ΠΑΝΕΠΙΣΤΗΜΙΟ ΜΑΚΕΔΟΝΙΑΣ
ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΤΜΗΜΑΤΟΣ ΕΦΑΡΜΟΣΜΕΝΗΣ ΠΛΗΡΟΦΟΡΙΚΗΣ

Υπολογιστική Νέφους:

ΑΝΑΠΤΥΞΗ ΜΙΚΡΟΥΠΗΡΕΣΙΩΝ ΣΕ DOCKER

Διπλωματική Εργασία

του

Χρήστου Γιβανούδη

Θεσσαλονίκη: 24/10/2017

Υπολογιστική Νέφους:
ΑΝΑΠΤΥΞΗ ΜΙΚΡΟΥΠΗΡΕΣΙΩΝ ΣΕ DOCKER

Χρήστος Γιβανούδης

Πτυχίο Εφαρμοσμένης Πληροφορικής, ΠΑΝΕΠΙΣΤΗΜΙΟ ΜΑΚΕΔΟΝΙΑΣ, 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 24/10/2017

Κωνσταντίνος Μαργαρίτης	**Αλέξανδρος Χατζηγεωργίου**	**Θεόδωρος Κασκάλης**
.....	.....	.....

Χρήστος Γιβανούδης

.....

Περίληψη

Πολλές εταιρείες, κυβερνήσεις και οργανισμοί στο εξωτερικό, έχουν δείξει μεγάλο ενδιαφέρον για τις μικρουπηρεσίες. Παράλληλα, αναπτύσσονται πλατφόρμες για την απομόνωση των μικρουπηρεσιών σε Containers, όπως το Docker. Ωστόσο δεν υπάρχει διαθέσιμη έρευνα, στα ελληνικά, για την ανάπτυξη μικρουπηρεσιών στην πλατφόρμα του Docker. Για την εργασία αυτή αναπτύχθηκε μια εφαρμογή, βασισμένη στην αρχιτεκτονική των μικρουπηρεσιών. Η εφαρμογή αυτή θα χρησιμοποιηθεί, σαν παράδειγμα, για την δόμηση, την εκτέλεση και την κλιμάκωση των μικρουπηρεσιών στην πλατφόρμα του Docker. Συμπεραίνεται, έτσι, ότι η πλατφόρμα του Docker είναι μια αποτελεσματική και εύκολη πλατφόρμα για την ανάπτυξη των μικρουπηρεσιών. Επίσης, αναλύονται τα πλεονεκτήματα για την ανάπτυξη των μικρουπηρεσιών σε Containers. Ο σκοπός της διπλωματικής εργασίας είναι η σωστή καθοδήγηση και εκπαίδευση των μελλοντικών προγραμματιστών, που ενδιαφέρονται για την ανάπτυξη των μικρουπηρεσιών και η χρήση του κώδικα αυτής σαν ένα πρακτικό υπόβαθρο στην υλοποίηση μικρουπηρεσιών σε περιβάλλον Docker.

Λέξεις Κλειδιά: Docker, Μικρουπηρεσίες, Python, Rest, API Gateway, Docker Swarm, Docker Stack, Dockerfile, Docker-Compose.

Abstract

Many companies, governments and organization in the world have show a great interesting at Microservices. In the mean time, many platforms are developing to isolate Containers, like the Docker. But, the is not enough research write in Greek, for the development of Microservices in Docker. An application was develop for this research with Microservices architecture. The application can be use as an example for the structure, building and scaling of Microservices in the Docker. The result of the research is that Docker can be a useful and easy platform for the development and execution of Microservices. Also the benefits of execute Microservices in Containers. The purpose of the research is to be a good guide and education tool for the future programmers, whose have interesting in the development of Microservices. They can also use the code as a base for development of their own Microservices applications.

Keywords: Docker, Microservices, Python, Rest, API Gateway, Docker Swarm, Docker Stack, Dockerfile, Docker-Compose.

Πρόλογος – Ευχαριστίες

Η παρούσα μεταπτυχιακή εργασία εκπονήθηκε στο Τμήμα Εφαρμοσμένης Πληροφορικής του Πανεπιστημίου Μακεδονίας Θεσσαλονίκης, υπό την επίβλεψη του καθηγητή κ. Κωνσταντίνου Μαργαρίτη.

Αρχικά, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου, ο οποίος μου εμπιστεύτηκε την παρούσα διπλωματική εργασία και με δέχθηκε στην ομάδα του. Τον ευχαριστώ θερμά για την πολύτιμη βοήθειά του, την επιμονή και υπομονή, την καθοδήγηση και υποστήριξη κατά τη διάρκεια της δουλειάς μου. Ακόμα, τον ευχαριστώ για τις πολύτιμες υποδείξεις του κατά την ανάγνωση και διόρθωση της παρούσας εργασίας.

Ακόμα, θα ήθελα να ευχαριστήσω τους καθηγητές του Μεταπτυχιακού Προγράμματος του Τμήματος Εφαρμοσμένης Πληροφορικής του Πανεπιστημίου Μακεδονίας κ. Χατζηγεωργίου Αλέξανδρο και κ. Κασκάλη Θεόδωρο για την συμμετοχή τους στην τριμελή εξεταστική επιτροπή.

Επίσης, δεν ξεχνώ όλα όσα διδάχθηκα από τους καθηγητές μου και από τους συμφοιτητές μου κατά τη διάρκεια των μαθημάτων στον πρώτο χρόνο του μεταπτυχιακού προγράμματός μου και γι’ αυτό τους ευχαριστώ θερμά.

Ολοκληρώνοντας ένα μεγάλο ευχαριστώ προς τους γονείς και τα αδέλφια μου για την πολύπλευρη υποστήριξη που μου προσέφεραν σ΄ όλη τη διάρκεια των σπουδών μου.

Περιεχόμενα

Κατάλογος Εικόνων

Κατάλογος Πινάκων

1 Εισαγωγή

1.1 Πρόβλημα – Σημαντικότητα του θέματος

Στην σημερινή εποχή έχει παρατηρηθεί αύξηση στο αριθμό διαδικτυακών εφαρμογών. Μαζί αυξάνεται ο αριθμός των προγραμματιστών που απαιτούνται για την ανάπτυξή τους. Οι αρχιτεκτονικές των διαδικτυακών εφαρμογών, όμως, έχουν παραμείνει ίδιες. Η μονολιθική μεθοδολογία απαιτεί από τους προγραμματιστές, την ανάπτυξη μιας εφαρμογής, χρησιμοποιώντας μόνο μια κοινή βάση κώδικα. Καθώς αυξάνεται ο αριθμός των προγραμματιστών, που δουλεύουν, στην εφαρμογή τόσο αυξάνεται ο αριθμός των συγκρούσεων από αλλαγές στην βάση του κώδικα. Οι μικρουπηρεσίες είναι μια καινούργια αρχιτεκτονική για την ανάπτυξη διαδικτυακών εφαρμογών. Οι μικρουπηρεσίες διασπούν τις ογκώδεις και πολύπλοκες εφαρμογές σε μικρότερες και απομονωμένες εφαρμογές. Το Docker είναι ένα εργαλείο για την απομόνωση εφαρμογών, στα περιβάλλοντα που ονομάζονται Containers. Η διπλωματική εργασία θα προσπαθήσει να ενώσει τις δυο οντότητες.

1.2 Σκοπός – Στόχοι

Σκοπός της διπλωματικής είναι η ανάπτυξη και η δόμηση μιας εφαρμογής με την αρχιτεκτονική των μικρουπηρεσιών στην πλατφόρμα του Docker. Επίσης, μπορεί να χρησιμοποιηθεί σαν οδηγός για την εκμάθηση της πλατφόρμας του Docker και την μεθοδολογία των μικρουπηρεσιών από νέους προγραμματιστές, που δείχνουν ενδιαφέρον για το πεδίο, διδάσκοντας τους τα λάθη που πρέπει να αποφεύγουν όταν ξεκινούν.

Στόχος της διπλωματικής είναι η εύρεση του βαθμού δυσκολίας στην υλοποίηση των μικρουπηρεσιών στην πλατφόρμα του Docker. Επίσης, στόχος είναι η απάντηση στο ερώτημα σχετικά με την αξία των μικρουπηρεσιών και την αποτελεσματικότητα του Docker σαν πλατφόρμα ανάπτυξής τους. Η διπλωματική θέτει τα πλαίσια των παραπάνω στόχων.

Αντικείμενα, όπως, το περιεχόμενο της εφαρμογής , η ασφάλεια του συστήματος, η

χρήση πολύπλοκων μεθοδολογιών για την δόμηση και την βελτιστοποίηση του κώδικα ή των βάσεων δεδομένων, δεν περιέχονται στο πλαίσιο της διπλωματικής εργασίας.

1.3 Συνεισφορά

Τα πεδία του Docker και των μικρουπηρεσιών είναι σχετικά καινούργια και δεν υπάρχει αρκετή κλασσική βιβλιογραφία. Κατά την διάρκεια συγγραφής της διπλωματικής εργασίας, για την ανάπτυξη της εφαρμογής χρησιμοποιήθηκε, ως πηγή, η επίσημη τεκμηρίωση του Docker. Επιπλέον χρησιμοποιήθηκαν άρθρα, μαθήματα και ομιλίες από ειδικούς μέσω του διαδικτύου.

1.4 Διάρθρωση της μελέτης

Εργασίες σχετικά με το αντικείμενο της διπλωματικής παρουσιάζονται στο δεύτερο κεφάλαιο. Το τρίτο κεφάλαιο συζητά την αρχιτεκτονική , την δομή της εφαρμογής και οδηγίες για την εκτέλεση της εφαρμογής στο Docker. Στο τέταρτο κεφάλαιο αναπτύσσονται τα συμπεράσματα σχετικά με την αποτελεσματικότητα του Docker στην ανάπτυξη των μικρουπηρεσιών και την αποδοτικότητα της μεθοδολογίας αυτών σε σχέση με τη μονολιθική αρχιτεκτονική.

2 Βιβλιογραφική Επισκόπηση – Θεωρητικό Υπόβαθρο

2.1 Μονολιθική Αρχιτεκτονική

Η μονολιθική αρχιτεκτονική είναι μια παλιά αρχιτεκτονική στην οποία όλη η λογική των οντοτήτων της εφαρμογής, βρίσκεται σε μια βάση κώδικα. Οι μονολιθικές εφαρμογές χωρίζουν τις λειτουργίες τους σε τρεις βασικές στρώσεις.

Στην πρώτη στρώση, η οποία ονομάζεται Front-End, βρίσκονται οι λειτουργίες που αλληλεπιδρούν με τους χρήστες της εφαρμογής. Οι λειτουργίες αυτές είναι υπεύθυνες για την υποδοχή των αιτημάτων και την προβολή των αποτελεσμάτων προς τους χρήστες.

Η δεύτερη στρώση ονομάζεται Business Logic. Η στρώση αυτή αποτελείται από λειτουργίες, που επεξεργάζονται τα αιτήματα, σύμφωνα με την λογική και τους κανόνες της επιχείρησης. Γι' αυτό το λόγο η στρώση αυτή καταλαμβάνει το μεγαλύτερο ποσοστό του κώδικα.

Η τρίτη στρώση είναι γνωστή ως Back-End. Οι λειτουργίες της στρώσης αυτής είναι υπεύθυνες για την επικοινωνία των δυο άλλων στρώσεων με τις βάσεις δεδομένων ή άλλων υπηρεσιών. Η αρχιτεκτονική αυτή είναι καλή σε μικρές διαδικτυακές εφαρμογές, με απλή λογική και ένα ελάχιστο αριθμό οντοτήτων, όπως για παράδειγμα, online καταστήματα. Σε αντίθεση, εφαρμογές, με πολλές οντότητες, οι οποίες αλληλεπιδρούν με πολύπλοκη λογική, παρουσιάζουν υπερβολικά μεγάλες, σε μέγεθος, Business Logic. Αυτό έχει σαν συνέπεια την εμφάνιση σφαλμάτων κατά την λειτουργία τους. Επιπλέον, όλες οι ομάδες ανάπτυξης, πρέπει να δουλεύουν, με την ίδια βάση κώδικα, δυσκολεύοντας έτσι την δοκιμή και την κατανόησή του. Γι' αυτές τις εφαρμογές, την καλύτερη λύση, αποτελεί η Service-Oriented αρχιτεκτονική. [12]

2.2 Service-oriented Αρχιτεκτονική (SOA)

Η Service-oriented Αρχιτεκτονική (SOA) δημιουργήθηκε, με σκοπό, να λύσει τα

προβλήματα που δημιουργεί η μονολιθική, όταν η εφαρμογή γίνεται πολύ μεγάλη ή αποκτά πολύπλοκη λογική. Σε αντίθεση με τη μονολιθική, κάθε οντότητα της εφαρμογής αυτής υλοποιείται σε διαφορετικές βάσεις κώδικα. Κάθε βάση κώδικα, που ικανοποιεί τις λειτουργίες μιας οντότητας, ονομάζεται υπηρεσία (Service). Οι υπηρεσίες επικοινωνούν μεταξύ τους, μέσω web Services ή Messages Queues. Τα Web Services είναι πρωτόκολλα, με σκοπό τη διατύπωση μιας κοινής γλώσσας, που επιτρέπει δυο προγράμματα, γραμμένα σε διαφορετικές γλώσσες προγραμματισμού να επικοινωνούν μεταξύ τους. Οι πιο δημοφιλείς Web Services είναι το SOAP και το Rest. Τα Messages Queues είναι προγράμματα που επιτρέπουν την μετάδοση μηνυμάτων ή συμβάντων. Τα μηνύματα ή συμβάντα μπορούν να αναγνωστούν από διάφορα άλλα προγράμματα. Γνωστά Messages Queues είναι το JMS , RabbitMQ και ActiveMQ.

Η υποδομή, με την οποία, συνδέονται οι υπηρεσίες μεταξύ τους, ονομάζεται Enterprise Services Bus (ESB). Το ESB διαχειρίζεται την ασφάλεια, το φόρτο εργασίας, την καταγραφή σφαλμάτων, την ανακάλυψη νέων υπηρεσιών και τη διαδικτυακή διασύνδεση των υπηρεσιών. Η Service-Oriented Αρχιτεκτονική είναι κατάλληλη για εφαρμογές με πολλές οντότητες και πολύπλοκη λογική, όπως, για παράδειγμα, τα πληροφοριακά συστήματα εταιριών. Οντότητες σ' αυτά τα πληροφοριακά συστήματα, αποτελούν, συνήθως, οι πελάτες, το προσωπικό, τα προϊόντα , οι πρώτες ύλες , οι παραγγελίες και οι πωλήσεις. Κάθε οντότητα έχει την δική της υπηρεσία. Για παράδειγμα, η Customer Service και η Order Service είναι, για τους πελάτες και τις παραγγελίες, αντίστοιχα. Η υπηρεσία παρέχει όλες τις λειτουργίες σχετικά με την οντότητα της. Η υπηρεσία διαχειρίσεως προσωπικού, για παράδειγμα, είναι υπεύθυνη για τη μισθοδοσία και την παροχή αδειών στο προσωπικό της εταιρίας. Για τους παραπάνω λόγους, εφαρμογές, που είναι γραμμένες, με βάση την Service-Oriented αρχιτεκτονική, τείνουν να έχουν κατανοητές και οργανωμένες βάσεις κώδικα. Επιπλέον, κάθε υπηρεσία, μπορεί να αναπτυχθεί και να δοκιμασθεί από μια ομάδα ανάπτυξης, ανεξάρτητα από τις άλλες ομάδες. Επίσης η απομόνωση των υπηρεσιών κάνει εύκολη την αντικατάσταση ή αναβάθμισή τους στα συστήματα. Σε αντίθεση, εφαρμογές βασισμένες στην Service-oriented αρχιτεκτονική έχουν μεγαλύτερο κόστος ανάπτυξης και συντήρησης από τις μονολιθικές εφαρμογές, εξαιτίας της επιβάρυνσης

κόστους του EBS. Υπάρχουσες υπηρεσίες είναι δύσκολο να αφαιρεθούν από το σύστημα, καθώς παρέχουν λειτουργίες που χρειάζονται άλλες υπηρεσίες.

2.3 Μικρουπηρεσίες (Microservices)

Η αρχιτεκτονική των μικρουπηρεσιών είναι η νεότερη από τις δύο προηγούμενες αρχιτεκτονικές. Οι μικρουπηρεσίες παρέχουν μια λειτουργία ανά υπηρεσία, σε αντίθεση με την SOA, που παρέχει όλες τις λειτουργίες μιας οντότητας σε μια υπηρεσία [11]. Γι΄ αυτό το λόγο ονομάζεται μικρουπηρεσία, δηλαδή μικρή υπηρεσία. Κάθε μικρουπηρεσία έχει την δική της βάση κώδικα και την δική της βάση δεδομένων, με σκοπό να είναι ανεξάρτητη από τις άλλες. Η επικοινωνία με τους εξωτερικούς χρήστες γίνεται με την χρήση του Rest [14], ενώ η εσωτερική επικοινωνία με άλλες μικρουπηρεσίες επιτυγχάνεται μέσω της Messages Queues [14].

Η λογική αυτή δημιουργήθηκε με σκοπό την αφαίρεση Microservices από το σύστημα χωρίς την δημιουργία προβλημάτων με άλλες Microservices, λόγω της αφαίρεσης λειτουργιών. Κάτι που στην SOA δεν μπορούσε να γίνει με ευκολία. Επιπλέον λιγότερες λειτουργίες επιτρέπουν τις Microservices, να επικοινωνούν άμεσα με τους εξωτερικούς χρήστες. Αυτό μειώνει τις εσωτερικές αλληλεπιδράσεις της εφαρμογής. Γι' αυτό το λόγο, η αρχιτεκτονική των μικρουπηρεσιών αφαιρεί την ανάγκη για ένα κοινό ESB, μειώνοντας έτσι το κόστος ανάπτυξης και συντήρησης της εφαρμογής. Εξαιτίας, του μικρού κόστους, τα Microservices μπορούν να αναπτυχθούν από μικρές ομάδες ανάπτυξης [1]. Τα μέλη των ομάδων αυτών μπορούν να επικοινωνήσουν πιο εύκολα, καθώς όλοι δουλεύουν για να υλοποιήσουν μια απλή λειτουργία. Η λειτουργία αυτή αποτελεί ένα ξεκάθαρο κοινό στόχο για την ομάδα. Η ομάδα μπορεί να υλοποιήσει το στόχο αυτό, σε ένα απομονωμένο περιβάλλον, χωρίς περισπασμούς από εξωτερικούς παράγοντες ή συγκρούσεις με άλλες ομάδες ανάπτυξης. Εξαιτίας των ελλιπών στοιχείων της, σε σχέση με την SOA, πολλοί προγραμματιστές τη θεωρούν ως μια υποκατηγορία του SOA. Στην πραγματικότητα, η ελλιπής φύση της MicroService αρχιτεκτονικής είναι σχεδιασμένη για εταιρίες που χρειάζονται να προωθήσουν καινούργιες λειτουργίες, λόγω του συνεχούς ανταγωνισμού.

Για παράδειγμα, το Netflix είναι ένα ηλεκτρονικό κατάστημα ταινιών, με μηνιαία συνδρομή. Το Netflix αποτελείται από λειτουργίες, όπως, εξερευνητής του καταστήματος, αναζήτηση ταινιών, αποθήκευση ταινιών, κριτικές, προτάσεις για ταινίες ανάλογα με τις προτιμήσεις του χρήστη και διάφορες άλλες λειτουργίες [19]. Οι περισσότερες από τις λειτουργίες είναι ανεξάρτητες μεταξύ τους, δηλαδή η λειτουργία της αποθήκευσης ταινιών στο Cloud, δεν χρειάζεται να γνωρίζει για τη λειτουργία, που προτείνει ταινίες στους χρήστες, παρόλο που και οι δυο ανήκουν στην ίδια οντότητα. Αυτό επιτρέπει στο Netflix να υλοποιήσει καινούργιες λειτουργίες σε μορφή μικρουπηρεσιών, όπως, για παράδειγμα, την αγορά μιας ταινίας ως δώρο. Η Netflix μπορεί να δώσει την ιδέα αυτή σε μια μικρή ομάδα ανάπτυξης, να την υλοποιήσει σε μια μικρουπηρεσία και να την εγκαταστήσει στο σύστημα. Αν αυτό πετύχει τότε η ομάδα θα είναι ελεύθερη να δοκιμάσει και άλλες ιδέες. Διαφορετικά σε περίπτωση που αποτύχει, το Netflix μπορεί εύκολα να αφαιρέσει την μικρουπηρεσία από το σύστημα. Σε αντίθεση, ένα πληροφοριακό σύστημα μιας βιομηχανίας σιδήρου δεν χρειάζεται να χρησιμοποιήσει Microservices, καθώς οι λειτουργίες του είναι συγκεκριμένες και είναι πολύ μικρή η πιθανότητα να αλλάξουν. Αυτό που αλλάζει είναι οι κανόνες της εταιρίας όσο αφορά τις λειτουργίες. Στον τομέα αυτόν είναι καλύτερη η SOA αρχιτεκτονική.

Συνοψίζοντας, η αρχιτεκτονική των μικρουπηρεσιών, προσφέρει μία πιο οικονομική λύση για τις εταιρίες, που χωρίζουν, τους προγραμματιστές τους σε μικρές ομάδες ανάπτυξης και χρειάζεται να κάνουν συνεχείς αλλαγές στην εφαρμογή τους.

2.4 Εισαγωγή στο Docker

Το Docker είναι μια εργαλειοθήκη. Τα εργαλεία της χρησιμοποιούνται από προγραμματιστές, για τη δημιουργία εκτελέσιμων προγραμμάτων υπολογιστών, γνωστά ως εφαρμογές. Το Docker δουλεύει μόνο για Linux. Μέσω εικονικών μηχανών, όμως, μπορούμε να εκτελέσουμε το Docker σε λειτουργικά συστήματα όπως τα Windows ή τα Mac. Ο υπολογιστής, στον οποίον εκτελείται το Docker καθώς και οι εφαρμογές του, ονομάζεται Host. Υπάρχουν μικρές εφαρμογές σχεδιασμένες, για απλούς χρήστες, όπως

επεξεργαστές κειμένων ή προγράμματα ζωγραφικής. Αντίθετα, υπάρχουν πανάκριβες εφαρμογές που χρησιμοποιούνται από τράπεζες και μεγάλες εταιρίες, για την επεξεργασία συναλλαγών και προσωπικών δεδομένων χιλιάδων ή εκατομμυρίων χρηστών. Το Docker, συνήθως, ασχολείται με την δημιουργία αυτών των ακριβών εφαρμογών.

Το νόημα κάθε εφαρμογής είναι να λάβει, να επεξεργαστεί και να εξάγει σε ένα σύνολο δεδομένων. Τα δεδομένα μπορούν να κατέβουν από το διαδίκτυο μέσω της κάρτας δικτύου ή να διαβαστούν από αρχεία μέσω του σκληρού δίσκου. Ευτυχώς, υπάρχουν έτοιμα προγράμματα, που αναλαμβάνουν την κουραστική εργασία της ανάγνωσης, της επεξεργασίας και της εγγραφής των δεδομένων σε διάφορες συσκευές. Τα παραπάνω προγράμματα ονομάζονται βιβλιοθήκες. Οι βιβλιοθήκες δεν είναι σχεδιασμένες να επικοινωνούν με το χρήστη, αλλά με τις εφαρμογές. Κάθε εφαρμογή χρειάζεται συγκεκριμένες εκδόσεις της βιβλιοθήκης. Νεότερες ή παλαιότερες εκδόσεις μπορεί να μη λειτουργήσουν σωστά με την εφαρμογή. Τι γίνεται, όμως, όταν δύο εφαρμογές χρειάζονται την ίδια βιβλιοθήκη, αλλά διαφορετικές εκδόσεις της.

Βιβλιοθήκες και αρχεία ρυθμίσεων, που υποστηρίζουν μια εφαρμογή, μπορούν να ομαδοποιηθούν σε ένα περιβάλλον. Το περιβάλλον απομονώνει την εφαρμογή, από τις άλλες εφαρμογές του λειτουργικού συστήματος. Μέσω του περιβάλλοντος, κάθε εφαρμογή μαζί με τις υποστηρικτικές βιβλιοθήκες και τις ρυθμίσεις της, μπορεί να αποθηκευτεί σε ένα μικρό πακέτο. Το πακέτο στην ορολογία του Docker ονομάζεται Container. [2]

2.5 Docker Container

Το Docker Container είναι ένα απομονωμένο πακέτο ενός περιβάλλοντος, με σκοπό την εκτέλεση μιας εφαρμογής. Σε ένα λειτουργικό σύστημα μπορούν να εκτελεστούν πολλαπλά Containers. Ο κύκλος ζωής ενός Container περιλαμβάνει την αρχικοποίηση, την εκτέλεση, τη διακοπή και τη διαγραφή του. Κάθε Container είναι αυτόνομο. Δηλαδή το Container υποστηρίζεται από τους δικούς του πόρους, όπως, μεταβλητές περιβάλλοντος, αναγνωριστικά διεργασιών (PID), network Stack και συστήματα αρχείων. Σε κάθε Container παρέχεται μια τοπική IP διεύθυνση, εσωτερική στο Host. Η IP χρησιμοποιείται

με σκοπό την επικοινωνία μεταξύ Containers. Το Docker εκτελεί αυτόματα τη σύνδεση μέσω του Link. Το πρότυπο για την κατασκευή ενός Container ονομάζεται Docker Image. [2]

2.6 Docker Image

Το Docker Image είναι ένα πρότυπο με σκοπό την κατασκευή ενός Container. Το Image μπορεί να παράγει ένα ή πολλαπλά Containers. Τα Images μπορούν εύκολα να αποθηκευτούν, να διαμοιράσουν και να αποτελέσουν θεμέλια άλλων Images.

Ένα Image αποτελείται από μια στοίβα συστημάτων αρχείων. Συστήματα αρχείων είναι μια συλλογή από αρχεία και καταλόγους. Κάθε σύστημα αρχείων αποτελεί μια στρώση στη στοίβα. Το Docker χρησιμοποιεί το Ενιαίο Σύστημα Αρχείων (UnionFS), το οποίο εμφανίζει το Image σαν ένα ομοιόμορφο σύστημα αρχείων, παρόλο που στην πραγματικότητα αποτελείται από ανεξάρτητα συστήματα αρχείων. Όταν ένα Container αρχικοποιείται, του παρέχεται ένα καινούργιο σύστημα αρχείων, στο οποίο μπορεί να γράψει χωρίς να προκαλέσει αλλαγές στις άλλες στρώσεις.

Η παραπάνω αρχιτεκτονική είναι το μυστικό στη δημιουργία γρήγορων και ελαφρών Containers. Το Docker δεν χρειάζεται να αντιγράψει όλα τα αρχεία του Image στο Container, παρά μόνο να του παραχωρήσει μια άδεια στρώση. Αυτό επιταχύνει την δημιουργία πολλαπλών Containers και εξοικονομεί αποθηκευτικό χώρο στο Host.

Υπάρχουν δυο τρόποι για την δημιουργία ενός Image. Στον πρώτο τρόπο, ένα Container μετατρέπεται σε ένα καινούργιο Image μέσω στης εντολής commit. Στο δεύτερο τρόπο, το Image μπορεί να κατασκευαστεί μέσω του Dockerfile. [2]

2.7 Dockerfile

Το Dockerfile είναι ένα αρχείο σύνταξης οδηγιών με σκοπό την κατασκευή ενός Image. Στην αρχή του Dockerfile δηλώνεται ένα έτοιμο Image, γνωστό ως Base . Το Base θα αποτελέσει την πρώτη στρώση της στοίβας. Έπειτα κάθε οδηγία θα δημιουργήσει μια

καινούργια στρώση στη στοίβα. Είναι σημαντικό η συνένωση όμοιων οδηγιών σε μια οδηγία, με σκοπό την αποδοτικότητα του Image. Κατά την κατασκευή του Image, ο προγραμματιστής μπορεί να περιλάβει ένα γενικό πλαίσιο (Context), στο οποίο περιέχεται η εφαρμογή. [3]

2.8 Docker Registry

Το Docker Registry είναι μια αποθήκη από Images. Το Registry μπορεί να είναι τοπικό ή απομακρυσμένο. Σε ένα Registry, τα Images ονομάζονται από το repository και το tag της μορφής repository:tag. Tag είναι η έκδοση του Image. [2]

2.9 Docker Compose

Το Docker Compose είναι ένα εργαλείο για την ομαδοποίηση πολλαπλών Containers σε μια Υπηρεσία (Service). Όλες οι ρυθμίσεις των Containers αποθηκεύονται στο "docker-compose.yml" αρχείο. Το αρχείο αυτό προσδιορίζει ένα σύνολο από Images, τις οποίες, ονομάζει Services. Κάθε Service αποτελείται από διάφορες ρυθμίσεις. Οι ρυθμίσεις αυτές παρουσιάζονται στον πίνακα 1 [8].

Πίνακας 1: Σημαντικές ρυθμίσεις του "docker-compose.yml" [8]

Εντολές	Παράμετροι	Περιγραφή
build	διαδρομή πλαισίου	Το build χτίζει το Dockerfile που βρίσκεται στην διαδρομή, την οποία ορίζει η παράμετρος.
ports	-"host θύρα:Container θύρα" -"host θύρα>:Container θύρα"	Συνδέει τις θύρες του Host, ώστε να ακούνε με τις αντίστοιχες θύρες του Container.
Links	- Όνομα Service …	Συνδέει το Container με τις άλλες Services, με σκοπό την εσωτερική επικοινωνία.
Volumes	-" διαδρομή πελάτη: διαδρομή Container" …	Κάνει mount μεταξύ του καταλόγου που βρίσκεται στην διαδρομή του πελάτη και του καταλόγου στην διαδρομή του Container.

Το Docker-Compose επιτρέπει το κτίσιμο και την έναρξη πολλαπλών διαφορετικών Containers με τις εντολές “docker-compose build” και “docker-compose up” αντίστοιχα. Σε αντίθεση, η εντολή “docker-compose kill” σταματά όλα τα Containers και η εντολή “docker-compose down” διαγράφει τα Containers από το σύστημα. Η εντολή “docker logs" εμφανίζει τα Logs μιας Service. Το Docker Compose διευκολύνει τους προγραμματιστές στην ανάπτυξη, δοκιμή και εγκατάσταση πολλαπλών Containers, καθώς, μπορούν να συντάξουν διαφορετικά “docker-compose.yml” για κάθε στάδιο ανάπτυξης της εφαρμογής. Επιπλέον, διευκολύνεται η μεταφορά της εφαρμογής από σύστημα σε σύστημα, καθώς όλες οι ρυθμίσεις βρίσκονται σε ένα αρχείο. Το αρχείο, αυτό, κρατάει όλες τις εξαρτήσεις της εφαρμογής, όπως βάσεις δεδομένων, Messages Queues, Caches και Web Service APIs.

2.10 Docker Swarm και Stack

Μέχρι, τώρα, δουλεύαμε μόνο σε ένα μηχάνημα Docker. Μπορούμε, όμως, να συνδέσουμε πολλά Docker μηχανήματα μαζί σε μια ομάδα. Η ομάδα αυτή ονομάζεται Swarm. Το Docker μηχάνημα που είναι μέρος ενός Swarm ονομάζεται Node. Τα Nodes χωρίζονται σε δυο ομάδες τους Managers και τους Workers. Οι Managers είναι υπεύθυνοι για τη διαχείριση των Containers που εκτελούνται μέσα στο Swarm. Το Docker προτιμά περιττούς αριθμούς Managers για να διαχειρίζεται τις αποσυνδέσεις των Managers από το σύστημα, σε αντίθεση με την δουλειά των Workers, που είναι να εκτελεί τα Containers του Swarm. Μπορούμε να αρχικοποιήσουμε ένα Swarm ενεργοποιώντας το Swarm Mode στο Docker με την εντολή "docker swarm init". Στη συνέχεια, το Docker αυτό, δημιουργεί tokens με την εντολή "docker swarm join-token".

Τα tokens αυτά θα δοθούν στα άλλα Nodes για να συνδέσουν στο Swarm με την εντολή "docker swarm join". Η Docker Service αποτελείται από πολλαπλά αντίγραφα ενός Container, τα οποία τρέχουν μέσα στο Swarm. Τα Docker services μπορούν να κλιμακωθούν, δηλαδή, να αυξήσουν ή να μειώσουν τον αριθμό των αντιγράφων. Τα Docker

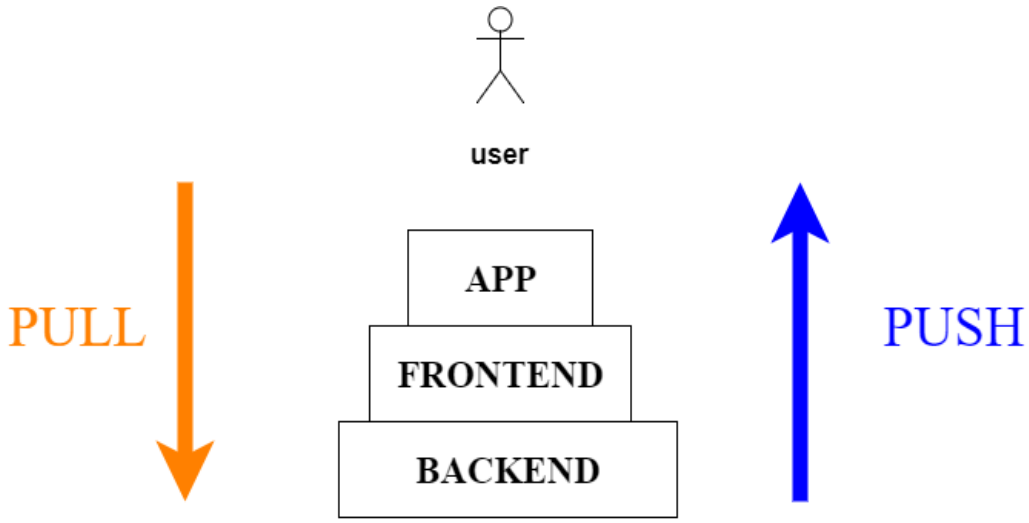
Networks είναι εικονικά δίκτυα στο Swarm, στα οποία επικοινωνούν τα Containers της κάθε Service. Πολλά Docker Services και Docker Networks δημιουργούν ένα Docker Stack. Οι πληροφορίες για κάθε service και network που θα δημιουργηθούν στο Stack μπορούν να αποθηκευτούν στο "docker-compose.yml" αρχείο. Η Docker Stack είναι μια συλλογή από εντολές για τη δημιουργία και τη συντήρηση ενός Stack στο Swarm [7].

3 Μεθοδολογία

3.1 Αρχιτεκτονική της Εφαρμογής

Η αρχιτεκτονική της εφαρμογής βασίζεται σε ένα Pull μοντέλο μικρουπηρεσιών. Το Pull μοντέλο δηλώνει ότι οι εξωτερικοί χρήστες της εφαρμογής δίνουν τις εντολές για τη λήψη ή εγγραφή πληροφοριών στο σύστημα. Σε αντίθεση το Push μοντέλο δηλώνει ότι το σύστημα θα ενημερώσει αυτόματα τους εξωτερικούς χρήστες όταν τα δεδομένα αλλάξουν. Τα Pull και Push μοντέλα μπορούν να χρησιμοποιηθούν σε μονολιθικές εφαρμογές και μικρουπηρεσίες.

Στην εφαρμογή χρησιμοποιείται ένα Pull μοντέλο με μικρουπηρεσίες. Η βασική αρχιτεκτονική χωρίζει την εφαρμογή σε τρεις στρώσεις. Αυτές είναι η Back-End, Front-End και App.



Εικόνα 1: Pull και Push μοντέλο.

Η Back-End στρώση είναι υπεύθυνη για την εγγραφή και ανάγνωση δεδομένων της εφαρμογής. Τα δεδομένα αποθηκεύονται σε διαφορετικές βάσεις δεδομένων και θα σταλθούν στην εφαρμογή, όταν αυτή τα ζητήσει με Http αιτήματα. Το Back-End

αποτελείται από διάφορες βάσεις δεδομένων και υπηρεσίες. Η λέξη Back-End μεταφράζεται στα αγγλικά σε “Πίσω τέλος” που σημαίνει ότι είναι το τελικό κομμάτι στο οποίο θα ταξιδέψουν τα αιτήματα. Συνήθως, οι εξωτερικοί χρήστες δεν έχουν άμεση πρόσβαση στο Back-End, καθώς, εάν μπουν θα προκαλέσουν τεράστια προβλήματα ασφαλείας. Πρόσβαση στο Back-End δίνεται μόνο σε διαχειριστές του συστήματος. Γι’ αυτόν το λόγο, η επικοινωνία των εξωτερικών χρηστών με το Back-End γίνεται μόνο μέσω αιτημάτων διάμεσου του Front-End.

Η Front-End στρώση επεξεργάζεται τα αιτήματα που δέχεται από τους εξωτερικούς χρήστες. Η επεξεργασία αυτή αποτελείται από λειτουργίες, όπως, αυθεντικοποίηση των χρηστών, έλεγχος δικαιωμάτων των χρηστών, διασταύρωση της εγκυρότητας των δεδομένων, που στάλθηκαν μέσω των αιτημάτων, δρομολόγηση των διαφόρων αιτημάτων στις ανάλογες υπηρεσίες, αποθήκευση στατικών αρχείων και καταγραφή συμβάντων και στατιστικών. Το API είναι μια διεπαφή για την επικοινωνία της εφαρμογής, με προγραμματιστές ή προγράμματα, σε μορφή απλών αιτήσεων και έχει προσωρινή μνήμη για γρήγορη πρόσβαση δεδομένων (Caching). Σε αντίθεση με την Back-End, η Front-End επικοινωνεί με τους εξωτερικούς χρήστες διαμέσου ενός Rest API. Το Rest είναι μια μεθοδολογία για τη δόμηση των αιτημάτων και των αποτελεσμάτων που θα δώσει μια εφαρμογή σε HTTP, σε μορφή απλών αιτήσεων, χωρίς την χρήση ενός γραφικού περιβάλλοντος (GUI) για τους απλούς χρήστες. Την δυνατότητα αυτή παρέχει η App στρώση.

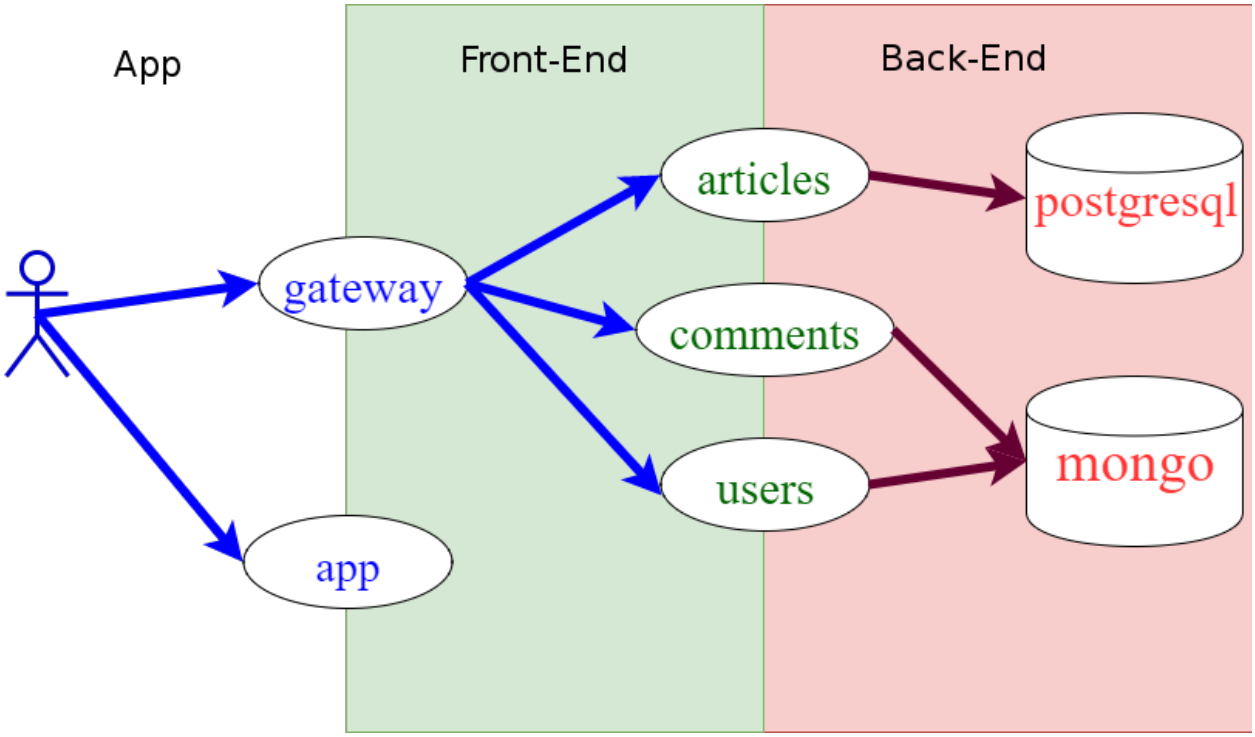
Η App στρώση παρέχει ένα γραφικό περιβάλλον (GUI) για την επικοινωνία χρηστών χωρίς προγραμματιστικές γνώσεις. Το γραφικό περιβάλλον περιλαμβάνει κουμπιά , πλαίσια εισαγωγής κειμένου, πλαίσια προβολής κειμένου. Η εφαρμογή χρησιμοποιεί μια μοναδική σελίδα εφαρμογής (Single-Page Application) για την αναπαράσταση του γραφικού περιβάλλοντος. Λεπτομέρειες για τη μοναδική σελίδα εφαρμογής(Single-Page Application) θα δοθούν παρακάτω στη δόμηση της εφαρμογής. Η App στρώση μπορεί να περιέχει διάφορες άλλες μορφές , όπως εφαρμογή σε κινητή συσκευή, εντολές σε τερματικό παράθυρο και βιβλιοθήκη σε μια γλώσσα προγραμματισμού. Ανεξάρτητα με την μορφή το App μπορεί να επικοινωνήσει στο παρασκήνιο με την εφαρμογή μέσω του Rest API. Στην

περίπτωσή μας η εφαρμογή είναι μόνο για φυσικά πρόσωπα. Οι εξωτερικοί χρήστες όμως μπορούν να είναι και υπολογιστές , συσκευές ή άλλες εφαρμογές.

Συνοψίζοντας, η αρχιτεκτονική της εφαρμογής αυτής βασίζεται σε ένα Pull μοντέλο και περιέχει τρεις στρώσεις, Back-End , Front-End και App. Όμοια αρχιτεκτονική μπορεί να έχει μια μονολιθική εφαρμογή. Αυτό που διαχωρίζει τις μικρουπηρεσίες από τις μονολιθικές είναι η δόμησή τους.

3.2 Δομή της Εφαρμογής

Όπως, αναφέραμε, παραπάνω, η εφαρμογή είναι χωρισμένη σε 3 διαφορετικές στρώσεις. Αυτές παρουσιάζονται στην εικόνα 2.



Εικόνα 2: Δομή της αρχιτεκτονικής.

Η Front-End και Back-End στρώσεις έχουν το δικό τους δίκτυο με το όνομα Front-Net και Back-Net που βρίσκονται μέσα στο Cluster του Docker. Σε αντίθεση η App στρώση εκτελείται στο μηχάνημα του εξωτερικού χρήστη. Η επικοινωνία του Docker cluster και του μηχανήματος του χρήστη γίνεται μέσω του διαδικτύου. Ενώ όλες οι υπηρεσίες και οι βάσεις δεδομένων της εφαρμογής βρίσκονται πακεταρισμένες μέσα σε Docker Containers. Το Docker διαχειρίζεται τα Containers αυτά. Η επικοινωνία μεταξύ των Containers γίνεται αυτόματα από το Docker. Ο λόγος που έχουμε δύο δίκτυα είναι η ασφάλεια και η ταχύτητα των αιτημάτων που ταξιδεύουν σε κάθε δίκτυο. Οι εντολές που δηλώνουν στο Docker τη σύνθεση της εφαρμογής, βρίσκονται στο "docker-compose.yml" αρχείο.

Η Front-End στρώση περιέχει την Gateway, την App υπηρεσία και διαμοιράζεται τις Users, Comments και Articles υπηρεσίες με την Back-End. Στην Gateway βρίσκεται ένας "Nginx" Server, ο οποίος αποτελεί την είσοδο της εφαρμογής. Η Gateway υπηρεσία δρομολογεί τα αιτήματα που έρχονται από τους εξωτερικούς χρήστες στις ανάλογες υπηρεσίες. Αναγνωρίζει τις IP διευθύνσεις των υπηρεσιών καθώς , το Docker, παρέχει ένα DNS Server σε κάθε Container, ο οποίος συσχετίζει την φυσική IP διεύθυνση του Container με το όνομα της υπηρεσίας μέσα στο δίκτυο. Για παράδειγμα, η υπηρεσία Users έχει το domain όνομα "users" και αντιστοιχεί στην 10.0.0.1 10.0.0.2 διεύθυνση στο Front-Net. Το Front-End είναι προγραμματισμένο να ανανεώνει τις διευθύνσεις αυτές με το DNS Server ανά χρονικά διαστήματα σε περίπτωση κλιμάκωσης των υπηρεσιών. Για τη δρομολόγηση των αιτημάτων η Gateway χρησιμοποιεί το URL τους [9]. Για παράδειγμα εάν το URL ξεκινάει με την διαδρομή "users", αυτή δρομολογείται στην Users υπηρεσία. Η διαδρομή αυτή λέγεται Endpoint. Στον πίνακα 2 βρίσκονται όλα τα Endpoints της εφαρμογής και οι ανάλογες υπηρεσίες τους [18].

Πίνακας 2: Endpoints

Endpoint	Μικρουπηρεσία	Περιγραφή
POST /users	Users	Δημιουργεί έναν καινούργιο χρήστη.
GET /users/:id	Users	Προβάλει τις προσωπικές πληροφορίες του χρήστη με αναγνωριστικό :id.
POST /login	Users	Κάνει αυθεντικοποίηση του χρήστη στο σύστημα.
POST /users/:id/isValidToken	Users	Ελέγχει την αυθεντικότητα του token.
GET /articles	Articles	Προβάλει όλα τα άρθρα.
POST /articles	Articles	Δημιουργεί ένα καινούργιο άρθρο.
GET /articles/:id	Articles	Προβάλει το άρθρο με αναγνωριστικό :id
POST /articles/:id/comments	Comments	Δημιουργεί ένα καινούργιο σχόλιο στο άρθρο με αναγνωριστικό :id .
GET /articles/:id/comments	Comments	Προβάλει όλα τα σχόλια στο άρθρο με αναγνωριστικό :id .

Η App υπηρεσία περιέχει τα στατικά αρχεία της App στρώσης. Τέλος, τα υπόλοιπα κομμάτια των υπηρεσιών Users, Articles και Comments ακούν στα αιτήματα που δρομολογεί η Gateway, μέσω του Flask, που είναι μια Rest βιβλιοθήκη για την Python. Οι υπηρεσίες, αυτές, ελέγχουν την αυθεντικότητα των αιτημάτων και πιστοποιούν τα δεδομένα τους.

Στην Back-End στρώση βρίσκεται το υπόλοιπο μισό των υπηρεσιών Users, Articles και Comments, τα οποία επικοινωνούν με τις βάσεις δεδομένων. Η υπηρεσία Users έχει την Postgresql για βάση δεδομένων, καθώς οι SQL βάσεις δεδομένων είναι πιο ασφαλείς για τις συναλλαγές και την αποθήκευση των προσωπικών δεδομένων για τους χρήστες. Η Users υπηρεσία, που είναι υπεύθυνη για τους χρήστες της εφαρμογής αποθηκεύει το “username” και “password” των χρηστών και επίσης, δημιουργεί προσωρινά Tokens για κάθε χρήστη. Αυτά χρησιμοποιούνται από το χρήστη για την αυθεντικοποίησή του σε άλλες υπηρεσίες [10]. Η Users υπηρεσία είναι συνδεμένη με την Postgresql μέσω της Python με την Pony ORM. Οι υπηρεσίες Articles και Comments διαχειρίζονται τους πόρους, τα άρθρα και τα σχόλια αντίστοιχα. Είναι γραμμένες σε Python , τα δεδομένα αποθηκεύονται στην κοινή βάση δεδομένων Mongo, που είναι σχεδιασμένη να κρατάει συλλογές και έγγραφα αρχείων σε μορφή Json. Οι Articles και Comments υπηρεσίες συνδέονται μέσω της PyMongo στη

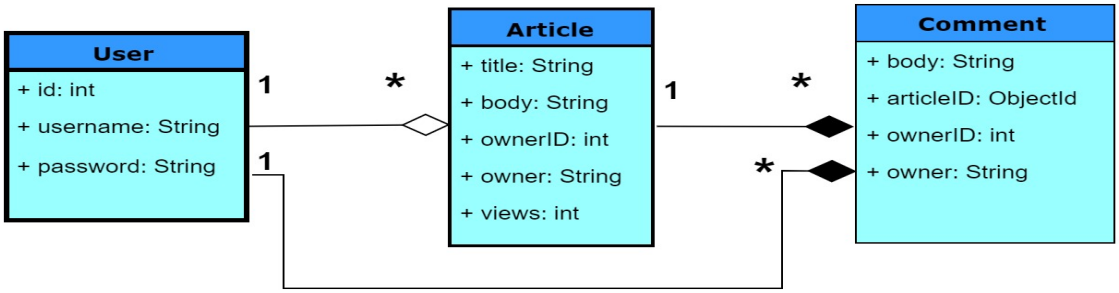
Mongo βάση δεδομένων. Η Mongo κρατάει το “username” και το “ID” για τον συγγραφέα κάθε άρθρου και των σχολίων, για την γρηγορότερη προβολή των δεδομένων. Επίσης οι Articles και Comments υπηρεσίες επικοινωνούν με την Users υπηρεσία για την αυθεντικοποίηση του χρήστη μέσω του “ID” και του “token”.

Η App είναι μία μοναδική σελίδα εφαρμογής (Simple Page Application). Είναι γραμμένη σε Html και JavaScript. Χρησιμοποιεί το Vue Framework και την βιβλιοθήκη Bootstrap για το γραφικό περιβάλλον. Μία μοναδική σελίδα εφαρμογής (Simple Page Application) δηλώνει, ότι η εφαρμογή θα εκτελεστεί μόνο σε μια σελίδα ενός Browser, σε αντίθεση με την παραδοσιακή μέθοδο στην οποία η ιστοσελίδα έρχεται από τον Server. Με την μοναδική σελίδα εφαρμογής (Simple Page Application), η σελίδα ζωγραφίζεται από τον Vue. Το App περιέχει Components, τα οποία απομονώνουν τις βασικές λειτουργίες της εφαρμογής. Για παράδειγμα, ένα Component ζωγραφίζει ένα άρθρο στη σελίδα και ένα άλλο ζωγραφίζει τα σχόλια. Τα Components επικοινωνούν μόνα τους με το Rest API μέσω του Vue-Resource πακέτου και έτσι είναι αυτόνομα μεταξύ τους. Πληροφορίες σχετικά με την ταυτότητα του χρήστη κρατούνται σε μια παγκόσμια αποθήκη με το Vue-Global πακέτο. Η εσωτερική δρομολόγηση μεταξύ των Components γίνεται μέσω του Vue-Router πακέτου. Ο εκτελέσιμος κώδικας του App βρίσκεται συμπιεσμένος στην App υπηρεσία. Κατεβαίνει και εκτελείται στο μηχάνημα του χρήστη, όταν ο χρήστης συνδέεται στο Docker Cluster στην θύρα 80.

Η δομή της εφαρμογής βοηθάει στην κλιμάκωσή της. Ο συνδυασμός των μικρουπηρεσιών και components βοηθάει στην προσθήκη νέων λειτουργιών στην εφαρμογή χωρίς την αλλαγή του υπάρχοντος κώδικα ή των υπηρεσιών. Επίσης παρέχει την δυνατότητα για χρήση ατομικής βάσης δεδομένων. Επιτρέπει τη σύνθεση μικτής ομάδας από προγραμματιστές, διαχειριστές βάσεων δεδομένων και σχεδιαστές γραφικού περιβάλλοντος, για τη δημιουργία και τη συντήρηση μόνο μιας υπηρεσίας.

3.3 Διάγραμμα κλάσεων

Το Domain της εφαρμογής αποτελείται από τρεις διαφορετικές κλάσεις. Οι κλάσεις αυτές είναι η User (Χρήστης), η Article (Άρθρο) και η Comment (Σχόλιο), όπως φαίνεται στην εικόνα 3:



Εικόνα 3: Διάγραμμα Κλάσεων της εφαρμογής.

Η κλάση User δηλώνει το χρήστη της εφαρμογής και έχει τα εξής πεδία:

-το “id” πεδίο, που είναι ένας ακέραιος αριθμός, ο οποίος δηλώνει το αναγνωριστικό κλειδί του κάθε χρήστη

-το “username” πεδίο, που δηλώνει ένα αλφαβητικό στοιχείο, στο οποίο αποθηκεύεται το όνομα του χρήστη και

-το “password” πεδίο, που δηλώνει τον κωδικό πρόσβασης του χρήστη και είναι ένα αλφαβητικό στοιχείο.

Στην κλάση Article ορίζονται τα άρθρα που δημοσιεύονται στην εφαρμογή. Τα πεδία της κλάσης αυτής είναι τα εξής:

-το πεδίο “title” είναι ο τίτλος του άρθρου

-το πεδίο “body” περιέχει το κύριο σώμα του άρθρου

-το πεδίο “ownerID” δηλώνει τον αναγνωριστικό αριθμό του χρήστη, που έχει δημοσιεύσει το άρθρο

-το πεδίο “owner” αποθηκεύει το όνομα του ιδιοκτήτη του άρθρου και

-το πεδίο “views” δηλώνει τον αριθμό των επισκεπτών του άρθρου.

Η κλάση Comment αποθηκεύει τα δεδομένα για κάθε σχόλιο. Τα σχόλια δημοσιεύονται σ΄ ένα άρθρο από τους χρήστες. Τα πεδία της κλάσης είναι:

-το πεδίο “body” περιέχει το σχόλιο

-το πεδίο “articleID” ορίζει το αναγνωριστικό του άρθρου, στο οποίο δημοσιεύθηκε το σχόλιο και τέλος

-τα δυο τελευταία πεδία της Comment κλάσης είναι όμοια με τα πεδία “ownerID” και “owner” της Article κλάσης.

Οι κλάσεις User και Article συνδέονται μεταξύ τους με ένα Association σύμβολο. Στην Article κλάση είναι γνωστή η παρουσία της User, ενώ η User κλάση δεν γνωρίζει την ύπαρξη της Article κλάσης. Κάθε χρήστης μπορεί να δημοσιεύσει πολλά άρθρα, αλλά κάθε άρθρο έχει μόνο έναν ιδιοκτήτη.

Οι κλάσεις Article και Comment συσχετίζονται μεταξύ τους με Composition σύμβολο. Για να υπάρξει η κλάση Comment είναι απαραίτητη η ύπαρξη της Article κλάσης. Κάθε σχόλιο ανήκει μόνο σ΄ ένα άρθρο, ενώ τα άρθρα μπορούν να έχουν πολλαπλά σχόλια.

Η σχέση μεταξύ της User και της Comment συμβολίζεται με το Association σύμβολο. Η ύπαρξη της User είναι γνωστή από την Comment, ενώ αντίθετα η User δεν γνωρίζει ότι υπάρχει η Comment.

3.4 Διάγραμμα αλληλεπίδρασης

Στην εικόνα 4 εμφανίζεται το σχετικό διάγραμμα αλληλεπίδρασης της Users μικρουπηρεσίας με το χρήστη. Η Users μικρουπηρεσία διαθέτει πέντε συναρτήσεις. Η κάθε συνάρτηση λαμβάνει Http μηνύματα, που συμβολίζονται με μαύρα βελάκια. Κάθε συνάρτηση εκτελείται χωριστά. Όταν η Users μικρουπηρεσία λαμβάνει ένα μήνυμα, τότε η μικρουπηρεσία δημιουργεί ένα καινούργιο στιγμιότυπο της αντίστοιχης συνάρτησης.

Αναλυτικά:

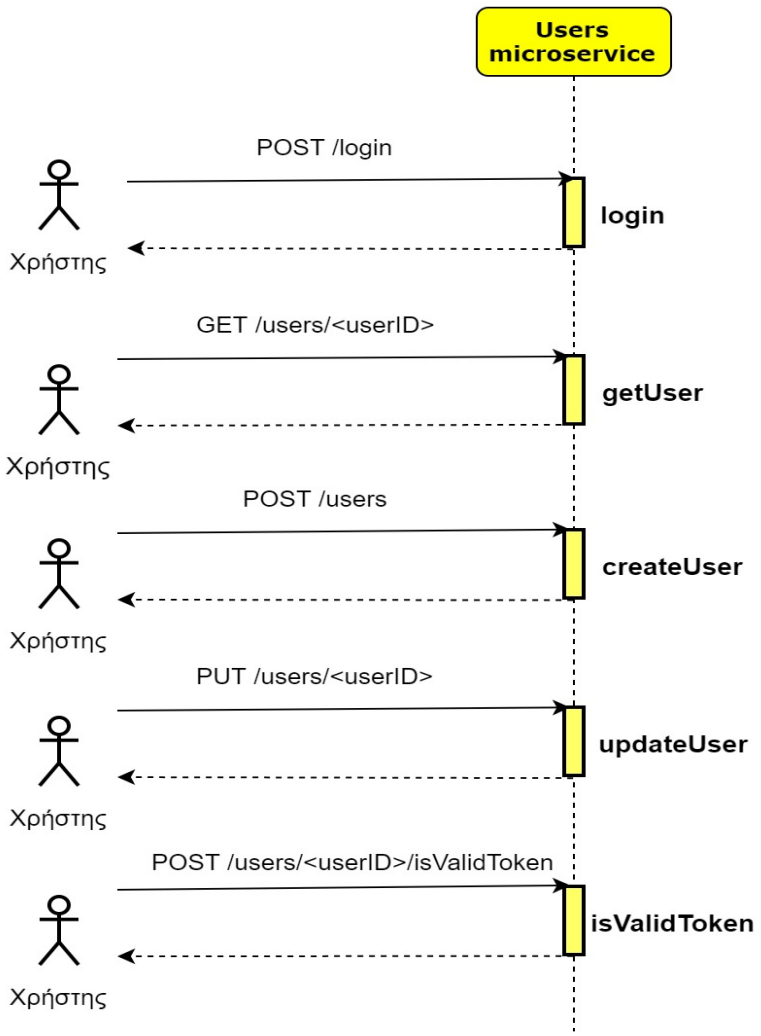
-η συνάρτηση “login” λαμβάνει το “POST /login” Endpoint. Η συνάρτηση αυτή κάνει την αυθεντικοποίηση του χρήστη στην εφαρμογή.

-η “getUser” συνάρτηση ακούει στο “GET /users/<userID>” Endpoint. Σκοπός της είναι η επιστροφή του ανάλογου χρήστη.

-η συνάρτηση “createUser” δημιουργεί έναν καινούργιο χρήστη στην εφαρμογή και ακούει στο “POST /users” Endpoint.

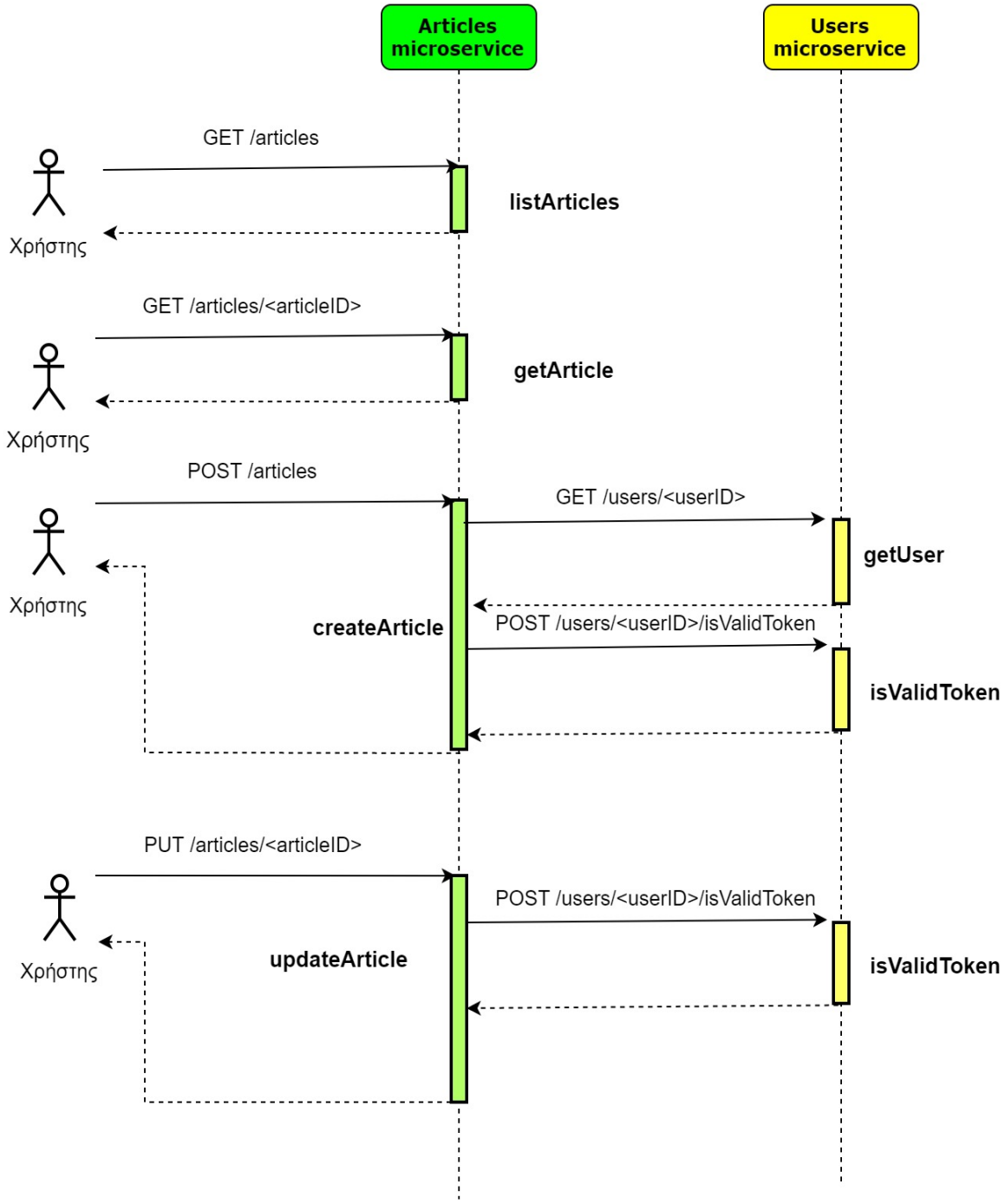
-η updateUser ακούει στο “PUT /users/<userID>” Endpoint και αλλάζει τα δεδομένα του χρήστη και τέλος

-η isValidToken συνάρτηση ελέγχει την αυθεντικότητα του token ασφαλείας που έχει ο χρήστης. Η συνάρτηση αυτή λαμβάνει τα μηνύματα του “POST /users/<userID>/isValidToken” Endpoint.



Εικόνα 4: Διάγραμμα Αλληλεπίδρασης της Users Μικρουπηρεσία.

Η εικόνα 5 παρουσιάζει το διάγραμμα αλληλεπίδρασης της Articles μικρουπηρεσίας. Το διάγραμμα είναι όμοιο στην κατανόηση με το διάγραμμα της εικόνας 4.



Εικόνα 5: Διάγραμμα Αλληλεπίδρασης της Articles Μικρουπηρεσία.

Η Articles μικρουπηρεσία διαθέτει τέσσερις συναρτήσεις:

-η “listArticles” συνάρτηση ακούει στο “GET /articles” Endpoint και ο σκοπός της είναι η προβολή όλων των άρθρων της εφαρμογής πίσω στον χρήστη

-η “getArticle” συνάρτηση επιστρέφει μόνο ένα άρθρο πίσω στο χρήστη και λαμβάνει μηνύματα από το “GET /articles/<articleID>” Endpoint

-η συνάρτηση “createArticle” ακούει στο “POST /articles” Endpoint και ο σκοπός της είναι η δημιουργία ενός καινούργιου άρθρου. Κατά την διάρκεια εκτέλεσης της συνάρτησης θα σταλλεί ένα Http μήνυμα στην Users μικρουπηρεσία, μέσω του “GET /users/<userID>” Endpoint. Σκοπός του μηνύματος αυτού είναι η επιστροφή των δεδομένων του χρήστη που δημιούργησε το άρθρο. Κατόπιν η “createArticle” στέλνει ένα άλλο μήνυμα στο “POST /users/<userID>/isValidToken” Endpoint για να ελέγξει την αυθεντικότητα του “token”, που παρέχει ο χρήστης

-η “updateArticle” συνάρτηση αλλάζει τα δεδομένα του άρθρου. Η συνάρτηση αυτή ακούει στο “PUT /articles/<articleID>” Endpoint. Η συνάρτηση χρειάζεται να στείλει ένα μήνυμα στο “POST /users/<userID>/isValidToken” Endpoint, για να ελέγξει τα δικαιώματα πρόσβασης του χρήστη.

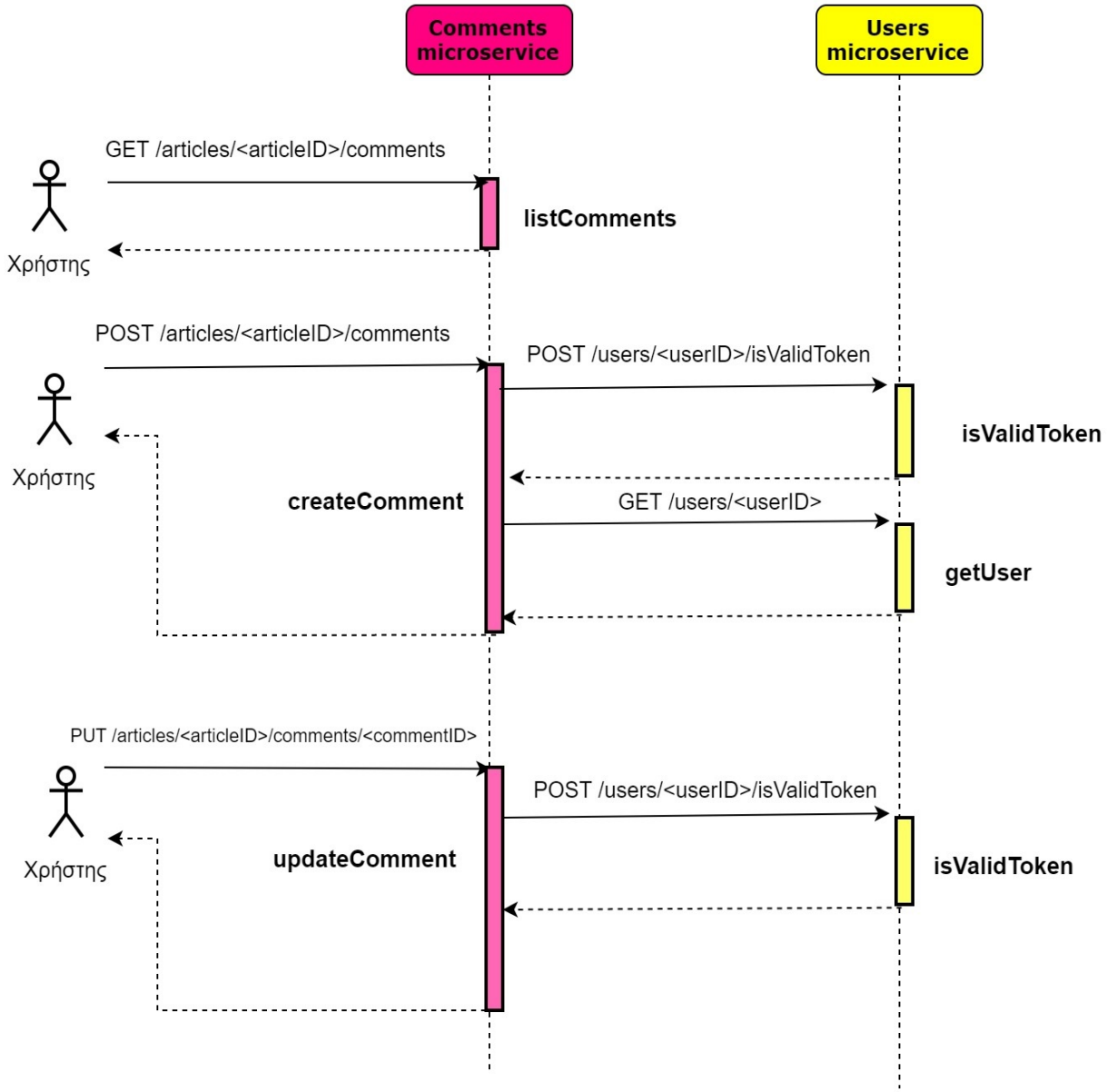
Στην εικόνα 6 παρουσιάζεται το διάγραμμα αλληλεπίδρασης της Comments μικρουπηρεσίας. Η Comments μικρουπηρεσία περιέχει τρεις συναρτήσεις:

-η “ListComments” επιστρέφει πίσω όλα τα σχόλια ενός άρθρου και ακούει στο “/articles/<articleID>/comments” Endpoint.

-η συνάρτηση “createComment” λαμβάνει μηνύματα από το “POST /articles/<articleID>/comments” και σκοπός της είναι η δημιουργία ενός σχολίου στο ανάλογο άρθρο. Η συνάρτηση αυτή χρειάζεται να στέλνει μηνύματα στις “isValidToken” και “getUser” συναρτήσεις της Users μικρουπηρεσίας, με σκοπό τον έλεγχο της αυθεντικότητας και την προβολή του χρήστη.

-η συνάρτηση “updateComment” ακούει στο “PUT /articles/<articleID>/comments/<commentID>” Endpoint. Σκοπός της είναι η αλλαγή ενός

σχολίου. Η συνάρτηση αυτή χρειάζεται να επικοινωνήσει με το “isValidToken” για να ελέγξει τα δικαιώματα του χρήστη.



Εικόνα 6: Διάγραμμα Αλληλεπίδρασης της Comments Μικρουπηρεσία.

3.5 docker-compose.yml

Το “docker-compose.yml” αρχείο κρατάει όλες της οδηγίες για την εκτέλεση της εφαρμογής στο Docker Stack. Επιπλέον, με την εντολή “docker-compose build” μπορούμε να χτίσουμε αυτόματα των κώδικα των μικρουπηρεσιών σε εικόνες, μέσω της Docker-Compose. Το περιεχόμενο του “docker-compose.yml” παρουσιάζεται παρακάτω:

```
version: '3'
services:
  app:
    build: app
    image: chgivan/micro/app:latest
    networks:
      - front-net
    deploy:
      restart_policy:
        condition: on-failure
  gateway:
    build: gateway
    image: chgivan/micro/gateway:latest
    ports:
      - "80:80"
    networks:
      - front-net
    deploy:
      restart_policy:
        condition: on-failure
  comments:
    build: comments
    image: chgivan/micro/comment:latest
    deploy:
      restart_policy:
        condition: on-failure
    environment:
      - USERS_HOST=users:5000
      - DB_HOST=mongodb
    networks:
      - front-net
      - back-net
  articles:
    build: articles
    image: chgivan/micro/articles:latest
```

```
      deploy:
        restart_policy:
          condition: on-failure
      environment:
        - USERS_HOST=users:5000
        - DB_HOST=mongodb
      networks:
        - front-net
        - back-net
    users:
      build: users
      image: chgivan/micro/users:latest
      deploy:
        restart_policy:
          condition: on-failure
      environment:
        - DB_HOST=userdb
        - DB_PASSWORD=123456
        - DB_DATABASE=usersDB
        - DB_USER=my_user
      networks:
        - front-net
        - back-net
    userdb:
      image: Postgres
      environment:
        - POSTGRES_PASSWORD=123456
        - POSTGRES_USER=my_user
        - POSTGRES_DB=usersDB
      networks:
        - back-net
    mongodb:
      image: "mongo"
      networks:
        - back-net
  networks:
    front-net:
    back-net:
```

Σύμφωνα με το εγχειρίδιο [8], όλες οι υπηρεσίες της εφαρμογής, βρίσκονται κάτω από το κλειδί “services: ”. Οι περισσότερες υπηρεσίες έχουν πολιτική επανεκκίνησης, σε περίπτωση αποτυχίας τους. Αυτό γίνεται, επειδή, δεν μπορούμε να ελέγξουμε τη σειρά εκκίνησης των υπηρεσιών και πολλές από αυτές, βασίζονται στην ετοιμότητα άλλων υπηρεσιών, κατά την εκκίνησή τους. Η πολιτική επανεκκίνησης εγγυάται τη συνεχή

επανεκκίνηση μιας υπηρεσίας, μέχρις ότου, οι αναγκαίες υπηρεσίες να έχουν, ήδη, ξεκινήσει.

-Η πρώτη υπηρεσία είναι η App. Το κλειδί “build” αυτής δηλώνει ότι ο κώδικας, που χρειάζεται η App, για να χτιστεί μια εικόνα, βρίσκεται στον κατάλογο “app”. Η App είναι μέλος του Front-Net δικτύου.

-Η δεύτερη υπηρεσία είναι η Gateway. Ο κώδικας για το κτίσιμο της εικόνας, βρίσκεται στο “gateway” κατάλογο. Η υπηρεσία ακούει στη θύρα 80 και ανήκει στο Front-Net δίκτυο.

Στη συνέχεια, ακολουθούν οι μικρουπηρεσίες Users, Articles και Comments. Ο κώδικας κτισίματος των μικρουπηρεσιών αυτών, βρίσκεται στους αντίστοιχους καταλόγους. Οι μικρουπηρεσίες ανήκουν στο Front-Net και Back-Net δίκτυα ταυτόχρονα. Κάτω από το κλειδί “environment:” ορίζονται οι μεταβλητές περιβάλλοντος, που χρειάζεται κάθε μικρουπηρεσία.

Οι τελευταίες υπηρεσίες είναι η UserDB και η MongoDB βάσεις δεδομένων.

Στην UserDB τρέχει η τελευταία έκδοση της Postgres βάσης δεδομένων, η οποία αποθηκεύει τα δεδομένα των χρηστών της εφαρμογής.

Στην MongoDB εκτελείται η Mongo βάση δεδομένων, η οποία περιέχει τα δεδομένα των άρθρων και των σχολίων.

Στο “environment:” κλειδί της UserDB υπηρεσίας δηλώνεται ο χρήστης της βάσης δεδομένων, ο κωδικός πρόσβασης του χρήστη και η βασική βάση δεδομένων που θα έχει το Container. Η UserDB και η mongodb είναι μέλη του Back-Net δικτύου.

Στο τέλος του αρχείου ορίζονται τα Back-Net και Front-Net δίκτυα, κάτω από το κλειδί “networks:”.

3.6 Εκτέλεση της εφαρμογής

Ο πηγαίος κώδικας της εφαρμογής είναι καταχωρημένος στην διαδικτυακή αποθήκη Git και μπορεί να κατέβει με την παρακάτω εντολή, εάν το πρόγραμμα Git Cli είναι εγκαταστημένο.

git clone https://github.com/chgivan/MicroArticles.git

Μέσα στο κατάλογο "microArticles" βρίσκεται όλος ο πηγαίος κώδικας της εφαρμογής. Κάθε υπηρεσία βρίσκεται στον ανάλογο κατάλογο με το αντίστοιχο όνομα. Όλη η δομή των καταλόγων της εφαρμογής παρουσιάζονται στον πίνακας 3.

Πίνακας 3: Κατάλογοι εφαρμογής

Διαδρομή	**Περιγραφή**
docker-compose.yml	Περιέχει τις οδηγίες για το χτίσιμο και την εκτέλεση της εφαρμογής.
app/src/	Περιέχει τον πηγαίο κώδικα στην app στρώση.
app/dist/	Όλος ο προαγωγικός κώδικας που θα αποσταλεί στο μηχάνημα του χρήστη.
gateway/proxy.conf	Κρατάει τις ρυθμίσεις για την δρομολόγηση των αιτημάτων του Rest API.
articles/articlesBackend.py	Ο κώδικας της Articles μικρουπηρεσίας.
comments/commentsBackend.py	Ο κώδικας της Comments μικρουπηρεσίας.
users/usersBackend.py	Ο κώδικας της Users μικρουπηρεσίας.
*/Dockerfile	Κρατάει της εντολές για το χτίσιμο των Docker Images κάθε μικρουπηρεσίας.
*/libs.txt	Κρατάει τις βιβλιοθήκες της Python που χρειάζεται κάθε μικρουπηρεσία.

Όλες οι οδηγίες για την εκτέλεση της εφαρμογής στο Docker βρίσκονται στο "docker-compose.yml" αρχείο στον κεντρικό φάκελο. Μέσα στο αρχείο δηλώνονται όλες οι υπηρεσίες και βάσεις δεδομένων της εφαρμογής, κάτω από το κλειδί Services. Ενώ τα δίκτυα βρίσκονται κάτω από το κλειδί "networks". Το κλειδί "build" χτίζει την Docker

εικόνα της υπηρεσίας, μέσω οδηγιών που παρέχονται από το Dockerfile αρχείο, το οποίο βρίσκεται στον υποκατάλογο κάθε υπηρεσίας. Το κλειδί “image” δηλώνει το όνομα στην Docker εικόνα που θα χτιστεί. Η παρακάτω εντολή χτίζει όλες της υπηρεσίες. Εάν δηλωθεί το όνομα των υπηρεσιών το Docker-Compose θα χτίσει μόνο αυτές.

docker-compose build

Για την εκτέλεση της εφαρμογής στο Docker πρέπει να είναι ενεργοποιημένο το Docker Swarm. Το Docker Swarm μπορεί να ενεργοποιηθεί με την παρακάτω εντολή και στην παράμετρο με σημαία “–advertise-addr” δηλώνουμε την δημόσια IP διεύθυνση του μηχανήματος [5].

docker swarm init –advertise-addr < IP διεύθυνση μηχανήματος>

Όταν το Docker Swarm έχει ενεργοποιηθεί τότε μπορούμε να αρχικοποιήσουμε την εφαρμογή με την εντολή [5].

docker stack deploy -c docker-compose.yml app

Το Docker Stack Deploy παίρνει σαν παράμετρο στη σημαία -c τη διαδρομή του “docker-compose.yml” αρχείου. Στο τέλος της εντολής δηλώνεται το όνομα της εφαρμογής , που στην περίπτωση μας είναι App. Το Docker Stack θα αρχικοποιήσει πρώτα τα δίκτυα και στη συνέχεια τις υπηρεσίες . Για να δούμε εάν όλες οι υπηρεσίες της εφαρμογής τρέχουν, μπορούμε να εκτελέσουμε την εντολή “ps”.

docker stack ps app

Πολλές υπηρεσίες, βασίζονται σε άλλες, για να ξεκινήσουν πριν από αυτές. Το Docker Stack, όμως, δεν μπορεί να υποχρεώσει τις υπηρεσίες αυτές να περιμένουν, γι’ αυτό

δηλώνουμε στο “docker-compose.yml” αρχείο, κάτω από το κλειδί “deploy”, την πολιτική επανεκκίνησης της υπηρεσίας, σε περίπτωση αποτυχίας. Για παράδειγμα, η Users υπηρεσία πρέπει να συνδεθεί στην αρχή με την Users βάση δεδομένων. Εάν η Users αποτύχει να συνδεθεί, γιατί η Users βάση δεδομένων δεν είναι έτοιμη ακόμα, τότε η Users θα ξεκινήσει αργότερα. Αλλιώς η διαδικασία αυτή επαναλαμβάνεται μέχρι να είναι έτοιμη. Τέλος, μπορούμε να σταματήσουμε την εφαρμογή με την εντολή “rm”.

```
docker stack rm app
```

3.7 Περιήγηση στο γραφικό περιβάλλον

Όταν η εφαρμογή τρέχει, μπορούμε να δούμε την App της εφαρμογής, εφόσον συνδεθούμε με τον Browser στη θύρα 80, στη διεύθυνση του μηχανήματος, που βρίσκεται το Docker.

Η πρώτη σελίδα που εμφανίζεται στο γραφικό περιβάλλον είναι η σελίδα που προβάλει μια λίστα με όλα τα άρθρα. Στην περίπτωση που η εφαρμογή ξεκινά για πρώτη φορά, η λίστα θα είναι άδεια γιατί δεν υπάρχουν άρθρα στη βάση δεδομένων. Προτού δημιουργήσουμε ένα καινούργιο άρθρο, πρέπει, πρώτα να είμαστε εγγεγραμμένοι στο σύστημα. Στο πάνω δεξιό μέρος της σελίδας εμφανίζονται δυο κουμπιά, το πρώτο, δηλώνει, την δημιουργία ενός καινούργιου άρθρου και το δεύτερο, την εγγραφή ενός καινούργιου χρήστη. Πατώντας το δεύτερο κουμπί μεταφερόμαστε στην σελίδα εγγραφής νέου χρήστη. Πατάμε το Checkbox “Register” για να δημιουργήσουμε έναν καινούργιο χρήστη έναντι της αυθεντικοποίησης χρήστη. Εισάγουμε το όνομα του χρήστη και τον κωδικό πρόσβασης δύο φορές στα ανάλογα πλαίσια του κειμένου.

Στη συνέχεια, πατάμε το πράσινο κουμπί για την αποστολή του μηνύματος στο Rest API και εάν η εγγραφή έγινε με επιτυχία εμφανίζεται το μήνυμα “δημιουργία χρήστη” και πατάμε ξανά το πράσινο κουμπί για την αυθεντικοποίησή μας στο σύστημα. Τότε θα γυρίσει ξανά στην αρχική σελίδα. Από την αρχική σελίδα, μπορούμε, πατώντας το πρώτο κουμπί για την δημιουργία ενός καινούργιου άρθρου, να εμφανιστεί η σελίδα.

Εισάγουμε στα πεδία κειμένου τον τίτλο και το σώμα του άρθρου και πατάμε το πράσινο κουμπί “save”. Η σελίδα ανακατευθύνεται στη σελίδα προβολής του καινούργιου άρθρου. Για να προβάλλουμε οποιοδήποτε άρθρο πατάμε το μπλε κουμπί “view”, που βρίσκεται δίπλα σε κάθε άρθρο, στην αρχική σελίδα. Στη σελίδα που προβάλλεται το άρθρο εμφανίζεται ο αριθμός των προβολών του άρθρου , ο συγγραφέας του, το σώμα του και τα σχόλια. Όλοι οι χρήστες μπορούν να γράψουν ένα σχόλιο για κάθε άρθρο. Εισάγουν το κείμενο του άρθρου στο πλαίσιο κειμένου και πατούν το πράσινο κουμπί της αποστολής.

4 Ανάλυση του κώδικα

4.1 Dockerfiles των Μικρουπηρεσιών

Τα Dockerfiles δηλώνουν τις οδηγίες, που δημιουργούν την εικόνα της κάθε υπηρεσίας στο Docker [3]. Οι υπηρεσίες Users, Articles και Comments έχουν Dockerfiles αρχεία με όμοιες οδηγίες, οι οποίες έχουν την ίδια δομή μεταξύ τους.

Η οδηγία "FROM Python:3" δηλώνεται στην αρχή και ορίζει την αρχική εικόνα, πάνω στην οποία θα χτιστεί η νέα εικόνα. Στην περίπτωσή μας η αρχική εικόνα είναι η Python έκδοση 3.

Μέσω της οδηγίας "CMD ["Python", "app/<όνομα αρχείο της υπηρεσία>.py"]" δηλώνεται στο Docker η εντολή του τερματικού, η οποία θα εκτελεστεί στην εκκίνηση του Container. Στην περίπτωσή μας το Container θα εκτελέσει το Rest Server μέσω της Python και ο Server ακούει τα Rest API αιτήματα.

Με την οδηγία "ADD libs.txt /app/" αντιγράφεται το αρχείο "libs.txt" στο κατάλογο App της εικόνας. Το αρχείο αυτό κρατάει όλες τις βιβλιοθήκες που χρειάζεται η υπηρεσία για να τρέξει.

Η οδηγία "RUN pip install -r /app/libs.txt" εκτελεί την εντολή "pip" στο τερματικό της εικόνας. Σκοπός της "pip" είναι η εγκατάσταση των βιβλιοθηκών της Python στην εικόνα. Τα ονόματα αυτών δηλώνονται στα αρχεία "libs.txt" .

Σκοπός της οδηγίας "ADD <όνομα αρχείο της υπηρεσία>.py /app/<όνομα αρχείο της υπηρεσία>.py" είναι να αντιγράψει το αρχείο με τον κώδικα της υπηρεσίας στον κατάλογο App της εικόνας του Docker.

Η σειρά των οδηγιών είναι σημαντική, καθώς το Docker θα δημιουργήσει μια καινούργια στρώση στην εικόνα. Όταν η εικόνα ξαναχτίζεται, το Docker ελέγχει για αλλαγές στο περιεχόμενο των οδηγιών. Εάν δεν βρεθούν αλλαγές, τότε χρησιμοποιεί την Cache της στρώσης. Αλλιώς θα χτίσει μια καινούργια στρώση στην εικόνα και θα ξαναχτίσει όλες τις στρώσεις των επόμενων οδηγιών ανεξάρτητα εάν αυτές έχουν αλλάξει ή όχι.

Οι οδηγίες πρέπει να ταξινομούνται με αύξουσα σειρά. Οι οδηγίες με τη μικρότερη

πιθανότητα να αλλάξουν το περιεχόμενό τους βρίσκονται στην αρχή, ενώ οι οδηγίες με τη μεγαλύτερη πιθανότητα βρίσκονται στο τέλος του Dockerfile.

4.2 Flask Βιβλιοθήκη

Το Flask είναι μια βιβλιοθήκη της Python με σκοπό την επεξεργασία των Rest αιτημάτων [20]. Το Flask τρέχει στις υπηρεσίες Users, Articles, Comments. Η Flask class εισάγεται στον κώδικα της υπηρεσίας από το Flask Module με την εντολή "from flask import Flask". Ο Rest Server βρίσκεται στο αντικείμενο "app" του κώδικα της υπηρεσίας και δημιουργείται από τον κατασκευαστή της κλάσης Flask, με την εντολή "app = Flask(__name__)", ενώ παίρνει σαν παράμετρο το όνομα του Module που τρέχει η υπηρεσία. Επειδή η υπηρεσία εκτελείται σαν κύριο πρόγραμμα η τιμή της "__name__" θα είναι "__main__" .

Τα Endpoints μπορούν να δηλωθούν με την μέθοδο "@app.route(url , methods=[])". Η "@app.route" παίρνει σαν πρώτη παράμετρο την URL διαδρομή, που ακούει το Endpoint. Η δεύτερη προαιρετική παράμετρος δέχεται μια λίστα με όλες τις Http μεθόδους του αιτήματος που θα δεχτεί το Endpoint, όπως "GET", "POST" κ.α [20]. Κάτω από την "@app.route" δηλώνεται η συνάρτηση, που καλείται από το Flask, με σκοπό να εξυπηρετήσει το Endpoint τις υπηρεσίας.

Με τη μέθοδο "app.run" εκτελείται ο Rest Server. Η μέθοδος παίρνει σαν παράμετρο το "host" με τιμή "0.0.0.0" [20], για να πάρει τη διεύθυνση του Container, στο οποίο εκτελείται η υπηρεσία. Η παράμετρος "debug", που δηλώνει στο Server, να εμφανίζει τα λεπτομερή μηνύματα σφαλμάτων και η παράμετρος port, που δηλώνει ,τη θύρα που ακούει ο Server.

Με το αντικείμενο "request" του Module Flask μπορούμε να πάρουμε τις παραμέτρους των Http αιτημάτων. Με τη μέθοδο "request.args.get" μπορούμε να λάβουμε την τιμή της παραμέτρου των αιτημάτων της μεθόδου "GET". Η μέθοδος "GET" έχει τρείς παραμέτρους, την "key", την "default" και την "type" [20]. Η "key" ορίζει το όνομα της παραμέτρου, η "default" την τιμή, σε περίπτωση που η παράμετρος δεν υπάρχει και η

“type” δηλώνει τον τύπο της τιμής, που θα επιστρέψει πίσω.

Αντιθέτως, αιτήματα με την μέθοδο “POST” αποθηκεύουν τις τιμές στο σώμα του αιτήματος σε μορφή Json. Μπορούμε να λάβουμε το αντικείμενο του Json με τη μέθοδο “request.get_Json” και να ελέγξουμε εάν η τιμή υπάρχει στο Json με την έκφραση “in”.

Οι συναρτήσεις των Endpoints ελέγχουν εάν μια παράμετρος υπάρχει στο αίτημα. Σε περίπτωση, που δεν υπάρχει, τότε επιστρέφει μια απάντηση με Http status code 400, που δηλώνει, ότι οι παράμετροι του αιτήματος δεν είναι επαρκείς ή σωστές [12].

Για την επιστροφή μιας Json απάντησης πίσω στον πελάτη, χρησιμοποιείται η “getResponse” συνάρτηση, που είναι μια γενική συνάρτηση και χρησιμοποιείται σε όλες τις μικρουπηρεσίες. Σκοπός της είναι η δημιουργία μιας Http απάντησης προς το χρήστη. Το σώμα της “getResponse” συνάρτησης παρουσιάζεται παρακάτω:

```
def getResponse(status, **kwargs):
    obj = {}
    if kwargs is not None:
        obj = kwargs
    resp = jsonify(obj)
    resp.status_code = status
    return resp
```

Η πρώτη παράμετρος της “getResponse” συνάρτησης είναι ένας ακέραιος αριθμός, με το όνομα “status” και δηλώνει την κατάσταση του Http αιτήματος. Η δεύτερη παράμετρος “**kwargs” είναι ένα λεξικό, που κρατάει σαν κλειδιά το όνομα των επιπλέον παραμέτρων , οι οποίες δηλώνονται στο κάλεσμα της συνάρτησης. Αυτό γίνεται λόγω των ** που δηλώνονται στην αρχή της παραμέτρου. Ενώ, σαν τιμές στο λεξικό, ορίζονται οι τιμές των παραμέτρων αυτών. Εάν δεν δοθούν επιπλέον παράμετροι στην “getResponse” συνάρτηση, η “kwargs” θα είναι ίση με None.

Για παράδειγμα, στο κάλεσμα της παρακάτω συνάρτησης:

```
getResponse (200, body="Everything is ok", code = 4, reply=False)
```

το "kwargs" θα είναι ίσο με:

```
kwargs = {
 "body": "Everything is ok",
 "code": 4,
 "reply": False
}
```

Στην αρχή της "getResponse" συνάρτησης, δηλώνεται ένα κενό αντικείμενο "obj", το οποίο θα επιστρέφει όταν το "kwargs" είναι None. Σε περίπτωση που το "kwargs" δεν είναι None τότε το "kwargs" θα αντιγράφει το "obj" αντικείμενο.

Στην συνέχεια το "obj" θα μετατραπεί σε Http απάντηση με Json σώμα μέσω της Flask συνάρτησης "jsonify" [20]. Έπειτα με το πεδίο "resp.status_code" ορίζεται η Http κατάσταση της απάντησης. Τέλος, η απάντηση επιστρέφεται πίσω από τη συνάρτηση.

Το αντικείμενο "environ" του Module OS κρατάει τις μεταβλητές του περιβάλλοντος του Container. Οι μεταβλητές του περιβάλλοντος δηλώνονται στο "docker-compose.yml" στο κλειδί "environment:". Μέσω αυτών των μεταβλητών, μπορούμε να δηλώσουμε τιμές για τη σύνδεση της υπηρεσίας με τη βάση δεδομένων. Για παράδειγμα, οι μεταβλητές μπορούν να έχουν τιμές, όπως, τον κωδικό πρόσβασης, το όνομα του χρήστη και τα Alias άλλων υπηρεσιών. Με τη μέθοδο environ.get επιστρέφεται η τιμή της μεταβλητής στην υπηρεσία. Η πρώτη παράμετρος δηλώνει το όνομα της μεταβλητής και η δεύτερη παράμετρος την τιμή που θα επιστρέψει, εάν η μεταβλητή δεν υπάρχει.

4.3 Users Μικρουπηρεσία

Στην users μικρουπηρεσία βρίσκονται όλες οι λειτουργίες για τη διαχείριση των χρηστών στην εφαρμογή. Ο κώδικας της Articles μικρουπηρεσίας βρίσκεται στο αρχείο "users/usersBackend.py". Οι βιβλιοθήκες, που φορτώνονται, είναι η Pony, η Flask, η UUID. Η UUID χρησιμοποιείται για τη δημιουργία τυχερών μοναδικών αριθμών , που θα χρησιμοποιηθούν ως Tokens για την αυθεντικοποίηση των χρηστών, όπως θα ζητηθούν από άλλες μικρουπηρεσίες. Τέλος, η "environ" χρησιμοποιείται για τις μεταβλητές περιβάλλοντος, όπως φαίνεται παρακάτω:

```
from pony import orm
from flask import Flask, jsonify, request
import uuid, Json
from os import environ
```

Στη συνέχεια, γίνεται η εισαγωγή των μεταβλητών περιβάλλοντος με τις εντολές:

```
port = int(environ.get("PORT", 5000))
debug = bool(environ.get("DEBUG", False))
dbhost = environ.get("DB_HOST")
dbuser = environ.get("DB_USER")
dbpassword = environ.get("DB_PASSWORD")
dbase = environ.get("DB_DATABASE")
```

Εκτός από την "port" και την "debug", που δίνονται για την Flask, οι υπόλοιπες μεταβλητές δίνονται για τη βάση δεδομένων. Η "dbhost" δηλώνει τη διεύθυνση IP του μηχανήματος, που βρίσκεται η βάση δεδομένων Postgres. Εάν η μικρουπηρεσία δημιουργηθεί σε ένα Stack, η μεταβλητή αυτή θα έχει το όνομα της υπηρεσίας που εκτελεί την εικόνα της Postgres. Το όνομα αυτό χρησιμοποιείται σαν Alias για τις IP διευθύνσεις κάθε Container που εκτελεί την Postgres στην Stack. Το Alias γίνεται με ένα μικρο DNS Server που τρέχει σε κάθε Container της Stack [9]. Η "dbuser" δηλώνει το όνομα του

χρήστη της Postgres, ενώ το “dbpassword” τον κωδικό πρόσβασης του χρήστη αυτού. Τέλος, η μεταβλητή “dbase” δηλώνει σε ποια βάση δεδομένων μέσα στην Postgres θα συνδεθεί η μικρουπηρεσία.

Οι μεταβλητές αυτές κατά την ανάπτυξη του Stack μπορούν να αποθηκευτούν στο “docker-compose.yml” αρχείο κάτω από την Users μικρουπηρεσία , όπως φαίνεται παρακάτω:

```
environment:
      - DB_HOST=userdb
      - DB_PASSWORD=123456
      - DB_DATABASE=usersDB
      - DB_USER=my_user
```

Στο επόμενο κομμάτι του κώδικα αρχικοποιούμε τις βιβλιοθήκες Flask και Pony στα αντικείμενα “app” και “db” αντίστοιχα, με τις κάτωθι εντολές:

```
app = Flask(__name__)
db = orm.Database()
db.bind(
    provider='Postgres',
    user=dbuser,
    password=dbpassword,
    host=dbhost,
    database=dbase
)
```

Η Pony βιβλιοθήκη επικοινωνεί με την Postgres βάση δεδομένων και παρέχει το ORM (Object-Relational Mapping) [16]. Μπορούμε να εισάγουμε το ORM αντικείμενο από το Pony Module με την εντολή “from pony import orm”. Η κλάση “orm.Database” κατασκευάζει το αντικείμενο “db”.

Η “db.bind” μέθοδος συνδέει την Pony με τη βάση δεδομένων Postgres [16]. Στις παραμέτρους της, δηλώνονται οι πληροφορίες, που χρειάζεται να δοθούν για τη σύνδεση. Στην παράμετρο “provider” δηλώνεται στην Pony, ότι θα συνδέεται σε μια Postgres βάση δεδομένων με την αλφαβητική σταθερά 'Postgres'. Οι υπόλοιπες παράμετροι είναι οι

μεταβλητές περιβάλλοντος, που αναφέραμε παραπάνω.

Στον παρακάτω κώδικα δηλώνεται ένα λεξικό για τα Tokens και το Domain Model του χρήστη:

```
tokens = {}
class User(db.Entity):
    id = orm.PrimaryKey(int, auto=True)
    username = orm.Required(str, unique=True)
    password = orm.Required(str)
```

Στο λεξικό “tokens” υπάρχουν πολλές εγγραφές. Κάθε εγγραφή αποτελείται από ένα κλειδί και μια τιμή. Το αναγνωριστικό του χρήστη είναι το κλειδί, ενώ το Token αυθεντικοποίηση του χρήστη είναι η τιμή [10]. Η κλάση User δηλώνει το μοντέλο ενός χρήστη στην βάση δεδομένων. Η κλάση αυτή θα μετατραπεί σ΄ ένα πίνακα User μέσα στη βάση δεδομένων με τρεις στήλες, την “id” στήλη, την “username” στήλη και την “password” στήλη. Η “id” είναι ένα ακέραιο αναγνωριστικό κλειδί του χρήστη. Δημιουργείται αυτόματα και είναι μοναδικό για κάθε χρήστη. Η “username” είναι το αλφαβητικό όνομα του χρήστη και πρέπει να είναι μοναδικό σ΄ όλους τους χρήστες. Τέλος, το “password” είναι ένας αλφαβητικός κωδικός πρόσβασης, που χρησιμοποιεί ο χρήστης για να αυθεντικοποιήσει την ταυτότητα του στην εφαρμογή.

Τέλος, με την μέθοδο “db.generate_mapping” συνδέουμε τον πινάκα User της βάσης δεδομένων με την κλάση User [16]. Σε περίπτωση που δεν υπάρχει ο πίνακας User στη βάση, δημιουργείται αυτόματα από τη μέθοδο “db.generate_mapping”. Με τη μέθοδο “orm.sql_debug” δηλώνουμε, να εμφανίζονται στα Logs της μικρουπηρεσίας, τα SQL αιτήματα, που θα δημιουργεί αυτόματα η Pony. Αυτό είναι χρήσιμο στον εντοπισμό εσφαλμένων SQL αιτημάτων, καθώς το αυτόματο σύστημα της Pony μπορεί μερικές φορές να αποδειχθεί ανισόρροπο στη δημιουργία πολύπλοκων SQL αιτημάτων.

Οι παραπάνω εντολές υλοποιούνται ως εξής:

```
db.generate_mapping(create_tables=True)
orm.sql_debug(True)
```

Στην συνέχεια του κώδικα υλοποιούνται τα endpoints της μικρουπηρεσίας. Το πρώτο Endpoint είναι το “login”. Το “login” Endpoint ακούει στη διεύθυνση “/login” με την Http μέθοδο “POST”, όπως φαίνεται παρακάτω:

```
@app.route("/login", methods=['POST'])
def login():
```

Μέσα στη συνάρτηση του “login”, η πρώτη λειτουργία που γίνεται, είναι ο έλεγχος των Http παραμέτρων, όπως φαίνεται στον ακόλουθο κώδικα:

```
params = request.Json
errFlag, errMsg = validParams(params)
if errFlag:
    return getResponse(400, message=errMsg)
```

Η μέθοδος “request.json” επιστρέφει το σώμα του Http αιτήματος σε μορφή Json αντικειμένου, το οποίο ορίζεται στην “params” μεταβλητή. Η συνάρτηση “validParams” ελέγχει τις παραμέτρους του Http αιτήματος. Το σώμα της “validParams” συνάρτησης παρουσιάζεται παρακάτω:

```
def validParams(params):
    errorFlag = False
    errorMsg = ""
    if not "username" in params:
        errorMsg += "Missing username field.\n"
        errorFlag = True
    if not "password" in params:
        errorMsg += "Missing password field.\n"
        errorFlag = True
    return errorFlag, errorMsg
```

Η “validParams” αρχικοποιεί δυο μεταβλητές, την “errorFlag” και την “errorMsg”. Η “errorFlag” είναι μια λογική σημαία, στην οποία δηλώνεται, εάν υπάρχει κάποιο σφάλμα στις παραμέτρους, ενώ η “errorMsg” είναι ένα αλφαβητικό που περιγράφει το σφάλμα αυτό.

Στη συνέχεια, ελέγχεται εάν το “username” είναι δηλωμένο στις παραμέτρους. Εάν δεν είναι, τότε, σηκώνεται η σημαία “errorFlag” και αναγράφεται το σφάλμα στο “errorMsg”. Ομοίως γίνεται και με το “password” πεδίο. Τέλος, οι μεταβλητές “errorFlag” και “errorMsg” επιστρέφονται από τη συνάρτηση.

Πίσω στην “login” συνάρτηση, η “errFlag” ελέγχεται εάν είναι ενεργή. Εάν είναι, τότε η “login” συνάρτηση επιστρέφει πίσω μια απάντηση προς το χρήστη. Η απάντηση θα έχει Http κωδικό 400, πράγμα που δηλώνει, ότι τα πεδία του σώματος, που έδωσε ο χρήστης στο Http αίτημα είναι εσφαλμένα ή ελλειπή. Στην παράμετρο “message” δίνεται το “errMsg”, δηλαδή το μήνυμα του σφάλματος προς το χρήστη. Η δημιουργία της απάντησης γίνεται με την “getResponse” συνάρτησης, όπως φαίνεται παρακάτω:

```
errFlag, errMsg = validParams(params)
if errFlag:
    return getResponse(400, message=errMsg)
```

Μετά από τον έλεγχο στο σώμα του Http αιτήματος, εάν όλα πάνε καλά, ο κώδικας συνεχίζει στο try μπλοκ. Το μπλοκ αυτό εμφανίζεται παρακάτω:

```
try:
    with orm.db_session:
        user = orm.select(
            user for user in User
                if user.password == params["password"]
                and user.username == params["username"]
            ).first()
        if user is None:
            return getResponse(
```

```
                404,
                message="Wrong username or password"
            )
        global tokens
        token = str(uuid.uuid4())
        tokens[str(user.id)] = token
        return getResponse(
            200,
            userID = str(user.id),
            token=token
        )
    except orm.core.ObjectNotFound:
        return getResponse(
            404,
            message="Wrong username or password"
        )
```

Το try μπλοκ θα σταματήσει την εκτέλεση του κώδικα, εάν ο κώδικας παρουσιάσει κάποιο σφάλμα. Όταν ένα σφάλμα παρουσιασθεί, τότε θα εκτελεστεί ο κώδικας μέσα στο except μπλοκ. Στην περίπτωση μας καλούμε από τη βάση δεδομένων να μας επιστρέψει ένα User μοντέλο πίσω, με βάση το αναγνωριστικό του. Στην περίπτωση που το αναγνωριστικό αυτό δεν υπάρχει, τότε το Pony θα παρουσιάσει ένα ObjectNotFound σφάλμα [16], το οποίο θα λάβει το except μπλοκ. Θα επιστραφεί τότε πίσω στο χρήστη μια απάντηση με Http κωδικό 404, που σημαίνει ότι ο πόρος που ζητήθηκε δεν υπάρχει [12], με το αντίστοιχο μήνυμα.

Μέσα στο try μπλοκ καλείται ένα with μπλοκ. Για να επικοινωνήσουμε με τη βάση δεδομένων πρέπει να γίνει μια Session. Η Session κατά τη σύνδεση, μεταξύ της βάσης δεδομένων και της μικρουπηρεσίας, ανοίγει για την αποστολή πολλαπλών αιτημάτων. Η αλλαγές στη βάση δεδομένων θα αποθηκευθούν στην βάση, όταν η Session τελειώσει με επιτυχές Commit [16].

Το μπλοκ “with orm.db_session” δημιουργεί ένα καινούργιο Session με τη βάση δεδομένων. Όταν το μπλοκ with τελειώσει, τότε θα γίνει αυτόματο Commit στη βάση δεδομένων και η Session θα τερματιστεί, εκτός εάν καλεστεί η “orm.abort” μέθοδος [16].

Στην αρχή, πρέπει να βρούμε, εάν το όνομα χρήστη και ο κωδικός πρόσβασης αντιστοιχούν σε έναν υπάρχοντα χρήστη, με την “orm.select” μέθοδο.

```
user = orm.select(
            user for user in User
                if user.password == params["password"]
                and user.username == params["username"]
        ).first()
```

Ο παραπάνω κώδικας στην “orm.select” θα μετατραπεί από την orm στο ακόλουθο SQL αίτημα:

```
Select * from user
where username=”<Όνομα χρήστη που έδωσε ο χρήστης>”
and  password=”<κωδικό πρόσβαση που έδωσε ο χρήστης.>”
```

Τέλος, η μέθοδος θα επιστρέψει μια λίστα, με όλους τους χρήστες, που καλύπτουν τα χαρακτηριστικά αυτά. Εμείς θα πάρουμε τον πρώτο χρήστη από τη λίστα με την μέθοδο “first”. Εάν ο χρήστης είναι None, αυτό σημαίνει ότι ο κωδικός πρόσβασης ή το όνομα χρήστη δεν υπάρχουν και θα επιστρέψει τότε στο χρήστη μια απάντηση με αριθμό κατάστασης 404.

Στη συνέχεια, έχοντας το User μοντέλο, πρέπει να δημιουργήσουμε ένα Token για την αυθεντικοποίηση του χρήστη στην εφαρμογή, από άλλες μικρουπηρεσίες, όπως φαίνεται στον παρακάτω κώδικα:

```
global tokens
token = str(uuid.uuid4())
tokens[str(user.id)] = token
return getResponse(200, userID = str(user.id), token=token)
```

Η “global tokens” καθορίζει, ότι οι αλλαγές που θα γίνουν στην “tokens”, δεν θα είναι τοπικές, μόνο στην συνάρτηση, αλλά θα είναι καθολικές, σε όλη τη σκοπιά της μικρουπηρεσίας.

Στη συνέχεια, με την μέθοδο “uuid.uuid4” δημιουργούμε ένα καινούργιο και μοναδικό Token [10] . Αυτό θα το μετατρέψουμε σε αλφαβητικό και θα το αποθηκεύσουμε σαν τιμή στο λεξικό “tokens”, με κλειδί το αναγνωριστικό του χρήστη. Τέλος, θα επιτρέψουμε μια απάντηση, με αριθμό κατάστασης 200, που σημαίνει ότι όλα πήγαν καλά [12]. Μαζί θα στείλουμε το αναγνωριστικό και το Token του χρήστη.

Το δεύτερο Endpoint είναι το “getUser”. Το “getUser” Endpoint ακούει στη διεύθυνση /users/<userID> με την Http μέθοδο “GET”, όπως φαίνεται παρακάτω:

```
@app.route("/users/<userID>", methods=["GET"])
def getUser(userID):
    try:
        with orm.db_session():
            user = User[userID]
            return getResponse(
                200,
                username=user.username,
                get="/users/{}".format(userID)
            )
    except orm.core.ObjectNotFound:
        return getResponse(
            404,
            message="User id {} doesn't exist".format(userID)
        )
```

Η “getUser” συνάρτηση επιστρέφει πίσω τα δεδομένα του χρήστη. Παίρνει σαν παράμετρο το αναγνωριστικό του χρήστη, το οποίο δίνεται στη διαδρομή του Http αιτήματος. Μέσα στο with μπλοκ, δημιουργείται ένα καινούργιο Session με τη βάση δεδομένων. Με την εντολή “User[userID]”, επιστρέφεται το μοντέλο του χρήστη, με το αναγνωριστικό “userID”. Εάν το αναγνωριστικό αυτό δεν υπάρχει στη βάση δεδομένων, τότε θα παρουσιαστεί ένα ObjectNotFound σφάλμα. Το except μπλοκ θα εντοπίσει το σφάλμα αυτό και θα επιστρέψει μια 404 απάντηση.

Σε αντίθεση, εάν το αναγνωριστικό αυτό υπάρχει στη βάση δεδομένων, τότε θα επιστραφεί πίσω μια 200 απάντηση με το όνομα χρήστη και τη Rest διαδρομή του.

Το τρίτο Endpoint είναι το “createUser”. Το “createUser” Endpoint ακούει στην διεύθυνση “/users” με την Http μέθοδο “POST”, όπως φαίνεται παρακάτω:

```
@app.route("/users", methods=["POST"])
def createUser():
    params = request.Json
    errFlag, errMsg = validParams(params)
    if errFlag:
        return getResponse(400, message=errMsg)

    userID = None
    try:
        with orm.db_session:
            user = User(
                username=params["username"],
                password=params["password"]
            )
            orm.commit()
            userID = user.id
    except orm.core.TransactionIntegrityError:
            return getResponse(
                400,
                message="User with username {} already exist!"
                    .format(params["username"])
            )

    return getResponse(
        201,
        userID=str(userID),
        get="/users/{}".format(userID)
    )
```

Η “createUser” συνάρτηση δημιουργεί ένα καινούργιο χρήστη στην εφαρμογή. Η συνάρτηση αυτή παίρνει ως παραμέτρους, το όνομα του χρήστη και τον κωδικό πρόσβασής του. Οι παράμετροι αυτοί βρίσκονται στο Json σώμα του Http αιτήματος.

Στην αρχή, η “createUser” συνάρτηση ελέγχει το σώμα του αιτήματος, όπως γίνεται στην “login” συνάρτηση. Στη συνέχεια, η “createUser” προσπαθεί να δημιουργήσει έναν καινούργιο χρήστη με τις κάτωθι εντολές:

```
user = User(
            username=params["username"],
            password=params["password"]
            )
orm.commit()
```

Εδώ η συνάρτηση κάνει ένα χειροκίνητο Commit καθώς θέλουμε να ελέγξουμε εάν το όνομα του χρήστη υπάρχει ήδη στην βάση δεδομένων. Εάν δεν υπάρχει τότε δημιουργείται ένα TransactionIntegrityError σφάλμα από την Pony [16]. Το σφάλμα λαμβάνεται από την except και επιστρέφεται μια 400 απάντηση πίσω στο χρήστη.

Τέλος, εάν όλα πάνε καλά, τότε θα λάβουμε το αναγνωριστικό του χρήστη και θα το επιστρέψουμε μια 201 απάντηση, που σημαίνει ότι ένας καινούργιος πόρος δημιουργήθηκε [12].

```
userID = user.id
return getResponse(
        201,
        userID=str(userID),
        get="/users/{}".format(userID)
    )
```

Το τέταρτο Endpoint είναι το “updateUser”. Το “updateUser” Endpoint ακούει στη διεύθυνση “/users/<userID>” με την Http μέθοδο “PUT”, όπως φαίνεται παρακάτω:

```
@app.route("/users/<userID>", methods=["PUT"])
def updateUser(userID):
    params = request.Json
    if not "token" in params:
        return getResponse(400,message="Missing token!")
    if not validToken(token=params["token"],userID = userID):
        return getResponse(403,message="Access Denied!!")

    if not "password" in params:
        return getResponse(400, message="No password given!!")

    try:
        with orm.db_session:
            user = User[userID]
            user.password = params["password"]
            return getResponse(
                200,
```

```
                get="/users/{}".format(userID),
            message="Updated data of user with id {}"
                  .format(userID)
            )
    except orm.core.ObjectNotFound:
        return getResponse(
            404,
            message="User ID {} doesn't exist!".format(userID)
        )
```

Η "updateUser" συνάρτηση αλλάζει τον κωδικό πρόσβασης του χρήστη. Παίρνει σαν παράμετρο το αναγνωριστικό του χρήστη από τη διαδρομή του αιτήματος. Στην αρχή, η updateUser συνάρτηση ελέγχει το Token και τον κωδικό πρόσβασης, που βρίσκονται στο σώμα του Http αιτήματος και στέλνει τις ανάλογες απαντήσεις, εάν παρουσιασθούν κάποια σφάλματα.

Έπειτα η u"pdateUser" συνάρτηση, ελέγχει την αυθεντικότητα του Token με τη συνάρτηση "validToken", η οποία παρουσιάζεται παρακάτω:

```
def validToken(token, userID):
    global tokens
    if not userID in tokens:
        return False
    return tokens[userID] == token
```

Η "validToken" παίρνει ως παραμέτρους το Token και το αναγνωριστικό του χρήστη. Ελέγχει εάν το αναγνωριστικό υπάρχει σαν κλειδί στο λεξικό "tokens". Στην περίπτωση που δεν υπάρχει θα επιστρέψει μια Ψευδής τιμή. Αλλιώς θα επιστρέψει την λογική τιμή της ισότητας μεταξύ του "token" και του " tokens[userID]".

Η "updateUser" συνάρτηση βρίσκει το User μοντέλο στην βάση δεδομένων και αλλάζει το "password" πεδίο. Τελειώνοντας επιστρέφει πίσω στο χρήστη μια απάντηση.

Το πέμπτο Endpoint είναι το "isValidToken", που ακούει στην διεύθυνση "/users/<userID>/isValidToken" με την Http μέθοδο "POST", όπως φαίνεται παρακάτω:

```
@app.route("/users/<userID>/isValidToken", methods=["POST"])
def isValidToken(userID):
```

```
    data = request.get_Json(force=True, silent=True)
    if data is None:
        return getResponse(400, message="Fail to readed Json data")
    errFlag = False
    errMsg = ''
    if not "token" in data:
        errFlag = True
        errMsg += "Missing field token"
    if errFlag:
        return getResponse(400, message=errMsg)

    return getResponse(
        200,
        isValid=validToken(token=data["token"],userID=userID)
    )
```

Η "isValidToken" συνάρτηση ελέγχει την αυθεντικότητα του Token, που παρέχει ο χρήστης, όταν συνδέεται σε άλλες μικρουπηρεσίες. Παίρνει ως παράμετρο το αναγνωριστικό του χρήστη από τη διαδρομή του αιτήματος. Το Token ελέγχεται εάν βρίσκεται στο Json σώμα του αιτήματος. Τέλος, ελέγχεται η αυθεντικότητα του Token με την "validToken" συνάρτηση.

4.4 Articles Μικρουπηρεσία

Στην Articles μικρουπηρεσία βρίσκονται όλες οι λειτουργίες για την διαχείριση των άρθρων της εφαρμογής. Ο κώδικας της Articles μικρουπηρεσίας βρίσκεται στο αρχείο "articles/articlesBackend.py".

Οι βιβλιοθήκες, που φορτώνονται, είναι η Pymongo, η Flask, η Bson.Objectid, η Environ και η Requests . Η Bson.Objectid χρησιμοποιείται για τη δημιουργία των αναγνωριστικών στα άρθρα [15]. Η environ χρησιμοποιείται για τις μεταβλητές περιβάλλοντος. Η Requests για την επικοινωνία της μικρουπηρεσίας με την users μικρουπηρεσία, μέσω του Http πρωτοκόλλου.

Οι μεταβλητές περιβάλλοντος που εισάγονται, εκτός από την "port" και "debug", είναι η "users_host" και η "db_host". Η "users_host" κρατάει την IP διεύθυνση της Users μικρουπηρεσίας και η "db_host" ορίζει την IP διεύθυνση της Mongodb βάσης δεδομένων,

όπως φαίνεται παρακάτω:

```
users_host = environ.get("USERS_HOST")
db_host = environ.get("DB_HOST")
debug = bool(environ.get("DEBUG",False))
port = int(environ.get("PORT",5000))
```

Οι μεταβλητές περιβάλλοντος παρέχονται από το “docker-compose.yml” αρχείο και βρίσκονται κάτω από το “article” κλειδί.

```
environment:
      - USERS_HOST=users:5000
      - DB_HOST=mongodb
```

Στην έναρξη της μικρουπηρεσίας αρχικοποιούμε την Flask και συνδεόμαστε με την Mongodb βάση δεδομένων.

```
clientDB = pymongo.MongoClient('mongodb://{}:27017'.format(db_host))
db = clientDB["articles"]
articles = db["articles-collection"]
app = Flask(__name__)
```

Με την κλάση MongoClient γίνεται η σύνδεση με τη βάση δεδομένων [15]. Ως παράμετρο παίρνει την διαμορφωνόμενη IP διεύθυνση της Mongodb. Έπειτα φορτώνεται στην “db” η βάση δεδομένων "articles" με την εντολή “db = clientDB[‘articles’]”. Στη συνέχεια, η συλλογή “articles-collectin” φορτώνεται στην “articles” μεταβλητή μέσω της “db” [15].

Το πρώτο Endpoint για την “articles” μικρουπηρεσία είναι το “listArticle”. Το “listArticle” Endpoint ακούει στη διεύθυνση “/articles” με την Http μέθοδο “GET”. Η “listArticle” συνάρτηση παρουσιάζεται παρακάτω:

```
@app.route("/articles", methods=["GET"])
def listArticle():
```

```
    params = request.args
    limit = params.get(key="limit",default=10,type=int)
    ownerID = params.get(key="ownerID",default=None,type=int)
    query = {}
    if not ownerID is None:
        query["ownerID"] = ownerID
    cursor = articles.find(query)
                   .limit(limit)
                   .sort('views',pymongo.DESCENDING)

    results = []
    cursor.rewind()
    for article in cursor:
        results.append({
            "id":str(article["_id"]),
            "title":article["title"],
            "owner":article["owner"],
            "ownerID":article["ownerID"],
            "views":article["views"]
        })
    resp = jsonify(results)
    resp.status_code = 200
    return resp
```

Η "listArticle" συνάρτηση επιστρέφει μια λίστα με όλα τα άρθρα της εφαρμογής. Στην αρχή, η "listArticle" παίρνει δυο παραμέτρους από το αίτημα, έναν ακέραιο "limit", που ορίζει πόσα άρθρα θα επιστρέψουν πίσω και έναν άλλον ακέραιο "ownerID", που ορίζει το αναγνωριστικό του χρήστη, με σκοπό να επιστραφούν μόνο τα άρθρα που έγραψε ο συγκεκριμένος χρήστης.

Στη συνέχεια με τη μέθοδο "articles.find" αναζητούμε τα άρθρα και εάν ο χρήστης έδωσε "ownerID", το περνάει ως παράμετρο στο "query", με σκοπό να επιστρέψει όλα τα άρθρα που ανήκουν στο χρήστη [15]. Έπειτα γίνεται pipe των αποτελεσμάτων στη μέθοδο "limit", η οποία μειώνει τον αριθμό των αποτελεσμάτων στον αριθμό της μεταβλητής "limit". Στα καινούργια αποτελέσματα ξαναγίνεται pipe με την μέθοδο "sort", η οποία ταξινομεί τα αποτελέσματα με βάση το πεδίο "views", σε φθίνουσα σειρά, όπως φαίνεται παρακάτω:

```
query = {}
if not ownerID is None:
    query["ownerID"] = ownerID
cursor = articles.find(query)
                    .limit(limit)
                    .sort('views',pymongo.DESCENDING)
```

Τέλος, τα δεδομένα των αποτελέσματων μετατρέπονται σε μια λίστα και περνούν στο σώμα την απάντηση μέσω της jsonify. Έπειτα η απάντηση επιστρέφεται στο χρήστη με αριθμό κατάστασης 200.

Το δεύτερο Endpoint είναι το “getArticle”. Το “getArticle” Endpoint ακούει στην διεύθυνση “/articles/<articleID>” με την Http μέθοδο “GET”. Η “getArticle” συνάρτηση παρουσιάζεται παρακάτω:

```
@app.route("/articles/<articleID>")
def getArticle(articleID):
    article = articles.find_one({'_id': ObjectId(articleID)})
    if article is None:
        return getResponse(
           404,
           message="Articles with id {} doesn't exist".format(articleID)
         )
    articles.update({'_id': ObjectId(articleID)},{"$inc":{"views":1}})
    return getResponse(
        200,
        title=article["title"],
        body=article["body"],
        owner=article["owner"],
        ownerID=article["ownerID"],
        views=article["views"] + 1
    )
```

Η “getArticle” συνάρτηση επιστρέφει μόνο ένα άρθρο. Το αναγνωριστικό του άρθρου ορίζεται στη διαδρομή του αιτήματος. Πρώτα η “getArticle” συνάρτηση ψάχνει το άρθρο στη συλλογή με την μέθοδο “articles.find_one”. Ως παράμετρος, περνάει το αναγνωριστικό του άρθρου σε μορφή αντικείμενου ObjectId [15]. Εάν το άρθρο δεν βρεθεί επιστρέφει ως απάντηση, ότι το άρθρο δεν βρέθηκε.

Στην συνέχεια με την “articles.update” μέθοδο παίρνει ως πρώτη παράμετρο το

αναγνωριστικό του άρθρου και ως δεύτερη ένα “query” με τα πεδία που θέλουμε να αλλάξουμε. Στην περίπτωση μας το κλειδί “$inc” [15] δηλώνει την αύξηση του πεδίου “views” κατά μια μονάδα . Στο τέλος η συνάρτηση επιστρέφει πίσω τα δεδομένα του άρθρου.

```
articles.update({'_id': ObjectId(articleID)},{"$inc":{"views":1}})
```

Το τρίτο Endpoint είναι το “createArticle”. Το “createArticle” Endpoint ακούει στη διεύθυνση “/articles” με την Http μέθοδο “POST.” Η “createArticle” συνάρτηση παρουσιάζεται παρακάτω:

```
@app.route("/articles", methods=["POST"])
def createArticle():
    params = request.get_Json()
    if params is None:
        return getResponse(400,message="Request body must be Json type")
    errFlag = False
    errMsg = ""
    if not "title" in params:
        errFlag = True
        errMsg += "Missing field title"
    if not "body" in params:
        errFlag = True
        errMsg += "Missing field body"
    if not "userID" in params:
        errFlag = True
        errMsg += "Missing userID field"
    if not "token" in params:
        errFlag = True
        errMsg += "Missing token field"
    if errFlag:
        return getResponse(400, message=errMsg)

    if not isValidToken(token=params["token"],userID=params["userID"]):
        return getResponse(403, message="Access Denied!!")

    r = requests.get(
        "http://{}/users/{}".format(users_host,params["userID"])
    )
    if r.status_code != 200:
        return getResponse(503, message="Users service not unvailable")
```

```
    newArticle = {
        "title": params["title"],
        "body": params["body"],
        "ownerID": params["userID"],
        "owner": r.Json()["username"],
        "views": 0
    }
    articleID = articles.insert_one(newArticle).inserted_id

    return getResponse(
        201,
        id=str(articleID),
        get="/articles/{}".format(str(articleID)),
    )
```

Η "createArticle" συνάρτηση δημιουργεί ένα καινούργιο άρθρο. Καταρχάς, η συνάρτηση "createArticle" ελέγχει εάν τα δεδομένα του άρθρου βρίσκονται σωστά στο σώμα του αιτήματος. Εάν δεν είναι σωστά, ενεργοποιείται η σημαία "errFlag" και επιστρέφονται πίσω στον χρήστη οι ανάλογες απαντήσεις με τα αντίστοιχα σφάλματα.

Στη συνέχεια με τη συνάρτηση "isValidToken" ελέγχεται η αυθεντικότητα του χρήστη. Εάν ο χρήστης, δεν έχει εγγραφεί στην εφαρμογή, επιστρέφεται μια απάντηση με κατάσταση 403, πράγμα που σημαίνει ,ότι ο χρήστης δεν έχει δικαίωμα πρόσβασης [12].

Έπειτα η "createArticle" συνάρτηση στέλνει μέσω της Requests στην Users μικρουπηρεσία, ένα "GET" Http αίτημα. Το αίτημα αυτό ζητάει το όνομα του χρήστη, που δημιούργησε το άρθρο. Το όνομα αποθηκεύεται στην Mongodb βάση δεδομένων για γρήγορη πρόσβαση (Caching). Εάν η μικρουπηρεσία Users δεν ανταποκρίνεται, τότε επιστρέφεται απάντηση με αριθμό κατάσταση 503, που σημαίνει ότι μια υπηρεσία δεν είναι διαθέσιμη [12].

Τέλος, τα δεδομένα του άρθρου αποθηκεύονται στην βάση δεδομένων, με τη μέθοδο "articles.insert_one". Το αναγνωριστικό του άρθρου επιστρέφεται πίσω στο χρήστη.

Το τέταρτο Endpoint είναι το “updateArticle”. Το “updateArticle” Endpoint ακούει στη διεύθυνση “/articles/<articleID>” με την Http μέθοδο “PUT”. Η “updateArticle” συνάρτηση παρουσιάζεται παρακάτω:

```
@app.route("/articles/<articleID>", methods=["PUT"])
def updateArticle(articleID):
   params = request.get_Json()
   if params is None:
      return getResponse(400,message="Request body must be Json type")
   updateObj = {}
   if "body" in params:
      updateObj["body"] = params["body"]
   if "title" in params:
      updateObj["title"] = params["title"]
   if not "token" in params:
      return getResponse(400, message="Missing token!")

   article = articles.find_one({'_id': ObjectId(articleID)})
   if article is None:
    return getResponse(
       404,
       message="Articles with id {} doesn't exist".format(articleID)
     )
   if not isValidToken(token=params["token"],userID=article["ownerID"]):
      return getResponse(403, message="Access Denied!!")

   articles.update({'_id': ObjectId(articleID)},{"$set":updateObj})
   return getResponse(
      200,
      get="/articles/{}".format(str(articleID)),
   )
```

Η “updateArticle” συνάρτηση αλλάζει τα δεδομένα ενός άρθρου. Το αναγνωριστικό του άρθρου δίνεται από τη διαδρομή του αιτήματος. Η συνάρτηση “updateArticle” ελέγχει τα δεδομένα του άρθρου στο σώμα του αιτήματος και την αυθεντικότητα του χρήστη. Ελέγχει, επίσης, εάν ο χρήστης είναι ο ιδιοκτήτης του άρθρου. Τέλος, αλλάζει το άρθρο με την μέθοδο “articles.update”.

Η συνάρτηση “isValidToken” στέλνει ένα Http αίτημα στην Users μικρουπηρεσία, με σκοπό την αυθεντικοποίηση του χρήστη, με βάση το Token που στέλνει. Το σώμα της “isValidToken” συνάρτησης παρουσιάζεται παρακάτω:

```
def isValidToken(userID, token):
    url = "http://{}/users/{}/isValidToken".format(users_host,userID)
    r = requests.post(url, Json = {'token':token})
    if r.status_code != 200:
        return False
    return r.Json()["isValid"]
```

Η “isValidToken” συνάρτηση δέχεται ως παραμέτρους το “userID” και το “token”. Έπειτα, στέλνει στο Endpoint “isValidToken” τις αντίστοιχες παραμέτρους, μέσω της μεθόδου “requests.post”. Στην περίπτωση που κάτι πάει λάθος, επιστρέφεται η λογική τιμή Ψευδής, αλλιώς επιστρέφεται η λογική τιμή του πεδίου “isValid” της απάντησης.

4.5 Comments Μικρουπηρεσία

Η Comments μικρουπηρεσία είναι όμοια με την Articles μικρουπηρεσία, αντί, όμως, για άρθρα η Comments μικρουπηρεσία διαχειρίζεται τα σχόλια που αφήνουν οι χρήστες σε ένα άρθρο. Ο κώδικας της Comments μικρουπηρεσίας βρίσκεται στο αρχείο “comments/commentsBackend.py”.

Οι βιβλιοθήκες που εισάγονται στη Comments μικρουπηρεσία είναι οι ίδιες με την Articles μικρουπηρεσία. Επίσης η Comments μικρουπηρεσία δέχεται τις ίδιες μεταβλητές περιβάλλοντος με την Articles μικρουπηρεσία.

Η Comments μικρουπηρεσία συνδέεται με την Mongodb βάση δεδομένων μέσω της MongoClient. Στη μεταβλητή Comments ορίζεται η συλλογή των σχολίων από την “comment” βάση δεδομένων, όπως φαίνεται στον κώδικα παρακάτω:

```
clientDB = MongoClient('mongodb://{}:27017'.format(db_host))
db = clientDB["comments"]
comments = db["comments-collection"]
```

Το πρώτο Endpoint της Comments μικρουπηρεσίας είναι το “listComments”. Το “listComments” Endpoint ακούει στη διεύθυνση “/articles/<articleID>/comments” με την Http μέθοδο “GET”. Η “listComments” συνάρτηση παρουσιάζεται παρακάτω:

```
@app.route("/articles/<articleID>/comments", methods=["GET"])
def listComments(articleID):
    commentList = comments.find({"articleID":ObjectId(articleID)})
    resultList = []
    for comment in commentList:
        resultList.append({
            "body": comment["body"],
            "get":"/articles/{}/comments/{}"
                     .format(articleID, str(comment["_id"])),
            "owner": comment["owner"],
            "ownerID": comment["ownerID"]
        })
    return getResponseList(200, resultList)
```

Η “listComments” συνάρτηση επιστρέφει όλα τα σχόλια, που ανήκουν στο άρθρο, με αναγνωριστικό “articleID”. Η παράμετρος “articleID” δηλώνεται από τη διαδρομή του αιτήματος. Η “listComments” συνάρτηση βρίσκει όλα τα σχόλια, με πεδίο “articleID”, διαμέσου της μεθόδου “comments.find”. Στη συνέχεια μετατρέπει τα αποτελέσματα σε μια λίστα και επιστρέφει τη λίστα πίσω στο χρήστη.

Το δεύτερο Endpoint είναι το “createComment”. Το “createComment” Endpoint ακούει στην διεύθυνση “/articles/<articleID>/comments” με την Http μέθοδο “POST”. Η “createComment” συνάρτηση παρουσιάζεται παρακάτω:

```
@app.route("/articles/<articleID>/comments", methods=["POST"])
def createComment(articleID):
    params = request.get_Json()
    if params is None:
        return getResponse(400,message="Request body must be Json type")
    errFlag = False
    errMsg = ""
    if not "body" in params:
        errFlag = True
        errMsg += "Missing field body"
    if not "token" in params:
        errFlag = True
```

```
        errMsg += "Missing field token"
    if not "userID" in params:
        errFlag = True
        errMsg += "Missing field userID"
    if errFlag:
        return getResponse(400, message=errMsg)

    if not isValidToken(token=params["token"],userID=params["userID"]):
        return getResponse(403, message="Access Denied!!")

    r = requests.get(
        "http://{}/users/{}".format(users_host,params["userID"])
    )
    if r.status_code != 200:
        return getResponse(503, message="Users service not unvailable")

    newComment = {
        "body": params["body"],
        "articleID": ObjectId(articleID),
        "ownerID": params["userID"],
        "owner":r.Json()["username"]
    }
    commentID = comments.insert_one(newComment).inserted_id

    return getResponse(
        201,
        get="/articles/{}/comments/{}".format(articleID, str(commentID)),
        id=str(commentID)
    )
```

Η “createComment” συνάρτηση δημιουργεί ένα καινούργιο σχόλιο στο άρθρο, με αναγνωριστικό “articleID”. Η παράμετρος “articleID” δίνεται από τη διαδρομή του αιτήματος. Στην αρχή, η συνάρτηση ελέγχει για τα δεδομένα του καινούργιου σχολίου, τα οποία παρέχονται από το χρήστη, στο σώμα του αιτήματος. Εάν τα δεδομένα είναι λάθος, τότε, επιστρέφονται οι ανάλογες απαντήσεις με τα αντίστοιχα σφάλματα. Έπειτα η “createComment” συνάρτηση ελέγχει την αυθεντικότητα του χρήστη. Στην συνέχεια η συνάρτηση αυτή ζητάει το όνομα του χρήστη από την Users μικρουπηρεσία . Το όνομα αποθηκεύεται στην Mongodb βάση δεδομένων για γρήγορη πρόσβαση (Caching). Τέλος, το σχόλιο αποθηκεύεται στη βάση δεδομένων με την μέθοδο “comments.insert_one” και το αναγνωριστικό του επιστρέφεται στο χρήστη.

Το τρίτο Endpoint της Comments μικρουπηρεσίας είναι το “updateComment”. Το

“updateComment” Endpoint ακούει με την Http μέθοδο “PUT” στη διεύθυνση “/articles/<articleID>/comments/<commentID>”.

Η “updateComment” συνάρτηση παρουσιάζεται παρακάτω:

```
@app.route("/articles/<articleID>/comments/<commentID>", methods=["PUT"])
def updateComment(articleID, commentID):
    params = request.get_Json()
    if params is None:
        return getResponse(
               400,
               message="Request body must be Json type"
        )
    if not "token" in params:
        return getResponse(400, message="Missing token!")

    comment = comments.find_one({'_id': ObjectId(commentID)})
    if comment is None:
      return getResponse(
          404,
          message="Comment id {} isn't found".format(commentID)
    )
    if not isValidToken(
        token=params["token"],
        userID=comment["ownerID"]
    ):
        return getResponse(403, message="Access Denied!!")

    updateObj = {}
    if "body" in params:
        updateObj["body"] = params["body"]
    comments.update({'_id': ObjectId(commentID)},{"$set":updateObj})
    return getResponse(
       200,
       get="/articles/{}/comments/{}".format(articleID, str(commentID))
    )
```

Η “updateComment” συνάρτηση αλλάζει το σχόλιο με αναγνωριστικό “commentID”. Το “commentID” δίνεται από τη διαδρομή του αιτήματος. Στην αρχή, η συνάρτηση ελέγχει την ορθότητα των παραμέτρων στο αίτημα και εάν οι παράμετροι είναι εσφαλμένες, τότε, επιστρέφονται οι ανάλογες απαντήσεις με τα αντίστοιχα σφάλματα.

Στη συνέχεια η συνάρτηση ψάχνει το σχόλιο στη βάση δεδομένων με το αναγνωριστικό του, μέσω της μεθόδου “comments.find_one”. Εάν το σχόλιο δεν υπάρχει, τότε επιστρέφει

την ανάλογη απάντηση στο χρήστη. Μετά η "updateComment" συνάρτηση ελέγχει την αυθεντικότητα του χρήστη. Τέλος, η συνάρτηση αλλάζει το πεδίο "body" στο σχόλιο, το αποθηκεύει στη βάση δεδομένων μέσω της μεθόδου "comments.update" και επιστρέφει το αντίστοιχο αναγνωριστικό πίσω στο χρήστη.

4.6 Getaway

Στο Dockerfile του Gateway δηλώνονται οι οδηγίες για την δημιουργία της εικόνας της Gateway υπηρεσίας. Σκοπός της είναι η δρομολόγηση των αιτημάτων στις άλλες υπηρεσίες. Αυτό γίνεται διαμέσου του Nginx, ο οποίος είναι ένας Load-Balancer δρομολογητής.

Με την οδηγία "FROM nginx:alpine" βάζουμε σαν αρχική εικόνα την Alpine έκδοση του Nginx, η οποία είναι η πιο ελαφριά έκδοσή του.

Μέσω της οδηγίας "RUN rm /etc/nginx/conf.d/*" διαγράφουμε όλες τις υπάρχουσες ρυθμίσεις του Nginx, με σκοπό να χρησιμοποιήσουμε μόνο τις ρυθμίσεις που δηλώνουμε εμείς.

Τέλος, με την εντολή "COPY proxy.conf /etc/nginx/conf.d/" αντιγράφουμε της δικές μας ρυθμίσεις στον κατάλογο του Nginx.

Στο αρχείο "proxy.conf" βρίσκονται οι οδηγίες για την δρομολόγηση των Http αιτημάτων στις υπηρεσίες. Στο μπλοκ του Server δηλώνονται ένας Http Server. Ο Server ακούει στην θύρα 80 που ορίζεται με την εντολή "listen 80;" Έπειτα ορίζεται ένας "resolver" με τιμή 127.0.0.11.

```
resolver 127.0.0.11 valid=30s;
```

Στην IP διεύθυνση "127.0.0.11" σε κάθε Container ακούει ένας DNS Server που κρατάει τα IP άλλων Containers, που βρίσκονται στο ίδιο εικονικό δίκτυο. Ο "resolver" έχει την παράμετρο "valid=30s", που σημαίνει ότι το Nginx θα ανανεώσει τις τιμές του DNS Server κάθε 30 δευτερόλεπτα σε περίπτωση που οι υπηρεσίες κλιμακωθούν [9].

Στη συνέχεια δηλώνονται με την "set", η Alias διεύθυνση των υπηρεσιών Users, Articles

και Comments στις μεταβλητές του Nginx.

```
set $app_upstream http://app:80;
set $users_upstream http://users:5000;
set $articles_upstream http://articles:5000;
set $comments_upstream http://comments:5000;
```

Τέλος, κάθε μπλοκ με όνομα “location”, δηλώνει τη δρομολόγηση των αιτημάτων στην αντίστοιχη υπηρεσία , με βάση την URL διαδρομή.

```
location / { proxy_pass $app_upstream;}
location /users { proxy_pass $users_upstream;}
location /login { proxy_pass $users_upstream;}
location /articles{proxy_pass $articles_upstream;}
location ~ \/articles\/[A-Za-z0-9]+\/comments{proxy_pass $comments_upstream;}
```

Url διαδρομές που ξεκινάνε με “/users” ή “/login” δρομολογούνται στην Users υπηρεσία. Αντίθετα, διαδρομές που αρχίσουν με “/articles” δρομολογούνται στην Articles υπηρεσία. Τέλος, για την Comments υπηρεσία, η διαδρομή ελέγχεται με μια Regex έκφραση, η οποία δέχεται τις διαδρομές με οποιοδήποτε αναγνωριστικό άρθρο, όπως για παράδειγμα “/articles/32f23f2s/comments. Τέλος, όλες οι υπόλοιπες δρομολογούνται στο App.

4.7 App

Το Dockerfile του app είναι παρόμοιο με το Dockerfile της Gateway υπηρεσίας. Η μόνη διαφορά είναι η τελευταία οδηγία “COPY dist/ /www/”, η οποία παίρνει την εκτελέσιμη έκδοση της εφαρμογής που βρίσκεται στον κατάλογο “dist” και την αντιγράφει στον κατάλογο “/www” της εικόνας. Η εκτελέσιμη έκδοση της εφαρμογής μπορεί να χτιστεί με

την εντολή του τερματικού “npm run build”. Ο “www” κατάλογος παρέχει πρόσβαση στα αρχεία από τους πελάτες του διαδικτύου. Οι ρυθμίσεις του Nginx βρίσκονται στο αρχείο “proxy.conf”.

```
Server{
    listen 80;
    root /www;
    location / {}
}
```

Ο Http Server ακούει στην θύρα 80 και το ριζικό σύστημα αρχείων του Server βρίσκεται στον κατάλογο “/www” μέσω της εντολής “root /www”.

Ο βασικός κώδικας της εφαρμογής βρίσκεται μέσα στο αρχείο “/src/main.js”, ενώ κάθε σελίδα της εφαρμογής βρίσκεται στα αρχεία του κατάλογου “/src/components” με προέκταση “.vue”. Στο “main.js” δηλώνεται ο τοπικός δρομολογητής της εφαρμογής.

Ο τοπικός δρομολογητής των Simple-Page Applications βασίζεται στη λειτουργία του Browser. Η λειτουργία αυτή δεν στέλνει αιτήματα στο Server ούτε ανανεώνει την σελίδα, όταν στην URL διαδρομή περιέχεται ο χαρακτήρας ‘#’.

Για παράδειγμα, σε μια δοκιμαστική ιστοσελίδα, εάν η URL διαδρομή αλλάξει από “localhost/app/page1” σε “localhost/app/page2”, ο Browser, θα ζητήσει από το Server τη δεύτερη σελίδα και θα ανανεώσει την υπάρχουσα σελίδα διαγράφοντας τα δεδομένα της εφαρμογής στο Browser. Σε αντίθεση, η αλλαγή του “localhost/app#page1” σε “localhost/app#page2” δεν θα ανανεώσει τη σελίδα του Browser.

Η JavaScript βιβλιοθήκη για την τοπική δρομολόγηση λέγεται Vue-Router. Μπορούμε να δημιουργήσουμε ένα καινούργιο αντικείμενο “router” μέσω της κλάσης VueRouter. Στις παραμέτρους του κατασκευαστή δίνεται μια λίστα με το όνομα “routes”, η οποία αποτελείται από εγγραφές. Κάθε εγγραφή δηλώνει την τοπική διαδρομή και το Component. Το Component θα φορτωθεί όταν η τοπική διαδρομή είναι ίδια με τη διαδρομή του πεδίου.

Η Vue εφαρμογή ξεκινά με τη δημιουργία ενός καινούργιου αντικειμένου στην κλάση

Vue. Στις παραμέτρους του κατασκευαστή δίνεται το αντικείμενο “router” και τα καθολικά δεδομένα στο πεδίο “data”. Τα καθολικά δεδομένα είναι διαθέσιμα σε κάθε Component μέσω της βιβλιοθήκης Vue-Global. Τα καθολικά δεδομένα της εφαρμογής είναι η URL διαδρομή του Rest API και τα στοιχεία του ενεργού χρήστη, δηλαδή το όνομα, το αναγνωριστικό και το Token. Η template παράμετρος κρατάει το βασικό Html, που θα εμφανιστεί σε κάθε σελίδα, όπως επικεφαλίδες και μενού.

Κάθε Component βρίσκεται στον Components κατάλογο. Τα αντικείμενα που βρίσκονται στην ετικέτα “template” δηλώνουν τα Html αντικείμενα, που θα ζωγραφισθούν στο χρήστη. Κάτω από την ετικέτα “script” βρίσκεται JavaScript κώδικας, που εκτελείται, όταν ξεκινά ο Component. Στο “script” εξάγεται ένα αντικείμενο, το οποίο κρατάει τις ρυθμίσεις του Component, δηλαδή, το όνομα, τα καθολικά δεδομένα, τα Components , τα τοπικά δεδομένα , τον κατασκευαστή του και τις μεθόδους του.

Η επικοινωνία με το Rest API γίνεται μέσω της βιβλιοθήκης Vue-Resource. Μπορούμε να στείλουμε ένα Http αίτημα μέσα στην μέθοδο, με το αντικείμενο “this.$http”, καλώντας τη μέθοδο “get”, “put” ή “post” που δηλώνει τον τύπο του αιτήματος. Η πρώτη παράμετρος των μεθόδων είναι η διαδρομή του Endpoint και οι υπόλοιπες διαδρομές δηλώνουν το σώμα του αιτήματος. Κατόπιν γίνεται pipe στη μέθοδο “then”, η οποία παίρνει σαν πρώτη παράμετρο, την ανώνυμη συνάρτηση, που θα εκτελεσθεί, όταν η απάντηση επιστρέψει με επιτυχία και σαν δεύτερη παράμετρο την ανώνυμη συνάρτηση, που θα εκτελεσθεί, όταν η απάντηση δεν επιστρέψει ή δεν είναι σωστή.

5 Επίλογος

5.1 Σύνοψη και συμπεράσματα

Οι μικρουπηρεσίες είναι μια καινούργια μεθοδολογία στην ανάπτυξη διαδικτυακών εφαρμογών. Χρειάζονται ακόμα αρκετά χρόνια για να φανεί η αποτελεσματικότητα τους, σε σχέση με τις παλιές μεθοδολογίες. Πολλά άτομα θεωρούν τις μικρουπηρεσίες ως συνέχεια ή ως τμήμα της SOA (Service Oriented Architecture). Στην πραγματικότητα οι μικρουπηρεσίες προσφέρουν περισσότερη ελευθερία στους προγραμματιστές από τη SOA, η οποία βασίζεται σε πανάκριβα επαγγελματικά εργαλεία. Η σημαντική ερώτηση, σχετικά με τις μικρουπηρεσίες, είναι το πως συγκρίνονται με τη μονολιθική, η οποία είναι η κυρίαρχη μεθοδολογία ανάπτυξης διαδικτυακών εφαρμογών.

Σε αντίθεση, με τις μονολιθικές, οι μικρουπηρεσίες μπορούν να αναπτυχθούν ανεξάρτητα από πολλές μικρές ομάδες ατόμων. Κάθε μικρουπηρεσία, μπορεί να αναπτύσσεται σ΄ ένα απομονωμένο περιβάλλον, ανεξάρτητα από το τελικό περιβάλλον παραγωγής, καθώς, στις ομάδες αυτές, μπορούν να εργάζονται μαζί άτομα με διαφορετικές δεξιότητες και γνώσεις. Για παράδειγμα, οι μικτές ομάδες μπορούν να αποτελούνται από προγραμματιστές, διαχειριστές βάσεων δεδομένων, σχεδιαστές γραφικού περιβάλλοντος και δοκιμαστές. Επιπλέον, τα άτομα της ομάδας, μπορούν να σχεδιάσουν την μικρουπηρεσία, όπως οι ίδιοι επιθυμούν, χωρίς να κατευθύνονται από ένα σχεδιαστικό τμήμα. Κάθε μικτή ομάδα είναι ανεξάρτητη από τις άλλες ομάδες, που δουλεύουν στην εφαρμογή, και έτσι μειώνεται ο αριθμός των συγκρούσεων μεταξύ των ομάδων και η διαμάχη για την κατανομή των πόρων. Όλες οι ομάδες μπορούν να αναπτύξουν τις μικρουπηρεσίες τους αυτόματα και να δουλεύουν παράλληλα, μειώνοντας έτσι το χρόνο ανάπτυξης της εφαρμογής. Η επικοινωνία μεταξύ των ατόμων είναι καλύτερη καθώς τα άτομα γνωρίζονται προσωπικά μεταξύ τους και δεν χρειάζεται να επικοινωνούν μέσω μεγάλων τμημάτων. Επιπλέον, οι περισσότερες μικρουπηρεσίες , συντηρούνται από την ομάδα ανάπτυξης και τα άτομα αυτής έρχονται σε επαφή με τους πραγματικούς πελάτες της μικρουπηρεσίας, παίρνοντας πολύτιμο feedback. Βελτιώνεται έτσι η επίδοση και η

ποιότητα της μικρουπηρεσίας [1].

Οι μικρουπηρεσίες αναπτύσσονται σε απομονωμένα περιβάλλοντα και αυτό σημαίνει, ότι, οι μικτές ομάδες ανάπτυξης, μπορούν, να χρησιμοποιούν οτιδήποτε εργαλεία, γλώσσες προγραμματισμού και βάσεις δεδομένων, επιθυμούν, σε αντίθεση με τις μονολιθικές που περιορίζονται σε συγκεκριμένες γλώσσες. Έτσι, επιτρέπεται στην ομάδα ανάπτυξης να κάνει χρήση εργαλείων ειδικά σχεδιασμένων για μια συγκεκριμένη εργασία ή άλλα εργαλεία που είναι πιο γνωστά στα άτομα της ομάδας. Τα απομονωμένα περιβάλλοντα των μικρουπηρεσιών βοηθούν στην εγκατάσταση εργαλείων τρίτων (plugins, add-on, βιβλιοθήκες) ή τη χρήση Containers από τρίτους για την παροχή υπηρεσιών (Elastic Search). Επιπλέον, τα απομονωμένα περιβάλλοντα διασφαλίζουν το σύστημα από τυχόν λάθη πρωτάρηδων προγραμματιστών ή επιτήδειων, που αποκτούν, πρόσβαση στο Back-End της μικρουπηρεσίας. Καθώς κάθε μικρουπηρεσία έχει τη δική της Back-End, η ζημία θα είναι περιορισμένη. Ένα συνεχές πρόβλημα στη ανάπτυξη των εφαρμογών είναι ότι τα διάφορα εργαλεία χρειάζονται διαφορετικές εκδόσεις από γλώσσες προγραμματισμού για να λειτουργήσουν. Για παράδειγμα , πολλές βιβλιοθήκες χρειάζονται την Python έκδοση 2.7, η τρέχουσα έκδοση της Python, όμως, είναι η 3.6. Πριν τα Containers οι διαχειριστές των συστημάτων έπρεπε να εγκαταστήσουν διαφορετικές εκδόσεις για να ελέγχουν τις συγκρούσεις μεταξύ τους. Με τα Containers όμως οι προγραμματιστές μπορούν να χρησιμοποιήσουν όποια έκδοση αυτοί επιθυμούν. Επιπλέον, κάθε μικρουπηρεσία μπορεί να πακεταριστεί σε ένα Container. Αυτό μπορεί να πολλαπλασιαστεί μέσα στο Docker cluster ώστε να εξυπηρετήσει περισσότερους πελάτες [1].

Ένα πολύ σημαντικό κομμάτι των μικρουπηρεσιών είναι το Rest API διάμεσου του Gateway. Κάτι που πολλές μονολιθικές εφαρμογές έχουν αρχίσει ήδη να χρησιμοποιούν. Καθώς, όλες οι μέθοδοι, για την επικοινωνία με την εφαρμογή βρίσκονται σε ένα Server, όλοι οι πελάτες μπορούν να επικοινωνήσουν με τον Server ανεξάρτητα με την πλατφόρμα που βρίσκονται. Επιπλέον λόγω του Rest API πολλοί διαφορετικοί χρήστες μπορούν να επικοινωνήσουν με την εφαρμογή είτε είναι φυσικά πρόσωπα είτε είναι προγράμματα σε υπολογιστή. Επίσης, το Rest API παρέχει την δυνατότητα επαναχρησιμοποίησης των υπηρεσιών, σε διαφορετικές υπηρεσίες, επιτρέποντας έτσι την κοινή είσοδο των χρηστών.

Για παράδειγμα, μια εταιρεία που έχει διαφορετικές εφαρμογές μπορεί να κάνει αυθεντικοποίηση του χρήστη σε μια κοινή πύλη και να χρησιμοποιήσει ένα token για την αυθεντικοποίησή του στις άλλες εφαρμογές όπως το OAuth2 πρωτόκολλο. Επειδή το Rest API μπορεί να επικοινωνήσει με άλλες εφαρμογές, μπορεί να χρησιμοποιηθεί ως μία υπηρεσία επί πληρωμή από τρίτους πελάτες. Η ιδέα αυτή λέγεται λογισμικό ως υπηρεσία (Software as a Service) ή SaaS. Για παράδειγμα, μια εταιρεία, που παρέχει λογαριασμούς email σε πελάτες, μπορεί να έχει μια δωρεάν ιστοσελίδα, ώστε να μπορούν να βλέπουν οι πελάτες τα Emails τους και να πουλάει πακέτα σε τρίτες εταιρείες με την παροχή μαζικών λογαριασμών στο όνομα της εταιρείας. Αυτό επιτυγχάνεται εύκολα καθώς το Rest API μπορεί να υλοποιηθεί και να ενσωματωθεί σε οποιοδήποτε σύστημα έχει ο πελάτης.

Συνοψίζοντας, είναι φανερό ότι οι μικρουπηρεσίες παρέχουν πολλά πλεονεκτήματα σε αντίθεση με τις μονολιθικές , ιδιαίτερα σε εταιρείες με μεγάλο αριθμό προγραμματιστών. Οι μονολιθικές είναι αποτελεσματικές για μικρότερες εταιρείες, που παρέχουν λιγότερες λειτουργίες στους πελάτες.

Το μειονέκτημα των μικρουπηρεσιών είναι η δυσκολία επικοινωνίας μεταξύ των υπηρεσιών.

5.2 Μελλοντικές Επεκτάσεις.

Σ΄ όλη την εφαρμογή παρουσιάζεται μειωμένη ασφάλεια, με αποτέλεσμα την εύκολη πρόσβαση από τρίτα άτομα με σκοπό τον δόλο και συγκεκριμένα την μερική ή ολική διαγραφή στις βάσεις δεδομένων. Η λύση του προβλήματος αυτού είναι η εγκατάσταση του OAuth 2 και Https στην εφαρμογή.

Όταν μια υπηρεσία καταρρέει δεν υπάρχει έγκαιρη προειδοποίηση, με αποτέλεσμα να μην μπορούν να λειτουργήσουν οι υπηρεσίες που στηρίζονται σ΄ αυτήν. Υλοποιώντας τα Circuit Breaker Pattern στο κώδικα της εφαρμογής λύνεται το παραπάνω πρόβλημα.

Δεν υπάρχει έλεγχος για τη σωστή λειτουργεία της εφαρμογής, σε περίπτωση λογικών σφαλμάτων στο κώδικα από άλλους προγραμματιστές. Αυτό λύνεται με τη δημιουργία του κώδικα δοκιμών (Unit Testing). Έτσι, μπορούμε να ελέγξουμε τη σωστή λειτουργία της εφαρμογής.

Εφαρμόζοντας διαφορετικές παραμέτρους στο Docker-Compose βρίσκουμε τη βέλτιστη ταχύτητα εκτέλεσης των μικρουπηρεσιών της εφαρμογής.

Μια μελλοντική επέκταση της εφαρμογής είναι η καταγραφή των δεδομένων για τις σελίδες του κάθε άρθρου, με σκοπό τη συλλογή και την ανάλυση στατιστικών δεδομένων.

Μελλοντικοί προγραμματιστές έχοντας σαν βάση την εφαρμογή αυτή, μπορούν να χτίσουν νέες εφαρμογές ή να επεκτείνουν την παρούσα, προσθέτοντας καινούργιες μικρουπηρεσίες στο "docker-compose.yml" αρχείο και στο API Gateway.

5.3 Βιβλιογραφία

1. James Lewis and Martin Fowler, 2014. Microservices a definition of this new architectural term. [online] Available at: <https://martinfowler.com/articles/microservices.html> [Accessed August 2017].

2. Docker, Inc., 2017. Docker overview [online] Available at: <https://docs.Docker.com/engine/Docker-overview/#Docker-engine> [Accessed August 2017].

3. Docker, Inc., 2017. Best practices for writing Dockerfiles. [online] Available at: <https://docs.Docker.com/engine/userguide/eng-image/dockerfile_best-practices//> [Accessed August 2017].

4. Docker, Inc., 2017. Swarm mode overview. [online] Available at: <https://docs.Docker.com/engine/swarm/> [Accessed August 2017].

5. Docker, Inc., 2017. Create a swarm. [online] Available at: <https://docs.Docker.com/engine/swarm/swarm-tutorial/create-swarm/> [Accessed August 2017].

6. Docker, Inc., 2017. Add nodes to the swarm. [online] Available at: <https://docs.Docker.com/engine/swarm/swarm-tutorial/add-nodes/> [Accessed August 2017].

7. Docker, Inc., 2017. Get Started, Part 5: Stacks. [online] Available at: <https://docs.Docker.com/get-started/part5/#introduction> [Accessed August 2017].

8. Docker, Inc., 2017. Compose file version 3 reference. [online] Available at: <https://docs.Docker.com/compose/compose-file/> [Accessed August 2017].

9. Quentin Stafford-Fraser, 2016. Nginx and Docker [YouTube Video] Available at:

<https://www.youtube.com/watch?v=HJ9bECmuwKo> [Accessed August 2017].

10. Okonkwo Vincent Ikem, 2017. How We Solved Authentication and Authorization in Our Microservice Architecture [online] Available at: <https://medium.com/technology-learning/how-we-solved-authentication-and-authorization-in-our-microservice-architecture-994539d1b6e6> [Accessed August 2017].

11. Chris Richardson, 2017. Pattern: Microservice Architecture[online] Available at: <http://microservices.io/patterns/microservices.html > [Accessed August 2017].

12. Mozilla, 2017. Http response status codes [online] Available at: <https://developer.mozilla.org/en-US/docs/Web/Http/Status > [Accessed August 2017].

13. Chris Richardson, 2015, Building Microservices: Using an API Gateway [online] Available at: <https://www.Nginx.com/blog/building-microservices-using-an-api-gateway/ > [Accessed August 2017].

14. Chris Richardson, 2015, Building Microservices: Inter-Process Communication in a Microservices Architecture [online] Available at: <https://www.Nginx.com/blog/building-microservices-inter-process-communication/ > [Accessed August 2017].

15. MongoDB Inc., 2017, PyMongo Tutorial [online] Available at: <http://api.mongodb.com/Python/current/tutorial.html/ > [Accessed August 2017].

16. Pony ORM, LLC., 2017, PONY ORM API Reference [online] Available at: <https://docs.ponyorm.com/api_reference.html> [Accessed August 2017].

17. Pony ORM, LLC., 2017, Getting Started with Pony [online] Available at: <https://docs.ponyorm.com/firststeps.html> [Accessed August 2017].

18. Alex Rodriguez, 2015. RESTful Web services: The basics[online] Available at: <https://www.ibm.com/developerworks/library/ws-restful/index.html > [Accessed August 2017].

19. Tony Mauro of NGINX, Inc., 2015, Adopting Microservices at Netflix: Lessons for Architectural Design [online] Available at: <https://www.Nginx.com/blog/microservices-at-netflix-architectural-best-practices/> [Accessed August 2017].

20. Armin Ronacher , 2017, Flask API [online] Available at: <http://flask.pocoo.org/docs/0.12/api/> [Accessed August 2017].

5.4 Παράρτημα

Πηγαίος Κώδικας: https://github.com/chgivan/MicroArticles