται να έχει κάποιο γραφικό περιβάλλον διεπαφής). Επίσης, επειδή το spring-boot χρησιμοποιεί το Jpa, κι επειδή η ανάπτυξη της εφαρμογής έγινε με το IntelliJ IDEA (Ultimate edition)[21], χρησιμοποιήθηκε το jpaBuddy[20], plugin που κάνει αυτόματα την μοντελοποίηση των οντοτήτων σε Java, διαβάζοντας τη ΒΔ. Το plugin αυτό συνέβαλλε σημαντικά στην μείωση του χρόνου υλοποίησης, αλλά -όπως θα φανεί και αργότερα- συνέβαλλε στην αύξηση του διπλότυπου κώδικα (duplicate code). Τα αίτια αναλύονται διεξοδικά στο κεφάλαιο 4

Από την άλλη, με τη χρήση του πακέτου com.sun.net.httpserver, δεν υπάρχει κάποια αυτόματη ρύθμιση όσον αφορά τα ζητήματα που θίχθηκαν παραπάνω, κάτι που σημαίνει ότι για κάθε χαρακτηριστικό της εφαρμογής που προστίθεται, θα πρέπει να γίνεται και υλοποίηση του εξ'αρχής.

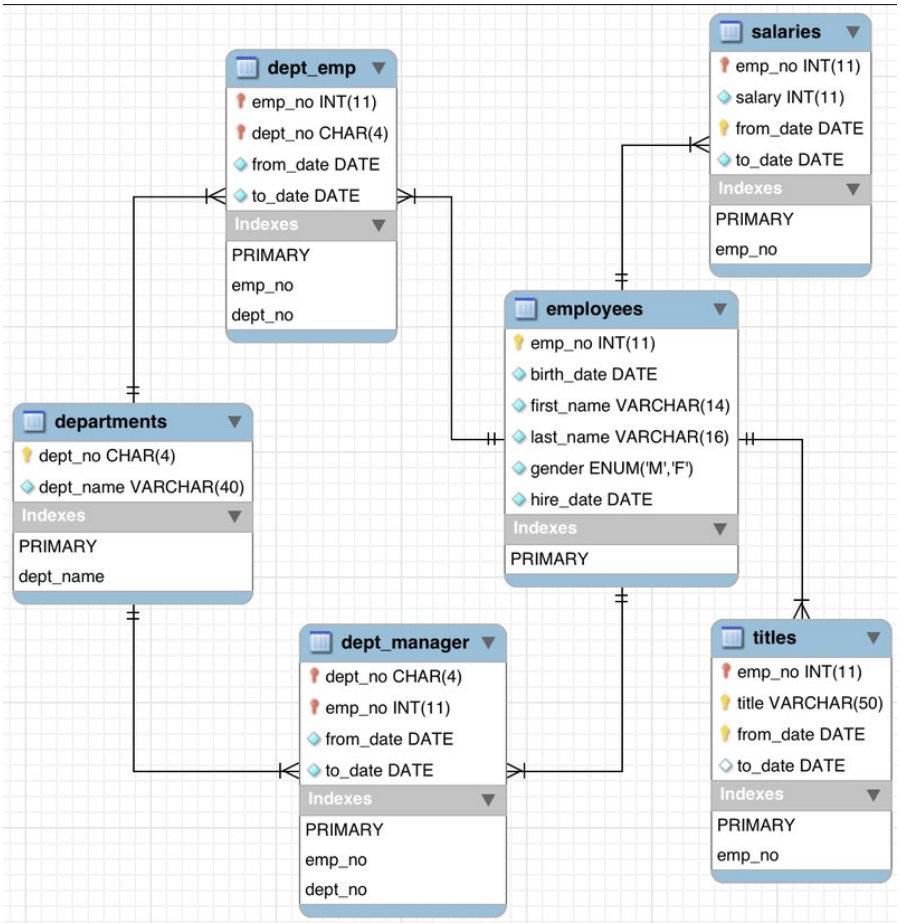
Με βάση τα παραπάνω λοιπόν, αφού υλοποιηθούν και οι δύο εκδοχές της εφαρμογής, θα εισαχθεί ο πηγαίος κώδικας σε αντίστοιχο λογισμικό αξιολόγησης (συγκεκριμένα, χρησιμοποιείται το Sonarqube), και θα γίνει μια συγκριτική ανάλυση διαφόρων μετρικών (ποσοτικών και ποιοτικών), η οποία θα περιγραφεί στο κεφάλαιο 4.

3.1.1 Οργάνωση της ΒΔ

Για τις ανάγκες τις εργασίας χρησιμοποιήθηκε η βάση δεδομένων “Employees Sample Database”, που έχει αναπτυχθεί ειδικά για να χρησιμοποιείται ως εργαλείο δοκιμής και ανάπτυξης εφαρμογών και εξυπηρετητών ΒΔ. Η βάση δεδομένων αποτελείται από 6 πίνακες, 4 κύριους (employees, salaries, departments, titles) και 2 βοηθητικούς πίνακες που χρησιμοποιούνται για να γίνει η απεικόνιση των σχέσεων πολλά-προς-πολλά. Συγκεκριμένα, στον πίνακα ”employee” μοντελοποιούνται τα στοιχεία των υπαλλήλων που χρησιμοποιούμε στην εργασία. Στον πίνακα “departments” περιγράφονται τα τμήματα στα οποία μπορεί να δουλέυει κάποιος employee, ενώ στον πίνακα titles οι τίτλοι που μπορεί να έχει ένας υπάλληλος. Τέλος, στον πίνακα salaries αποτυπώνονται οι μισθοί των υπαλλήλων.

Ο πίνακας dept_emp είναι υπεύθυνος για τη σύζευξη των πινάκων departments, employees, με σύνθετο κύριο κλειδί τον συνδυασμο των κύριων κλειδιών των δύο πινάκων (dept_no, emp_no). Ο πίνακας dept_manager ακολουθεί παρόμοια λογική, χρησιμοποιώντας τα ίδια πεδία ως κλειδιά.

Στο παρακάτω σχεσιακό διάγραμμα (ER Diagram) φαίνεται αναλυτικά η διάρθρωση της βάσης δεδομένων:



Εικόνα 3-1: Η διάρθρωση της ΒΔ

Η βάση αυτή επιλέχθηκε αφ'ενός διότι τα δοκιμαστικά δεδομένα που περιέχει είναι αρκετά (στο σύνολο τους, κυμαίνονται περίπου στις 4000000 εγγραφές) ώστε να αποτελούνε ικανοποιητικό δείγμα για μετρήσεις, αφ'ετέρου δε γιατί παρέχουνε έννοιες που είναι χρήσιμο να μελετηθούνε ως προς την μοντελοποίηση, και την υλοποίηση τους στις δύο εκδόσεις της εφαρμογής. Πιο συγκεκριμένα η μοντελοποίηση των σύνθετων κλειδιών, αλλά και των σχέσεων πολλά-προς-πολλά που εμφανίζονται στη βάση χρησιμεύει ιδιαιτέρως στο να αναδείξει τις διαφορές στην προσέγγιση και υλοποίηση στις δύο εκδοχές της διαδικτυακής εφαρμογής. Η άδεια χρήσης της ΒΔ είναι η "creative Commons Attribution-Sharealike 3.0 Unported License"

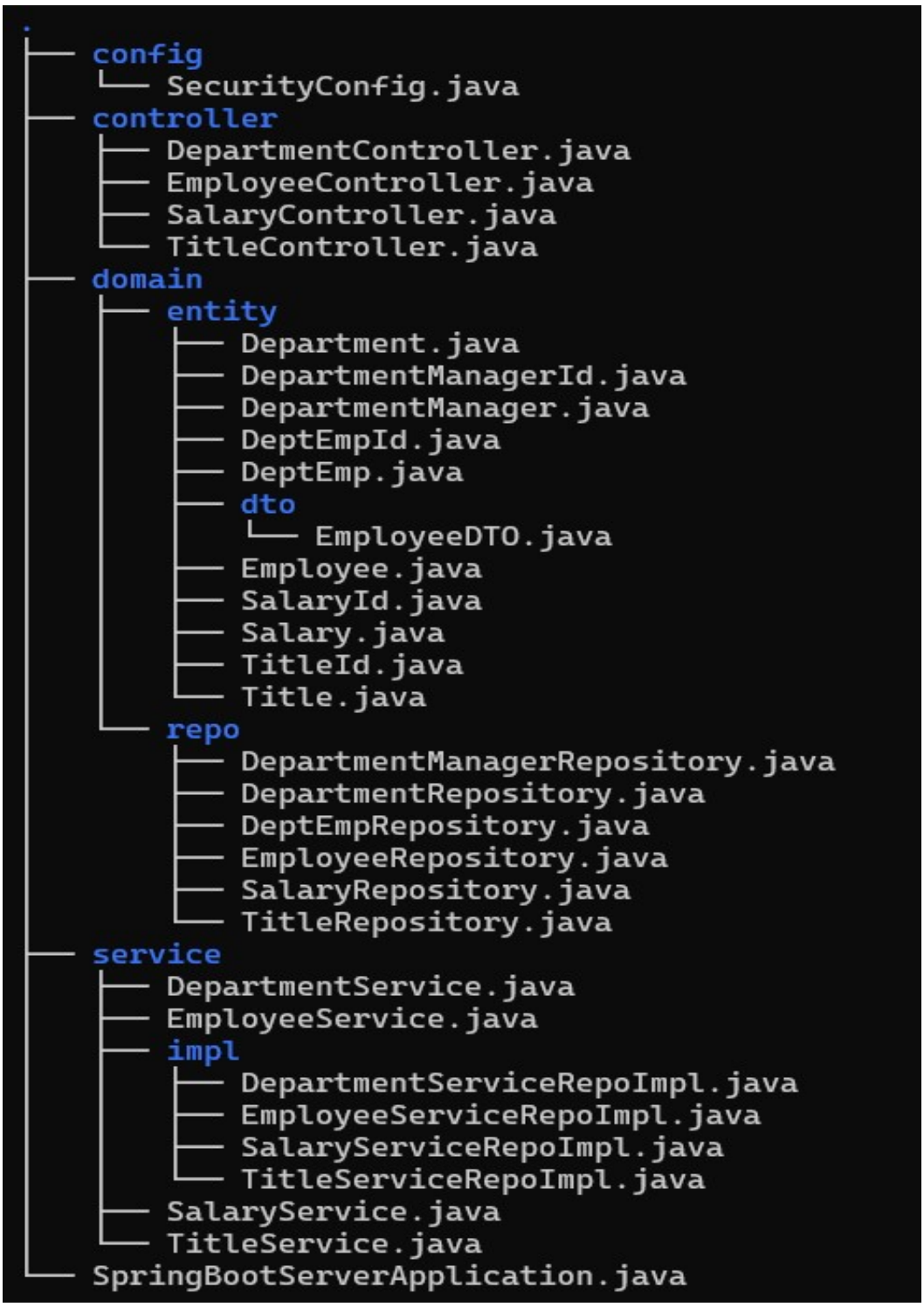
3.1.2 Διαμόρφωση εφαρμογής με το Spring-Boot

Η διάρθρωση της εφαρμογής ακολουθεί την μία από τις προτεινόμενες προσεγγίσεις όσον αφορά την υλοποίηση διαδικτυακών εφαρμογων με το Spring, κατά

την οποία τα πακέτα της εφαρμογής οργανώνονται αναλόγως με τα επίπεδα της εφαρμογής, όπως αυτά ορίζονται από το αρχιτεκτονικό πρότυπο “MVC”. Συγκεκριμένα, για κάθε ξεχωριστή οντότητα δημιουργούνται αντίστοιχα κλάσεις για τα παρακάτω:

- Controllers: (Χρησιμοποιείται για να εξυπηρετεί τα HTTP αίτηματα που γίνονται στον διακομιστή, ώστε να ανακατευθύνει το κάθε αίτημα στο αντίστοιχο Service (όπου εκεί εφαρμόζεται το Business logic της εφαρμογής)
- Entities: Σε αυτές τις κλάσεις μοντελοποιείται σε Java αντικείμενο ο κάθε πίνακας, ώστε να γίνεται ευκολότερη η διαχείριση της κάθε εγγραφής στην εφαρμογή. Στην προσέγγιση αυτή, τα σύνθετα κλειδιά του κάθε πίνακα αντιστοιχούνε σε μια ξεχωριστή οντότητα που ενσωματώνεται στην κύρια οντότητα με το annotation @EmbeddedId.
- Repositories: Οι κλάσεις αυτές χρησιμοποιούνται για να ενθυλακώσουνε λειτουργικότητα που αφορά την ανάκτηση, αποθήκευση, και αναζήτηση από τη βάση δεδομένων. Στις κλάσεις αυτές εντοπίζεται σημαντικό μέρος λογικών αφαιρέσεων, καθώς με αυτή την προσέγγιση εξαλείφεται εντελώς η ανάγκη για συγγραφή εντολών SQL.
- DTO: Αυτές οι κλάσεις χρησιμοποιούνται για την ενθυλάκωση των οντοτήτων της ΒΔ ώστε να διευκολύνεται η διεπαφή μεταξύ της εφαρμογής και των προγραμμάτων-πελατών.
- Services: Σε αυτές τις κλάσεις εφαρμόζεται η επιχειρηματική λογική (Business Logic). Επιπλέον σε αυτό το επίπεδο χρησιμοποιούνται οι διεπαφές, ώστε να υπάρχει ευκολα η δυνατότητα της επεκτασιμότητας στην εφαρμογή, κι επιπλέον για να αποφεύγεται το tight coupling μεταξύ των επιμέρους κομματιών της εφαρμογής.

Επιπλέον, στην εφαρμογή υπάρχει το πακέτο που χρησιμοποιείται για την υλοποίηση συστήματος αυθεντικοποίησης. Στο παρακάτω σχήμα φαίνεται η διάρθρωση της εφαρμογής:



Εικόνα 3-2 : Η διάρθρωση της εφαρμογής με την υλοποίηση του Spring-boot

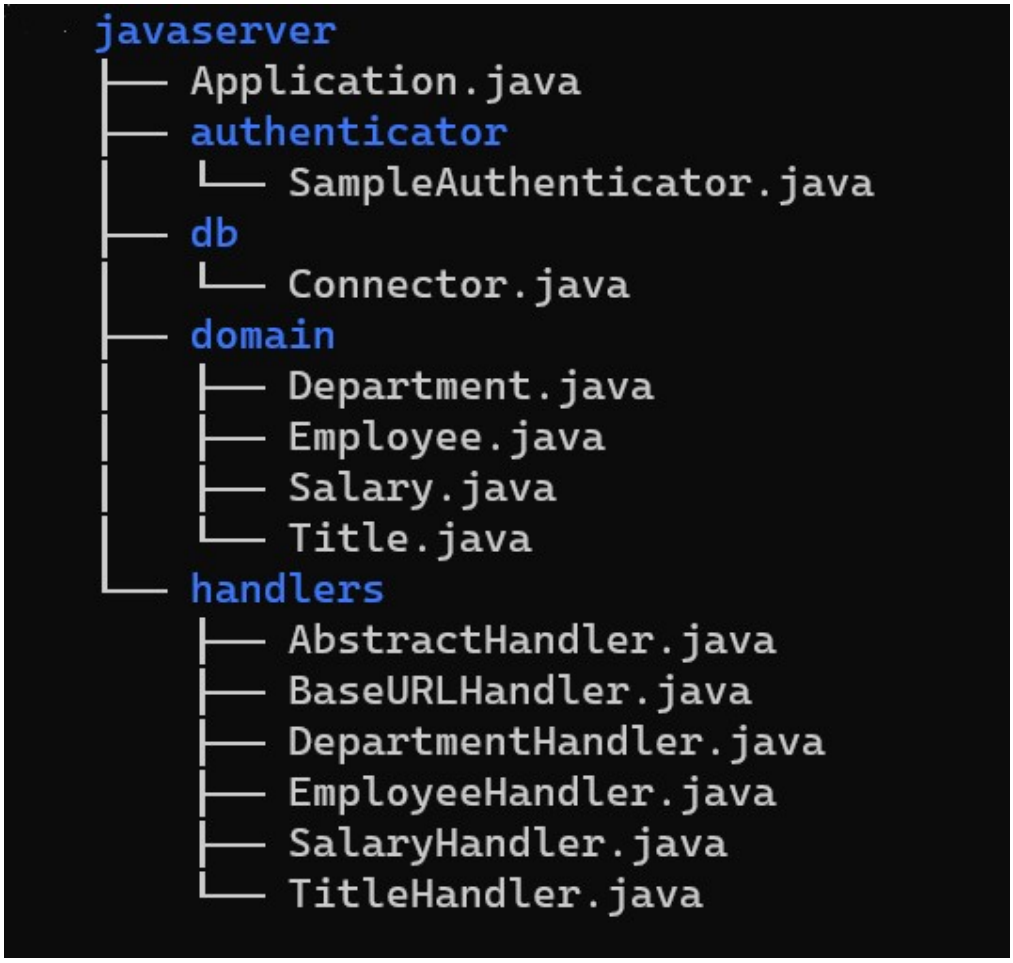
Ο πηγαίος κώδικας της υλοποίησης βρίσκεται στη σελίδα https://github.com/AlcBrains/spring-boot-webserver

3.1.3 Διαμόρφωση εφαρμογής ιδιότυπη υλοποίηση

Η διάρθρωση της εφαρμογής σε αυτή τη μορφή της, διαφέρει από αυτήν της υλοποίησης με spring boot, όχι μόνο σε επίπεδο οργάνωσης κώδικα, αλλά και στο θεμελιώδες κομμάτι του τρόπου συγγραφής κώδικα, καθώς δεν υπάρχει η έννοια του Inversion of Control. Επειδή ακριβώς δεν υπάρχει αυτός ο περιορισμός στη συγγραφή του κώδικα, κάθε εφαρμογή έχει και διαφορετική διάρθρωση. Στην συγκεκριμένη περίπτωση η προσέγγιση γίνεται με βάση τις οντότητες που διαχειρίζεται η εφαρμογή και όχι με αντίστοιχους διαχωρισμούς όπως αυτούς της υλοποίησης με spring boot. Εδώ απλώς υπάρχει για κάθε οντότητα και ένας HttpHandler που «ακούει» για http αιτήματα σχετικά με την οντότητα, και αναλόγως τον τύπο του αιτήματος (GET, POST, PUT, κλπ.) αντίστοιχα εφαρμόζει επιχειρηματική λογική για τις λειτουργίες ανάκτησης, διαγραφής, ενημέρωσης, και αποθήκευσης εγγραφών. Οι οντότητες μοντελοποιούνται κάτω από το πακέτο “domain”, και έχουνε σαφώς πιο απλή υλοποίηση από την προηγούμενη προσέγγισή (πχ. δεν απεικονίζεται κάπου η έννοια του σύνθετου κλειδιού για τους πίνακες που έχουνε τέτοια ιδιαιτερότητα).

Επειδή είναι απαραίτητη η υλοποίηση διεπαφής για πρόσβαση στην βάση δεδομένων, έχει γίνει η δημιουργία της αντίστοιχης κλάσης ονόματι Connector.java η οποία παρέχει ένα αντικείμενο που κάνει ακριβώς αυτή τη δουλειά.

Τέλος, για την υλοποίηση του συστήματος αυθεντικοποίησης έχει γίνει υλοποίηση της κλάσης SampleAuthenticator.java που είναι υπεύθυνη για την αυθεντικοποίηση των χρηστών. Επειδή η εφαρμογή προορίζεται για API server (και άρα δεν έχει κάπου γραφικό περιβάλλον για είσοδο χρηστών) η διαδικασία της αυθεντικοποίησης γίνεται με Bearer Token όπως αυτή περιγράφεται στο αντίστοιχο RFC[12]. Στο παρακάτω σχεδιάγραμμα απεικονίζεται η διάρθρωση της εφαρμογής:



Εικόνα 3-3 : Η διάρθρωση της εφαρμογής με την ιδιότυπη υλοποίηση

Τέλος, για τις λεπτομέρειες υλοποίησης που αφορούν τον ίδιο τον Server, για τους σκοπούς της εργασίας στον server ανατέθηκαν 2 νήματα για επεξεργασία αιτημάτων, και επιλέχθηκε η θύρα 9000 για τη λειτουργία.

```java
public class Application {
    private static final int THREAD_COUNT = 2;
    private static final int PORT_NO = 9999;


    public static void main(String[] args) throws IOException {

        HttpServer server = HttpServer.create(new InetSocketAddress(PORT_NO),  backlog: 0);

        List<HttpContext> contextArrayList = new ArrayList<>();

        contextArrayList.add(server.createContext( path: "/", new BaseURLHandler()));
        contextArrayList.add(server.createContext( path: "/employees", new EmployeeHandler()));
        contextArrayList.add(server.createContext( path: "/departments", new DepartmentHandler()));
        contextArrayList.add(server.createContext( path: "/salaries", new SalaryHandler()));
        contextArrayList.add(server.createContext( path: "/titles", new TitleHandler()));

        for (HttpContext context: contextArrayList) {
            context.setAuthenticator(new SampleAuthenticator( realm: "test"));
        }

        Executor executor = Executors.newFixedThreadPool(THREAD_COUNT);

        server.setExecutor(executor);
        server.start();

        System.out.println("server started at " + PORT_NO);

    }
}
```

Εικόνα 3-4 : Η αρχική κλάση που δηλώνεται ο server και οι αντίστοιχοι handlers, στην υλοποίηση χωρίς τη βοήθεια πλαισίου

Ο πηγαίος κώδικας της υλοποίησης βρίσκεται στη σελίδα: https://github.com/AlcBrains/simple-http-webserver

3.2 Διασύνδεση με ΒΔ

3.2.1 Με τη χρήση του Spring boot

Η διασύνδεση με τη χρήση του Spring-boot γίνεται χρησιμοποιώντας το πακέτο `spring-boot-starter-data-jpa`, για την παροχή των διεπαφών με τη βάση δεδομένων. Το

πακέτο αυτό περιλαμβάνει όλη τη λειτουργικότητα που χρειάζεται για τη σωστή διασύνδεση, ξεκινώντας από το πρόγραμμα-οδηγό (driver) για τη διασύνδεση με τη ΒΔ, (στην περίπτωση μας MySQL), και καταλήγοντας στην υλοποίηση βασικών μεθόδων για ανάκτηση ή/και επεξεργασία δεδομένων για την κάθε οντότητα. Σε πρακτικό επίπεδο αυτό σημαίνει ότι το μόνο που χρειάζεται να γίνει από την πλευρά του προγραμματιστή είναι να δημιουργηθεί ένα Interface αρχείο, με το @Repository annotation, που να κληρονομεί από το JpaRepository, δηλώνοντας παράλληλα το όνομα της οντότητας, και το κύριο της κλειδί. Για παράδειγμα, στην περίπτωση του πίνακα employee, η υλοποίηση είναι ως εξής:

```
@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {
}
```

Εικόνα 3-5: Δείγμα από κλάση τύπου Repository, υπεύθυνη για ανάκτηση/επεξεργασία δεδομένων ΒΔ

Αυτή η κλάση μπορεί πλέον να εισαχθεί ως dependency σε σημεία που εφαρμόζουμε επιχειρηματική λογική (πχ. σε κάποιο service) και παρέχει όλες τις βασικές λειτουργίες που αφορούνε την ανάκτηση δεδομένων πχ., αναζήτηση όλων των εγγραφών με την μέθοδο υπερκλάσης findAll()) ή για την αποθήκευση μιας καινούργιας ή υπάρχουσας εγγραφής με την μέθοδο save() (στο επίπεδο συγγραφής κώδικα δεν γίνεται διαχωρισμός ανάμεσα σε create και update, καθώς η υλοποίηση της διεπαφής φροντίζει για το συγκεκριμένο ζήτημα), καθώς επίσης και για τη διαγραφή μιας εγγραφής με τη μέθοδο delete().

Ένα μεγάλο πλεονέκτημα που αναφέρθηκε και πρωτύτερα είναι ότι η χρήση του Spring data jpa μας δίνει τη δυνατότητα να κάνουμε και ερωτήματα με βάση συγκεκριμένη στήλη πίνακα, ή συδυασμό στηλών, να θέτουμε κριτήρια αναζήτησης αναλόγως τον τύπο των στηλών που γίνεται η αναζήτηση (πχ. LIKE σε πεδία με χαρακτήρες) , και να βάζουμε κριτήρια ταξινόμησης (OrderBy), χωρίς να χρειάζεται να γράψουμε καθόλου SQL. Η ονομασία της μεθόδου που θα φτιαχτεί, καθώς και ο αριθμός των ορισμάτων που θα χρησιμοποιήσουμε, είναι αρκετά για να κατανοήσει το σύστημα πως πρέπει να «μεταφραστεί» η μέθοδος της διεπαφής σε ερώτημα sql. Για παράδειγμα, στον πίνακα employees, εάν θέλουμε να έχουμε ένα ερώτημα που αναζητούν

υπαλλήλους με βάση το όνομα τους και την ημερομηνία γέννησης τους , και αντίστοιχο ερώτημα για τη διαγραφή με βάση όνομα και ημερομηνία γέννησης, τότε οι αντίστοιχες μέθοδοι φαίνονται στην παρακάτω εικόνα:

```
Optional<Employee> findEmployeeByFirstNameAndBirthDate(String firstName, LocalDate birthDate);

void deleteEmployeeByFirstNameAndBirthDate(String firstName, LocalDate birthDate);
```

Εικόνα 3-6 : Δείγμα διαφόρων ερωτημάτων χωρίς τη χρήση SQL με τη χρήση του Spring-boot

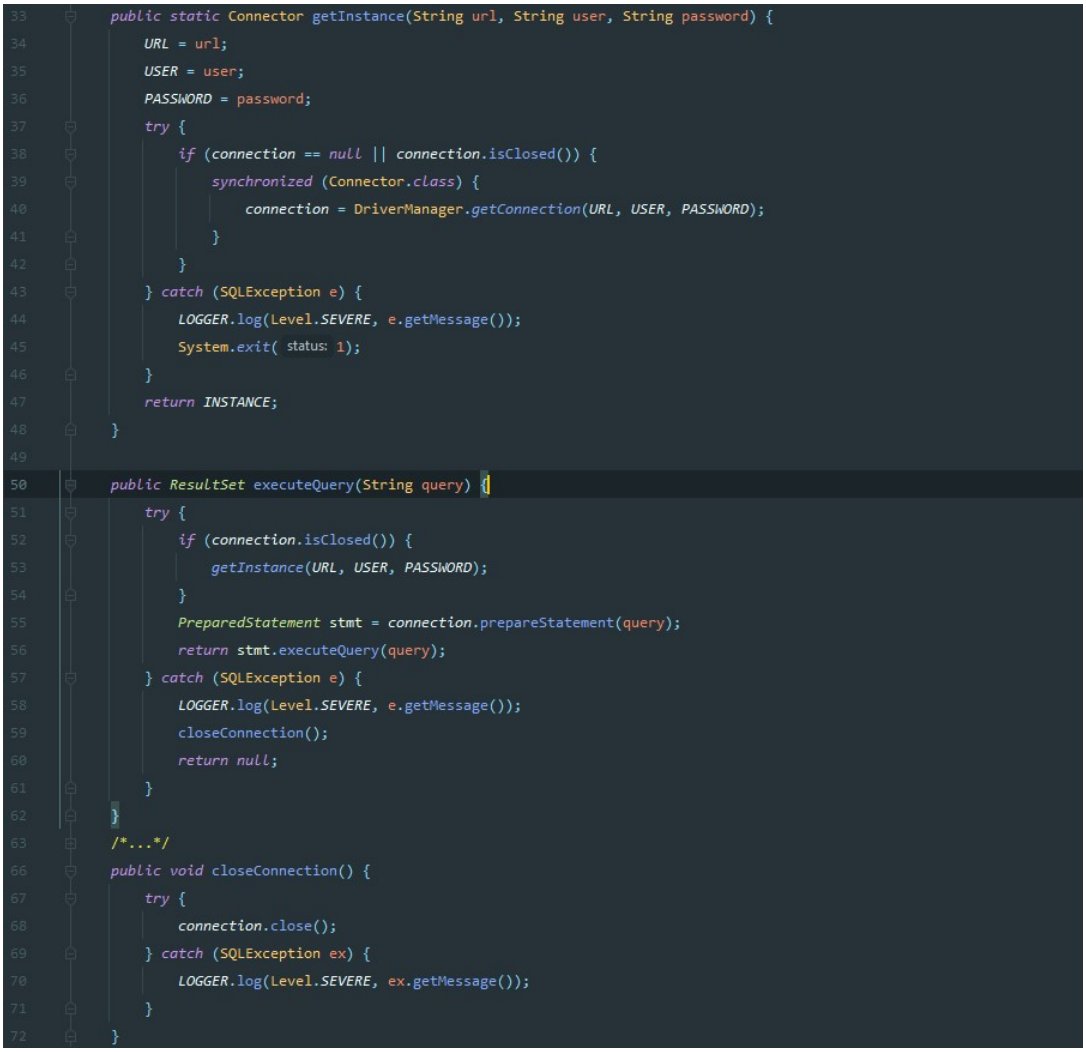
Στην παραπάνω υλοποίηση χρησιμοποιείται η έννοια του Optional για να αποφεύγεται η πιθανότητα να υπάρχουνε αναφορές σε null αντικείμενα.

3.2.2 Με ιδιότυπη υλοποίηση

Για τη διασύνδεση με τη ΒΔ στην ιδιότυπη υλοποίηση, γίνεται χρήση του πακέτου java.sql[17], το οποίο καθορίζει την διεπαφή JDBC (Java Database Connectivity) και αποτελεί ένα από τα ενσωματωμένα πακέτα στο JDK (από το 1997)[14]. Η υλοποίηση της λειτουργικότητας της διασύνδεσης γίνεται μέσα στην κλάση Connector.java. Μέσα σε αυτήν ορίζονται μέθοδοι υπεύθυνες για την αρχικοποίηση της διασύνδεσης, για την εκτέλεση των ερωτημάτων προς τη βάση, καθώς και για το άνοιγμα/κλείσιμο της διασύνδεσης προς τη ΒΔ ώστε να μην δημιουργούνται προβλήματα διαρρροής (leak errors). Πιο συγκεκριμένα, υλοποιούνται 3 βασικές μέθοδοι. Τα ονόματα και η επεξήγηση της λειτουργικότητας τους δίνονται παρακάτω:

- getInstance(): ανακτά το αντικείμενο τύπου Connector, αρχικοποιεί και ανοίγει τη σύνδεση προς τη ΒΔ, για να εκτελεστούν τα ερωτήματα.
- executeQuery(): εκτελεί τα ερωτήματα προς τη ΒΔ και περιέχει λογική που διαχειρίζεται την περίπτωση λάθους
- closeConnection(): κλείνει την ανοικτή σύνδεση προς τη ΒΔ για να μην υπάρχει διαρροή πόρων που πιθανώς να καταλήξει σε άρνηση υπηρεσίας (denial of service)

Στην εικόνα φαίνεται η υλοποίηση των μεθόδων που μόλις περιγράφηκαν:



```
33    public static Connector getInstance(String url, String user, String password) {
34        URL = url;
35        USER = user;
36        PASSWORD = password;
37        try {
38            if (connection == null || connection.isClosed()) {
39                synchronized (Connector.class) {
40                    connection = DriverManager.getConnection(URL, USER, PASSWORD);
41                }
42            }
43        } catch (SQLException e) {
44            LOGGER.log(Level.SEVERE, e.getMessage());
45            System.exit( status: 1);
46        }
47        return INSTANCE;
48    }
49
50    public ResultSet executeQuery(String query) {
51        try {
52            if (connection.isClosed()) {
53                getInstance(URL, USER, PASSWORD);
54            }
55            PreparedStatement stmt = connection.prepareStatement(query);
56            return stmt.executeQuery(query);
57        } catch (SQLException e) {
58            LOGGER.log(Level.SEVERE, e.getMessage());
59            closeConnection();
60            return null;
61        }
62    }
63    /*...*/
66    public void closeConnection() {
67        try {
68            connection.close();
69        } catch (SQLException ex) {
70            LOGGER.log(Level.SEVERE, ex.getMessage());
71        }
72    }
```

Εικόνα 3-7 : Η υλοποίηση της κλάσης Connector, που είναι υπεύθυνη για ερωτήματα προς τη ΒΔ, στην ιδιότυπη υλοποίηση

Όσον αφορά την επιχειρηματική λογική, οι μέθοδοι που αφορούν την ανάκτηση, εγγραφή, διαγραφή, και επεξεργασία των δεδομένων, βρίσκονται στις κλάσεις που είναι υπεύθυνες για την διαχείριση της κάθε οντότητας ξεχωριστά. Παίρνοντας ως παράδειγμα λοιπόν την κλάση DepartmentHandler.java, οι μέθοδοι που εκτελούν εργασίες σχετικές με τη ΒΔ, και η λειτουργικότητα τους, περιγράφεται ως εξής:

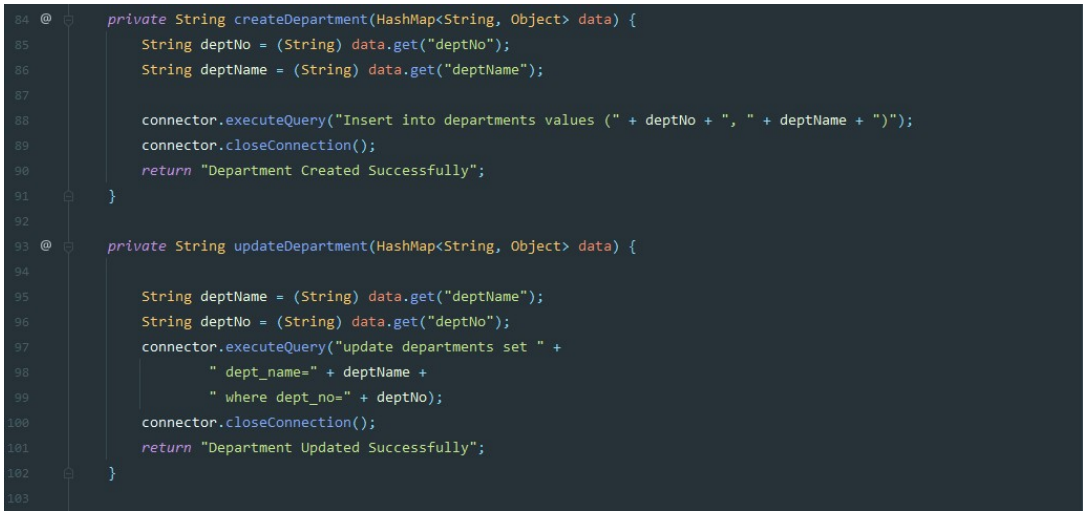
- getAllDepartment(): Ανακτά όλα τα Departments από τη ΒΔ
- getDepartment(): Ανακτά ένα τμήμα με βάση τον κωδικό που παρέχεται στο http αίτημα του προγράμματος-πελάτη

- updateDepartment(): Αλλάζει το όνομα ενός Department με βάση αυτό που παρέχεται στο http αίτημα του προγράμματος-πελάτη
- createDepartment(): Δημιουργεί ένα καινούργιο Department με δεδομένα αυτά που παρέχονται από το πρόγραμμα-πελάτη.

Στις εικόνες φαίνεται η υλοποίηση της καθεμιάς από τις μεθόδους που αναφέρθηκαν:

```
50 @    private List<Object> getAllDepartments() {
51          ArrayList<Object> departments = new ArrayList<>();
52          try {
53              ResultSet resultset = connector.executeQuery("select * from departments d");
54              while (resultset.next()) {
55                  Department department = new Department();
56                  getDepartmentData(resultset, department);
57                  departments.add(department);
58              }
59          } catch (SQLException e) {
60              LOGGER.log(Level.SEVERE, e.getMessage());
61          } finally {
62              connector.closeConnection();
63          }
64          return departments;
65      }
66
67 @    private List<Object> getDepartment(String deptNo) {
68          ArrayList<Object> singletonList = new ArrayList<>();
69          try {
70              ResultSet resultset = connector.executeQuery("select * from departments d where dept_no = " + deptNo);
71              Department department = new Department();
72              while (resultset.next()) {
73                  getDepartmentData(resultset, department);
74              }
75              singletonList.add(department);
76          } catch (SQLException e) {
77              LOGGER.log(Level.SEVERE, e.getMessage());
78          } finally {
79              connector.closeConnection();
80          }
81          return singletonList;
82      }
83
```

Εικόνα 3-8 : Οι μέθοδοι της κλάσης DepartmentHandler.java υπεύθυνες για την ανάκτηση δεδομένων



```
84 @    private String createDepartment(HashMap<String, Object> data) {
85          String deptNo = (String) data.get("deptNo");
86          String deptName = (String) data.get("deptName");
87
88          connector.executeQuery("Insert into departments values (" + deptNo + ", " + deptName + ")");
89          connector.closeConnection();
90          return "Department Created Successfully";
91      }
92
93 @    private String updateDepartment(HashMap<String, Object> data) {
94
95          String deptName = (String) data.get("deptName");
96          String deptNo = (String) data.get("deptNo");
97          connector.executeQuery("update departments set " +
98                  " dept_name=" + deptName +
99                  " where dept_no=" + deptNo);
100         connector.closeConnection();
101         return "Department Updated Successfully";
102     }
103
```

Εικόνα 3-9 : Οι μέθοδοι της κλάσης DepartmentHandler.java υπεύθυνες για την επεξεργασία/δημιουργία δεδομένων

Αξίζει να σημειωθεί ότι εδώ η συγγραφή ερωτημάτων sql γίνεται ξεχωριστά για κάθε ερώτημα που θέλουμε στη βάση δεδομένων. Η γενική δομή μεθόδων αναζήτησης ειναι ως εξής:

1. Δημιουργία δομής δεδομένων που φιλοξενεί τα αποτελέσματα που ζητήθηκαν
2. Δημιουργία ερωτήματος SQL, και παραμετροποίηση όπου χρειάζεται (πχ. στις περιπτώσεις που έχουμε κριτήρια αναζήτησης)
3. Εκτέλεση του ερωτήματος μέσω της κλάσης Connector.java
4. Μετατροπή των δεδομένων ΒΔ από εγγραφές σε αντικείμενα της Java, όπως αυτά έχουν οριστεί στο πακέτο domain.
5. Επιστροφή της δομής δεδομένων με τα αντικείμενα που ζητήθηκαν.

Σε περιπτώσεις όπου γίνεται επεξεργασία δεδομένων, γίνεται πρώτα ένα ερώτημα sql με βάση τα κριτήρια που έχουνε δοθεί από το πρόγραμμα-πελάτη, και αφού γίνεται επεξεργασία των δεδομένων, αυτά αποθηκεύονται στη βάση, και η ροή του προγράμματος συνεχίζει χωρίς ο server να αποστείλλει κάποιο payload, πέρα από την αντίστοιχη http απάντηση (201 για αιτήματα δημιουργίας, 204 για αιτήματα διαγραφής ή επεξεργασίας)

3.3 Υλοποίηση για διασύνδεση μέσω HTTP

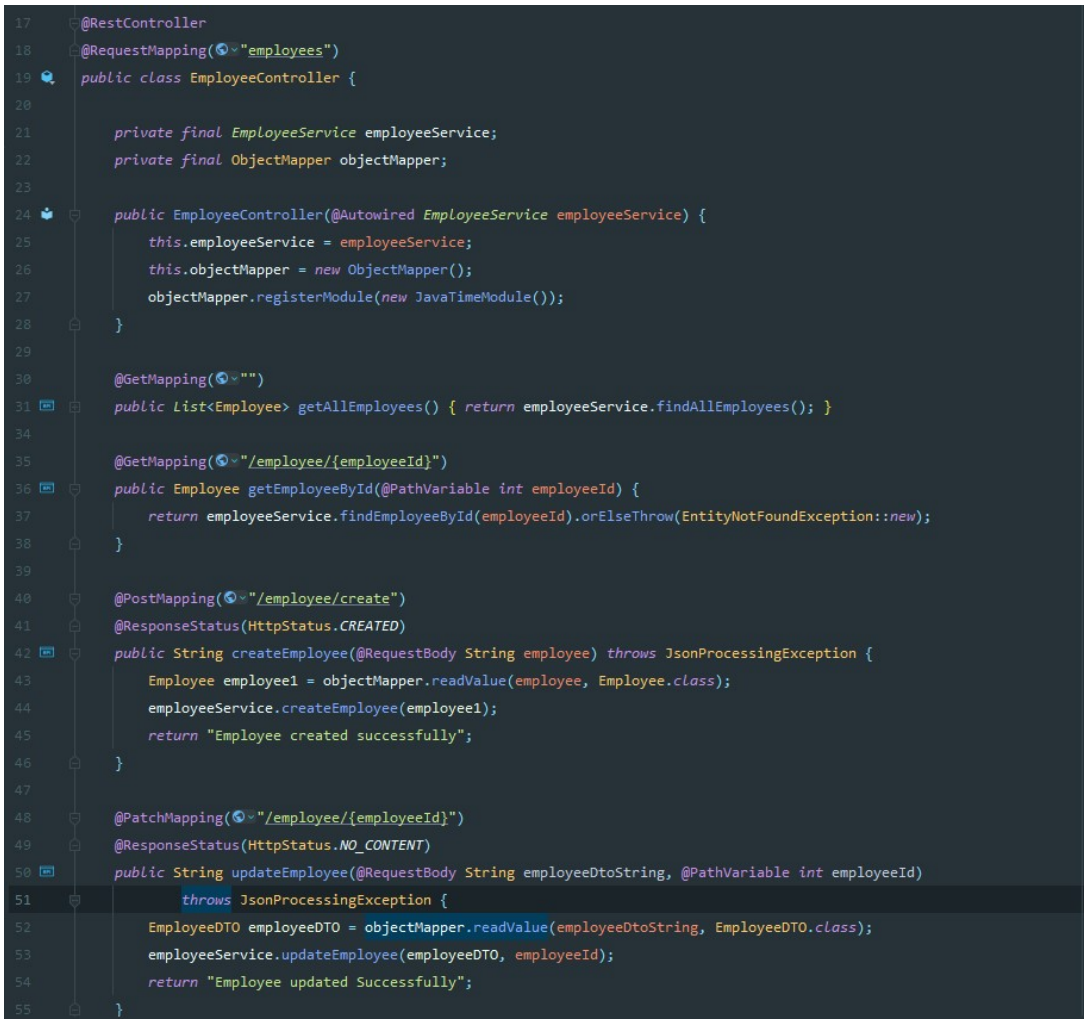
3.3.1 Με τη χρήση του Spring boot

Με τη χρήση του πλαισίου Spring boot, η διαχείριση των αιτημάτων http γίνεται στο επίπεδο των ελεγκτών (Controllers). Οι controllers είναι κλάσεις Java όπου έχουνε το ειδικό annotation (@RestController) για να υποδείξουνε στο πρόγραμμα ότι είναι υπεύθυνες για διαχείριση αιτημάτων http, και το annotation @RequestMapping για να προσδιορίσουμε ποιο είναι το σχετικό path στο οποίο θέλουμε να «ακούμε» για αιτήματα http.

Η κάθε μέθοδος έχει επίσης σχετικό annotation που υποδηλώνει αφ'ενός τον τύπο του αιτήματος που περιμένει (πχ. GET, POST, PUT, DELETE), και μία παράμετρο με το path στο οποίο γίνεται η ακρόαση (το οποίο λειτουργεί αθροιστικά πάνω στο προηγούμενο path. Έτσι, εαν μια μέθοδος έχει path το "/department/create" και ο controller έχει path το "departments" τότε το πλήρες μονοπάτι που πρέπει να ακολουθήσουμε είναι το "departments/department/create"). Επιπλέον, στα ορίσματα που δέχεται η εκάστοτε μέθοδος, υπάρχουνε και ειδικά annotations @Pathvariable, και @RequestBody. Το πρώτο χρησιμοποιείται για να υποδηλώσει ότι στο path που παρέχεται υπάρχει και κάποια μεταβλητή που μπορεί να χρησιμοποιηθεί μέσα στη μέθοδο (πχ το ID κάποιου employee) και το δεύτερο για να έχουμε πρόσβαση στο σώμα του αιτήματος (σε μορφή αλφαριθμητικού, η οποία πρέπει να μετατραπεί είτε σε Hashmap είτε σε λίστα, αναλόγως της αναμενόμενης μορφής των εισερχόμενων δεδομένων).

Πέρα από τον μεγάλο αριθμό λεπτομερειών που αποκρύπτονται μέσω των αφαιρέσεων και διευκολύνσεων που παρέχονται από τα annotations αυτά, απουσιάζει και η ανάγκη επιλογής τύπου απάντησης στα αιτήματα http, καθώς το πλάισιο φροντίζει να δώσει και τη σωστό http status αναλόγως με τον τύπο του αιτήματος που δέχεται. Το βασικότερο όμως πλεονέκτημα που φαίνεται να υπάρχει στην υλοποίηση των ελεγκτών με τη βοήθεια του πλαισίου, είναι ότι αφ'ενός γίνεται νοητικός διαχωρισμός των ξεχωριστών επιπέδων που απαρτίζουν την εφαρμογή, οπότε γίνεται πιο ομοιόμορφη η κατανομή κώδικα, κάνοντας την εφαρμογή πιο ευανάγνωστη και κατανοητή στον προγραμματιστή. Κατα μία έννοια, το πλαίσιο λογισμικού μετακινεί τις ανάγκες συγγραφής κώδικα, δίνοντας προτεραιότητα στην συγγραφή επιχειρηματικής λογικής, και βάζοντας σε μικρότερη προτεραιότητα τις τεχνικές υλοποιήσεις. Χρησιμοποιώντας

ως παράδειγμα την κλάση EmployeeController.java, βλέπουμε ότι γράφεται ελάχιστος κώδικας που αφορά τα πιο τεχνικά ζητήματα, και αντ'αυτού το μεγαλύτερο μέρος κώδικα αφορά τη λογική που πρέπει να ακολουθεί η εφαρμογή:



```java
@RestController
@RequestMapping("employees")
public class EmployeeController {

    private final EmployeeService employeeService;
    private final ObjectMapper objectMapper;

    public EmployeeController(@Autowired EmployeeService employeeService) {
        this.employeeService = employeeService;
        this.objectMapper = new ObjectMapper();
        objectMapper.registerModule(new JavaTimeModule());
    }

    @GetMapping("")
    public List<Employee> getAllEmployees() { return employeeService.findAllEmployees(); }

    @GetMapping("/employee/{employeeId}")
    public Employee getEmployeeById(@PathVariable int employeeId) {
        return employeeService.findEmployeeById(employeeId).orElseThrow(EntityNotFoundException::new);
    }

    @PostMapping("/employee/create")
    @ResponseStatus(HttpStatus.CREATED)
    public String createEmployee(@RequestBody String employee) throws JsonProcessingException {
        Employee employee1 = objectMapper.readValue(employee, Employee.class);
        employeeService.createEmployee(employee1);
        return "Employee created successfully";
    }

    @PatchMapping("/employee/{employeeId}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public String updateEmployee(@RequestBody String employeeDtoString, @PathVariable int employeeId)
            throws JsonProcessingException {
        EmployeeDTO employeeDTO = objectMapper.readValue(employeeDtoString, EmployeeDTO.class);
        employeeService.updateEmployee(employeeDTO, employeeId);
        return "Employee updated Successfully";
    }
```

Εικόνα 3-10 : Οι μέθοδοι της κλάσης EmployeeController.java υπεύθυνες για την διαχείριση των αιτημάτων Http

3.3.2 Με ιδιότυπη υλοποίηση

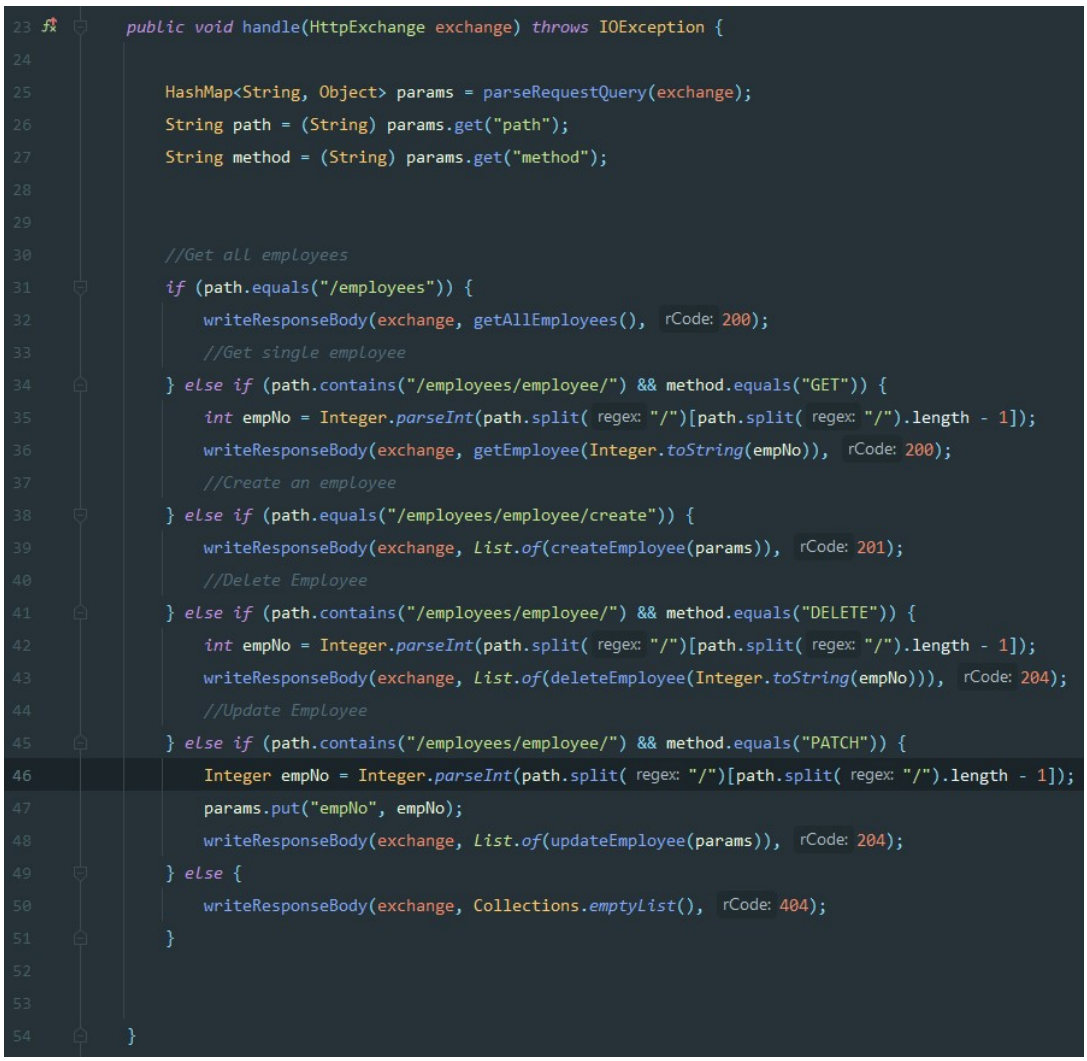
Στην περίπτωση της ιδιότυπης υλοποίησης, η διαδικασία είναι ελαφρώς πιο πολύπλοκη: Το πακέτο που χρησιμοποιείται για να δώσει δυνατότητα στην εφαρμογή μας να ξεκινάει έναν εξυπηρετητή, παρέχει και λειτουργικότητα για διαχείριση URL μέσω της διεπαφής HttpContext. Η διεπαφή αντιπροσωπεύει μια αντιστοίχιση από μια διαδρομή URI σε έναν χειριστή ανταλλαγής στο πρόγραμμα εξυπηρετητή. Μόλις δημιουργηθεί το Context για μια συγκεκριμένη διαδρομή, όλα τα αιτήματα που

λαμβάνονται από τον διακομιστή για τη διαδρομή θα αντιμετωπίζονται καλώντας το συγκεκριμένο αντικείμενο χειριστή.

Ένας σημαντικός περιορισμός που προκύπτει από την υλοποίηση του HttpContext έχει να κάνει με τον τρόπο που η εφαρμογή αντιστοιχίζει διαδρομές. Όταν λαμβάνεται ένα αίτημα HTTP, το κατάλληλο HttpContext (και ο χειριστής) εντοπίζονται βρίσκοντας κάνοντας αντιστοίχιση της διαδρομής με το πρόθεμα που είναι πιο κοντά στη διαδρομή αυτή. Για παράδειγμα, εάν έχουμε ενα context με διαδρομή "/departments" τότε όλες οι παρακάτω διαδρομές θα αντιστοιχιστούν σε αυτό το context :

- /departments/department/
- /departments/department/create
- /departments/department/24
- Κ.ο.κ.

Αυτό πρακτικά σημαίνει ότι εφ'όσον έχουμε οργανώσει την εφαρμογή με βάση τις οντότητες που διαχειρίζεται, τότε στην αρχική δήλωση των Context πρέπει να μπει μόνο ή αρχική διαδρομή, και μέσα στον Handler να εφαρμόζεται λογική για να γίνεται διάκριση στις διαδρομές και τους τύπους των υποστηριζόμενων αιτημάτων. Παίρνοντας για παράδειγμα την υλοποίηση του EmployeeHandler.java :



```
public void handle(HttpExchange exchange) throws IOException {

    HashMap<String, Object> params = parseRequestQuery(exchange);
    String path = (String) params.get("path");
    String method = (String) params.get("method");



    //Get all employees
    if (path.equals("/employees")) {
        writeResponseBody(exchange, getAllEmployees(), rCode: 200);
        //Get single employee
    } else if (path.contains("/employees/employee/") && method.equals("GET")) {
        int empNo = Integer.parseInt(path.split( regex: "/")[path.split( regex: "/").length - 1]);
        writeResponseBody(exchange, getEmployee(Integer.toString(empNo)), rCode: 200);
        //Create an employee
    } else if (path.equals("/employees/employee/create")) {
        writeResponseBody(exchange, List.of(createEmployee(params)), rCode: 201);
        //Delete Employee
    } else if (path.contains("/employees/employee/") && method.equals("DELETE")) {
        int empNo = Integer.parseInt(path.split( regex: "/")[path.split( regex: "/").length - 1]);
        writeResponseBody(exchange, List.of(deleteEmployee(Integer.toString(empNo))), rCode: 204);
        //Update Employee
    } else if (path.contains("/employees/employee/") && method.equals("PATCH")) {
        Integer empNo = Integer.parseInt(path.split( regex: "/")[path.split( regex: "/").length - 1]);
        params.put("empNo", empNo);
        writeResponseBody(exchange, List.of(updateEmployee(params)), rCode: 204);
    } else {
        writeResponseBody(exchange, Collections.emptyList(), rCode: 404);
    }


}
```

Εικόνα 3-11 : Οι μέθοδοι της κλάσης EmployeeHandler.java υπεύθυνες για την διαχείριση των αιτημάτων Http

Βλέπουμε ότι πρέπει να χρησιμοποιηθεί μια αρκετά περίπλοκη δομή ελέγχου, ώστε να διαχωρίζουμε τις διάφορες διαδρομές και τους τύπους των αιτημάτων που μπορεί να επεξεργαστεί η εφαρμογή. Αξίζει να γίνει αναφορά στο πως γίνεται η εξαγωγή δεδομένων από τη διαδρομή, εφ'όσον απουσιάζει η δυνατότητα χρήσης annotations για αυτόν τον σκοπό. Στην περίπτωση αυτή γίνεται επεξεργασία της διαδρομής χειρωνακτικά, με σκοπό την άντληση των παραμέτρων που χρειάζονται για την ευρυθμη λειτουργία της εφαρμογής. Η αντιμετώπιση αυτή, αν και αναγκαία, φαίνεται ότι ανεβάζει τον δείκτη Κυκλωματικής πολυπλοκότητας, καθώς και αυτόν της γνωστικής

πολυπλοκότητας, καθιστώντας έτσι την εφαρμογή ελαφρώς πιο στρυφνή στην κατανόηση και επέκταση της (σε επίπεδο κώδικα).

Ενδιαφέρον επίσης παρουσιάζουνε οι μέθοδοι parseRequestQuery(), και writeResponseBody(). Οι δύο αυτές μέθοδοι έχουνε υλοποιηθεί σε μια abstract Υπερκλάση που ονομάζεται AbstractHandler.java, και είναι η κλάση από την οποία κληρονομούνε όλοι οι Handlers που έχουμε ορίσει στην εφαρμογή μας. Ειδικότερα, η parseRequestQuery είναι μέθοδος που ελέγχει τον τύπο του αιτήματος που έγινε, και αναλόγως με το έαν είναι GET η POST/PATCH προσπαθεί να κάνει εξαγωγή παραμέτρων από τις παραμέτρους της διαδρομής ή από το σώμα του αιτήματος αντίστοιχα, και αν τα τοποθετήσει σε ένα HashMap για επεξεργασία. Στην εικόνα φαίνεται η υλοποίηση για την κάθε περίπτωση:

```
67    /** Parses GET request parameters ...*/
73  @ private HashMap<String, Object> parseGetRequestParam(String path, String query) {
74        HashMap<String, Object> parameters = new HashMap<>();
75        parameters.put("path", path);
76        parameters.put("method", "GET");
77        if (query == null) {
78            return parameters;
79        }
80        String[] pairs = query.split( regex: "[&]");
81        for (String pair : pairs) {
82            String[] param = pair.split( regex: "[=]");
83            if (param.length == 0) {
84                continue;
85            }
86            String key = URLDecoder.decode(param[0], StandardCharsets.UTF_8);
87            String value = URLDecoder.decode(param[1], StandardCharsets.UTF_8);
88            parameters.merge(key, value, (oldVal, newVal) -> oldVal + "&" + newVal);
89        }
90        return parameters;
91    }
92
93    /** Parses POST request body ...*/
100 @ private HashMap<String, Object> parsePostRequestBody(String path, InputStream requestBody) throws IOException {
101       String query = new String(requestBody.readAllBytes(), StandardCharsets.UTF_8);
102       HashMap<String, Object> parameters = (HashMap<String, Object>) objectMapper.readValue(query, Map.class);
103       parameters.put("path", path);
104       parameters.put("method", "POST");
105       return parameters;
106   }
```

Εικόνα 3-12 : Οι μέθοδοι της κλάσης AbstractHandler.java υπεύθυνες για την συλλογή των παραμέτρων από τα αιτήματα.

Η writeResponseBody είναι υπεύθυνη για την δημιουργία και αποστολή της απάντησης στα αιτήματα που δέχεται η εφαρμογή. Συγκεκριμένα, για κάθε http αίτημα, ορίζει

πρώτα τα απαραίτητα headers που αφορούν τον τύπο των δεδομένων απάντησης (application/json), το μέγεθος των δεδομένων (content-length), και αποστέλλει τα δεδομένα μέσω της κλάσης OutputStream. Ολοκληρώνοντας τη δουλειά, κλείνει το OutputStream ώστε να μην υπάρχουνε αχρησιμοποίητοι πόροι που μπορεί να καταλήξουνε σε memory leak. Στην παρακάτω εικόνα φαίνεται η υλοποίηση της μεθόδου:

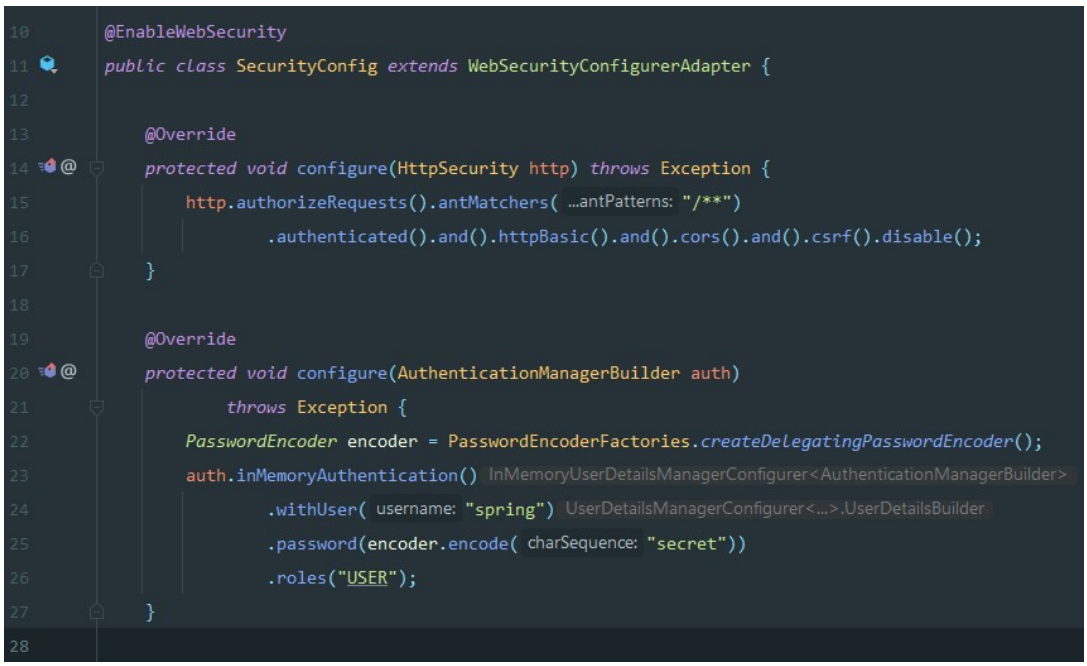
```
41 @    protected void writeResponseBody(HttpExchange exchange, List<Object> responseData, int rCode) throws IOException {
42
43          String response = objectMapper.writeValueAsString(responseData);
44          exchange.getResponseHeaders().set("Content-Type", "application/json");
45          exchange.sendResponseHeaders(rCode, response.length());
46          OutputStream os = exchange.getResponseBody();
47          os.write(response.getBytes());
48          os.close();
49      }
50
```

Εικόνα 3-13: Η μέθοδος της κλάσης AbstractHandler.java υπεύθυνη για την παραγωγή μηνυμάτων-απαντήσεων στα αιτήματα.

3.4 Υλοποίηση security

3.4.1 Με τη χρήση του Spring boot

Για την περίπτωση του Spring boot η υλοποίηση της αυθεντικοποίησης γίνεται με τη χρήση του πακέτου org.springframework.boot.spring-boot-starter-test. Το πακέτο αυτό στην απλούστερη υλοποίηση του προσφέρει δυνατότητα αυθεντικοποίηση με χρήστη ονόματι “user” και κατα τη διάρκεια της εκκίνησης της εφαρμογής ορίζεται ένα τυχαίο αλφαριθμητικό για password. Στην συγκεκριμένη υλοποίηση αλλάχτηκε ελαφρώς για να έχουμε τη δυνατότητα να ορίσουμε εμείς το συνθηματικό του χρήστη, και να διασφαλίσουμε ότι όλα τα αιτήματα που γίνονται προς την εφαρμογή έχουνε και τα απαραίτητα διαπιστευτήρια. Για το σκοπό αυτό, η βιβλιοθήκη ορίζει ότι πρέπει να δημιουργηθεί μια κλάση που να κληρονομεί την WebSecurityConfigurerAdapter και να υλοποιεί δύο μεθόδους ονόματι configure. Στην παρακάτω εικόνα φαίνεται η υλοποίηση.



Εικόνα 3-13: Η κλάση SecurityConfig.java υπεύθυνη για την παραμετροποίηση της αυθεντικοποίησης στην υλοποίηση με spring-boot.

Απο την εικόνα γίνεται εμφανές ότι ορίζεται ο χρήστης με όνομα “spring” και συνθηματικό “secret”, ο οποίος είναι και ο μόνος ικανός να κάνει αιτήματα προς την εφαρμογή και να λαμβάνει τις επιθυμητές απαντήσεις. Οποιαδήποτε απόπειρα αιτήματος με διαφορετικό (ή καθόλου) χρήστη, έχει ως αποτέλεσμα την λήψη απάντησης με κωδικό 401 (unauthorized).

3.4.2 Με ιδιότυπη υλοποίηση

Στην περίπτωση της custom υλοποίησης χρησιμοποιήθηκε το παρεχόμενο πακέτο που ονομάζεται com.sun.net.httpServer.BasicAuthenticator και παρέχει διεπαφή για έλεγχο διαπιστευτηρίων. Η διεπαφή που παρέχεται ειναι αρκετά ευέλικτη, καθώς επιτρέπει στην εφαρμογή να προσδιορίσει μόνο τα κριτήρια με βάση τα οποία θα γίνεται η αυθεντικοποίηση, χωρίς να περιορίζει την πηγή των δεδομένων που θα γίνει ταυτοποίηση. Με βάση την δοκιμαστική υλοποίηση που έγινε για τους σκοπούς της εργασίας, η υλοποίηση της κλάσης είναι αρκετά απλή, και είναι αμέσως εμφανείς τόσο η ευκολία, όσο και η επεκτασιμότητα της προσέγγισης που προσφέρεται μέσα από αυτό το πακέτο. Στην εικόνα παρατίθεται ένας απλός authenticator που κληρονομεί από την κλάση που συζητήθηκε παραπάνω:

```java
import com.sun.net.httpserver.BasicAuthenticator;

/**
 * Basic in-memory authenticator
 */
public class SampleAuthenticator extends BasicAuthenticator {
    /**
     * Creates a BasicAuthenticator for the given HTTP realm
     *
     * @param realm The HTTP Basic authentication realm
     * @throws NullPointerException if the realm is an empty string
     */
    public SampleAuthenticator(String realm) { super(realm); }

    @Override
    public boolean checkCredentials(String username, String password) {
        return "username".equals(username) && password.equals("password");
    }
}
```

Εικόνα 3-14: Η κλάση SampleAuthenticator.java υπεύθυνη για την παραμετροποίηση της αυθεντικοποίησης στην ιδιότυπη υλοποίηση.

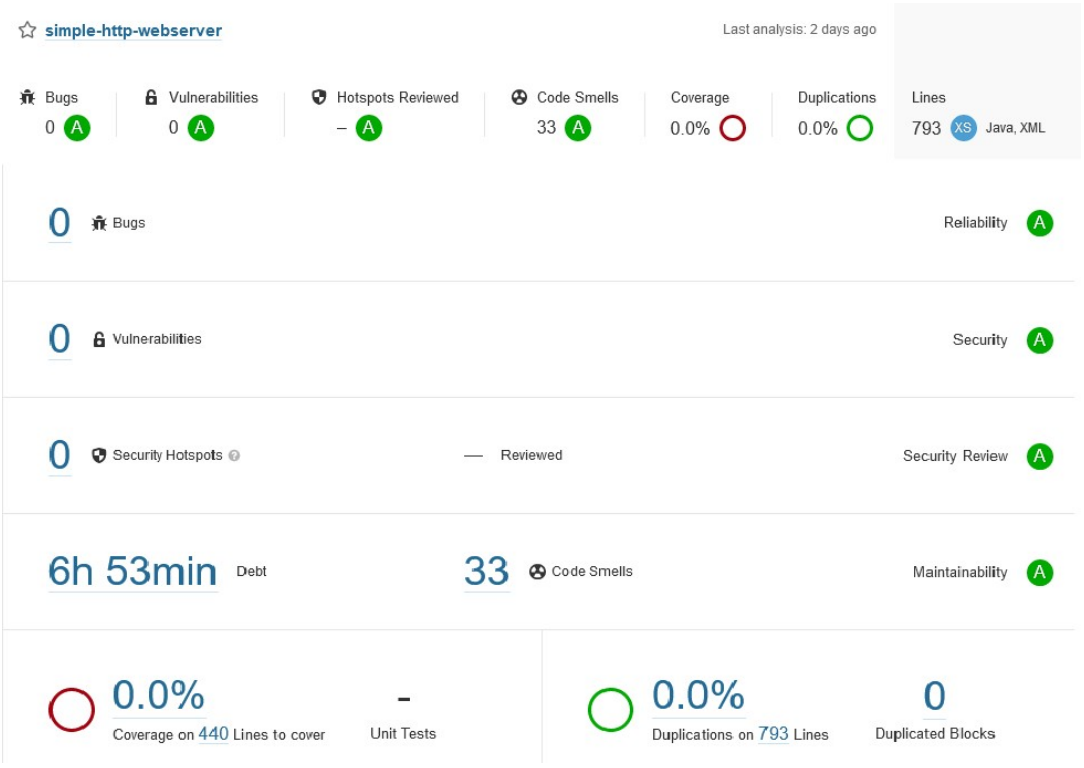
Πράγματι, με αυτή την υλοποίηση όλες τα αιτήματα που γίνονται προς την εφαρμογή πρέπει να έχουνε όνομα χρήστη “username” και συνθηματικό “password” για να καθίσταται δυνατή η λήψη απαντήσεων με δεδομένα.

4 Επίλογος

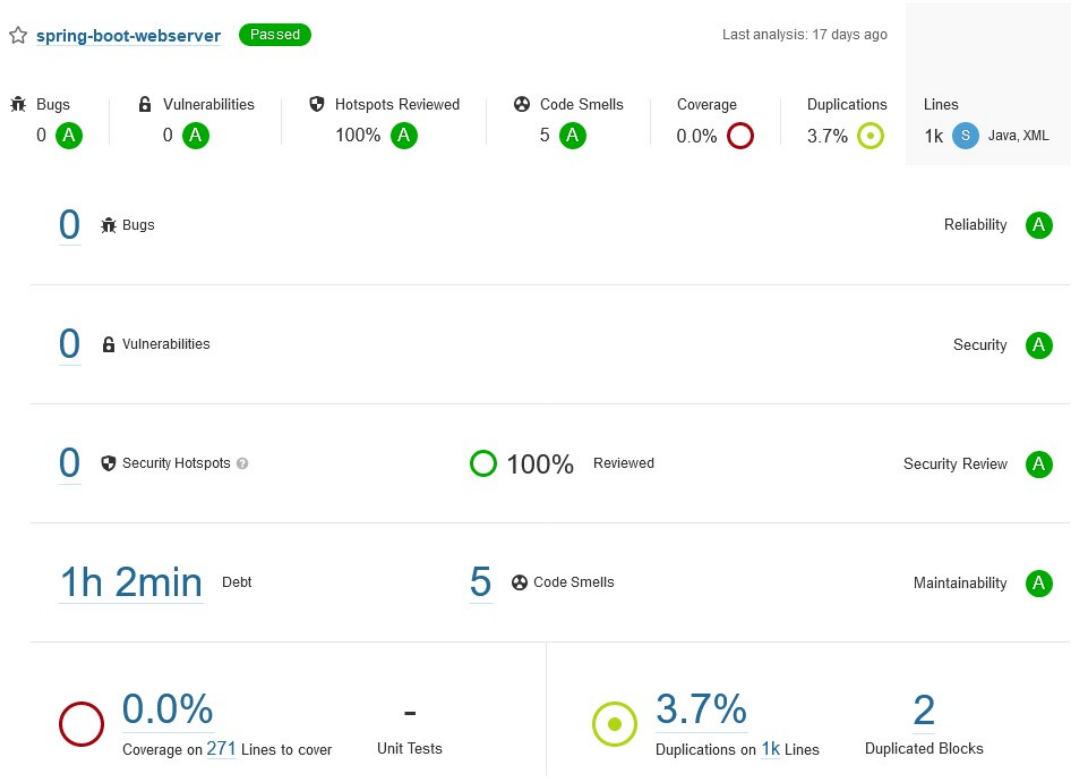
4.1 Σύνοψη και συμπεράσματα

4.1.1 Γενικά συμπεράσματα

Θεωρώντας ως τελικό το στάδιο κατά το οποίο έχει ολοκληρωθεί η υλοποίηση της λειτουργικότητας (χωρίς την υλοποίηση testing), παρακάτω βλέπουμε τη γενική εικόνα που δίνει το Sonarqube για την κάθε εφαρμογή:



Εικόνα 4-1: Αρχική αξιολόγηση από το Sonarqube για την ιδιότυπη υλοποίηση της εφαρμογής



Εικόνα 4-2: Αρχική αξιολόγηση από το Sonarqube για την υλοποίηση της εφαρμογής με τη βοήθεια του Spring-boot

Σε πρώτη φάση λοιπόν μπορούμε να εξάγουμε τα παρακάτω συμπεράσματα :

- Η υλοποίηση με το spring-boot έχει κάποια Duplications τα οποία οφείλονται στην αυτόματη μοντελοποίηση και υλοποίηση των πινάκων dept_emp, dept_manager από το plugin που ήταν υπεύθυνο για την αυτόματη δημιουργία οντοτήτων. Οι δύο αυτοί πίνακες κατ'ουσία είναι ίδιοι, καθώς έχουνε τα ίδια πεδία και συνδέουν τους δύο ίδιους πίνακες με σχέση πολλά-προς-πολλά. Αυτό πρακτικά μεταφράζεται στην ύπαρξη διπλότυπου κώδικα όσον αφορά τα getter και setter methods των δύο οντοτήτων.
- Ο αριθμός των code smells στην ιδιότυπη εφαρμογή είναι σαφώς μεγαλύτερος στην ιδιότυπη υλοποίηση. Τα code smells που εμφανίζονται είναι ως επι το πλείστον τύπου "major" και έχουνε να κάνουνε με τη χρήση πακέτων com.sun. Η χρήση αυτού του πακέτου θεωρείται "Implementation specific" και δεν προτείνεται, καθώς δεν υπάρχει εγγύηση ότι θα λειτουργεί με μεταγενέστερες εκδόσεις της java. Άλλες συνηθισμένες περιπτώσεις code smells στην εφαρμογή

αφορούν την επαναχρησιμοποίηση αλφαριθμητικών χωρίς να γίνει κάποια ανάθεση σε σταθερά. Αυτές οι οσμές έχουνε υψηλότερο impact (τύπου “critical”) και αποτελούν σοβαρό πρόβλημα, καθώς κάνουνε τη διαδικασία του refactoring ιδιαιτέρα χρονοβόρα και επικίνδυνη (αφού υπάρχει περίπτωση να μην αλλάξουνε όλες οι θέσεις που εμφανίζεται το αλφαριθμητικό που θέλουμε να αντικαταστήσουμε). Τα αλφαριθμήτικά που φαίνονται να επαναλαμβάνονται είναι λεκτικά που χρησιμοποιούνται ως παράμετροι για το hashmap που χρησιμοποιείται στην εφαρμογή κατά τη διάρκεια εξαγωγής παραμέτρων από τα αιτήματα Http. Στην εικόνα 4-3 φαίνεται μια τέτοια περίπτωση

Εικόνα 4-3 : Οσμή σχετική με τη μη δημιουργία σταθεράς για επαναλαμβανόμενο αλφαριθμητικό

- Οι οσμές κώδικα της υλοποίησης με Spring-boot αφορούνε 2 περιπτώσεις διπλότυπου κώδικα (που αναλύθηκε παραπάνω), τη μετονομασία μιας μεταβλητής κλάσης που έχει ίδιο όνομα με την ίδια κλάση (και άρα υπάρχει περίπτωση να προκαλέσει σύγχιση), και την απουσία Assertion στην αυτόματα δημιουργημένη κλάση SpringBootServerApplicationTests.java.

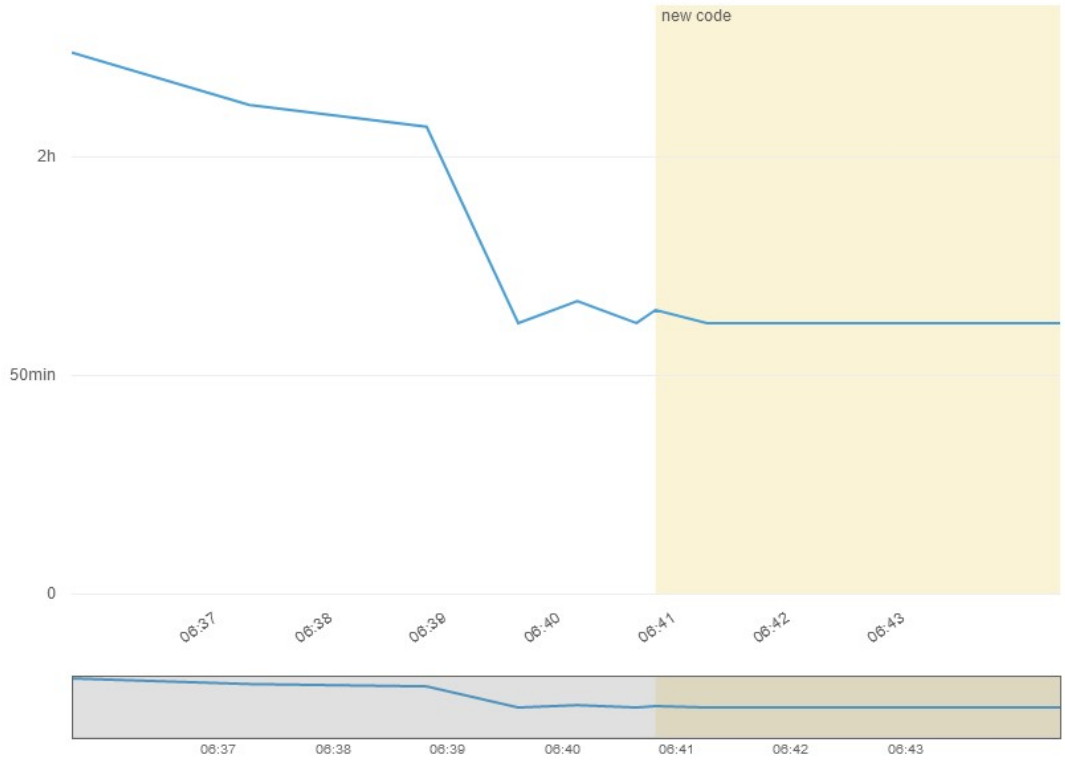
Από τις παραπάνω παρατηρήσεις, όσον αφορά τις οσμές κώδικα, είναι ασφαλές να εξάγουμε ως συμπέρασμα ότι οι οσμές κώδικα που αφορούν τα duplications, οφείλονται κυρίως στο γεγονός ότι η υλοποίηση της αποκωδικοποίησης των δεδομένων που γίνεται κατά τη διάρκεια της ανάλυσης των παραμέτρων του εκάστοτε αιτήματος http, δεν είναι γραμμένη με το γενικότερο δυνατό τρόπο. Από την άλλη πλευρά, η υλοποίηση με

Spring-boot επειδή έχει αναλάβει εσωτερικά την υλοποίηση αυτού του χαρακτηριστικού, μας γλυτώνει από τέτοιου είδους προβλήματα.

4.1.2 Τεχνικό χρέος

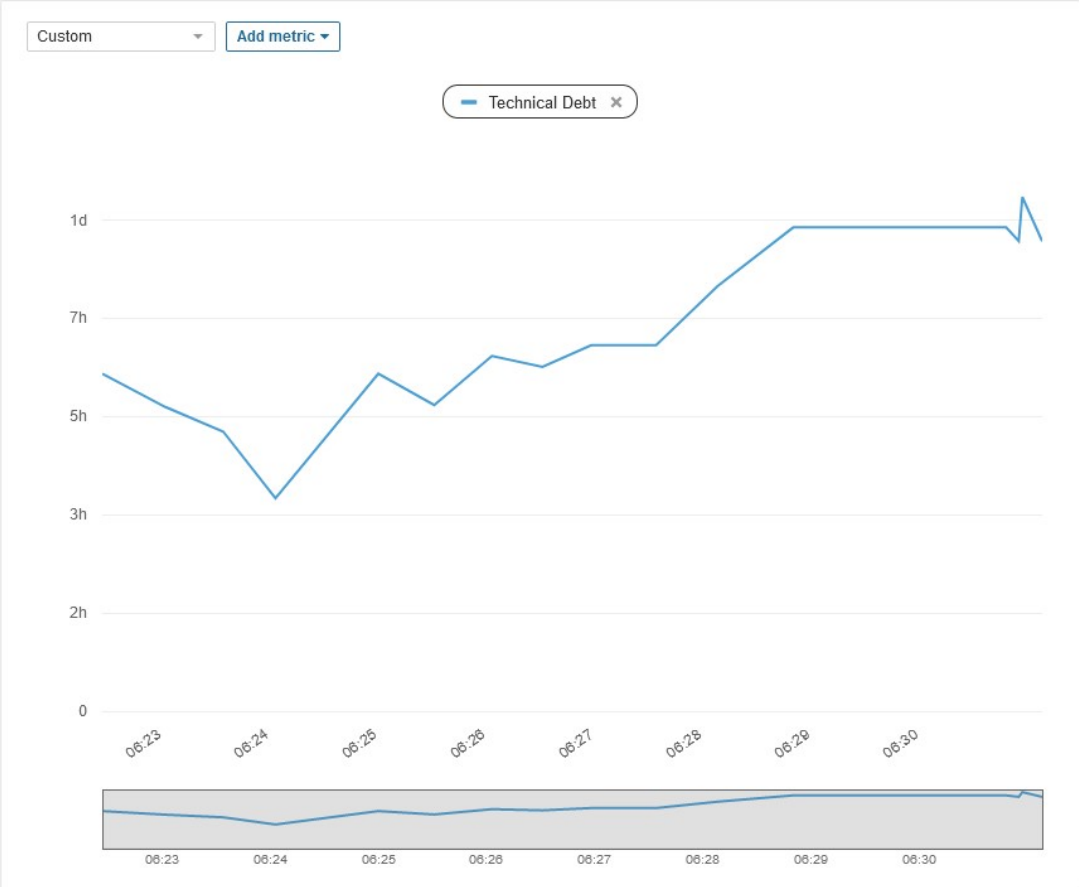
Από την ανάλυση των οσμών κώδικα, των προβλημάτων (bugs) αλλά και των Security hotspots προκύπτει το σύνολο του τεχνικού χρέους, το οποίο και φαίνεται στις εικόνες 4-1, 4-2. Όπως παρατηρούμε, το τεχνικό χρέος στην ιδιότυπη υλοποίηση είναι σαφώς μεγαλύτερο(5ω23λ) σε μέγεθος από αυτό της υλοποίησης με spring-boot (1ω2λ). Η διαφορά αυτή προκύπτει από τις οσμές κώδικα που αναλύθηκαν παραπάνω, και συνηγορεί υπέρ της χρήσης πλαισίου λογισμικού, καθώς δείχνει ότι όταν το επίκεντρο της υλοποίησης αφορά την επιχειρηματική λογική και όχι τεχνικές λέπτομέρειες, τότε εμφανίζονται και λιγότερες περιπτώσεις που μπορεί να προκύψουνε ζητήματα μη βέλτιστης υλοποίησης.

Κοιτώντας τα ιστορικά δεδομένα, όσον αφορά το τεχνικό χρέος, γίνεται φανερό ότι στην υλοποίηση με spring-boot το τεχνικό χρέος με την πάροδο του χρόνου μειώνεται, καθώς οι σχετικές οσμές κώδικα που προέκυπταν είτε αφορούσαν μη χρησιμοποιούμενα imports, είτε αφορούσαν κλάσεις που είχαν δημιουργηθεί αλλά δεν είχε υλοποιηθεί το σώμα τους. Στην τελική μορφή της η εφαρμογή είτε δεν είχε κάποια σημαντική οσμή κώδικα, είτε ήταν περιπτώσεις όπου έγινε χειρωνακτική τους επίλυση καθώς δεν ήταν δυνατό να γίνουν οι αλλαγές που προτείνονταν από το Sonarqube.



Εικόνα 4-4 : Γράφημα ιστορικής ανάλυσης τεχνικού χρέους για την υλοποίηση της εφαρμογής με spring-boot

Αντιθέτως, στην περίπτωση της ιδιότυπης υλοποίησης, το τεχνικό χρέος φαίνεται να ακολουθεί ανοδική πορεία κατά τη διάρκεια της ανάπτυξης. Το τεχνικό χρέος προκύπτει από τον αυξανόμενο αριθμό κλάσεων που χρησιμοποιούν το implementation specific πακέτο com.sun.net.http. Εκ πρώτης όψεως το πρόβλημα που αφορά αυτό το πακέτο (τουτέστιν, η πιθανή του ασυμβατότητα με μελλοντικές εκδόσεις της java) θα μπορούσε να θεωρηθεί ήσσονος σημασίας. Παρ'όλα αυτά, το γεγονός ότι δεν εμφανίζεται κάποιο αντίστοιχο πρόβλημα με τη χρήση των πακέτων του spring-boot συνηγορεί και πάλι υπέρ της υλοποίησης με πλαίσιο λογισμικού. Αυτό προκύπτει από το γεγονός ότι πίσω από το συγκεκριμένο πλαίσιο λογισμικού υπάρχει μια ομάδα υπεύθυνη για τη συντήρηση και προσθήκη χαρακτηριστικων σε αυτό (ως είθισται). Στη συντήρηση, προφανώς, περιλαμβάνεται και η διατήρηση συμβατότητας με την έκδοση της γλώσσας που χρησιμοποιείται, διαφορετικά το πλαίσιο θα σταματήσει να χρησιμοποιείται. Επιπλέον, καθώς η διαδικασία αυτή της ανάπτυξης γίνεται από εξειδικευμένη ομάδα, η αναβάθμιση του πλαισίου θεωρείται πάντα η σωστή και ασφαλής επιλογή για τον τελικό χρήστη του πλαισίου.

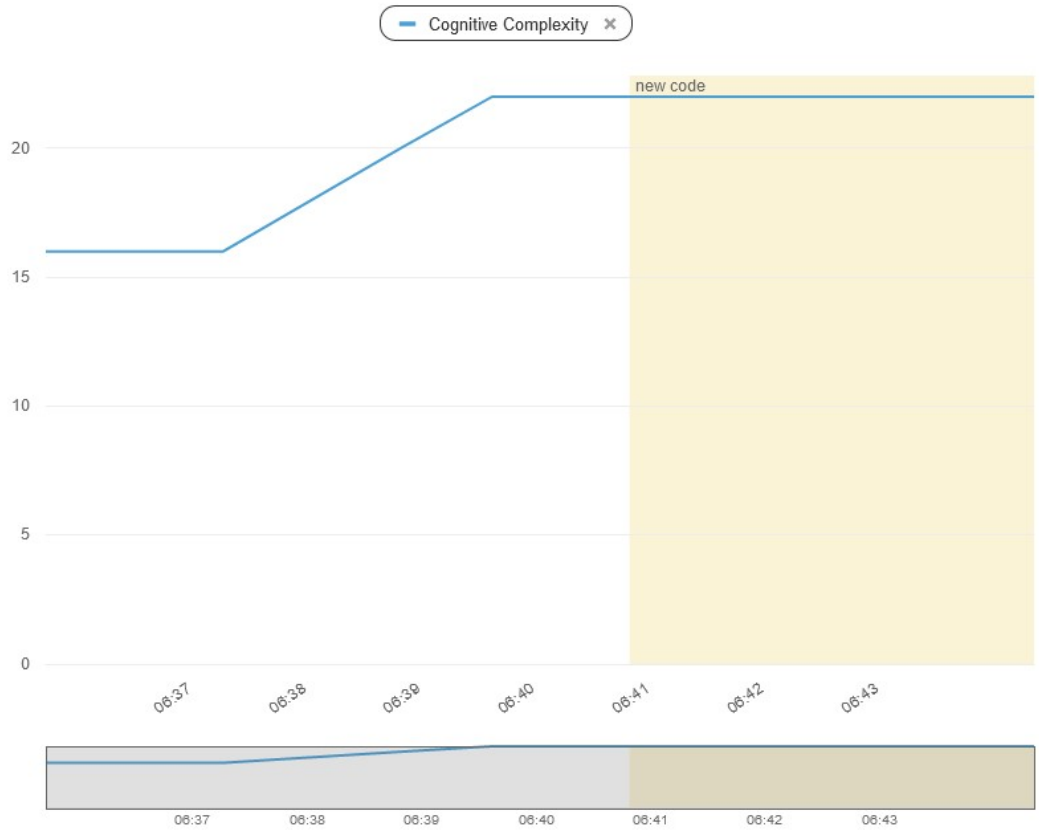


Εικόνα 4-5: : Γράφημα ιστορικής ανάλυσης τεχνικού χρέους για την ιδιότυπη υλοποίηση

4.1.3 Ποσοτικές μετρικές

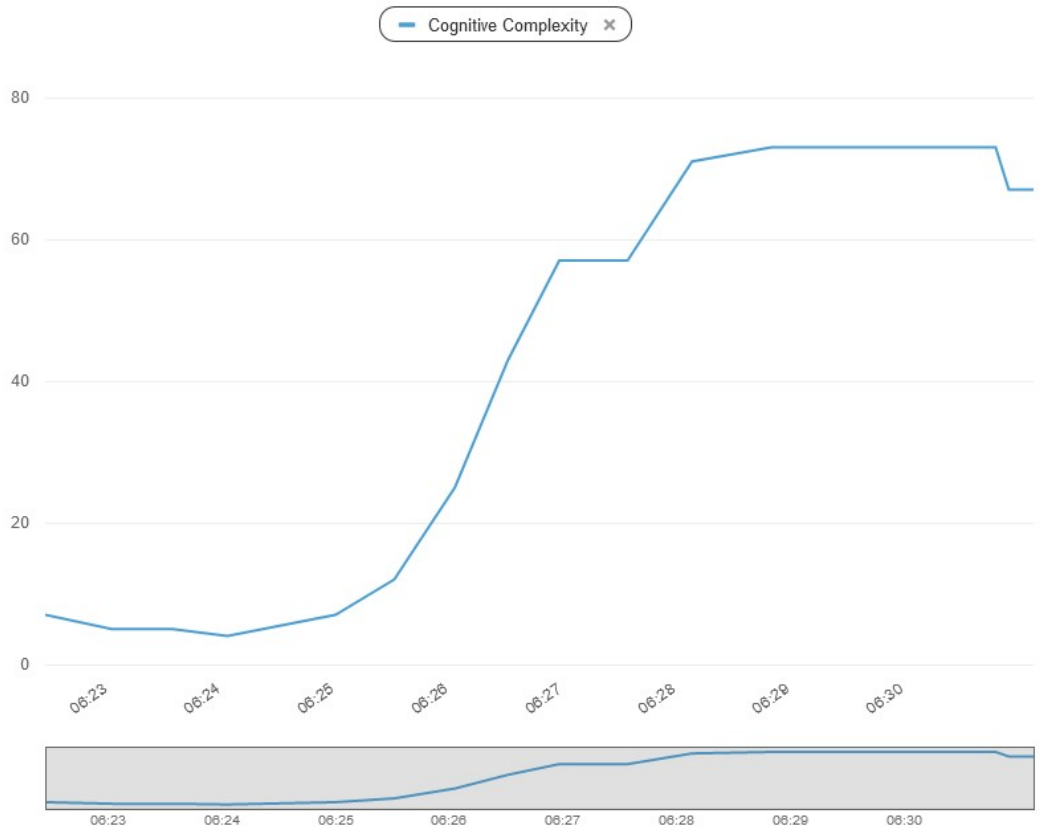
4.1.3.1 Cognitive complexity

Όπως φαίνεται στην εικόνα 4-6, η μετρική που αφορά το cognitive complexity παραμένει σταθερή στην περίπτωση της υλοποίησης με Spring-boot. Από πρακτικής σκοπιάς αυτό μπορεί να αποδοθεί στο γεγονός ότι στην εφαρμογή δεν εμφανίζεται κάποια ιδιαίτερη δυσκολία στην επιχειρηματική λογική (εφ'όσον έχουνε λυθεί ζητήματα αφαίρεσης, και άρα η δυσκολία έγκειται στο να υπακούσει ο προγραμματιστής στις απαιτήσεις του πλαισίου όσον αφορά τη συγγραφή κώδικα). Ως αποτέλεσμα, η συγγραφή κώδικα εκφυλίζεται στην σωστή διαχείριση των αφαιρέσεων που παρέχονται στην εφαρμογή από το πλαίσιο.



Εικόνα 4-6 : Cognitive complexity, σε υλοποίηση με Spring-boot

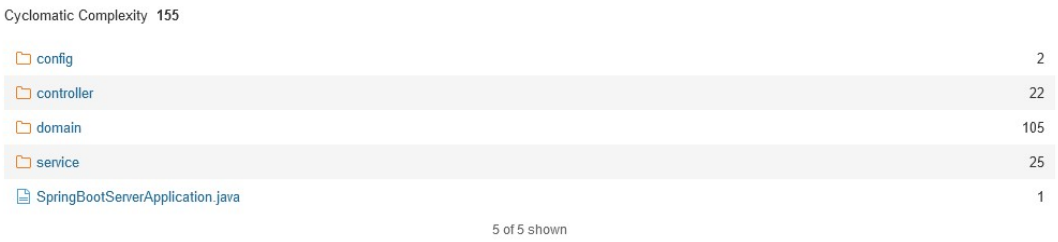
Στην εφαρμογή χωρίς τη χρήση του πλαισίου, από τη μετρική αυτή φαίνεται ότι όσο προχωράει η προσθήκη χαρακτηριστικών, τόσο αυξάνεται και η πολυπλοκότητα. Εφ'όσον η λειτουργικότητα είναι η ίδια και στις δύο εφαρμογές τότε είναι ασφαλές να συμπεράνουμε ότι οι αφαιρέσεις που δε μας παρέχονται (και άρα χρειάζεται να υλοποιήσουμε μόνοι μας) είναι αυτές που ανεβάζουνε την λογική πολυπλοκότητα. Αυτό είναι και λογικό, εάν λάβουμε υπ'όψιν μας ότι η υλοποίηση της αφαίρεσης από το πλαίσιο γίνεται με συνδρομή περισσότερων προγραμματιστών, με μεγαλύτερη εμπειρία στο ζήτημα, και με πολύ περισσότερο χρόνο για ανάλυση και υλοποίηση χαρακτηριστικών.



Εικόνα 4-7: Cognitive complexity σε υλοποίηση χωρίς πλαίσιο

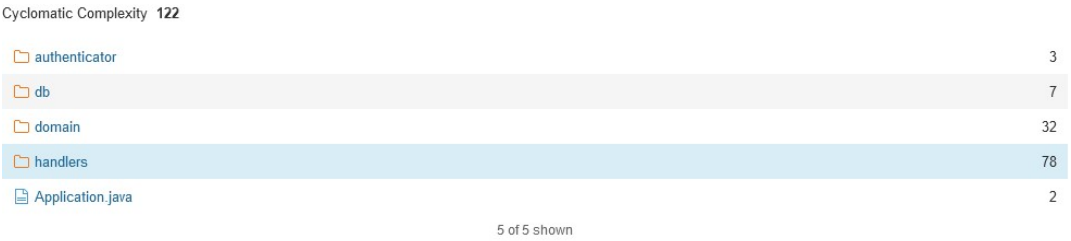
4.1.3.2 Cyclomatic Complexity

Πιο ενδιαφέροντα είναι τα αποτελέσματα στην μέτρηση της κυκλωματικής πολυπλοκότητας: Στην υλοποίηση με Spring-boot έχουμε αρκετά υψηλότερο νούμερο στη μετρική αυτή, απ'ό,τι στην υλοποίηση χωρίς κάποιο πλαίσιο. Το μεγαλύτερο κομμάτι της κυκλωματικής πολυπλοκότητας εμφανίζεται στην μοντελοποίηση των οντοτήτων που έχει γίνει απο το αντίστοιχο plugin στην υλοποίηση με Spring-boot. Πράγματι, στα συστήματα που χρησιμοποιούνε JPA, οι οντότητες με σύνθετο κλειδί απεικονίζονται σε διαφορετική οντότητα, πράγμα που έχει ως αποτέλεσμα την αύξηση της πολυπλοκότητας.



Εικόνα 4-8: Κυκλωματική πολυπλοκότητα στην υλοποίηση με spring-boot

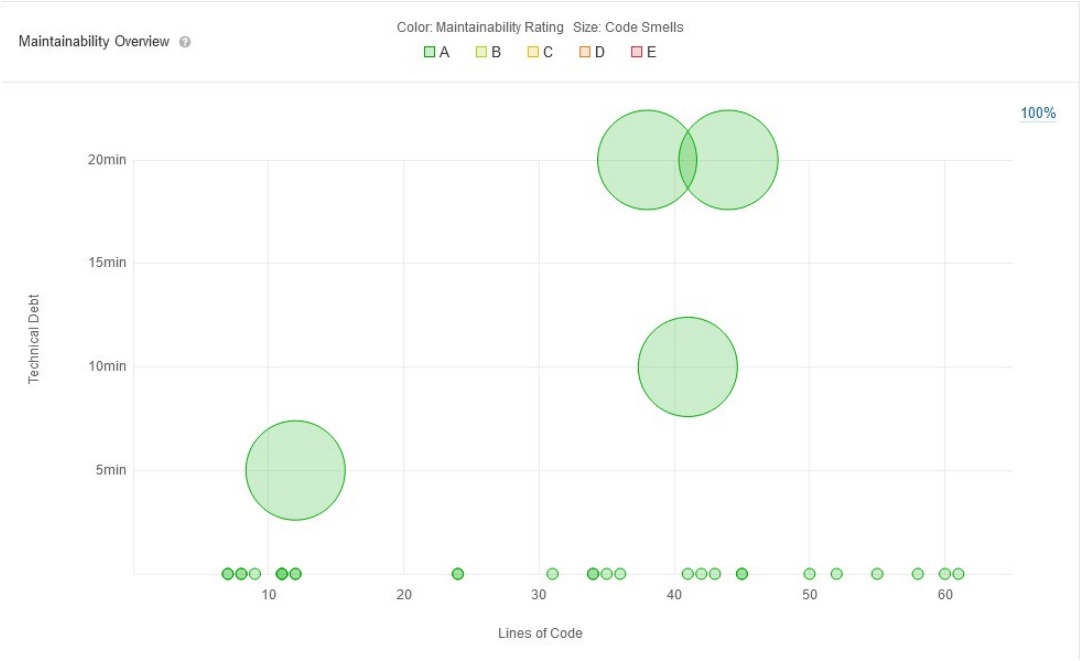
Στην περίπτωση της μη χρήσης πλαισίου, εφ'όσον τα ερωτήματα στη βάση γίνονται με sql που γράφεται από το χρήστη κατευθείαν, αυτό δίνει την δυνατότητα να ακολουθηθεί πιο ευθύς δρόμος όσον αφορά τη μοντελοποίηση των οντοτήτων. Αυτό πρακτικά σημαίνει ότι δεν εμφανίζεται ανάγκη για δημιουργία ξεχωριστών κλάσεων για τα σύνθετα κλειδιά, με αποτέλεσμα να μην υπάρχει υψηλή πολυπλοκότητα στις κλάσεις των μοντέλων. Αντιθέτως, το σημαντικότερο πακέτο που παρουσιάζει υψηλή πολυπλοκότητα είναι αυτό των handlers, όπου ουσιαστικά είναι συμπυκνωμένη και όλη η επιχειρηματική λογική (μαζί με τις υλοποιήσεις των ερωτημάτων SQL).



Εικόνα 4-9: Κυκλωματική πολυπλοκότητα με υλοποίηση χωρίς πλαίσιο

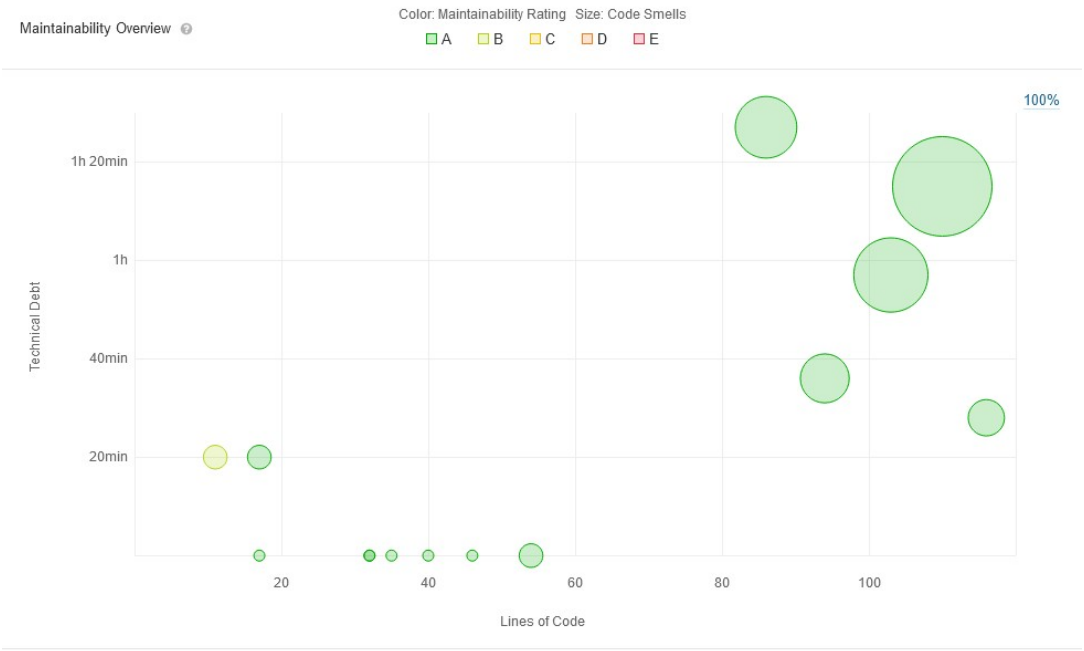
4.1.3.3 Maintainability Rating

Όσον αφορά το maintainability rating, επειδή είναι μια μετρική που επηρεάζεται άμεσα από το τεχνικό χρέος, παρατηρείται πάλι μια καλύτερη εικόνα στην υλοποίηση με πλαίσιο λογισμικού. Στην περίπτωση του Spring-boot παρατηρείται ότι τα κύρια σημεία που παρουσιάζουνε πρόβλημα στο rating αφορούν είτε τις περιπτώσεις διπλότυπου κώδικα, είτε τις περιπτώσεις κακής ονομασίας μεταβλητών. Παρ'όλο που υπάρχει μεγάλος όγκος των code smells (που προσδιορίζεται από το μέγεθος των κύκλων), η αναλογία κώδικα προς τεχνικό χρέος παραμένει αρκετά χαμηλή (το χρώμα των κύκλων είναι που οπτικοποιεί αυτή την μετρική).



Εικόνα 4-10 : Maintainability rating στην υλοποίηση με Spring-boot

Στην περίπτωση της υλοποίησης χωρίς πλαίσιο λογισμικού γίνεται πιο φανερή η έκταση του τεχνικού χρέους που έχει συσσωρευτεί κατά την ανάπτυξη της δημιουργίας του λογισμικού. Συγκεκριμένα, εκτός του ότι παρατηρείται μεγαλύτερος όγκος από οσμές κώδικα στην εφαρμογή, παρατηρούμε ότι υπάρχουνε και περιπτώσεις που η αναλογία τεχνικού χρέους είναι αρκετή ώστε να αποτελεί σημαντικό μέρος του μεγέθους μιας κλάσης.



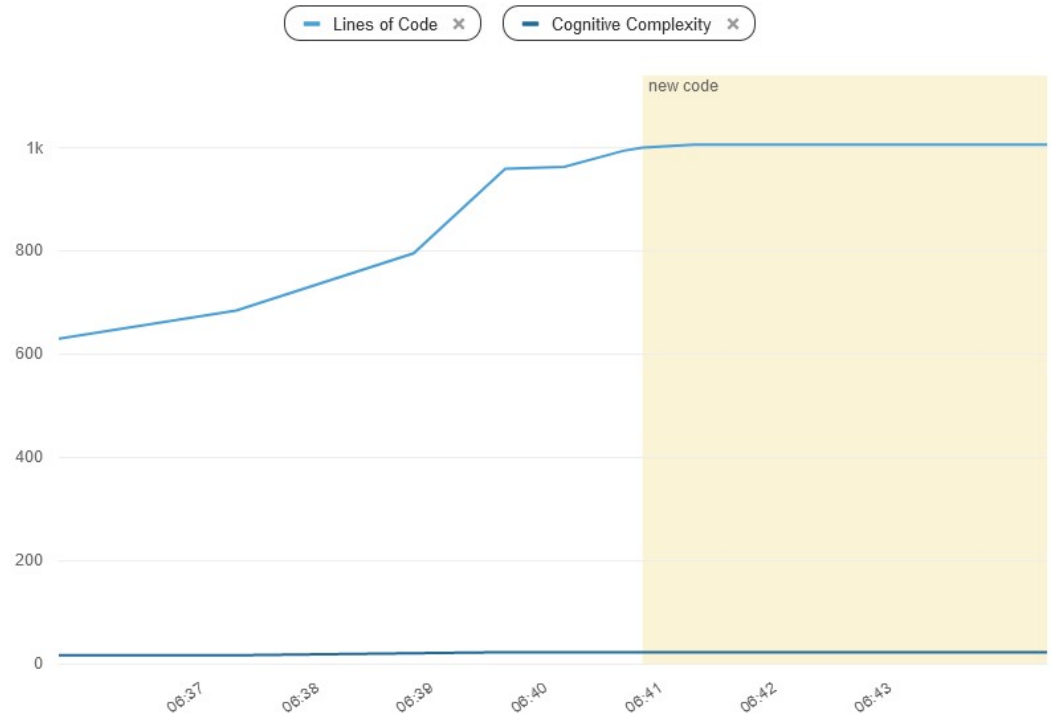
Εικόνα 4-11 : Maintainability rating στην υλοποίηση χωρίς Spring-boot

4.1.3.4 Lines of Code

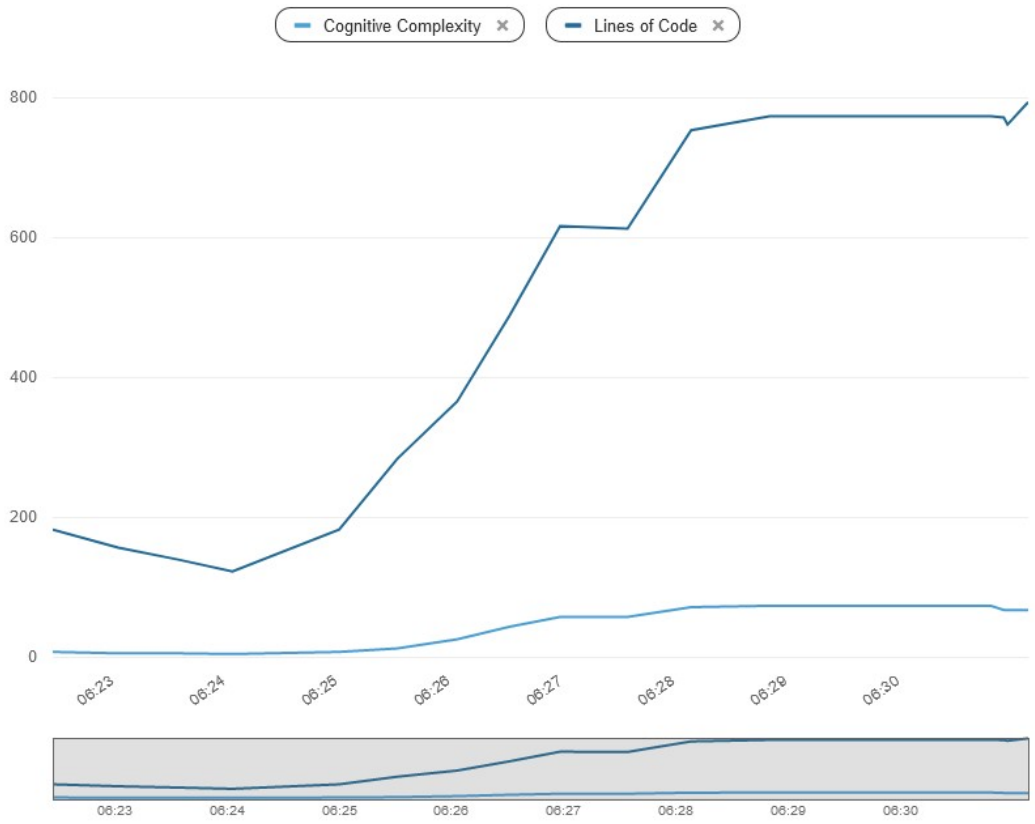
Όσον αφορά τον αριθμό των γραμμών κώδικα, παρατηρούμε ότι η ιδιότυπη υλοποίηση της εφαρμογής έχει σημαντικά λιγότερο αριθμό γραμμών κώδικα (1007 γραμμές κώδικα για την υλοποίηση σε spring-boot στην τελευταία γενιά, προς 773 γραμμές κώδικα στην ιδιότυπη υλοποίηση)· το ποσοστό της διαφοράς είναι κοντά στο 30%. Αυτή η διαφορά μπορεί να εξηγηθεί εάν παρατηρήσουμε την υλοποίηση της μοντελοποίησης στο spring-boot. Απ'όσο φαίνεται, επειδή οι οντότητες έχουνε γίνει auto-generated (όπως αναφέρθηκε και πιο πάνω) με τη χρήση αντίστοιχου plugin, υπάρχουν αρκετές γραμμές κώδικα που αφορούν getters/setters και κατασκευαστές, οι οποίες όμως ως επί το πλείστον δε χρησιμοποιούνται από την υλοποίηση. Αντιθέτως, επειδή η υλοποίηση χωρίς πλαίσιο λογισμισκού έγινε εξ' όλοκλήρου χειρωνακτικά, σε καθεμιά από τις δημιουργίες των οντοτήτων, δημιουργούνταν μόνο οι απαραίτητες μέθοδοι, με αποτέλεσμα να κρατηθεί το πλήθος των γραμμών κώδικα σε χαμηλές τιμές.

Αξίζει να σημειωθεί ότι παρ'όλο που η υλοποίηση με πλαίσιο λογισμικού έχει περισσότερες γραμμές κώδικα, η μετρική της λογικής πολυπλοκότητας διατηρείται

σταθερή καθ'όλη τη διάρκεια ανάπτυξης, ενώ αντιθέτως στην ιδιότυπη υλοποίηση υπάρχει μια αντιστοιχία στην αύξηση των γραμμών κώδικα με την αύξηση της πολυπλοκότητας. Αυτό μπορεί να αποδοθεί στο γεγονός ότι επειδή η οργάνωση της λογικής της εφαρμογής βασίζεται στο MVC, η οριζόντια επέκταση της εφαρμογής δε συμβάλλει στην αύξηση της δυσκολίας κατανόησης της. Αντιθέτως, επειδή η υλοποίηση της εφαρμογής χωρίς πλαίσιο έγινε με επιχειρηματικό διαχωρισμό (δηλαδή για κάθε οντότητα δημιουργείται μια ροή μόνο για αυτήν, και περνάει απ'όλα τα στάδια που αναλύονται στην παρούσα διπλωματική -http, διασύνδεση με ΒΔ, κτλ.) ο διαχωρισμός μεταξύ των επιμέρους κομματιών, αλλά και ο τρόπος υλοποίησης τους καταλήγουν να χρειάζονται μεγαλύτερη προσπάθεια (και κατ'επέκταση, αυξάνουνε τις μετρικές που συζητώνται).



Εικόνα 4-12 : Lines of Code, Cognitive Complexity στην υλοποίηση με Spring-boot



Εικόνα 4-13 : Lines of Code, Cognitive Complexity στην υλοποίηση χωρίς Spring-boot

4.1.3.5 Chidamber & Kemerer metrics

Όπως έχει αναφερθεί και στο κεφάλαιο 2, αυτή η «σουίτα» αποτελείται από 6 μετρικές, καθεμία από τις οποίες υπολογίζεται σε επίπεδο κλασης. Στην περίπτωση των δύο υλοποιήσεων, περισσότερο νόημα έχει να γίνει σύγκριση σε επίπεδο συνόλων (όπου εξάγονται και οι μέσες τιμές για την εκάστοτε μετρική), καθώς οι δύο εφαρμογές είναι διαφορετικά υλοποιημένες ως προς τον αριθμό κλάσεων και τη διάρθρωση τους. Στον παρακάτω πίνακα φαίνεται η μέση τιμή για καθεμια από τις 6 μετρικές, και πως αυτή διαμορφώνεται σε κάθε υλοποίηση:

Πίνακας 4-1: Σύγκριση μετρικών Chidamber-Kemerer για τις δυο υλοποιήσεις

	CBO	DIT	LCOM	NOC	RFC	WMC
Spring	3,09	1,05	2,00	0	8,00	6,55
Custom	3,46	1,54	1,92	0,38	17,23	8,23
A/B	0,9	0,68	1,04	0	0,46	0,79

Στη μετρική της σύζευξης μεταξύ αντικειμένων οι μετρήσεις δε δείχνουν κάποια σημαντική διαφορά ανάμεσα στις δυο μετρικές· παρ'όλα αυτά, γίνεται σαφές ότι υπάρχει μικρότερη σύζευξη μεταξύ κλάσεων στην υλοποίηση με Spring-boot απ'ό,τι σε αυτήν χωρίς υλοποίηση με τη βοήθεια πλαισίου. Στην μετρική του βάθους κληρονομικότητας τα πράγματα είναι ελαφρώς πιο ξεκάθαρα, και φαίνεται ότι η υλοποίηση με Spring έχει αρκετά μικρότερο ποσοστό, συμβάλλοντας έτσι και στη διατήρηση της λογικής πολυπλοκότητας σε χαμηλότερα επίπεδα. Στη μετρική που αφορά τον αριθμό κλάσεων-παιδιών πάλι φαίνεται να δείχνει το spring-boot ως την υλοποίηση που κρατάει τις τιμές στα χαμηλότερα επίπεδα, αλλά και το ποσοστό στην ιδιότυπη υλοποίηση φαίνεται να είναι κι αυτό σε επίπεδα που ενδεχομένως να επηρεάζουν ελάχιστα. Σημαντικό δείκτη αποτελεί η μετρική RFC (Response for Class) η οποία δείχνει ότι ειναι σαφώς πιο ξεκάθαρα διαρθρωμένη η υλοποίηση σε spring-boot, καθώς φαίνεται ότι κρατάει το επίπεδο της πολυπλοκότητας των κλάσεων σε χαμηλά επίπεδα (έστω κι αν ο αριθμός των κλάσεων είναι σαφώς μεγαλύτερος από αυτόν της ιδιότυπης υλοποίησης). Τέλος η μετρική που αφορά τον αριθμό των μεθόδων πάλι φαίνεται οτι η υλοποίηση με τη βοήθεια πλαισίου λογισμικού συμβάλλει στην διατήρηση του αριθμού των μεθόδων σε μια κλαση σε χαμηλό επίπεδο.

4.1.4 Ποιοτικές μετρικές

Οι ποιοτικές μετρικές που χρησιμοποιούνται εδώ αφορούνε τον χρόνο υλοποίησης (σε εργατοημέρες-man days), την ευκολία υλοποίησης (στην οποία λαμβάνονται υπ'όψιν παράγοντες όπως ύπαρξη τεκμηρίωσης, ύπαρξη forum για επίλυση απορίων σχετικά με την υλοποίηση, προπαρασκευαστική δουλειά που χρειάστηκε να γίνει πριν την ανάπτυξη -provisioning-), και την επεκτασιμότητα της εκάστοτε υλοποίησης.

Όσον αφορά τον χρόνο υλοποίησης, η διαδικασία ανάπτυξης με τη βοήθεια πλαισίου λογισμικού αποδείχθηκε σαφώς πιο γρήγορη (το κόστος υπολογίστηκε περίπου στις 2-3 εργατοημέρες). Κύριοι παράγοντες που συνέβαλλαν σε αυτό ήταν αφ'ενός η

εξοικείωση του γράφοντος με το Spring-boot, αλλά και με τα διάφορα εργαλεία που λειτουργούν ως πρόσθετα σε αυτό και επιταχύνουν τη διαδικασία ανάπτυξης λογισμικού. Πιο συγκεκριμένα, η ύπαρξη του εργαλείου Spring initializr[15] γλύτωσε σημαντικό χρόνο από τη διαδικασία της δημιουργίας του project, καθώς μέσω της γραφικής διεπαφής που παρέχει, προσφέρει γρήγορη δημιουργία ενός project, και επιταχύνει τη διαδικασία εισαγωγής των απαραίτητων βιβλιοθηκών για τη σωστή λειτουργία της εφαρμογής. Επιπλέον η χρήση του plugin jpaBuddy βοήθησε στην ταχεία μοντελοποίηση της βάσης δεδομένων και στη δημιουργία των αντίστοιχων οντοτήτων. Τέλος, το intellij Idea Ultimate έρχεται με ενσωματωμένα αρκετά plugins που αφορούν την ανάπτυξη λογισμικού με Spring (Spring data, Spring MVC, Spring Security, Spring Messaging κα), γεγονός που επίσης συνέβαλλε στην ταχύτητα ανάπτυξης λογισμικού. Όσον αφορά την ευκολία, τα παραπάνω plugins συνέβαλλαν δραστικά στην μείωση δυσκολίας της υλοποίησης, και για τις ελάχιστες περιπτώσεις που χρειάστηκε να γίνει αναζήτηση είτε στην τεκμηρίωση χρήσης κάποιου πακέτου, είτε στον τρόπο με τον οποίο υλοποιεί κάποια από τις αφαιρέσεις το spring-boot, οι διαθέσιμες πηγές ήταν πολλές σε αριθμό, και με διεξοδική ανάλυση (και αρκετά παραδείγματα). Ως παράδειγμα, η ιστοσελίδα https://www.baeldung.com/[19] παρείχε πληθώρα από άρθρα που αφορούν συγκεκριμένα τη χρήση του spring-boot, και καλύπτουνε ένα μεγάλο εύρος ζητημάτων που μπορεί να αντιμετωπίσει ένας προγραμματιστής κατά τη διάρκεια ανάπτυξης με το συγκεκριμένο πλαίσιο.

Στην περίπτωση της ιδιότυπης υλοποίησης ο χρόνος που χρειάστηκε ήταν ελαφρώς περισσότερος (3-4 εργατοημέρες), καθώς εκτός του ότι έπρεπε να γίνει και σχεδιασμός της εφαρμογής ως προς τη διάρθρωση της, και η ίδια η υλοποίηση έκρυβε αρκετά προβλήματα τα οποία έπρεπε να αντιμετωπιστούν (πχ, στην πρώτη γενία της, η εφαρμογή δε χρησιμοποιούσε PreparedStatements, με αποτέλεσμα να παρουσιάζει σοβαρό πρόβλημα ασφαλείας, καθώς η μη χρήση τους μπορεί να καταλήξει σε περιπτώσης sql injection). Ένα άλλο παράδειγμα που αφορούσε τη διασύνδεση με τη βάση δεδομένων, είναι η υλοποίηση της κλάσης που συνδέεται με τη ΒΔ. Σε αυτή την περίπτωση, και για αρκετές γενιές, λόγω παράλειψης από την πλευρά του προγραμματιστή, έμενε ανοικτή η σύνδεση με τη βάση δεδομένων ακόμη κι όταν τελείωναν τα ερωτήματα προς αυτήν, γεγονός που συνέβαλλε στο να δημιουργείται memory leak.

Στην περίπτωση της υλοποίησης σε επίπεδο HTTP, η διεπαφή που προσφέρει το πακέτο com.sun.net.http αποδείχθηκε ελαφρώς πιο στρυφνό στην υλοποίηση, και ξοδεύτηκε αρκετός χρόνος ώστε να γίνει μια αναδιάρθρωση της ιεραρχίας των κλάσεων ώστε να παραχθεί μια λύση με όσο το δυνατόν λιγότερο διπλότυπο κώδικα.

Συνοψίζοντας, όσον αφορά τις ποιοτικές μετρικές, η ανάπτυξη με το πλαίσιο λογισμικού αποδείχθηκε πιο εύκολη και με λιγότερη εργασία. Είναι ασφαλές να πούμε ότι μεγάλο μέρος της ευκολίας αυτής προέκυψε από το γεγονός ότι ο γράφων είχε πρότερη εμπειρία από χρήση πλαισίων λογισμικού, και δεν είχε τα συνήθη προβλήματα που παρουσιάζονται στους προγραμματιστές όταν χρησιμοποιούν για πρώτη φορά πλαίσια λογισμικού. Από την άλλη, η ιδιότυπη υλοποίηση είχε μια πιο ευθεία προσέγγιση στον τρόπο με τον οποίο υλοποιήθηκε η λειτουργικότητα της, και έδωσε την δυνατότητα αποφυγής δημιουργίας περιττού κώδικα. Σε περιπτώσεις λοιπόν που χρειάζεται να ληφθεί απόφαση για το αν θα χρησιμοποιηθεί ή όχι πλαίσιο λογισμικού στην ανάπτυξη μιας εφαρμογής, θα πρέπει να λαμβάνεται υπ’όψιν και η εμπειρία της συγγραφικής ομάδας όσον αφορά τη χρήση πλαισίων, καθώς η καμπύλη εκμάθησης τους είναι αρκετά μεγάλη, και δεν είναι ξεκάθαρο εάν συμφέρει να αναλωθεί χρόνος στην εκμάθηση, ειδικά εάν η εφαρμογή δεν είναι μεγάλης κλίμακας.

5 Όρια και περιορισμοί της έρευνας

Από τις υλοποιήσεις που παρουσιάστηκαν, είναι προφανές ότι υπάρχει ένα επίπεδο αφαίρεσης που χρησιμοποιείται ακόμα και στην πιο απλή εκδοχή της εφαρμογής. Συγκεκριμένα η χρήση του πακέτου com.sun.net.HttpServer, παρ'όλο που αποτελεί την πιο απλή εκδοχή της υλοποίησης ενός εξυπηρετητή, παρέχει αρκετές αφαιρέσεις και κρύβει αρκετό κώδικα, ο οποίος θα μπορούσε να υλοποιηθεί ξεχωριστά, ώστε να υπάρξουνε παραπάνω δεδομένα στις μετρικές και τη σύγκριση μεταξύ των διαφόρων προσεγγίσεων. Επιπλέον, δεν έχει εξερευνηθεί η διαδικασία του testing όσον αφορά τις δύο προσεγγίσεις, και το κατά πόση διαφορά έχει η υλοποίηση σουίτας δοκιμών με τη χρήση πλαισίου και χωρίς αυτήν.

5.1 Μελλοντικές Επεκτάσεις

Μια ενδιαφέρουσα οπτική για την επέκταση αυτής της εργασίας θα μπορούσε να είναι η υλοποίηση αυτής της εφαρμογής χωρίς τη στήριξη από κανένα πακέτο, πέρα από αυτά που προσφέρονται με το Java Development Kit. Σε τέτοια περίπτωση, γίνεται αναγκαία και η υλοποίηση λογικής για τη δημιουργία νημάτων που θα αναλαμβάνουνε να εξυπηρετήσουνε τα αιτήματα που έρχονται από τα προγράμματα-πελάτες. Επιπλέον θα πρέπει να δοθεί και υλοποίηση για την δημιουργία των διαύλων (Sockets) επικοινωνίας ανάμεσα στα προγράμματα πελάτες και τον εξυπηρετητή. Μια τέτοιου είδους προσέγγιση σίγουρα θα μπορεσει να δώσει ακόμα περισσότερες πληροφορίες και μετρικές για να απαντήσει περαιτέρω στο ερώτημα της παρούσας διπλωματικής. Έχοντας και τις τρεις υλοποιήσεις μπορεί να γίνει μια σύγκριση μεταξύ τους για να διαπιστωθεί κατά πόσο και από ποιο σημείο αφαίρεσης κι έπειτα αρχίζει να συμφέρει η χρήση πλαισίων λογισμικού έναντι ιδιότυπων υλοποιήσεων.

Βιβλιογραφία

[1] D. Riehle Dipl, "Framework Design A Role Modeling Approach," 2000.

[2] G. Kiczales, "Towards a New Model of Abstraction in Software Engineering," 1992.

[3] J. Greenfield and K. Short, "Software Factories Assembling Applications with Patterns, Models, Frameworks and Tools".

[4] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns, Elements of Reusable OO Software (CD), 2001.

[5] O. Vogel, A. Ingo, C. Arif and K. Timo, Software Architecture: A Comprehensive Framework and Guide for Practitioners, Springer, 2011.

[6] T. Reenskaug "The Model-View-Controller (MVC) Its Past and Present", 2003

[7] T. McCabe, Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric, IEEE Computer Society Press, 1983.

[8] R. E. Johnson and B. Foote, "Designing Reusable Classes," June/July 1988. [Online]. Available: http://www.laputan.org/drc/drc.html. [Accessed 15 February 2022].

[9] M. Fowler, "Inversion of Control Containers and the Dependency Injection pattern," 23 January 2004. [Online]. Available: https://www.martinfowler.com/articles/injection.html. [Accessed 15 February 2022].

[10] M. Fowler, "Inversion of Control," 26 June 2005. [Online]. Available: https://martinfowler.com/bliki/InversionOfControl.html. [Accessed 15 Februrary 2022].

[11] P. Beynon-Davies, Database Systems: Third Edition, Red Globe Press, 2003.

[12] "The OAuth 2.0 Authorization Framework: Bearer Token Usage," October 2012. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6750. [Accessed 15 February 2022].

[13] "Technical Debt," 20 March 2017. [Online]. Available: https://www.techopedia.com/definition/27913/technical-debt. [Accessed 15 February 2022].

[14] "SUN SHIPS JDK 1.1 -- JAVABEANS INCLUDED," [Online]. Available: https://web.archive.org/web/20080210044125/http://www.sun.com/smi/Press/sunflas

h/1997-02/sunflash.970219.0001.xml. [Accessed 15 February 2022].

[15] VMware, Inc, "spring initializr," [Online]. Available: https://start.spring.io/. [Accessed 15 February 2022].

[16] SonarSource S.A., "SonarQube," [Online]. Available: http://www.sonarqube.org/. [Accessed 15 February 2022].

[17] Oracle, "Module java.sql," [Online]. Available: https://docs.oracle.com/javase/10/docs/api/java.sql-summary.html. [Accessed 15 February 2022].

[18] "Metric Definitions," SonarSource , [Online]. Available: https://docs.sonarqube.org/latest/user-guide/metric-definitions/. [Accessed 15 February 2022].

[19] Baeldung, "Learn Spring Boot," [Online]. Available: https://www.baeldung.com/spring-boot. [Accessed 15 February 2022].

[20] "JpaBuddy," [Online]. Available: https://www.jpa-buddy.com/. [Accessed 15 February 2022].

[21] Jetbrains s.r.o., "Intellij Idea," Jetbrains s.r.o., [Online]. Available: https://www.jetbrains.com/idea/. [Accessed 15 February 2022].

[22] "Chidamber & Kemerer object-oriented metrics suite," [Online]. Available: https://www.aivosto.com/project/help/pm-oo-ck.html. [Accessed 15 February 2022].

[23] " What is Object/Relational Mapping?," [Online]. Available: http://hibernate.org/orm/what-is-an-orm/. [Accessed 15 February 2022].

[24] G. A. Campbell, "Cognitive Complexity, A new way of measuring understandability," 2021. [Online]. Available: https://www.sonarsource.com/resources/white-papers/cognitive-complexity/. [Accessed 15 February 2022].

[25] J.Spolsky, "The Law of Leaky Abstractions", 11 November 2002. [Online]. Available: https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/. [Accessed February 2022]

Κυρώσεις για λογοκλοπή

Η λογοκλοπή είναι ένα πολύ σοβαρό παράπτωμα. Με απόφαση της ΓΣΕΣ φοιτητής που διαπιστώνεται ότι υποπίπτει σε λογοκλοπή κατά την εκπόνηση της διπλωματικής του εργασίας αποβάλλεται από το ΠΜΣ. Εάν έχει ήδη αποφοιτήσει ανακαλείται το Μεταπτυχιακό δίπλωμα Ειδίκευσης και προωθείται το θέμα στο Δικαστικό Γραφείο του Πανεπιστημίου για την έναρξη των ανάλογων νομικών διαδικασιών.