ΠΑΝΕΠΙΣΤΗΜΙΟ ΜΑΚΕΔΟΝΙΑΣ
ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΤΜΗΜΑΤΟΣ ΕΦΑΡΜΟΣΜΕΝΗΣ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΑΝΑΠΤΥΞΗ ΕΦΑΡΜΟΓΩΝ ΜΕ ΑΡΧΙΤΕΚΤΟΝΙΚΗ ΜΙΚΡΟΥΠΗΡΕΣΙΩΝ

Διπλωματική Εργασία

του

Νικόλαου Μπάγια

Θεσσαλονίκη, Ιούνιος 2020

ΑΝΑΠΤΥΞΗ ΕΦΑΡΜΟΓΩΝ ΜΕ ΑΡΧΙΤΕΚΤΟΝΙΚΗ ΜΙΚΡΟΥΠΗΡΕΣΙΩΝ

Νικόλαος Μπάγιας

Πτυχίο Εφαρμοσμένης Πληροφορικής, ΠΑΜΑΚ, 2015

Διπλωματική Εργασία

υποβαλλόμενη για τη μερική εκπλήρωση των απαιτήσεων του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΤΙΤΛΟΥ ΣΠΟΥΔΩΝ ΣΤΗΝ ΕΦΑΡΜΟΣΜΕΝΗ ΠΛΗΡΟΦΟΡΙΚΗ

Επιβλέπων Καθηγητής
Κωνσταντίνος Μαργαρίτης

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την

Κωνσταντίνος Μαργαρίτης	Ηλίας Σακελλαρίου	Ελευθέριος Μαμάτας
.....	.....	.....

Νικόλαος Μπάγιας

.....

Περίληψη

Η αρχιτεκτονική των μικρουπηρεσιών κερδίζει ολοένα και περισσότερο έδαφος τα τελευταία χρόνια. Όσο ωριμάζει και δημιουργεί το δικό της οικοσύστημα κερδίζει το ακαδημαϊκό ενδιαφέρον για τα όσα έχει να προσφέρει ως προσέγγιση στην ανάπτυξη λογισμικού. Αυτή η εργασία δίνει μεγαλύτερη έμφαση στη σκοπιά του προγραμματιστή, παρουσιάζοντας τόσο το θεωρητικό υπόβαθρο που απαιτείται για την υλοποίηση ενός τέτοιου συστηματος, όσο και μια δειγματική εφαρμογή. Για μια πιο ολοκληρωμένη παρουσίαση, χρησιμοποιήθηκαν διάφορα εργαλεία που, αν και προαιρετικής φύσεως, συμβάλλουν τα μέγιστα σε μια επιτυχημένη μεθοδολογία ανάπτυξης. Αξιοποιήθηκε η τεχνολογία των containers μέσω Docker, πρακτικές Συνεχούς Ενσωμάτωσης με Jenkins, προσέγγιση του προβλήματος με την θεωρία του Domain-Driven-Design, καθώς και διάφορες προγραμματιστικές εργαλειοθήκες. Ο στόχος αυτής της διπλωματικής εργασίας είναι να αποτελέσει ένα καλό σημείο εκκίνησης για την εξοικείωση με τις μικρουπηρεσίες.

Λέξεις Κλειδιά: μικρουπηρεσίες, containers, Docker, Domain-Driven-Design, Jenkins, PostgreSQL, Redis, REST, gRPC, Java, Go, Spring Boot

Abstract

The microservices architecture is steadily increasing in popularity over the past few years. As it reaches a greater level of maturity, there is an academic interest to explore what it has to offer on software development. This study focuses on the developer's perspective of microservices, attempting to clarify the theoretical background needed to develop such a system and, also, present a sample application. For a more complete approach, we used various tools and practices that, although optional, play a vital role in ensuring a good developer experience. These include containers with Docker, Continuous Integration practices with Jenkins, a Domain-Driven-Design approach and various programming frameworks. This dissertation aims to be a useful starting point for getting familiar with microservices.

Keywords: microservices, containers, Docker, Domain-Driven-Design, Jenkins, PostgreSQL, Redis, REST, gRPC, Java, Go, Spring Boot

Πρόλογος - Ευχαριστίες

Η εκπόνηση της διπλωματικής εργασίας πραγματοποιήθηκε στα πλαίσια του μεταπτυχιακού προγράμματος σπουδών του Τμήματος Εφαρμοσμένης Πληροφορικής του Πανεπιστημίου Μακεδονίας, υπό την επίβλεψη του καθηγητή κ. Κωνσταντίνου Μαργαρίτη. Αφορμή για την ενασχόληση με την αρχιτεκτονική των μικρουπηρεσιών στάθηκε τόσο η προτροπή του κ. Μαργαρίτη, όσο και προσωπικό ενδιαφέρον για αυτήν. Η διαδικασία της συγγραφής δεν ήταν πάντα εύκολη, με πεισματικά σφάλματα κώδικα να λειτουργούν αποθαρρυντικά, αλλά με επιμονή και υπομονή ξεπεράστηκαν τα όποια προβλήματα.

Θα ήθελα να ευχαριστήσω τον κ. Μαργαρίτη που με καθοδήγησε κατά την διάρκεια της εργασίας, έδειξε κατανόηση και με εμπιστεύτηκε για την ολοκλήρωση της.

Τέλος, θα ήθελα να εκφράσω την ευγνωμοσύνη μου για όλα όσα μου έδωσε το πρόγραμμα μεταπτυχιακών σπουδών στη διάρκεια της φοίτησης μου. Η έκθεση σε σημαντικές έννοιες και τεχνολογίες της πληροφορικής διεύρυνε τους ορίζοντες μου και μου έμαθε πως πρέπει να προσπαθώ να γίνομαι καλύτερος συνέχεια.

Περιεχόμενα

Κατάλογος Εικόνων

1 Εισαγωγή

1.1 Πρόβλημα - Σημαντικότητα του θέματος

Τα τελευταία χρόνια η αρχιτεκτονική των μικρουπηρεσιών έχει αρχίσει να βρίσκει μεγάλη ανταπόκριση από τους μηχανικούς λογισμικού. Αν και δεν αποτελεί καινούργια ιδέα στο χώρο της πληροφορικής τα σημεία στα οποία δίνει έμφαση είναι αυτά που εξυπηρετούν καλύτερα τις σύγχρονες ανάγκες ανάπτυξης λογισμικού. Ένα σύστημα μικρουπηρεσιών οργανώνεται, συνήθως, γύρω από επιχειρησιακές έννοιες τονίζοντας την ανάγκη για γρήγορη ανάπτυξη και κυκλοφορία νέων χαρακτηριστικών προϊόντος. Ταυτόχρονα, δημιουργεί την απαίτηση για ευελιξία και προσαρμοστικότητα από την μεριά του συστήματος ώστε να ακολουθεί την επιχειρηματική πραγματικότητα. Αυτοί οι ταχείς ρυθμοί δεν συμβαδίζουν με τον παραδοσιακό τρόπο ανάπτυξης λογισμικού, που απαιτεί εξαντλητική ανάλυση των απαιτήσεων και μια ολιστική προσέγγιση στα τεχνικά χαρακτηριστικά του συστήματος. Έτσι οι μικρουπηρεσίες συνοδεύονται συχνά με ευέλικτες πρακτικές ανάπτυξης λογισμικού. Τέλος, όπως θα αναφερθεί και σε ακόλουθη ενότητα, ένα σύστημα που δεν έχει τις ανάγκες που εξυπηρετούν οι μικρουπηρεσίες ή δεν μπορεί να σηκώσει το διαχειριστικό τους κόστος δεν πρόκειται να επωφεληθεί αλλά, ίσως, και να ζημιωθεί από την υιοθέτηση τους.

1.2 Σκοπός - Στόχοι

Ο σκοπός αυτής της εργασίας είναι να παρουσιάσει την αρχιτεκτονική των μικρουπηρεσιών σε θεωρητικό επίπεδο δίνοντας, όμως, ιδιαίτερη βαρύτητα στο τεχνικό κομμάτι. Για αυτό, πέρα από την ανάπτυξη δειγματικής εφαρμογής θα παρουσιαστούν και τεχνολογίες που συχνά συμπεριλαμβάνονται στο οικοσύστημα της και προσπαθούν να απαντήσουν στις προκλήσεις που αντιμετωπίζει. Επίσης, είναι σημαντικό να γίνει κατανοητό ότι η αρχιτεκτονική των μικρουπηρεσιών αποτελεί άλλο ένα εργαλείο στην εργαλειοθήκη του μηχανικού λογισμικού και δεν πρέπει να εφαρμόζεται απλά επειδή είναι η δημοφιλής προσέγγιση.

1.3 Συνεισφορά

Πριν την συγγραφή της εργασίας δόθηκε αρκετός χρόνος στην μελέτη αναγνωρισμένων βιβλίων πάνω στο θέμα. Λόγω της περιορισμένης βιβλιογραφίας, η ερευνητική διαδικασία εμπλουτίστηκε με ανάγνωση αρκετών ηλεκτρονικών άρθρων και μελετών.

1.4 Διάρθρωση της μελέτης

Το κεφάλαιο 2 είναι αφιερωμένο στην θεωρητική μελέτη της αρχιτεκτονικής των μικρουπηρεσιών, παρουσιάζοντας τα χαρακτηριστικά τους σε σύγκριση με άλλες προσεγγίσεις στο χώρο της αρχιτεκτονικής λογισμικού. Στο κεφάλαιο 3 παρουσιάζεται ο τρόπος ανάπτυξης που ακολουθήθηκε για τη δημιουργία της δειγματικής εφαρμογής, τόσο σχεδιαστικά όσο και τεχνικά. Το κεφάλαιο 4 περιλαμβάνει την υλοποίηση της εφαρμογής και στο κεφάλαιο 5 κλείνουμε με συμπεράσματα και μελλοντικές βελτιώσεις.

2 Βιβλιογραφική Επισκόπηση - Θεωρητικό Υπόβαθρο

2.1 Μικρουπηρεσίες - Τι είναι

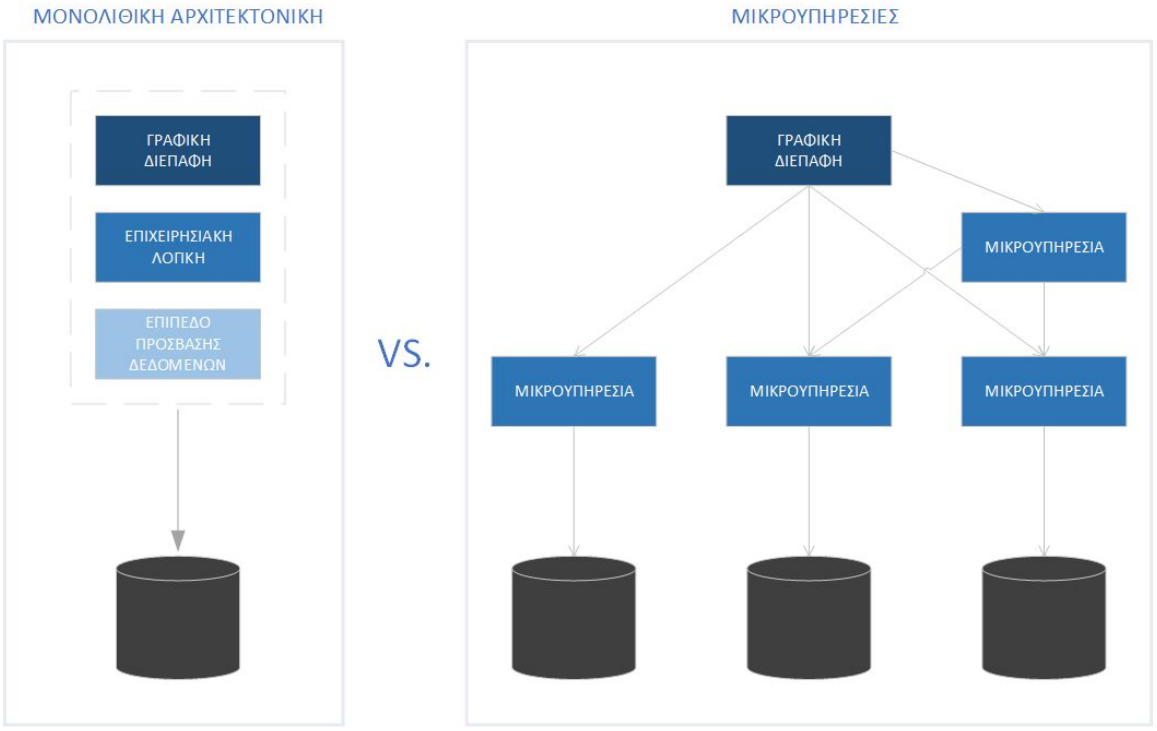
Οι μικρουπηρεσίες δεν έχουν κάποιο κοινώς αποδεκτό ορισμό. Σύμφωνα με την IBM, οι μικρουπηρεσίες είναι μία αρχιτεκτονική προσέγγιση στην οποία μία εφαρμογή αποτελείται από πολλά μικρότερα μέρη που έχουν χαμηλή σύζευξη μεταξύ τους και έχουν την δυνατότητα να αλλάζουν ανεξάρτητα[1]. Αυτά τα μέρη, ή πιο σωστά υπηρεσίες, έχουν συνήθως δική τους τεχνολογική στοίβα (stack) και δικό τους μοντέλο δεδομένων, είναι οριοθετημένα γύρω από επιχειρησιακές έννοιες και επικοινωνούν μεταξύ τους με ένα συνδυασμό τεχνολογιών. Για τα δύο τελευταία χαρακτηριστικά θα μιλήσουμε με μεγαλύτερη λεπτομέρεια στα κεφάλαια 2.3 και 2.4 αντίστοιχα.

Το αρχιτεκτονικό στυλ των μικρουπηρεσιών δεν υπαγορεύει ούτε απαγορεύει κάποιο συγκεκριμένο προγραμματιστικό μοντέλο. Αρκείται στο να προσφέρει κατευθυντήριες γραμμές για την διαίρεση των τμημάτων μιας κατανεμημένης εφαρμογής σε ανεξάρτητες οντότητες, με την κάθε μία να ασχολείται μόνο με όσα βρίσκονται εντός των ορίων της. Έτσι, κάθε μικρουπηρεσία, προσφέροντας την λειτουργικότητα της μέσω μιας ορισμένης μορφής μηνυμάτων, μπορεί να έχει υλοποιηθεί με οποιαδήποτε γλώσσα προγραμματισμού ή προγραμματιστικό μοντέλο[2].

Η στρατηγική που εφαρμόζει η αρχιτεκτονική των μικρουπηρεσιών έχει τις ρίζες της στην φιλοσοφία του Unix. Σύμφωνα με αυτήν, ένα πρόγραμμα πρέπει να ασχολείται με μία μόνο εργασία και να την ολοκληρώνει σωστά και αξιόπιστα. Επίσης πρέπει να είναι σε θέση να συνεργάζεται με άλλα προγράμματα μέσω μιας γνωστής διεπαφής[3]. Είναι εύκολο να δούμε πως αυτή η προσέγγιση έχει επηρεάσει τις μικρουπηρεσίες που, ουσιαστικά, προσπαθούν να εφαρμόσουν αυτήν την φιλοσοφία σε επίπεδο διεργασιών.

2.2 Μονολιθική Αρχιτεκτονική

Η μονολιθική αρχιτεκτονική θεωρείται ο παραδοσιακός τρόπος ανάπτυξης ενός συστήματος λογισμικού. Σε αυτές τις εφαρμογές όλη η λειτουργικότητα αναπτύσσεται σε μία ενιαία βάση κώδικα που χρησιμοποιείται από όλους τους εμπλεκόμενους προγραμματιστές. Αυτό σημαίνει ότι αλλαγές ή προσθήκες σε ένα λειτουργικά ανεξάρτητο κομμάτι του συστήματος πρέπει να συνοδεύονται από έλεγχο ότι οι υπόλοιπη λειτουργικότητα παραμένει αμετάβλητη, αλλα και απο συντονισμό με άλλες ομάδες[4].



Εικόνα 1: Σύγκριση Μονολιθικής Αρχιτεκτονικής με Μικρουπηρεσίες

Μία άλλη συνέπεια της μονολιθικής προσέγγισης είναι πως κρίσιμα σφάλματα σε μία υπηρεσία (με την στενή έννοια ως αρθρωτή λειτουργικότητα) προκαλούν την ολοκληρωτική κατάρρευση του συστήματος. Έτσι, η ύπαρξη ενός κεντρικού σημείου αποτυχίας (single point of failure) μειώνει τις δυνατότητες του συστήματος να μεγιστοποιήσει την διαθεσιμότητα του, μια σημαντική απαίτηση για πολλές εφαρμογές. Φυσικά, το ίδιο μπορεί να συμβεί και σε συστήματα μικρουπηρεσιών ως αποτέλεσμα

κακού σχεδιασμού, αλλά σίγουρα δεν αποτελεί εγγενές χαρακτηριστικό της αρχιτεκτονικής.

Επιπρόσθετα, οι επιλογές για αυτόματη κλιμάκωση του συστήματος είναι αρκετά περιορισμένες. Η κάλυψη των υπολογιστικών αναγκών μιας απαιτητικής υπηρεσίας συνήθως περιλαμβάνει την ποσοτική ή/και ποιοτική αναβάθμιση της εφαρμογής (οριζόντια/κάθετη κλιμάκωση). Από τη στιγμή που η υπηρεσία συνοδεύεται από όλη την υπόλοιπη λειτουργικότητα η τελική υπολογιστική ανάγκη είναι μεγαλύτερη από την πραγματική, διογκώνοντας έτσι τα κόστη εκτέλεσης και συντήρησης. Αν η υπηρεσία ήταν απομονωμένη, όπως προτείνεται στην αρχιτεκτονική των μικρουπηρεσιών, θα μπορούσε να κλιμακωθεί ανεξάρτητα και ακριβώς όσο χρειαζόταν, βελτιστοποιώντας τα κόστη του συστήματος.

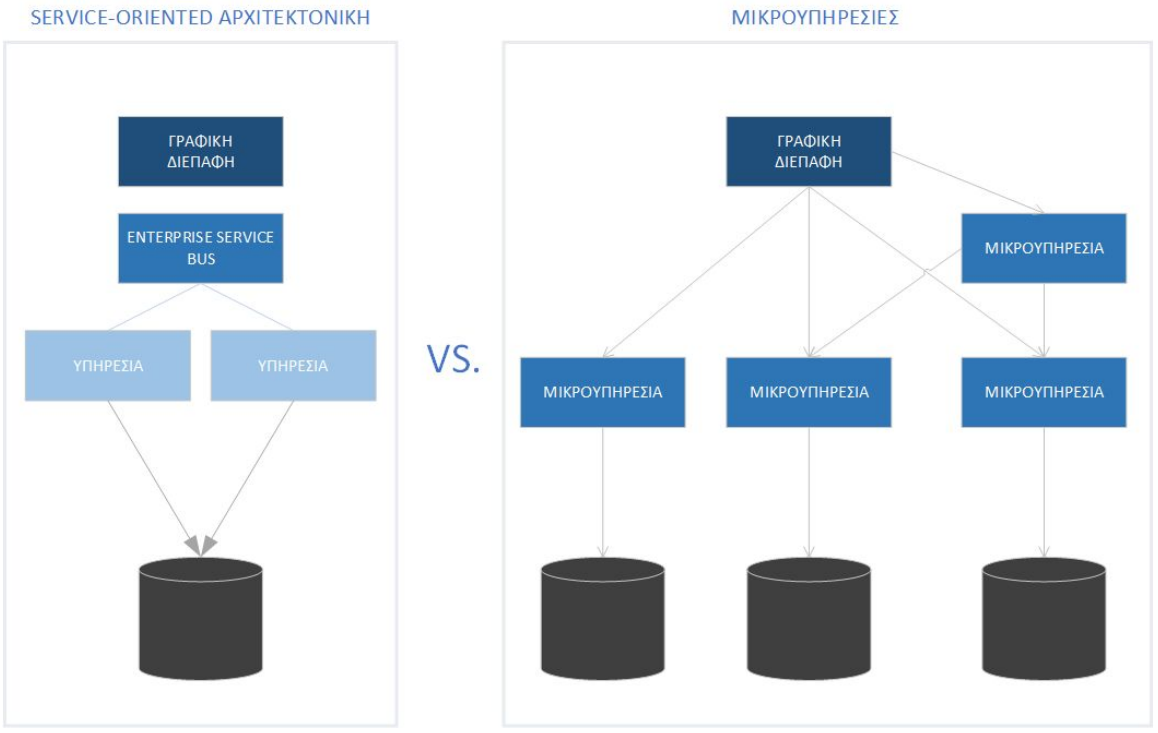
Με όλα τα παραπάνω είναι πιθανό κάποιος να συμπεράνει πως η μονολιθική αρχιτεκτονική δεν είναι καλή επιλογή για την ανάπτυξη λογισμικού. Όμως πέρα από αδυναμίες έχει και δυνατά σημεία. Ένα από αυτά είναι πως η διαχείριση της εφαρμογής είναι πιο εύκολη καθώς βασικές λειτουργίες όπως συναρμογή διαγνωστικών δεδομένων, δρομολόγηση αιτημάτων ή διαχείριση προσωρινών δεδομένων έχουν πιο απλή υλοποίηση. Επίσης, στις μικρουπηρεσίες, η έντονη χρήση του δικτύου για την εσωτερική επικοινωνία της εφαρμογής δημιουργεί πρόσθετες προκλήσεις κατά την διαδικασία της αποσφαλμάτωσης και δοκιμής, με τις μονολιθικές εφαρμογές να μην επηρεάζονται από αυτήν την κατηγορία προβλημάτων τόσο έντονα[5].

Έτσι, υπάρχει η αντίληψη πως είναι καλύτερα ένα σύστημα να ξεκινά με μονολιθική αρχιτεκτονική και στην συνέχεια να μεταφέρεται σε μικρουπηρεσίες[6]. Ο βασικός λόγος είναι πως η διαχειριστική πολυπλοκότητα επιβραδύνει την ανάπτυξη στα αρχικά στάδια. Τις περισσότερες φορές, η ιδέα ενός πρότζεκτ δεν είναι αρκετά ξεκάθαρη από την σύλληψη της, επομένως κρίνεται αναγκαία η δημιουργία μιας δοκιμαστικής έκδοσης που θα καταδείξει εάν αξίζει να επενδυθεί χρόνος και πόροι σε αυτήν. Αυτό αποτελεί ένα επιχείρημα ενάντια στην υιοθέτηση των μικρουπηρεσιών κατά το αρχικό στάδιο. Ακόμα όμως και με ξεκάθαρο όραμα από την αρχή, οι μικρουπηρεσίες έχουν θετικό αντίκτυπο μόνο εάν η επιχειρησιακή αποσύνθεση του προβλήματος έχει πραγματοποιηθεί σωστά. Αλλά αυτό είναι εξαιρετικά δύσκολο να επιτευχθεί προτού

υπάρξει πρακτική εμπειρία που αποκαλύπτει τις ιδιαιτερότητες του κάθε συστήματος. Με την υβριδική προσέγγιση, δίνεται χρόνος για εύρεση των ορίων χωρίς να θυσιάζεται ταχύτητα.

2.3 SOA (Service Oriented Architecture)

Όπως και οι μικρουπηρεσίες, η SOA δεν έχει κάποιο αυστηρό ορισμό. Κάποιοι από αυτούς θα οδηγούσαν στο συμπέρασμα πως είναι το ίδιο με τις μικρουπηρεσίες, ενώ κάποιοι άλλοι αρκετά διαφορετική[3]. Σίγουρα, θα μπορούσε να ειπωθεί πως οι μικρουπηρεσίες είναι μια κατηγορία της SOA, καθώς χαρακτηρίζεται ως μια προσέγγιση που δίνει βαρύτητα στην αξιοποίηση επαναχρησιμοποιούμενων μερών λογισμικού, ή πιο απλά υπηρεσιών, αφιερωμένες σε λειτουργίες που καλύπτουν το μεγαλύτερο δυνατό εύρος της επιχειρησιακής δραστηριότητας (enterprise-wide)[7].



Εικόνα 2: Σύγκριση Μονολιθικής Αρχιτεκτονικής με SOA

Συνήθως, η SOA σχετίζεται με πρωτόκολλα διαδικτυακών υπηρεσιών (web services) όπως SOAP και WSDL, ενώ οι μικρουπηρεσίες με REST και HTTP, που θεωρούνται πιο ελαφριές τεχνολογίες[8]. Ακόμα, η SOA κάνει έντονη χρήση του ESB (Enterprise Service Bus), δηλαδή ενός κεντρικού διαύλου πληροφορίας, που χρησιμοποιείται από τις

υπηρεσίες για επικοινωνία και συντονισμό. Σε αντίθεση οι μικρουπηρεσίες χρησιμοποιούν απλά και πιο άμεσα συστήματα μετάδοσης μηνυμάτων. Από μια πιο αφαιρετική σκοπιά, η αρχιτεκτονική των μικρουπηρεσιών προάγει το δόγμα «share-as-little-as-possible» (μοιράσου όσο το δυνατόν λιγότερα), ενώ η SOA «share-as-much-as-possible» (μοιράσου όσο το δυνατόν περισσότερα)[9]. Η κατανόηση αυτών των διαφορών είναι ιδιαίτερα χρήσιμη κατά τον σχεδιασμό ενός συστήματος.

2.4 Πλεονεκτήματα

Έχοντας παρουσιάσει την θέση των μικρουπηρεσιών στο ευρύτερο πλαίσιο της αρχιτεκτονικής συστημάτων λογισμικού μπορούμε να περάσουμε στην καταγραφή των πλεονεκτημάτων τους. Σύμφωνα με τον Sam Newman[10], κάποια από τα βασικότερα είναι:

Τεχνολογική ετερογένεια

Με την διάσπαση της εφαρμογής σε αυτόνομα μέρη δίνεται αυτόματα ένας μεγαλύτερος βαθμός ελευθερίας ως προς την επιλογή των υποκείμενων τεχνολογιών. Αυτό δίνει την δυνατότητα να αξιοποιηθεί το κατάλληλο εργαλείο για την κάθε εργασία, καθώς σπάνια μια τεχνολογία μπορεί να ικανοποιήσει ένα μεγάλο φάσμα απαιτήσεων το ίδιο καλά. Πέρα από αυτό, η ύπαρξη διαφορετικών τεχνολογιών αφαιρεί τον προαιρετικό χαρακτήρα των «καθαρών» και ανεξάρτητων διεπαφών, υποχρεώνοντας τον προγραμματιστή να ακολουθήσει μια καλή πρακτική. Ως αποτέλεσμα, η αντικατάσταση μιας τεχνολογίας δεν αποτελεί επίπονη διαδικασία ικανοποιώντας έτσι την απαίτηση για προσαρμοστικότητα σε ένα δυναμικό περιβάλλον.

Ανθεκτικότητα

Ένα ιδιαίτερα επιθυμητό χαρακτηριστικό για οποιοδήποτε σύστημα είναι η ανοχή σε σφάλματα. Για να επιτευχθεί αυτό, πρέπει το σύστημα να μπορεί να υποστηρίξει μία κατάσταση μερικής αποτυχίας (partial failure). Υποθέτοντας πως υπάρχουν μηχανισμοί εντοπισμού σφαλμάτων, οι μικρουπηρεσίες μπορούν εύκολα να περιορίσουν την διάδοση τους (τεχνική bulkhead), δίνοντας την ευκαιρία στο σύστημα να ανακάμψει

αλλά και αποφεύγοντας να επηρεαστούν αρνητικά τα υπόλοιπα μέρη της εφαρμογής. Η ανθεκτικότητα του συστήματος βελτιώνει την συνολική εμπειρία του χρήστη.

Κλιμάκωση

Όπως έχει ήδη αναφερθεί στην σύγκριση με την μονολιθική αρχιτεκτονική, οι μικρουπηρεσίες έχουν το πλεονέκτημα του μεγαλύτερου ελέγχου στην κλιμάκωση των πόρων του συστήματος. Τα λιγότερα απαιτητικά μέρη του συστήματος μπορούν να εκτελούνται σε πιο αδύναμο (άρα και πιο φθηνό) υλισμικό και τα υπόλοιπα σε ακριβώς όσο χρειάζονται. Σε συνδυασμό με την τεχνολογία των container και την κατα απαίτηση κλιμάκωση που προσφέρουν οι πλατφόρμες νέφους δίνεται η δυνατότητα βελτιστοποίησης του κόστους, κάτι που αποτελεί ισχυρό κίνητρο για τις επιχειρήσεις.

Ευκολία δημοσίευσης (deployment)

Σε ένα πολύπλοκο μονολιθικό σύστημα η κυκλοφορία μιας νέας έκδοσης της εφαρμογής, συνήθως, θεωρείται επίφοβη διαδικασία καθώς χιλιάδες γραμμές κώδικα προσπαθούν να «συνεννοηθούν» μεταξύ τους, δημιουργώντας χώρο για διάφορα απρόβλεπτα προβλήματα. Για την μείωση αυτού του ρίσκου η δημοσίευση μιας νέας έκδοσης στους χρήστες καταλήγει να περιλαμβάνει πολλές αλλαγές και διορθώσεις μαζί. Αυτό όμως αυξάνει τις πιθανότητες κάποια νέα συνθήκη που εισήγαγαν οι αλλαγές να περάσει απαρατήρητη από τις δοκιμαστικές φάσεις, με αποτέλεσμα να υπάρχουν δυσάρεστες εκπλήξεις, δημιουργώντας ένα αρνητικό βρόχο ανατροφοδότησης του προβλήματος. Αντίθετα, στις μικρουπηρεσίες οι αλλαγές σε μια υπηρεσία μπορούν να δημοσιευτούν ανεξάρτητα από τις υπόλοιπες, μειώνοντας τόσο τον χρόνο κυκλοφορίας νέας λειτουργικότητας όσο και το μέγεθος του αντίκτυπου που θα έχει ένα σφάλμα.

2.5 Προκλήσεις

Σπάνια η εφαρμογή μιας λύσης σε ένα πολύπλοκο τεχνικό πρόβλημα έχει μόνο θετική επίδραση. Συνήθως κερδίζει κάτι σε ένα τομέα, αλλά χάνει κάτι σε κάποιο άλλο. Η αρχιτεκτονική των μικρουπηρεσιών δεν αποτελεί εξαίρεση. Έτσι, υπάρχει μια σειρά από προκλήσεις που είναι καλό να γνωρίζουν όσοι την επιλέξουν.

Διαχείριση και Παρακολούθηση Συστήματος

Όσο ο αριθμός και η πολυπλοκότητα των μικρουπηρεσιών ανεβαίνει η ανάγκη για συνεχή παρακολούθηση του συστήματος γίνεται επιτακτική. Η ύπαρξη διαφόρων τεχνολογιών, όπως εικονικές μηχανές, containers, ουρές μηνυμάτων και ενδιάμεσο λογισμικό κάθε τύπου, απαιτεί την συλλογή όλων αυτών των διαγνωστικών δεδομένων και παρουσίαση τους σε εύπεπτη και βολική μορφή στον διαχειριστή του συστήματος. Επίσης τα μηχανήματα της εφαρμογής μπορεί να βρίσκονται σε πάνω από μία ζώνες διαθεσιμότητας που σημαίνει διαφορετικές ζώνες ώρας, κάνοντας την διαδικασία αποσφαλμάτωσης ακόμα πιο δύσκολη και επίπονη[10]. Η δημιουργία τέτοιων εργαλείων από το μηδέν είναι αρκετά απαιτητική και χρονοβόρα διαδικασία. Ευτυχώς η κοινότητα ανοιχτού κώδικα προσφέρει αρκετές λύσεις, όπως Graphite, Prometheus και InfluxDB.

Εξαρτήσεις κώδικα

Οπως αναφέρθηκε στην προηγούμενη ενότητα, ένα από τα πλεονεκτήματα των μικρουπηρεσιών είναι η ανεξάρτητη δημοσίευση νέων εκδόσεων. Χωρίς όμως μια συνειδητοποιημένη στρατηγική για τη κοινή χρήση κώδικα μεταξύ των μικρουπηρεσιών, μπορούν να δημιουργηθούν εξαρτήσεις που πλέον επιτάσσουν συντονισμό για την κυκλοφορία μιας νέας έκδοσης με άλλες μικρουπηρεσίες[3]. Φυσικά, οι εξαρτήσεις κώδικα δεν πρέπει πάντα να αποφεύγονται. Για παράδειγμα, βιβλιοθήκες που αφορούν την διασύνδεση με κάποια τεχνική υποδομή (infrastructure code) έχει νόημα να επαναχρησιμοποιούνται αλλά αυτές που περιέχουν επιχειρησιακή λογική μπορούν να λειτουργήσουν αρνητικά. Αυτός είναι και ένας από τους λόγους που η ύπαρξη διπλότυπου κώδικα στην αρχιτεκτονική των μικρουπηρεσιών είναι αποδεκτή πρακτική.

Ενσωμάτωση Γραφικής Διεπαφής

Η γραφική διεπαφή αποτελεί σημαντικό κομμάτι στα περισσότερα συστήματα, αλλά επειδή η αρχιτεκτονική δίνει έμφαση στο λειτουργικό κομμάτι (backend) πολλές φορές παραβλέπεται στην συνολική προσέγγιση. Μία εύλογη ανησυχία είναι πως θα καταλήξουμε με ένα σύστημα μικρουπηρεσιών που όμως έχει μονολιθική γραφική διεπαφή[8]. Μια τέτοια κατάσταση μπορεί να είναι βιώσιμη, αλλά σίγουρα δεν βοηθά τους στόχους των μικρουπηρεσιών. Μια μερική λύση είναι η χρήση μιας πύλης API που απομονώνει τις μικρουπηρεσίες από την γραφική διεπαφή, αλλά το πρόβλημα του συγχρονισμού και συντονισμού των μικρουπηρεσιών σε γραφικό επίπεδο παραμένει. Απαντήσεις σε αυτό το ζήτημα θέλουν να δώσουν τόσο τα αυτοτελή συστήματα (self-contained systems), μια παραλλαγή των μικρουπηρεσιών, που προτάσσουν πως μια μικρουπηρεσία πρέπει να έχει την κυριότητα της αντίστοιχης γραφικής διεπαφής όσο και η ανερχόμενη τεχνολογία micro-frontends. Η ανάλυση τους είναι εκτός των σκοπών της εργασίας.

2.6 Μοντελοποίηση μικρουπηρεσιών

Μετά την απόφαση για χρήση της αρχιτεκτονικής των μικρουπηρεσιών έρχεται η φάση της σχεδίασης. Μία δημοφιλής προσέγγιση είναι να ακολουθηθεί μια σχεδίαση οδηγούμενη από το τομέα στον οποίο δραστηριοποιείται το σύστημα (DDD - Domain Driven Design)[11]. Σύμφωνα με αυτή την θεωρία, οι μηχανικοί λογισμικού πρέπει να συμπεριλάβουν σε τεχνικό επίπεδο το λεξιλόγιο που χρησιμοποιούν οι ειδικοί του τομέα. Αυτή η απαίτηση δημιουργήθηκε από την παρατήρηση πως πολλές φορές τα λειτουργικά προβλήματα ενός συστήματος οφείλονται σε κακή επικοινωνία και συνεννόηση μεταξύ προγραμματιστών και ειδικών, παρά σε καθαρά τεχνικά ζητήματα. Έτσι προτάσσεται μια σειρά από έννοιες και πρακτικές που στοχεύουν στην γεφύρωση αυτών των διαφορετικών θέσεων. Εδώ θα αναφέρουμε μόνο αυτά που έχουν άμεση συνάφεια με τις μικρουπηρεσίες αλλά όλη η θεωρία είναι εξαιρετικά ενδιαφέρουσα.

Διάχυτη ορολογία (Ubiquitous Language)

Μια κεντρική ιδέα του DDD είναι πως πρέπει να καθιερωθεί μία ορολογία οντοτήτων και συσχετίσεων που περιγράφει τις δραστηριότητες ενός οργανισμού με καθολικό χαρακτήρα. Αυτή η ορολογία θα λειτουργεί ως γλώσσα επικοινωνίας μεταξύ συναδέλφων αλλά και ομάδων[11]. Στόχος είναι κατά τη καταγραφή των απαιτήσεων ή περιγραφή ενός προβλήματος να υπάρχει ο μεγαλύτερος δυνατός βαθμός κατανόησης της κατάστασης. Για να επιτευχθεί αυτό, η προγραμματιστική ομάδα πρέπει να δημιουργήσει μια αρχιτεκτονική που αντικατοπτρίζει και εκφράζει αυτήν την ορολογία σε τεχνικό επίπεδο, δηλαδή σε επίπεδο κώδικα.

Οριοθετημένο πλαίσιο (Bounded Context)

Τα περισσότερα πρότζεκτ που έχουν μεγάλο εύρος δραστηριοτήτων διαπραγματεύονται πολλαπλά επιχειρησιακά μοντέλα. Πολλές φορές μια ομάδα καταλήγει να χρησιμοποιεί ένα προϋπάρχον μοντέλο που έχει δημιουργηθεί για άλλο σκοπό για επιτάχυνση της ανάπτυξης ή για σκοπούς επαναχρησιμοποίησης κώδικα. Όταν όμως ο ίδιος κώδικας προσπαθεί να εξυπηρετήσει δύο ξεχωριστά μοντέλα (ίδια οντότητα αλλα με διαφορετικές απαιτήσεις για αυτή), γίνεται πηγή σφαλμάτων και αναξιόπιστης λειτουργικότητας. Για αυτό είναι αναγκαίο να οριστεί ξεκάθαρα το πλαίσιο υπό το οποίο λειτουργεί ένα μοντέλο. Αυτό μπορεί να περιλαμβάνει διάφορα πράγματα, όπως τη σύσταση της ομάδας, το πεδίο χρήσης του εντός του οργανισμού και, πιο πρακτικά, την απομόνωση της βάσης του κώδικα και του σχήματος βάσης δεδομένων. Εντός αυτού του πλαισίου η χρήση του μοντέλου πρέπει να γίνεται με συνέπεια και να μην επηρεάζεται από εξωγενείς παράγοντες[12].

Έτσι, η συγκεκριμένη προσέγγιση διασπά το τομέα δράσης του συστήματος σε υποτομείς που περιέχουν οριοθετημένα πλαίσια και εκφράζονται μέσω της διάχυτης ορολογίας. Αυτή η πρακτική προσφέρει ένα καλό τρόπο σχεδιασμού μικρουπηρεσιών. Κάθε οριοθετημένο πλαίσιο αποτελεί μία μικρουπηρεσία[13]. Παραβλέποντας την απότομη καμπύλη εκμάθησης όλης της θεωρίας και των εννοιών, η προτροπή για ευθυγράμμιση επιχειρησιακών και τεχνικών διαδικασιών έχει θετική συμβολή στον σχεδιασμό των μικρουπηρεσιών.

2.7 Τεχνολογίες Επικοινωνίας και Ενσωμάτωσης

Σε μία μονολιθική εφαρμογή όλη η ενδοεπικοινωνία του συστήματος γίνεται με κλήσεις μεθόδων ή αντίστοιχων μηχανισμών επικοινωνίας, πάντα εντός της τρέχουσας διεργασίας. Στις μικρουπηρεσίες όλη αυτή η λογική πρέπει να μεταφραστεί σε δικτυακή επικοινωνία μεταξύ απομακρυσμένων διεργασιών. Πέρα από την προσθήκη μεγαλύτερης καθυστέρησης, το σύστημα είναι πλέον κατανεμημένο κληρονομώντας, έτσι, μία διαβόητη κατηγορία ζητημάτων[14]. Για την αντιμετώπιση αυτών των προκλήσεων, οι μικρουπηρεσίες χρησιμοποιούν διάφορους τρόπους επικοινωνίας ανάλογα τις απαιτήσεις της κατάστασης. Στη συνέχεια, θα περιγραφούν τρεις από αυτούς, το REST, το AMQP και το ανερχόμενο gRPC.

REST (Representational State Transfer)

Οι διαδικτυακοί πόροι (resources) αποτελούν την βάση των περισσότερων διαδικτυακών συστημάτων. Το REST είναι ένα αρχιτεκτονικό στυλ για κατανεμημένα συστήματα πολυμέσων που στοχεύει στην αποτελεσματική διαχείριση τους[15]. Χρησιμοποιώντας το μοντέλο του διακομιστή-πελάτη θέτει κανόνες για την ανταλλαγή των αναπαραστάσεων τους, με έναν βασικό να είναι πως δεν επιτρέπεται η διατήρηση οποιαδήποτε κατάστασης (state) στην πλευρά του διακομιστή ανάμεσα σε διαφορετικά αιτήματα. Η κατάσταση συνεδρίας μπορεί να διατηρείται μόνο στον πελάτη.

Ανεξάρτητα από τον πρωτότυπο ορισμό που παρουσιάζεται στον σύνδεσμο 14, ο βαθμός συμμόρφωσης στο πρότυπο να ποικίλλει. Έτσι, υπάρχει μία ιεραρχία, ονόματι Richardson Maturity Model, που παρουσιάζει τα διάφορα επίπεδα ωρίμανσης[16]. Στο επίπεδο μηδέν η υπηρεσία προσφέρει μόνο ένα πόρο και μία μέθοδο HTTP. Το επίπεδο ένα παρέχει διάφορους πόρους, με μία μόνο μέθοδο, αλλά και παραμέτρους. Στο επίπεδο δύο η υπηρεσία κάνει χρήση αρκετών μεθόδων και κωδικών HTTP για πολλούς πόρους που, συνήθως, αντιπροσωπεύουν επιχειρησιακές οντότητες. Οι υπηρεσίες CRUD (Create, Read, Update, Delete) που θα χρησιμοποιήσουμε στην εφαρμογή βρίσκονται σε αυτή την κατηγορία. Το τελευταίο επίπεδο περιλαμβάνει την έννοια HATEOAS (Hypermedia-as-the-engine-of-application-state), όπου οι αναπαραστάσεις των πόρων

περιλαμβάνουν υπερσυνδέσμους σε άλλους πόρους, καθοδηγώντας έτσι τις μεταβολές στην κατάσταση της εφαρμογής. Σύμφωνα με τον κύριο εμπνευστή του REST, μόνο υλοποιήσεις τρίτου επιπέδου μπορούν να ονομάζονται REST[17].

AMQP (Advanced Message Queuing Protocol)

Η επικοινωνία μέσω REST είναι σύγχρονη ακολουθώντας το μοντέλο διακομιστή πελάτη. Το AMQP παρέχει ασύγχρονη επικοινωνία μέσω της μετάδοσης μηνυμάτων σε ουρές που ακούν τα ενδιαφερόμενα μέρη[18]. Κεντρικό σημείο του συστήματος είναι ο διακομιστής μηνυμάτων (message broker) που δέχεται μηνύματα από εφαρμογές -παραγωγούς και τα δρομολογεί σε εφαρμογές-καταναλωτές. Για την σωστή μετάδοση των μηνυμάτων ο διακομιστής κοιτάει το κλειδί δρομολόγησης που περιέχεται στην κεφαλίδα του μηνύματος για να αποφασίσει σε τι τύπο ανταλλαγής (exchange type) απευθύνεται. Για να δοθεί μεγαλύτερη ευκινησία στην συμπεριφορά των καταναλωτών παρέχονται διάφοροι τέτοιοι τύποι. Στον τυπο direct το μήνυμα στέλνεται στις ουρές που αφορούν αυστηρά αυτήν την διεύθυνση. Ο τύπος fanout λειτουργεί ως αναμεταδότης στέλνοντας το μήνυμα σε όλες τις ουρές. Ο τύπος topic εφαρμόζει την τεχνική της πολυεκπομπής του μηνύματος σε γκρουπ ουρών και ο header κάνει το ίδιο αλλά αντί για το κλειδί δρομολόγησης ο διακομιστής κοιτάει παραμέτρους στην κεφαλίδα του μηνύματος[19]. Κάποιες από τις πιο δημοφιλείς υλοποιήσεις του προτύπου AMQP είναι το RabbitMQ και το Apache ActiveMQ.

gRPC (gRemote Procedure Calls)

Το gRPC είναι ένα εργαλείο, ανοιχτού κώδικα, για υλοποίηση του μηχανισμού Απομακρυσμένης Κλήσης Διαδικασιών (RPC). Η τεχνική RPC βρίσκεται σε δυσμένεια στον κόσμο των μικρουπηρεσιών (ίσως και γενικότερα), λόγω κάποιων αρνητικών χαρακτηριστικών αλλά και επειδή αυτός ο τύπος επικοινωνίας καλύπτεται εν μέρει από το REST[10]. Το gRPC παρέχει κάποια ενδιαφέροντα χαρακτηριστικά που του επιτρέπουν να διεκδικήσει την επιστροφή στο προσκήνιο. Χρησιμοποιεί ως πρωτόκολλο μεταφοράς δεδομένων το Protocol Buffers[20], ένα πρωτόκολλο δυαδικής μορφής και ανεξάρτητο πλατφόρμας, προσφέροντας όμως και τις συμβατικές επιλογές σειριοποίησης (JSON/XML). Επίσης, αξιοποιεί την έκδοση HTTP/2, υποστηρίζοντας πλήρως αμφίδρομη μετάδοση δεδομένων (bidirectional streaming). Τα παραπάνω κάνουν το gRPC να είναι μια ελκυστική επιλογή για την ενδοεπικοινωνία μικρουπηρεσιών.

WebSockets

Η τεχνολογία των WebSockets δεν έχει κάποια ιδιαίτερη θέση στον κόσμο των μικρουπηρεσιών, αλλά αποτελεί μία καλή επιλογή για αμφίδρομη επικοινωνία ανάμεσα στον φυλλομετρητή του χρήστη και τον διακομιστή. Ένας πελάτης WebSockets επικοινωνεί με μηνύματα σε μορφή κειμένου ή δυαδική, μέσω μιας απλής σύνδεσης TCP με τον διακομιστή. Η χειραψία περιλαμβάνει την ανταλλαγή πληροφοριών καταγωγής (origin-based security model)[21]. Αξίζει να σημειωθεί πως το gRPC δεν υποστηρίζει άμεση σύνδεση με τον χρήστη (αν και υπάρχει ως πειραματική έκδοση το gRPC Web που όμως δεν προσφέρει τη πλήρη λειτουργικότητα).

3. Μεθοδολογία

Στο προηγούμενο κεφάλαιο παρουσιάστηκε η αρχιτεκτονική των μικρουπηρεσιών από την θεωρητική της σκοπιά. Για την καλύτερη κατανόηση τους έχει αναπτυχθεί μια εφαρμογή βαθμολόγησης βιβλίων. Η εφαρμογή θα προσφέρει απλές λειτουργίες στον χρήστη, αλλά αρκετά εκφραστικές ώστε να υπάρχει τεχνική πρόκληση για την ανάπτυξη τους. Με δεδομένο ότι η εργασία διαπραγματεύεται την αρχιτεκτονική των μικρουπηρεσιών θα δοθεί μεγαλύτερη έμφαση στο λειτουργικό μέρος (backend) του συστήματος παρά στο παρουσιαστικό (frontend).

Όπως έχει ήδη αναφερθεί, η φιλοσοφία των μικρουπηρεσιών συνδυάζεται καλύτερα με πιο ευέλικτες (agile) μορφές ανάπτυξης λογισμικού σε σχέση με τους παραδοσιακούς τρόπους (waterfall). Έτσι, για την ανάπτυξη της εφαρμογής ακολουθήθηκε μία ανάλογη μεθοδολογία όπου η ομάδα δημιουργεί αφαιρετικές περιγραφές των επιθυμητών χαρακτηριστικών (epics) και αποδομεί το πρόβλημα σε υπομέρη (stories/tasks), που μπορούν να υλοποιούνται ανεξάρτητα. Λόγω της μονοπρόσωπης σύστασης της ομάδας ανάπτυξης δεν χρησιμοποιήθηκαν όλες οι προτεινόμενες πρακτικές, μιας και δεν θα έβρισκαν εφαρμογή, αλλά τηρήθηκε η γενικότερη φιλοσοφία. Πιο συγκεκριμένα, αξιοποιήθηκε το δημοφιλές εργαλείο διαχείρισης πρότζεκτ JIRA με χρήση του προτύπου Kanban Board[22].

Η ανάπτυξη της εφαρμογής είχε δυναμικό χαρακτήρα δοκιμάζοντας διάφορες τεχνολογίες που εν τέλει δεν επιλέχθηκαν. Για παράδειγμα, η μικρουπηρεσία Ανακοινώσεις που είναι ένας διακομιστής WebSocket/gRPC σε Go, αρχικά ήταν σε Spring Boot με έναν Envoy Proxy. Αλλά επειδή το αντίκτυπο της αλλαγής δεν επηρέαζε άλλα μέρη του συστήματος η μεταφορά σε άλλη τεχνολογία δεν ήταν μια επίπονη διαδικασία, καταδεικνύοντας ένα από τα πλεονεκτήματα των μικρουπηρεσιών. Στις επόμενες ενότητες παρουσιάζεται το ολοκληρωμένο αποτέλεσμα με το πρώτο μέρος «Καταγραφή απαιτήσεων» (3.1) να παρουσιάζει τα επιθυμητά χαρακτηριστικά, το δεύτερο «Αρχιτεκτονική συστήματος» (3.2) τον τεχνικό σχεδιασμό και το τρίτο

«Παρουσίαση εργαλείων ανάπτυξης» (3.3) να κάνει μία σύντομη παρουσίαση των εργαλείων που θα χρησιμοποιηθούν.

3.1 Καταγραφή Απαιτήσεων

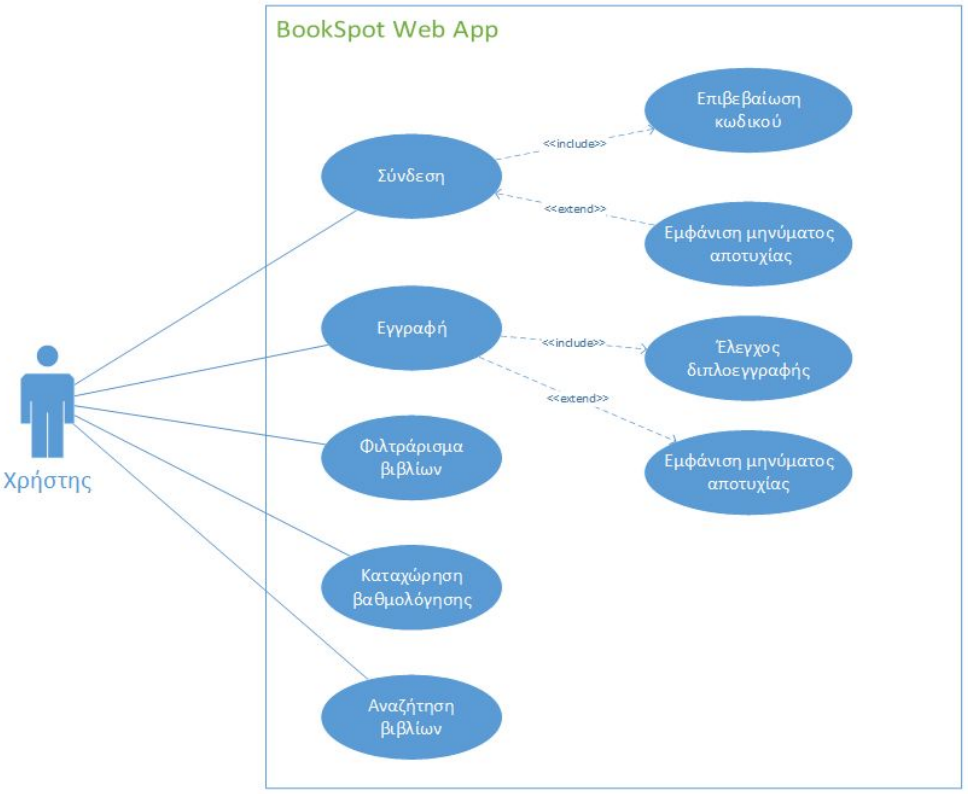
Οι απαιτήσεις που υπάρχουν από μια εφαρμογή καθορίζουν τα τεχνικά χαρακτηριστικά της. Τις περισσότερες φορές ορίζονται από τρίτα πρόσωπα, όπως συνεργάτες, επενδυτές ή πελάτες. Για τις ανάγκες της εργασίας θα οριστούν με τέτοιο τρόπο ώστε να εξυπηρετούν τους εκπαιδευτικούς σκοπούς της. Η κεντρική ιδέα της εφαρμογής είναι η βαθμολόγηση βιβλίων (εξού και το όνομα BookSpot), αλλά μπορούμε να προσθέσουμε συνοδευτικές λειτουργίες και χαρακτηριστικά που μπορεί να συναντήσει κανείς στις περισσότερες εφαρμογές, όπως χρήστες, ταξινόμηση και αναζήτηση περιεχομένου.

Η έννοια του χρήστη αυτόματα εισάγει τις λειτουργίες της εγγραφής και της σύνδεσης στο σύστημα. Από τη στιγμή που η ασφάλεια δεν αποτελεί κεντρικό μέρος του θέματος θα αρκεστούμε σε έλεγχο εγκυρότητας των εισαχθέντων στοιχείων στην πλευρά του χρήστη, αποφυγή διπλότυπων εγγραφών με βάση την ηλεκτρονική διεύθυνση και αποθήκευση του κωδικού σε κρυπτογραφημένη μορφή md5 (χρησιμοποιείται αλλά δεν προτείνεται[1]). Για την διατήρηση της σύνδεσης θα αποθηκεύσουμε το παραγόμενο αναγνωριστικό τοπικά στον φυλλομετρητή του χρήστη και θα θέσουμε μια χρονική περίοδο λήξης, δημιουργώντας έτσι απλοϊκά την έννοια της συνεδρίας (session).

Η προβολή και πλοήγηση βιβλίων δεν θα απαιτεί την σύνδεση του χρήστη στο σύστημα, αλλά η βαθμολόγηση θα επιτρέπεται μόνο σε συνδεδεμένους χρήστες. Τα φίλτρα ταξινόμησης θα περιέχουν βασικές επιλογές όπως την ημερομηνία κυκλοφορίας, την κατηγορία και, φυσικά, την καλύτερη/χειρότερη βαθμολογία. Από τεχνικής άποψης, τα φίλτρα είναι απλά ερωτήματα στην βάση πάνω σε διαφορετικές ιδιότητες οπότε μπορούν να υλοποιηθούν σχετικά εύκολα.

[1] https://www.sciencedirect.com/science/article/pii/S0167404818308332

Όλη η παραπάνω λειτουργικότητα εμφανίζεται στο ακόλουθο διάγραμμα περιπτώσεων χρήσης (use case diagram):



Εικόνα 3: Διάγραμμα περιπτώσεων χρήσης

Η τελευταία λειτουργία που θέλουμε να υποστηρίζεται είναι η προώθηση ειδοποιήσεων σε συνδεδεμένους χρήστες. Για τη εφαρμογή μας το περιεχόμενο των ειδοποιήσεων είναι ανακοινώσεις με ενέργειες άλλων χρηστών στην μορφή *ο x (χρήστης) βαθμολόγησε το y (βιβλίο) με z (αστέρια)*. Μιας και δεν υπάρχει πραγματική δραστηριότητα στην εφαρμογή θα παράγονται τεχνητά με τυχαία στοιχεία και ανά τυχαίο χρονικό διάστημα.

3.2 Αρχιτεκτονική εφαρμογής

Με την παρουσίαση της βασικής ιδέας της εφαρμογής και κατανοώντας τι θέλουμε να φτιάξουμε μπορούμε να περάσουμε στο τεχνικό σχεδιασμό της. Αρχικά, η προσέγγιση που θα ακολουθήσουμε είναι να διαιρέσουμε το τομέα της εφαρμογής σε υποτομείς (decompose by subdomain). Κάθε ένα από αυτά τα τμήματα θα αποτελέσει τη δική του μικρουπηρεσία(ες) καθορίζοντας έτσι τα λειτουργικά της όρια. Από την περιγραφή της εφαρμογής μπορούμε να διακρίνουμε τρεις υποτομείς ως βασικούς υποψήφιους: τα βιβλία, το χρήστη και τις ανακοινώσεις.

Κατάλογος

Τα βιβλία έχουν πρωταγωνιστικό ρόλο στην εφαρμογή αποτελώντας την βάση της κεντρικής ιδέας (βαθμολόγηση) αλλά και ουσιαστικά το "προϊόν" της (μέσω της παρουσίασης τους με την επαυξημένη αξία της βαθμολογίας). Αυτη η οπτική γωνία φανερώνει την θέση τους με πιο εμπορικούς όρους, δηλαδή ως τον κατάλογο προϊόντων της εφαρμογής. Οριοθετώντας την μικρουπηρεσία σε αυτά τα πλαίσια ανεβάζουμε το αφαιρετικό επίπεδο αρκετά ώστε να συμπεριλάβουμε τις συνοδευτικές ιδιότητες των βιβλίων (κατηγορίες, συγγραφείς κ.α.) και δίνουμε ένα επιπλέον βαθμό ελευθερίας για μελλοντικές επεκτάσεις.

Φυσικά η συγκεκριμένη απόφαση είναι απόρροια των τρεχουσών απαιτήσεων και επηρεάζεται από την δραστηριότητα της εφαρμογής. Έστω, παραδείγματος χάρη, ότι υπήρχε η απαίτηση να προσφέρεται και μια πλατφόρμα αυτοέκδοσης βιβλίων (self-publishing platform) εντός της εφαρμογής. Σε αυτή την περίπτωση ιδιότητες όπως συγγραφείς και εκδότες, που τώρα αποτελούν απλώς μετα-δεδομένα των βιβλίων, θα μπορούσαν να μεταφερθούν σε μία ξεχωριστή μικρουπηρεσία, δημιουργώντας το δικό τους οριοθετημένο πλαίσιο ή ακόμα και υποτομέα, έτσι ώστε αλλαγές εντός του τομέα να μην απαιτούν επαναδημοσίευση της μικρουπηρεσίας *Κατάλογος* και όλη η σχετική λειτουργικότητα να παραμείνει απομονωμένη.

Από τεχνικής πλευράς, παρατηρούμε ότι οι λειτουργίες που θέλουμε να προσφέρει η μικρουπηρεσία περιορίζονται σε ανάγνωση και τροποποίηση των εγγραφών (CRUD operations). Αυτό το μοτίβο πρόσβασης των δεδομένων εξυπηρετείται ιδιαίτερα αποτελεσματικά από προγραμματιστικές διεπαφές (APIs) τύπου REST. Άρα για την συγκεκριμένη υλοποίηση θα χτίσουμε ένα REST API σε Java με την βοήθεια του εργαλείου Spring Boot. Ως μέσο μόνιμης αποθήκευσης και προσπέλασης των εγγραφών θα χρησιμοποιηθεί το σύστημα διαχείρισης βάσεων δεδομένων PostgreSQL.

Χρήστης

Οι λειτουργίες που αφορούν τον χρήστη είναι αρκετά απομονωμένες διαδικασίες οπότε δεν είναι δύσκολο να σχεδιάσουμε μια μικρουπηρεσία που τις εμπερικλείει εξ ολοκλήρου. Αν είχαμε κάποιο σημείο επαφής, όπως αν έπρεπε να δείξουμε τις παρελθοντικές βαθμολογήσεις του χρήστη, με το να διατηρούμε στην μικρουπηρεσία *Χρήστης* την βαθμολογία και το αναγνωριστικό του βιβλίου, μπορούμε εύκολα να επικοινωνήσουμε με την υπηρεσία *Κατάλογος* για την εμφάνιση των απαραίτητων μετα-δεδομένων. Ακόμα και αν το μόνο που χρειαζόμαστε είναι ο τίτλος του βιβλίου πρέπει να αποφύγουμε την διατήρηση της πληροφορίας σε πολλαπλά σημεία, καθώς οποιαδήποτε αλλαγή σε αυτήν πλέον επεκτείνεται πέραν των ορίων της υπηρεσίας που είναι υπεύθυνη για αυτήν.

Για την υλοποίηση αυτής της μικρουπηρεσίας θα χρησιμοποιήσουμε την ίδια προσέγγιση με την προηγούμενη, δηλαδή ένα REST API με Spring Boot και PostgreSQL, μιας και το μοτίβο πρόσβασης των δεδομένων παραμένει ίδιο. Όμως για την παροχή συνεδρίας σε ένα συνδεδεμένο χρήστη είναι καλύτερο να χρησιμοποιήσουμε το Redis ως in-memory βάση δεδομένων που διατηρεί τις εγγραφές για ορισμένο χρόνο (TTL - time to live). Η ακύρωση τους θα σημαίνει την λήξη της συνεδρίας. Παρόλα αυτά ένα πραγματικό σύστημα πολύ πιθανόν να χρησιμοποιούσε και κάποια βάση δεδομένων για να προστατευθεί από πιθανές καταστροφές.

Τέλος, θα προσθέσουμε στην αρχιτεκτονική μία υποστηρικτική μικρουπηρεσία για καλύτερη διαχείριση και έλεγχο των APIs. Θα χρησιμοποιηθεί μία πύλη (gateway) που θα δρομολογεί τα αιτήματα των χρηστών στις κατάλληλες μικρουπηρεσίες. Όλη η κίνηση των αιτημάτων θα εκτελείται μέσω της πύλης δίνοντας μεγαλύτερη ευελιξία σε μελλοντικές αλλαγές.

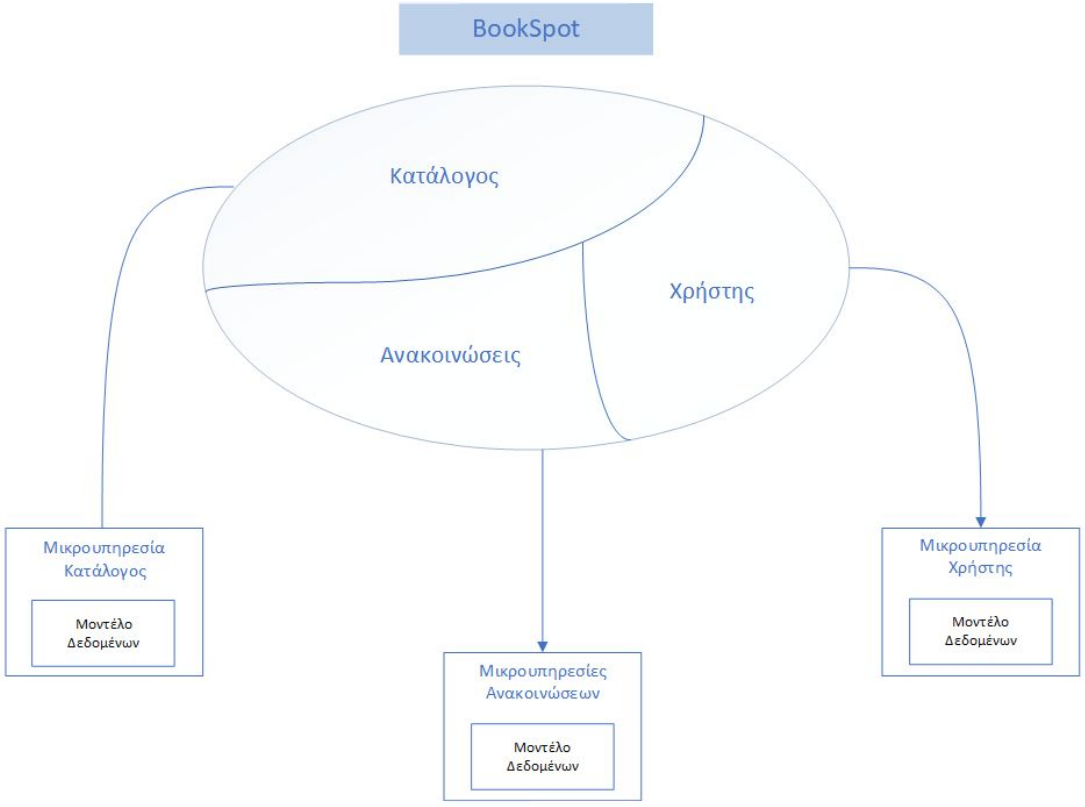
Ανακοινώσεις

Οι ανακοινώσεις, αν και αφορούν μόνο βαθμολογίες χρηστών, είναι προτιμότερο να έχουν εξαρχής γενικό και αυτόνομο χαρακτήρα. Για να επιτευχθεί αυτό πρέπει να διαχωριστεί ο μηχανισμός παραγωγής βαθμολογιών με τον μηχανισμό δημοσίευσης. Έτσι, ενώ είναι εφικτό (και πιο εύκολο) να παράγονται οι τυχαίες βαθμολογίες στην ίδια διεργασία όπου συνδέονται οι χρήστες για να "ακούσουν" για ανακοινώσεις, πρέπει να αποφευχθεί, καθώς δημιουργεί προβλήματα σε μελλοντικές επεκτάσεις. Έστω ότι προσθέτουμε την δυνατότητα προώθησης κάποιου κωδικού προσφοράς στον χρήστη και οι προσφορές λαμβάνονται ασύγχρονα μέσα από μία εξωτερική ουρά μηνυμάτων. Ένα κρίσιμο σφάλμα στον κώδικα παραγωγής των βαθμολογιών πλέον διακόπτει την λειτουργία προώθησης προσφορών ακόμα και αν δεν έχουν καμία λειτουργική συνάφεια. Εδώ έχουμε δύο επιλογές, είτε κρατάμε ένα υποτομέα με δύο οριοθετημένα πλαίσια, είτε δημιουργούμε δύο υποτομείς. Θα ακολουθήσουμε την πρώτη προσέγγιση. Έτσι έχουμε την μικρουπηρεσία *Ανακοινώσεις* και την μικρουπηρεσία *Παραγωγός Βαθμολογιών*.

Οι προηγούμενες μικρουπηρεσίες βασίζονται σε σύγχρονη (synchronous) επικοινωνία, όπου ο χρήστης είναι αυτός που αιτείται πόρους ή διαδικασίες του συστήματος, περιμένει και λαμβάνει τις αντίστοιχες απαντήσεις. Για την προώθηση ειδοποιήσεων, η χρήση ενός τέτοιου τρόπου δεν ενδείκνυται, καθώς η δημοσίευση των μηνυμάτων γίνεται σε ακαθόριστο χρόνο. Ο ασύγχρονος τρόπος επικοινωνίας εφαρμόζει καλύτερα σε αυτές τις απαιτήσεις. Η παραγωγή των βαθμολογιών θα προσφέρεται μέσω ενός διακομιστή gRPC και η δημοσίευση των ανακοινώσεων μέσω ενός διακομιστή WebSocket, που εκτελεί και χρέη πελάτη gRPC. Η υλοποίηση τους θα γίνει στην

γλώσσα προγραμματισμού Go καταδεικνύοντας έτσι και τον τεχνολογικό πλουραλισμό που επιτρέπουν οι μικρουπηρεσίες.

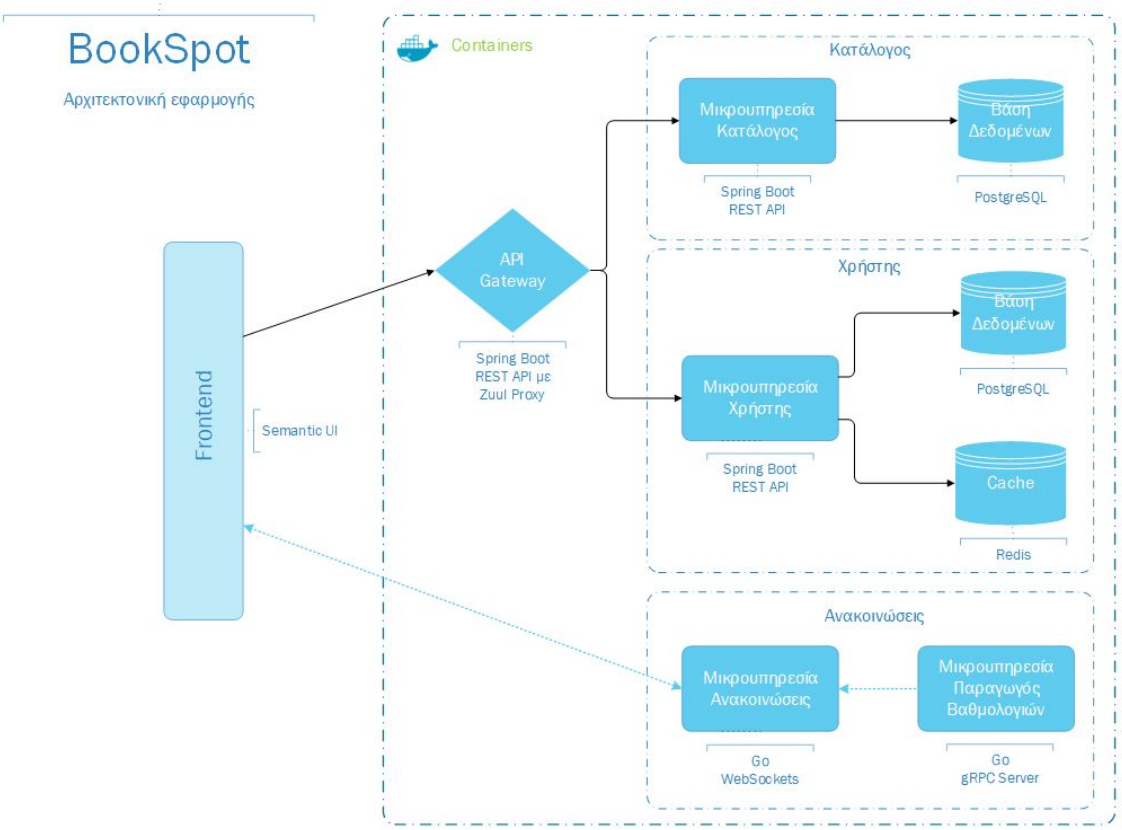
Η προσέγγιση που ακολουθήσαμε για την αποσύνθεση του επιχειρησιακού τομέα μας οδήγησε στο παρακάτω αποτέλεσμα:



Εικόνα 4: Αποσύνθεση του επιχειρησιακού τομέα της εφαρμογής

Όπως γίνεται φανερό και από το διαγράμμα κάθε υποτομέας έχει το δικό του μοντέλο. Αυτό σημαίνει ότι η διαχείριση των εγγραφών γίνεται σε βάση δεδομένων που είναι προσβάσιμη μόνο από τις περικλείουσες μικρουπηρεσίες και άρα η αντιπροσώπευση των οντοτήτων της εφαρμογής είναι εσωτερική για αυτές. Εύλογα δημιουργείται η απορία αν θα καταλήξουμε να έχουμε δύο μοντέλα για την ίδια οντότητα. Αυτό είναι ένα ρίσκο που δέχεται να αναλάβει η αρχιτεκτονική των μικρουπηρεσιών. Θεωρώντας πως ο διαχωρισμός των μικρουπηρεσιών είναι σωστός, θα έχουμε δύο μοντέλα για την ίδια οντότητα που καλύπτουν διαφορετικές ανάγκες για αυτήν.

Ακολουθεί η τεχνική απεικόνιση της αρχιτεκτονικής:



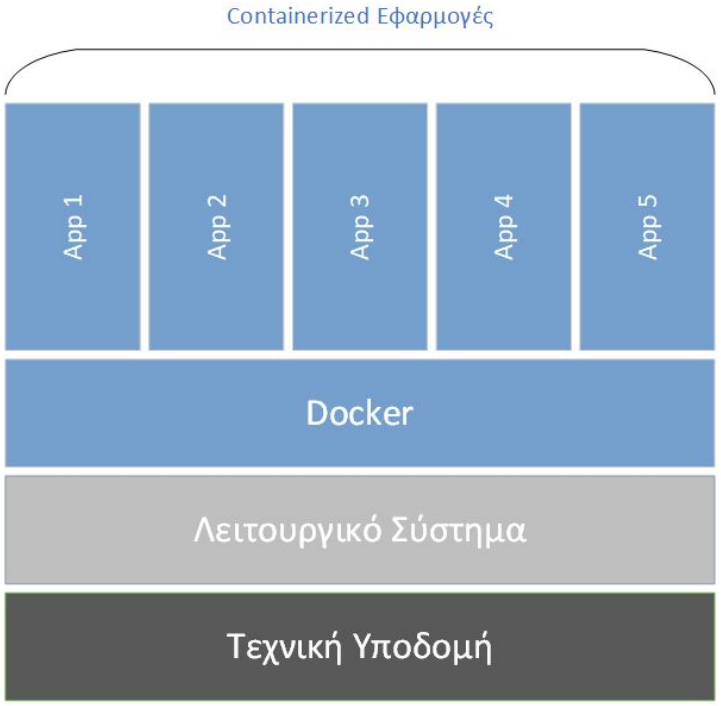
Εικόνα 5: Αρχιτεκτονική εφαρμογής

Τέλος, για να υπάρχει ένα ανεξάρτητο περιβάλλον εκτέλεσης, αποκομμένο από το περιβάλλον ανάπτυξης, όλες οι μικρουπηρεσίες θα τρέχουν σε containers στο Docker. Πέρα από την καλύτερη απομόνωση και αυτονομία τους, θα δοθεί η δυνατότητα τοπικής ενορχήστρωσης τους μέσω του εργαλείου docker compose, διευκολύνοντας αρκετά την ανάπτυξη της εφαρμογής.

3.3 Παρουσίαση εργαλείων ανάπτυξης

Docker

Το Docker είναι μία πλατφόρμα λογισμικού που χρησιμοποιεί τεχνικές εικονικοποίησης σε επίπεδο λειτουργικού συστήματος για να τρέχει εφαρμογές απομονωμένα[2]. Αυτό επιτυγχάνεται με την έννοια του container, μιας διεργασίας που εκτελεί σε απομονωμένη περιοχή χρήστη (user space) τον κώδικα της εφαρμογής. Για να το πετύχει αυτό σε χαμηλό επίπεδο αξιοποιεί χαρακτηριστικά του Linux kernel όπως namespaces και cgroups[3]. Το ακόλουθο διάγραμμα παρουσιάζει το Docker στα πλαίσια ενός τεχνικού συστήματος:



Εικόνα 6: Απεικόνιση συστήματος με Docker

[2] https://www.docker.com/resources/what-container
[3] https://docs.docker.com/engine/faq/#what-does-docker-technology-add-to-just-plain-lxc

Εγκατάσταση

Η εγκατάσταση του Docker σε λειτουργικά συστήματα Windows και Mac γίνεται μέσω του Docker Desktop. Απαιτείται μόνο το κατέβασμα και η εκτέλεση του προγράμματος εγκατάστασης ακολουθώντας τις εμφανιζόμενες οδηγίες[4], ενώ σε περιβάλλοντα Linux αρκεί η εγκατάσταση του Docker Engine[5].

Για να ελέγξουμε την τρέχουσα έκδοση μπορούμε να εκτελέσουμε την εντολή:

```
C:\Bookspot>docker --version
Docker version 19.03.1, build 74b1e89
```

ή για περισσότερες πληροφορίες:

```
C:\Bookspot>docker version
Client: Docker Engine - Community
 Version:           19.03.1
 API version:       1.40
 Go version:        go1.12.5
 Git commit:        74b1e89
 Built:             Thu Jul 25 21:17:08 2019
 OS/Arch:           windows/amd64
 Experimental:      false

Server: Docker Engine - Community
 Engine:
  Version:          19.03.1
  API version:      1.40 (minimum version 1.12)
  Go version:       go1.12.5
  Git commit:       74b1e89
  Built:            Thu Jul 25 21:17:52 2019
  OS/Arch:          linux/amd64
  Experimental:     false
```

[4] https://docs.docker.com/get-started/#download-and-install-docker-desktop
[5] https://docs.docker.com/engine/install/ubuntu/

Images/Containers

Για την δημιουργία ενός container απαιτείται η ύπαρξη ενός προτύπου (template) που περιέχει οδηγίες για το ποιο θα είναι το περιβάλλον εκτέλεσης του[6]. Αυτό το πρότυπο ονομάζεται image. Υπάρχει η δυνατότητα να βασίζεται σε κάποιο άλλο, γεγονός που διευκολύνει κατά πολύ την δουλειά του προγραμματιστή. Τις περισσότερες φορές αρκεί να προσθέσουμε το εκτελέσιμο, τις εξαρτήσεις του, καθώς και κάποιους παραμέτρους περιβάλλοντος για να έχουμε ένα ολοκληρωμένο image.

Η σύνταξη των οδηγιών γίνεται μέσω του Dockerfile, ενός αρχείου κειμένου που αναγνωρίζει ένα προκαθορισμένο σετ εντολών. Για παράδειγμα, ας υποθέσουμε ότι θέλουμε απλά να εμφανίσουμε τα περιεχόμενα του τρέχοντος ευρετηρίου. Αρκεί να δημιουργήσουμε ένα αρχείο με όνομα Dockerfile με τις ακόλουθες οδηγίες:

```
FROM alpine:3.7
ENTRYPOINT [ "ls", "-l" ]
```

Η οδηγία FROM δημιουργεί ένα επιπεδο (layer) από το επίσημο image του alpine, έκδοση 3.7 και η οδηγία ENTRYPOINT θέτει την εντολή που θα εκτελεστεί όταν εκκινήσουμε τον container. Το alpine είναι μία μίνιμαλ έκδοση του Linux που λόγω του μικρού του μεγέθους (~5MB) προτιμάτε, όπου είναι δυνατόν, από αυτή του ubuntu[7]. Πριν εκκινήσουμε τον container, πρέπει πρώτα να χτίσουμε (build) το image. Υποθέτοντας πως η γραμμή εντολών "κοιτάει" στο ίδιο ευρετήριο που βρίσκεται το αρχείο μας αρκεί να εκτελέσουμε την εντολή:

```
C:\Bookspot>docker build -t test-ls .
Sending build context to Docker daemon  2.048kB
Step 1/2 : FROM alpine:3.7
3.7: Pulling from library/alpine
5d20c808ce19: Pull complete
Digest:
sha256:8421d9a84432575381bfabd248f1eb56f3aa21d9d7cd2511583c68c9b7511d10
Status: Downloaded newer image for alpine:3.7
 ---> 6d1ef012b567
Step 2/2 : ENTRYPOINT [ "ls", "-l" ]
 ---> Running in ca1755c3949e
```

[6] https://docs.docker.com/get-started/overview/#docker-objects
[7] https://hub.docker.com/_/alpine

```
Removing intermediate container ca1755c3949e
 ---> 8c84874c4836
Successfully built 8c84874c4836
Successfully tagged test-ls:latest
```

Όπως φαίνεται και από τα δεδομένα εξόδου, όταν τρέχουμε την εντολή *docker build* όλα τα αρχεία του τρέχοντος ευρετηρίου αντιγράφονται (ως build context) στην διεργασία του Docker (docker daemon), άρα θεωρείται καλή πρακτική να είμαστε προσεκτικοί ως προς το που εδρεύει το Dockerfile. Η παράμετρος *-t test-ls* δίνει όνομα στην ετικέτα (tag) του image που αποτελεί βολικό σημείο αναφοράς για μελλοντικές εντολές. Μπορούμε να ανατρέξουμε τα υπάρχοντα images με την εντολή *docker images*:

```
C:\Bookspot>docker images
REPOSITORY   TAG      IMAGE ID           CREATED       SIZE
test-ls     latest  982baa038104     12 minutes ago   4.21MB
<none>      <none>  8c84874c4836     16 minutes ago   4.21MB
alpine       3.7    6d1ef012b567     14 months ago    4.21MB
```

Μετά το χτίσιμο του image το Docker θα επαναλάβει την ίδια διαδικασία μόνο εάν εντοπίσει κάποια αλλαγή στις οδηγίες του Dockerfile, καθώς χρησιμοποιεί δικιά του cache για να αποθηκεύσει τα παραχθέντα επίπεδα. Η εκκίνηση του container γίνεται με την εντολή *docker run {image}*. Αξίζει να σημειωθεί πως εάν το Docker δεν εντοπίσει τοπικά το όνομα του image θα το αναζητήσει στο Docker Hub, ένα απομακρυσμένο αποθετήριο με images, απο το οποίο θα το κατεβάσει, χτίσει και τρέξει εφόσον βρεθεί.

Άρα, για να τρέξουμε το δικό μας container εκτελούμε:

```
C:\Bookspot>docker container run test-ls
total 52
drwxr-xr-x    2 root     root          4096 Mar  6  2019 bin
drwxr-xr-x    5 root     root           340 May 17 20:02 dev
drwxr-xr-x    1 root     root          4096 May 17 20:02 etc
drwxr-xr-x    2 root     root          4096 Mar  6  2019 home
drwxr-xr-x   11 root     root          4096 Mar  6  2019 var
...
```

Το Docker client ξεκίνησε τον container, έτρεξε εσωτερικά την εντολή *ls -l* και τερμάτισε την λειτουργία του. Εάν είχαμε ξεκινήσει κάποια μακροχρόνια διεργασία θα μπορούσαμε να την τερματίσουμε εκτελώντας *docker container stop*. Όλες οι

μικρουπηρεσίες που θα αναπτύξουμε στο επόμενο κεφάλαιο θα έχουν το δικό τους Dockerfile και όσα αναφέραμε σε αυτή την ενότητα αρκούν για την ανεξάρτητη διαχείριση τους.

Docker compose

Για να αποφύγουμε την επίπονη διαδικασία της χειροκίνητης εκκίνησης/ τερματισμού κάθε μίας μικρουπηρεσίας ξεχωριστά θα χρησιμοποιηθεί ένα εργαλείο του Docker, το docker compose. Εάν η εγκατάσταση έχει γίνει μέσω του Docker Desktop, τότε είναι ήδη εγκατεστημένο, διαφορετικά μπορείτε να ακολουθήσετε τις επίσημες οδηγίες[8]. Το docker compose χρησιμοποιεί ένα αρχείο ρυθμίσεων, σε μορφή YAML, για να προσφέρει τις βασικές λειτουργίες (build, run, stop κ.α.) στο καθορισμένο γκρουπ υπηρεσιών. Για παράδειγμα, στα πλαίσια της ανάπτυξης της εργασίας ήταν συνηθισμένο να χρειάζονται μόνο οι βάσεις δεδομένων. Με το παρακάτω docker compose μπορούμε να πετύχουμε την ομαδική διαχείριση τους:

```yaml
version: '3.7'
services:
  db:
    image: bookspot/postgres
    container_name: postgres
    restart: always
    build:
      context: ./postgres
    environment:
      POSTGRES_PASSWORD: "******"
    healthcheck:
        test: ["CMD-SHELL", "pg_isready -U postgres"]
        interval: 10s
        timeout: 5s
        retries: 5
    ports:
      - "5432:5432"

  redis:
```

[8] https://docs.docker.com/compose/install/

```
    image: redis:5.0.7-alpine
    container_name: redis
    command: ["redis-server", "--appendonly", "yes"]
    hostname: redis
    ports:
      - "6379:6379"
```

Υπάρχουν αρκετές δυνατότητες που προσφέρει το docker compose, όπως η εισαγωγή μεταβλητών περιβάλλοντος, ακολουθιακή εξάρτηση των υπηρεσιών (οδηγία depends to), έλεγχος κατάστασης υπηρεσίας κ.α. Επίσης σε απλές περιπτώσεις μπορούμε να μη χρησιμοποιήσουμε Dockerfile θέτοντας κατευθείαν το επιθυμητό image, όπως φαίνεται στη δεύτερη υπηρεσία *redis*. Σε αντίθεση, η πρώτη θέτει ως build context τον φάκελο postgres, όπου περιμένει να βρει ένα αρχείο Dockerfile.

Το docker compose είναι ένα πολύ χρήσιμο εργαλείο, αλλά προορίζεται κυρίως για τοπικά περιβάλλοντα ανάπτυξης[9]. Για τις ανάγκες μιας πραγματικής (production-level) εφαρμογής πολλές πλατφόρμες νέφους προσφέρουν κάποια υπηρεσία ενορχήστρωσης containers, όπως το Amazon ECS του AWS ή το ACS του Azure. Μία καλή επιλογή ανοιχτού κώδικα είναι το Kubernetes[10], καθώς ήδη έχει ενσωματωθεί στις περισσότερες πλατφόρμες νέφους και, επίσης, μειώνει τους φόβους για πιθανή εξάρτηση από έναν συγκεκριμένο πάροχο (vendor lockdown).

[9] https://docs.docker.com/compose/#common-use-cases
[10] https://kubernetes.io/

PostgreSQL

Η PostgreSQL είναι ένα σχεσιακό σύστημα διαχείρισης βάσεων δεδομένων, ανοιχτού κώδικα και ελεύθερης χρήσης. Βασίστηκε στο πρότζεκτ POSTGRES του πανεπιστημίου της Καλιφόρνιας και με εθελοντικές συνεισφορές από όλο τον κόσμο και μετρά πλέον πάνω δύο δεκαετίες ενεργούς ανάπτυξης. Διατηρεί συμβατότητα με το SQL standard σε μεγάλο βαθμό (160 απο τις 179 υποχρεωτικές απαιτήσεις) αν και σε κάποιες περιπτώσεις χρησιμοποιεί ελαφρώς διαφορετική σύνταξη ή λειτουργία[11]. Θεωρείται ένα από τα πιο προηγμένα συστήματα διαχείρισης προσφέροντας πολλές λειτουργίες και επιλογές παραμετροποίησης.

Για την τρέχουσα εργασία δεν απαιτείται χειροκίνητη εγκατάσταση, καθώς θα χρησιμοποιηθεί το Docker. Η δημιουργία Dockerfile για την PostgreSQL μπορεί να προβεί πολύπλοκη διαδικασία και για αυτό το λόγο παρέχεται επίσημη υποστήριξη προσφέροντας Dockerfiles και script αρχικοποίησης για διάφορες εκδόσεις. Για τις δικές μας απαιτήσεις θα χρησιμοποιήσουμε την έκδοση 12. Θα απλοποιήσουμε το Dockerfile αφαιρώντας περιττές εξαρτήσεις και λειτουργικότητα για μεγαλύτερη αναγνωσιμότητα. Όλα τα απαραίτητα αρχεία και πληροφορίες μπορούν να βρεθούν στο αποθετήριο docker-library/postgres[12]. Το Dockerfile είναι το παρακάτω:

```
FROM postgres:12.0-alpine

LABEL maintainer="mai18043@uom.edu.gr"

ENV LANG en_US.utf8

# Add local dump to initdb
COPY /init/dump.sql /docker-entrypoint-initdb.d/

RUN mkdir -p /var/run/postgresql && chown -R postgres:postgres
/var/run/postgresql && chmod 2777 /var/run/postgresql

ENV PGDATA /var/lib/postgresql/data
# this 777 will be replaced by 700 at runtime (allows semi-arbitrary
"--user" values)
RUN mkdir -p "$PGDATA" && chown -R postgres:postgres "$PGDATA" && chmod 777
```

[11] https://www.postgresql.org/about/
[12] https://github.com/docker-library/postgres

```
"$PGDATA"
VOLUME /var/lib/postgresql/data

COPY docker-entrypoint.sh /usr/local/bin/
ENTRYPOINT ["docker-entrypoint.sh"]

EXPOSE 5432
CMD ["postgres"]
```

Η αντιγραφή του τοπικού SQL αρχείου στο ειδικό ευρετήριο *docker-entrypoint-initdb.d* προκαλεί την αυτόματη εκτέλεση του, μόνο για την πρώτη φορά που δημιουργείται η βάση δεδομένων. Περισσότερες πληροφορίες ως προς την λογική πίσω από αυτήν την οδηγία θα δοθούν στο κεφάλαιο Μικροϋπηρεσία Κατάλογος (4.1). Ακολουθεί η δημιουργία ευρετηρίων της PostgreSQL με τα απαραίτητα δικαιώματα και αντιγράφεται το script αρχικοποίησης που πραγματοποιεί ελέγχους στο περιβάλλον εγκατάστασης. Τέλος, δίνουμε πρόσβαση στον container από την θύρα 5432 και θέτουμε ως σημείο αρχικοποίησης πρώτα το script και μετά την εκτέλεση της ίδιας της Postgres.

Για να δοκιμάσουμε την εγκατάσταση κάνουμε build, όπως είδαμε στην προηγούμενη ενότητα:

```
C:\Bookspot\postgres> docker build -t postgres .
Step 1/12 : FROM postgres:12.0-alpine
12.0-alpine: Pulling from library/postgres
...
Status: Downloaded newer image for postgres:12.0-alpine
 ---> 5b681acb1cfc
Step 2/12 : LABEL maintainer="mai18043@uom.edu.gr"
 ---> Running in 914edba67edd
Removing intermediate container 914edba67edd
...
Step 11/12 : EXPOSE 5432
 ---> Running in a1ddf3340dc2
Removing intermediate container a1ddf3340dc2
 ---> 206eab6fa4a7
Step 12/12 : CMD ["postgres"]
 ---> Running in 4d5ef368a80e
Removing intermediate container 4d5ef368a80e
 ---> 417fc095e00e
Successfully built 417fc095e00e
Successfully tagged postgres:latest
```

Για να τρέξουμε χειροκίνητα τον container πρέπει να ορίσουμε την μεταβλητή περιβάλλοντος POSTGRES_PASSWORD και να επιτρέψουμε στο Docker να ανακατευθύνει όλα τα αιτήματα στη θύρα 5432 στην αντίστοιχη θύρα του container. Με τη χρήση του docker compose που δείξαμε προηγουμένως αυτές οι επιλογές τίθενται στο αρχείο ρυθμίσεων και εφαρμόζονται αυτόματα.

```
C:\BookSpot\postgres> docker run -it -p 5432:5432 -e POSTGRES_PASSWORD=admin
postgres
...
PostgreSQL init process complete; ready for start up.
2020-05-20 21:06:01.797 UTC [1] LOG:  starting PostgreSQL 12.0 on
x86_64-pc-linux-musl, compiled by gcc (Alpine 8.3.0) 8.3.0, 64-bit
2020-05-20 21:06:01.797 UTC [1] LOG:  listening on IPv4 address "0.0.0.0",
port 5432
2020-05-20 21:06:01.797 UTC [1] LOG:  listening on IPv6 address "::", port
5432
2020-05-20 21:06:01.806 UTC [1] LOG:  listening on Unix socket
"/var/run/postgresql/.s.PGSQL.5432"
2020-05-20 21:06:01.831 UTC [50] LOG:  database system was shut down at
2020-05-20 21:06:01 UTC
2020-05-20 21:06:01.841 UTC [1] LOG:  database system is ready to accept
connections
```

Αν και θεωρητικά δεν είναι απαραίτητο, ένα εργαλείο επισκόπησης βάσεων δεδομένων αυξάνει κατά πολύ την παραγωγικότητα. Για την PostgreSQL, η πιο δημοφιλής επιλογή είναι το PgAdmin[13] και χρησιμοποιήθηκε τόσο για την προβολή και τροποποίηση των δεδομένων όσο και για την εξαγωγή τους (dump.sql). Ανοίγοντας το PgAdmin και δημιουργώντας μία σύνδεση στο *localhost:5432* με τον κωδικό που δώσαμε μπορούμε να περιηγηθούμε στα περιεχόμενα της βάσης επιβεβαιώνοντας πως η διαδικασία ολοκληρώθηκε επιτυχώς.

[13] https://www.pgadmin.org/

Redis

Το Redis είναι μια NoSQL, in-memory, βάση δεδομένων προσφέροντας όμως και την δυνατότητα μόνιμης αποθήκευσης. Το βασικό μοντέλο των δεδομένων έχει την μορφή κλειδιού-τιμής (key-value) αλλά υποστηρίζονται διάφοροι τύποι όπως λίστες και σετς[14]. Πέρα των βασικών λειτουργιών της ανάγνωσης, τροποποίησης και διαγραφής εγγραφών υπάρχουν δυνατότητες όπως εγγραφή σε κανάλια για δημοσίευση/παραλαβή δεδομένων, μετάδοση σε πραγματικό χρόνο (streaming), συσταδοποίηση (clustering).[15] Το Redis χρησιμοποιείτε συχνά ως μνήμη cache για γρήγορη προσπέλαση δεδομένων.

Εγκατάσταση

Όπως και στην PostgreSQL δεν θα γίνει χειροκίνητη εγκατάσταση αλλά μέσω Docker. Επειδή δεν χρειάζεται κάποια ιδιαίτερη ρύθμιση ή αρχικοποίηση δεν θα δημιουργηθεί Dockerfile, αλλά θα χρησιμοποιηθεί το επίσημο image και μόνο. Για την εκκίνηση από το docker compose αρκεί η παρακάτω εγγραφή:

```
redis:
  image: redis:5.0.7-alpine
  container_name: redis
  command: ["redis-server", "--appendonly", "yes"]
  hostname: redis
  ports:
    - "6379:6379"
```

Η παράμετρος *--appendonly* ενεργοποιεί την μόνιμη αποθήκευση με την μέθοδο AOF[16], έτσι ώστε αν κλείσουμε και ανοίξουμε τον container να μην έχουν χαθεί τα δεδομένα. Για χρήση χωρίς docker compose μπορούμε να τρέξουμε κατευθείαν τον container με την επίσημη ετικέτα και θα το κατεβάσει το Docker αυτόματα:

```
C:\Bookspot> docker run -it redis:5.0.7-alpine
Unable to find image 'redis:5.0.7-alpine' locally
5.0.7-alpine: Pulling from library/redis
...
```

[14] https://github.com/antirez/redis
[15] https://redis.io/topics/introduction
[16] https://redis.io/topics/persistence

```
1:C 21 May 2020 20:50:12.939 # Redis is starting
1:C 21 May 2020 20:50:12.939 # Redis version=5.0.7, bits=64,
commit=00000000, modified=0, pid=1, just started
1:C 21 May 2020 20:50:12.939 # Warning: no config file specified, using the
default config. In order to specify a config file use redis-server
/path/to/redis.conf.
1:M 21 May 2020 20:50:12.940 * Ready to accept connections
```

Για την ευκολότερη πλοήγηση στο Redis χρησιμοποιήθηκε το Redis Desktop Manager[17] (μέχρι την έκδοση 0.9.3 είναι δωρεάν) που είναι διαπλατφορμικό και ανοιχτού κώδικα.

Spring Boot

Το Spring Boot είναι μία, ανοιχτού κώδικα, επέκταση της εργαλειοθήκης Spring που στοχεύει στην απλοποίηση της προγραμματιστικής εμπειρίας για την ταχύτερη ανάπτυξη εφαρμογών[18]. Έτσι, προσφέρει αρκετές λειτουργίες που εξυπηρετούν αυτόν τον στόχο, όπως αυτόματη φόρτωση ρυθμίσεων (auto-configuration) και ενσωματωμένους servlet containers (Tomcat, Jetty, Undertow). Μπορούμε εύκολα να συμπεράνουμε πως είναι μια προσπάθεια να φέρει το Spring πιο κοντά στις σύγχρονες απαιτήσεις που ζητούν ταχείς κύκλους ανάπτυξης και ευέλικτες εφαρμογές που τρέχουν σε εικονοποιημένο περιβάλλον. Αυτός είναι και ο λόγος που αποτελεί μια φυσική επιλογή για την ανάπτυξη μικρουπηρεσιών σε Java.

Για επιτάχυνση της διαδικασίας ανάπτυξης λογισμικού προσφέρεται μια πληθώρα από πρόσθετες βιβλιοθήκες (ονόματι Starters[19]) που καλύπτουν τόσο τις ανάγκες για συνηθισμένα σενάρια διασύνδεσης με εξωτερικές υπηρεσίες (βάσεις δεδομένων, cache κ.α.) όσο και για διαχείριση της ίδιας της εφαρμογής, όπως δυνατότητα ελέγχου κατάστασης, δυναμική παροχή εξωτερικών ρυθμίσεων και διάφορες μετρικές.

[17] https://github.com/uglide/RedisDesktopManager
[18] https://github.com/spring-projects/spring-boot
[19] https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-starter

Για την ανάπτυξη όλων των υπηρεσιών Spring Boot χρησιμοποιήθηκε το εργαλείο ανάπτυξης Intellij IDEA Ultimate Edition (εκπαιδευτική άδεια)[20]. Η έκδοση Ultimate παρέχει κάποια έξτρα λειτουργικότητα και επαγγελματική υποστήριξη για το Spring Boot αλλά μιας και δεν αξιοποιήθηκε κάποια από τα προσφερόμενα χαρακτηριστικά το αποτέλεσμα θα ήταν ίδιο και με τη δωρεάν έκδοση Community. Επίσης, μπορούν να χρησιμοποιηθούν και άλλα εργαλεία όπως Eclipse, Microsoft Visual Code, Theia μέσω του STS[21]. Ωστόσο, το Spring Boot δεν απαιτεί κάποιο συγκεκριμένο εργαλείο άρα θεωρητικά αρκεί ο αγαπημένος σας επεξεργαστής κειμένου/IDE και η εγκατεστημένη Java, αν και έτσι θα πρέπει να γίνει χειροκίνητα η διαχείριση των εξαρτήσεων, γεγονός που λειτουργεί σίγουρα αποτρεπτικά. Οι βασικές επιλογές για αυτόματη διαχείριση εξαρτήσεων είναι Gradle, Maven και Ant, με το τελευταίο να μην προτείνεται[22].

Για την ανάπτυξη της εφαρμογής χρησιμοποιήθηκαν οι ακόλουθες εκδόσεις:

- Spring Boot 2.2.0
- JDK 1.8.0_151 (μικρότερη επιτρεπτή 1.8, μέγιστη 13)
- Gradle 5.2.1 (μικρότερη επιτρεπτή 4.10+)

[20] https://www.jetbrains.com/idea/download/#section=windows
[21] https://spring.io/tools
[22] https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-build-systems

Jenkins

Το Jenkins είναι ένας διακομιστής για αυτοματοποιημένες εργασίες, ανοικτού κώδικα και ελεύθερης χρήσης[23]. Για την εγκατάσταση του πρέπει να υπάρχει το περιβάλλον εκτέλεσης Java (JRE). Εναλλακτικά μπορεί να εγκατασταθεί μέσω Docker[24]. Για τις ανάγκες της εργασίας χρησιμοποιήθηκε ένας απομακρυσμένος διακομιστής Ubuntu στην πλατφόρμα νέφους Amazon Lightsail (εκπαιδευτική προσφορά).

Στα πλαίσια της ανάπτυξης λογισμικού υπό της αρχές της Συνεχούς Ενσωμάτωσης (Continuous Integration) δημιουργήθηκε μία αλυσιδωτή διαδικασία εργασιών με το Jenkins να διαδραματίζει κεντρικό ρόλο. Αρχικά, για την καλύτερη διαχείριση του πρότζεκτ χρησιμοποιήθηκε το σύστημα ελέγχου εκδόσεων Git. Αν και προτείνεται κάθε μικρουπηρεσία να έχει ανεξάρτητο αποθετήριο Git, για πρακτικούς λόγους θα αρκεστούμε σε ένα (monorepo). Για μεγαλύτερη διαφάνεια και πρόληψη από τοπικές καταστροφές θα χρησιμοποιήσουμε το GitHub, έναν ιστότοπο που προσφέρει, δωρεάν, φιλοξενία αποθετηρίων Git.

Με ενσωμάτωση GitHub και Jenkins μπορούμε να ξεκινάμε διαδικασίες όταν υπάρχει κάποια αλλαγή στον πηγαίο κώδικα[25]. Για την περίπτωση μας, οι διαδικασίες αυτές είναι το χτίσιμο (build) του κώδικα και η εκτέλεση των τεστ. Αν και αυτές είναι ενέργειες που πρέπει να εκτελεί έτσι και αλλιώς ο προγραμματιστής τοπικά πριν ανεβάσει τις αλλαγές του, οι συνθήκες που επικρατούν στο Jenkins είναι πιο ανεξάρτητες από το τοπικό περιβάλλον ανάπτυξης προσφέροντας μεγαλύτερη ασφάλεια. Πέρα από αυτό, σε πραγματικές συνθήκες οι διαδικασίες θα ήταν πιο πολλές (π.χ. performance tests) με το τελικό στάδιο να είναι η δημοσίευση του εκτελέσιμου στους διακομιστές του συστήματος (application servers).

[23] https://www.jenkins.io/
[24] https://www.jenkins.io/doc/book/installing/
[25] https://resources.github.com/whitepapers/practical-guide-to-CI-with-Jenkins-and-GitHub/

Για αποφυγή αχρείαστης πολυπλοκότητας θα δημιουργήσουμε ένα build.gradle στη ρίζα του πρότζεκτ που θα λειτουργεί ως συντονιστής και θα εκτελεί όλα τα builds/tests των μικρουπηρεσιών μαζί.

build.gradle

```
allprojects {
   repositories {
       jcenter()
   }
}
```

και

settings.gradle

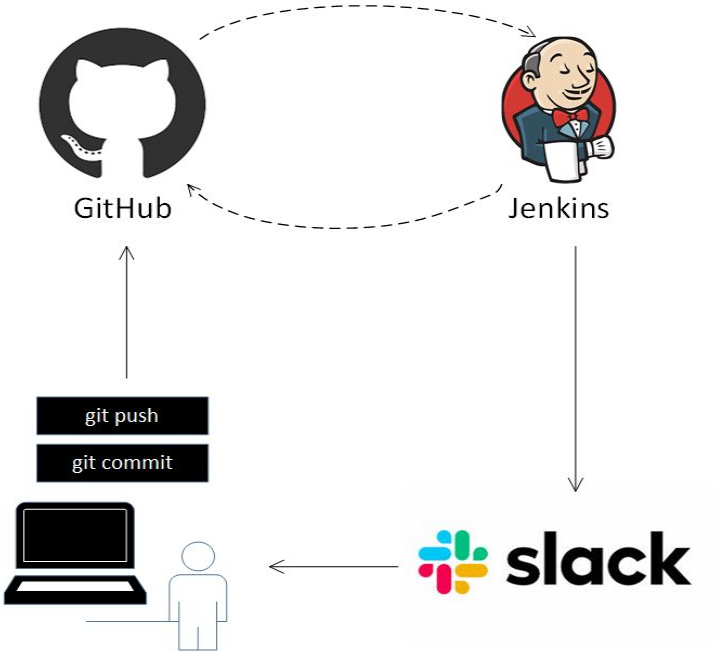
```
rootProject.name = 'BookSpot'
include 'gateway'
include 'user'
include 'catalog'
```

Για την σύνταξη των οδηγιών που πρέπει να ακουθήσει το Jenkins χρησιμοποιείται το Jenkinsfile.

```
pipeline {
    agent any
    stages {
         stage("Build") {
              agent {
                   docker {
                        image 'gradle:5.2.1-jdk8-alpine'
                   }
              }
              steps {
                   sh './gradlew build -x test'
                   sh 'echo "Build completed"'
              }
         }
         stage("Test") {
              agent {
                   docker {
                        image 'gradle:5.2.1-jdk8-alpine'
                   }
              }
              steps {
```

```
                    sh './gradlew test'
                    sh 'echo "Testing completed"'
                }
            }
        }
        post {
            success {
                slackSend channel: '#ci', color: 'good', failOnError: true,
message: "Build succeeded: '${env.BRANCH_NAME}'
(<${env.BUILD_URL}|${env.BUILD_NUMBER}>)", teamDomain: 'book-spot',
tokenCredentialId: '94413a97-7f5e-456b-aa13-d8abf57fca3d'
            }
            failure {
                slackSend channel: '#ci', color: 'danger', failOnError: true,
message: "Build failed: '${env.BRANCH_NAME}'
(<${env.BUILD_URL}|${env.BUILD_NUMBER}>)", teamDomain: 'book-spot',
tokenCredentialId: '94413a97-7f5e-456b-aa13-d8abf57fca3d'
            }
        }
}
```

Για να μην χρειάζεται να επισκεπτόμαστε τον διαχειριστικό ιστότοπο του Jenkins για να μάθουμε το αποτέλεσμα της διαδικασίας έγινε ενσωμάτωση με το Slack, μια εφαρμογή ανταλλαγής μηνυμάτων για επαγγελματικές ομάδες:



Εικόνα 7: Απεικόνιση διαδικασίας Συνεχούς Ενσωμάτωσης

4. Ανάλυση Κώδικα Μικρουπηρεσιών

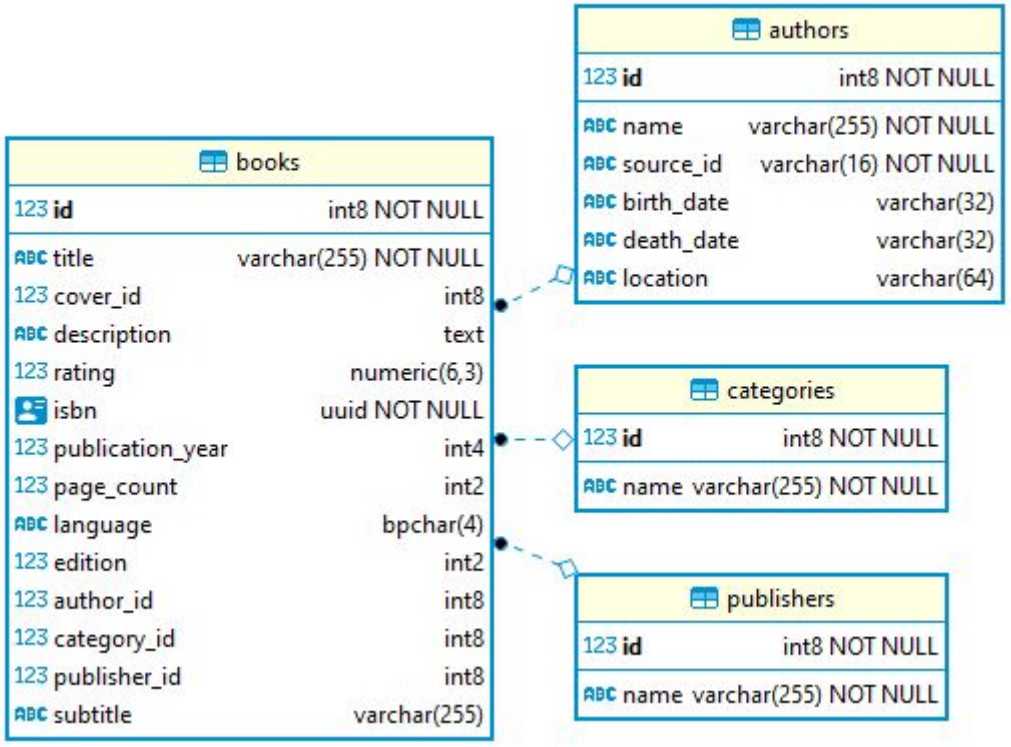
4.1 Μικρουπηρεσία Κατάλογος

Ο βασικός ρόλος της μικρουπηρεσίας Κατάλογος είναι να υποστηρίξει τις λειτουργίες ανάγνωσης και τροποποίησης βιβλίων. Αρχικά, πρέπει να ορίσουμε την επιθυμητή αναπαράσταση τους στο σχήμα της βάσης. Θα αρκεστούμε στα βασικά χαρακτηριστικά όπως τίτλος, κατηγορία, συγγραφέας, εκδότης, περιγραφή, βαθμολογία για την παρουσίαση τους στο χρήστη αλλά θα συμπεριλάβουμε και άλλα στο πίνακα όπως ISBN, αριθμός σελίδων κ.α.

Αν και είναι δελεαστικό να προσθέσουμε χαρακτηριστικά όπως κατηγορία, εκδότης, συγγραφέας με ονομαστικό τρόπο κατευθείαν στην εγγραφή του βιβλίου, είναι καλή πρακτική να κανονικοποιήσουμε την αναπαράσταση, δημιουργώντας ξεχωριστούς πίνακες για αυτά. Κάνει τον χειρισμό των πόρων πιο επίπονη διαδικασία για τον προγραμματιστή, αλλά μειώνει την περιττή πληροφορία και βοηθά σε μελλοντικές επεκτάσεις. Ακόμα, η ύπαρξη του ISBN στα βιβλία διευκολύνει ιδιαίτερα την διατήρηση της μοναδικότητας στον πίνακα, διαφορετικά θα έπρεπε να δημιουργηθεί ένα κλειδί ως συνδυασμός από διάφορα πεδία.

Όπως έχει ήδη αναφερθεί η κάθε μικρουπηρεσία πρέπει να έχει την δική της βάση δεδομένων. Έτσι, υποθέτοντας πως η PostgreSQL είναι διαθέσιμη (όπως δείξαμε σε προηγούμενη ενότητα) δημιουργούμε μία μέσω του PgAdmin με *δεξί κλικ → Create → Database* και δίνουμε το όνομα *catalog*. Οι προεπιλεγμένες ρυθμίσεις μας ικανοποιούν. Στην PostgreSQL μία βάση εμπεριέχει σχήματα (schema) και αυτά με την σειρά τους πίνακες. Θα χρησιμοποιήσουμε το προυπάρχον σχήμα (public) για να δημιουργήσουμε τους πίνακες μας, κάνοντας *δεξί κλικ πάνω του → Create → Table*. Για

την εξαγωγή της εικονικοποιημένης αναπαράστασης του σχήματος χρησιμοποιήθηκε το, δωρεάν διαθέσιμο, εργαλείο DBeaver[26], με το αποτέλεσμα να απεικονίζεται παρακάτω:



Εικόνα 8: Σχήμα βάσης δεδομένων για την μικρουπηρεσία Κατάλογος

Για να έχουμε πιο ρεαλιστικά δεδομένα επιλέχθηκε ως «πρώτη ύλη» η ανοιχτή ηλεκτρονική βιβλιοθήκη Open Library[27] η οποία προσφέρει όλες τις εγγραφές της σε ογκώδη αρχεία μορφής κειμένου (.txt dump). Μετά από την προγραμματιστική εξαγωγή των εγγραφών ακολούθησε η εκκαθάριση τους, αφαιρώντας λήμματα χωρίς περιγραφή, ημερομηνία κυκλοφορίας κ.α. Ακόμα και έτσι η ποιότητα των εναπομείναντων βιβλίων δεν είναι ιδιαίτερα υψηλή αλλά εξυπηρετεί τους σκοπούς της εργασίας. Αξίζει να σημειωθεί πως η ύπαρξη συνδέσμων για τα εξώφυλλα των βιβλίων είναι ιδιαίτερα βολική και βελτιώνει το αισθητικό αποτέλεσμα. Έχοντας την βάση δεδομένων έτοιμη μπορούμε να περάσουμε στην κατασκευή του API.

[26] https://dbeaver.io/
[27] https://openlibrary.org/

Το Spring Boot προσφέρει ένα πολύ χρήσιμο διαδικτυακό εργαλείο, το Spring Initializr[28], όπου επιλέγοντας τις επιθυμητές ρυθμίσεις και εξαρτήσεις, παράγεται ένα έτοιμο πρότζεκτ, επιταχύνοντας το αρχικό στάδιο της ανάπτυξης. Έτσι, για την μικρουπηρεσία Κατάλογος ορίστηκαν οι ακόλουθες ρυθμίσεις και εξαρτήσεις:

Πίνακας 1: Επιλογές πρότζεκτ Spring Initializr

Project	Gradle
Language	Java
Spring Boot	2.2.0
Group	com.github.bagiasn.bookspot
Artifact	catalog
Packaging	Jar
Java	8
Dependencies	Spring Data REST, Spring Data JPA

Πατώντας *Generate* κατεβάζουμε το αρχείο και το αποσυμπιέζουμε. Θα προσθέτουμε κάθε μικρουπηρεσία στο πρότζεκτ BookSpot ως ξεχωριστό module για την καλύτερη οργάνωση του κώδικα. Άρα από το μενού του IntelliJ IDEA επιλέγοντας, *File → New → Module From Existing Sources → Πλοήγηση στο αποσυμπιεσμένο πρότζεκτ και επιλογή του build.gradle → OK*, έχουμε εισάγει την βασική δομή.

Spring Data JPA

Με την βοήθεια της βιβλιοθήκης Spring Data JPA μπορούμε να προσθέσουμε ORM (Object Relational Mapping) δυνατότητες στην εφαρμογή. Με την χρήση σημάνσεων (annotation) στις κλάσεις των μοντέλων αναπαριστούμε το σχήμα της βάσης σε προγραμματιστικό επίπεδο. Έτσι ενώ διαχειριζόμαστε μόνο αντικείμενα αυτών των κλάσεων, εκτελούμε αυτόματα ερωτήματα στη βάση δεδομένων αποφεύγοντας την

[28] https://start.spring.io/

χειροκίνητη συγγραφή τους και μειώνοντας το περιθώριο σφάλματος. Ακολουθεί η οντότητα που αντιπροσωπεύει το βιβλίο:

Book.java

```
package com.github.bagiasn.bookspot.catalog.models;

import com.fasterxml.jackson.annotation.JsonProperty;
import org.hibernate.annotations.Type;

import javax.persistence.*;
import java.io.Serializable;
import java.util.UUID;

@Entity()
@Table(name = "books")
public class Book implements Serializable {

   private long id;
   private String title;
   @Column(name = "cover_id")
   @JsonProperty(value = "cover_id")
   private long coverId;
   private String description;
   private double rating;
   @Type(type = "pg-uuid")
   private UUID isbn;
   @Column(name = "publication_year")
   @JsonProperty(value = "publication_year")
   private int publicationYear;
   @JsonProperty(value = "page_count")
   private long pageCount;
   private String language;
   private long edition;

   private Author author;
   private Category category;
   private Publisher publisher;

   public Book() {}

   @Id
   @GeneratedValue(strategy = GenerationType.SEQUENCE, generator =
"books_seq_gen")
   @SequenceGenerator(name = "books_seq_gen", sequenceName = "books_id_seq",
allocationSize = 1)
   public long getId() {
       return id;
   }
```

```
    @ManyToOne
    @JoinColumn(name = "publisher_id")
    public Publisher getPublisher() {
        return publisher;
    }

    @ManyToOne
    @JoinColumn(name = "category_id")
    public Category getCategory() {
        return category;
    }

    @ManyToOne
    @JoinColumn(name = "author_id")
    public Author getAuthor() {
        return author;
    }
    // Υπόλοιποι getters και setters...
}
```

Η σήμανση Entity μαρκάρει την συγκεκριμένη κλάση ως JPA οντότητα και με την Table θέτουμε το όνομα του πίνακα στο οποίο αντιστοιχεί. Αν δεν το ορίζαμε θα έψαχνε για πίνακα με όνομα ίδιο με αυτό της κλάσης. Για την διατήρηση της καλής πρακτικής της ονομασίας μεταβλητών με camelCase στην Java, ορίζεται ρητά το αναμενόμενο όνομα σε όσες ιδιότητες της κλάσης δεν αντιστοιχεί με αυτό της βάσης. Το ίδιο και για την σειριοποίηση των ιδιοτήτων σε JSON.

Η σήμανση Id ορίζει το κύριο κλειδί της οντότητας και οι ακόλουθες την στρατηγική που πρέπει να ακολουθήσει το JPA όταν την δημιουργούμε. Αυτό χρειάστηκε κατά την εξαγωγή των δεδομένων από τα αρχεία κειμένου του Open Library και την εισαγωγή τους στη βάση. Χρησιμοποιούμε την έννοια της αλληλουχίας (sequence) της PostgreSQL για να δημιουργούνται κύρια κλειδιά μέσω αυτής, καθώς δίνει περισσότερες επιλογές από την εναλλακτική του bigserial. Αυτή είναι, παραδείγματος χάρη, η αλληλουχία για τα βιβλία:

```
CREATE SEQUENCE public.books_id_seq
    START WITH 1
    INCREMENT BY 1
    NO MINVALUE
    NO MAXVALUE
    CACHE 1;
```

Επίσης με την χρήση των σημάνσεων ManyToOne και JoinColumn ορίζουμε τα ξένα (foreign) κλειδιά της οντότητας Book. Αυτό είναι ιδιαίτερα βολικό στην περίπτωση μας καθώς τις περισσότερες φορές που θέλουμε να εμφανίσουμε ένα βιβλίο χρειαζόμαστε και τα μεταδεδομένα. Το JPA αναλαμβάνει να φέρει αυτές τις πληροφορίες αυτόματα, κάνοντας τα απαραίτητα JOINs στο παρασκήνιο.

Έτσι, για να ολοκληρώσουμε την ενσωμάτωση του JPA αρκεί να επεκτείνουμε την επιθυμητή διεπαφή και όλες οι βασικές λειτουργίες όπως ανάγνωση, αποθήκευση και αναζήτηση είναι διαθέσιμες. Επειδή μας ενδιαφέρει η λειτουργία της σελιδοποίησης (paging) θα επεκτείνουμε τη διεπαφή PagingAndSortingRepository.

BookRepository.java

```
package com.github.bagiasn.bookspot.catalog.api;

import com.github.bagiasn.bookspot.catalog.models.Book;

import org.springframework.data.repository.PagingAndSortingRepository;

public interface BookRepository extends PagingAndSortingRepository<Book,
Long>
{
}
```

Spring Data REST

Το Spring Data REST χτίζει πάνω σε διάφορες βιβλιοθήκες του Spring Data για να προσφέρει αυτόματα διεπαφές REST για το μοντέλο της εφαρμογής, χρησιμοποιώντας ως μορφή πολυμεσικής αναπαράστασης το HAL (Hypertext Application Language)[29]. Επίσημα υποστηρίζει JPA, MongoDB, Neo4J, GemFire και Cassandra. Άρα με την προσθήκη της σήμανσης RestResource σε συνδυασμό με τις

[29] https://tools.ietf.org/html/draft-kelly-json-hal-08

δυνατότητες αναζήτησεις που μας δίνει το JPA[30] μπορούμε να ανανεώσουμε το BookRepository ως εξής:

BookRepository.java

```
package com.github.bagiasn.bookspot.catalog.api;

import com.github.bagiasn.bookspot.catalog.models.Book;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.data.rest.core.annotation.RestResource;

import java.util.List;

public interface BookRepository extends PagingAndSortingRepository<Book,
Long> {

   @RestResource(path = "/byRatingAsc", rel = "/byRatingAsc")
   Page<Book> findAllByOrderByRatingAsc(Pageable pageable);

   @RestResource(path = "/byRatingDesc", rel = "/byRatingDesc")
   Page<Book> findAllByOrderByRatingDesc(Pageable pageable);

   @RestResource(path = "/byYearDesc", rel = "/byYearDesc")
   Page<Book> findAllByOrderByPublicationYearDesc(Pageable pageable);

   @RestResource(path = "/byYearAsc", rel = "/byYearAsc")
   Page<Book> findAllByOrderByPublicationYearAsc(Pageable pageable);

   @RestResource(path = "/byCategory", rel = "/byCategory")
   Page<Book> findAllByCategory_Name(@Param("name") String categoryName,
Pageable pageable);

   @RestResource(path = "/byTitle", rel = "/byTitle")
   List<Book> findBooksByTitleContainingIgnoreCase(@Param("title") String
title);
}
```

Δημιουργώντας τις αντίστοιχες κλάσεις για συγγραφείς, κατηγορίες και εκδότες, καθώς και ανάλογες διεπαφές που επεκτείνουν το απλούστερο CrudRepository έχουμε ένα ολοκληρωμένο REST API για την μικρουπηρεσία Κατάλογος. Πριν το δοκιμάσουμε πρέπει να θέσουμε κάποιες ρυθμίσεις ώστε το JPA να μπορεί να συνδεθεί στην

[30] https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation

PostgreSQL. Στο αρχείο application.properties που εδρεύει στον φάκελο resources, θέτουμε τα παρακάτω:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/catalog
spring.datasource.username=dev
spring.datasource.password=!QL5S7BDFRs8
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQL94Di
alect
spring.data.rest.base-path=/catalog
spring.application.name=catalog
server.port=8090
```

Για να αποφύγουμε να συμπεριλάβουμε ευαίσθητες πληροφορίες στο αποθετήριο git του πηγαίου κώδικα μπορούμε να τις θέσουμε στις παρακάτω μεταβλητές περιβάλλοντος και το Spring Boot θα τις εντοπίσει αυτόματα.

- SPRING_DATASOURCE_PASSWORD
- SPRING_DATASOURCE_USERNAME
- SPRING_DATASOURCE_URL

Το build.gradle είναι κατά πολύ όπως το δημιούργησε το Spring Initializr, με την διαφορά ότι έχουμε ενεργοποιήσει Jetty αντί για Tomcat και προσθέσαμε την εξάρτηση της PostgresSQL. Υπάρχει ήδη ο Gradle Wrapper στο κατεβασμένο πρότζεκτ άρα η εγκατάσταση του Gradle θα γίνει αυτόματα την πρώτη φορά που το τρέξουμε.

build.gradle

```
plugins {
  id 'org.springframework.boot' version '2.2.0.RELEASE'
}

apply plugin: 'java'
apply plugin: 'io.spring.dependency-management'

group = 'com.github.bagiasn.bookspot'
version = '0.1.0'

repositories {
```

```
  jcenter()
  maven { url 'https://repo.spring.io/milestone' }
}

configurations {
  developmentOnly
  runtimeClasspath {
     extendsFrom developmentOnly
  }
  compile.exclude module: 'spring-boot-starter-tomcat'
}

dependencies {
  implementation 'org.springframework.boot:spring-boot-starter-data-rest'
  implementation 'org.springframework.boot:spring-boot-starter-jetty'
  implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
  developmentOnly 'org.springframework.boot:spring-boot-devtools'
  runtimeOnly 'org.postgresql:postgresql'
  testImplementation('org.springframework.boot:spring-boot-starter-test') {
     exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
  }
}

test {
  useJUnitPlatform()
}
```

Μπορούμε να τρέξουμε την εφαρμογή από την γραμμή εντολών με *./gradlew bootRun* (ή *gradlew bootRun* σε Windows) ή μέσα από το IntelliJ IDEA. Εφόσον δεν παρουσιαστεί κάποιο σφάλμα ελέγχουμε πως το API ανταποκρίνεται με ένα εργαλείο όπως το Postman ή απλά απο την μπάρα του περιηγητή μας:

```
http://localhost:8090/catalog/books/search/byTitle?title=philosophy
```


```
{
  "_embedded" : {
    "books" : [ {
      "title" : "Breakfast with Socrates: the philosophy of everyday life",
      "description" : "**A very ordinary day in the company of some...",
      "rating" : 1.0,
      "isbn" : "1afbb9d9-bdcf-4393-b4bb-7d391a4f926d",
      "language" : null,
      "edition" : 0,
      "cover_id" : 6349615,
      "publication_year" : 2010,
```

```
      "page_count" : 192,
      "_embedded" : {
        "author" : {
          "name" : "Robert Rowland Smith",
          "location" : null,
          "source_id" : "OL6724406A",
          "birth_date" : null,
          "death_date" : null
        },
        "category" : {
          "name" : "Cookbooks"
        },
        "publisher" : {
          "name" : "Other"
        }
      },
      "_links" : {
        "self" : {
          "href" : "http://localhost:8090/catalog/books/1333638"
        },
        "book" : {
          "href" : "http://localhost:8090/catalog/books/1333638"
        },
        "author" : {
          "href" : "http://localhost:8090/catalog/books/1333638/author"
        },
        "category" : {
          "href" : "http://localhost:8090/catalog/books/1333638/category"
        },
        "publisher" : {
          "href" : "http://localhost:8090/catalog/books/1333638/publisher"
        }
      }
    } ]
  },
  "_links" : {
    "self" : {
      "href" :
"http://localhost:8090/catalog/books/search/byTitle?title=philosophy"
    }
  }
}
```

Docker compose

Το τελευταίο βήμα για την ολοκλήρωση της μικρουπηρεσίας είναι να υποστηρίζει εκτέλεση σε περιβάλλον Docker. Από την στιγμή που οι εφαρμογές Spring Boot μπορούν να τρέξουν με μια απλή εντολή *java* αρκεί να έχουμε το εκτελέσιμο και το Java runtime (JRE) στο τελικό παραγόμενο image. Από την στιγμή όμως που πρέπει να χτίσουμε τον πηγαίο κώδικα χρειαζόμαστε τόσο το Gradle όσο και το JDK. Για να διατηρήσουμε το μέγεθος του image σε φυσιολογικά επίπεδα θα χρησιμοποιήσουμε την τεχνική του multistage build[31]. Με αυτήν την τεχνική δίνεται η δυνατότητα να πραγματοποιήσουμε το χτίσιμο της εφαρμογής με τις απαραίτητες εξαρτήσεις στο πρώτο στάδιο και στο δεύτερο να αντιγράψουμε απλά το εκτελέσιμο από την πρώτη, κρατώντας στο τελικό image μόνο τα απαραίτητα. Επίσης, παρατηρώντας πως οι απαιτήσεις για το Dockerfile είναι ίδιες για όλες τις μικρουπηρεσίες σε Spring Boot θα το συντάξουμε με τέτοιο τρόπο ώστε να είναι επαναχρησιμοποιήσιμο.

Dockerfile

```
# Using global ARG in order to access main class on the second build stage.
# More info: https://github.com/moby/moby/issues/37345
ARG MAIN_CLASS

FROM gradle:5.2.1-jdk8-alpine AS build

LABEL maintainer="mai18043@uom.edu.gr"

# Gradle needs permission to build
COPY --chown=gradle:gradle . /workspace/src

WORKDIR /workspace/src

RUN gradle clean build --no-daemon

# Inflate application jar to take advantage of layered dependencies
RUN mkdir -p build/dependency && \
   cd build/dependency && \
   jar -xf ../libs/*.jar

# Use JRE version on runtime
FROM openjdk:8-jre-alpine
```

[31] https://docs.docker.com/develop/develop-images/multistage-build/

```
ARG MAIN_CLASS
ENV ENV_MAIN_CLASS=${MAIN_CLASS}

ARG SRC_PATH=/workspace/src/build/dependency
COPY --from=build ${SRC_PATH}/BOOT-INF/lib /app/lib
COPY --from=build ${SRC_PATH}/META-INF /app/META-INF
COPY --from=build ${SRC_PATH}/BOOT-INF/classes /app

ENTRYPOINT java -cp "app:app/lib/*"
com.github.bagiasn.bookspot.${ENV_MAIN_CLASS}
```

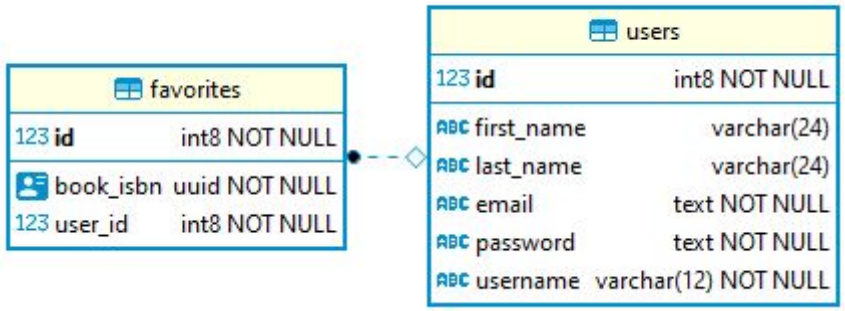
Έτσι μπορούμε να προσθέσουμε την μικρουπηρεσία Κατάλογος στο docker compose:

docker-compose.dev.all.yml

```
...
catalog-api:
  image: bookspot/catalog-api
  container_name: catalog
  build:
    context: ./catalog
    args:
      - MAIN_CLASS=catalog.CatalogApplication
  environment:
    - SPRING_DATASOURCE_URL=jdbc:postgresql://postgres:5432/catalog
    - SPRING_DATASOURCE_USERNAME=dev
    - SPRING_DATASOURCE_PASSWORD=!QL5S7BDFRs8
  ports:
    - "8090:8090"
  depends_on:
    - db
```

4.2 Μικρουπηρεσία Χρήστης

Η μικρουπηρεσία Χρήστης προσφέρει τις λειτουργίες της εγγραφής και σύνδεσης στο σύστημα. Από τεχνικής άποψης έχει παρόμοια σύνθεση με την μικρουπηρεσία Κατάλογος, είναι ένα Spring Boot REST API με διαχείριση δεδομένων σε PostgreSQL. Για πρακτικούς λόγους θα χρησιμοποιήσουμε τον υπάρχον postgres container αλλά θα φτιάξουμε διαφορετική βάση δεδομένων. Εφόσον υπάρχει λογικός διαχωρισμός είναι σχετικά εύκολο να προχωρήσουμε σε φυσικό όταν το απαιτούν οι συνθήκες. Το σχήμα της βάσης για τον χρήστη θα περιέχει τις βασικές πληροφορίες στο πινακα users και στο favorites θα αποθηκεύουμε τα αγαπημένα:



Εικόνα 9: Σχήμα βάσης δεδομένων μικρουπηρεσίας Χρήστης

Χρησιμοποιώντας το Spring Initializr εισάγουμε τις ίδιες επιλογές, με την διαφορά ότι προσθέτουμε την βιβλιοθήκη Spring Data Redis πέρα από τις προηγούμενες και φυσικά το όνομα είναι πλέον user. Εισάγοντας το αποσυμπιεσμένο αρχείο ως module στο πρότζεκτ μπορούμε να ξεκινήσουμε την ανάπτυξη.

Spring Data Redis

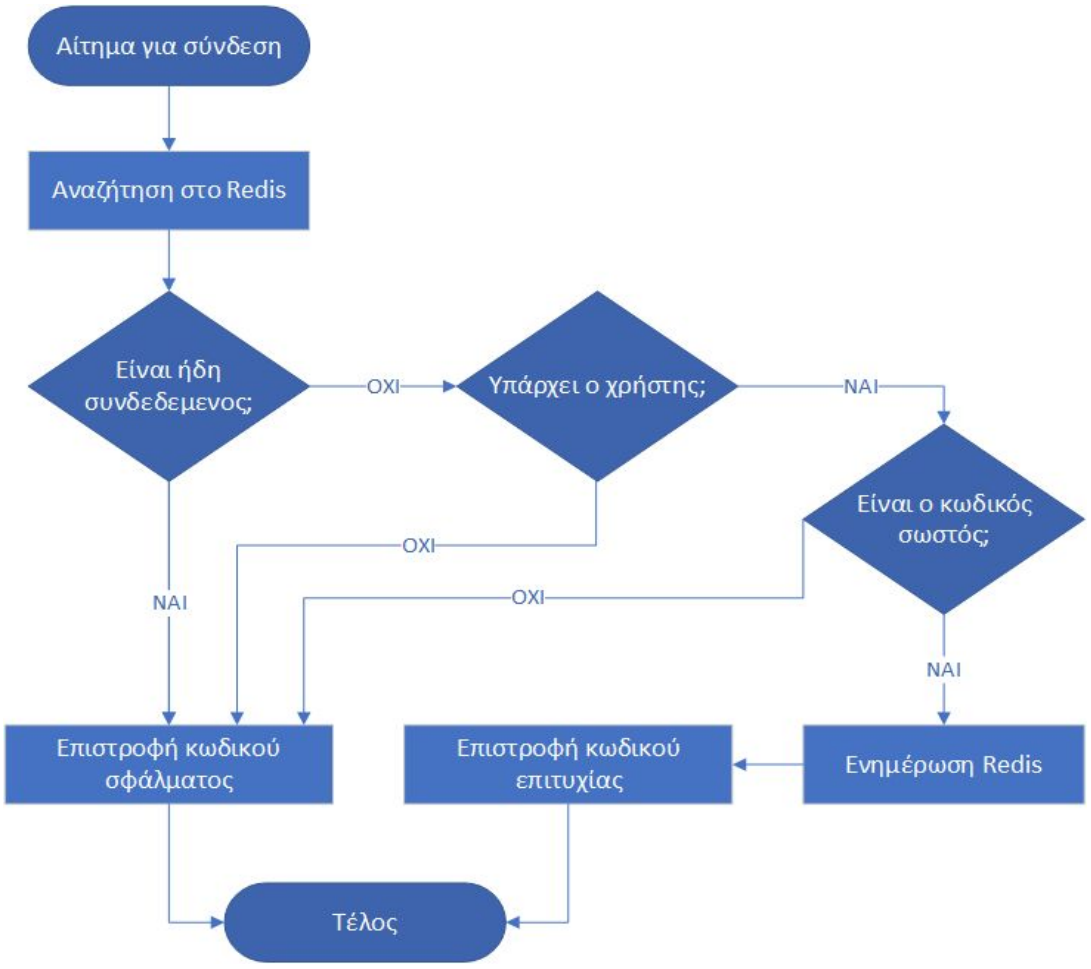
Το Spring Data Redis παρέχει υποστήριξη για σύνδεση και διαχείριση εγγραφών στο Redis[32]. Θεωρώντας πως το redis container έχει εκκινηθεί (όπως δείξαμε στην ενότητα Redis), για σύνδεση χρειάζεται απλά να θέσουμε την μεταβλητή περιβάλλοντος

[32] https://spring.io/projects/spring-data-redis

SPRING_REDIS_HOST σε *localhost*. Προγραμματιστικά, με την χρήση της σήμανσης EnableCaching στη κύρια κλάση και αξιοποιώντας τις μεθόδους της κλάσης RedisTemplate μπορούμε να αλληλεπιδράσουμε με το Redis.

Σύνδεση

Σε αντίθεση με την μικρουπηρεσία Κατάλογος πρέπει να υλοποιήσουμε χειροκίνητα τις λειτουργίες αυτής της μικρουπηρεσίας. Η σύνδεση του χρήστη θέλουμε να περιλαμβάνει την έννοια της συνεδρίας, άρα με την επιτυχή επαλήθευση των πληροφοριών θα αποθηκεύσουμε ένα αναγνωριστικό στο Redis ώστε να ξέρουμε εάν ο συγκεκριμένος χρήστης είναι ήδη συνδεδεμένος στο μέλλον. Η εγγραφή θα λήγει μετά από δύο ώρες. Οι απαιτήσεις από τη διαδικασία σύνδεσης φαίνονται παρακάτω:



Εικόνα 10: Διάγραμμα ροής για σύνδεση χρήστη

Το παραπάνω διάγραμμα ροής μεταφράζεται προγραμματιστικά στην ακόλουθη μέθοδο:

```
@RequestMapping(value = "/login", method = RequestMethod.POST)
public ResponseEntity<?> login(@RequestBody Credentials credentials) {

    String providedEmail = credentials.getEmail();
    logger.info("Login request received: {}", providedEmail);

    String userToken = redisTemplate.opsForValue().get(providedEmail);
    if (userToken != null && !userToken.isEmpty()) {
        logger.info("User already logged-in");

        return ResponseEntity.ok().body("User is already logged-in");
    } else {
        // First, find the requested user.
        User user = userRepository.findByEmail(providedEmail);
        if (user == null) {
            logger.warn("Could not find user with email {}", providedEmail);

            return ResponseEntity.notFound().build();
        } else {
            // Check if the password is correct.
            String providedPassword =
HashGenerator.GetPasswordHash(credentials.getPassword());
            if (providedPassword.equals(user.getPassword())) {
                logger.info("Password is correct.");
                // Generate a token
                String token = UUID.randomUUID().toString();
                // Update redis.
                redisTemplate.opsForValue().set(providedEmail, token,
Duration.ofSeconds(UserTokenTtl));
                Credentials creds = new Credentials();
                creds.setEmail(providedEmail);
                creds.setToken(token);

                return ResponseEntity.ok().body(creds);
            } else {
                logger.info("Wrong password was provided.");

                return
ResponseEntity.status(HttpStatus.UNAUTHORIZED).build();
            }
        }
    }
}
```

Εγγραφή

Για την εγγραφή του χρήστη η διαδικασία είναι πιο απλή αφού πρέπει να ελέγχουμε μόνο εάν υπάρχει ήδη η ηλεκτρονική διεύθυνση. Αξιοποιώντας τους ελέγχους που εκτελεί ήδη η PostgreSQL για την μοναδικότητα των εγγραφών, μπορούμε να επιστρέφουμε ένα συγκεκριμένο κωδικό (Conflict - 409) στην περίπτωση που εντοπιστεί τέτοιο σφάλμα. Ετσι έχουμε:

```
@RequestMapping(value = "/signup", method = RequestMethod.POST)
public ResponseEntity<?> signUp(@RequestBody User user) {
   logger.info("Sign-up request received");

   if (user != null) {
       try {
           // Store an MD5 hash instead of plain text.

user.setPassword(HashGenerator.GetPasswordHash(user.getPassword()));
           userRepository.save(user);
           return ResponseEntity.status(HttpStatus.CREATED).build();
       } catch (DataIntegrityViolationException dex) {
           logger.warn("Postgres exception: {}", dex.getMessage());
           return ResponseEntity.status(HttpStatus.CONFLICT).build();
       } catch (Exception ex) {
           logger.error("Sign up failed", ex);
           return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
       }
   } else {
       return ResponseEntity.badRequest().build();
   }
}
```

Docker compose

Θα χρησιμοποιήσουμε το ίδιο Dockerfile με την μικρουπηρεσία Κατάλογος άρα αρκεί να κάνουμε την παρακάτω προσθήκη στο docker compose:

```
...
user-api:
  image: bookspot/user-api
  container_name: user
  build:
    context: ./user
    args:
      - MAIN_CLASS=user.UserApplication
  environment:
    - SPRING_DATASOURCE_URL=jdbc:postgresql://postgres:5432/users
    - SPRING_DATASOURCE_USERNAME=dev
    - SPRING_DATASOURCE_PASSWORD=!QL5S7BDFRs8
    - SPRING_REDIS_HOST=redis
  ports:
    - "8100:8100"
  depends_on:
    - db
```

4.3 Μικρουπηρεσία Gateway API

Λόγω της αυξημένης πολυπλοκότητας των μικρουπηρεσιών και της ανάγκης για εσωτερική επικοινωνία μεταξύ τους, η δρομολόγηση αιτημάτων παίζει πολύ σημαντικό ρόλο στην διαχείριση του συστήματος. Για την παρουσίαση αυτής της τεχνολογίας θα χρησιμοποιήσουμε το εργαλείο ανοιχτού κώδικα Zuul του Netflix[33] για να δρομολογούμε όλα τα αιτήματα μέσω αυτού στις υπόλοιπες μικρουπηρεσίες. Πέρα από αυτό θα εξυπηρετούμε και αιτήματα προσκόμισης ιστοσελίδων ώστε να αποφύγουμε ζητήματα διαφορετικής καταγωγής αιτουμένων πόρων (CORS)[34].

Έτσι, οι δύο βιβλιοθήκες που πρέπει να συμπεριλάβουμε στο Spring Initializr είναι το Spring Web και το Zuul στη κατηγορία Spring Cloud Routing. Για την δρομολόγηση το Zuul χρειάζεται να γνωρίζει τα αιτήματα που αντιστοιχούν σε κάθε υπηρεσία, αλλά και την τοποθεσία της κάθε μίας. Αυτη η διαδικασία πραγματοποιείται στα πλαίσια της αναζήτησης υπηρεσιών (service discovery) και μπορεί να παρέχεται με την συνεργασία τρίτων εφαρμογών, όπως το Eureka[35], ή μέσω ρητής καταχώρησης ρυθμίσεων. Έχουμε λίγες υπηρεσίες άρα θα μπουν στο αρχείο application.properties:

```
spring.application.name=gateway

zuul.routes.catalog.path=/api/catalog/**
zuul.routes.catalog.service-id=catalog

zuul.routes.user.path=/api/user/**
zuul.routes.user.service-id=user

ribbon.eureka.enabled=false

server.port=8080
```

Το μόνο που μένει είναι να εισάγουμε την σήμανση EnableZuulProxy στη κύρια κλάση.

[33] https://github.com/Netflix/zuul
[34] https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
[35] https://github.com/Netflix/eureka

Docker compose

Καθώς το Dockerfile παραμένει ίδιο αρκεί να συμπεριλάβουμε το Gateway API στο docker compose, ορίζοντας ως εξαρτήσεις τις μικρουπηρεσίες Κατάλογος και Χρήστης για ευνόητους λόγους:

```
gateway-api:
  image: bookspot/gateway-api
  container_name: gateway
  build:
    context: ./gateway
    args:
      - MAIN_CLASS=gateway.GatewayApplication
  environment:
    - ZUUL_ROUTES_CATALOG_URL=http://catalog-api:8090/catalog
    - ZUUL_ROUTES_USER_URL=http://user-api:8100/user
  ports:
    - "8080:8080"
  depends_on:
    - catalog-api
    - user-api
```

4.4 Μικρουπηρεσία Παραγωγός Βαθμολογιών

Για την κατασκευή του συστήματος προώθησης ανακοινώσεων θα απομακρυνθούμε από τις προηγούμενες τεχνολογίες, μιας και δεν εξυπηρετούν ικανοποιητικά το επιθυμητό μοτίβο επικοινωνίας. Έχουμε χωρίσει το σύστημα των ανακοινώσεων σε δύο Go μικρουπηρεσίες που η μία έχει αναλάβει την παραγωγή βαθμολογιών και η άλλη την δημοσίευση τους. Θα συμπεριλάβουμε αυτές τις μικρουπηρεσίες στο ίδιο πρότζεκτ με τις προηγούμενες αλλά σε δικούς τους ανεξάρτητους φακέλους. Η απλότητα της Go κάνει περιττή την χρήση κάποιου προηγμένου εργαλείου (IDE). Για την συγγραφή του κώδικα χρησιμοποιήθηκε ο επεξεργαστής κειμένου Visual Studio Code[36].

Για την παραγωγή τυχαίων βαθμολογιών θα δημιουργήσουμε ένα διακομιστή gRPC που ανά τυχαίο χρονικό διάστημα στέλνει μηνύματα σε συνδεδεμένους χρήστες. Πρέπει να συντάξουμε τον ορισμό της υπηρεσίας δηλαδή ποιες μεθόδους θα εμπεριέχει και τον τύπο των μηνυμάτων που πραγματεύονται. Η περίπτωση μας είναι αρκετά απλή καθώς χρειαζόμαστε μόνο μία μέθοδο με την οποία θα "ακούει" ο χρήστης (server-side streaming). Το μήνυμα θα είναι σε μορφή κειμένου. Θεωρήθηκε προτιμότερο ο ορισμός της υπηρεσίας να βρίσκεται σε δικό του πακέτο (Go package) με όνομα api για μεγαλύτερη ανεξαρτησία μεταξύ των μικρουπηρεσιών. Όσοι το υλοποιούν θα το συμπεριλαμβάνουν ως εξάρτηση από το GitHub. Έτσι έχουμε το παρακάτω .proto αρχείο:

api.proto

```
syntax = "proto3";
package api;

// The announcement service definition.
service Announcement {
   // Start listening for announcements.
   rpc Listen(Request) returns (stream AnnouncementMessage) {}
}

message AnnouncementMessage {
```

[36] https://code.visualstudio.com/

```
    string message = 1;
}

message Request {
    int32 clientId = 1;
}
```

Το επόμενο βήμα είναι να μεταγλωττίσουμε αυτό το αρχείο σε πηγαίο κώδικα με την βοήθεια της επέκτασης protoc[37], παράγοντας το αντίστοιχο αρχείο api.pb.go. Αυτό περιέχει κώδικα για την σειριοποίηση των μηνυμάτων αλλά και την υλοποίηση τόσο του διακομιστή όσο και του πελάτη. Άρα μένει να προσθέσουμε την δικιά μας λογική στην πλευρά του διακομιστή. Η εσωτερική δομή των μηνυμάτων θέλουμε να είναι της μορφής *x (χρήστης) βαθμολόγησε το y (βιβλίο) με z (αστέρια)*. Έτσι για την παραγωγή ονομάτων χρηστών, βαθμολογίας αλλά και διάρκειας αναμονής πριν την επαναποστολή θα χρησιμοποιήσουμε την βιβλιοθήκη faker[38] και για τους τίτλους βιβλίων θα διαβάζουμε τυχαία από ένα αρχείο JSON που περιέχει πραγματικούς τίτλους της βάσης δεδομένων.

server.go

```
package main

import (
  "encoding/json"
  "flag"
  "fmt"
  "io/ioutil"
  "math/rand"
  "net"
  "os"
  "time"

  pb "github.com/bagiasn/book-spot/announcement/api"
  "github.com/bxcodec/faker/v3"

  log "github.com/sirupsen/logrus"
  "google.golang.org/grpc"
)

var (
  port     = flag.Int("port", 12000, "The server's port")
  jsonFile = flag.String("json_file", "./book_names.json", "The local json
file to read book names from")
```

[37] https://grpc.io/docs/languages/go/quickstart/
[38] https://github.com/bxcodec/faker

```
)

var bookData bookNames

type bookNames struct {
  Version int
  Names   []string
}

type fakeAnnouncement struct {
  Username string `faker:"first_name"`
  Rating   int    `faker:"boundary_start=1, boundary_end=6"`
  Delay    int    `faker:"boundary_start=10, boundary_end=20"`
}

type announcementServer struct {
}

func init() {
  // Output to stdout instead of the default stderr
  log.SetOutput(os.Stdout)

  // Read book names from local json file.
  data, err := ioutil.ReadFile(*jsonFile)
  if err != nil {
     log.Error("File reading error", err)
  }

  err = json.Unmarshal(data, &bookData)
  if err != nil {
     log.Error("Json deserialization error", err)
  }
}

// Start a gRPC server that produces fake announcements.
func main() {
  log.Info("Starting announcement server")

  lis, err := net.Listen("tcp", fmt.Sprintf(":%d", *port))
  if err != nil {
     log.Fatalf("Failed to bind port %d: %v", *port, err)
  }

  grpcServer := grpc.NewServer()

  pb.RegisterAnnouncementServer(grpcServer, newServer())
  grpcServer.Serve(lis)
}

func newServer() *announcementServer {
  s := &announcementServer{}
```

```
  return s
}

func (s *announcementServer) Listen(in *pb.Request, stream
pb.Announcement_ListenServer) error {
  log.WithFields(log.Fields{
     "clientId": in.ClientId,
  }).Info("Received client request")

  data := fakeAnnouncement{}

  for {
     // Populate announcement with fake data.
     _ = faker.FakeData(&data)
     // Get a random int as book index.
     index := rand.Intn(len(bookData.Names))
     message := fmt.Sprintf("%v rated \"%v\" with %d", data.Username,
bookData.Names[index], data.Rating)
     // Send to client.
     if err := stream.Send(&pb.AnnouncementMessage{Message: message}); err
!= nil {
        return err
     }

     time.Sleep(time.Duration(data.Delay) * time.Second)
  }
}
```

Μπορούμε να τρέξουμε την εφαρμογή με την εντολή *go run server.go.*

Docker compose

Όπως και στις προηγούμενες μικρουπηρεσίες πρέπει πρώτα να χτίσουμε την εφαρμογή και μετά να την εκτελέσουμε. Θα χρησιμοποιήσουμε και εδώ την τεχνική του multistage build για να μειώσουμε το μέγεθος του image, κρατώντας μόνο το εκτελέσιμο και το αρχείο JSON με τους τίτλους. Επίσης χρειάζεται να εγκαταστήσουμε πριν το build το Git καθώς είναι απαραίτητο για την διαχείριση των εξαρτήσεων. Αξίζει να σημειωθεί πως το τελικό image είναι μόνο ~10MB καθώς εδώ μπορούμε να χρησιμοποιήσουμε το βασικό image scratch[39].

Dockerfile

[39] https://docs.docker.com/develop/develop-images/baseimages/

```
FROM golang:1.14.1-alpine AS builder

LABEL maintainer="mai18043@uom.edu.gr"

ENV GO111MODULE=on \
   CGO_ENABLED=0

# Use a different path to avoid mod conflicts.
WORKDIR /app
# Git is needed for dependency installation.
RUN apk update && apk add --no-cache git

COPY go.mod .
COPY go.sum .
# Fetch dependencies.
RUN go mod download
# Verify dependencies.
RUN go mod verify

COPY . .
# Build the executable.
# Target linux and omit debug info
RUN GOOS=linux GOARCH=amd64 go build -ldflags="-w -s"

FROM scratch

# Copy the executable from the previous stage.
COPY --from=builder /app/server /
COPY --from=builder /app/book_names.json /

EXPOSE 12000
ENTRYPOINT ["/server"]
```

Έτσι μπορούμε να προσθέσουμε και την μικρουπηρεσία Παραγωγός Βαθμολογιών:

```
announcement-server:
  image: bookspot/announcement-server
  container_name: announcement-server
  build:
    context: ./announcement/server
  ports:
    - "12000:12000"
```

4.5 Μικρουπηρεσία Ανακοινώσεις

Η τελευταία μικρουπηρεσία είναι υπεύθυνη για την μετάδοση των ανακοινώσεων στους συνδεδεμένους χρήστες. Η επικοινωνία με τον περιηγητή του χρήστη θα γίνει μέσω της τεχνολογίας των WebSockets. Θα χρειαστούμε ένα διακομιστή WebSockets και ένα πρόγραμμα-πελάτη gRPC που συνδέεται στην μικρουπηρεσία Παραγωγός Βαθμολογιών και λαμβάνει τα μηνύματα. Για το στήσιμο του διακομιστή θα αξιοποιήσουμε την βιβλιοθήκη gorilla[40]. Για το gRPC την επίσημη βιβλιοθήκη για Go, ίδια με αυτή που φτιάξαμε τον διακομιστή στην προηγούμενη ενότητα.

Επειδή η επιθυμητή λειτουργικότητα απαιτεί να τρέχουν ταυτόχρονα δύο διαδικασίες, αρχικά θα ξεκινήσουμε μία ρουτίνα Go που παραλαμβάνει τα μηνύματα από τον διακομιστή gRPC και τα στέλνει σε ένα κανάλι (channel):

```
func (m *MessageBus) receiveAnnouncements(client pb.AnnouncementClient) {
  log.Info("Starting announcement client.")

  stream, err := client.Listen(context.Background(), &pb.Request{ClientId:
int32(23)})
  if err != nil {
     log.Fatalf("%v.Listen(_) = _, %v", client, err)
  }

  for {
     announcement, err := stream.Recv()
     if err == io.EOF {
        break
     }
     if err != nil {
        log.Fatalf("%v.Recv(_) = _, %v", client, err)
     }

     log.Debug(announcement.Message)

     m.messages <- announcement.Message
  }
}
```

[40] https://github.com/gorilla/websocket

Όταν ένας χρήστης συνδεθεί (αίτημα στο μονοπάτι /listen) ξεκινάει μία ρουτίνα που ακούει συνεχώς σε αυτό το κανάλι και προωθεί τα μηνύματα μέσω της WebSocket σύνδεσης:

```
func (m *MessageBus) publishAnnouncements(w http.ResponseWriter, r
*http.Request) {
  conn, err := upgrader.Upgrade(w, r, nil)
  if err != nil {
     log.Print("upgrade:", err)
     return
  }
  defer conn.Close()

  for {
     announcement := []byte(<-m.messages)
     err = conn.WriteMessage(websocket.TextMessage, announcement)
     if err != nil {
        log.Error(err)
     }
  }
}
```

Μπορούμε να τρέξουμε την εφαρμογή με την εντολη *go run publisher.go*.

Docker compose

Το Dockerfile είναι σχεδόν ίδιο με αυτό της μικρουπηρεσίας Παραγωγός Βαθμολογιών αλλά εδώ θα περάσουμε ως μεταβλητή περιβάλλοντος την τοποθεσία του gRPC διακομιστή. Επίσης, καθώς το scratch image δεν έχει το shell εγκατεστημένο θα χρησιμοποιήσουμε το alpine:

Dockerfile

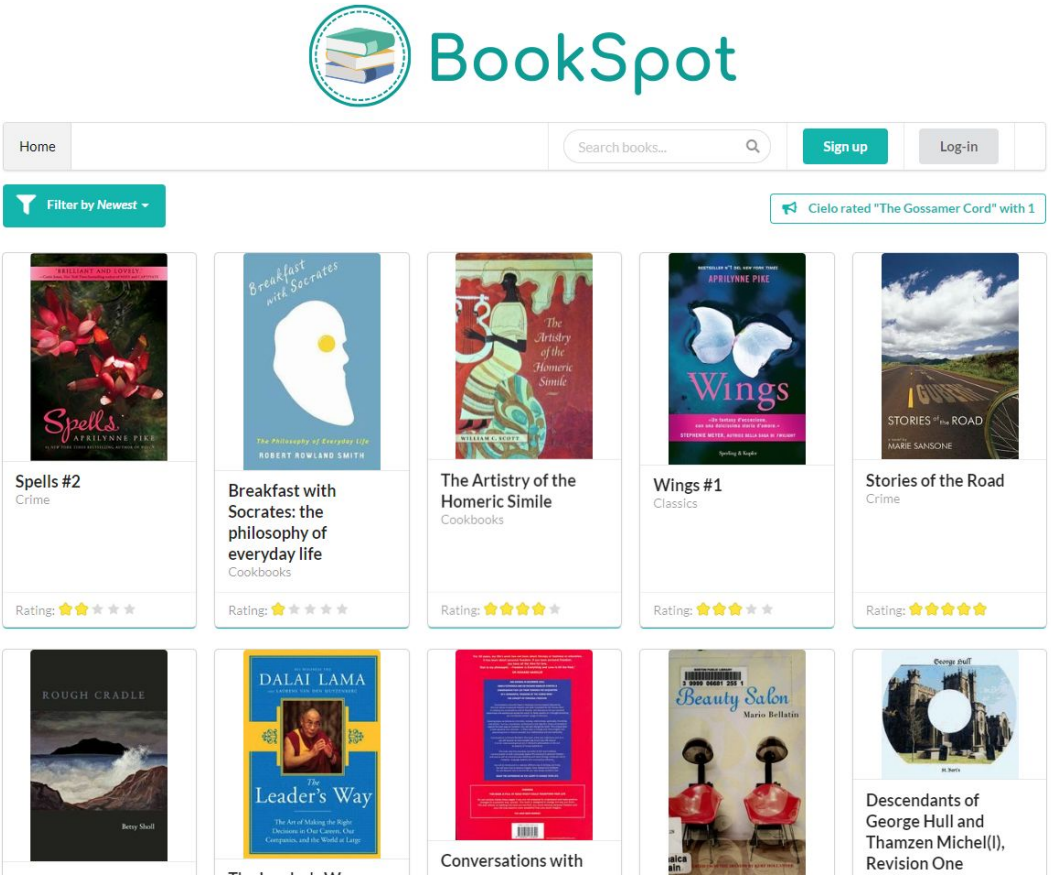
```
ARG SERVER_URI

FROM golang:1.14.1-alpine AS builder

LABEL maintainer="mai18043@uom.edu.gr"

ENV GO111MODULE=on \
   CGO_ENABLED=0
```

```
# Use a different path to avoid mod conflicts.
WORKDIR /app
# Git is needed for dependency installation.
RUN apk update && apk add --no-cache git

COPY go.mod .
COPY go.sum .

# Fetch dependencies.
RUN go mod download
# Verify dependencies.
RUN go mod verify

COPY . .

# Build the executable.
# Target linux and omit debug info
RUN GOOS=linux GOARCH=amd64 go build -ldflags="-w -s"

FROM alpine:3.11

ARG SERVER_URI
ENV ENV_SERVER_URI=${SERVER_URI}

# Copy the executable from the previous stage.
COPY --from=builder /app/publisher /

EXPOSE 8585
ENTRYPOINT /publisher --grpc_server_addr "$ENV_SERVER_URI" --addr ":8585"
```

Άρα μπορούμε να ολοκληρώσουμε το docker compose προσθέτοντας την μικρουπηρεσία Ανακοινώσεις:

```
announcement-publisher:
  image: bookspot/announcement-publisher
  container_name: announcement-publisher
  build:
    context: ./announcement/publisher
    args:
      - SERVER_URI=announcement-server:12000
  ports:
    - "8585:8585"
  depends_on:
    - announcement-server
```

Εκτελώντας την εντολή *docker-compose -f docker-compose.dev.all.yml up* και περιμένοντας για την επιτυχή εκκίνηση των μικρουπηρεσιών, μπορούμε να πλοηγηθούμε στην εφαρμογή πληκτρολογώντας την διεύθυνση http://localhost:8080/.



Εικόνα 11: Κεντρική σελίδα BookSpot

5. Επίλογος

5.1 Σύνοψη και συμπεράσματα

Οι μικρουπηρεσίες είναι μια καινούργια αρχιτεκτονική προσέγγιση, που έχει να επιδείξει τοσο ιστορίες επιτυχίες όσο και αποτυχίες[23]. Η ενασχόληση του συγγραφέα για την υλοποίηση αυτής της εργασίας οδηγησε στην άποψη πως αφορά κυρίως μεγάλα και πολύπλοκα πρότζεκτ. Αυτό δεν σημαίνει πως δεν μπορεί να χρησιμοποιηθεί με επιτυχία για μικρότερου μεγέθους πρότζεκτ, αλλά πρέπει να ζυγιστεί προσεκτικά αν αξίζει μακροπρόθεσμα το διαχειριστικό κόστος που φέρνουν. Όχι μόνο από την οικονομική άποψη, αλλά και για τον ανθρώπινο κόπο της διαχείρισης των απαραίτητων υποδομών.

Με αυτό κατά νου, υπάρχουν αρκετά θετικά στοιχεία. Η αποσύνθεση του επιχειρησιακού τομέα κάνει πιο εύκολη την δουλειά του προγραμματιστή καθώς περιορίζει το γνωστικό φορτίο ανά δεδομένη εργασία. Αυτό οδηγεί σε καλύτερο κώδικα στον οποίο υπάρχει μεγαλύτερη εμπιστοσύνη και μπορεί να συντηρηθεί πιο εύκολα. Επίσης, οι μικρουπηρεσίες τονίζουν την κυριότητα του κώδικα ανά ομάδα, αυξάνοντας το αίσθημα ατομικής ευθύνης και μειώνοντας το χρόνο που σπαταλάτε σε συνεδριάσεις με άλλες ομάδες. Αξίζει να σημειωθεί όμως πως ο προγραμματιστής πρέπει να είναι πρόθυμος να μάθει νέα πράγματα, καθώς ο αριθμός εργαλείων που θα κληθεί να χρησιμοποιήσει ή έστω αλληλεπιδράσει είναι αρκετά υψηλός.

Επίσης, οι μικρουπηρεσίες, προσπαθώντας να ευθυγραμμίσουν επιχειρησιακές και τεχνικές διαδικασίες βοηθούν τον προγραμματιστή να είναι σε θέση να επικοινωνεί πιο αποτελεσματικά με συναδέλφους και διοικητικά στελέχη. Η σωστή επικοινωνία δεν εκτιμάται ιδιαίτερα από πολλούς προγραμματιστές καθώς τεχνικοί όροι μπερδεύονται με επιχειρησιακές έννοιες και οι διευκρινίσεις γίνονται κατόπιν κάποιου δυσάρεστου αποτελέσματος. Έτσι, οι μικρουπηρεσίες εφόσον υλοποιηθούν σωστά μπορούν να φέρουν μια αλλαγή στην γενικότερη φιλοσοφία του οργανισμού.

5.2 Μελλοντικές επεκτάσεις

Η εργασία έδωσε μεγάλη βαρύτητα στην ανάπτυξη μικρουπηρεσιών σε τοπικό περιβάλλον. Η επόμενη κίνηση είναι να γίνουν τα απαραίτητα βήματα ώστε να γίνει η εφαρμογή λειτουργική διαδικτυακά. Δεν προβλέπεται να χρειάζονται μεγάλες αλλαγές για να επιτευχθεί αυτό, αλλα θα είχε ενδιαφέρον να δούμε ποιες προκλήσεις θα αντιμετωπίζαμε για να μεταφέρουμε όλες αυτές τις μικρουπηρεσίες σε μία πλατφόρμα νέφους. Πέρα από αυτό θα μπορούσαμε να εμπλουτίσουμε την επιχειρησιακή λειτουργικότητα ώστε να χρειαστεί να πάρουμε πιο τολμηρές αποφάσεις για την αρχιτεκτονική του συστήματος και να δούμε τα πλην και τα συν κάθε μίας εναλλακτικής.

Παράρτημα

Όλος ο κώδικας της εφαρμογής μπορεί να βρεθεί στον ακόλουθο σύνδεσμο
https://github.com/bagiasn/book-spot

Όλες οι εικόνες δημιουργήθηκαν με χρήση του Microsoft Visio.

Βιβλιογραφία

1. IBM, *Microservices*, 23 Οκτωβρίου 2019.

Διαθέσιμο: https://www.ibm.com/cloud/learn/microservices (Ιούνιος 2020)

2. Dragoni N. et al. (2017), *Microservices: Yesterday, Today, and Tomorrow*. In: Mazzara M., Meyer B. (eds) Present and Ulterior Software Engineering. Springer, Cham

3. Wolff, Eberhard (2016), *Microservices: Flexible Software Architecture*. Addison Wesley Professional

4. M. Villamizar et al., *"Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud"* 2015 10th Computing Colombian Conference (10CCC), Bogota, 2015, pp. 583-590, doi: 10.1109/ColumbianCC.2015.7333476.

5. N-iX, *Microservices vs Monoliths, Which architecture is the best choice for your business?*, 3 Οκτωβρίου 2018.

Διαθέσιμο: https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/ (Ιούνιος 2020)

6. Martin Fowler, *Monolith First*, 3 Ιουνίου 2015.

Διαθέσιμο: https://www.martinfowler.com/bliki/MonolithFirst.html(Ιούνιος 2020)

7. IBM, *SOA vs. microservices*, 6 Σεπτεμβρίου 2018.

Διαθέσιμο: https://www.ibm.com/blogs/cloud-computing/2018/09/06/soa-versus-microservices/ (Ιούνιος 2020)

8. P. Jamshidi et. al., *"Microservices: The Journey So Far and Challenges Ahead"* in IEEE Software, vol. 35, no. 3, pp. 24-35, May/June 2018, doi: 10.1109/MS.2018.2141039.

9. Mark Richards, *Microservices vs. service-oriented architecture*, 6 Ιουλίου 2016. Διαθέσιμο: https://www.oreilly.com/radar/microservices-vs-service-oriented-architecture/ (Ιούνιος 2020)

10. Sam Newman (2015), *Building Microservices*, O'Reilly Media Inc.

11. Eric Evans (2003), *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison Wesley Professional

12. Martin Fowler, *Bounded Context*, 15 Ιανουαρίου 2014. Διαθέσιμο: https://martinfowler.com/bliki/BoundedContext.html (Ιούνιος 2020)

13. Microsoft, *Designing a DDD-oriented microservice*, 8 Οκτωβρίου 2018. Διαθέσιμο: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/ddd-oriented-microservice (Ιούνιος 2020)

14. Microsoft, *Communication in a microservice architecture*, 30 Ιανουαρίου 2020. Διαθέσιμο: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture (Ιούνιος 2020)

15. Roy Tomas Fielding, *Representational State Transfer (REST)*, 2000. Διαθέσιμο: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm (Ιούνιος 2020)

16. Jim Webber, Savas Parastatidis, Ian Robinson (2010), *REST in Practice*, O'Reilly Media Inc.

17. Roy Tomas Fielding, *REST APIs must be hypertext driven*, 20 Οκτωβρίου 2008. Διαθέσιμο: https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (Ιούνιος 2020)

18. AMQP.org (2020), *About*. Διαθέσιμο: https://www.amqp.org/about/what (Ιούνιος 2020)

19. RabbitMQ, Pivotal (2020), *AMQP 0-9-1 Model Explained*. Διαθέσιμο: https://www.rabbitmq.com/tutorials/amqp-concepts.html (Ιούνιος 2020)

20. Google Developers Guide (2020), *Protocol Buffers Language Guide*. Διαθέσιμο: https://developers.google.com/protocol-buffers/docs/proto3 (Ιούνιος 2020)

21. IETF.org, *RFC 6455, The WebSocket Protocol*, Δεκέμβριος 2011. Διαθέσιμο: https://tools.ietf.org/html/rfc6455 (Ιούνιος 2020)

22. Atlassian (2020), *Jira Features*. Διαθέσιμο: https://www.atlassian.com/software/jira/features (Ιούνιος 2020)

23. Jimmy Bogard, *Avoiding Microservice Megadisasters* (YouTube), Ιανουάριος 2017. Διαθέσιμο: https://www.youtube.com/watch?v=gfh-VCTwMw8 (Ιούνιος 2020)