ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ – ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Πρόγραμμα Μεταπτυχιακών Σπουδών

«Προηγμένα Συστήματα Πληροφορικής – Ανάπτυξη Λογισμικού και Τεχνητής Νοημοσύνης»

Μεταπτυχιακή Διατριβή

Τίτλος Διατριβής	**Ανάπτυξη εφαρμογής που ενσωματώνει τεχνολογία Blockchain με χρήση Reactive Microservices** **Application Development incorporating Blockchain technology using Reactive Microservices**
Ονοματεπώνυμο Φοιτητή	**Ηλίας Παπανικολάου**
Πατρώνυμο	**Παναγιώτης**
Αριθμός Μητρώου	**ΜΠΣΠ 20039**
Επιβλέπων	**Αλέπης Ευθύμιος, Αναπληρωτής Καθηγητής**

Ημερομηνία Παράδοσης **ΙΟΥΛΙΟΣ 2022**

Τριμελής Εξεταστική Επιτροπή

Αλέπης Ευθύμιος Αναπληρωτής Καθηγητής	Βίρβου Μαρία Καθηγήτρια	Κωνσταντίνος Πατσάκης Αναπληρωτής Καθηγητής

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή κ. Ευθύμιο Αλέπη, καθώς και τον υποψήφιο διδάκτορα κ. Σπυρίδων Παπαδημητρίου για τις πολύτιμες συμβουλές και την καθοδήγηση τους.
Επίσης, θα ήθελα να ευχαριστήσω θερμά τη συνάδελφο μου Ιωάννα Δαβλιάνη, με την οποία συνεργάστηκα για την ανάπτυξη της εφαρμογής που παρουσιάζεται στην μεταπτυχιακή διατριβή.

Περίληψη

Η παρούσα μεταπτυχιακή διατριβή αναπτύχθηκε στο πλαίσιο του Προγράμματος Μεταπτυχιακών Σπουδών “Προηγμένα Συστήματα Πληροφορικής - Ανάπτυξη Λογισμικού και Τεχνητής Νοημοσύνης” του Πανεπιστημίου Πειραιώς και έχει ως στόχο την υλοποίηση web εφαρμογής, που ενσωματώνει τεχνολογία Blockchain, σε αρχιτεκτονική μικροϋπηρεσιών με χρήση reactive programming.

Η τεχνική του reactive programming αφορά στη non-blocking επικοινωνία μεταξύ των στοιχείων που αποτελούν την εφαρμογή, με αποτέλεσμα την κατά πολύ καλύτερη αξιοποίηση των πόρων υλικού των συστημάτων που φιλοξενούν τις μικροϋπηρεσίες. Το Java framework που χρησιμοποιήθηκε για την δημιουργία των μικροϋπηρεσιών είναι το Spring Boot 2.0, καθώς διαθέτει ενσωματωμένη υποστήριξη στην ανάπτυξη εφαρμογών με reactive programming, εισάγοντας ένα νέο web framework, το Spring WebFlux.

Η εφαρμογή αφορά στη διασφάλιση της γνησιότητας εγγράφων ενός εκπαιδευτικού ιδρύματος, αξιοποιώντας την τεχνολογία blockchain. Ο σχεδιασμός και η υλοποίηση της εφαρμογής έχει βασιστεί σε βέλτιστες πρακτικές και μοτίβα σχεδίασης μικροϋπηρεσιών, όπως η χρήση ασύγχρονων μηνυμάτων μέσω του RabbitMQ message broker, εξισορρόπηση φόρτου, service discovery, api gateway, distributed tracing και token relay.

Ένα από τα ζητήματα που έχει δοθεί ιδιαίτερη έμφαση, καθώς αποτελεί πρόκληση στις μικροϋπηρεσίες, είναι η ασφάλεια και η εξουσιοδότηση χρηστών που αφορά στη πρόσβαση, τόσο στους διακομιστές πόρων μέσω των διεπαφών (APIs), όσο και στις τοποθεσίες της frontend εφαρμογής, με χρήση των προτύπων OAuth2.0 και OpenID Connect.

Για την ανάπτυξη (deployment) της εφαρμογής, χρησιμοποιήθηκε η τεχνολογία Docker. Οι μικροϋπηρεσίες τοποθετούνται σε docker images και στη συνέχεια μεταφορτώνονται στο αποθετήριο DockerHub. Με τη δημιουργία κατάλληλου αρχείου docker-compose.yml, η εφαρμογή αναπτύσσεται μέσα σε λίγα λεπτά.

Abstract

This master's thesis was developed in the scope of the MSc Program "Advanced Informatics and Computing Systems - Software Development and Artificial Intelligence" of the University of Piraeus and aims to implement a web application development, incorporating Blockchain technology, in microservices architecture using reactive programming.

The reactive programming technique refers to the non-blocking communication between the components of the application, resulting in a much better utilization of the hardware resources of the systems hosting the microservices. The Java framework used to create the microservices is Spring Boot 2.0, as it has built-in support for developing applications with reactive programming, introducing a new web framework, the Spring WebFlux.

The application is about ensuring the authenticity of documents of an educational institution, leveraging blockchain technology. The design and implementation of the application is based on best practices and microservices design patterns, such as the use of asynchronous messaging via the RabbitMQ message broker, load balancing, service discovery, api gateway, distributed tracing, and token relay.

One of the topics that has been given special emphasis, as it consists of a challenge in microservices, is security and user authorization regarding access, both to the resource servers through the interfaces (APIs), and to the frontend application sites, using the OAuth2.0 and OpenID Connect standards.

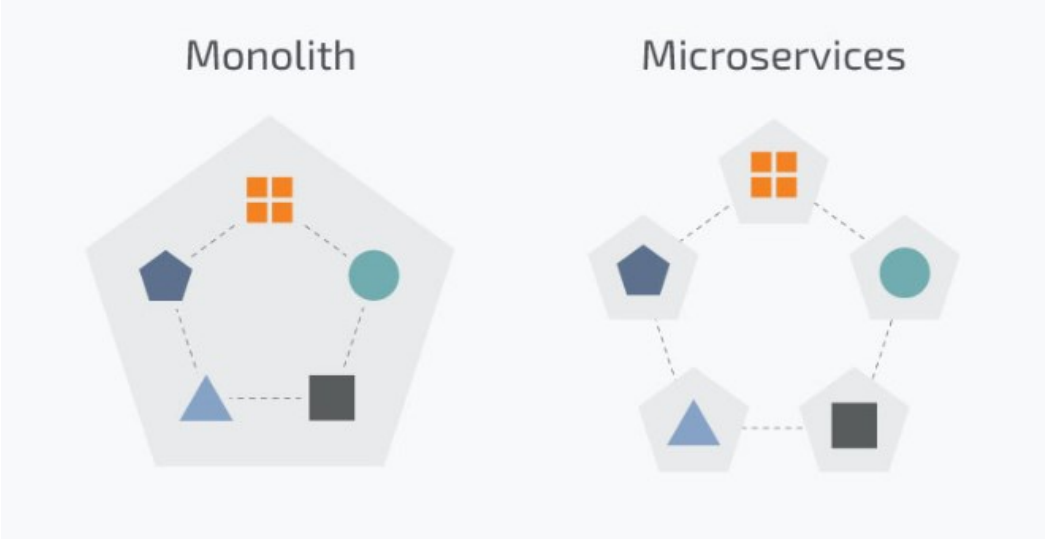
For the development of the application, Docker technology was used. Microservices are put into docker images and then uploaded to the DockerHub repository. By creating a suitable docker-compose.yml file, the application is deployed in minutes.

Πίνακας Περιεχομένων

Εισαγωγή στα Microservices

Οι μικροϋπηρεσίες αποτελούν μια επιταχυνόμενη τάση στις μέρες μας που προσφέρει απτά οφέλη στην ανάπτυξη συστημάτων, συμπεριλαμβανομένης της αύξησης της επεκτασιμότητας, της ευελιξίας και άλλων σημαντικών πλεονεκτημάτων. Η Google, η Amazon, το Netflix και άλλοι τεχνολογικοί ηγέτες έχουν πραγματοποιήσει επιτυχημένη μετάβαση από τη μονολιθική αρχιτεκτονική, στις μικροϋπηρεσίες. Πολλές εταιρίες ακολουθούν το ίδιο παράδειγμα ως πιο αποδοτικό τρόπο για την ανάπτυξη τους.

Αντίθετα, η τάση της μονολιθικής[1] προσέγγισης, η οποία αποτελεί το παραδοσιακό μοντέλο ανάπτυξης εφαρμογών, αρχίζει να μειώνεται διότι η δημιουργία μιας μονολιθικής εφαρμογής θέτει μια σειρά από προκλήσεις που σχετίζονται με τη διαχείριση της τεράστιας βάσης κώδικα (code base), τη δυσκολία να υιοθετηθούν νέες τεχνολογίες στον υφιστάμενο κώδικα, τη δυσκολία στην υλοποίηση νέων αλλαγών, καθώς και την περιορισμένη δυνατότητα ανάθεσης επιπλέον πόρων με δυναμικό τρόπο, όταν αυτό απαιτείται λόγω φόρτου εργασίας (lack of scalability and elasticity).



Σχήμα 1: Monolithic – Microservices

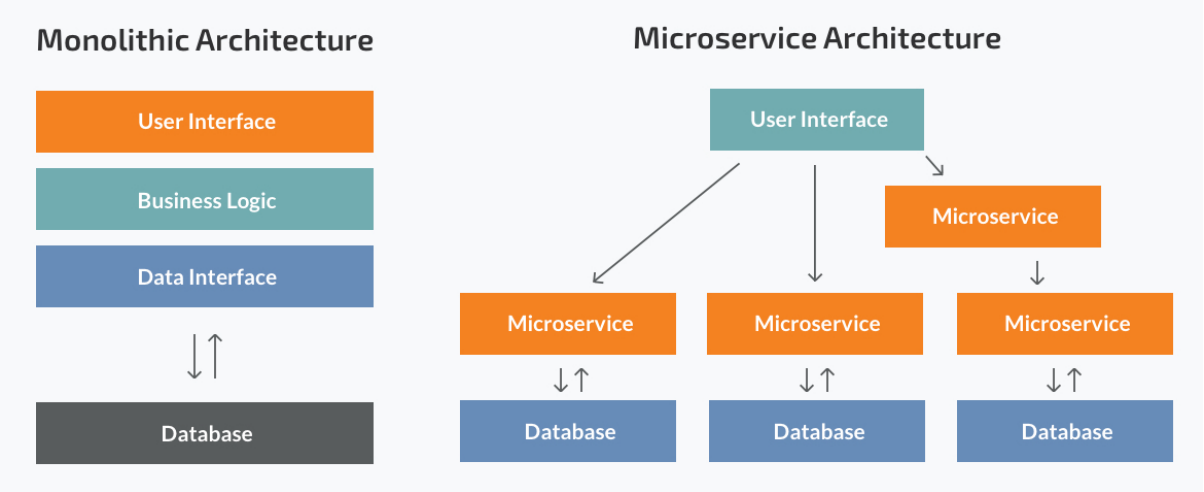
Η αρχιτεκτονική των μικροϋπηρεσιών αναφέρεται στο μοντέλο που δομεί μια εφαρμογή ως μια συλλογή από επιμέρους ανεξάρτητες υπηρεσίες που διακρίνονται από χαρακτηριστικά όπως:

- Υψηλή δυνατότητα συντήρησης και ελέγχου (highly maintainable and testable).
- Συνεχής ανάπτυξη (continuous deployment).
- Χαμηλή εξάρτηση (loosely coupled).
- Ανάπτυξη από μικρές ομάδες προγραμματιστών.
- Οργάνωση της κάθε υπηρεσίας σε μια συγκεκριμένη επιχειρησιακή απαίτηση (single responsibility). Η αρχιτεκτονική των μικροϋπηρεσιών επιτρέπει την αξιόπιστη, γρήγορη και συχνή παράδοση εφαρμογών μεγάλης κλίμακας, παρέχοντας τη δυνατότητα επέκτασης για την κάλυψη νέων απαιτήσεων.

[1] Μονολιθική αρχιτεκτονική ονομάζεται ο παραδοσιακός τρόπος δημιουργίας μιας εφαρμογής. Η εφαρμογή δημιουργείται ως μια μονάδα που ενσωματώνει το σύνολο των λειτουργιών που εξυπηρετούν τις περιπτώσεις χρήσης.

Κάθε μικροϋπηρεσία θα πρέπει να πληροί κάποιες προϋποθέσεις:

- Να αποτελεί ένα αυτόνομο πακέτο λογισμικού, ανεξάρτητα αναβαθμίσιμο, επεκτάσιμο και αντικαταστάσιμο από μια νέα υπηρεσία που εξυπηρετεί τον ίδιο σκοπό.
- Να διαθέτει ανεξάρτητη βάση δεδομένων, καθώς οι μικροϋπηρεσίες δε συνιστάται να μοιράζονται την ίδια βάση δεδομένων μεταξύ τους.
- Να επικοινωνεί μόνο μέσω καλά ορισμένων διεπαφών, είτε χρησιμοποιώντας API[2], είτε κατά προτίμηση στέλνοντας ασύγχρονα μηνύματα. Η δομή του API ή του ασύγχρονου μηνύματος θα πρέπει να είναι καλά ορισμένη και τεκμηριωμένη, ώστε να διευκολύνει τους προγραμματιστές που θέλουν να αλληλεπιδράσουν με την υπηρεσία.
- Να δημιουργείται σε ξεχωριστό περιβάλλον εκτέλεσης, όπως το Docker Container [3], έτσι ώστε να μπορεί να εκτελεστεί ως αυτόνομη διεργασία. Η χρήση του Docker Container δίνει τη δυνατότητα στις υπηρεσίες να αναπτυχθούν πάνω από μια φορά (instance scaling) σε περίπτωση αυξημένου φόρτου εργασίας.
- Να είναι stateless[4] ώστε τα εισερχόμενα αιτήματα να μπορούν να εξυπηρετηθούν από οποιοδήποτε instance του microservice.



Σχήμα 2: Monolithic - Microservices

Ένα σύνολο μικροϋπηρεσιών που αποτελούν μια εφαρμογή, μπορεί να αναπτυχθεί σε πολλούς, μικρότερης ισχύος διακομιστές (servers), αντί για ένα μεγαλύτερης ισχύος, όπως θα ήμασταν αναγκασμένοι να χρησιμοποιήσουμε στην περίπτωση μιας μονολιθικής εφαρμογής.

Αν πληρούνται οι προαναφερθείσες προϋποθέσεις, τότε πολλά στιγμιότυπα (instances) μιας μικροϋπηρεσίας μπορούν να τεθούν ταυτόχρονα σε λειτουργία (instance scale up), κάνοντας χρήση των δυνατοτήτων autoscaling που μας παρέχουν οι υπηρεσίες cloud[5].

[2] Application Programming Interface ή διεπαφή προγραμματισμού εφαρμογών, χρησιμοποιείται για τη μεταβίβαση δεδομένων μεταξύ εφαρμογών λογισμικού με έναν τυποποιημένο τρόπο. Πολλές υπηρεσίες προσφέρουν δημόσια API που επιτρέπουν σε οποιονδήποτε να στέλνει και να λαμβάνει περιεχόμενο από την υπηρεσία. Τα API που λειτουργούν μέσω Διαδικτύου χρησιμοποιώντας URL http:// αναφέρονται ως API Ιστού.

[3] Docker Container είναι ένα πακέτο λογισμικού που ενσωματώνει τον κώδικα και όλες τις εξαρτήσεις του, ώστε η εφαρμογή να εκτελείται γρήγορα και αξιόπιστα σε διαφορετικά υπολογιστικά περιβάλλοντα. https://www.docker.com/

[4] Stateless: Ο διακομιστής (server) δε διατηρεί στοιχεία του πελάτη (client) που έχει συνδεθεί με session, αντιθέτως τα στοιχεία του πελάτη περνούν σε κάθε αίτημα (request) προς τον διακομιστή (server).

Δυσκολίες που προκύπτουν με τη χρήση μικροϋπηρεσιών.

Στην αρχιτεκτονική μικροϋπηρεσιών, εκτός από τα οφέλη που προσφέρονται, προκύπτουν και δυσκολίες όπως:

- Ως κατανεμημένο σύστημα, οι μικροϋπηρεσίες αποτελούνται από πολλαπλά στοιχεία όπως διαφορετικές βάσεις δεδομένων και ανεξάρτητες υπηρεσίες, που καθιστούν τη μεταξύ τους συνδεσιμότητα αρκετά πολύπλοκη. Τα κατανεμημένα συστήματα έχουν υψηλό βαθμό δυσκολίας στην ανάπτυξη και διαχείριση τους.
- Η σύγχρονη επικοινωνία (synchronous communication) μεταξύ πολλών υποσυστημάτων μπορεί να προκαλέσει αλυσιδωτή αποτυχία, ειδικά όταν υπάρχει υψηλό φόρτο εργασίας.
- Η έννοια των ACID transactions στις βάσεις δεδομένων γίνεται αρκετά πολύπλοκο ζήτημα στις μικροϋπηρεσίες και απαιτεί υψηλής ειδίκευσης προγραμματιστές για να το διαχειριστούν.
- Η ενημέρωση των παραμέτρων (configuration), όταν το πλήθος των υποσυστημάτων αυξάνεται, γίνεται δυσκολότερη.
- Σε περίπτωση διερεύνησης σφαλμάτων, η παρακολούθηση της διαδρομής ενός αιτήματος (request tracing) όπου εμπλέκονται πολλά microservices γίνεται ιδιαίτερα πολύπλοκη.
- Η ανάλυση χρήσης πόρων συστήματος σε επίπεδο μικροϋπηρεσιών είναι δυσκολότερη.

Η σχεδίαση και υλοποίηση των μικροϋπηρεσιών θα πρέπει εξαρχής να προβλέπει και να αντιμετωπίζει όλες τις παραπάνω δυσκολίες, διότι όσο αυξάνεται ο αριθμός των υπηρεσιών, τόσο μεγαλύτερη είναι και η πιθανότητα προβλημάτων.

Design Patterns για Microservices

Τα σχεδιαστικά μοτίβα είναι η περιγραφή δοκιμασμένων λύσεων που έχουν δοθεί ώστε να αντιμετωπιστούν συγκεκριμένα προβλήματα, όπως οι παραπάνω δυσκολίες που παρατηρούνται στις μικροϋπηρεσίες. Μερικά από τα σχεδιαστικά μοτίβα που αφορούν την ορθή υλοποίηση των μικροϋπηρεσιών είναι τα εξής:

Service discovery pattern: Η ανάθεση των IP διευθύνσεων και των θυρών στα instances των microservices, πραγματοποιείται δυναμικά κατά την εκκίνηση τους. Οι υπολογιστές πελάτες (clients) δε είναι σε θέσει να γνωρίζουν τη συγκεκριμένη IP και θύρα ώστε να αποστείλουν κάποιο αίτημα (request). Τη λύση στο συγκεκριμένο πρόβλημα δίνει η χρήση του service discovery ή naming server service το οποίο καταγράφει τα διαθέσιμα instances και τις IP διευθύνσεις των υπόλοιπων μικροϋπηρεσιών του συστήματος. (Spring Cloud Eureka Server).

Edge Server pattern: Σε ένα οικοσύστημα μικροϋπηρεσιών, πολλές φορές είναι επιθυμητό να επιτρέψουμε σε ορισμένες υπηρεσίες να είναι ορατές και εκτός του εσωτερικού δικτύου ενώ άλλες υπηρεσίες θα πρέπει να αποκρύπτονται από εξωτερική πρόσβαση. Οι εκτεθειμένες υπηρεσίες θα πρέπει να προστατεύονται από αιτήματα κακόβουλων χρηστών. Τη λύση στο συγκεκριμένο ζήτημα, δίνει ο Edge Server, πρόκειται για ένα reverse proxy και μπορεί εύκολα να ενσωματωθεί στο Service discovery microservice και να παρέχει δυνατότητα δυναμικού load-balancing[6]. (Spring Cloud Gateway).

---

[5] Cloud: Κατ' απαίτηση διαθεσιμότητα πόρων υπολογιστικών συστημάτων, όπως μέσα αποθήκευσης δεδομένων και υπολογιστική ισχύ χωρίς άμεση ενεργή διαχείριση από τον χρήστη. Το cloud computing βασίζεται στην κοινή χρήση πόρων και χρησιμοποιεί το μοντέλο χρέωσης pay-as-you-go.

[6] Load-balancing: Η εξισορρόπηση φόρτου είναι ένας όρος, ο οποίος περιγράφει την κατανομή μίας λειτουργίας που χρειάζεται να εκτελείται απρόσκοπτα μεταξύ ενός αριθμού διακομιστών με σκοπό τη μείωση του φόρτου εργασίας. Αυτό γίνεται για να επιτευχθεί συνεχής και απρόσκοπτη λειτουργία ενός συστήματος.

Reactive microservice pattern: Ο συνήθης τρόπος που υλοποιείται η επικοινωνία μεταξύ δυο συστημάτων, είναι μέσω σύγχρονης blocking επικοινωνίας (synchronous communication using blocking I/O), για παράδειγμα μια κλήση σε RESTful JSON API μέσω HTTP. Ο όρος blocking I/O σημαίνει πως το λειτουργικό σύστημα δεσμεύει ένα νήμα (thread) για κάθε request το οποίο παραμένει σε αναμονή μέχρι να φθάσει κάποιο response ή να συμβεί timeout μετά από ορισμένο χρονικό διάστημα. Αν το πλήθος των ταυτόχρονων request αυξηθεί, τότε ο διακομιστής μπορεί να οδηγηθεί σε εκτεταμένο χρόνο απόκρισης ή ακόμα και σε κατάρρευση. Στη περίπτωση της αρχιτεκτονικής των μικροϋπηρεσιών, το πρόβλημα γίνεται ακόμη πιο έντονο, καθώς συμμετέχει μια σειρά από μικροϋπηρεσίες για να εξυπηρετηθεί ένα request. Όσο μεγαλύτερο είναι το πλήθος των μικροϋπηρεσιών που συμμετέχουν σε ένα request, τόσο πιο γρήγορα εξαντλούνται οι πόροι του διακομιστή. Η λύση βρίσκεται στη χρήση τεχνολογίας non-blocking I/O, όπου τα νήματα (threads) δεν δεσμεύονται καθόλη τη διάρκεια ενός request, είτε πρόκειται για μια κλήση σε κάποιο service, είτε για κλήση σε βάση δεδομένων. Στη συνέχεια της εργασίας θα πραγματοποιηθεί εκτενής αναφορά στο reactive programming. (Spring Webflux).

Central configuration pattern: Μια εφαρμογή εγκαθίσταται μαζί με αρχεία ιδιοτήτων (properties files) που περιέχουν μεταβλητές συστήματος (environment variables) και ρυθμίσεις παραμέτρων. Σε ένα σύστημα που αποτελείται από πλήθος μικροϋπηρεσιών, προκύπτει η ανάγκη να υπάρχει κεντρική διαχείριση των ρυθμίσεων και έλεγχος για το αν οι ρυθμίσεις πραγματοποιούνται με επιτυχία σε κάθε υπηρεσία. Ο configuration server μας επιτρέπει να διαχειριζόμαστε τις ρυθμίσεις παραμέτρων όλων των μικροϋπηρεσιών από ένα σημείο. (Spring Cloud Configuration Server – Kubernetes).

Centralized log analysis: Καλή πρακτική σε μια εφαρμογή, αποτελεί η δυνατότητα καταγραφής συμβάντων (log events) σε αρχεία τα οποία αποθηκεύονται τοπικά στους διακομιστές. Στη περίπτωση ενός οικοσυστήματος που αποτελείται από πλήθος μικροϋπηρεσιών, όπου η κάθε υπηρεσία έχει τη δυνατότητα να τρέχει παράλληλα σε πολλά στιγμιότυπα (instances), προκύπτει η ανάγκη για κεντρική παρακολούθηση και αποθήκευση των log files. Μια υπηρεσία που διαχειρίζεται την κεντρική ανάλυση συμβάντων, έχει τη δυνατότητα να ανιχνεύει νέα στιγμιότυπα μικροϋπηρεσιών (new μS[7] instances), να συλλέγει τα event logs, να τα αποθηκεύει και να τα παρουσιάζει με γραφική απεικόνιση. (ELK stack or ELF stack).

Distributed tracing pattern: Είναι απαραίτητο να υπάρχει η δυνατότητα παρακολούθησης της ροής ενός αιτήματος μέσα από τα microservices (request tracing) σε περίπτωση σφάλματος. Για να καταστεί αυτό εφικτό, θα πρέπει όλα τα επιμέρους αιτήματα και μηνύματα, που αφορούν το αρχικό αίτημα, να φέρουν ένα κοινό κωδικό αναγνώρισης (correlation ID) ο οποίος θα καταγράφεται σε όλα τα log events. Για να είμαστε σε θέση να αναλύσουμε καθυστερήσεις και αποτυχίες στη ροή ενός αιτήματος μέσα από τα microservices, θα πρέπει να γίνεται καταγραφή της χρονικής στιγμής (timestamp) που ένα αίτημα ή μια απάντηση σε ένα αίτημα ή μήνυμα εισέρχεται και εξέρχεται από κάθε microservice. (Sleuth – Zipkin).

Circuit breaker pattern: Ένα σύστημα που χρησιμοποιεί σύγχρονη blocking τεχνολογία για την επικοινωνία μεταξύ των microservices, μπορεί να εκτεθεί σε μια αλληλουχία αποτυχιών κατά την έκβαση ενός αιτήματος. Αυτό συμβαίνει διότι μια αποτυχία επικοινωνίας ενός microservice με ένα

Ο φόρτος μοιράζεται σε έναν αριθμό από διακομιστές που εξυπηρετούν συγκεκριμένο σκοπό. Αν κάποιος από αυτούς παρουσιάσει πρόβλημα, οι υπόλοιποι συνεχίζουν το έργο απρόσκοπτα, αν και επιβαρύνονται περισσότερο.

[7] Ο συμβολισμός μS αποτελεί συντομογραφία της λέξης microservice.

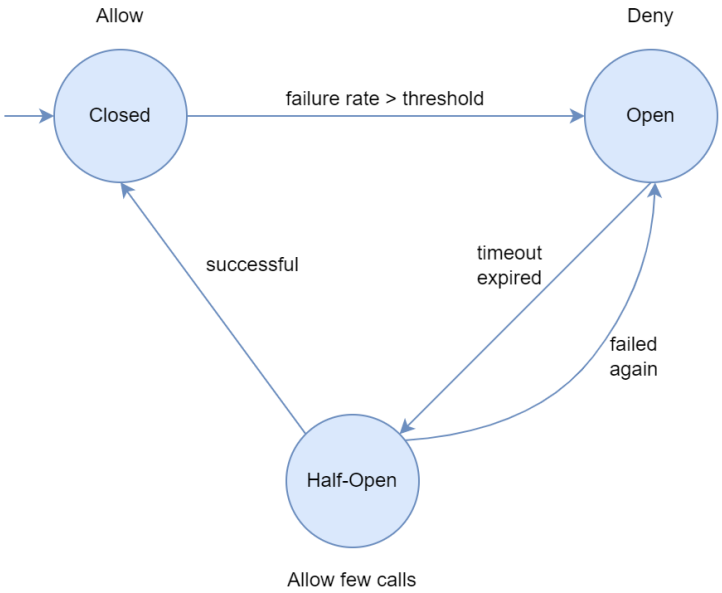
άλλο, μεταφέρει την αποτυχία και στα υπόλοιπα που συμμετέχουν στο αίτημα. Κατά τη διάρκεια της επικοινωνίας μεταξύ των microservices παρατηρούνται δύο ειδών σφάλματα:

- Το πρώτο ονομάζεται **Immediate failure**. Όταν το service αποστολέας στείλει ένα http request και το service παραλήπτης βρίσκεται εκτός λειτουργίας (down status), τότε το service αποστολέας λαμβάνει απευθείας σφάλμα σύνδεσης (connection refused exception). Σε αυτή την περίπτωση, το thread που έχει δεσμευτεί για την συγκεκριμένη διεργασία, απελευθερώνεται άμεσα.
- Το δεύτερο είδος σφαλμάτων είναι τα **Timeout failures**. Όταν η υπηρεσία παραλήπτης έχει λάβει επιτυχώς το http request και καθυστερεί να στείλει την απάντηση (response) στον αποστολέα, τα threads του service αποστολέα παραμένουν σε αναμονή έως ότου λάβουν απάντηση. Εάν ο ρυθμός των requests είναι αρκετά υψηλός, τότε όλα τα thread του thread pool του service αποστολέα θα παραμείνουν απασχολημένα, περιμένοντας απάντηση από το service παραλήπτη, με αποτέλεσμα να εξαντληθούν οι πόροι και να μη μπορεί να εξυπηρετήσει με τη σειρά του τα εισερχόμενα requests προς αυτό. Κατά συνέπεια, η αποτυχία ενός service δημιουργεί αποτυχία και στα services που εξαρτώνται από το πρώτο (cascading failures).

Για τον λόγο αυτό, χρειαζόμαστε ένα μηχανισμό που να διακόπτει προσωρινά το service αποστολέα από το να πραγματοποιεί κλήσεις στο service παραλήπτη που δεν αποκρίνεται. Ο μηχανισμός αυτός ονομάζεται Circuit Breaker και πρόκειται για έναν request interceptor που εφαρμόζεται στο service αποστολέα και ελέγχει την κατάσταση των responses που προέρχονται από το service παραλήπτη. Όσο τα responses επιστρέφουν κανονικά στον χρόνο που προβλέπεται, ο interceptor βρίσκεται στη κατάσταση «Closed» (κλειστό κύκλωμα) επιτρέποντας τη ροή των requests. Αν παρατηρηθεί διακοπή σύνδεσης ή καθυστέρηση των responses πάνω από το επιτρεπόμενο όριο (threshold), τότε διακόπτεται η ροή προς τον συγκεκριμένο παραλήπτη, έτσι τα threads απελευθερώνονται άμεσα, κατάσταση «Open» (ανοικτό κύκλωμα). Υπάρχει επίσης η δυνατότητα να εφαρμοστεί μηχανισμός μετάπτωσης (fallback mechanism), ώστε κατά τη στιγμή της διακοπής του κυκλώματος, να εξυπηρετηθεί με διαφορετικό τρόπο το request ή να εμφανιστεί κάποιο σχετικό μήνυμα. Τα όρια (thresholds) που αποτελούν κριτήριο για το αν θα διακοπή η ροή, ορίζονται στις ρυθμίσεις συστήματος του κάθε service από τον προγραμματιστή. Ένα τέτοιο όριο θα μπορούσε να είναι το ποσοστό αποτυχημένων requests (για παράδειγμα >30%) σε Ν προσπάθειες.

Με τη χρήση του Circuit Breaker design pattern, επιτρέπουμε στην υπηρεσία παραλήπτη να ανακάμψει από το φόρτο διεργασιών και ταυτόχρονα διατηρούμε διαθέσιμους τους πόρους της υπηρεσίας αποστολέα σταματώντας τη ροή. Θα πρέπει όμως, ο μηχανισμός του circuit breaker να είναι σε θέση να ανιχνεύσει την ανάκαμψη του service παραλήπτη ώστε να επιτρέψει εκ νέου τη ροή. Αυτό πραγματοποιείται στη κατάσταση «Half-Open». Ο request interceptor αλλάζει σε κατάσταση «Half-Open», σε προκαθορισμένο χρόνο (timeout), επιτρέποντας μερικά από τα requests να περάσουν προς το service παραλήπτη, δοκιμάζοντας έτσι την κατάσταση του, ενώ τα υπόλοιπα requests εξυπηρετούνται με τον μηχανισμό μετάπτωσης που έχει οριστεί. Αν τα δοκιμαστικά requests εξυπηρετηθούν επιτυχώς, τότε ο request interceptor περνάει σε κατάσταση «Closed» επιτρέποντας σε όλα τα requests να διέρχονται προς το service παραλήπτη. (Resilience4j – Spring Cloud).

Ουσιαστικά πρόκειται για ένα Ντετερμινιστικό Πεπερασμένο Αυτόματο (Finite State Machine) με τρεις καταστάσεις όπως φαίνεται στο παρακάτω σχήμα 3:



Σχήμα 3: Circuit Breaker

Control loop pattern: Σε ένα σύστημα με μεγάλο αριθμό στιγμιοτύπων μικροϋπηρεσιών (instances of μS) τα οποία βρίσκονται διασκορπισμένα σε διαφορετικούς διακομιστές, είναι δύσκολο να εντοπιστούν και να διορθωθούν αποτυχίες που οδηγούν σε πάγωμα ή τερματισμό λειτουργίας ενός μS instance. Το control loop pattern έχει τη δυνατότητα να παρακολουθεί την πραγματική κατάσταση (actual state) του συστήματος, συγκρίνοντας την με την επιθυμητή κατάσταση (desired state). Αν οι δυο καταστάσεις διαφέρουν μεταξύ τους, τότε προχωράει σε ενέργειες ώστε να επαναφέρει το σύστημα στην επιθυμητή κατάσταση (Kubernetes container Orchestration).

Centralized monitoring and alarms: Σε περίπτωση που παρατηρηθεί αργοπορία στην απόκριση του συστήματος και η χρήση πόρων συστήματος παραμένουν σε μη επιτρεπτά υψηλά επίπεδα, είναι δύσκολο να βρούμε την αιτία του προβλήματος χωρίς κάποιο κεντρικό σύστημα παρακολούθησης των πόρων υλικού (hardware resources). Με τη χρήση κατάλληλης υπηρεσίας, που έχει τη δυνατότητα να συλλέγει και να παρουσιάζει με γραφικό τρόπο τις μετρήσεις για τα επίπεδα χρήσης πόρων κάθε μικροϋπηρεσίας, μπορούμε να παρακολουθούμε, να αναλύουμε και να ρυθμίζουμε τις παραμέτρους του συστήματος, ώστε να αποφεύγονται οι αποτυχίες. (Prometheus – Grafana).

Reactive Programming

Με τον όρο Reactive Programming αναφερόμαστε σε ένα προγραμματιστικό μοντέλο κατά το οποίο τα δεδομένα μεταδίδονται μέσω ροών (data streams) από μια ή περισσότερες πηγές που ονομάζονται publishers ή producers. Όλα τα δεδομένα επεξεργάζονται μέσα στις ροές και φθάνουν στον προορισμό με τη σειρά που έχει πραγματοποιηθεί η εκπομπή τους. Μαζί με τα δεδομένα, εκπέμπονται δύο ακόμη σήματα, το πρώτο αφορά σε σφάλματα που ενδέχεται να προκύψουν (onError) και το δεύτερο αφορά στην ολοκλήρωση της μετάδοσης (onComplete). Οι ροές δεδομένων καταλήγουν σε ένα ή περισσότερους αποδέκτες που ονομάζονται subscribers ή consumers. Ένα ακόμη χαρακτηριστικό του Reactive Programming είναι το automatic propagation of change, δηλαδή η ιδιότητα, σε περίπτωση αλλαγής της τιμής μιας μεταβλητής, να ενημερώνονται αυτόματα όλες οι εξαρτώμενες μεταβλητές. Η μετάδοση της ενημέρωσης των εξαρτώμενων μεταβλητών συνεχίζεται αυτόματα μέχρι να ολοκληρωθούν όλες οι αλλαγές που αφορούν άμεσα ή έμμεσα την αλλαγή της αρχικής μεταβλητής.

Ο κύριος λόγος της προσέγγισης του Reactive programming είναι η ανάπτυξη λογισμικού με ασύγχρονο I/O (non-blocking) όπου μετριάζεται η αναποτελεσματική χρήση πόρων, ανακτώντας πόρους που διαφορετικά θα ήταν αδρανείς περιμένοντας τη δραστηριότητα I/O. Με αυτό τον τρόπο, ένας client ειδοποιείται για νέα δεδομένα αντί να τα ζητήσει, έτσι απελευθερώνονται πόροι συστήματος, ώστε να εξυπηρετηθούν άλλες διεργασίες ενώ περιμένει νέες ειδοποιήσεις.

Υπάρχει πάντα ο κίνδυνος κατά τον οποίο πάρα πολλές ειδοποιήσεις μπορεί να κατακλύσουν τον client. Ένας client θα πρέπει να έχει τη δυνατότητα να απορρίψει όγκο διεργασιών που δεν είναι σε θέση να διαχειριστεί. Αυτό αποτελεί μια θεμελιώδη αρχή του ελέγχου ροής (flow control) των κατανεμημένων συστημάτων. Στο reactive programming, η δυνατότητα του client να σηματοδοτεί τον όγκο διεργασιών που είναι σε θέση να διαχειριστεί, ονομάζεται backpressure.

Πολλά projects ομάδων ανάπτυξης λογισμικού υποστηρίζουν το Reactive programming, όπως για παράδειγμα τα projects RxJS και RxJava. Το project της ομάδας του Spring framework που υποστηρίζει το Reactive programming, ονομάζεται project reactor. Όλες οι προσεγγίσεις ακολουθούν ένα de-facto πρότυπο που ονομάζεται Reactive Streams.

Reactive Streams

Το πρότυπο Reactive Streams ορίζει τέσσερις τύπους:

Ο Publisher<T> παράγει τις τιμές τύπου T που προορίζονται να φτάσουν στους αποδέκτες:

```
public interface Publisher<T> {

    void subscribe(Subscriber<? super T> s);
}
```

Ο Subscriber<T> εγγράφεται σε έναν Publisher<T> έτσι ώστε να λαμβάνει ειδοποιήσεις για τις νέες τιμές τύπου T που παράγονται από τον Publisher<T>:

```
public interface Subscriber<T> {

    public void onSubscribe(Subscription s);

    public void onNext(T t);

    public void onError(Throwable t);

    public void onComplete();
}
```

Όταν ένας Subscriber<T> εγγράφεται σε έναν Publisher<T>, το αποτέλεσμα ονομάζεται Subscription<T>:

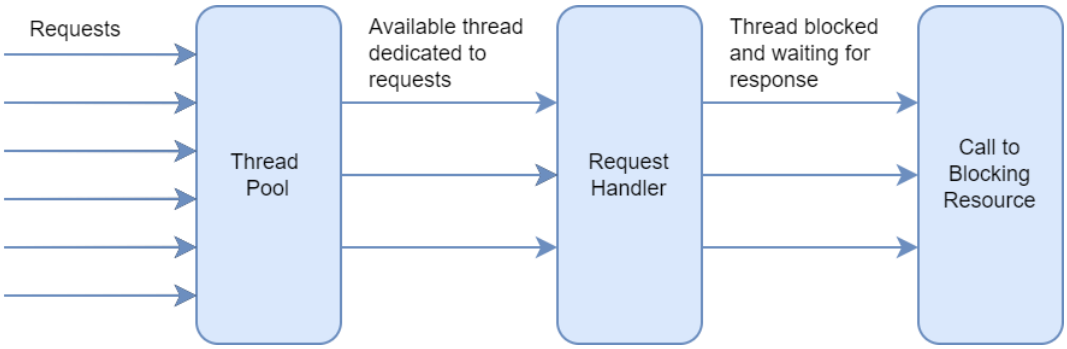
```
public interface Subscription {

    public void request(long n);

    public void cancel();
}
```

Ένας Publisher<T> που είναι επίσης και Subscriber<T> ονομάζεται Processor<T>:

```
public interface Processor<T, R> extends Subscriber<T>, Publisher<R> { }
```

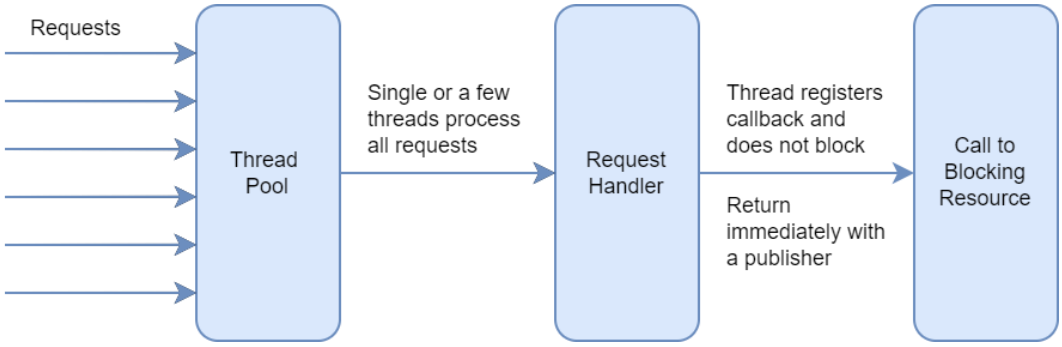
Διαφορά blocking και non-blocking αιτημάτων

Thread-per-request model - Όταν ένα αίτημα εισέρχεται σε έναν διακομιστή (server) από έναν υπολογιστή πελάτη (client), δημιουργείται ένα νήμα αιτήματος (servlet thread) το οποίο αναθέτει το αίτημα σε νήματα εργάτες (worker threads) για λειτουργίες εισόδου/εξόδου (I/O), όπως η πρόσβαση σε βάσεις δεδομένων ή η κλήση σε εξωτερικά APIs. Κατά τη διάρκεια που τα νήματα εργάτες είναι απασχολημένα, το νήμα αιτήματος (servlet thread) παραμένει σε κατάσταση αναμονής και ως εκ τούτου μπλοκάρεται. Καθώς ο διακομιστής διαθέτει πεπερασμένο αριθμό νημάτων αιτημάτων (request threads), όταν βρίσκεται σε πλήρη φόρτο, περιορίζεται η δυνατότητα εξυπηρέτησης των αιτημάτων, που έχει ως αποτέλεσμα την μείωση της απόδοσης και τον περιορισμό της πλήρους χρήσης των δυνατοτήτων του διακομιστή.



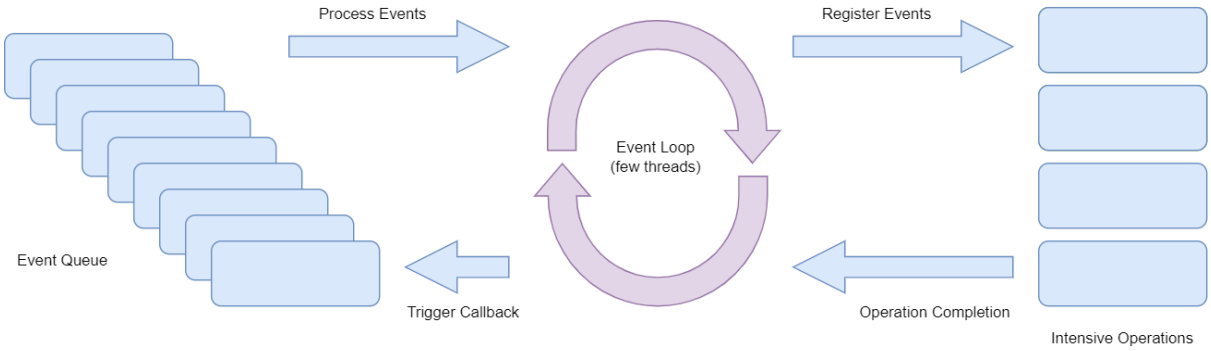
Σχήμα 4, blocking-requests

Στο ασύγχρονο (non-blocking) μοντέλο, η ροή του προγράμματος μετατρέπεται από μια ακολουθία σύγχρονων (blocking) πράξεων σε ασύγχρονη ροή γεγονότων (asynchronous stream of events). Μια κλήση σε βάση δεδομένων δε μπλοκάρει το request thread ενόσω διαβάζονται τα δεδομένα. Το request επιστέφει άμεσα έναν publisher όπου κάποιος μπορεί να πραγματοποιήσει εγγραφή (subscribe). Ο subscriber μπορεί να καταναλώσει (consume) και να επεξεργαστεί το event όταν αυτό συμβεί, εν συνεχεία μπορεί να παράξει με τη σειρά του νέο event:



Σχήμα 5, non-blocking-request

Το reactive programming δε δίνει έμφαση σε ποιο νήμα (thread) θα πρέπει να δημιουργηθούν ή να καταναλωθούν τα events, αλλά στη δομή του προγράμματος ως ασύγχρονη ροή γεγονότων.



Σχήμα 6, event-loop

Στο σχήμα 6 διακρίνεται μια αφηρημένη σχεδίαση ενός event-loop που παρουσιάζει τις ιδέες του reactive programming.

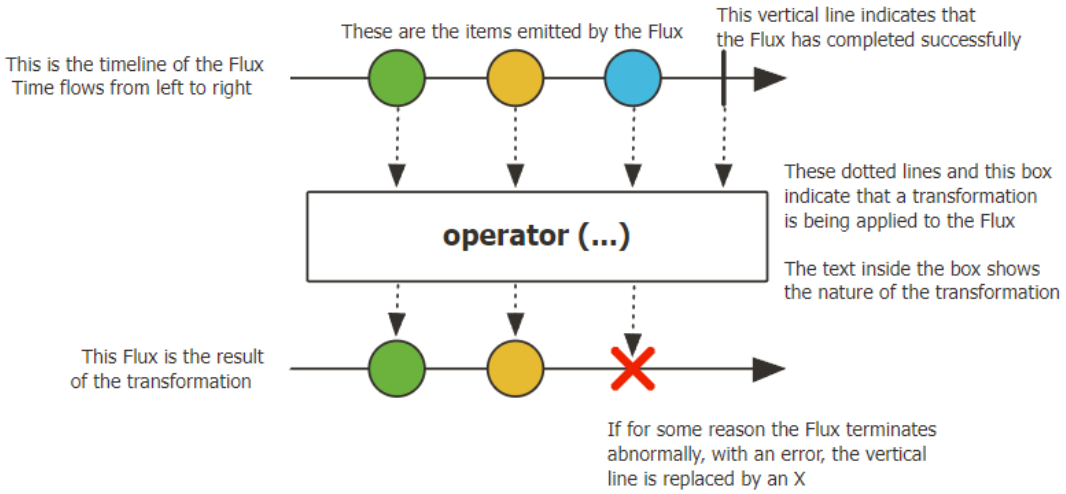
- Το event-loop τρέχει συνεχόμεναι σε ένα μοναδικό νήμα (thread), ωστόσο υπάρχει η δυνατότητα να δημιουργηθούν τόσα event loops όσοι και οι διαθέσιμοι πυρήνες του επεξεργαστή του συστήματος.
- Το event-loop επεξεργάζεται διαδοχικά τα events από μια ουρά γεγονότων (event queue), καταχωρώντας στο σύστημα την πληροφορία callback που φέρουν τα events και στη συνέχεια απελευθερώνονται άμεσα.
- Το σύστημα ειδοποιεί για την ολοκλήρωση μιας ενέργειας, όπως η κλήση σε μια βάση δεδομένων ή η κλήση σε μια εξωτερική υπηρεσία.
- Το event-loop ενεργοποιεί την callback function κατά την ολοκλήρωση μιας ενέργειας, ώστε να σταλούν τα αποτελέσματα πίσω στο σύστημα που είχε πραγματοποιήσει το αρχικό αίτημα.

Το μοντέλο του event loop υλοποιείται σε γνωστές πλατφόρμες όπως το Node.js, Netty και Nginx.

Project Reactor

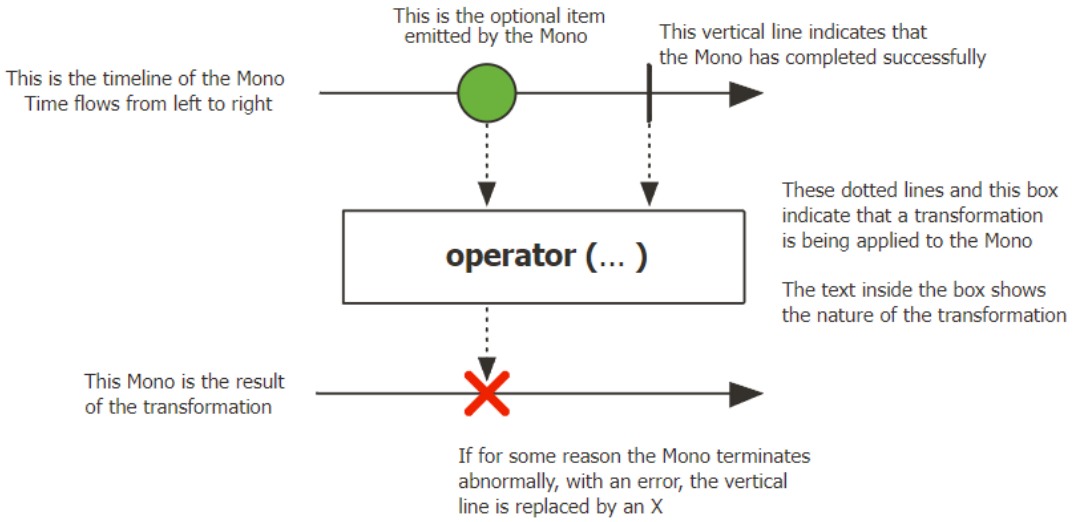
Οι τύποι του προτύπου reactive streams δεν επαρκούν και χρειάζονται υλοποιήσεις υψηλότερου επιπέδου, έτσι ώστε να υποστηρίζονται τελεστές (operators) όπως φιλτράρισμα (filtering) και μετασχηματισμός (transformation) των δεδομένων. Το project reactor έχει βασιστεί πάνω στο πρότυπο των Reactive Streams και παρέχει δύο εξειδικεύσεις του Publisher<T>.

Η τάξη Flux<T> αποτελεί τη πρώτη υλοποίηση της διεπαφής Publisher<T> και CorePublisher<T>. Η υλοποίηση Flux<T> εκπέμπει 0 έως N στοιχεία και στη συνέχεια ολοκληρώνεται είτε επιτυχώς, είτε με σφάλμα (onComplete, onError).



Σχήμα 7, πηγή projectreactor.io

Η τάξη Mono<T> αποτελεί τη δεύτερη υλοποίηση της διεπαφής Publisher<T> και CorePublisher<T>. Η υλοποίηση Mono<T> εκπέμπει 0 έως 1 στοιχεία και στη συνέχεια ολοκληρώνεται είτε επιτυχώς, είτε με σφάλμα (onComplete, onError).



Σχήμα 8, πηγή projectreactor.io

Και οι δυο υλοποιήσεις της διεπαφής Publisher<T> παρέχουν τελεστές και τρόπους επεξεργασίας των ροών δεδομένων. Εάν γνωρίζουμε εκ των προτέρων πως ο υποκείμενος publisher θα εκπέμψει 0 ή 1 στοιχεία, τότε θα πρέπει να χρησιμοποιηθεί η υλοποίηση Mono<T>.

Reactive Spring

Το project reactor αποτελεί τη βάση, μια εφαρμογή χρειάζεται να επικοινωνεί με data sources, να παράγει και να καταναλώνει HTTP, SSE (Server Sent Events) και WebSocket endpoints. Επίσης χρειάζεται να υποστηρίζει έλεγχο ταυτότητας χρηστών (authentication) και εξουσιοδότηση χρηστών που αφορά τη πρόσβαση στους πόρους της εφαρμογής (authorization). Το Reactive Spring, από το Spring Framework 5.0 και ύστερα, πλαισίωσε το project reactor και το πρότυπο reactive streams με όλα τα απαραίτητα στοιχεία για την ανάπτυξη reactive εφαρμογών, καθώς ανέπτυξε ένα νέο μοντέλο εκτέλεσης και στοιχείων (runtime and component model) που ονομάζεται Spring WebFlux. Το Spring WebFlux δεν εξαρτάται από κάποιο συγκεκριμένο Servlet API προκειμένου να λειτουργήσει, έρχεται με προσαρμογείς (adapters) που του επιτρέπουν να λειτουργεί πάνω από μια Servlet-engine, καθώς επίσης προσφέρει και ενσωματωμένο reactive non-servlet-based διακομιστή (web-server) βασισμένο στο Netty[8].

Reactive Spring Data module

Η έκδοση Spring Data Kay, που αποτελεί την μεγαλύτερη ενημέρωση του Spring Data module από τη δημιουργία του, ξεκίνησε να προσφέρει υποστήριξη για reactive data access σε συγκεκριμένες NoSQL βάσεις δεδομένων όπως η MongoDB, Cassandra και Couchbase, καθώς και σε in-memory data structure stores όπως το Redis. Η εν λόγω έκδοση, εισήγαγε reactive repository και template υλοποιήσεις, καθώς και οδηγούς (drivers) για τη Reactive MongoDB, η οποία έχει χρησιμοποιηθεί στη παρούσα μεταπτυχιακή διατριβή.

Οι reactive βάσεις δεδομένων ακολουθούν το πρότυπο του reactive programming. Οι κλήσεις προς τη βάση δεδομένων είναι non-blocking, με αποτέλεσμα την απελευθέρωση πόρων συστήματος.

Επιπλέον, η Reactive MongoDB δίνει τη δυνατότητα να πραγματοποιηθούν live feeds μέσω streams από MongoDB Capped Collections[9] διαμέσου WebSocket, comet ή άλλων streaming πρωτοκόλλων. Capped Collection είναι ένα MongoDB Collection σταθερού μεγέθους (FIFO), όπου κάθε φορά που μια εγγραφή (Document) εισάγεται στο Collection, η web εφαρμογή το μεταδίδει με non-blocking τρόπο σε όσους clients έχουν εγγραφεί (subscribers).

Επίσης, η βιβλιοθήκη GridFS[10] της MongoDB μπορεί να χρησιμοποιηθεί και ως non-blocking (ReactiveGridFsTemplate), όπου ανακτά ένα προς ένα τα κομμάτια ενός αρχείου προς αποθήκευση και τα μεταδίδει μέσω ροής μέχρι ο client να μην έχει άλλα δεδομένα προς αποστολή. Και σε αυτή την περίπτωση, τα πλεονεκτήματα είναι πως δεν μπλοκάρονται νήματα (threads) κατά τη διάρκεια της διαδικασίας και δε σπαταλάται μνήμη.

[8] Netty is an asynchronous event-driven network application framework for rapid development of maintainable high performance protocol servers & clients. https://netty.io/

[9] Η MongoDB αποθηκεύει τα δεδομένα σε εγγραφές που ονομάζονται Documents (BSON documents), τα οποία συγκεντρώνονται σε μια συλλογή Collection.

[10] Η βιβλιοθήκη GridFS χρησιμοποιείται για την αποθήκευση και ανάκτηση αρχείων, τύπου BSON-document στη MongoDB, που υπερβαίνουν το όριο μεγέθους των 16MB.

Spring Boot

Το Spring Boot είναι ένα Java framework ανοικτού κώδικα που έχει αναπτυχθεί πάνω στο Spring framework από την ομάδα της Pivotal και προσφέρει τη δυνατότητα κατασκευής production-ready web εφαρμογών και μικροϋπηρεσιών. Το Spring framework πρωτοπαρουσιάστηκε το 2004 και ένας από τους στόχους του ήταν να μειώσει την πολυπλοκότητα του J2EE (Java 2 Platforms, Enterprise Edition). Ενώ προσφέρεται πλήθος από dependencies, όπως τα Spring Web Services, το Spring Security και Spring JPA, οι προγραμματιστές πρέπει να ρυθμίζουν από μόνοι τους την εφαρμογή με χρήση αρχείων XML ή annotations[11]. Εν αντιθέσει, το Spring Boot, που πρωτοπαρουσιάστηκε το 2014 (Spring Boot v1.0), επικεντρώνεται στη μείωση του κώδικα ώστε να παρέχει ευκολότερο τρόπο δημιουργίας μιας Spring εφαρμογής.

Κάποια από τα πλεονεκτήματα του Spring framework είναι:

- Προσφέρει ευελιξία μέσω του dependency injection[12] δημιουργώντας χαμηλή εξάρτηση μεταξύ των στοιχείων που συνθέτουν την εφαρμογή.
- Πρόκειται για ένα ελαφρύ framework.
- Παρέχει υποστήριξη παραμετροποίησης της εφαρμογής, είτε με αρχεία XML, είτε με annotations.
- Παρέχει υποστήριξη ORM[13] λογισμικού μέσω abstraction.
- Συμβατότητα με υπηρεσίες ενδιάμεσου κώδικα (middleware services).
- Υποστήριξη JDBC framework[14] το οποίο βελτιώνει την παραγωγικότητα και μειώνει τα σφάλματα που δυνητικά θα μπορούσαν να προκύψουν.

Το Spring Boot, βασισμένο πάνω στο Spring framework, έχει τα εξής χαρακτηριστικά:

Autoconfiguration: Ο convention-based μηχανισμός του autoconfiguration μπορεί να ενεργοποιηθεί χρησιμοποιώντας το annotation *@SpringBootApplication* στην Application.class, η οποία περιέχει τη στατική main μέθοδο της εφαρμογής. Το *@SpringBootApplication* παρέχει την εξής λειτουργικότητα:

- Ενεργοποιεί το component scanning το οποίο αναζητά Spring components και configuration classes στα packages της εφαρμογής. Αν χρειαστεί να χρησιμοποιηθούν components από packages που βρίσκονται εκτός του application package, τότε θα πρέπει να χρησιμοποιηθεί το annotation *@ComponentScan* σε συνδυασμό με το *@SpringBootApplication* annotation ώστε να δηλωθεί η διαδρομή της τοποθεσίας των packages. Στη συνέχεια τα εξωτερικά components μπορούν να χρησιμοποιηθούν στις τάξεις (classes) της εφαρμογής κάνοντας τα inject (dependency injection) με χρήση του *@Autowired* annotation.
- Η application class γίνεται και η ίδια configuration class.

[11] Τα Spring Annotations είναι μια μορφή metadata που παρέχουν συμπληρωματικές πληροφορίες σχετικά για τις ρυθμίσεις των dependencies και δημιουργούν dependency injection στην εφαρμογή.

[12] Dependency Injection είναι μια τεχνική προγραμματισμού που επιτυγχάνει να δημιουργήσει χαμηλή εξάρτηση μιας τάξης (class) από άλλες τάξεις, αποσυνδέοντας την χρήση ενός αντικειμένου από την αρχικοποίηση του.

[13] Object Relational Mapping είναι μια τεχνική προγραμματισμού που δίνει τη δυνατότητα να διαχειριζόμαστε δεδομένα από μια βάση δεδομένων χρησιμοποιώντας αντικειμενοστρεφή προγραμματισμό. https://en.wikipedia.org/wiki/Object-relational_mapping

[14] Το JDBC framework προσφέρει μεθόδους που αφορούν την επικοινωνία με μια βάση δεδομένων, όπως η σύνδεση με τη βάση, η εκτέλεση επερωτήσεων, η διαχείριση σφαλμάτων, η διαχείριση των SQL transactions και το κλείσιμο της σύνδεσης με τη βάση.

- Ενεργοποιεί το autoconfiguration, όπου το Spring Boot αναζητά JAR αρχεία στο Classpath, ώστε να αλλάξει ρυθμίσεις αυτόματα, ανάλογα με τα dependencies που έχουν χρησιμοποιηθεί. Για παράδειγμα, αν βρεθεί ο Tomcat server στο Classpath, τότε το Spring Boot θα ρυθμιστεί αυτόματα ώστε να χρησιμοποιεί τον Tomcat ως ενσωματωμένο web server.

Standalone: Δε χρειάζεται η εγκατάσταση της εφαρμογής σε κάποιο διακομιστή (server), η εφαρμογή μπορεί να εκκινήσει χρησιμοποιώντας μια εντολή (mvn spring-boot: run).

Opinionated: Το Spring boot έχει τη δυνατότητα να αποφασίζει ποιες θα είναι οι προεπιλεγμένες παράμετροι που θα χρησιμοποιήσει, επίσης, εάν συμπεριλάβουμε "starter" dependencies στο pom.xml, τότε θα συμπεριλάβει αυτόματα και άλλα dependencies που κρίνει απαραίτητα. Για παράδειγμα το starter JPA dependency, συμπεριλαμβάνει dependencies όπως in-memory database, hibernate entity manager και simple data source.

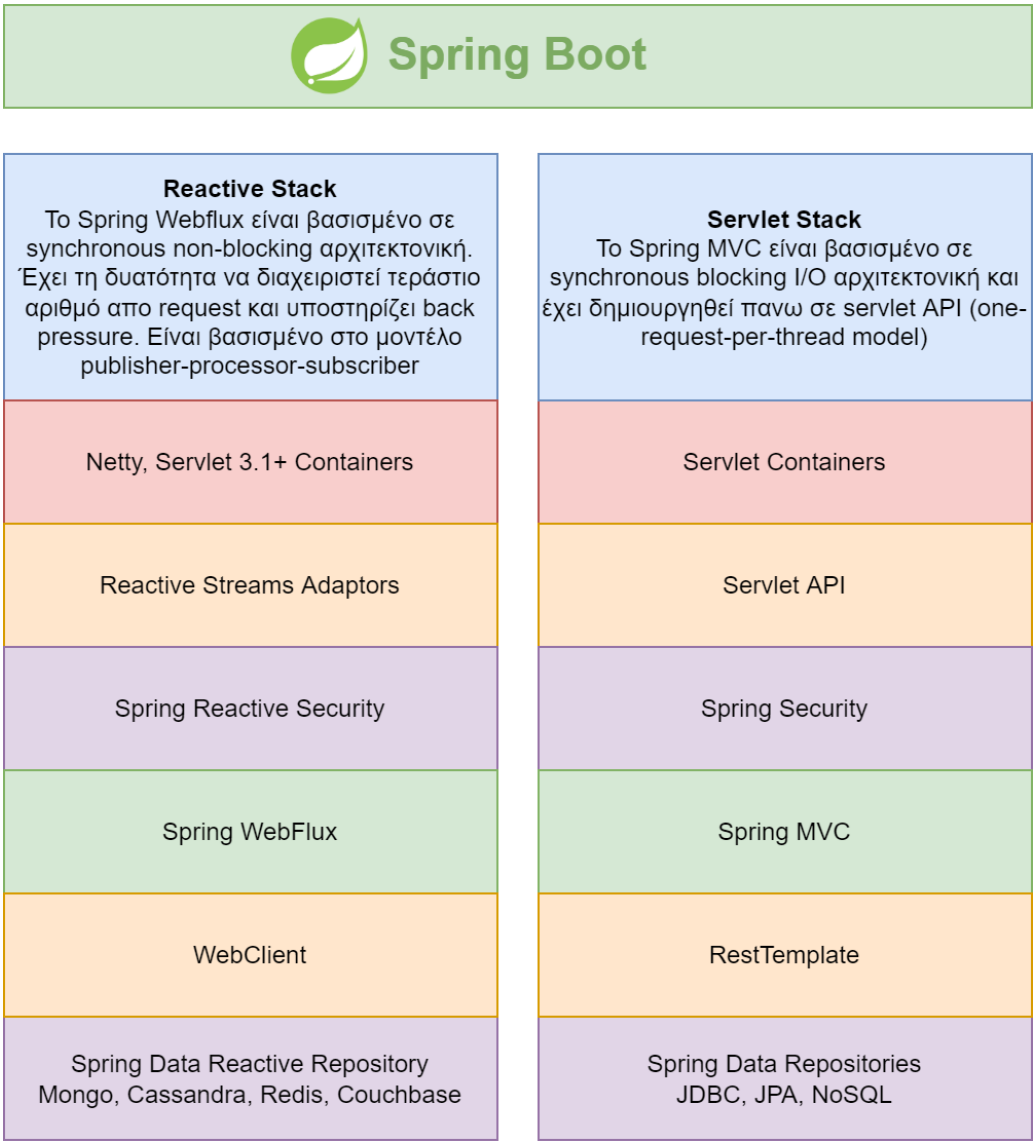
Συνοπτικά τα πλεονεκτήματα που διαθέτει το Spring Boot είναι:

- Η μείωση του χρόνο ανάπτυξης του λογισμικού, που έχει ως συνέπεια την αύξηση της παραγωγικότητας.
- Η βοήθεια που παρέχει μέσω του autoconfiguration, ώστε να ρυθμιστεί κατάλληλα η εφαρμογή και να είναι έτοιμη για παραγωγή.
- Η διευκόλυνση που παρέχει στους προγραμματιστές, ώστε να δημιουργήσουν και να τεστάρουν τις εφαρμογές, παρέχοντας υποδομή για unit και integration tests.
- Η αποτροπή της δημιουργίας τετριμμένου κώδικα (boilerplate code), annotations και ρυθμίσεων XML.
- Περιέχει ενσωματωμένο HTTP server, όπως το Apache Tomcat, Jetty ή Netty.
- Παρέχει πλήθος plugins, ώστε οι προγραμματιστές να μπορούν εύκολα να εργαστούν με in-memory βάσεις δεδομένων όπως η H2, να συνδεθούν με βάσεις δεδομένων όπως η Oracle, PostgreSQL, MySQL, MongoDB, με caching in-memory data structure stores όπως το Redis, καθώς και με message queue brokers όπως το RabbitMQ, ActiveMQ και Apache Kafka.
- Επιτρέπει τη διαχείριση της εφαρμογής μέσω απομακρυσμένης σύνδεσης.

Μειονέκτημα του Spring Boot που έχει αναφερθεί από κάποιους προγραμματιστές, είναι η έλλειψη ελέγχου. Το "opinionated" χαρακτηριστικό εγκαθιστά πολλά dependencies που θεωρεί απαραίτητα, με αποτέλεσμα να αυξάνεται το μέγεθος της εφαρμογής. Επίσης, ένα μειονέκτημα είναι η δυσκολία της μετατροπής legacy Spring κώδικα σε Spring Boot, ωστόσο υπάρχουν εργαλεία όπως το Spring Boot CLI tool που μπορούν να βοηθήσουν στη μετατροπή legacy code σε Spring Boot.

Spring WebFlux

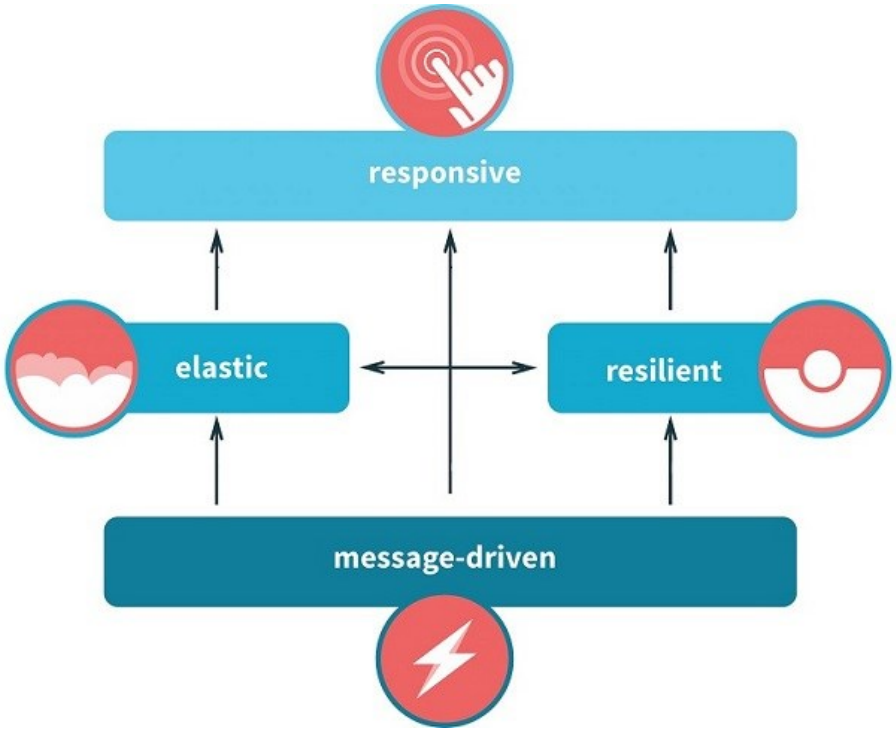
Το Spring Boot 2.0 είναι βασισμένο στο Spring framework 5.0 και παρέχει ενσωματωμένη υποστήριξη στην ανάπτυξη εφαρμογών με reactive programming, εισάγοντας ένα νέο web framework, το Spring WebFlux, που αποτελεί υλοποίηση του project Reactor. Το Spring WebFlux υποστηρίζει δυο διαφορετικά προγραμματιστικά μοντέλα. Το πρώτο είναι βασισμένο στη χρήση annotations όπως συμβαίνει με το προϋπάρχων Spring Web framework και το Spring Web MVC αλλά με υποστήριξη reactive programming. Το δεύτερο είναι ένα νέο function-oriented μοντέλο βασισμένο σε routes και handlers. Επίσης, το Spring WebFlux παρέχει ένα νέο reactive HTTP Client που ονομάζεται WebClient και αντικαθιστά το προϋπάρχων RestTemplate. Τέλος, το Spring WebFlux μπορεί να τρέξει σε Servlet (≥ specification v3.1) όπως το Apache Tomcat αλλά και σε non-Servlet ενσωματωμένο web server όπως το Netty.



Σχήμα 9: Spring Boot Web Stacks

Reactive Microservices

Η σχεδίαση reactive (non-blocking) συστημάτων έχει ως βάση την επικοινωνία μεταξύ των υποσυστημάτων με ασύγχρονο τρόπο, χρησιμοποιώντας μηνύματα (message-driven asynchronous communication). Με αυτό τον τρόπο, δίνεται η δυνατότητα στο σύστημα να είναι ανθεκτικό στις αποτυχίες (resilient and fault tolerant), καθώς επίσης, έχει τη δυνατότητα να αυξάνει την ισχύ του δημιουργώντας παράλληλα στιγμιότυπα (instance scaling) όταν αυτό απαιτείται ή αντίθετα να μειώνει τα στιγμιότυπα όταν υπάρχει περιορισμένο φόρτο εργασίας, ώστε να εξοικονομεί πόρους και κόστος (scalable and elastic). Τα παραπάνω χαρακτηριστικά κάνουν τα microservices **responsive,** που σημαίνει πως ανταποκρίνονται άμεσα στα αιτήματα και έχουν zero-downtime.



Σχήμα 10: Reactive Microservices

Όταν κατασκευάζουμε μικροϋπηρεσίες, δεν είναι πάντα εμφανές σε ποια περίπτωση πρέπει να χρησιμοποιήσουμε non-blocking σύγχρονα (synchronous) APIs και πότε message-driven ασύγχρονη επικοινωνία (asynchronous communications). Προκειμένου να είναι ένα microservice robust και scalable, θα πρέπει να είναι όσο το δυνατό αυτόνομο, έχοντας ελάχιστες εξαρτήσεις (runtime dependencies), αυτό ονομάζεται **loose coupling**. Έτσι, ένα ασύγχρονο (async) μήνυμα που αποστέλλει events, είναι προτιμότερο από ένα σύγχρονο (synchronous) API. Με τον τρόπο αυτό, το microservice θα εξαρτάται αποκλειστικά από το σύστημα μηνυμάτων (message broker), όπως το RabbitMQ ή το Apache Kafka, που είναι προτιμότερο από το να εξαρτάται από σύγχρονη επικοινωνία με πλήθος από άλλα microservices.

Ωστόσο, υπάρχουν περιπτώσεις που τα σύγχρονα (synchronous) APIs είναι καταλληλότερα:

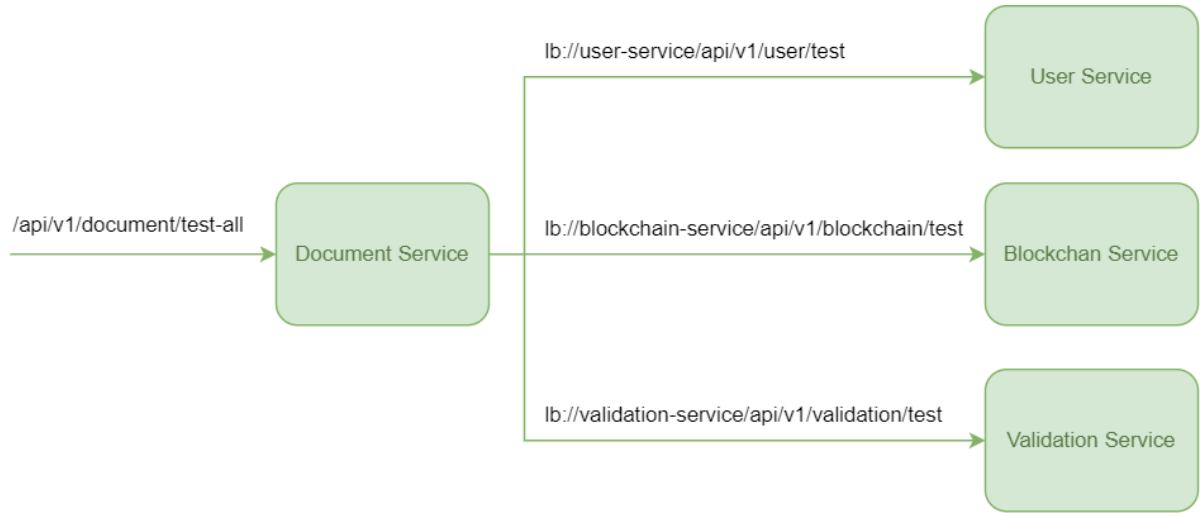
- Όταν πρόκειται για διαδικασίες ανάγνωσης όπου ο τελικός χρήστης περιμένει κάποια απόκριση (response) του συστήματος.
- Όταν ένα σύστημα πελάτης (client) είναι πιο κατάλληλο να καταναλώνει σύγχρονα (synchronous) APIs, για παράδειγμα SPAs (Single Page Applications) ή εφαρμογές κινητών συσκευών.
- Όταν ένα σύστημα συνδέεται με εξωτερικά συστήματα, μπορεί να μην είναι εφικτό τα εξωτερικά συστήματα να έχουν πρόσβαση στον message broker, είτε να μην χρησιμοποιούν το ίδιο σύστημα αποστολής μηνυμάτων.

Το σύστημα που έχει σχεδιαστεί στην παρούσα εργασία, κάνει αποκλειστικά χρήση reactive non-blocking programming και αξιοποιεί τόσο τη χρήση reactive σύγχρονων APIs, όσο και reactive ασύγχρονων μηνυμάτων με χρήση της τεχνολογίας RabbitMQ.

Σύγχρονη (synchronous) non-blocking επικοινωνία

Non-blocking σύγχρονη επικοινωνία (synchronous communication) είναι η μέθοδος κατασκευής APIs με reactive τρόπο, ώστε όταν πραγματοποιείται μια κλήση στο API ενός microservice και εκείνο με τη σειρά του απαιτείται να πραγματοποιήσει κλήσεις σε άλλα υποσυστήματα, για παράδειγμα σε τρία ακόμη microservices, τότε αυτό να γίνεται ταυτόχρονα με παράλληλο τρόπο. Όταν το αρχικό microservice λάβει απάντηση (response) και από τα τρία microservices, τότε συνθέτει την πληροφορία και επιστρέφει το αποτέλεσμα στον αποστολέα του αρχικού request.

Η παραπάνω λογική έχει χρησιμοποιηθεί εκτενώς στα microservices του έργου που έχει υλοποιηθεί στο πλαίσιο της μεταπτυχιακής διατριβής. Ακολουθεί ένα παράδειγμα από τον κώδικα της εφαρμογής, όπου τεστάρουμε το status όλων των microservices με μια κλήση στο API του Document Service.



Σχήμα 11: Test-all

DocumentController.class

```
@GetMapping("/test-all")
public Flux<TestDto> testAllServices() {
    return documentService.testAllServices().map(TestDto::new);
}
```

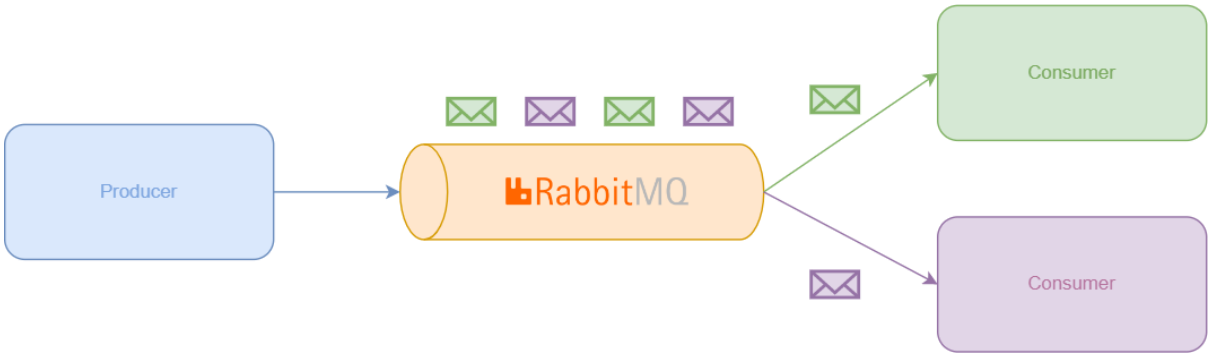
DocumentServiceImpl.class

```
@Override
public Flux<String> testAllServices() {
    List<String> services = Collections.synchronizedList(Arrays.asList(
            "user-service", "document-service", "blockchain-service",
            "validation-service"
    ));

    return Flux.fromIterable(services)
            .map(s -> webClient.build()
                    .get()
                    .uri("lb://" + s + "/api/v1/" + s.split("-")[0] + "/test")
                    .retrieve()
                    .bodyToMono(String.class))
            .flatMap(Flux::from);
}
```

Ασύγχρονη (asynchronous) message-driven επικοινωνία

Για την υλοποίηση της message-driven επικοινωνίας, απαιτείται η χρήση message broker όπως το RabbitMQ ή το Apache Kafka. Πρόκειται για ένα διακομιστή ή σύστημα διακομιστών (Kafka cluster[15]) που έχει τον ρόλο του μεσάζοντα. Τα microservices μπορούν να στείλουν μηνύματα στον message broker μέσω των producer/publisher μεθόδων. Τα μηνύματα αποθηκεύονται σε ουρές (queues) ή topics για όσο χρόνο κριθεί σκόπιμο από τη λογική της υλοποίησης. Στη συνέχεια, τα microservices που καταναλώνουν τα μηνύματα, προκειμένου να προχωρήσουν σε κάποια ενέργεια, πραγματοποιούν εγγραφή στον message broker μέσω των consumer/subscriber μεθόδων και περιμένουν να λάβουν νέα μηνύματα, είτε να καταναλώσουν τα ήδη αποθηκευμένα. Με αυτό τον τρόπο επιτυγχάνεται η ασύγχρονη επικοινωνία, διότι δεν απαιτείται ταυτόχρονη συνύπαρξη των publishers και consumers. Αυτό έχει ως συνέπεια να επιτυγχάνεται **loosely coupling** (χαλαρή σύζευξη) και **scalability** μεταξύ των microservices, δηλαδή η δυνατότητα να χρησιμοποιούνται πολλά στιγμιότυπα ενός microservice ταυτόχρονα και να καταναλώνουν μηνύματα από την ίδια ουρά.

Να σημειωθεί επίσης πως μέθοδοι producers/publishers και consumers/subscribers μπορούν να συνυπάρχουν στο ίδιο microservice και να παράγουν ή να καταναλώνουν αντίστοιχα, διαφορετικά μηνύματα που αφορούν σε διαφορετικές ενέργειες. Μια συνήθης χρήση είναι οι consumers να καταναλώνουν μηνύματα, τα μηνύματα να επεξεργάζονται στο microservice και στη συνέχεια οι producers να παράγουν νέα μηνύματα σε διαφορετική μορφή.

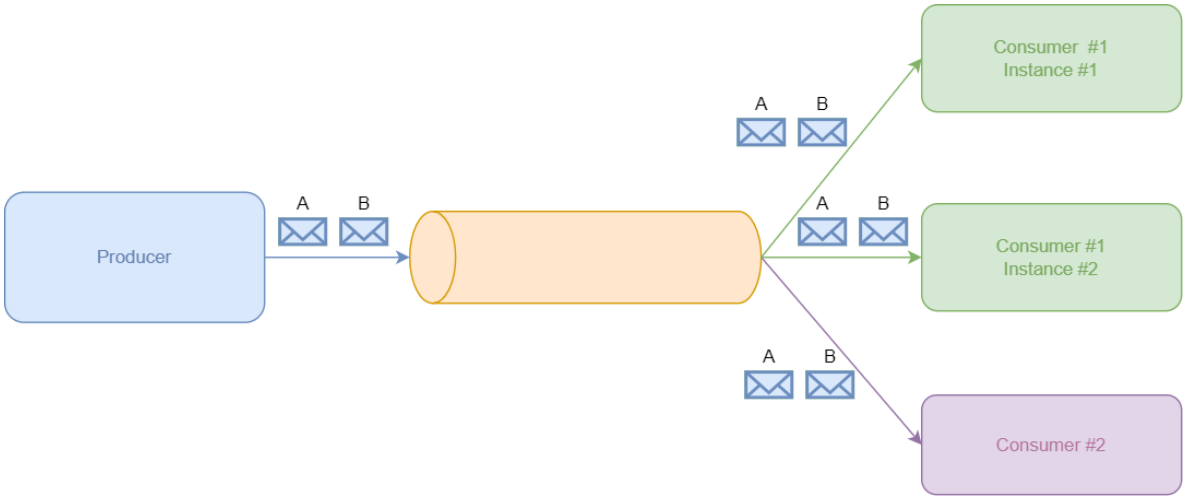


Σχήμα 12: Producer – Consumers

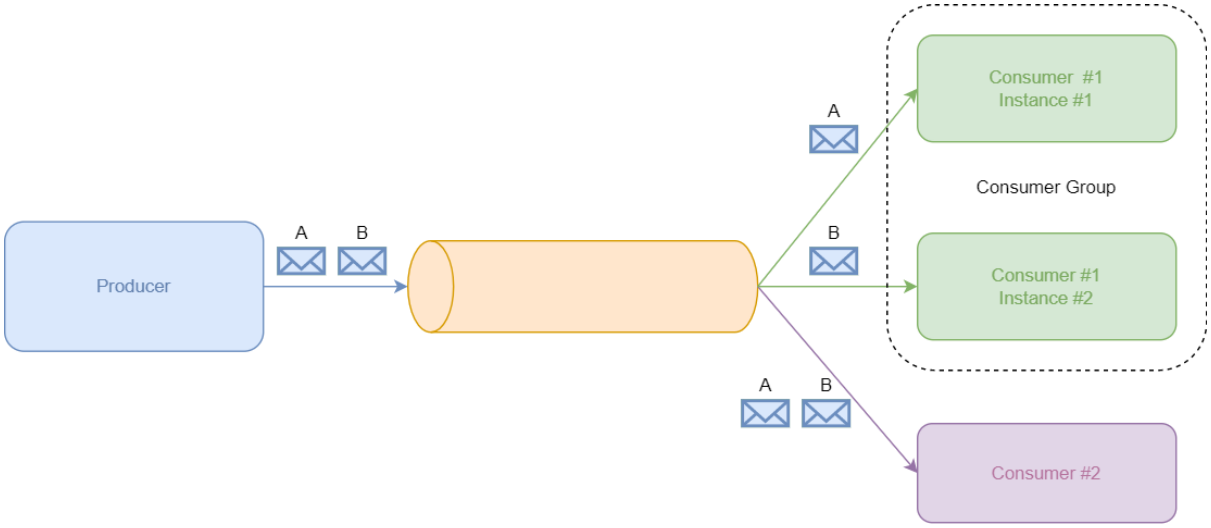
Οι προκλήσεις που προκύπτουν με τη χρήση message-driven ασύγχρονου προγραμματισμού, καθώς και τα σχεδιαστικά μοτίβα που ακολουθούνται για την επίλυση τους είναι τα παρακάτω:

Consumer groups: Αν ανεβάσουμε τον αριθμό των στιγμιοτύπων (scale up instances) ενός microservice που καταναλώνει μηνύματα από τον message broker, για παράδειγμα σε δύο στιγμιότυπα, τότε και τα δυο στιγμιότυπα θα καταναλώνουν το ίδιο μήνυμα και κατά συνέπεια θα προχωρούν σε επεξεργασία του ίδιου μηνύματος (Σχήμα 13). Αυτό, μπορεί να οδηγήσει σε διπλοεγγραφές σε μια βάση δεδομένων ή άλλες ανεπιθύμητες ενέργειες. Ο τρόπος με τον οποίο μπορούμε να επιλύσουμε το συγκεκριμένο πρόβλημα είναι με τη χρήση των **consumer groups**.

[15] A Kafka cluster is a system that consists of several Brokers, Topics, and Partitions for both. The key objective is to distribute workloads equally among replicas and Partitions. Kafka Clusters Architecture mainly consists of the following 5 components: Topics. Broker.



Σχήμα 13: No Consumer Group



Σχήμα 14: Consumer Group

Retries and dead-letter queues: Αν ένας consumer αποτύχει να επεξεργαστεί ένα μήνυμα, τότε το μήνυμα χρειάζεται να ξαναμπεί στην ουρά (queue) μέχρι να καταφέρει να επεξεργαστεί από τον consumer. Αν το περιεχόμενο του μηνύματος είναι λανθασμένο, γνωστό και ως poisoned message, τότε το μήνυμα θα μπλοκάρει τον consumer από το να επεξεργαστεί άλλα μηνύματα μέχρι να αφαιρεθεί. Αν η αποτυχία οφείλεται σε κάποιο προσωρινό πρόβλημα, όπως η αποτυχία σύνδεσης με μια βάση δεδομένων λόγω προσωρινού προβλήματος στο δίκτυο, τότε πιθανότατα η επεξεργασία θα επιτύχει μετά από κάποιες επαναλαμβανόμενες προσπάθειες.

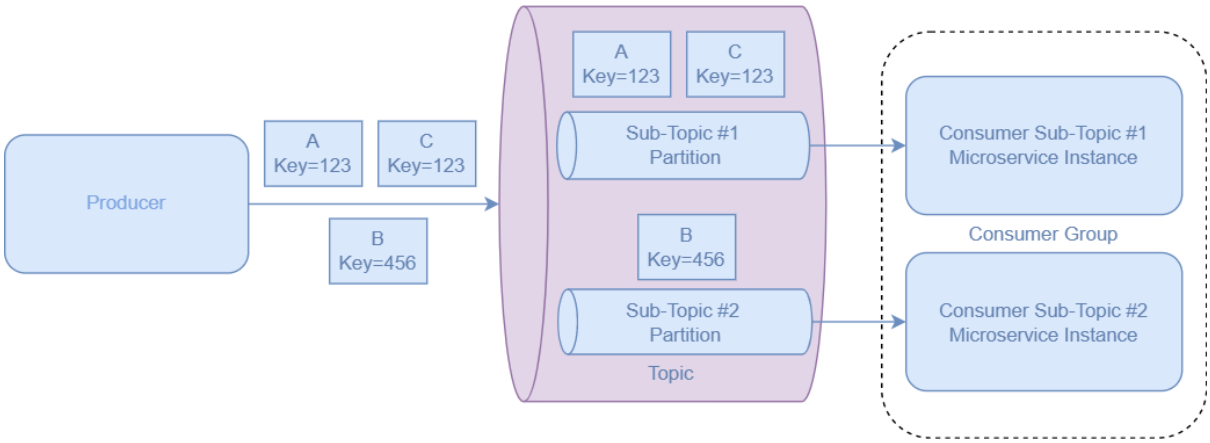
Θα πρέπει να υπάρχει σαφώς ορισμένος αριθμός επαναλήψεων στις προσπάθειες που πραγματοποιούνται προκειμένου να επεξεργαστεί ένα μήνυμα, στη συνέχεια θα πρέπει να μεταφέρεται σε μια ουρά που ονομάζεται dead-letter-queue. Για να αποφευχθεί η υπερφόρτωση της υποδομής από προσωρινές αποτυχίες, όπως τα σφάλματα δικτύου, θα πρέπει να υπάρχει η δυνατότητα ρύθμισης του πλήθους και της συχνότητας των επαναπροσπαθειών (retries) που πραγματοποιούνται, με τέτοιο τρόπο, ώστε να αυξάνεται ο χρόνος μεταξύ των προσπαθειών.

Guaranteed order and partitions: Αν η λογική της εφαρμογής απαιτεί την κατανάλωση των μηνυμάτων με τη σειρά που έχουν παραχθεί, τότε δε μπορούν να χρησιμοποιηθούν πολλά στιγμιότυπα ενός microservice ώστε να αυξηθεί η απόδοση του συστήματος, για παράδειγμα δε

μπορούν να χρησιμοποιηθούν consumer groups. Αυτό μπορεί να οδηγήσει σε μη αποδεκτές καθυστερήσεις στην επεξεργασία των εισερχόμενων μηνυμάτων. Για τον λόγο αυτό, υπάρχει η δυνατότητα να χρησιμοποιηθούν **partitions**, ώστε να διασφαλιστεί η κατανάλωση των μηνυμάτων με τη σειρά που έχουν παραχθεί, χωρίς να χαθεί η δυνατότητα δημιουργίας στιγμιοτύπων (scaling).

Τις περισσότερες φορές, η απαίτηση της κατανάλωσης μηνυμάτων με τη σειρά που έχουν παραχθεί, χρειάζεται μόνο σε περιπτώσεις που τα μηνύματα επηρεάζουν την ίδια οντότητα (business entity). Για παράδειγμα, τα μηνύματα που επηρεάζουν το προϊόν με product_id = 1, μπορούν να επεξεργαστούν ξεχωριστά από τα μηνύματα που επηρεάζουν το προϊόν με product_id = 2, αυτό σημαίνει πως η σειρά θα πρέπει να τηρηθεί μόνο για τα μηνύματα με ίδιο product_id.

Για να επιλυθεί το συγκεκριμένο πρόβλημα, θα πρέπει να οριστεί ένα **κλειδί (key)** για κάθε μήνυμα, όπου ο message broker θα μπορεί να εγγυηθεί την τήρηση της σειράς ανάμεσα στα μηνύματα που έχουν το ίδιο κλειδί. Η λύση βρίσκεται στη χρήση sub-topics, ή αλλιώς partitions in a topic. Ο message broker τοποθετεί τα μηνύματα ενός topic σε ένα sub-topic που βασίζεται σε ένα κλειδί. Τα μηνύματα με το ίδιο κλειδί, τοποθετούνται πάντοτε στο ίδιο sub-topic, οπότε ο message broker χρειάζεται να εγγυηθεί μόνο την τήρηση της σειράς αποστολής των μηνυμάτων μέσα σε κάθε sub-topic. Αυτό γίνεται, ρυθμίζοντας κατάλληλα τις παραμέτρους ενός microservice, ώστε να υπάρχει ένα consumer instance που να καταναλώνει μηνύματα από ένα sub-topic μέσα στο consumer group. Αυξάνοντας στη συνέχεια τα sub-topics (partitions), επιτρέπουμε την αύξηση των στιγμιοτύπων του microservice ώστε να βελτιωθεί η απόδοση του συστήματος χωρίς να χαθεί η σειρά παράδοσης των μηνυμάτων.



Σχήμα 15: Keys – SubTopics

RabbitMQ

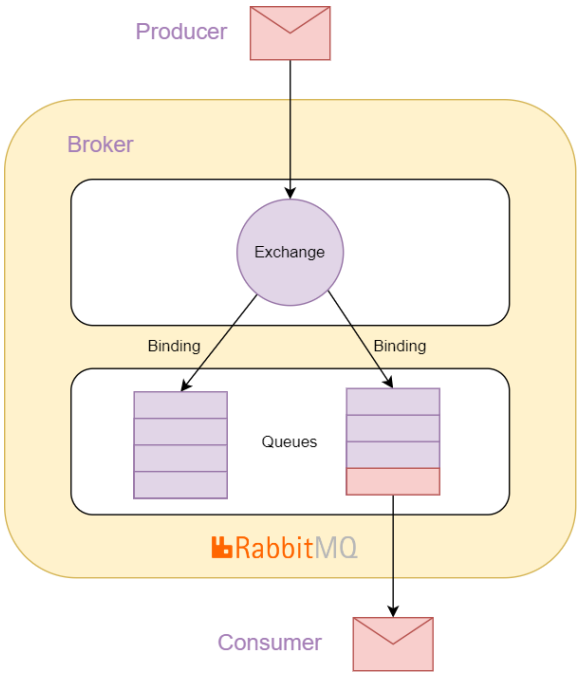
Για τις ανάγκες της εργασίας, ο message broker που χρησιμοποιήθηκε είναι το RabbitMQ. Πρόκειται για ένα message-queueing λογισμικό ανοικτού κώδικα που αρχικά υλοποίησε το πρωτόκολλο Advanced Message Queueing (AMQP), αλλά στη συνέχεια επεκτάθηκε ώστε να υποστηρίζει πρωτόκολλα όπως το Stream Text Oriented Protocol (STOMP), MQ Telemetry Transport κ.α. Ο διακομιστής RabbitMQ (server) έχει αναπτυχθεί σε γλώσσα προγραμματισμού Erlang και στο framework Open Telecom Platform, ώστε να υποστηρίζει clustering και failover. Οι RabbitMQ client βιβλιοθήκες που έχουν αναπτυχθεί για την επικοινωνία με τον διακομιστή, είναι διαθέσιμες σε όλες τις διαδεδομένες γλώσσες προγραμματισμού όπως Java, C#, Python, JavaScript, Ruby, PHP, Go κ.α.

Τα μηνύματα δεν καταχωρούνται απευθείας σε μια ουρά στον διακομιστή του RabbitMQ. Αρχικά ο producer αποστέλλει ένα μήνυμα σε ένα exchange, το exchange με τη σειρά του, είναι υπεύθυνο να

δρομολογήσει το μήνυμα σε διαφορετικές ουρές με τη βοήθεια του binding και των routing keys. Το binding είναι ένας σύνδεσμος ανάμεσα σε μια ουρά (queue) και ένα exchange.

Η ροή μηνυμάτων στο RabbitMQ είναι η εξής:

1. Ο producer δημοσιοποιεί ένα μήνυμα σε ένα exchange. Όταν δημιουργείται ένα exchange είναι απαραίτητο να οριστεί ο τύπος του.
2. Το exchange παραλαμβάνει το μήνυμα και είναι υπεύθυνο να το δρομολογήσει, λαμβάνοντας υπόψη τις διαφορετικές παραμέτρους των μηνυμάτων, όπως το routing key που εξαρτάται από τον τύπο του exchange.
3. Το binding θα πρέπει να γίνεται από το exchange προς τις ουρές (queues). Για παράδειγμα, στο σχήμα 16 υπάρχουν δύο bindings προς δύο διαφορετικές ουρές που ξεκινούν από το exchange. Το exchange δρομολογεί τα μηνύματα λαμβάνοντας υπόψη τις ιδιότητες που φέρουν.
4. Τα μηνύματα παραμένουν σε μια ουρά ώσπου να διαχειριστούν από τον consumer.
5. Ο consumer διαχειρίζεται τα μηνύματα.



Σχήμα 16: RabbitMQ

Τύποι exchanges:

- **Direct**: Το μήνυμα δρομολογείται στις ουρές των οποίων το binding key ταιριάζει με αυτό του μηνύματος. Για παράδειγμα, αν μια ουρά είναι συνδεδεμένη με το exchange, με binding key = “key1”, τότε τα μηνύματα που φθάνουν στο exchange με binding key = “key1”, δρομολογούνται στην αντίστοιχη ουρά.
- **Fanout**: Το fanout exchange δρομολογεί τα μηνύματα σε όλες τις ουρές που είναι συνδεδεμένες με αυτό.
- **Topic**: Το topic exchange δρομολογεί τα μηνύματα ταιριάζοντας το routing key με κάποιο routing pattern που έχει οριστεί στο binding. Για παράδειγμα eu.gr.* ή us.#
- **Headers**: Το header exchange χρησιμοποιεί τα headers ενός μηνύματος για τη δρομολόγηση.

Spring Cloud

Το Spring Cloud είναι ένα framework που διευκολύνει την ανάπτυξη εφαρμογών παρέχοντας λύσεις σε πολλές από τις δυσκολίες που συναντώνται κατά τη μετάβαση από μια μονολιθική εφαρμογή σε κατανεμημένο περιβάλλον μικροϋπηρεσιών. Οι εφαρμογές που βασίζονται σε αρχιτεκτονική μικροϋπηρεσιών στοχεύουν στην απλοποίηση της ανάπτυξης, της εγκατάστασης και της συντήρησης τους. Οι επιμέρους ανεξάρτητες υπηρεσίες της εφαρμογής επιτρέπουν στους προγραμματιστές να εστιάζουν σε ένα πρόβλημα τη φορά. Επίσης, μπορούν να πραγματοποιηθούν βελτιώσεις χωρίς να επηρεαστούν άλλα μέρη του συστήματος.

Από την άλλη πλευρά, προκύπτουν διαφορετικές προκλήσεις όταν υιοθετούμε την προσέγγιση μικροϋπηρεσιών. Το Spring Cloud framework παρέχει τις κατάλληλες βιβλιοθήκες για να αντιμετωπιστούν κάποιες από τις δυσκολίες:

- Service discovery, δίνει τη δυνατότητα ελέγχου και καταγραφής των στιγμιοτύπων (instances) μικροϋπηρεσιών που βρίσκονται στο δίκτυο.
- Το Spring Cloud Gateway, δίνει τη δυνατότητα δρομολόγησης των αιτημάτων από εξωτερικά συστήματα ή front-end εφαρμογές προς τις κατάλληλες μικροϋπηρεσίες εντός του δικτύου (reverse-proxy), καθώς επίσης μπορεί να χρησιμοποιηθεί και για εξισορρόπηση φόρτου (load-balancing).
- Externalized configuration, δυνατότητα παραμετροποίησης των μικροϋπηρεσιών από απομακρυσμένο σημείο, που δε θα απαιτεί rebuild της εφαρμογής μετά τις αλλαγές των ρυθμίσεων.
- Distributed Tracing, δυνατότητα ανίχνευσης και καταγραφής της διαδρομής των αιτημάτων προκειμένου να εντοπιστούν σφάλματα και καθυστερήσεις στο χρόνο απόκρισης των στιγμιοτύπων των μικροϋπηρεσιών.

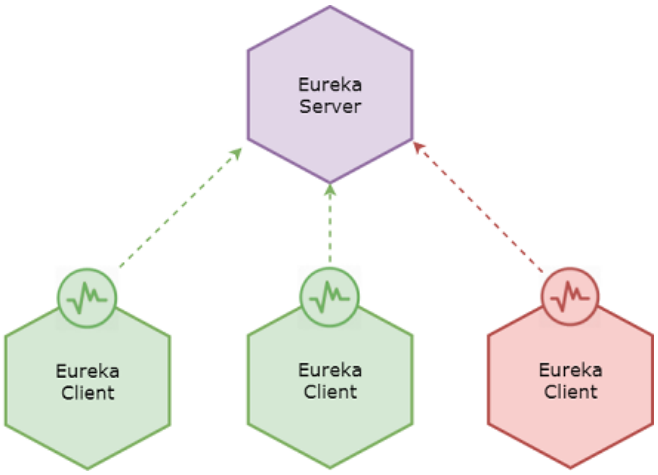
Spring Cloud Service Discovery – Eureka server

Το service discovery είναι μια από τις σημαντικότερες βιβλιοθήκες που προσφέρει το Spring Cloud framework, καθώς είναι απαραίτητο για τη δημιουργία ενός περιβάλλοντος μικροϋπηρεσιών που επικοινωνούν και συνεργάζονται μεταξύ τους.

Η υλοποίηση του discovery service που προσφέρει το Spring Cloud ονομάζεται Eureka server και σε συνεργασία με το Spring Cloud Gateway μπορεί να προσφέρει εξισορρόπηση φόρτου στο σύστημα (load-balancing). Ο Eureka server είναι μια εφαρμογή που αποθηκεύει προσωρινά τις πληροφορίες για όλα τα στιγμιότυπα των client-service εφαρμογών. Κάθε instance ενός microservice κατά την εκκίνηση του, εγγράφεται στον Eureka server και γνωστοποιεί το όνομα, την θύρα και τη διεύθυνση IP, ώστε να γνωρίζει ο server ανά πάσα στιγμή το πλήθος των στιγμιοτύπων που είναι διαθέσιμα και τις διευθύνσεις τους.

Στη συνέχεια, τα στιγμιότυπα των microservices στέλνουν αναφορά της κατάστασης τους στον Eureka Server, αυτή η διαδικασία ονομάζεται heartbeat. Το προεπιλεγμένο χρονικό διάστημα του heartbeat είναι 30 δευτερόλεπτα. Εάν ο Eureka server σταματήσει να λαμβάνει παλμό μετά από ορισμένο χρονικό διάστημα (η προεπιλογή είναι 90 δευτερόλεπτα), τότε αφαιρεί το instance του microservice από το μητρώο υπηρεσιών (service registry).

Για τη χρήση του Eureka Server απαιτείται εγκατάσταση των απαραίτητων dependencies μέσω Maven ή Gradle στον server και τους clients (eureka-server και eureka-client αντίστοιχα). Η παραμετροποίηση των microservices που συνδέονται στον Eureka Server μπορεί να γίνει εξολοκλήρου από τα configuration files του Spring Boot (application.yml ή application.properties).

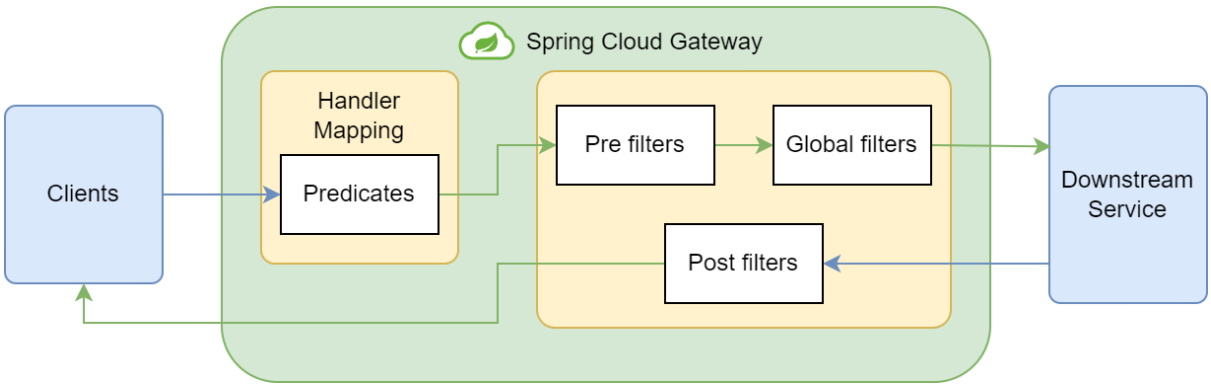


Σχήμα 17: Eureka Server

Spring Cloud Gateway

Το Spring Cloud Gateway είναι μια βιβλιοθήκη κατάλληλη για τη δημιουργία API Gateway, που καθιστά εύκολη τη δρομολόγηση αιτημάτων στα microservices εσωτερικά του δικτύου. Η δρομολόγηση των αιτημάτων βασίζεται σε κριτήρια, όπου αντιστοιχίζονται οι ιδιότητες ενός request σε προκαθορισμένες διαδρομές.

Το Spring Cloud Gateway βρίσκεται ανάμεσα σε μια εφαρμογή πελάτη (client) και σε ένα resource που ζητείται. Έχει τη δυνατότητα να παρεμποδίζει, να αναλύει και να τροποποιεί κάθε αίτημα. Αυτό σημαίνει ότι μπορεί να δρομολογήσει αιτήματα με βάση το περιεχόμενο τους, για παράδειγμα, αν υπάρχει κεφαλίδα (header) στο αίτημα που να υποδεικνύει μια συγκεκριμένη έκδοση του API, τότε το αίτημα μπορεί να δρομολογηθεί στο service με την κατάλληλη έκδοση.



Σχήμα 18: Gateway

Τα βασικά χαρακτηριστικά του Spring Cloud Gateway είναι:

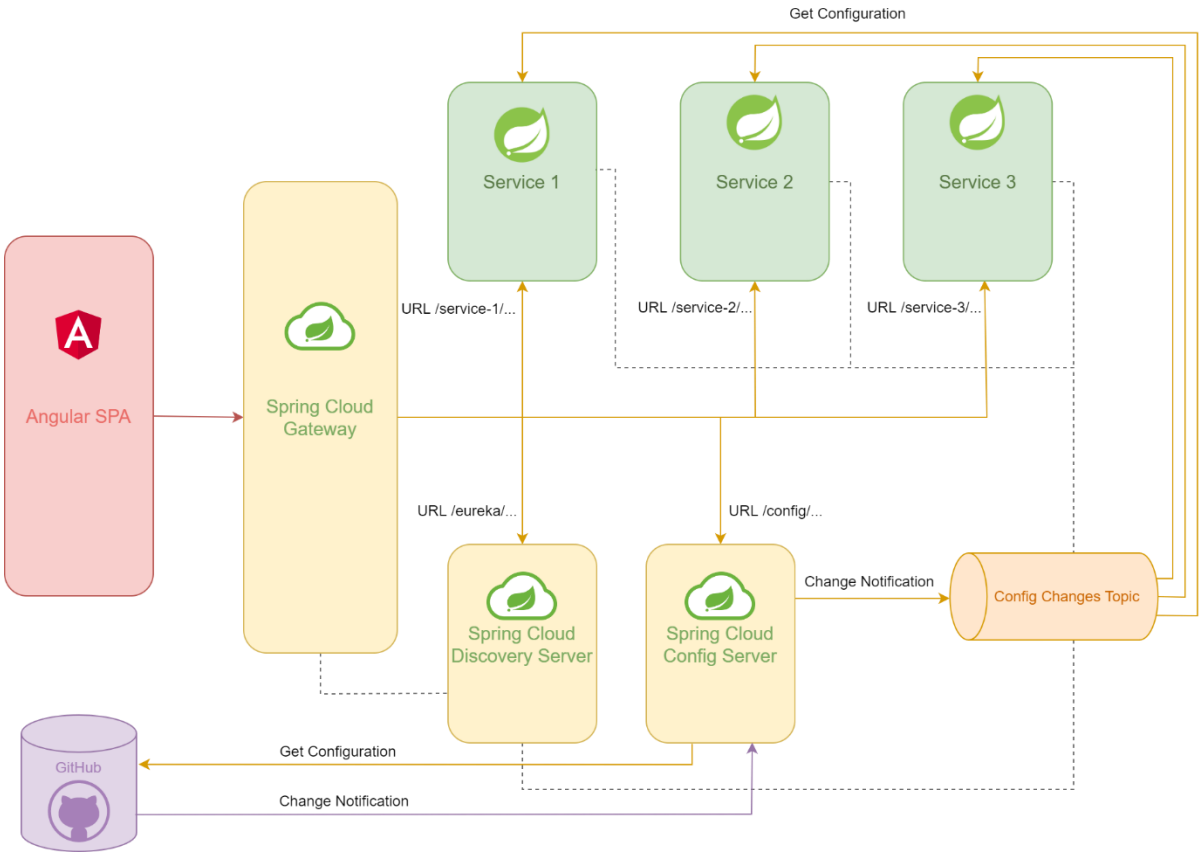
- Ο επανορισμός της διαδρομής ενός request.
- Εξισορρόπηση φόρτου (load balancing).
- Χρήση rate limiter στα αιτήματα που δέχεται η εφαρμογή μέσω του gateway.
- Χρήση φίλτρων και path matchers.
- Δυνατότητα χρήσης Circuit Breaker design pattern
- Υποστήριξη του project reactor για non-blocking (spring boot 2.0 και spring framework 5).

Spring Configuration Server

Όταν αυξάνεται το πλήθος των μικροϋπηρεσιών, αυξάνεται και το πλήθος των αρχείων ρυθμίσεων (configuration files) που θα πρέπει να διαχειρίζονται και να ενημερώνονται από τους προγραμματιστές. Το Spring Cloud Config δίνει τη δυνατότητα τοποθέτησης των αρχείων ρυθμίσεων σε ένα κεντρικό αποθετήριο, όπως το git, αποκτώντας έτσι version control system[16] και παραμετροποίηση των ρυθμίσεων εξ αποστάσεως (GitHub και GitLab κ.α.), χωρίς την ανάγκη re-deployment της εφαρμογής. Οι μικροϋπηρεσίες μπορούν να ενημερώνουν τις ρυθμίσεις τους μέσα από το αποθετήριο κατά τη διάρκεια της επανεκκίνησης.

Ένα ακόμη χαρακτηριστικό του Spring Cloud Server, είναι η υποστήριξη ανίχνευσης αλλαγών στα αρχεία ρυθμίσεων και η αποστολή ειδοποιήσεων στις μικροϋπηρεσίες που επηρεάζονται από τις αλλαγές, χρησιμοποιώντας το Spring Cloud Bus. Το Spring Cloud Bus είναι abstraction βασισμένο πάνω στο Spring Cloud Stream το οποίο υποστηρίζει την χρήση Apache Kafka και RabbitMQ για αποστολή μηνυμάτων στο message-driven programming.

Ο Spring Cloud Configuration server μπορεί να τοποθετηθεί πίσω από το Spring Cloud Gateway, όπως ο discovery server και οι υπόλοιπες μικροϋπηρεσίες της εφαρμογής (σχήμα 19).



Σχήμα 19: Spring Cloud Gateway, Discovery and Configuration Servers

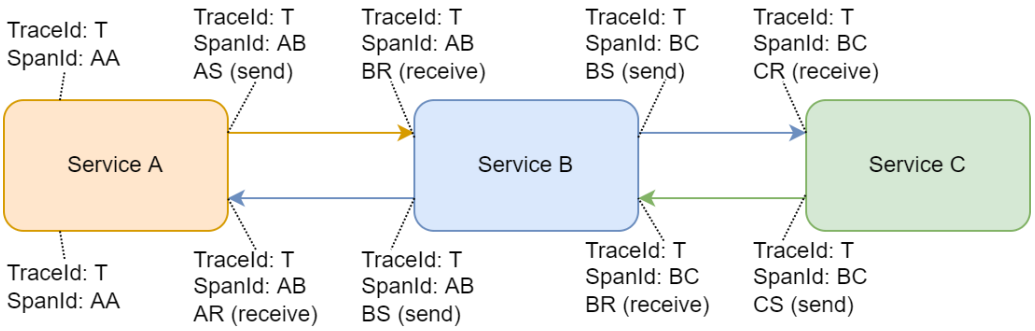
[16] Version Control είναι η πρακτική παρακολούθησης και διαχείρισης των αλλαγών στον πηγαίο κώδικα μιας εφαρμογής. Το VCS (Version Control System) είναι ένα εργαλείο λογισμικού που βοηθά τους προγραμματιστές να διαχειρίζονται αλλαγές στον πηγαίο κώδικα με την πάροδο του χρόνου.

Distributed Tracing – Spring Cloud Sleuth and Zipkin

Σε ένα κατανεμημένο σύστημα είναι πολύ σημαντικό να υπάρχει δυνατότητα ανίχνευσης (tracing) και οπτικοποίησης της ροής ενός request ή ενός ασύγχρονου μηνύματος μέσα από τις μικροϋπηρεσίες κατά την επεξεργασία του. Το distributed tracing είναι χρήσιμο για τον εντοπισμό σφαλμάτων όταν εμπλέκονται πολλά συστήματα στη ροή της επεξεργασίας ενός αιτήματος και η εφαρμογή δεν ανταποκρίνεται στον αναμενόμενο χρόνο. Σε τέτοιες περιπτώσεις, πρέπει πρώτα να αναγνωριστεί το υποσύστημα που καθυστερεί να ανταποκριθεί. Μόλις εντοπιστεί, μπορούμε να εργαστούμε προκειμένου να επιλύσουμε το επί μέρους πρόβλημα. Το Spring Cloud Sleuth και το Zipkin service είναι ευρέως διαδεδομένα και εξυπηρετούν τον παραπάνω σκοπό.

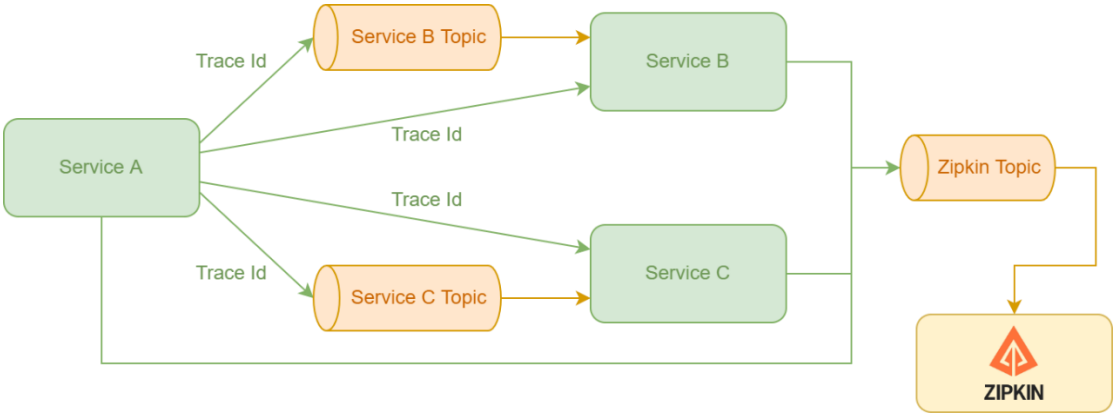
Το Spring Cloud Sleuth προσθέτει την παράμετρο **correlation ID** σε ένα request ή μήνυμα, ώστε να είναι εφικτό στη συνέχεια, να αναγνωριστούν όλα τα κομμάτια της ροής επεξεργασίας που συνοδεύονται από το ίδιο correlation ID. Επίσης, το Sleuth μπορεί να προσθέσει την παράμετρο correlation ID σε όλα τα logs που έχουν καταγραφεί σε διαφορετικά microservices που προέχονται από την ίδια ροή επεξεργασίας. Το Zipkin είναι ένα κατανεμημένο σύστημα ανίχνευσης, όπου το Spring Cloud Sleuth μπορεί να αποστείλει δεδομένα που αφορούν την ανίχνευση των requests και των μηνυμάτων, ώστε να αποθηκευτούν και να οπτικοποιηθούν.

Το Spring Cloud Sleuth και Zipkin είναι βασισμένα στο Google Dapper το οποίο χρησιμοποιεί trace tree προκειμένου να αναζητήσει πληροφορίες ανίχνευσης (tracing). Τα υποδένδρα του trace tree ονομάζονται spans και τα spans με τη σειρά τους αποτελούνται από sub-spans. Το Correlation ID ονομάζεται **TraceId** και κάθε span περιέχει μοναδικό **SpanId** που συνοδεύεται από το TraceId του trace tree στο οποίο ανήκει.



Σχήμα 20: Sleuth-Zipkin

Το Sleuth μπορεί να στείλει αιτήματα στο Zipkin, είτε σύγχρονα με http requests, είτε ασύγχρονα χρησιμοποιώντας RabbitMQ ή Apache Kafka. Προκειμένου να μην υπάρχει εξάρτηση του Zipkin server με τις μικροϋπηρεσίες του συστήματος, προτείνεται η ασύγχρονη μέθοδος αποστολής δεδομένων ανίχνευσης στο Zipkin μέσω ενός message broker.



Σχήμα 21: Async Tracing Information

Microservices Security

Υπάρχουν πολλά πλεονεκτήματα στην κατασκευή μιας εφαρμογής σε αρχιτεκτονική microservices, ωστόσο, μια από τις προκλήσεις που προκύπτουν είναι το θέμα της ασφάλειας το οποίο διαφέρει από μια παραδοσιακή μονολιθική εφαρμογή. Συγκεκριμένα, μια εφαρμογή μικροϋπηρεσιών μπορεί να αποτελείται από δεκάδες ή και εκατοντάδες στοιχεία, καθένα από τα οποία αποτελεί πιθανό στόχο επίθεσης και μπορεί να έχει ιδιαίτερα τρωτά σημεία και θέματα που εγείρουν ανησυχίες για την ασφάλεια.

Όταν μια εφαρμογή μικροϋπηρεσιών βρίσκεται σε περιβάλλον παραγωγής, πρέπει να λαμβάνονται μέτρα για την εξασφάλιση κάθε στοιχείου από κινδύνους, συμπεριλαμβανομένων και στοιχείων που μεσολαβούν για την επικοινωνία των μικροϋπηρεσιών, όπως οι message brokers. Η ασφάλεια των μικροϋπηρεσιών είναι εξαιρετικά περίπλοκη και πρέπει να ληφθεί υπόψη από τα αρχικά στάδια ανάπτυξης της εφαρμογής.

Το απλούστερο σενάριο υλοποίησης ταυτοποίησης και εξουσιοδοτημένης πρόσβασης χρηστών (authentication[17] and authorization[18]) στις μικροϋπηρεσίες είναι μέσω του API Gateway. Το API Gateway αποτελεί το σημείο εισόδου των αιτημάτων από εξωτερικές εφαρμογές και ιστοσελίδες, οπότε δίνεται η δυνατότητα αξιοποίησης του, ως σύστημα ταυτοποίησης και εξουσιοδότησης για τις μικροϋπηρεσίες που βρίσκονται τοποθετημένες πίσω από το API Gateway. Με τον τρόπο αυτό, εξαλείφεται η ανάγκη ταυτοποίησης σε κάθε μια από τις μικροϋπηρεσίες ξεχωριστά.

Η ασφάλεια σε επίπεδο API Gateway ονομάζεται και περιφερειακή ασφάλεια, γιατί προστατεύει περιμετρικά το οικοσύστημα των μικροϋπηρεσιών και έχει αρκετούς περιορισμούς:

- Αν ένας κακόβουλος χρήστης αποκτήσει πρόσβαση στο εσωτερικό δίκτυο των μικροϋπηρεσιών, τότε όλα τα υποσυστήματα εντός του δικτύου θα είναι εκτεθειμένα.
- Σε πολύπλοκα οικοσυστήματα μικροϋπηρεσιών, η εξουσιοδότηση πρόσβασης (authorization) γίνεται ιδιαίτερα πολύπλοκη λόγω του πλήθους των ρόλων και των κανόνων πρόσβασης.

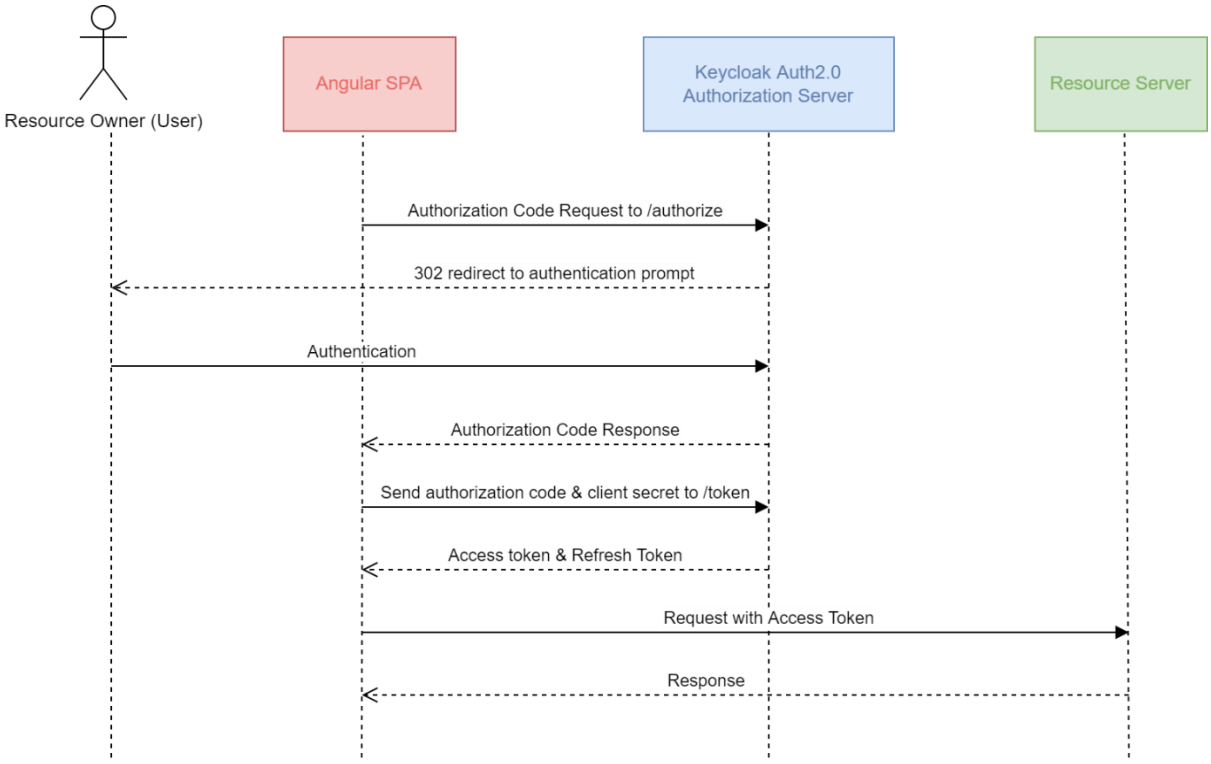
[17] Authentication είναι η αναγνώριση της ταυτότητας ενός χρήστη, επικυρώνοντας τα στοιχεία που εισάγει κατά την είσοδο του στην εφαρμογή (username, password ή access token).

[18] Authorization είναι τα δικαιώματα πρόσβασης σε μέρη της εφαρμογής που αποκτά ενός authenticated χρήστης, ανάλογα με τους ρόλους που διαθέτει.

- Επιβάρυνση του API Gateway με ενέργειες authentication και authorization. Ο σκοπός ενός API Gateway είναι να δρομολογεί τα αιτήματα στις κατάλληλές μικροϋπηρεσίες επαναορίζοντας τη διαδρομή. Επίσης βασικός σκοπός του είναι να λειτουργεί ως loadbalancer και rate limiter.
- Οι ομάδες που διαχειρίζονται το API Gateway (operation teams) είναι διαφορετικές από τις ομάδες ανάπτυξης των microservices (development teams), αυτό σημαίνει, πως δεν είναι πάντα εφικτό για τους προγραμματιστές, να κάνουν απευθείας αλλαγές στους κανόνες που αφορούν το authorization των χρηστών.

Από τους παραπάνω περιορισμούς μπορούμε να συμπεράνουμε πως η χρήση του API Gateway για authorization και authentication, ενώ αποτελεί μια λύση, δεν αποτελεί βέλτιστη πρακτική. Όπως επίσης δεν αποτελεί βέλτιστη πρακτική, η χρήση κάθε μικροϋπηρεσίες ξεχωριστά για την αναγνώριση της ταυτότητας και της εξουσιοδοτημένης πρόσβασης των χρηστών (Service-level authorization). Ο λόγος που δε προτείνεται το service-level-authorization είναι γιατί δημιουργεί άσκοπη κίνηση εντός του δικτύου των μικροϋπηρεσιών, πραγματοποιώντας πλήθος περιττών κλήσεων που εξυπηρετούν τον ίδιο σκοπό.

Μια από τις βέλτιστες λύση στο θέμα της ασφάλειας σε περιβάλλον μικροϋπηρεσιών, αποτελεί η χρήση **Authorization Server** και **Token Relay** pattern με χρήση του ανοικτού προτύπου **OAuth2.0** και **OpenID Connect**.



Σχήμα 22: OAuth2.0 – Sequence Diagram

Όταν ο χρήστης εισέρχεται στην εφαρμογή, ανακατευθύνεται στον Authorization Server προκειμένου να συνδεθεί, χρησιμοποιώντας τα στοιχεία του (username και password). Αν τα στοιχεία είναι έγκυρα, τότε λαμβάνει ένα κρυπτογραφημένο μήνυμα σε μορφή JSON που

ονομάζεται access token (JWT[19]) και στη συνέχεια ανακατευθύνεται στην εφαρμογή. Κάθε αίτημα που πραγματοποιείται στο backend της εφαρμογής, περιέχει το access token σε ένα Authorization header που ονομάζεται Bearer. Όταν το αίτημα φτάσει στο API Gateway, το gateway επικυρώνει το access token μέσω του Authorization Server και αν είναι έγκυρο, τότε επαναορίζει τη διαδρομή του αιτήματος, περνώντας το access token στο νέο αίτημα που δρομολογείται προς τις μικροϋπηρεσίες. Όταν το αίτημα φτάσει στην μικροϋπηρεσία που απευθύνεται, τότε και εκείνη με τη σειρά της, επικυρώνει το access token μέσω του Authorization Server. Με τον τρόπο αυτό, διασφαλίζεται κάθε στοιχείο στο εσωτερικό του δικτύου με το ελάχιστο δυνατό overhead στις κλήσεις που πραγματοποιούνται.

OAuth 2.0 και OpenID

Το OAuth 2.0 είναι ένα ευρέως διαδεδομένο ανοικτό πρότυπο για authorization που επιτρέπει σε έναν χρήστη να δώσει τη συγκατάθεση του, ώστε μια τρίτη (third-party) εφαρμογή να αποκτήσει πρόσβαση σε προστατευμένους πόρους του χρήστη, όπως το όνομα του. Η παραχώρηση του δικαιώματος σε μια τρίτη εφαρμογή να ενεργεί εκ μέρους του χρήστη, για παράδειγμα να καλεί ένα API, ονομάζεται **authorization delegation.**

Όταν μια εφαρμογή, όπως ένα single page application (client application), καταχωρείται στον Authorization Server, τότε αποκτά **clientId** και **client secret**. Το client secret θα πρέπει να παραμένει προστατευμένο όπως ένα password. Μαζί με την εφαρμογή, καταχωρούνται και URIs ανακατεύθυνσης (redirect URIs) προκειμένου να χρησιμοποιηθούν από τον Authorization Server μετά την επικύρωση των στοιχείων του χρήστη, για να αποσταλεί το authorization code και τα tokens που εκδόθηκαν, πίσω στην εφαρμογή.

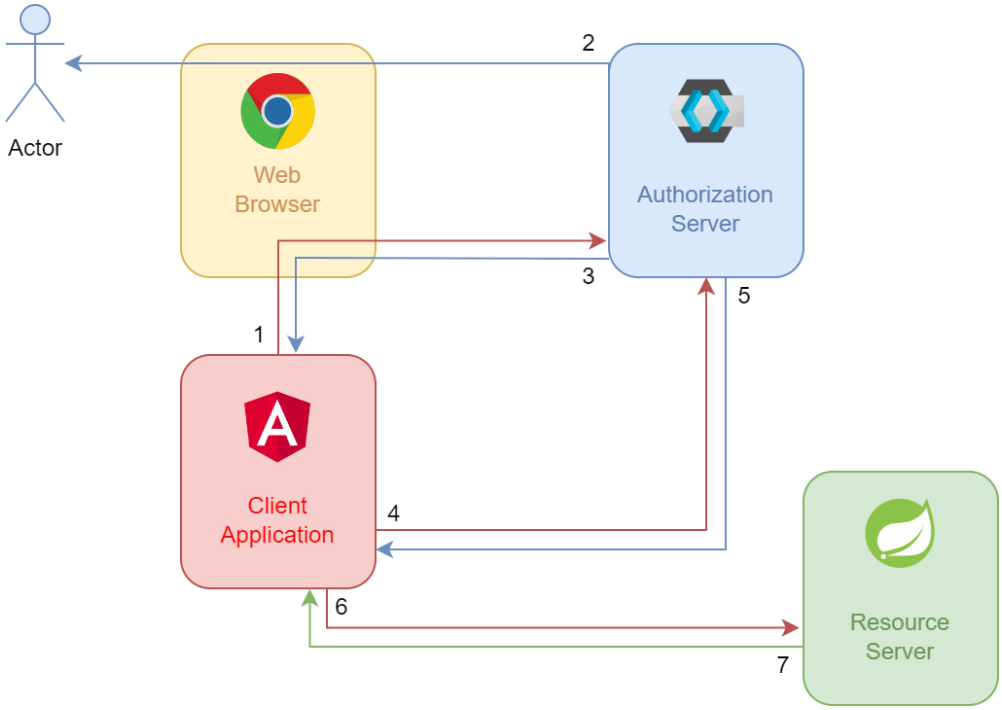
Ας υποθέσουμε πως ένας χρήστης έχει πρόσβαση σε μια web εφαρμογή και η εφαρμογή πρέπει να καλέσει ένα προστατευμένο API ώστε να εξυπηρετήσει τον χρήστη. Προκειμένου να επιτραπεί η πρόσβαση στο προστατευμένο API, η εφαρμογή (client application) θα πρέπει να δείξει στο API, ότι ενεργεί εκ μέρους του χρήστη. Για να αποφευχθεί μια λύση, όπου ο χρήστης θα πρέπει να μοιραστεί τα login credentials με την client εφαρμογή για την αυθεντικοποίηση, ο Authentication Server εκδίδει ένα access token που δίνει το δικαίωμα στην client εφαρμογή να έχει περιορισμένη πρόσβαση σε συγκεκριμένα στοιχεία του χρήστη. Αυτό σημαίνει πως ο χρήστης, δε είναι αναγκασμένος να αποκαλύψει τα login credentials στην client εφαρμογή. Ο χρήστης μπορεί να δώσει τη συγκατάθεση του στην client εφαρμογή, ώστε εκείνη να αποκτήσει πρόσβαση σε προκαθορισμένες πληροφορίες, για παράδειγμα το ονοματεπώνυμο και την ηλεκτρονική διεύθυνση του.

Ένα **access token** λήγει σε σύντομο χρονικό διάστημα και περιέχει πληροφορίες του χρήστη, καθώς και δικαιώματα πρόσβασης που αναφέρονται ως **scopes** σύμφωνα με το OAuth 2.0. Προκειμένου να μην ανακατευθύνεται ο χρήστης στον Authentication Server για επανασύνδεση κάθε φορά που λήγει ένα access token, κατά την πρώτη σύνδεση, μαζί με το access token, εκδίδεται και ένα **refresh token** το οποίο αποθηκεύεται στην client εφαρμογή. Το refresh token έχει μεγάλη χρονική διάρκεια και χρησιμοποιείται από την client εφαρμογή ώστε να ανανεώνει αυτόματα το access token χωρίς να εμπλέκει τον χρήστη.

[19] Το JSON Web Token (JWT) είναι ένα ανοιχτό πρότυπο (RFC 7519) που ορίζει έναν συμπαγή και αυτόνομο τρόπο για την ασφαλή μετάδοση πληροφοριών μεταξύ συστημάτων ως αντικείμενα JSON. Αυτές οι πληροφορίες μπορούν να επαληθευτούν και να είναι αξιόπιστες επειδή είναι ψηφιακά υπογεγραμμένες. Τα JWT μπορούν να υπογραφούν χρησιμοποιώντας ένα secret (με τον αλγόριθμο HMAC) ή ένα ζεύγος δημόσιου/ιδιωτικού κλειδιού χρησιμοποιώντας RSA ή ECDSA. Πηγή: jwt.io

Το πρότυπο OAuth 2.0 ορίζει τέσσερις διαφορετικές ροές για την έκδοση των access tokens:

Authorization code grant flow: Πρόκειται για την ασφαλέστερη ροή αλλά και την πιο πολύπλοκη. Ο χρήστης αλληλεπιδρά με τον Authorization Server μέσω του web browser, προκειμένου να προχωρήσει σε αυθεντικοποίηση και συναίνεση για τις προσβάσεις σε πόρους που ζητούνται από την εφαρμογή (client application).



Σχήμα 23: OAuth2.0 - Authorization grant flow

1. Η εφαρμογή (client application) ανακατευθύνει τον χρήστη στον authorization server μέσω του web browser.
2. Ο authorization server προχωρά σε αυθεντικοποίηση του χρήστη και στη συνέχεια ζητά τη συναίνεση του, προκειμένου να παραχωρήσει το δικαίωμα στο client application να έχει πρόσβαση σε ορισμένα από τα στοιχεία του χρήστη.
3. Ο authorization server ανακατευθύνει τον χρήστη πίσω στο client application, χρησιμοποιώντας το redirect URI που έχει δηλωθεί στο βήμα 1 από την εφαρμογή, ώστε να γνωρίζει ο authorization server που θα στείλει το authorization code.
4. Η εφαρμογή (client application) καλεί τον authorization server προκειμένου να ανταλλάξει το authorization code που έχει λάβει με ένα access token. Η εφαρμογή πρέπει να στείλει το clientId, το client secret και το authorization code ώστε να λάβει το access token. Η παραπάνω ενέργεια πρέπει να εκτελεστεί από server-side code του client application, διότι περιέχει το client secret που αποτελεί ευαίσθητη πληροφορία.
5. Ο authorization server εκδίδει ένα access token και το στέλνει πίσω στο client application. Ο authorization server εκδίδει και ένα refresh token προαιρετικά.
6. Η εφαρμογή (client application) χρησιμοποιώντας το access token, αποστέλλει αίτημα στο προστατευμένο API που έχει εκτεθεί από τον resource server.
7. Ο resource server επικυρώνει το access token και αν είναι έγκυρο, τότε εξυπηρετεί το αίτημα. Τα βήματα 6 και 7 μπορούν να επαναλαμβάνονται για όσο χρόνο είναι έγκυρο το access token. Όταν λήξει, το client application μπορεί να χρησιμοποιήσει το refresh token για να λάβει νέο access token μέσω το authorization server.

Implicit grant flow: Η implicit ροή βασίζεται επίσης στον web browser, αλλά προορίζεται για client application τα οποία δε μπορούν να κρατήσουν το client secret ασφαλές, για παράδειγμα ένα single page web application. Ο web browser λαμβάνει ένα access token από τον authorization server αντί για authorization code. Εφόσον η implicit ροή είναι λιγότερο ασφαλής από τη ροή authorization code, το client application δε μπορεί να ζητήσει refresh token.

Resource owner password credentials grant flow: Αν ένα client application δεν έχει τη δυνατότητα να αλληλεπιδράσει με ένα web browser, μπορεί να χρησιμοποιήσει αυτή την ροή. Στη ροή αυτή, ο χρήστης πρέπει να μοιραστεί τα login credentials με το client application, ώστε η εφαρμογή να μπορέσει να λάβει ένα access token.

Client credentials grant flow: Στην περίπτωση που ένα client application χρειάζεται να καλέσει ένα API, χωρίς αυτή η κλήση να είναι μέρος ενός αιτήματος που προέρχεται από χρήστη, τότε πρέπει να χρησιμοποιηθεί η συγκεκριμένη ροή, ώστε να λάβει από μόνος του ο client ένα access token χρησιμοποιώντας δικό το clientId και client secret.

Το πρότυπο OAuth 2.0 δημοσιεύτηκε το 2012, από τότε έχει φανεί ποιες από τις πρακτικές που χρησιμοποιεί λειτουργούν καλύτερα και ποιες λιγότερο καλά. Το 2019, ξεκίνησαν οι εργασίες ώστε να καθιερωθεί το πρότυπο OAuth 2.1, το οποίο συγκεντρώνει όλες τις βέλτιστες πρακτικές που έχουν προκύψει από την εμπειρία χρήσης του OAuth 2.0.

Κάποιες από τις σημαντικές βελτιώσεις του OAuth 2.1 είναι:

- Το PKCE (Proof Key for Code Exchange), πρόκειται για μια προσθήκη στη ροή «authorization code grant flow»**,** όπου απαιτεί από τους δημόσιους clients (client applications) να βελτιώσουν την ασφάλεια τους χρησιμοποιώντας ένα code challenge (RFC 7636). Για τους ιδιωτικούς clients, η χρήση PKCE θα προτείνεται αλλά δε θα απαιτείται.
- Κατάργηση της ροής «implicit grant flow» και αφαίρεση από το πρότυπο, λόγω της περιορισμένης ασφάλειας που προσφέρει.
- Κατάργηση της ροής «resource owner password credentials grant flow» και αφαίρεση από το πρότυπο, επίσης λόγω της περιορισμένης ασφάλειας που προσφέρει.

Στο πλαίσιο της παρούσας εργασίες, έχουν χρησιμοποιηθεί μόνο οι ροές **authorization code grant flow** και **client credentials grant flow.**

OpenID Connect

Το OpenID Connect (OIDC) είναι ένα πρότυπο που συμπληρώνει το OAuth 2.0 και επιτρέπει στις εφαρμογές-πελάτες (client applications) να επαληθεύουν την ταυτότητα των χρηστών. Το OIDC προσθέτει ένα επιπλέον token, που ονομάζεται ID token, το οποίο λαμβάνει η εφαρμογή από τον authorization server μαζί με το access token, όταν ολοκληρωθεί επιτυχώς η ροή αυθεντικοποίησης του χρήστη.

Το ID token είναι κωδικοποιημένο ως JSON Web Token (JWT) και περιέχει στοιχεία, όπως το id και το email του χρήστη. Το ID token περιέχει ψηφιακή υπογραφή με χρήση των JSON web signatures. Με τον τρόπο αυτό, δίνεται η δυνατότητα στο client application να θεωρήσει αξιόπιστες τις πληροφορίες που λαμβάνει από το ID token, επαληθεύοντας την ψηφιακή υπογραφή με το δημόσιο κλειδί (public key) που παρέχει ο authentication server.

Τα access tokens, μπορούν να κωδικοποιηθούν και να υπογραφούν με τον ίδιο τρόπο όπως τα id tokens, αλλά δεν απαιτείται σύμφωνα με το πρότυπο. Τέλος, το OIDC ορίζει ένα discovery endpoint που περιέχει μια λίστα με τα σημαντικότερα URIs που εξυπηρετούν τις ροές του OAuth 2.0, όπως authorization_endpoint, token_endpoint, issuer_endpoint και userinfo_endpoint.

Keycloak Authorization Server

Το Keycloak είναι ένα εργαλείο ανοιχτού κώδικα για τη διαχείριση της ταυτοποίησης και πρόσβασης χρηστών. Είναι βασισμένο πάνω σε industry standards όπως το OAuth 2.0, το OpenID Connect και SAML 2.0. Είναι εστιασμένο σε σύγχρονες εφαρμογές όπως τα Single Page Applications, εφαρμογές κινητών και REST APIs. Το project έκανε την πρώτη του εμφάνιση το 2014, έκτοτε εξελίχθηκε σε ένα ευρέως διαδεδομένο εργαλείο ανοιχτού κώδικα, που χρησιμοποιείται από μικρές και μεγάλες επιχειρήσεις και έχει ισχυρή υποστήριξη από την κοινότητα χρηστών.

Ο Keycloak authorization server παρέχει πλήρως παραμετροποιήσιμες σελίδες εισόδου και σύνδεσης χρηστών (login pages), λειτουργία ανάκτησης κωδικών πρόσβασης, χρήση SSO (Single Sign On) και πολλά ακόμη χαρακτηριστικά, ενώ μπορεί να ενσωματωθεί σε μια εφαρμογή με σχετικά απλό τρόπο. Αναθέτοντας την ταυτοποίηση χρηστών στο Keycloak, απαλλάσσεται η ομάδα ανάπτυξης μιας εφαρμογής από προκλήσεις, όπως οι μηχανισμοί ελέγχου ταυτότητας και αποθήκευσης κωδικών πρόσβασης με ασφάλεια. Παρέχεται επίσης, η δυνατότητα ταυτοποίησης χρηστών με τη μέθοδο δυο παραγόντων (two factor authentication), χωρίς να χρειάζεται να γίνουν αλλαγές στην εφαρμογή. Με την χρήση authorization server, όπως έχει αναφερθεί, η εφαρμογή γίνεται πιο ασφαλής διότι δεν διαθέτει πρόσβαση στα διαπιστευτήρια του χρήστη, παρά μόνο στις πληροφορίες που ο ίδιος ο χρήστης έχει συναινέσει να παρέχει.

Η διαχείριση του Keycloak authorization server, πραγματοποιείται μέσω του Keycloak Admin Console UI από τους προγραμματιστές και τους διαχειριστές της εφαρμογής. Οι ρυθμίσεις του Keycloak διακρίνονται στα εξής σημεία:

Realm: Ο Keycloak authorization server μπορεί να διαχειριστεί την ταυτοποίηση χρηστών πολλών εφαρμογών ταυτόχρονα. Η δημιουργία ενός realm είναι η δημιουργία κατηγορίας που αφορά τη διαχείριση μιας και μόνο εφαρμογής. Κάθε realm είναι εντελώς απομονωμένο από τα υπόλοιπα realms όσον αφορά τις ρυθμίσεις, τους χρήστες και τους ρόλους. Για παράδειγμα, θα μπορούσε να δημιουργηθεί ένα realm για μια εσωτερική εφαρμογή διαχείρισης των υπαλλήλων μιας εταιρίας και ένα realm εξωτερικής εφαρμογής που αφορά τους πελάτες της εταιρίας.

Client: Οι clients είναι οι οντότητες που μπορούν να ζητήσουν από το Keycloak, τον έλεγχο της ταυτότητας ενός χρήστη. Τις περισσότερες φορές, οι clients είναι web εφαρμογές, εφαρμογές κινητών συσκευών και native εφαρμογές που χρησιμοποιούν το Keycloak ως μια ενιαία λύση για την ταυτοποίηση των χρηστών τους. Clients μπορούν να αποτελέσουν επίσης, υπηρεσίες όπως REST APIs, gRPC και WebSocket που χρειάζονται πληροφορίες ταυτοποίησης ή access token, προκειμένου να καλέσουν με ασφάλεια άλλες υπηρεσίες που ανήκουν στο δίκτυο που προστατεύεται από το Keycloak.

Client Scope: Επιτρέπει τη δημιουργία ομάδων από στοιχεία χρηστών (claims), που προστίθενται στα tokens που εκδίδονται προς ένα client.

Roles: Δημιουργία ρόλων χρηστών που αφορούν στα δικαιώματα που θα έχουν οι χρήστες στο πλαίσιο της εφαρμογής. Για παράδειγμα, μπορεί να εκχωρηθεί ρόλος διαχειριστή (Admin) σε μια ομάδα χρηστών, αυτό σημαίνει πως οι συγκεκριμένοι χρήστες θα έχουν πρόσβαση σε όλους τους πόρους της εφαρμογής. Υπάρχει επίσης η δυνατότητα δημιουργίας σύνθετων ρόλων (composite roles), δηλαδή ρόλων που εμπεριέχουν άλλους ρόλους.

Identity Providers: Επιτρέπει στους χρήστες να ταυτοποιούνται στο Keycloak μέσω εξωτερικών παρόχων ταυτοποίησης ή μέσω κοινωνικών δικτύων. Το Keycloak έχει ενσωματωμένη υποστήριξη για OpenID Connect και SAML 2.0, καθώς και για κοινωνικά δίκτυα, όπως το Google, το Facebook, το GitHub και το Twitter.

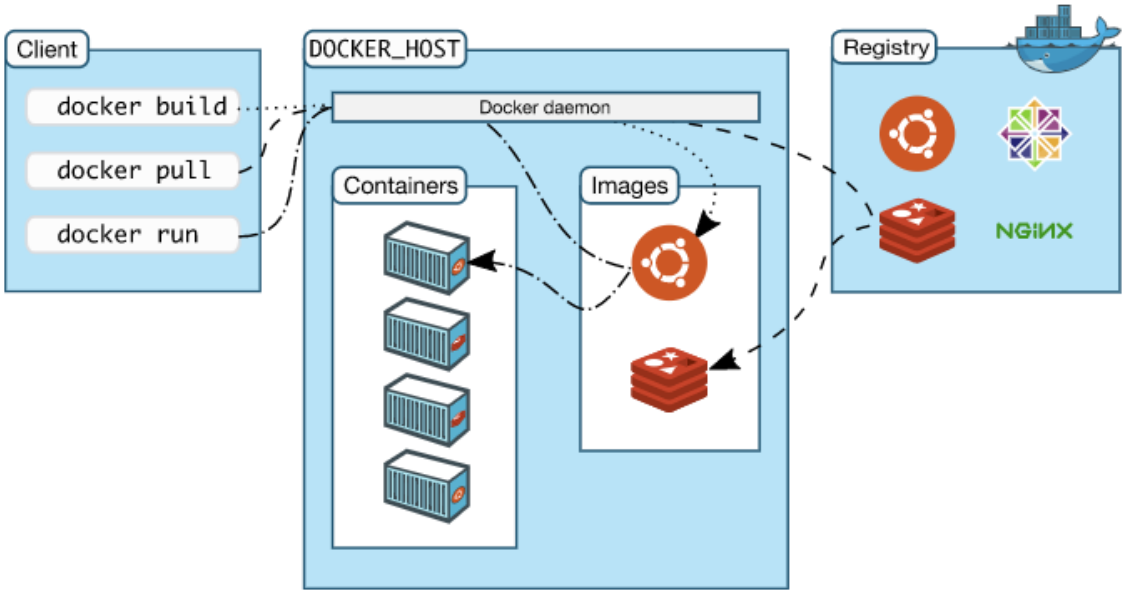
User federation: Αναφέρεται στη δυνατότητα ενσωμάτωσης του Keycloak με εξωτερικά identity stores, όπως το LDAP και το Kerberos.

Authentication: Αναφέρεται στα διαδοχικά βήματα προκειμένου να ταυτοποιηθεί ένας χρήστης. Τα βήματα είναι ομαδοποιημένα σε κατηγορίες και μπορούν να τροποποιηθούν ανάλογα με τις ανάγκες της εφαρμογής.

Η εγκατάσταση του Keycloak μπορεί να γίνει εύκολα με την τεχνολογία Docker σε docker-container και να συνδεθεί με κατάλληλες ρυθμίσεις του Dockerfile ή docker compose, με μια από τις υποστηριζόμενες βάσεις δεδομένων όπως η PostgreSQL. Η PostgreSQL είναι μια υψηλής ποιότητας, object-related βάση δεδομένων ανοικτού κώδικα και υποστηρίζεται από πολλούς cloud providers και cloud τεχνολογίες, όπως το Azure Database for PostgreSQL, η Amazon RDS for PostgreSQL, Cloud SQL for PostgreSQL και Crunch PostgreSQL for Kubernetes.

Docker containers

Docker και Docker Containers: Το Docker είναι μια ανοιχτή πλατφόρμα ανάπτυξης, διανομής και εκτέλεσης εφαρμογών που επιτρέπει τον διαχωρισμό μιας εφαρμογής από την υποδομή. Μπορούμε να εκτελέσουμε μια εφαρμογή ανεξάρτητα από το λειτουργικό σύστημα που είναι εγκατεστημένο στον host υπολογιστή. Μας παρέχει τη δυνατότητα να κατασκευάσουμε και να εκτελέσουμε μια εφαρμογή σε ένα απομονωμένο, ασφαλές και ελαφρύ περιβάλλον που ονομάζεται Container. Έχουμε τη δυνατότητα να συνδυάσουμε πολλά Containers, όπου το κάθε ένα περιέχει όλα τα προαπαιτούμενα που χρειάζεται η εφαρμογή για να εκτελεστεί. Πρόκειται για μια εναλλακτική – πιο ελαφριά λύση από τις γνωστές εικονικές μηχανές (virtual machines) οι οποίες είναι βασισμένες στους hypervisors.



Σχήμα 24: Docker architecture - πηγή: docs.docker.com

Η αρχιτεκτονική των Docker χρησιμοποιεί τη λογική του πελάτη – διακομιστή (client-server). Ο πελάτης επικοινωνεί με τον Docker daemon (dockerd), ο οποίος δημιουργεί, εκτελεί και διανείμει τα Docker Containers. Ο docker client και ο docker daemon μπορούν να βρίσκονται είτε στο ίδιο σύστημα, είτε σε διαφορετικό και να συνδέονται μέσω απομακρυσμένης σύνδεσης, όπου επικοινωνούν μέσω REST (Representational State Transfer).

Το Docker – Compose είναι ένας άλλος Docker client που μας δίνει τη δυνατότητα να δουλέψουμε με εφαρμογές που αποτελούνται από ένα σύνολο από Containers.

Το Docker – Swarm είναι ένα σμήνος είτε φυσικών, είτε εικονικών μηχανών που τρέχουν εφαρμογές Docker οι οποίες έχουν ρυθμιστεί με τέτοιο τρόπο ώστε να συνεργάζονται μεταξύ τους και αποτελούν μια συστάδα (cluster). Το Docker – Swarm αναφέρεται και ως Container Orchestration Tool (εργαλείο ενορχήστρωσης Docker Containers) που σημαίνει πως ο προγραμματιστής έχει τη δυνατότητα να διαχειριστεί πολλαπλά Docker Containers που έχουν εγκατασταθεί σε πολλαπλές μηχανές (εικονικές ή φυσικές). Το docker – swarm αποτελείται από τουλάχιστον ένα κόμβο manager (Manager Node) και τουλάχιστον δύο κόμβους workers (Worker Nodes). Ο κόμβος manager είναι υπεύθυνος για την άρτια διαχείριση των πόρων των worker κόμβων και την εξασφάλιση της αποδοτικής λειτουργίας της συστάδας (cluster).

Εισαγωγή στη τεχνολογία του Blockchain

Το blockchain είναι μια κατανεμημένη βάση δεδομένων που διαμοιράζεται μεταξύ των κόμβων ενός δικτύου υπολογιστών. Ως βάση δεδομένων, ένα blockchain αποθηκεύει πληροφορίες ηλεκτρονικά σε ψηφιακή μορφή. Τα blockchains είναι ευρέως γνωστά για τον ρόλο τους στα κρυπτονομίσματα, όπως το Bitcoin, για τη τήρηση ενός ασφαλούς και αποκεντρωμένου αρχείου συναλλαγών που ονομάζεται ledger. Η καινοτομία του blockchain είναι ότι εγγυάται την πιστότητα και την ασφάλεια ενός αρχείου συναλλαγών και προσφέρει εμπιστοσύνη χωρίς την ανάγκη ενός αξιόπιστου τρίτου μέρους, όπως για παράδειγμα μιας τράπεζας.

Μια βασική διαφορά μεταξύ μιας τυπικής βάσης δεδομένων και ενός blockchain είναι ο τρόπος που είναι δομημένα τα δεδομένα. Ένα blockchain συγκεντρώνει τις πληροφορίες που αφορούν σε συναλλαγές (**transactions**) και τις καταγράφει σε ένα μπλοκ (**block**). Τα μπλοκ έχουν περιορισμένη χωρητικότητα αποθήκευσης, οπότε όταν γεμίσουν, σταματούν να δέχονται συναλλαγές και συνδέονται με το προηγούμενο μπλοκ, σχηματίζοντας μια αλυσίδα δεδομένων γνωστή ως blockchain. Όλες οι νέες συναλλαγές που ακολουθούν το μπλοκ που προστέθηκε, καταχωρούνται σε ένα νέο μπλοκ που στη συνέχεια θα προστεθεί με τη σειρά του στην αλυσίδα.

Μια βάση δεδομένων καταχωρεί τα δεδομένα σε πίνακες, ενώ ένα blockchain, δομεί τα δεδομένα σε μπλοκ (blocks) που είναι ενωμένα μεταξύ τους. Αυτή η δομή δεδομένων δημιουργεί εγγενώς ένα αμετάβλητο (immutable) χρονοδιάγραμμα δεδομένων όταν υλοποιείται αποκεντρωμένα (decentralized). Όταν ένα μπλοκ γεμίζει και τοποθετείται στην αλυσίδα, γίνεται αμετάβλητο μέρος του χρονοδιαγράμματος. Κάθε μπλοκ της αλυσίδας περιέχει ως δεδομένο την ακριβή χρονική στιγμή (timestamp) που προστέθηκε στην αλυσίδα.

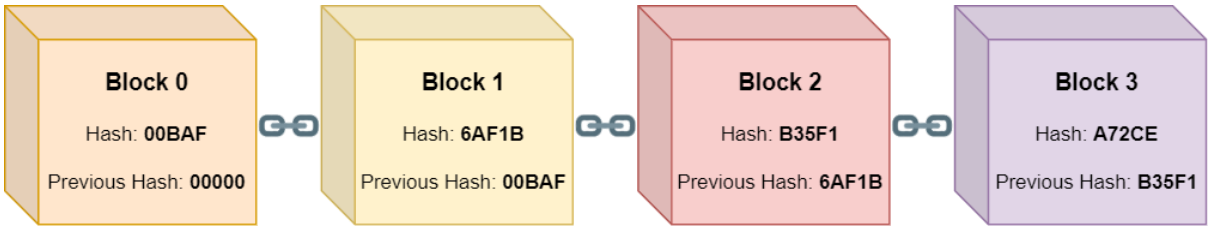
Συνοψίζοντας:

- Το Blockchain είναι ένας τύπος shared βάσης δεδομένων που διαφέρει από μια τυπική βάση δεδομένων στον τρόπο με τον οποίο αποθηκεύει πληροφορίες. Τα blockchain αποθηκεύουν δεδομένα σε μπλοκ που στη συνέχεια συνδέονται μεταξύ τους μέσω κρυπτογραφίας.
- Καθώς εισέρχονται νέα δεδομένα, εισάγονται σε ένα νέο μπλοκ. Μόλις το μπλοκ γεμίσει με δεδομένα, τότε συνδέεται στο προηγούμενο μπλοκ της αλυσίδας, πράγμα που κάνει τα δεδομένα να συνδέονται μεταξύ τους με χρονική σειρά.
- Μπορούν να αποθηκευτούν διαφορετικοί τύποι πληροφοριών σε ένα blockchain, αλλά η πιο συνηθισμένη χρήση μέχρι στιγμής είναι το αρχείο καταγραφής συναλλαγών (ledger).

- Στην περίπτωση του Bitcoin, το blockchain χρησιμοποιείται με αποκεντρωμένο τρόπο, έτσι ώστε κανένας χρήστης ή ομάδα χρηστών να μην έχει τον έλεγχο, ή κατά προτίμηση, όλοι οι χρήστες συλλογικά να διατηρούν τον έλεγχο.
- Τα αποκεντρωμένα blockchain είναι αμετάβλητα, πράγμα που σημαίνει ότι τα δεδομένα που εισάγονται είναι μη αναστρέψιμα. Για το Bitcoin, αυτό σημαίνει ότι οι συναλλαγές καταγράφονται μόνιμα και είναι ορατές σε οποιονδήποτε.

Τρόπος λειτουργίας του blockchain

Κάθε μπλοκ περιέχει δεδομένα συναλλαγών, timestamp[20], το hash του προηγούμενου μπλοκ καθώς και μια μεταβλητή nonce. Τα δεδομένα συναλλαγών εξαρτώνται από το είδος του blockchain, για παράδειγμα στο bitcoin, τα δεδομένα περιέχουν πληροφορίες σχετικά με τις συναλλαγές κρυπτονομισμάτων όπως τις διευθύνσεις του αποστολέα, του παραλήπτη και το ποσό της συναλλαγής. Όλα τα στοιχεία που συνθέτουν ένα block, παράγουν ένα ψηφιακό αποτύπωμα, το hash[21], που ταυτοποιεί το εκάστοτε μπλοκ και είναι μοναδικό. Όταν δημιουργείται ένα μπλοκ, υπολογίζεται το hash του, αν στη συνέχεια μεταβληθεί έστω και ένα στοιχείο, τότε αλλάζει η τιμή του hash. Ένα από τα στοιχεία που συνθέτουν ένα μπλοκ είναι το hash του προηγούμενου μπλοκ, με αυτό τον τρόπο δημιουργείται μια αλυσίδα η οποία είναι αμετάβλητη (immutable).



Σχήμα 25: Blockchain

Στο παραπάνω διάγραμμα (σχήμα 25) φαίνονται τα τέσσερα πρώτα blocks μιας αλυσίδας, κάθε block περιέχει το hash του προηγούμενου. Το αρχικό block ονομάζεται genesis και ως τιμή previous hash έχει 0. Αν κάποιος προσπαθήσει κακόβουλα να αλλοιώσει τα στοιχεία μιας συναλλαγής στο block 1, όπως για παράδειγμα το ποσό της συναλλαγής, τότε θα αλλάξει και το hash το οποίο παράγεται από τα στοιχεία του block 1. Αυτό θα έχει ως αποτέλεσμα, το block 2 να γίνει μη έγκυρο γιατί θα δείχνει διαφορετική τιμή στο πεδίο previous hash και κατά συνέπεια όλα τα blocks που ακολουθούν θα είναι επίσης μη έγκυρα.

Ο μόνος τρόπος για να ξαναγίνει μια αλυσίδα έγκυρη, εφόσον έχουν αλλοιωθεί κακόβουλα κάποια από τα στοιχεία ενός από τα blocks που την αποτελούν, είναι να υπολογιστούν εκ νέου όλα τα blocks από το αλλοιωμένο block και έπειτα. Αυτό είναι εφικτό με την ισχύ που διαθέτουν οι

[20] Unix timestamp είναι ένας τρόπος καταγραφής του χρόνου ως τρέχον σύνολο δευτερολέπτων. Η καταμέτρηση του χρόνου σε δευτερόλεπτα ξεκινά από το Unix Epoch, την 1η Ιανουαρίου του 1970 σε UTC.

[21] Η κρυπτογραφική συνάρτηση κατακερματισμού (cryptographic hash function) είναι ένας μαθηματικός αλγόριθμος που αντιστοιχίζει δεδομένα αυθαίρετου μεγέθους (αποκαλούνται και "μήνυμα") σε ένα πίνακα δυαδικών ψηφίων σταθερού μεγέθους ("hash value", "hash" ή "message digest"). Πρόκειται για μια μονόδρομη συνάρτηση, δηλαδή μια συνάρτηση για την οποία είναι πρακτικά αδύνατη η αντιστροφή της ή η αντιστροφή του υπολογισμού της. Ο μοναδικός τρόπος για να βρεθεί ένα μήνυμα από το οποίο έχει παραχθεί το hash, είναι να επιχειρηθεί με μέθοδο brute-force η αναζήτηση πιθανών εισόδων (μηνυμάτων) στη κρυπτογραφική συνάρτηση κατακερματισμού, με σκοπό να παραχθεί το αποτέλεσμα που να ταιριάζει στο ζητούμενο hash. Οι κρυπτογραφικές συναρτήσεις κατακερματισμού είναι ένα βασικό εργαλείο της σύγχρονης κρυπτογραφίας.

σύγχρονοι υπολογιστές, καθώς μπορούν να υπολογίσουν εκατοντάδες χιλιάδες hashes ανά δευτερόλεπτο. Ενδεικτικά να αναφερθεί πως μια σύγχρονη GPU (graphic processor unit) μπορεί να υπολογίσει μέχρι και 121MH/s (1,21 x 10^8 SHA-256 hashes per second).

Για να μετριαστεί το συγκεκριμένο πρόβλημα, χρησιμοποιείται το Proof-of-work. Proof-of-work είναι ένας μηχανισμός που επιβραδύνει τη δημιουργία ενός block με σκοπό να εμποδίσει τη δημιουργία μιας αλλοιωμένης αλυσίδας που θα μπορούσε να αντικαταστήσει την πραγματική. Η δυσκολία υπολογισμού του hash προκύπτει από έναν αλγόριθμο που προσπαθεί να παράξει hash με προκαθορισμένο πρόθεμα, όσο μεγαλύτερο μήκος έχει το προκαθορισμένο πρόθεμα, τόσο αυξάνεται η δυσκολία δημιουργίας του block.

Ο αλγόριθμος του proof-of-work είναι μια συνάρτηση που περιέχει μια αφηρημένη formula υπολογισμού του hash, που ορίζεται από τον προγραμματιστή. Η συνάρτηση δέχεται ως όρισμα τον ακέραιο αριθμό nonce του προηγούμενου block, έστω previous_nonce και επιστρέφει ένα νέο ακέραιο αριθμό nonce, έστω current_nonce, που θα καταχωρηθεί στο νέο block μαζί με τα δεδομένα των συναλλαγών. Παρακάτω παρουσιάζεται ο κώδικας της συνάρτησης proof-of-work.

```
// Proof of work
public static long proof_of_work(long previous_nonce) {
    // Δημιουργία δυσκολίας στην εύρεση του hash, ορίζοντας πρόθεμα 'aaaaaa'.
    int prefix = 6;
    String prefixString = new String(new char[prefix]).replace('\0', 'a');
    long current_nonce = 0; // Αρχικοποίηση του τρέχοντος nonce
    boolean check_proof = false;
    while (!check_proof) {
        // Η formula που υπολογίζει το hash_operation είναι αφηρημένη.
        // Υπολογισμός του SHA256 της διαφοράς τετραγώνων.
        String hash_operation = DigestUtils.sha256Hex(String.valueOf(
                Math.pow(current_nonce, 2) - Math.pow(previous_nonce, 2)));
        // Αν το πρόθεμα δεν είναι 'aaaaaa', αύξησε το current_nonce κατά 1.
        if (!hash_operation.substring(0, prefix).equals(prefixString)) {
            current_nonce++;
        } else {
            // Έξοδος από while loop.
            check_proof = true;
        }
    }
    // Επέστρεψε το current_nonce.
    return current_nonce;
}
```

Η ασφάλεια του blockchain προέρχεται από τον τρόπο χρήσης του κρυπτογραφικού hashing και του μηχανισμού proof-of-work, αλλά για να διασφαλιστεί η ακεραιότητα της αλυσίδας στο έπακρο, θα πρέπει να διαμοιράζεται σε πλήθος χρηστών ταυτόχρονα. Το blockchain χρησιμοποιεί P2P (peer-to-peer) network όπου ο κάθε χρήστης έχει τη δυνατότητα να συνδεθεί. Οι χρήστες που συνδέονται στο δίκτυο ενός blockchain, κατεβάζουν ένα αντίγραφο ολόκληρης της αλυσίδας στον υπολογιστή τους. Ο κόμβος (node) χρησιμοποιεί τον διαμοιρασμό της αλυσίδας στους χρήστες για να επαληθεύσει τη γνησιότητα της. Όταν δημιουργείται ένα νέο block, αποστέλλεται σε όλους τους χρήστες του δικτύου και στη συνέχεια κάθε κόμβος επαληθεύει πως το νέο block δεν έχει αλλοιωθεί. Αν ο έλεγχος είναι επιτυχής, τότε ο κάθε κόμβος προσθέτει το block στην αλυσίδα. Όλοι οι κόμβοι του δικτύου θα πρέπει να έχουν ομοφωνία στο αποτέλεσμα (consensus), δηλαδή όλοι μαζί να συμφωνούν πως ένα μπλοκ είτε είναι, είτε δεν είναι έγκυρο. Τα blocks που είναι αλλοιωμένα, απορρίπτονται από τους κόμβους και αντικαθίστανται από τα blocks που έχει συμφωνήσει η πλειοψηφία των κόμβων πως είναι έγκυρα. Για να αλλοιωθεί επιτυχώς ένα blockchain, θα πρέπει να υπολογιστούν εκ νέου όλα τα blocks μετά το αλλοιωμένο block, να

υπολογιστεί το proof-of-work για κάθε ένα από τα blocks και αυτό θα πρέπει να συμβεί σε πάνω από το 50% των κόμβων του peer-to-peer δικτύου. Μόνο τότε το αλλοιωμένο blockchain θα γίνει δεκτό από όλους τους κόμβους, πράγμα που είναι σχεδόν αδύνατο να συμβεί.

Περιγραφή εφαρμογής

Η εφαρμογή που υλοποιήθηκε στο πλαίσιο της μεταπτυχιακής διατριβής, είναι βασισμένη στην αρχιτεκτονική μικροϋπηρεσιών και την ασύγχρονη επικοινωνία μεταξύ των στοιχείων που την αποτελούν. Αφορά στη ψηφιοποίηση εγγράφων, καθώς και στη διασφάλιση της γνησιότητας τους, καταχωρώντας το ψηφιακό αποτύπωμα (hash SHA-256) των εγγράφων σε ιδιωτικό blockchain.

Πιο συγκεκριμένα, έχει δημιουργηθεί μια εφαρμογή ιστού (web application) για Εκπαιδευτικά Ιδρύματα, όπου εξουσιοδοτημένοι υπάλληλοι έχουν τη δυνατότητα να μεταφορτώσουν έγγραφα όπως πτυχία και αναλυτικές βαθμολογίες φοιτητών. Στη συνέχεια, οι φοιτητές μέσω του λογαριασμού τους στην εφαρμογή, έχουν τη δυνατότητα να αναζητήσουν και να πραγματοποιήσουν λήψη όλων των εγγράφων που τους αφορούν. Τα έγγραφα μπορούν να σταλούν ως δικαιολογητικά σε φορείς, οι οποίοι με τη σειρά τους, έχουν τη δυνατότητα μέσω της εφαρμογής, να διασταυρώσουν την γνησιότητα των δικαιολογητικών.

Οι τύποι αρχείων των εγγράφων που υποστηρίζονται από την εφαρμογή είναι Portable Document Format (.PDF) και αρχεία εικόνας JPG και PNG. Μετά τη μεταφόρτωση του εγγράφου, αν πρόκειται για αρχείο εικόνας, τότε ακολουθεί ψηφιοποίηση του εγγράφου σε PDF. Στη συνέχεια εξάγεται το ψηφιακό αποτύπωμα (hash SHA-256) του αρχείου PDF και ακολούθως αποθηκεύεται σε μη σχεσιακή βάση δεδομένων MongoDB. Το hash του εγγράφου, μαζί με τα στοιχεία του χρήστη και ημερομηνία καταχώρησης, αποθηκεύονται σε ένα ιδιωτικό blockchain. Στη περίπτωση της επικύρωσης γνησιότητας (validation) εγγράφων, ο χρήστης καλείται να μεταφορτώσει το αρχείο PDF στην εφαρμογή. Ύστερα, επανυπολογίζεται το ψηφιακό αποτύπωμα του εγγράφου (hash SHA-256) και στη συνέχεια, πραγματοποιείται αναζήτηση του hash μέσα στο blockchain. Αν βρεθεί το ίδιο hash στο blockchain, τότε η εφαρμογή επιστρέφει μήνυμα εγκυρότητας στον χρήστη.

Ρόλοι χρηστών - Use Cases

Οι ρόλοι χρηστών της εφαρμογής είναι:

Employee: Πιστοποιημένος υπάλληλος ενός εκπαιδευτικού ιδρύματος που είναι εξουσιοδοτημένος να μπορεί να πραγματοποιήσει τις παρακάτω ενέργειες:

- Μεταφόρτωση των εγγράφων των φοιτητών, όπως πτυχία και αναλυτικές βαθμολογίες, προκειμένου να καταχωρηθούν στο blockchain.
- Δυνατότητα αναζήτησης εγγράφων όλων των φοιτητών, βάσει του αριθμού μητρώου του φοιτητή ή βάσει του hash του εγγράφου.
- Προβολή του blockchain και των εγγράφων που περιέχει. Επιλέγοντας ένα block, ο employee έχει τη δυνατότητα να προβάλει λεπτομέρειες όπως το block hash, την ακριβή ώρα δημιουργίας του block, καθώς και τα transactions που περιέχει. Κάθε transaction αντιστοιχεί σε ένα έγγραφο που έχει μεταφορτωθεί και περιέχει λεπτομέρειες όπως το transaction id, student id και document hash. Επιλέγοντας το Student id, μεταφέρεται στην σελίδα αναζήτησης εγγράφων βάσει μητρώου φοιτητή. Επιλέγοντας το Document Hash, μεταφέρεται στη σελίδα αναζήτησης εγγράφου βάσει του hash. Από εκεί, μπορεί να πραγματοποιήσει λήψη του εγγράφου.

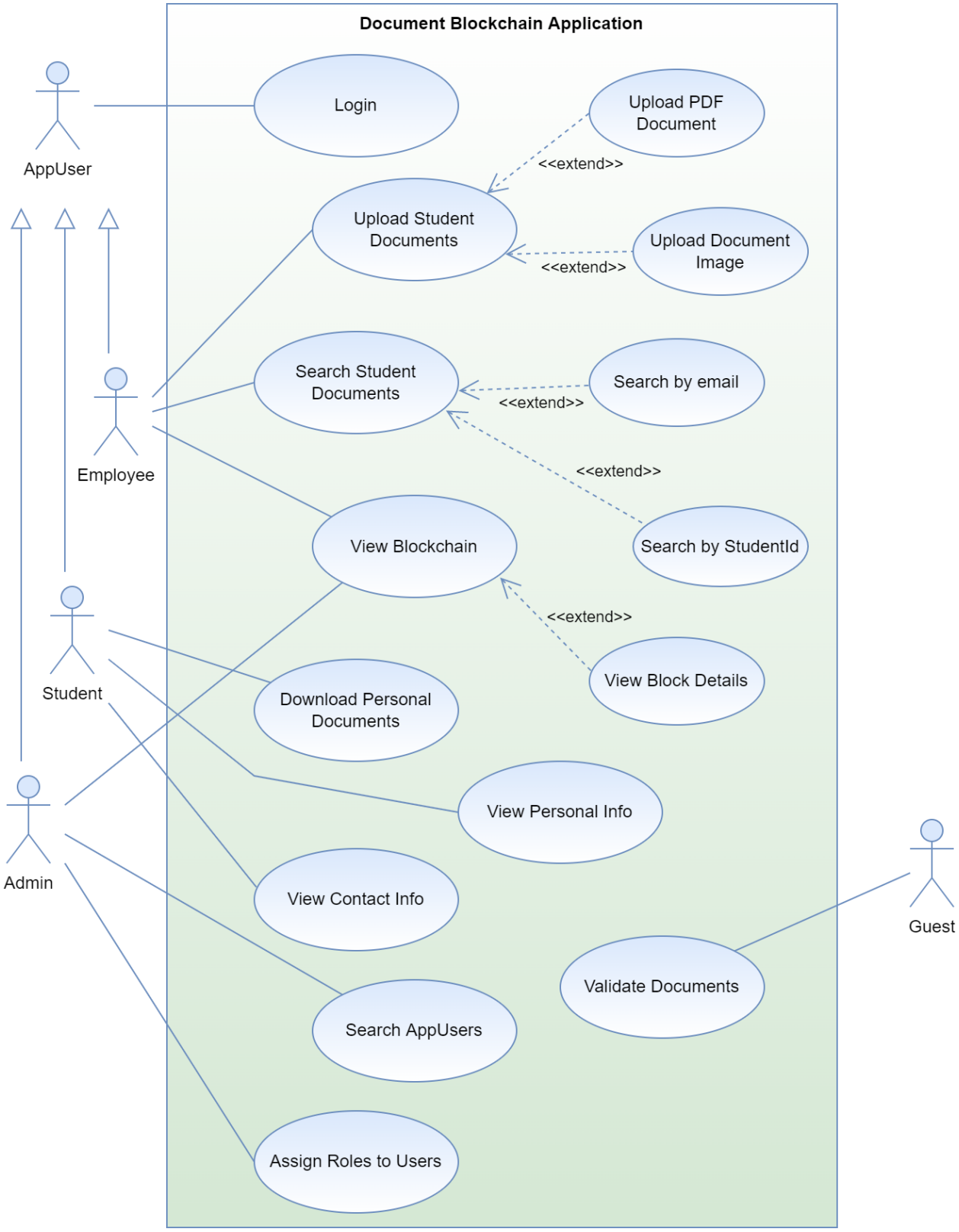
Student: Εγγεγραμμένος φοιτητής με εξουσιοδότηση για τις παρακάτω ενέργειες:

- Προβολή των εγγράφων που αντιστοιχούν στον φοιτητή.
- Προβολή των στοιχείων του.
- Προβολή στοιχείων επικοινωνίας γραμματείας.

Administrator: Διαχειριστής της εφαρμογής με εξουσιοδότηση για τις παρακάτω ενέργειες:

- Αναζήτηση χρηστών βάσει username, email και αριθμό μητρώου φοιτητή.
- Αλλαγή ρόλου ενός χρήστη. Ένας χρήστης μπορεί να εγγραφεί στην πλατφόρμα αλλά δεν έχει εξουσιοδότηση να πραγματοποιήσει ενέργειες. Ο διαχειριστής μπορεί να αναζητήσει τον νέο χρήστη στην εφαρμογή και να του αναθέσει ρόλο student, employee ή admin.
- Προβολή του blockchain και των εγγράφων που περιέχει.

Guest: Η μοναδική δυνατότητα που διαθέτουν οι μη εγγεγραμμένοι χρήστες που επισκέπτονται τη σελίδα, είναι να ελέγξουν τη γνησιότητα εγγράφων PDF που έχουν στη διάθεση τους. Η απάντηση που λαμβάνουν είναι είτε θετική, είτε αρνητική.



Σχήμα 26: Use Case Diagram

Σχεδίαση της εφαρμογής σε αρχιτεκτονική μικροϋπηρεσιών

Η backend εφαρμογή αποτελείται από τέσσερα βασικά microservices, το Document Service, το Blockchain Service, το Validation Service και το User Service.

Document Service: Αποτελεί το βασικότερο microservice της εφαρμογής, καθώς είναι υπεύθυνο για τη λήψη, τη μετατροπή, την εξαγωγή του ψηφιακού αποτυπώματος και την αποθήκευση των εγγράφων στη μη σχεσιακή βάση δεδομένων MongoDB. Επίσης, πραγματοποιεί κλήσεις στο User Service, προκειμένου να ελέγξει τα στοιχεία του φοιτητή, πριν περάσει στην ψηφιοποίηση και αποθήκευση ενός εγγράφου. Μετά την ολοκλήρωση της ψηφιοποίησης και αποθήκευσης του εγγράφου, αποστέλλει ασύγχρονο μήνυμα στο Blockchain Service μέσω του message broker RabbitMQ, που περιέχει στοιχεία του φοιτητή και το hash του εγγράφου.

Blockchain Service: Το Blockchain Service βρίσκεται σε αναμονή λήψης μηνυμάτων από το Document Service, μέσω του message broker RabbitMQ. Όταν καταναλωθεί ένα νέο μήνυμα, τότε προβαίνει στη δημιουργία μιας συναλλαγής (transaction) και εν συνεχεία, στην αποθήκευση της σε μη σχεσιακή βάση δεδομένων MongoDB. Με τη δημιουργία κατάλληλου API, το Blockchain microservice λαμβάνει requests από εξωτερικό scheduler, ώστε να προχωρήσει στο mining των νέων blocks. Όταν δημιουργείται ένα νέο block, όλες οι συναλλαγές (transactions) που βρίσκονται σε αναμονή, μεταφέρονται στο block και ξεκινά η διαδικασία του mining. Ακολουθεί η καταχώρηση του νέου block στην αλυσίδα, καθώς και η επαλήθευση ολόκληρης της αλυσίδας.

Validation Service: Το Validation Service λαμβάνει αρχεία PDF, εξάγει το ψηφιακό αποτύπωμα (hash SHA-256), και στη συνέχεια πραγματοποιεί κλήσεις, αρχικά στο Blockchain Service προκειμένου να ελέγξει αν το hash ανήκει στην αλυσίδα και στη συνέχεια, στο Document Service για να αντλήσει πληροφορίες για τον φοιτητή στον οποίο αφορά το έγγραφο.

User Service: Το User Service παρέχει κατάλληλα APIs για την εύρεση και διαχείριση των χρηστών και των ρόλων τους. Αυτό καθίσταται εφικτό, με χρήση κατάλληλων βιβλιοθηκών για τη σύνδεση και διαχείριση του Keycloak Authorization Server μέσω του User Service.

Η επικοινωνία των μικροϋπηρεσιών με τη frontend εφαρμογή ιστού δε θα ήταν εφικτή χωρίς τη χρήση του Spring Cloud Gateway και του Spring Cloud Discovery Server.

Api Gateway: Το Api Gateway αποτελεί το μοναδικό σημείο εισόδου στο backend της εφαρμογής. Η λειτουργία του ως reverse proxy, επιτρέπει στη frontend εφαρμογή να πραγματοποιεί όλα τα αιτήματα στη διεύθυνση και θύρα που βρίσκεται το Api Gateway. Εκείνο στη συνέχεια, δρομολογεί τα αιτήματα στα microservices που είναι ο τελικός προορισμός. Αν ένα microservice έχει δύο ή περισσότερα στιγμιότυπα (instances) ενεργά, τότε το Api Gateway ως loadbalancer, με μέθοδο round robin[22], δρομολογεί τα requests στα instances με τη σειρά, επιτυγχάνοντας εξισορρόπηση φόρτου. Επίσης, το Api Gateway κάνει χρήση του spring-security-oauth2 και token relay pattern για την προστασία των μικροϋπηρεσιών από μη εξουσιοδοτημένους χρήστες.

Naming Server: Αποτελεί την υλοποίηση του Spring Cloud Discovery Server στη παρούσα εφαρμογή μικροϋπηρεσιών. Ο naming ή eureka server αποθηκεύει προσωρινά τις πληροφορίες για όλα τα στιγμιότυπα των microservices της εφαρμογής, όπως όνομα, θύρα και διεύθυνση IP.

Keycloak Server: Όπως έχει αναφερθεί, πρόκειται για ένα εργαλείο ανοιχτού κώδικα για τη διαχείριση της ταυτοποίησης και πρόσβασης χρηστών και είναι OAuth 2.0 και OpenID Connect

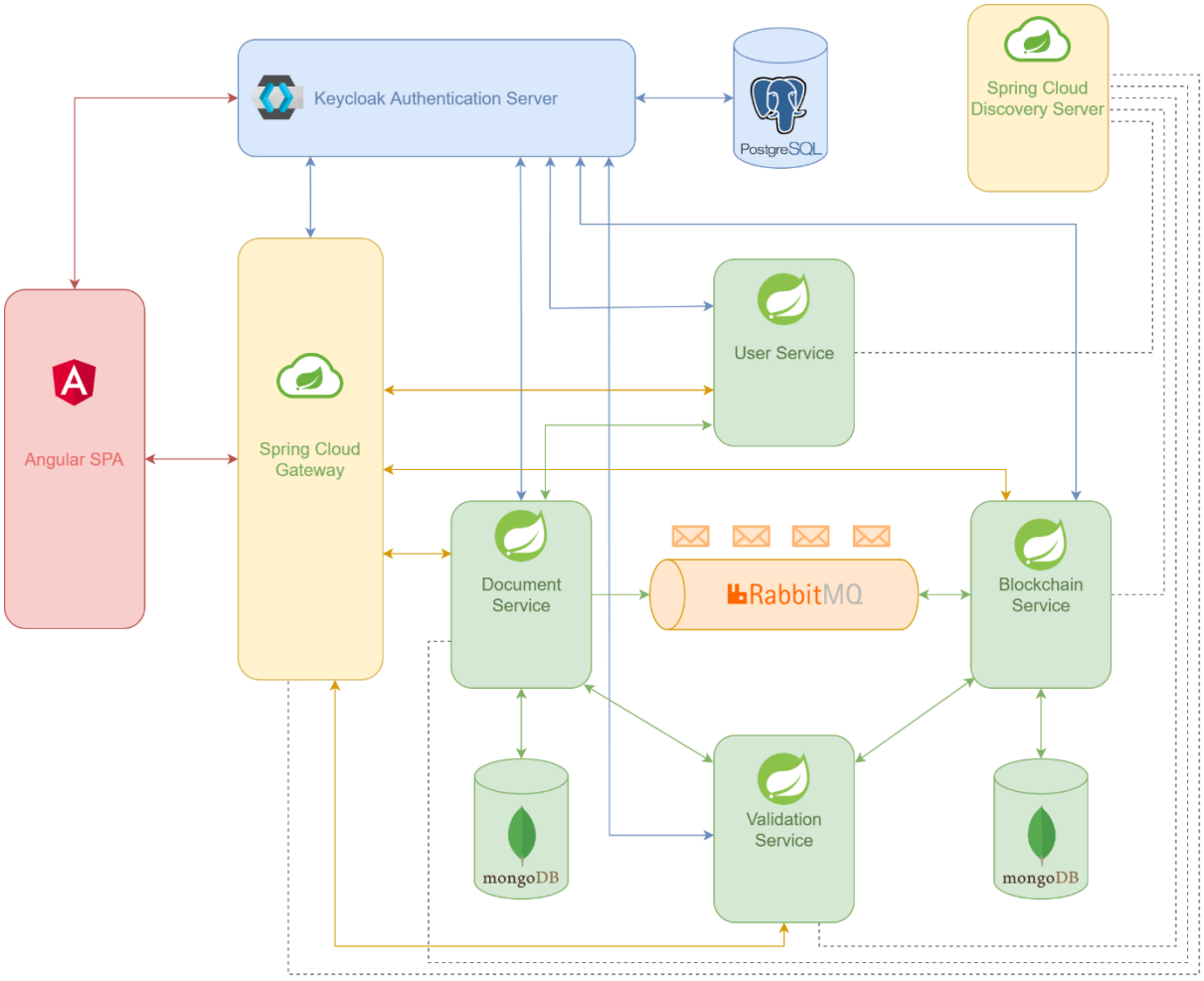
[22] Το round robin είναι η επιλογή όλων των στοιχείων ενός συνόλου με ορθολογική σειρά, συνήθως από την κορυφή μιας λίστας προς το κάτω και στη συνέχεια ξεκινώντας ξανά από την κορυφή της λίστας και ούτω καθεξής.

compliant. Είναι κατάλληλο για την παρούσα εφαρμογή διότι είναι εστιασμένο σε single page applications και παρέχει βιβλιοθήκες για τη διαχείριση χρηστών και ρόλων.

Οι βάσεις δεδομένων και ο message broker που χρησιμοποιήθηκαν είναι:

- **PostgreSQL** για την αποθήκευση χρηστών και ρόλων μέσω του Keycloak.
- **MongoDB** για την αποθήκευση των εγγράφων στο Document Service και των transactions και blocks στο Blockchain Service.
- **RabbitMQ** για την αποστολή ασύγχρονων μηνυμάτων από το Document Service προς το Blockchain Service.

Frontend: Η web εφαρμογή (single page application) έχει δημιουργηθεί σε γλώσσα TypeScript και Angular framework και έχει ως σκοπό την ανάδειξη της λειτουργικότητας της εφαρμογής μικροϋπηρεσιών.



Σχήμα 27: Application Diagram

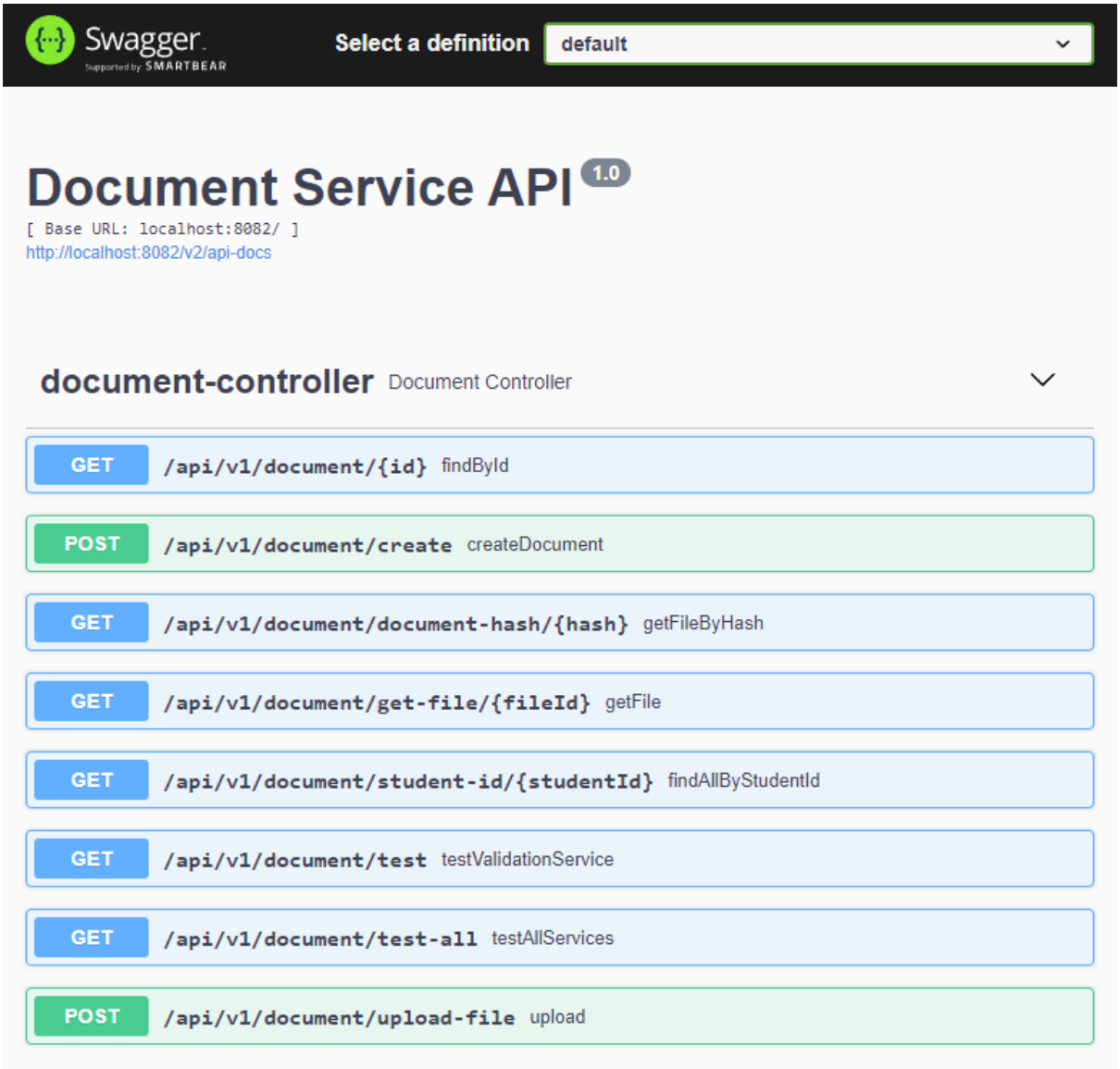
Όλα τα microservices περιέχουν τις βιβλιοθήκες Sleuth και Zipkin, προκειμένου να υπάρχει δυνατότητα tracing των request. Επίσης, όλα τα microservices περιέχουν το Jib plugin της Google στο pom.xml, ώστε να δημιουργούν Docker images όταν πραγματοποιείται maven package goal μετά από αλλαγές στον κώδικα. Το Jib plugin έχει ρυθμιστεί κατάλληλα, ώστε να μεταφορτώνει τα images στο DockerHub.

Αναλυτική παρουσίαση backend εφαρμογής

Σε όλα τα microservices, η δομή του κώδικα ακολουθεί την αρχή διαχωρισμού των αρμοδιοτήτων (separation of concerns principle) με τη χρήση του controller – service – repository pattern. Σε κάθε υπηρεσία, έχουν δημιουργηθεί packages όπως controller, service, model, repository, dto, config και util που περιέχουν στοιχεία, όπως τάξεις και μεθόδους, που αφορούν σε διαφορετικές αρμοδιότητες σύμφωνα με το σχεδιαστικό πρότυπο. Παρακάτω παρουσιάζονται αναλυτικά τα microservices της εφαρμογής:

Document Service

Παρακάτω παρουσιάζονται τα endpoints του Document Service:

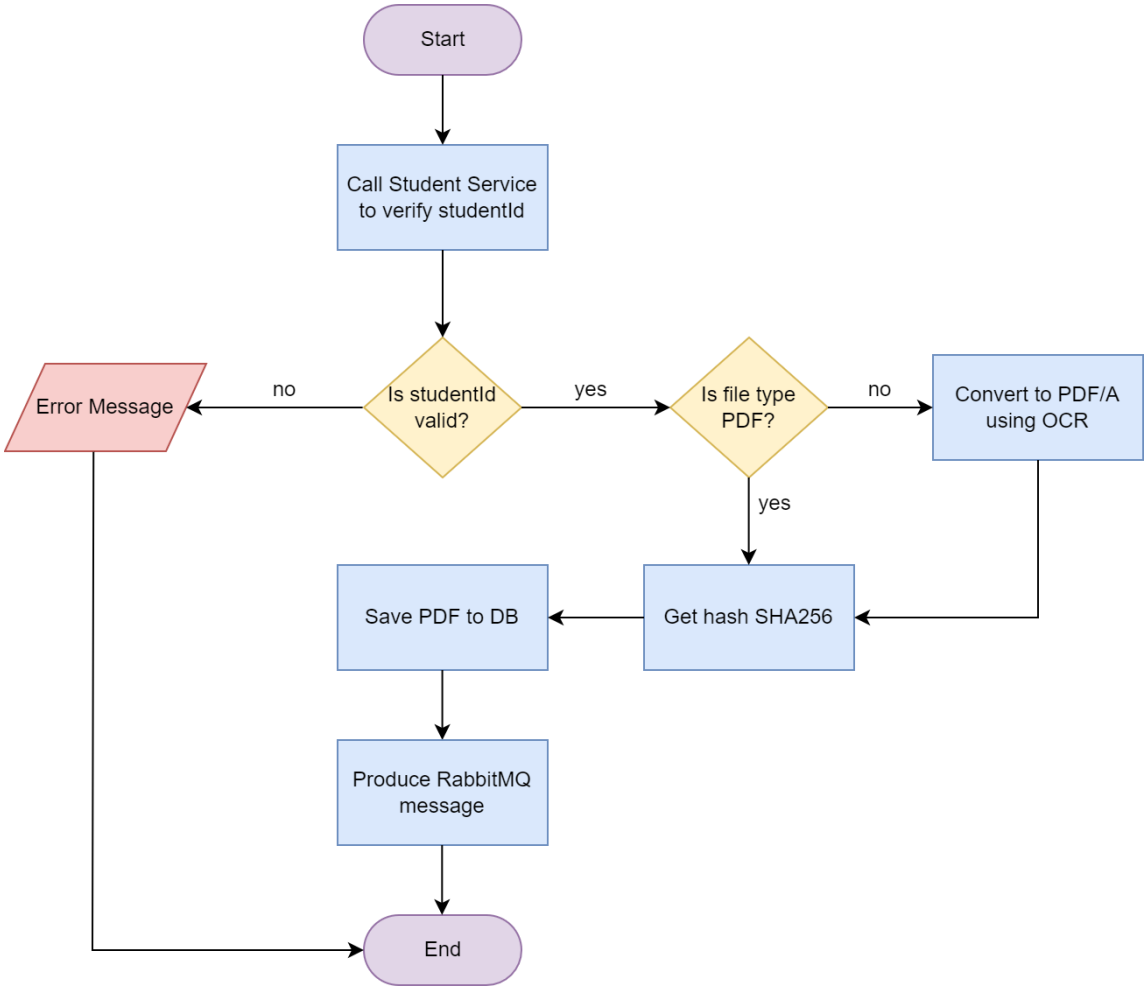


Εικόνα 1: Document Service API

Το Document Service παρέχει APIs για τη μεταφόρτωση, αναζήτηση και λήψη εγγράφων που είναι αποθηκευμένα στη μη σχεσιακή βάση δεδομένων MongoDB με όνομα doc-chain-document. Τα APIs καταναλώνονται κυρίως από την frontend εφαρμογή μέσω του Api Gateway και από το Validation Service.

Ο βασικός ρόλος του Document Service είναι η λήψη, μετατροπή και αποθήκευση εγγράφων των φοιτητών και στη συνέχεια η αποστολή ασύγχρονου μηνύματος στο Blockchain Service, ώστε να προχωρήσει στη δημιουργία transaction με το ψηφιακό αποτύπωμα (hash SHA256) του εγγράφου.

Διάγραμμα ροής και περιγραφή μεταφόρτωσης εγγράφου από employee



Σχήμα 28: Document upload action flow chart

Η μεταφόρτωση ενός εγγράφου φοιτητή από έναν υπάλληλο, έχει προορισμό το Document-Service και πραγματοποιείται με μέθοδο POST στο endpoint /api/v1/document/upload-file. Μετά την ολοκλήρωση της μεταφόρτωσης, το σύστημα προχωρά στην εξής διαδικασία:

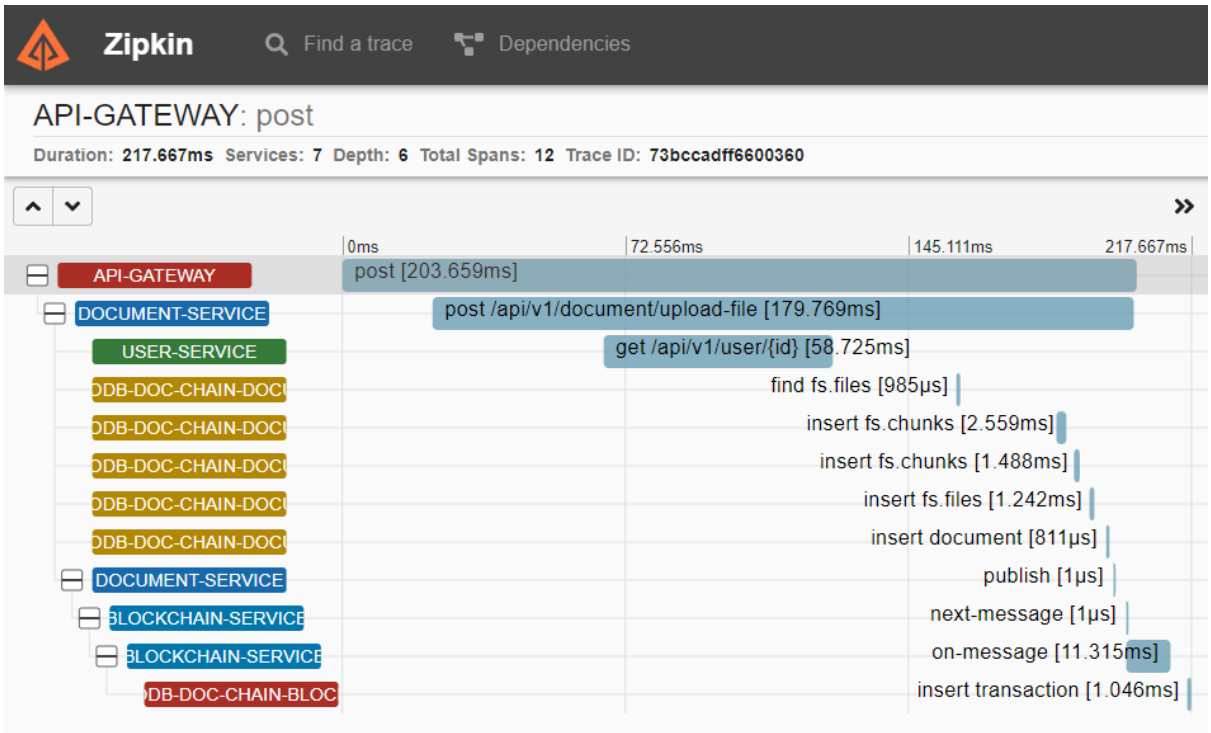
1. Έλεγχος του αριθμού μητρώου του φοιτητή με κλήση στο User-Service.
   1.1. Αν ο αριθμός μητρώου δεν αντιστοιχεί σε φοιτητή, τότε εμφανίζεται μήνυμα σφάλματος και η διαδικασία τερματίζει.
   1.2. Αν ο αριθμός μητρώου αντιστοιχεί σε φοιτητή, τότε η διαδικασία μεταβαίνει στο βήμα 2.
2. Έλεγχος τύπου αρχείου. Το σύστημα διαβάζει την επέκταση αρχείου.
   2.1. Αν πρόκειται για αρχείο pdf, τότε η διαδικασία μεταβαίνει στο βήμα 3.
   2.2. Αν πρόκειται για αρχείο png ή jpg, τότε το αρχείο εικόνας μετατρέπεται σε pdf/A με χρήση της βιβλιοθήκης tesseract-ocr και στη συνέχεια η διαδικασία μεταβαίνει στο βήμα 3.
3. Εύρεση του ψηφιακού αποτυπώματος (hash SHA-256) του αρχείου pdf.
4. Αποθήκευση του αρχείου pdf στη βάση δεδομένων MongoDB με χρήση της βιβλιοθήκης GridFS.

5. Αποστολή ασύγχρονου μηνύματος με το hash του αρχείου pdf στο message broker RabbitMQ που έχει ως αποδέκτη το Blockchain Service.

Το controller layer του Document Service περιέχει μεθόδους που συνδέονται με το service layer μέσω διεπαφών (interfaces), ώστε να επιτυγχάνεται χαλαρή σύζευξη (loose coupling) μεταξύ των δυο επιπέδων:

```
public interface DocumentService {

    // Testing Method
    Mono<String> testResponse();
    // Testing Method
    Flux<String> testAllServices();
    // Create Document
    Mono<DocumentDto> createDocument(Mono<CreateDocumentRequestDto> requestDtoMono);
    // Get Document by id
    Mono<DocumentDto> getDocumentById(String id);
    // Get Documents by student registration number
    Flux<DocumentDto> getDocumentsByStudentId(String id);
    // Delete Document
    Mono<Void> deleteDocument(String id);
    // Read file
    Flux<String> getLines(Flux<FilePart> filePartFlux);
    // Read document lines
    List<String> processAndGetLinesAsList(String string);
    // Find Document by hash
    Flux<DocumentDto> findByDocumentHash(String hash);

}
```

Όπως φαίνεται από τις υπογραφές μεθόδων, οι τύποι που επιστρέφουν είναι υλοποιήσεις της διεπαφής Publisher<T> του προτύπου reactive streams. Οι υλοποιήσεις είναι το Mono<T> και Flux<T> του project reactor.

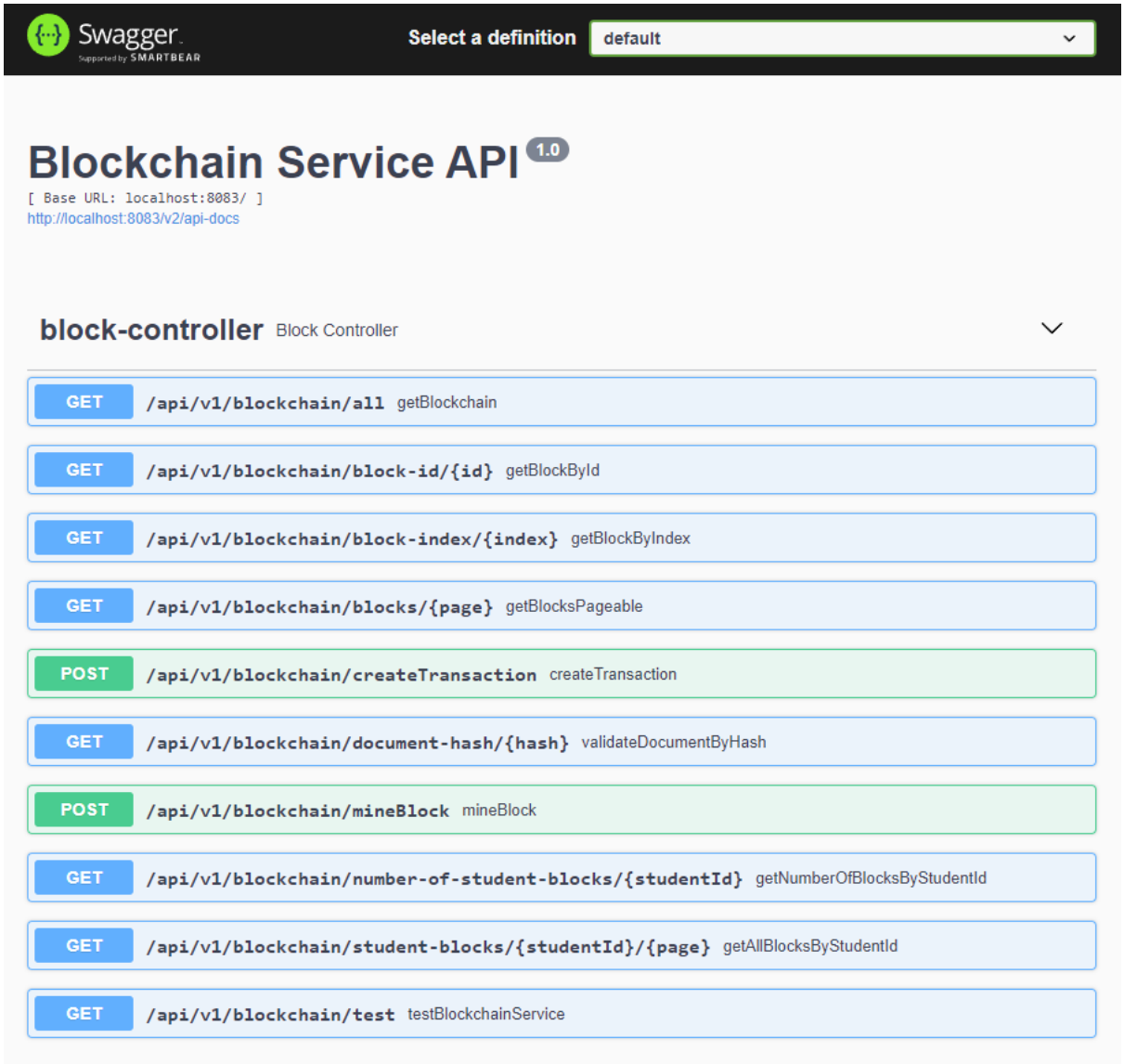


Εικόνα 2: Zipkin - document upload request tracing

Στην παραπάνω εικόνα (εικόνα 2) διακρίνεται η ροή ενεργειών που πραγματοποιούνται όταν ένας υπάλληλος μεταφορτώνει ένα έγγραφο PDF που ανήκει σε φοιτητή. Το POST request δρομολογείται μέσω του Api Gateway στο Document Service endpoint /api/v1/document/upload-file, στη συνέχεια πραγματοποιείται GET request στο endpoint /api/v1/user/{id} του User Service, ώστε να διαπιστωθεί αν αριθμός μητρώου ανήκει σε φοιτητή. Στη συνέχεια αποθηκεύεται το έγγραφο στη βάση δεδομένων (collection: doc-chain-document) και ακολουθεί η αποστολή μηνύματος (γραμμή 9) στον message broker RabbitMQ με το hash SHA-256 του εγγράφου. Το Blockchain λαμβάνει το μήνυμα (γραμμή 10) και προχωρά στη δημιουργία transaction που περιέχει το hash του εγγράφου. Τέλος, το Blockchain Service αποθηκεύει στη δική του βάση δεδομένων το transaction (collection: doc-chain-blockchain). Η συνολική διάρκεια των ενεργειών είναι 217,667ms.

Blockchain Service

Παρακάτω παρουσιάζονται τα endpoints του Blockchain Service:



Εικόνα 3: Blockchain Service API

Το Blockchain Service παρέχει APIs για την δημιουργία transactions, την αναζήτηση blocks και την επικύρωση ύπαρξης document hash στην αλυσίδα. Το blockchain και τα transactions βρίσκονται αποθηκευμένα στη μη σχεσιακή βάση δεδομένων MongoDB με όνομα doc-chain-blockchain. Τα

APIs καταναλώνονται κυρίως από την frontend εφαρμογή μέσω του Api Gateway και από το Validation Service.

Επίσης, έχει δημιουργηθεί endpoint το οποίο εκκινεί τη διαδικασία εξόρυξης (mining) νέου block. Η εφαρμογή έχει σχεδιαστεί, ώστε να δέχεται αίτημα για την εκκίνηση της εξόρυξης νέου block, προκειμένου να αποφευχθεί η ταυτόχρονη εκκίνηση της διαδικασίας από δύο ή παραπάνω instances του Blockchain service όταν έχει πραγματοποιηθεί scaling. Το αίτημα παράγεται από μια μέθοδο που χρησιμοποιεί scheduler και βρίσκεται στο Api Gateway:

```
@Scheduled(fixedRate = 600_000L) // 10 minutes
public void MineBlock() {
    this.webClient.build().post()
            .uri("lb://blockchain-service/api/v1/blockchain/mineBlock")
            .attributes(ServerOAuth2AuthorizedClientExchangeFilterFunction
                    .clientRegistrationId(clientId))
            .retrieve()
            .bodyToMono(Block.class)
            .subscribe(
                    r -> log.info("Block with id : " + r.getId()
                            + "  and index : " + r.getIndex() + " was mined."),
                    error -> log.error("error " + error.getMessage()),
                    () -> log.info("web client call completed"));
}
```

Το Api Gateway κρίθηκε ως το καταλληλότερο στοιχείο της εφαρμογής για να τοποθετηθεί ο scheduler, διότι δεν έχει τη δυνατότητα scaling, καθώς πρόκειται για το μοναδικό σημείο εισόδου προς τα microservices. Ο loadbalancer δρομολογεί το αίτημα σε ένα instance κάθε φορά.

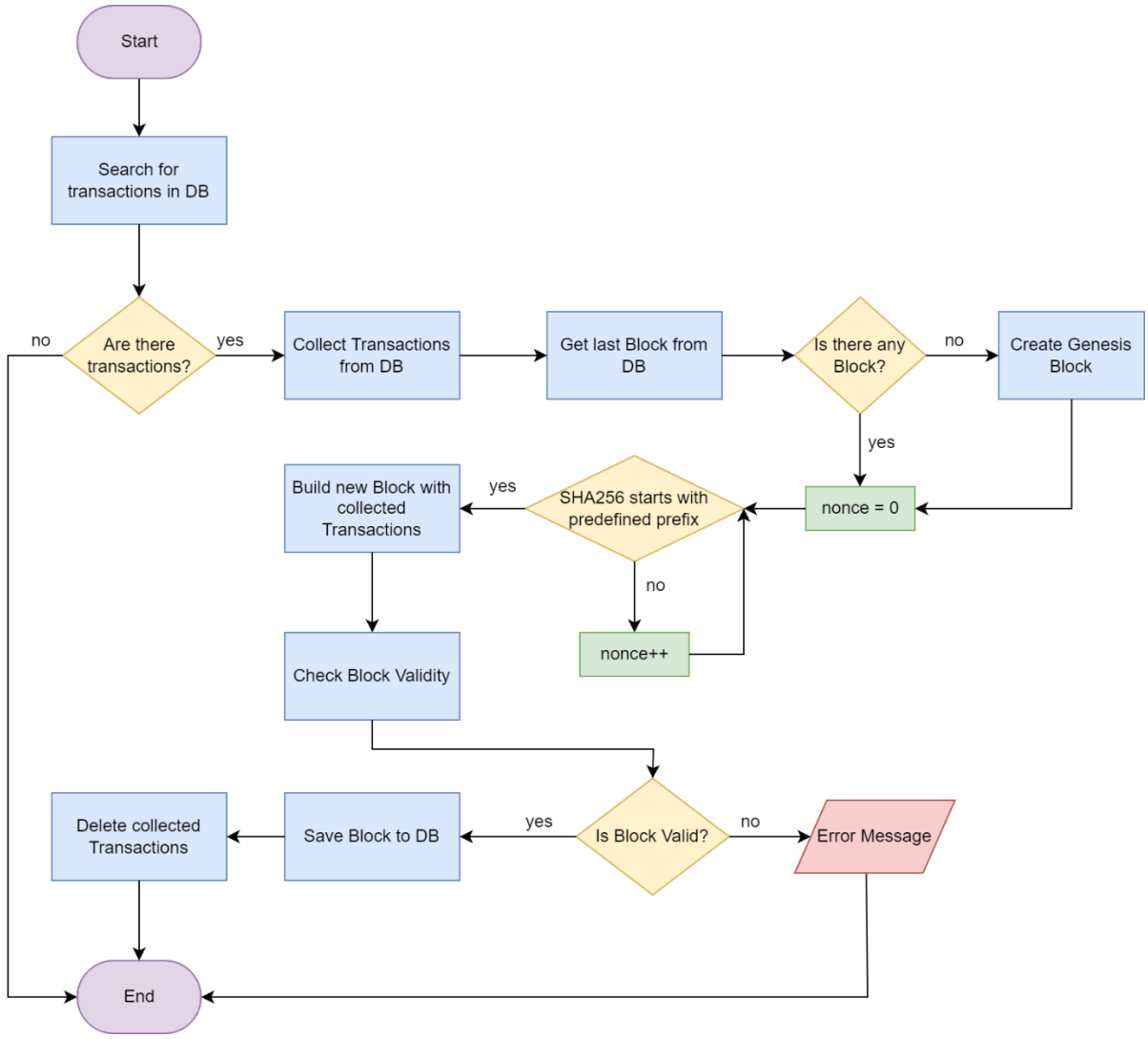
Το Blockchain Service βρίσκεται σε αναμονή λήψης μηνύματος από το Document Service μέσω του RabbitMQ. Όταν ληφθεί ένα μήνυμα, τότε δημιουργείται ένα αντικείμενο Transaction που περιέχει τον αριθμό μητρώου του φοιτητή και το ψηφιακό αποτύπωμα (hash SHA256) του εγγράφου, στη συνέχεια αποθηκεύει το transaction στη βάση δεδομένων.


```
_id: ObjectId("62b08f67ae6bd65f10940c88")
transactionId: "13555d3f-8d0b-4be2-be8a-17145cf6dcdd"
hash: "0e1aae3ab0c23eb27091ce5c6d1e6e11a48620f0ba70323f44f83349585cea8b"
data: Object
   studentId: "d5490867-1d25-48b5-8924-2d734d0c4f3e"
   documentHash: "4ddb2f5bbd512f011e7cfa7e984b57e5912d8b34aead5350de5fdce4e3f7b159"
_class: "com.unipi.blockchainservice.models.Transaction"

_id: ObjectId("62b08fe9ae6bd65f10940c89")
transactionId: "64cdc5de-22b9-43c3-b381-7a61ad560d43"
hash: "b0ad15a4afa5a0f04fc4a125177a83719c70f736003c71d3108d008315f1789e"
data: Object
   studentId: "d5490867-1d25-48b5-8924-2d734d0c4f3e"
   documentHash: "e6982ac025a08c1097c837fb7ab89bc8edf7a16cc9c34e408c30b6981727d40f"
_class: "com.unipi.blockchainservice.models.Transaction"
```


Εικόνα 4: Transactions stored in mongoDB – Transaction Collection

Στη παραπάνω εικόνα (εικόνα 4) εμφανίζονται δύο αποθηκευμένα transactions στη βάση δεδομένων. Όταν ληφθεί αίτημα για εξόρυξη, τα transactions θα περάσουν μέσα στο νέο block. Όταν ολοκληρωθεί η διαδικασία εξόρυξης και το νέο block αποθηκευτεί στο Block Collection της MongoDB, τα transactions που έχουν περάσει στο block, διαγράφονται από το Transaction Collection.

Διάγραμμα ροής και περιγραφή δημιουργίας νέου Block



Σχήμα 29: New block mining flow chart

Περιγραφή ροής δημιουργίας νέου Block:

1. Κλήση στη βάση δεδομένων προκειμένου να διαπιστωθεί η ύπαρξη transactions στο Transaction Collection.
   1.1. Αν δεν βρεθούν transactions, η διαδικασία τερματίζει.
   1.2. Αν βρεθούν transactions, τότε ανακτώνται από τη βάση δεδομένων και η διαδικασία μεταβαίνει στο βήμα 2.
2. Κλήση στη βάση δεδομένων προκειμένου να ανακτηθεί το τελευταίο block της αλυσίδας.
   2.1. Αν δεν υπάρχει block, τότε δημιουργείται genesis block, αποθηκεύεται και η διαδικασία μεταβαίνει στο βήμα 3.
   2.2. Αν υπάρχει προηγούμενο block, τότε ανακτάται και η διαδικασία μεταβαίνει στο βήμα 3.
3. Αρχικοποίηση της μεταβλητής nonce (nonce = 0).
4. Proof of Work: Ενόσω το hash SHA256 που παράγεται από την αφηρημένη formula που έχει χρησιμοποιηθεί hashSha256(nonce^2 – previous_nonce^2) δεν ξεκινά με το πρόθεμα που έχει οριστεί, για παράδειγμα: hash prefix = ‘aaaaaa’, τότε αυξάνεται το nonce κατά ένα (nonce++) και επαναλαμβάνεται το βήμα 4. Το μήκος του προθέματος, ορίζει τη δυσκολία εύρεσης του

hash. Στην υλοποίηση έχει χρησιμοποιηθεί prefix μήκους 2 ώστε το hash να παράγεται σχετικά άμεσα.

5. Δημιουργία νέου αντικειμένου Block. Το νέο block περιέχει τα εξής πεδία: Index, το hash SHA256 του προηγούμενου block, το hash του τρέχοντος block, τον ακέραιο nonce, τη χρονική στιγμή δημιουργίας του block σε μορφή timestamp και λίστα των αντικειμένων transactions.
6. Το νέο block που παράγεται, υποβάλλεται σε σειρά ελέγχων προκειμένου να διαπιστωθεί η εγκυρότητα του σε σχέση με την υπόλοιπη αλυσίδα.
   6.1. Αν διαπιστωθεί ανακολουθία, τότε παράγεται Error Message που καταγράφεται σε logs και η διαδικασία τερματίζει.
   6.2. Αν το block είναι έγκυρο, τότε η διαδικασία μεταβαίνει στο βήμα 7.
7. Αποθήκευση του νέου block στο Block Collection της βάσης δεδομένων.
8. Διαγραφή των transactions που έχουν εισαχθεί στο block από το Transaction Collection.


```
_id: ObjectId("62b09147ae6bd65f10940c8a")
index: 3
previousHash: "6e038dcce1bc372a1dcd37d4b22ec11bc3dc71e4d767675a38aa8a587ea926c8"
timestamp: 1655738695
nonce: 54
hash: "a61a6bdfcdece1349195f55bb40816e3f117f3a1ec08b6210943a936e6ce7a2b"
transactions: Array
  0: Object
      _id: ObjectId("62b08f67ae6bd65f10940c88")
      transactionId: "13555d3f-8d0b-4be2-be8a-17145cf6dcdd"
      hash: "0e1aae3ab0c23eb27091ce5c6d1e6e11a48620f0ba70323f44f83349585cea8b"
    data: Object
        studentId: "d5490867-1d25-48b5-8924-2d734d0c4f3e"
        documentHash: "4ddb2f5bbd512f011e7cfa7e984b57e5912d8b34aead5350de5fdce4e3f7b159"
  1: Object
      _id: ObjectId("62b08fe9ae6bd65f10940c89")
      transactionId: "64cdc5de-22b9-43c3-b381-7a61ad560d43"
      hash: "b0ad15a4afa5a0f04fc4a125177a83719c70f736003c71d3108d008315f1789e"
    data: Object
        studentId: "d5490867-1d25-48b5-8924-2d734d0c4f3e"
        documentHash: "e6982ac025a08c1097c837fb7ab89bc8edf7a16cc9c34e408c30b6981727d40f"
_class: "com.unipi.blockchainservice.models.Block"

_id: ObjectId("62b095fdae6bd65f10940c8c")
index: 4
previousHash: "a61a6bdfcdece1349195f55bb40816e3f117f3a1ec08b6210943a936e6ce7a2b"
timestamp: 1655739901
nonce: 111
hash: "0e75ff7b0ae26c7c035217c70f224addc633daf107135b733180133818d12941"
transactions: Array
  0: Object
      _id: ObjectId("62b09492ae6bd65f10940c8b")
      transactionId: "fcc7eecd-18ae-4c09-ada0-a20fb36f5559"
      hash: "02e95ba046a91df6721c848fe600db2f001bab4e9b8679f71758aa74dcb230a0"
    data: Object
        studentId: "d5490867-1d25-48b5-8924-2d734d0c4f3e"
        documentHash: "b1da68c0c6a7b127376a6335aba37ca967b189e0cdf6126700de2cef994fe7c1"
_class: "com.unipi.blockchainservice.models.Block"
```


Εικόνα 5: Blocks stored in mongoDB – Block Collection

Στη παραπάνω εικόνα (εικόνα 5) εμφανίζονται δυο διαδοχικά blocks που είναι αποθηκευμένα στο Block Collection της βάσης δεδομένων. Στο πρώτο block έχουν καταχωρηθεί τα στοιχεία δυο εγγράφων που αποτελούνται από το studentId και documentHash. Επίσης, μπορεί να παρατηρηθεί πως το previous hash του δεύτερου block είναι το hash του πρώτου block.

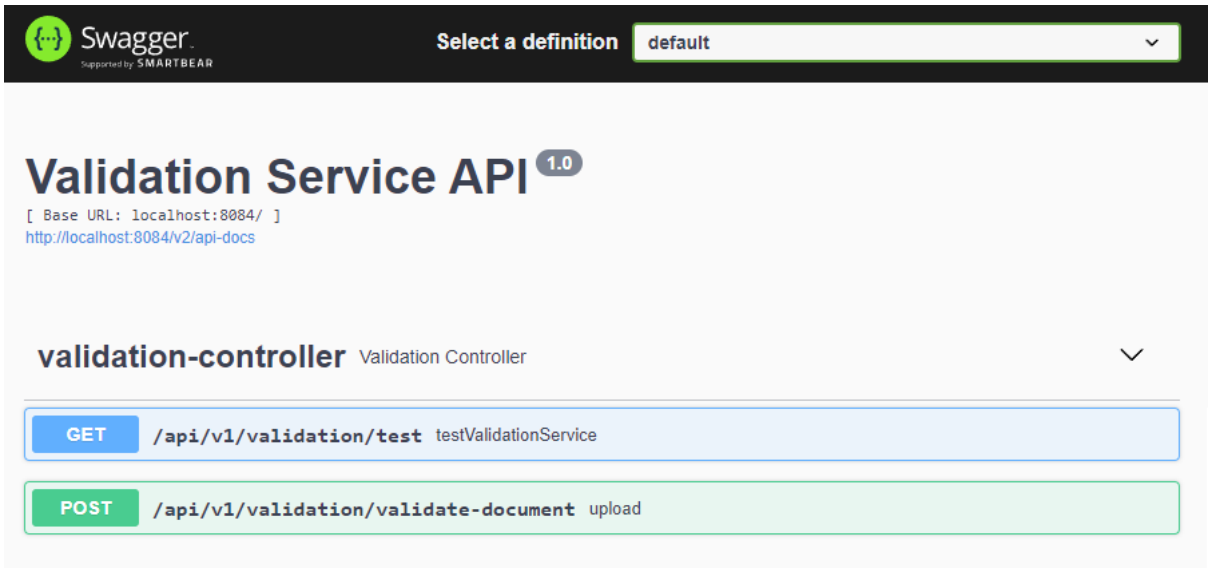
Ομοίως με το Document Service, Το controller layer του Blockchain Service περιέχει μεθόδους που συνδέονται με το service layer μέσω διεπαφών (interfaces), ώστε να επιτυγχάνεται χαλαρή σύζευξη (loose coupling) μεταξύ των δυο επιπέδων. Όπως φαίνεται από τις υπογραφές των μεθόδων, τα δεδομένα που επιστρέφουν είναι Mono<T> και Flux<T>.

```
public interface BlockService {
    // Testing Method
    Mono<String> testResponse();
    // Find blocks by transactions in blockchain
    Flux<Block> findTransactionInChain(String transactionId, Flux<Block> referenceBlockchain);
    // Get all blocks of the blockchain
    Flux<Block> getAllBlocks();
    // Get all blocks using pagination
    Flux<Block> getAllBlocksPageable(Pageable pageable);
    // Save block to database
    Mono<Block> saveBlock(Block block);
    // Add block to blockchain procedure
    Mono<Block> addBlock(Block newBlock);
    // Get last block from database
    Mono<Block> getLastBlock();
    // Find block by index
    Mono<Block> findByIndex(long index);
    // Remove transactions that have been added in blockchain
    Mono<Block> removeBlockTransactionsFromTransactions(Block newBlock);
    // Find if a transaction already exists in blockchain, if yes throw TransactionError
    Mono<Transaction> checkTransaction(Transaction transaction,
                                       Flux<Block> referenceBlockchain);
    // Find block by id
    Mono<Block> findById(String id);
    // Trigger block mining procedure
    Mono<Block> mineBlock();
    // Get all blocks ordered by index descending
    Flux<Block> getAllByOrderByIndexDesc();
    // Get all blocks by studentId
    Flux<Block> getAllBlocksByStudentId(String studentId, Pageable pageable);
    // Get number of blocks that include a specific studentId
    Mono<Integer> getNumberOfBlocksByStudentId(String studentId);
    // Get blocks that include a specific document hash
    Flux<ValidationResponseDto> getBlocksContainingDocumentHash(String hash);
}
```

```
public interface TransactionService {
    // RabbitMQ listener
    void listen(DocumentDto in);
    // Create a new transaction
    Mono<Transaction> createTransaction(DocumentDto documentDto);
    // Get all transactions
    Flux<Transaction> getAllTransactions();
    // Get transaction by id
    Mono<Transaction> getTransactionById(String id);
    // Delete transaction by id
    Mono<Void> deleteTransactionById(String id);
    // Delete transaction by transactionId
    Mono<Void> deleteByTransactionId(String transactionId);
    // Find transaction by transactionId
    Mono<Transaction> findByTransactionId(String transactionId);
    // Delete transaction
    Mono<Void> deleteTransaction(Transaction transaction);
    // Find transaction by document hash
    Mono<Transaction> findByData_DocumentHash(String documentHash);
    // Find transaction by studentId
    Mono<Transaction> findByData_StudentId(String studentId);
}
```

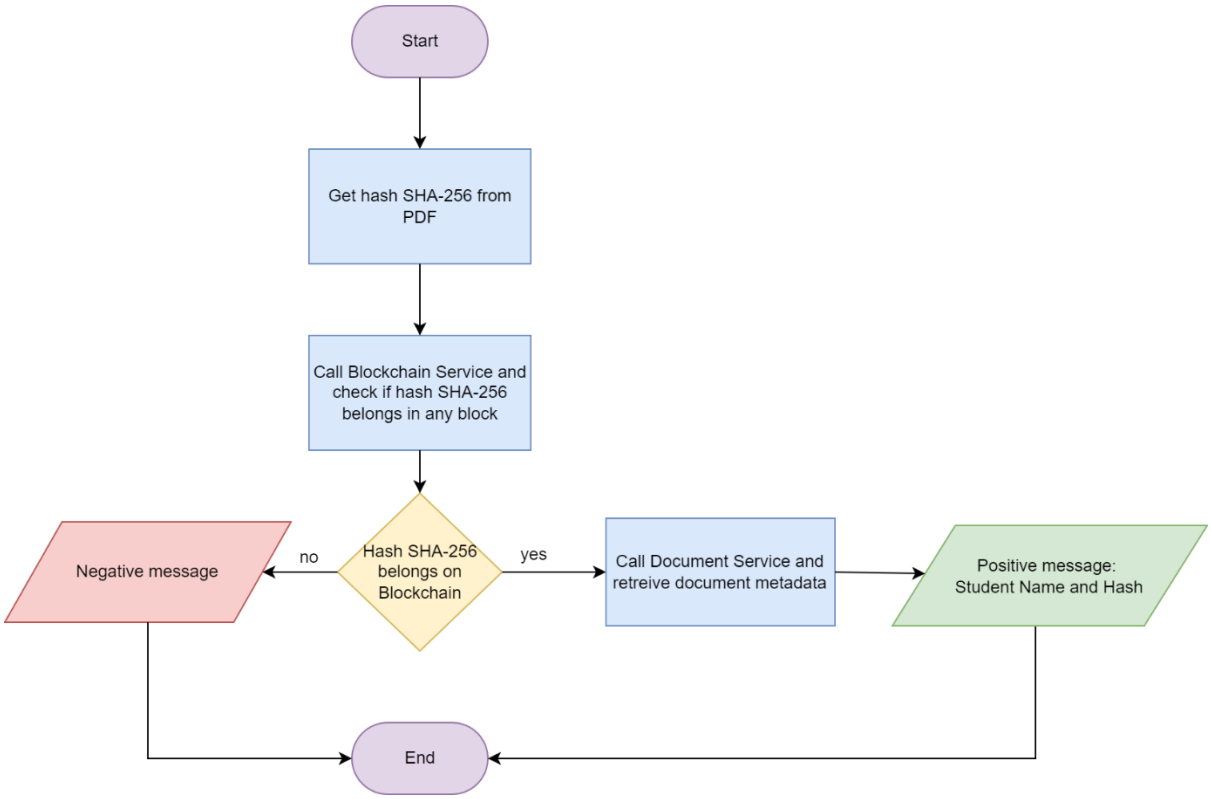
Validation Service

Παρακάτω παρουσιάζονται τα endpoints του Validation Service:



Εικόνα 6: Validation Service API

Το Validation Service παρέχει δημόσιο API για την επικύρωση εγγράφων σε μορφή pdf από τρίτους. Το αρχείο μεταφορτώνεται και στη συνέχεια παράγεται εκ νέου το ψηφιακό αποτύπωμα (hash SHA-256). Το διάγραμμα ροής περιγράφει τις ενεργείς που ακολουθούν μετά τη μεταφόρτωση του αρχείου, προκειμένου να διαπιστωθεί η εγκυρότητα ενός εγγράφου:



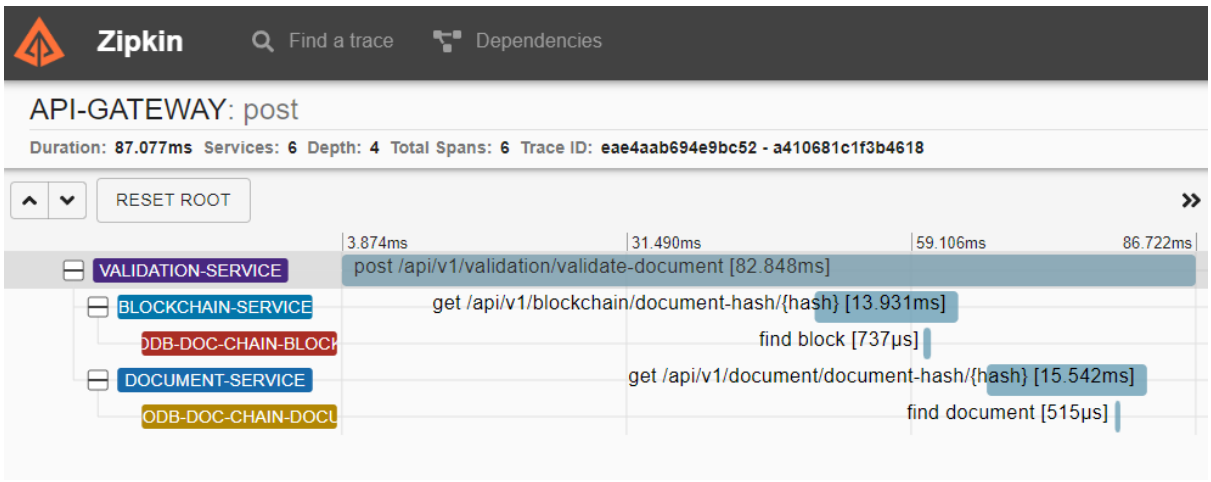
Σχήμα 30: Document validation flow chart

Περιγραφή ροής εγκυρότητας εγγράφου:

Μετά την ολοκλήρωση της μεταφόρτωσης, το σύστημα προχωρά στην εξής διαδικασία:

1. Εύρεση του ψηφιακού αποτυπώματος (hash SHA-256) του αρχείου pdf.
2. Κλήση στο Blockchain Service προκειμένου να διαπιστωθεί αν το hash SHA-256 βρίσκεται σε κάποιο από τα blocks της αλυσίδας.
   2.1. Αν το hash δε βρεθεί σε κάποιο block της αλυσίδας, το σύστημα επιστρέφει αρνητικό μήνυμα: «Δεν είναι δυνατή η πιστοποίηση της γνησιότητας του εγγράφου». Στη συνέχεια η διαδικασία τερματίζει.
   2.2. Αν το hash βρεθεί στην αλυσίδα, τότε η διαδικασία μεταβαίνει στο βήμα 3.
3. Κλήση στο Document Service, ώστε να ανακτηθεί το ονοματεπώνυμο του χρήστη και η ημερομηνία καταχώρησης του εγγράφου στην εφαρμογή.
4. Εμφάνιση θετικού μηνύματος: Το έγγραφο του χρήστη {fullname} με hash {hash} είναι έγκυρο. Στη συνέχεια η διαδικασία τερματίζει.

Στην εικόνα (εικόνα 7) που ακολουθεί, απεικονίζεται η ροή ενεργειών που πραγματοποιούνται όταν μεταφορτώνεται ένα έγγραφο PDF που ανήκει στην αλυσίδα. Το POST request δρομολογείται μέσω του Api Gateway στο Validation Service endpoint /api/v1/validation/validate-document, στη συνέχεια πραγματοποιείται GET request στο /api/v1/blockchain/document-hash/{hash} με το hash του εγγράφου. Το Blockchain Service αναζητά και βρίσκει στην αλυσίδα το hash. Ακολουθεί GET request στο /api/v1/document/document-hash/{hash}, ώστε να επιστρέψει τα στοιχεία του χρήστη.



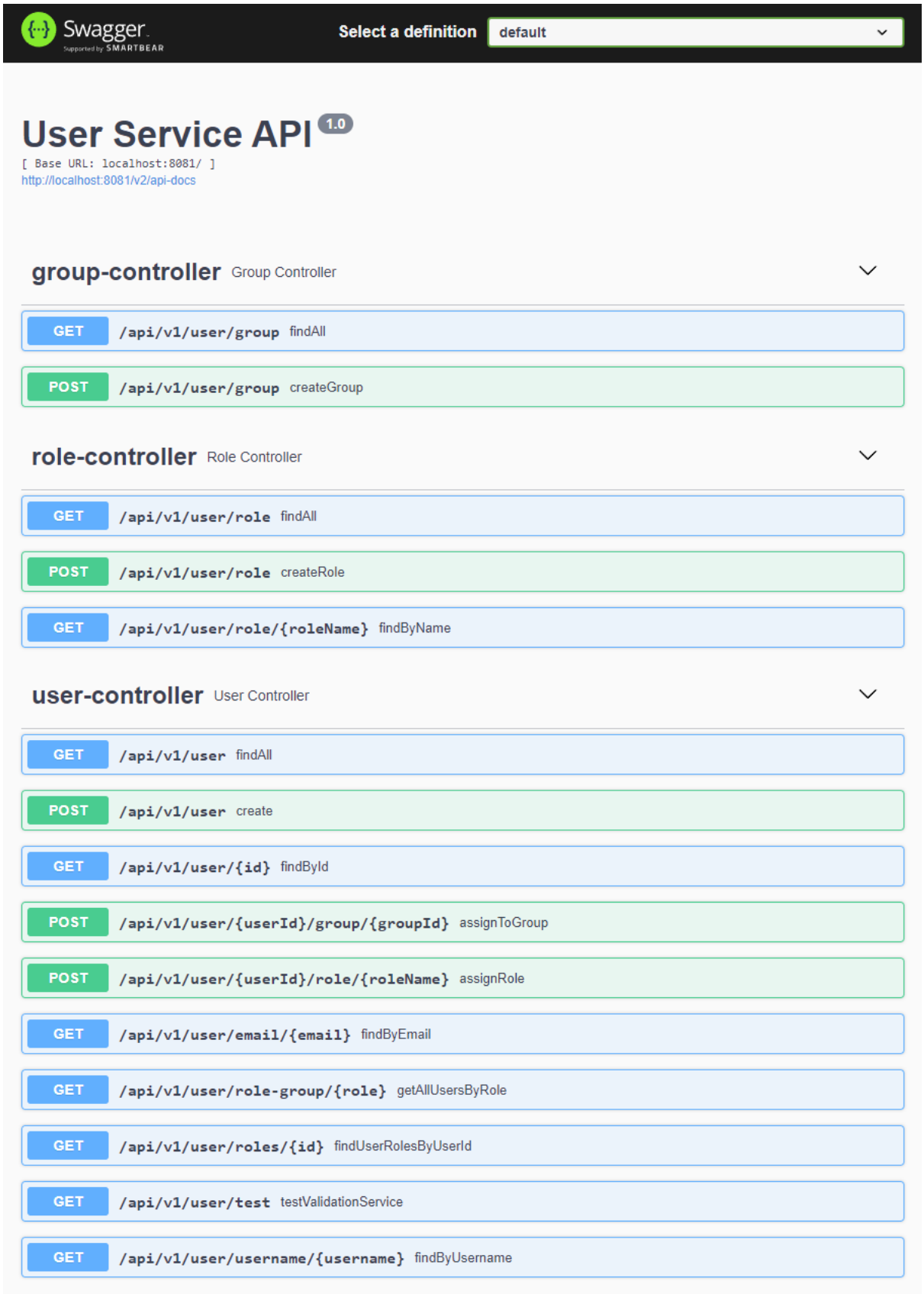
Εικόνα 7: Zipkin - Validation flow

Όπως φαίνεται από την υπογραφή της μεθόδου uploadFile, δέχεται ως όρισμα Flux<T> και επιστρέφει Mono<T>. Οι κλήσεις που πραγματοποιούνται από το Validation Service προς τα Blockchain και Document services είναι επίσης non-blocking.

```
public interface ValidationService {
    // Testing Method
    Mono<String> testResponse();
    // Upload file
    Mono<File> uploadFile(Flux<FilePart> filePart);
    // Delete temporarily stored files
    void deleteTempFiles();
}
```

User Service

Παρακάτω παρουσιάζονται τα endpoints του User Service:



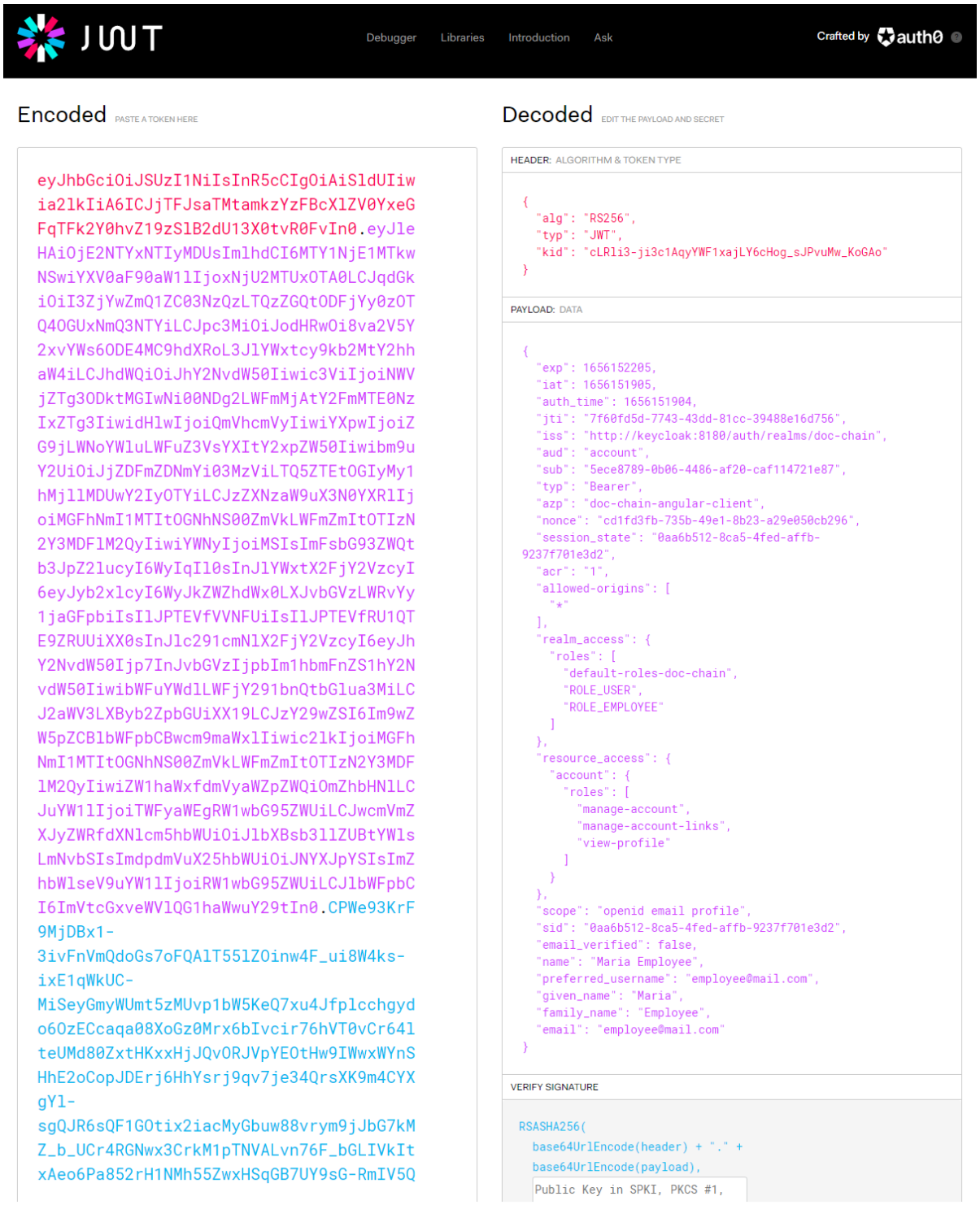
Εικόνα 8: User Service API

Το User Service συνδέεται με τον διακομιστή ταυτοποίησης και εξουσιοδότησης Keycloak με χρήση κατάλληλων dependencies στο pom.xml του service. Με αυτό τον τρόπο, παρέχεται πρόσβαση στη βάση δεδομένων Postgres μέσω του Keycloak για την αναζήτηση και διαχείριση των χρηστών της εφαρμογής. Η αναζήτηση μπορεί να γίνει βάσει του id, του email, του username ή του ρόλου χρήστη. Οι διαχειριστές του συστήματος έχουν τη δυνατότητα να δημιουργήσουν χρήστες, ρόλους και ομάδες χρηστών, καθώς επίσης, μπορούν να αναθέσουν ρόλους στους χρήστες και να ορίσουν τα μέλη μιας ομάδας. Στις ομάδες χρηστών μπορούν να προστεθούν ρόλοι, τους οποίους κληρονομούν οι χρήστες που ανήκουν στην ομάδα. Για παράδειγμα, ένας χρήστης που ανήκει στο studentGroup, κληρονομεί ρόλους STUDENT_ROLE και USER_ROLE και ένας χρήστης που ανήκει στο employeeGroup, κληρονομεί ρόλους EMPLOYEE_ROLE και USER_ROLE.

Το User Service καλείται από το Document Service για την αναζήτηση των φοιτητών πριν περάσει στην ψηφιοποίηση των εγγράφων τους. Επίσης καλείται απευθείας από τη frontend εφαρμογή μέσω του Api Gateway για τη διαχείριση των ρόλων χρηστών από την τοποθεσία ιστού του διαχειριστή.

Security Implementation και διακομιστής Keycloak

Όταν ένας μη ταυτοποιημένος χρήστης επισκεφθεί μια προστατευμένη τοποθεσία της frontend εφαρμογής (angular single page application), τότε ανακατευθύνεται στη σελίδα σύνδεσης χρήστη http:/keycloak:8180/auth/realms/doc-chain/protocol/openid-connect/auth/. Μετά την εισαγωγή των στοιχείων, πραγματοποιείται έλεγχος στον διακομιστή ταυτοποίησης Keycloak. Αν ο χρήστης ανήκει στο σύστημα, τότε αποστέλλεται authorization code στη frontend εφαρμογή και στη συνέχεια ανταλλάσσεται με JSON Web Token (JWT) το οποίο αποθηκεύεται προσωρινά στον web browser. Στη συνέχεια, ο χρήστης ανακατευθύνεται στη frontend εφαρμογή.



Εικόνα 9: jwt.io

Παραπάνω απεικονίζεται (εικόνα 9) το access token που λαμβάνει η frontend εφαρμογή από τον διακομιστή ταυτοποίησης. Στην αποκωδικοποιημένη μορφή του, διακρίνονται τα πεδία του payload, τα οποία περιέχουν στοιχεία του χρήστη όπως το id, το όνομα, το email και πίνακας με τους ρόλους που διαθέτει. Όπως φαίνεται στο παράδειγμα, πρόκειται για χρήστη με ρόλο employee.

Άλλες βασικές πληροφορίες που περιέχει το payload του token είναι:

- Auth time (auth time): Χρονική στιγμή αυθεντικοποίησης σε μορφή Epoch Unix Timestamp
- Issue timestamp (iat): Χρονική στιγμή έκδοσης του token σε μορφή Epoch Unix Timestamp.
- Expiration (exp): Χρονική στιγμή λήξης του token σε μορφή Epoch Unix Timestamp.
- Server Issuer Identification (iss): Η διεύθυνση του auth server που παράγεται το token.
- Type (typ): Τύπος του access token (Bearer).
- Authorized party (azp): Ο client, για λογαριασμό του οποίου, παράγεται το token.
- JWT Id (jti): Id που έχει παραχθεί από τον authentication server προκειμένου να αναγνωρίσει το token.

Όλες οι κλήσεις (request) που πραγματοποιούνται από έναν ταυτοποιημένο χρήστη στο backend μέσω της frontend εφαρμογής, περιέχουν Bearer Authorization header με το access token που έχει εκδοθεί:

```
curl --location --request GET 'localhost:8080/api/v1/document/test' --
header 'Authorization: Bearer {token}'
```

Όταν ένα request φτάσει στο Api Gateway, τότε το Api Gateway επικοινωνεί με τον διακομιστή Keycloak προκειμένου να διαπιστωθεί η εγκυρότητα του token και στη συνέχεια, η κλήση δρομολογείται στο microservice που αποτελεί τον τελικό αποδέκτη του request. Το request συνεχίζει να φέρει το Authorization header με το access token (token relay pattern), προκειμένου να επαναληφθεί η διαδικασία επικύρωσης του token από το microservice αποδέκτη.

Εξαίρεση στην προαναφερθείσα διαδικασία, αποτελεί το Validation Microservice, διότι η πρόσβαση στην υπηρεσία επικύρωσης εγγράφων είναι δημόσια. Παρόλα αυτά, όταν το Validation Service λαμβάνει ένα δημόσιο request, τότε επικοινωνεί με τον διακομιστή Keycloak για να λάβει access token, προκειμένου να έχει πρόσβαση στα Blockchain και Document services που διαθέτουν προστατευμένους πόρους. Η ροή αυτή ονομάζεται client credential grand flow.

Τα microservices που διαθέτουν προστατευμένους πόρους και απαιτούν access token, κάνουν χρήση του παρακάτω dependency στο pom.xml:

```
<!-- OAuth2 Resource Server -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

Αντίστοιχα, τα microservices που επικοινωνούν με άλλα microservices με προστατευμένους πόρους, όπως το Validation Service, κάνουν χρήση του παρακάτω dependency στο pom.xml:

```
<!-- OAuth2 Client -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
```

Όλα τα microservices μπορούν να είναι Oauth2 Resource servers και Clients ταυτόχρονα.

Όλα τα microservices και το api gateway κάνουν χρήση του *@EnableWebFluxSecurity* annotation στο security configuration, που αποτελεί την reactive εκδοχή του *@EnableWebSecurity* annotation του Spring Security Framework.

Τα endpoints των microservices είναι προστατευμένα από μη εξουσιοδοτημένους χρήστες. Οι εξουσιοδοτήσεις έχουν υλοποιηθεί με τρόπο, ώστε να εξυπηρετούν τον σκοπό της παρούσας εργασίας. Σε production εφαρμογές θα πρέπει να επαναπροσδιοριστούν με αυστηρότερους κανόνες.

Παρακάτω παρουσιάζονται πίνακες με τις εξουσιοδοτήσεις χρηστών ανά service:

Document Service Path Matchers	Method	Access
/api/v1/document/document-hash/**	HttpMethod.*GET*	Permit all authenticated users[23]
/api/v1/document/get-file/**	HttpMethod.*GET*	ROLE_STUDENT, ROLE_EMPLOYEE
/api/v1/document/upload-file/**	HttpMethod.*POST*	ROLE_EMPLOYEE

Blockchain Service Path Matchers	Method	Access
/api/v1/blockchain/all	HttpMethod.*GET*	Permit all authenticated users
/api/v1/blockchain/number-of-student-blocks/**		
/api/v1/blockchain/blocks/**		
/api/v1/blockchain/block-id/**		
/api/v1/blockchain/block-index/**		
/api/v1/blockchain/mineBlock	HttpMethod.*POST*	ROLE_COM_INT

Validation Service Path Matchers	Method	Access
/api/v1/validation/validate-document	HttpMethod.*POST*	Permit All

User Service Path Matchers	Method	Access
/api/v1/user/**	HttpMethod.*GET*	Permit all authenticated users
/api/v1/user/**	HttpMethod.*POST*	ROLE_EMPLOYEE

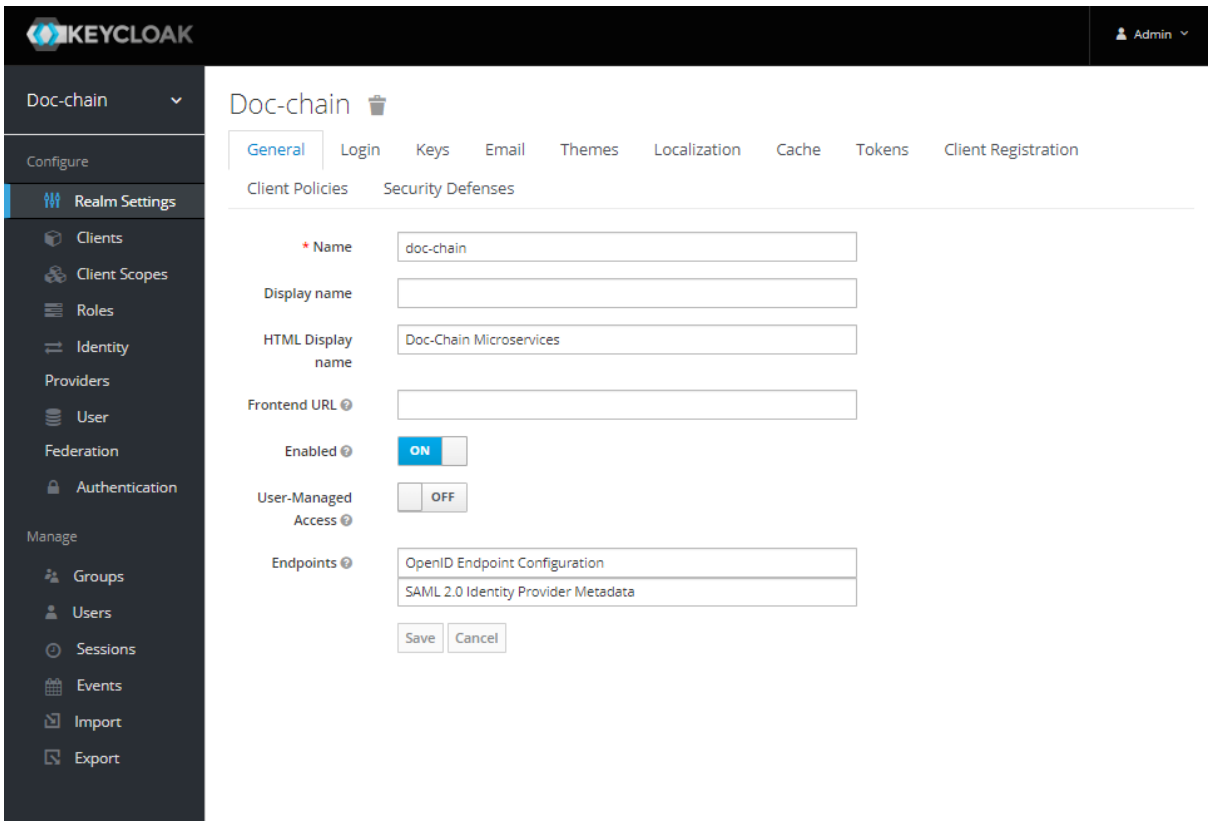
Api Gateway Path Matchers	Access
/api/v1/validation/validate-document	Public

[23] Ταυτοποιημένοι χρήστες (authenticated users) θεωρούνται όλοι όσοι διαθέτουν ένα ή περισσότερους από τους ρόλους ROLE_STUDENT, ROLE_EMPLOYEE, ROLE_ADMIN, ROLE_SUPER_ADMIN και ROLE_COM_INT.

Keycloak

Ο διαχειριστής του διακομιστή Keycloak έχει τη δυνατότητα να διαμορφώσει το περιβάλλον ταυτοποίησης και εξουσιοδότησης χρηστών και ρόλων της εφαρμογής, μέσα από ένα φιλικό προς τον χρήστη γραφικό περιβάλλον.

Στη κεντρική σελίδα του Keycloak, έχει δημιουργηθεί ένα Realm, μέσα από το οποίο παραμετροποιούνται τα επιμέρους στοιχεία που αφορούν στην ασφάλεια της εφαρμογής.



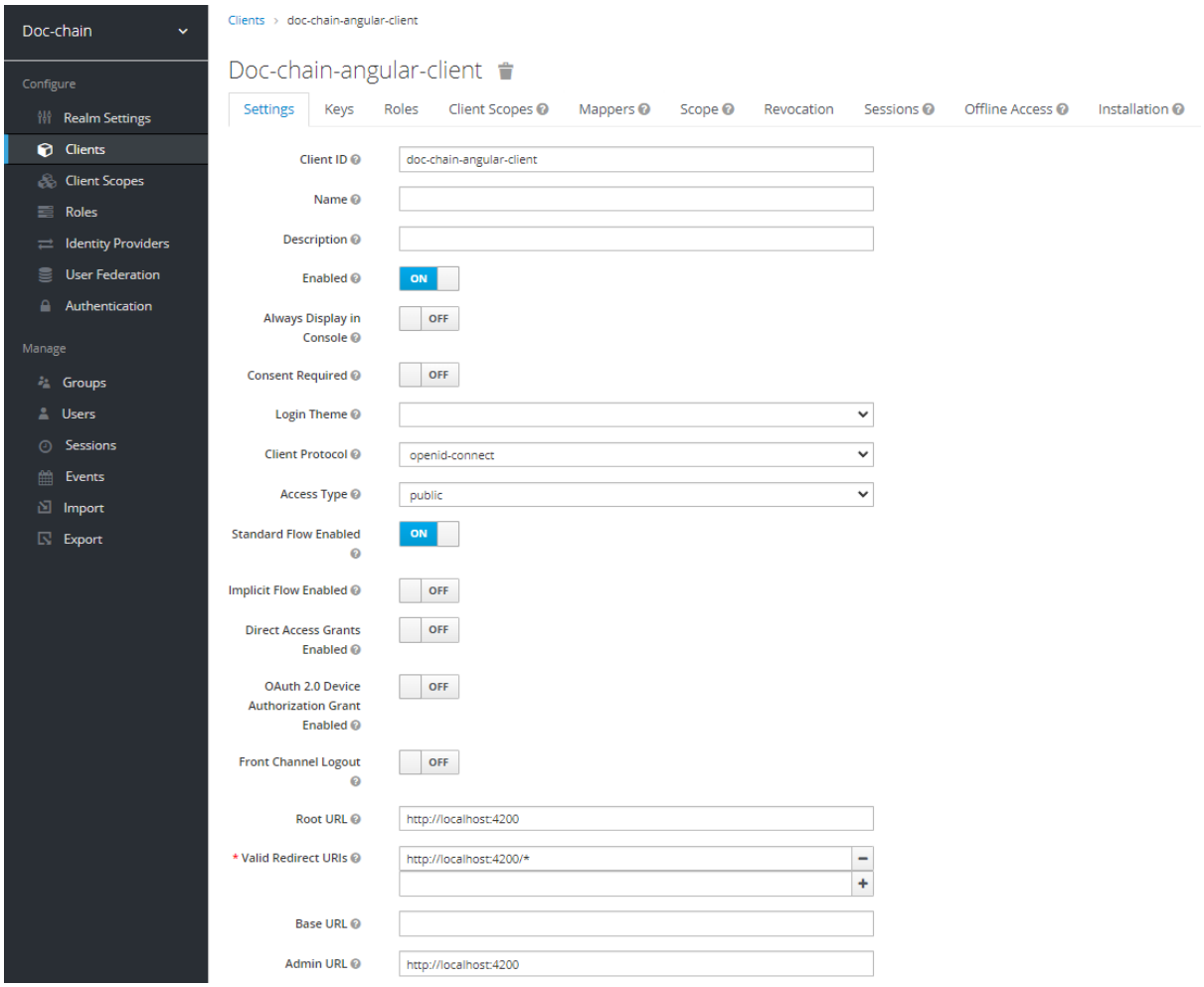
Εικόνα 10: Keycloak Realm creation

Το Realm παρέχει endpoints τα οποία χρησιμοποιούνται από όλα τα microservices της εφαρμογής για τον έλεγχο εγκυρότητας των access tokens, την έκδοση και ανανέωση τους, καθώς και την σύνδεση του χρήστη. Τα βασικότερα endpoints παρουσιάζονται στον παρακάτω πίνακα:

`issuer`	`http://keycloak:8180/auth/realms/doc-chain`
`authorization_endpoint`	`http://keycloak:8180/auth/realms/doc-chain/protocol/openid-connect/auth`
`token_endpoint`	`http://keycloak:8180/auth/realms/doc-chain/protocol/openid-connect/token`
`introspection_endpoint`	`http://keycloak:8180/auth/realms/doc-chain/protocol/openid-connect/token/introspect`
`userinfo_endpoint`	`http://keycloak:8180/auth/realms/doc-chain/protocol/openid-connect/userinfo`
`end_session_endpoint`	`http://keycloak:8180/auth/realms/doc-chain/protocol/openid-connect/logout`

Στην ενότητα Clients του Keycloak, έχουν δημιουργηθεί τρείς διαφορετικοί clients, ώστε να καλύψουν τις ανάγκες της εφαρμογής:

Doc-chain-angular-client: Αφορά στον client που χρησιμοποιείται από το frontend της εφαρμογής και ακολουθεί τη ροή του OAuth 2.0 Authorization code grant flow.

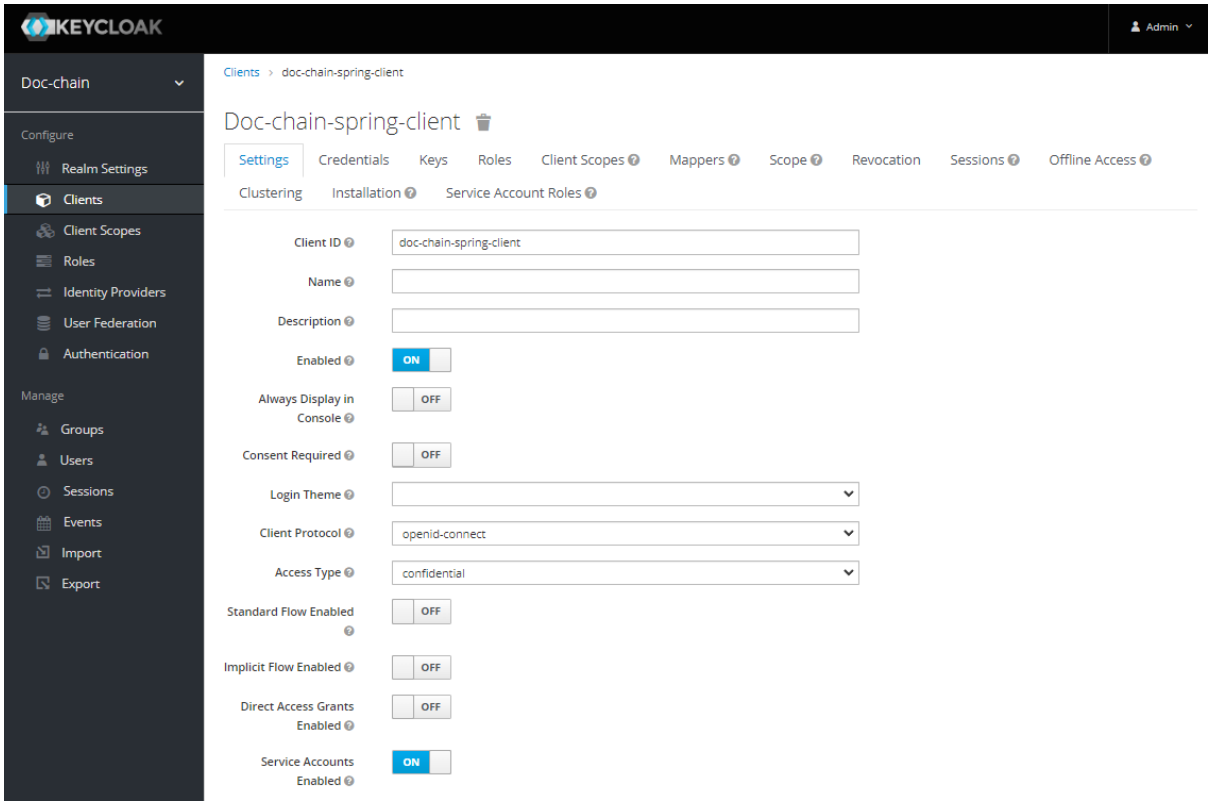


Εικόνα 11: Keycloak client - Doc-chain-angular-client

Η frontend εφαρμογή χρειάζεται την προσθήκη του παρακάτω κώδικα στο αρχείο app.module.ts:

```
function initializeKeycloak(keycloak: KeycloakService) {
  return () =>
    keycloak.init({
      config: {
        url: 'http://keycloak:8180/auth',
        realm: 'doc-chain',
        clientId: 'doc-chain-angular-client'
      },
      initOptions: {
        onLoad: 'check-sso',
        silentCheckSsoRedirectUri:
          window.location.origin + '/assets/silent-check-sso.html'
      }
    });
}

providers: [{ provide: APP_INITIALIZER, useFactory: initializeKeycloak, multi: true, deps:
[KeycloakService] }]
```

Doc-chain-spring-client: Ο doc-chain-spring-client χρησιμοποιείται από τα backend services, που απαιτείται να λάβουν access token προκειμένου να επικοινωνήσουν με άλλα microservices της εφαρμογής, όπως το document service, το validation service και το api gateway. Η ροή OAuth 2.0 που ακολουθείται είναι η Client credentials grant flow.



Εικόνα 12: Keycloak client - Doc-chain-spring-client

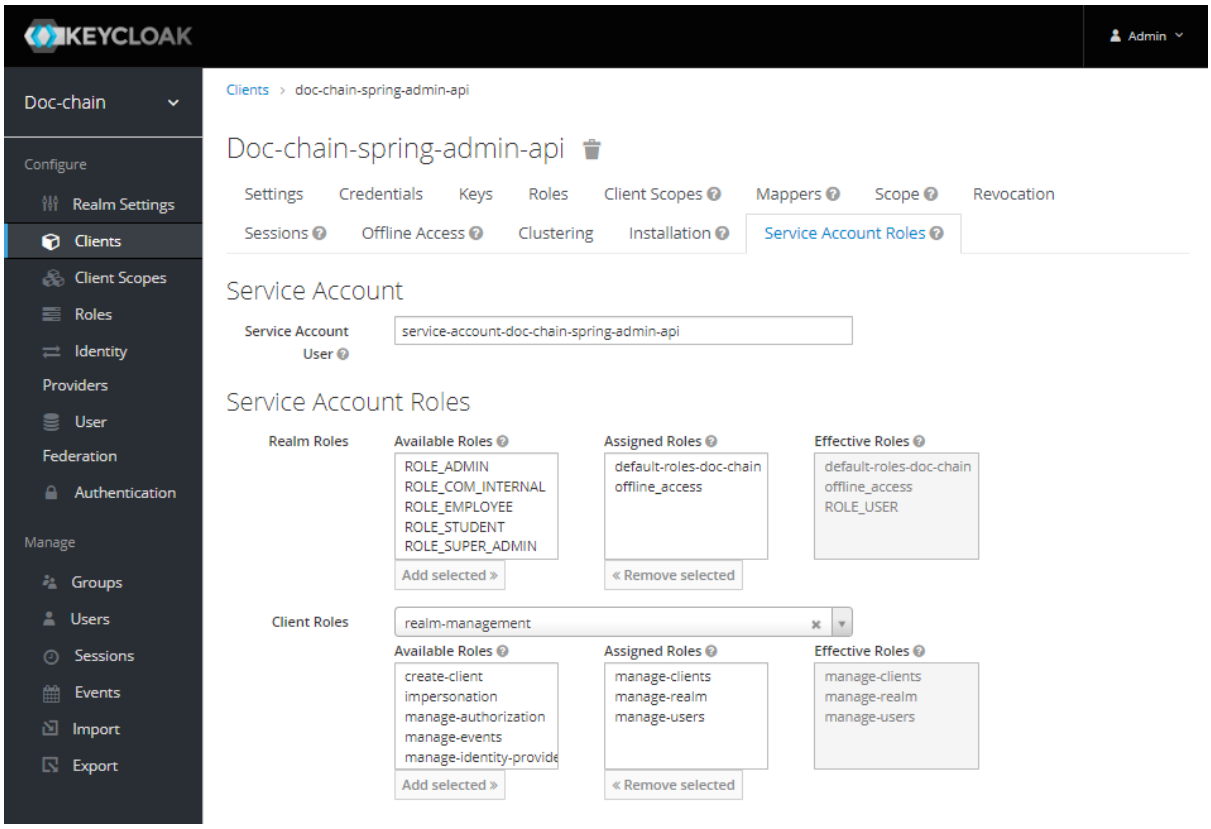
Στα services που κάνουν χρήση του Doc-chain-spring-client και του client credentials grant flow, απαιτείται παραμετροποίηση στο αρχείο application.yml (application.properties):

```
security:
  oauth2:
    client:
      provider:
        doc-chain-spring-client:
          issuer-uri: http://keycloak:8180/auth/realms/doc-chain
      registration:
        doc-chain-spring-client:
          client-id: doc-chain-spring-client
          client-secret: CM4IHbB4Y0xflF3aPz2An07kQjK4BxJ3
          authorization-grant-type: client_credentials
```

Επίσης, απαιτείται η δημιουργία κατάλληλου WebClient.Builder στο configuration του service.

```
@Bean
@LoadBalanced
public WebClient.Builder webClient(
        ReactiveClientRegistrationRepository clientRegistrations,
        ReactiveOAuth2AuthorizedClientService authorizedClientService) {
    ServerOAuth2AuthorizedClientExchangeFilterFunction oauth =
            new ServerOAuth2AuthorizedClientExchangeFilterFunction(
                    new AuthorizedClientServiceReactiveOAuth2AuthorizedClientManager(
                            clientRegistrations, authorizedClientService));
    return WebClient.builder().filter(oauth);
}
```

Doc-chain-spring-admin-api: Το Doc-chain-spring-admin-api χρησιμοποιείται αποκλειστικά από το User Service για τη διαχείριση χρηστών του Keycloak μέσω των APIs που παρέχει. Ομοίως με το Doc-chain-spring-client, χρησιμοποιεί OAuth 2.0 ροή Client credentials grant flow και η μοναδική διαφορά είναι ότι διαθέτει δικαιώματα διαχείρισης του realm (manage-clients, manage-realm, manage-users).



Εικόνα 13: Keycloak client creation - Doc-chain-spring-admin-api

Στα microservices που διαθέτουν προστατευμένους πόρους και χρειάζεται access token για την πρόσβαση σε αυτούς (OAuth 2.0 resource servers), απαιτείται παραμετροποίηση στο αρχείο application.yml (application.properties) ώστε να δηλωθεί το URI του issuer για την επικύρωση του access token:

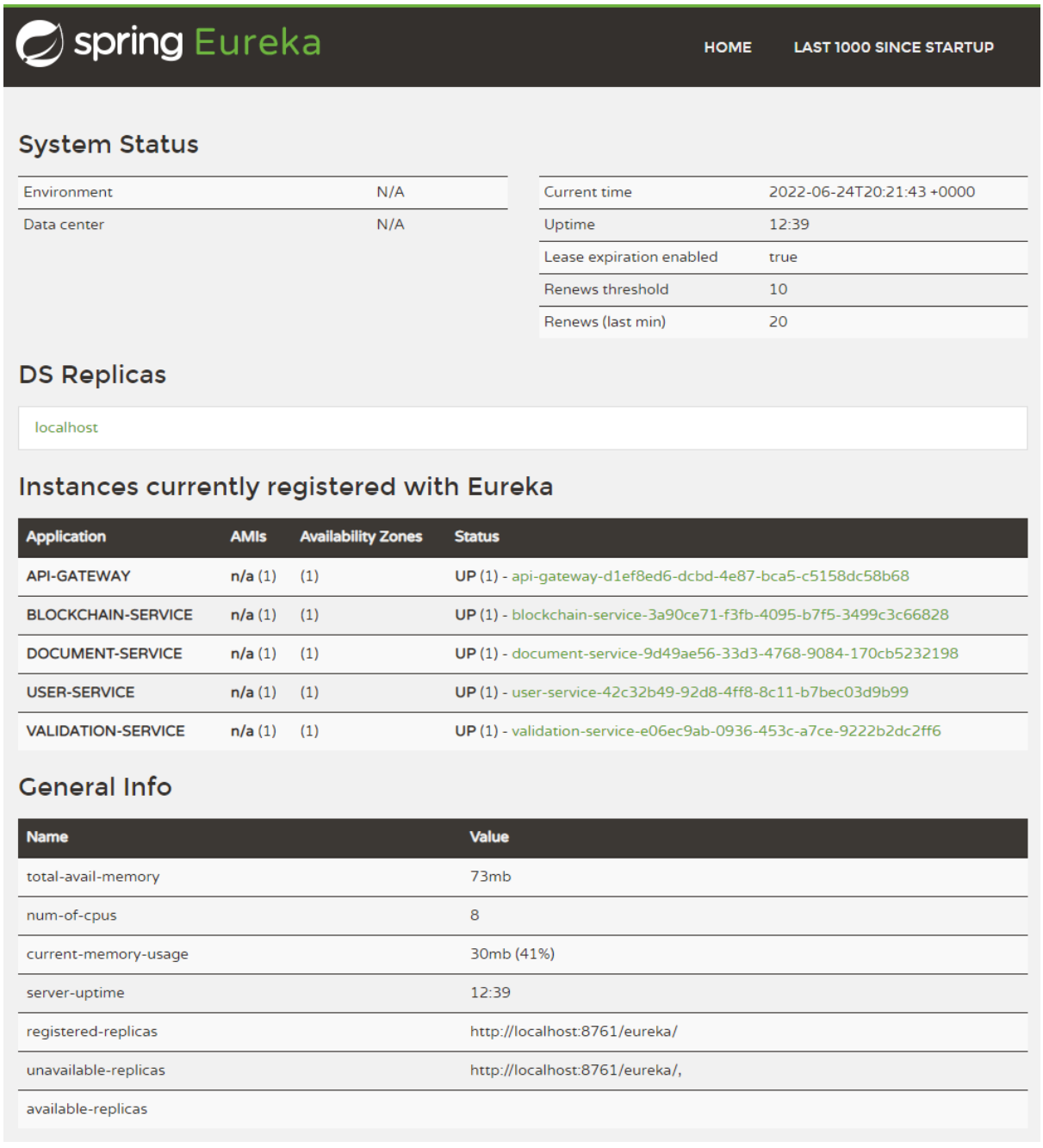
```
# OAuth2 Resource Server
resourceserver:
  jwt:
    issuer-uri: http://keycloak:8180/auth/realms/doc-chain
```

Τα request προς ένα OAuth 2.0 resource server αυθεντικοποιούνται με τη παρακάτω μέθοδο:

```
@Configuration
@EnableWebFluxSecurity
public class SecurityConfig {
    @Bean
    public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity httpSecurity) {
        return httpSecurity
                .anyExchange().authenticated()
                .and()
                .oauth2ResourceServer(ServerHttpSecurity.OAuth2ResourceServerSpec::jwt)
                .csrf().disable()
                .build();
    }
}
```

Naming Server

Ο naming ή eureka server αποθηκεύει προσωρινά τις πληροφορίες για όλα τα στιγμιότυπα των microservices της εφαρμογής. Στη παρακάτω εικόνα (εικόνα 14) διακρίνονται τα ονόματα των microservices που έχουν καταχωρηθεί στον eureka server (στήλη Application). Στη στήλη Status αναγράφονται τα στιγμιότυπα των services με το όνομα τους και ένα UUID, η κατάσταση τους (up) και το πλήθος των ενεργών στιγμιοτύπων. Άλλες πληροφορίες που αναφέρονται είναι το uptime του eureka server, οι ανανεώσεις (Renews), δηλαδή τα heartbeats από services που έχουν ληφθεί το τελευταίο λεπτό και τέλος το κατώφλι ανανεώσεων (Renews threshold). Αν το πλήθος ανανεώσεων γίνει μικρότερο από το κατώφλι ανανεώσεων, τότε ενεργοποιείται ο μηχανισμός «self-preservation mode» που σημαίνει πως ο server σταματά να κάνει expire τα instances, ώστε να προστατέψει την τρέχουσα κατάσταση των instances που είναι καταχωρημένα.



Εικόνα 14: Eureka Server

Api Gateway

Για την υλοποίηση του Api Gateway της εφαρμογής, έγινε χρήση του dependency spring-cloud-starter-gateway του spring cloud framework στο pom.xml του service. Η δρομολόγηση των αιτημάτων στα microservices, μπορεί να οριστεί απευθείας στο application.yml, χωρίς να είναι απαραίτητη η δημιουργία configuration bean.

Παρακάτω παρουσιάζεται η δρομολόγηση των αιτημάτων με επανορισμό διαδρομής.

```
# Routes Configuration
routes:
  - id: user-service
    uri: lb://user-service
    predicates:
      - Path=/api/v1/user/**
  - id: document-service
    uri: lb://document-service
    predicates:
      - Path=/api/v1/document/**
  - id: blockchain-service
    uri: lb://blockchain-service
    predicates:
      - Path=/api/v1/blockchain/**
  - id: validation-service
    uri: lb://validation-service
    predicates:
      - Path=/api/v1/validation/**
  - id: naming-server
    uri: http://localhost:8761
    predicates:
      - Path=/eureka/apps
```

Όλα τα URIs που ξεκινούν από /api/v1/user/ δρομολογούνται στο User Service με επανορισμό της διαδρομής σε lb://user-service/api/v1/user/. Ομοίως και τα υπόλοιπα requests, δρομολογούνται στα microservices που αντιστοιχούν.

Το Api Gateway λειτουργεί επίσης ως loadbalancer, κατανέμοντας το φόρτο εργασίας σε όλα τα ενεργά instances ενός microservice. Για το λόγο αυτό, γίνεται χρήση του προθέματος lb:// μπροστά από κάθε URI που επαναορίζεται. Το lb://validation-service για παράδειγμα, δεν αντιστοιχεί σε ένα μοναδικό instance, αλλά σε όλα τα ενεργά instances του validation service που βρίσκονται καταχωρημένα στον eureka server. Αν υπάρχουν τρία ενεργά instances, τότε τα request θα δρομολογηθούν στα τρία instances με τη σειρά, βάσει του προεπιλεγμένου αλγόριθμου round robin.
Στη παρακάτω εικόνα (εικόνα 15) του eureka server, έχουν δημιουργηθεί τρία instance του Validation Service. Το Api Gateway δρομολογεί τα request και στα τρία instances με τη σειρά.

Application	AMIs	Availability Zones	Status
API-GATEWAY	n/a (1)	(1)	UP (1) - api-gateway-e2f3d429-5e9f-4b30-8813-1add4196708c
BLOCKCHAIN-SERVICE	n/a (1)	(1)	UP (1) - blockchain-service-1ac3c35e-4f88-4afa-b242-0e010bc1c4a5
DOCUMENT-SERVICE	n/a (1)	(1)	UP (1) - document-service-c7a86e71-2bbf-4b9b-bcd6-60de027c4ffa
USER-SERVICE	n/a (1)	(1)	UP (1) - user-service-8232983a-94b5-4a6b-b3e8-3b26d3473556
VALIDATION-SERVICE	n/a (3)	(3)	UP (3) - validation-service-9a1ae05f-9cd3-423a-9deb-0fa6b16f097e , validation-service-6959b969-4f52-472e-87d3-1139346db83e , validation-service-a9e2a1cf-e81b-4398-bc6f-f2468e342bf8

Εικόνα 15: Validation service scaling

Docker images και deployment

Για τη δημιουργία docker images, χρησιμοποιήθηκε το plugin jib-maven-plugin στο parent pom.xml και το eclipse-temurin 17 image. Το Jib εκκινεί τη διαδικασία δημιουργίας των docker images όταν καλείται το maven goal package (mvn package). Όταν ολοκληρωθεί η δημιουργία των docker images, το Jib προχωρά στη μεταφόρτωση τους στο DockerHub. Αν πραγματοποιηθούν αλλαγές στον κώδικα της εφαρμογής, τρέχοντας το mvn package goal, το Jib μεταφορτώνει μόνο τις αλλαγές στο DockerHub, αντί για ολόκληρο το docker image.

Η frontend εφαρμογή χρησιμοποιεί Dockerfile για τη δημιουργία docker image. Με τις παρακάτω εντολές δημιουργείται το image και στη συνέχεια μεταφορτώνεται στο DockerHub:

```
docker build -t {DockerHub username}/doc-chain-frontend:latest .
```

```
docker push {DockerHub username}/doc-chain-frontend:latest
```

Application Deployment – Docker compose

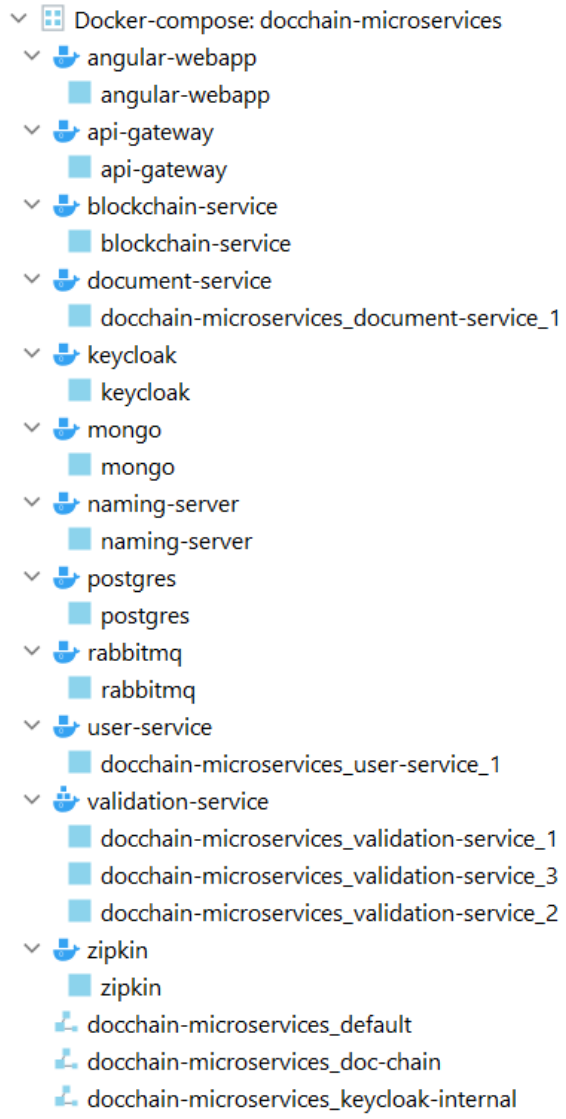
Για την ανάπτυξη της εφαρμογής σε ένα διακομιστή, έχει δημιουργηθεί αρχείο docker-compose.yml. Σε αυτό, παραμετροποιούνται όλες οι λεπτομέρειες για τη δημιουργία των docker containers της εφαρμογής.

```
version: "3.8"
services:
  postgres:
    image: postgres
    container_name: postgres
    volumes:
      - postgres_data:/var/lib/postgresql/data
    environment:
      POSTGRES_DB: keycloak
      POSTGRES_USER: keycloak
      POSTGRES_PASSWORD: password
    ports:
      - "5432:5432"
    networks:
      - keycloak-internal
    restart: unless-stopped

  mongo:
    image: mongo
    container_name: mongo
    ports:
      - "27017:27017"
    volumes:
      - mongo:/data/mongo
    networks:
      - doc-chain
    restart: unless-stopped

  zipkin:
    image: openzipkin/zipkin
    container_name: zipkin
    ports:
      - "9411:9411"
    networks:
      - doc-chain
    restart: unless-stopped

  keycloak:
    image: quay.io/keycloak/keycloak:16.1.1
    container_name: keycloak
    environment:
      DB_VENDOR: POSTGRES
      DB_ADDR: postgres
      DB_DATABASE: keycloak
      DB_USER: keycloak
      DB_SCHEMA: public
      DB_PASSWORD: password
      KEYCLOAK_USER: admin
      KEYCLOAK_PASSWORD: admin
      JAVA_OPTS: "-
Djboss.socket.binding.port-offset=100"
      KEYCLOAK_FRONTEND_URL:
http://keycloak:8180/auth
      KEYCLOAK_IMPORT: /tmp/doc-chain-
realm.json
    volumes:
      - ./.config/keycloak/doc-chain-
realm.json:/tmp/doc-chain-realm.json
    ports:
      - "8180:8180"
    networks:
      - doc-chain
      - keycloak-internal
    healthcheck:
      test: ["CMD-SHELL", "curl --fail
http://keycloak:8180/auth/realms/doc-
chain"]
      interval: 30s
      timeout: 3s
      start_period: 10s
      retries: 3
    depends_on:
      - postgres
    restart: unless-stopped
```

```yaml
  rabbitmq:
    image: rabbitmq:3-management
    container_name: rabbitmq
    environment:
      - RABBITMQ_DEFAULT_USER=guest
      - RABBITMQ_DEFAULT_PASS=guest
    ports:
      - "5672:5672"
      - "15672:15672"
    networks:
      - doc-chain
    restart: unless-stopped

  # Doc-Chain Services
  naming-server:
    image: {user}/doc-chain-naming-server:latest
    container_name: naming-server
    environment:
      - SPRING_PROFILES_ACTIVE=docker
    ports:
      - "8761:8761"
    networks:
      - doc-chain
      - keycloak-internal
    restart: unless-stopped

  user-service:
    image: {user}/doc-chain-user-service
    environment:
      - SPRING_PROFILES_ACTIVE=docker
    networks:
      - doc-chain
      - keycloak-internal
    depends_on:
      - keycloak
      - naming-server
    restart: unless-stopped

  blockchain-service:
    image: {user}/doc-chain-blockchain-service
    container_name: blockchain-service
    environment:
      - SPRING_PROFILES_ACTIVE=docker
    networks:
      - doc-chain
      - keycloak-internal
    depends_on:
      - mongo
      - naming-server
    restart: unless-stopped

  # Doc-Chain Frontend
  angular-webapp:
    image: {user}/doc-chain-frontend:latest
    container_name: angular-webapp
    ports:
      - "4200:80"
    restart: unless-stopped


  networks:
    doc-chain:
      driver: bridge
    keycloak-internal:
      driver: bridge
```

```yaml
  document-service:
    build:
      context: .
      dockerfile: document-
service/Dockerfile
    environment:
      - SPRING_PROFILES_ACTIVE=docker
    volumes:
      - document:/.tmp
    networks:
      - doc-chain
      - keycloak-internal
    depends_on:
      mongo:
        condition: service_started
      naming-server:
        condition: service_started
      keycloak:
        condition: service_healthy
    restart: unless-stopped

  validation-service:
    image: {user}/doc-chain-validation-
service
    environment:
      - SPRING_PROFILES_ACTIVE=docker
    volumes:
      - validation:/.tmp
    networks:
      - doc-chain
      - keycloak-internal
    depends_on:
      naming-server:
        condition: service_started
      keycloak:
        condition: service_healthy
    deploy:
      replicas: 3 # Scale up to 3 instances
    restart: unless-stopped

  api-gateway:
    image: {user}/doc-chain-api-
gateway:latest
    container_name: api-gateway
    environment:
      - SPRING_PROFILES_ACTIVE=docker
    ports:
      - "8080:8080"
    networks:
      - doc-chain
      - keycloak-internal
    depends_on:
      naming-server:
        condition: service_started
      blockchain-service:
        condition: service_started
      keycloak:
        condition: service_healthy
    restart: unless-stopped

  volumes:
    postgres_data:
    mongo:
    validation:
    document:
```

Για το deployment της εφαρμογής χρησιμοποιείται η εντολή: `docker compose up -d`



Εικόνα 16: docker compose up

Στη παραπάνω εικόνα (εικόνα 16) διακρίνονται όλα τα containers της εφαρμογής, καθώς και τα δίκτυα που έχουν δημιουργηθεί. Το Validation Service διαθέτει τρία instances διότι έχει γίνει scale up κατά το deployment με την παράμετρο deploy.replicas: 3 στο docker-compose.yml.

Παρουσίαση εφαρμογής frontend

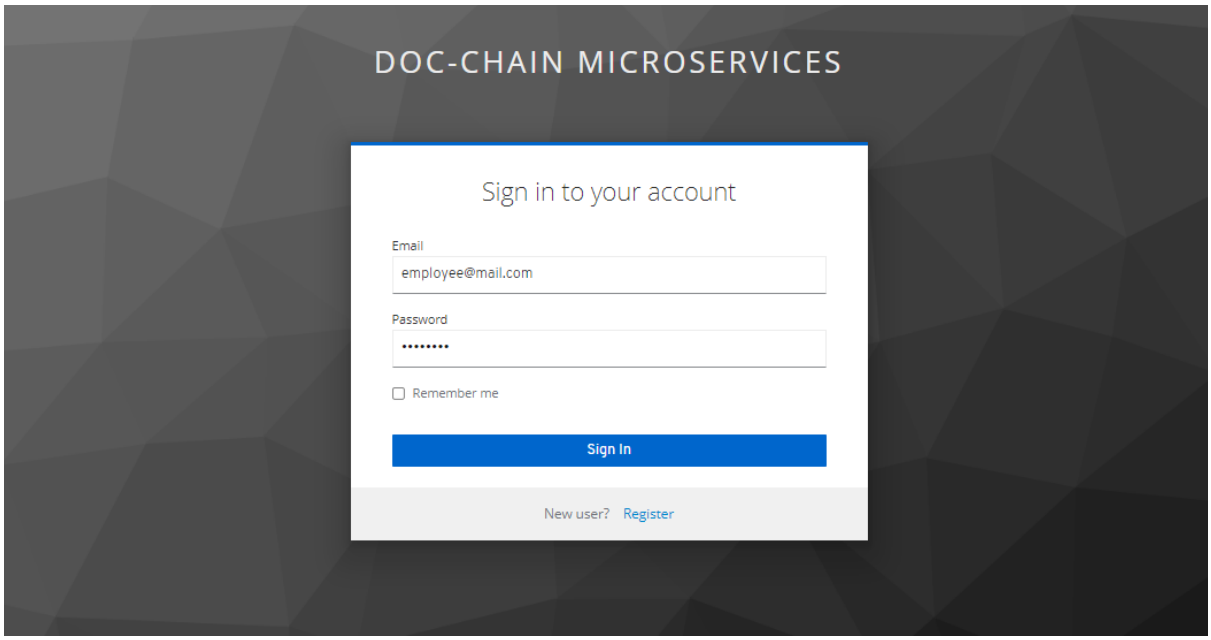
Το responsive single page application έχει δημιουργηθεί στο πλαίσιο της μεταπτυχιακής διατριβής σε Angular 13 framework και γλώσσα Typescript. Σκοπός του είναι, η παρουσίαση της λειτουργικότητας της backend εφαρμογής που είναι βασισμένη σε αρχιτεκτονική microservices.

Κεντρική σελίδα:

Εικόνα 17: Home page

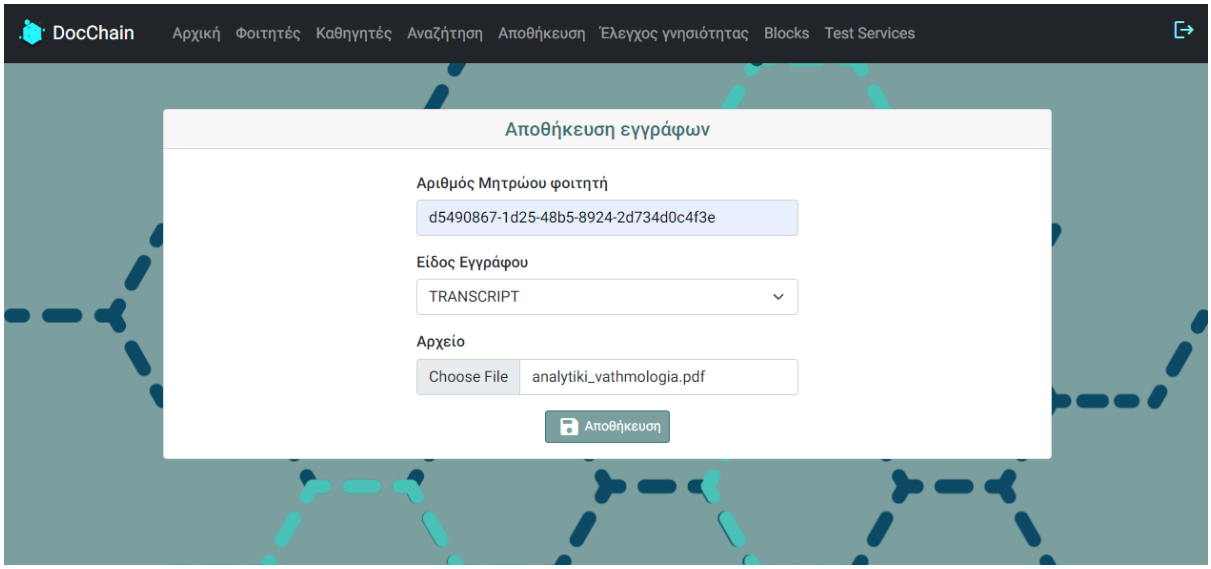
Σενάριο χρήσης υπαλλήλου εκπαιδευτικού ιδρύματος

Ο χρήστης employee συνδέεται στην εφαρμογή (εικόνα 18) προκειμένου να καταχωρίσει έγγραφα φοιτητών στο blockchain. Μετά την επιτυχημένη σύνδεση, ανακατευθύνεται στη κεντρική σελίδα.



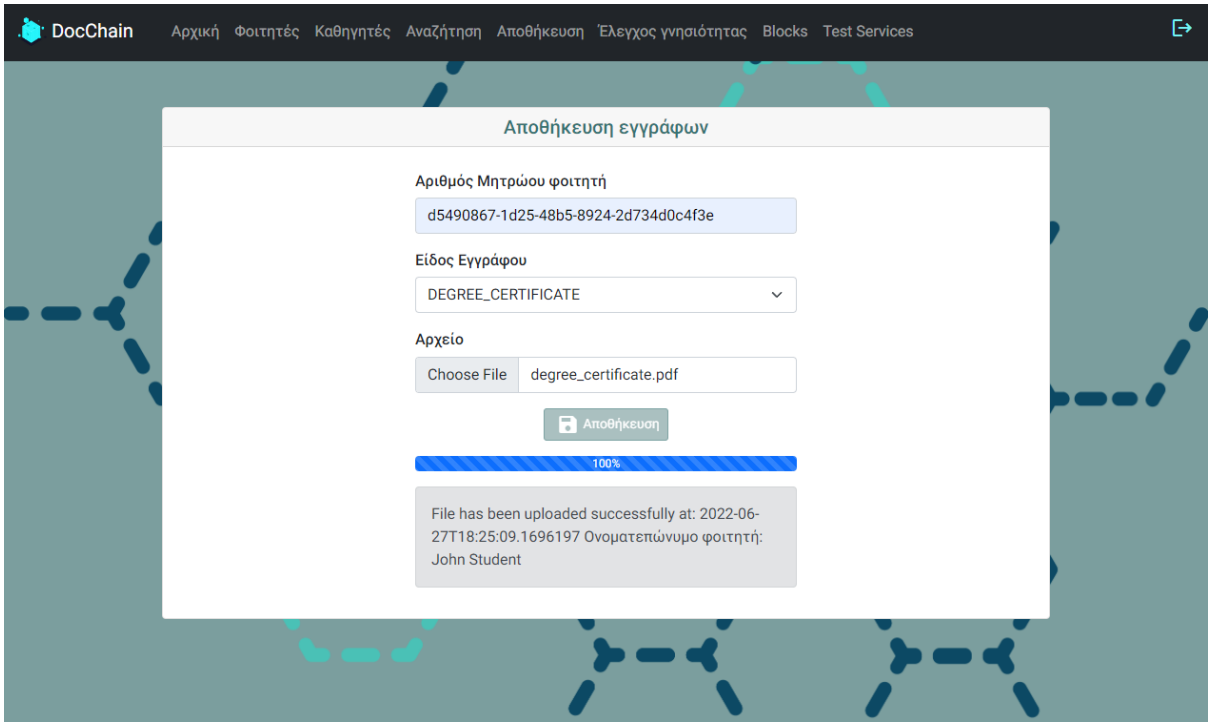
Εικόνα 18: Login page

Επιλέγει Αποθήκευση – Ανέβασμα αρχείου από το κεντρικό μενού, στη συνέχεια συμπληρώνει τα πεδία με τον αριθμό μητρώου του φοιτητή, το είδος εγγράφου και επιλέγει το αρχείο προς αποθήκευση. Ο τύπος αρχείων που γίνεται δεκτός από την εφαρμογή είναι pdf, png και jpg.



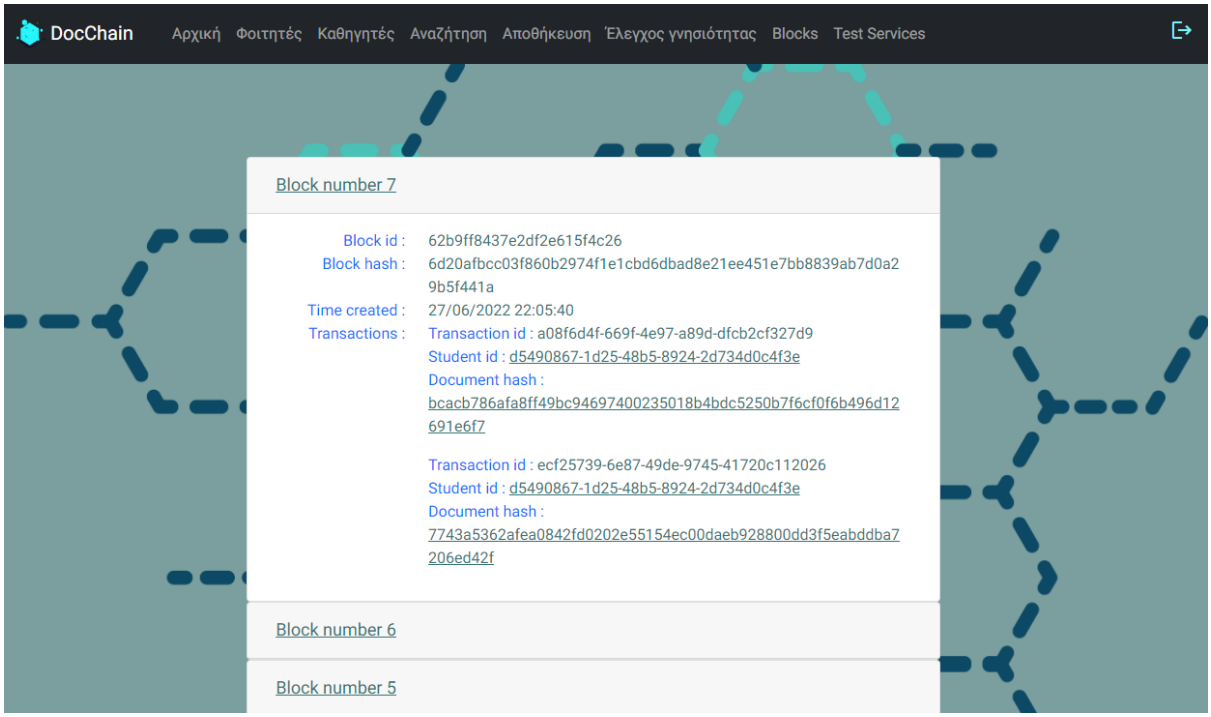
Εικόνα 19: Upload document form

Όταν ο χρήστης πιέσει το πλήκτρο «Αποθήκευση», τότε ξεκινά η μεταφόρτωση του αρχείου στο Document Service. Όταν η διαδικασία της μεταφόρτωσης ολοκληρωθεί, ακολουθεί σχετικό μήνυμα (εικόνα 20). Το μήνυμα διαγράφεται και η φόρμα καθαρίζει μετά από λίγα δευτερόλεπτα. Η διαδικασία μεταφόρτωσης εγγράφου έχει ολοκληρωθεί με επιτυχία, το έγγραφο έχει αποθηκευτεί στη βάση δεδομένων, έχει αποσταλεί ασύγχρονο μήνυμα στο Blockchain Service και το transaction έχει δημιουργηθεί στη βάση δεδομένων. Τα επόμενα λεπτά, το ψηφιακό αποτύπωμα hash SHA-256 του εγγράφου, θα περάσει στο επόμενο block της αλυσίδας.



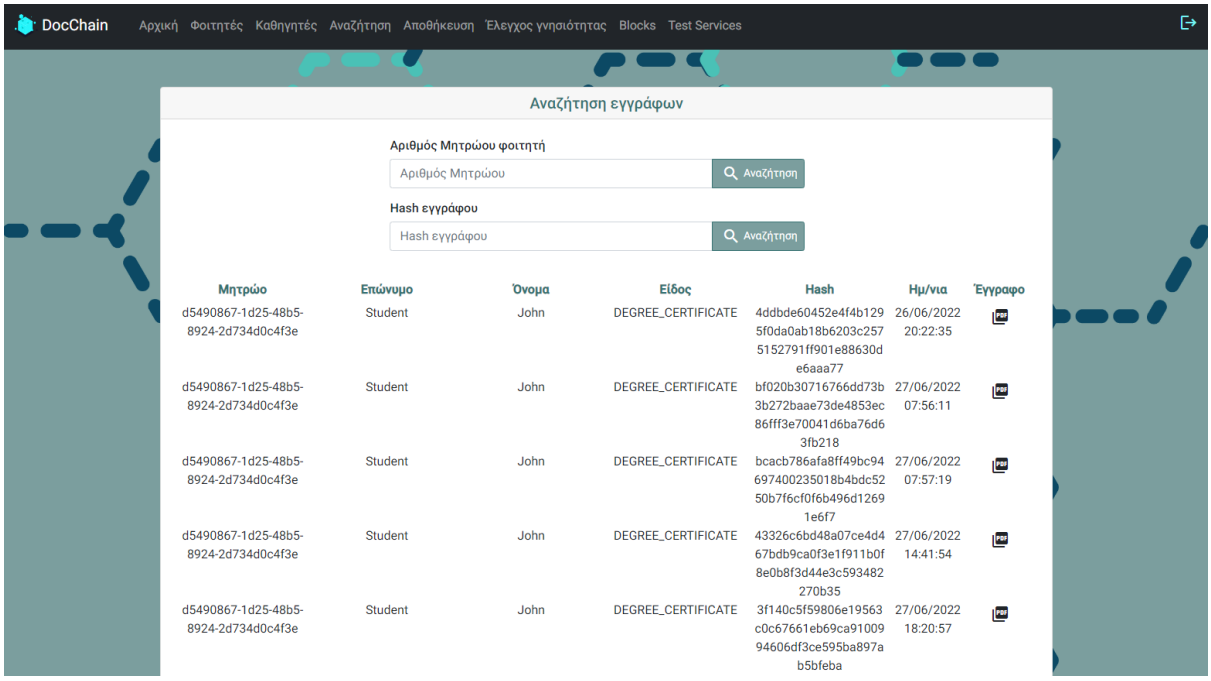
Εικόνα 20: Successful file upload

Μετά από λίγα λεπτά, τα έγγραφα εμφανίζονται καταχωρημένα στο blockchain (εικόνα 21):



Εικόνα 21: Documents in blockchain

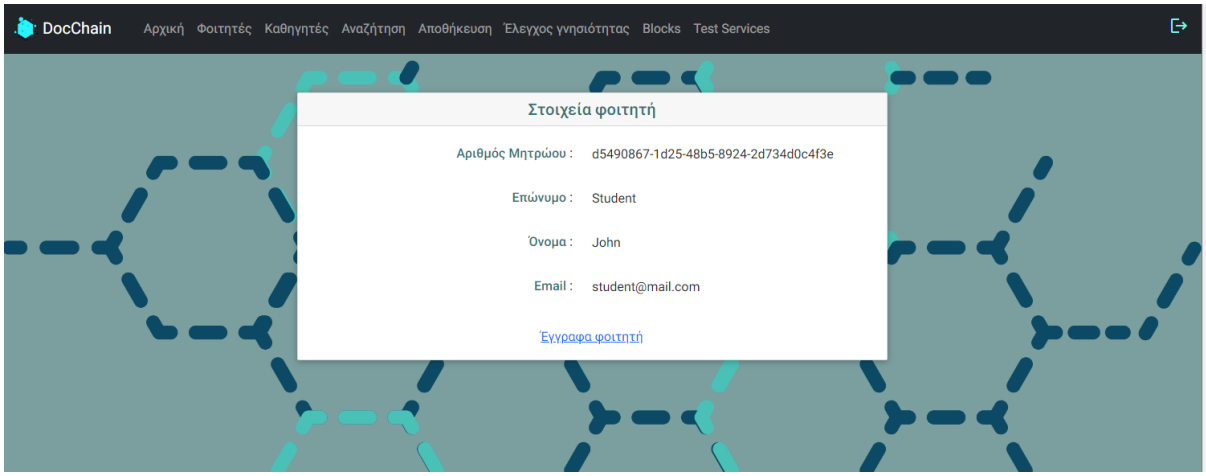
Ο χρήστης employee έχει τη δυνατότητα να μεταβεί στην αναζήτηση εγγράφων, επιλέγοντας το link του αριθμού μητρώου του φοιτητή ή το hash ενός εγγράφου, ώστε να δει τα στοιχεία του φοιτητή που αφορά το κάθε έγγραφο ή hash αντίστοιχα. Αν επιλεγεί ο αριθμός μητρώου του φοιτητή, τότε γίνεται μετάβαση στη τοποθεσία αναζήτησης εγγράφων, όπου εμφανίζονται όλα τα έγγραφα που έχουν καταχωρηθεί στην αλυσίδα και αφορούν τον συγκεκριμένο αριθμό μητρώου φοιτητή (εικόνα 22). Από εκεί, δίνεται η δυνατότητα στον χρήστη να κατεβάσει τα έγγραφα.



Εικόνα 22: Document search

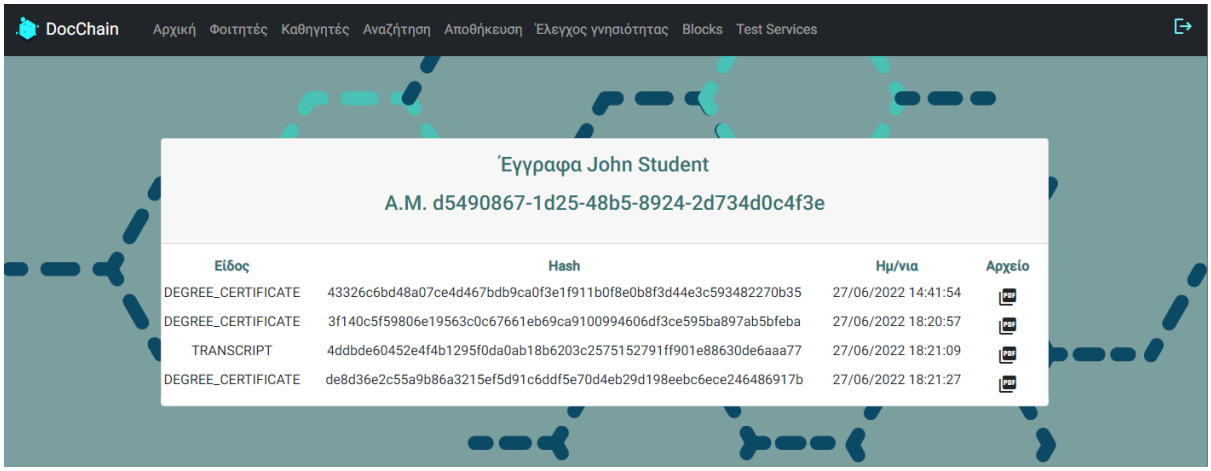
Σενάριο χρήσης φοιτητή εκπαιδευτικού ιδρύματος

Όταν ένας χρήστης με ρόλο student συνδεθεί στην εφαρμογή, έχει τη δυνατότητα να προβάλει την καρτέλα με τα στοιχεία του από τη γραμμή πλοήγησης (εικόνα 23).



Εικόνα 23: Student section

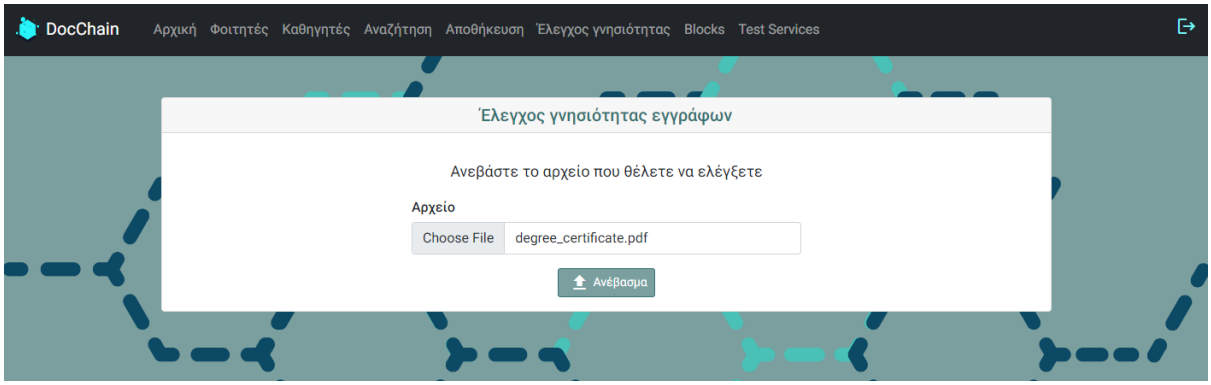
Από εκεί, μπορεί να επιλέξει τον σύνδεσμο «Έγγραφα φοιτητή», ώστε να μεταβεί στη τοποθεσία με τα έγγραφα του (εικόνα 24). Ο χρήστης με ρόλο student, έχει τη δυνατότητα να κατεβάσει τα έγγραφα που βρίσκονται στο προφίλ του.



Εικόνα 24: Student documents

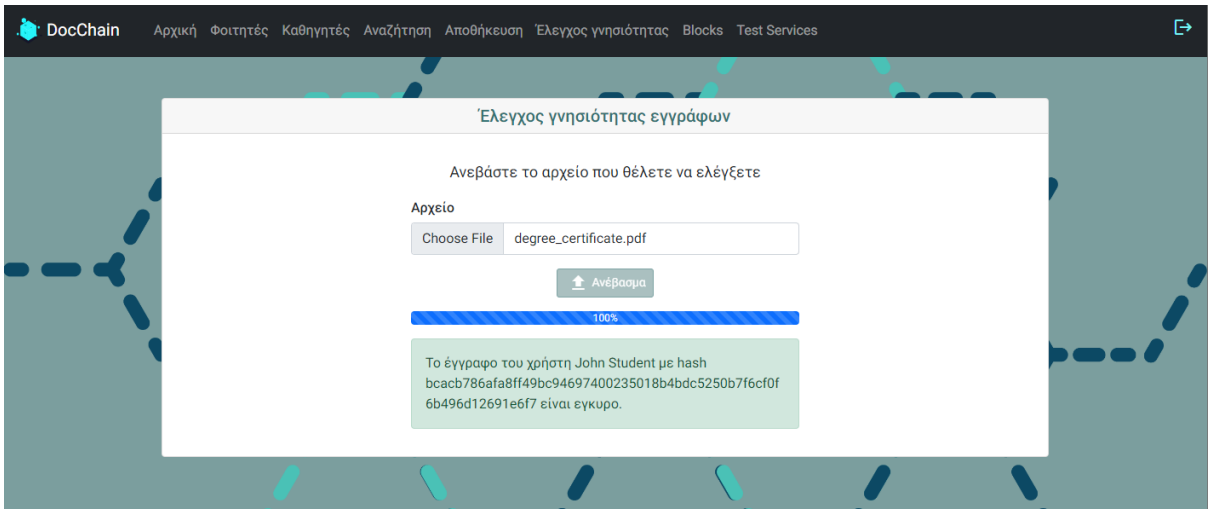
Έλεγχος γνησιότητας εγγράφων

Η λειτουργία του ελέγχου γνησιότητας εγγράφων δεν απαιτεί σύνδεση στην εφαρμογή και μπορεί να χρησιμοποιηθεί από οποιονδήποτε διαθέτει έγγραφα φοιτητών σε μορφή pdf. Οι επισκέπτες της σελίδας, μπορούν να μεταβούν στη τοποθεσία ελέγχου γνησιότητας από τη γραμμή πλοήγησης.



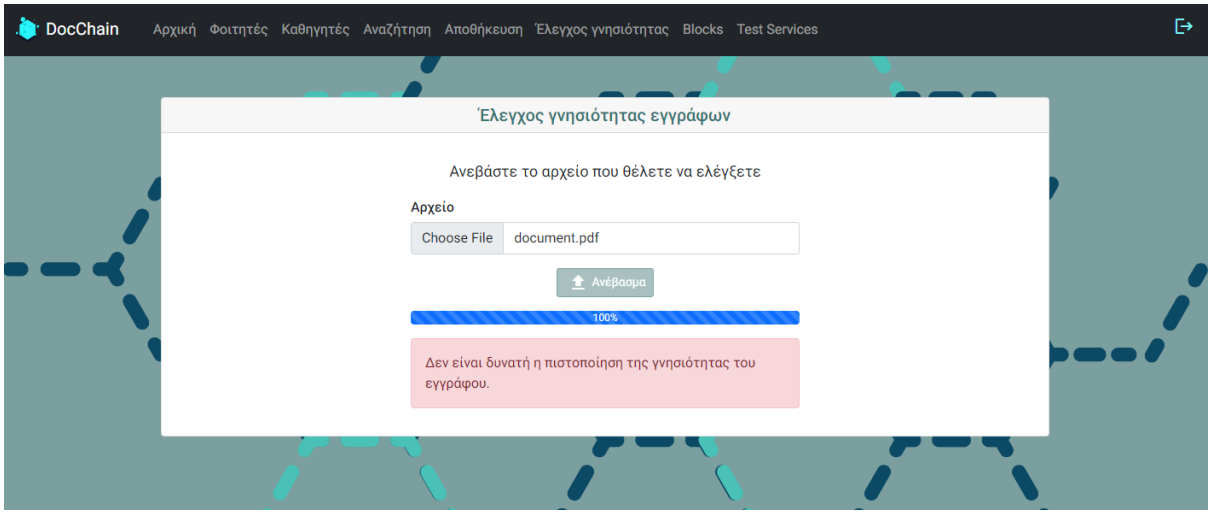
Εικόνα 25: Document validation

Όταν ο χρήστης της εφαρμογής πιέσει το πλήκτρο «Ανέβασμα», τότε το έγγραφο pdf μεταφορτώνεται σε ένα instance του Validation Service. Αν το hash του εγγράφου βρεθεί στην αλυσίδα, τότε επιστρέφει μήνυμα επιτυχούς ταυτοποίησης (εικόνα 26):



Εικόνα 26: Valid document

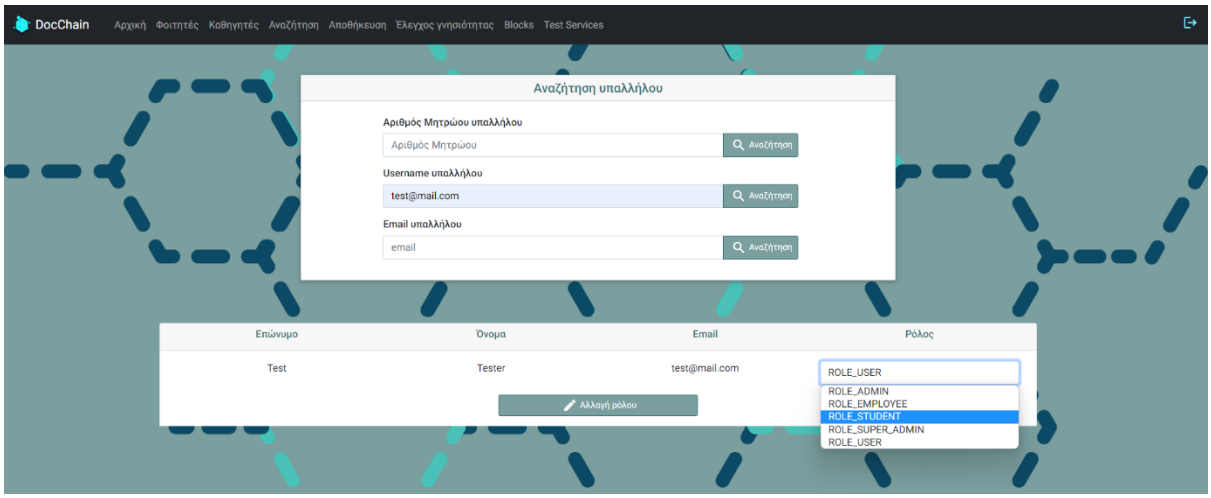
Όταν το hash του εγγράφου δε βρεθεί στην αλυσίδα, τότε επιστρέφει μήνυμα ανεπιτυχούς ταυτοποίησης (εικόνα 27):



Εικόνα 27: Invalid document

Ανάθεση ρόλων από τον διαχειριστή της εφαρμογής

Οι χρήστες με ρόλο διαχειριστή, έχουν εξουσιοδότηση να αλλάζουν ρόλους στους εγγεγραμμένους χρήστες της εφαρμογής. Μπορούν να αναζητήσουν ένα χρήστη βάσει του username ή του email και στη συνέχεια να αναθέσουν ένα ρόλο (εικόνα 28).



Εικόνα 28: Role assignment

Όταν ο χρήστης συνδεθεί στην εφαρμογή, θα μπορεί να επισκεφθεί τις τοποθεσίες της εφαρμογής που απαιτούν ανάλογη εξουσιοδότηση με τον ρόλο του.

Μελλοντικές επεκτάσεις

Μεταφορά και ενορχήστρωση των containers που αποτελούν την εφαρμογή, στην πλατφόρμα Kubernetes με χρήση Helm Charts και Service Mesh στην υλοποίηση Istio.

Kubernetes

Το όνομα Kubernetes (k8s) σημαίνει «Κυβερνήτης» και πρόκειται για μια πλατφόρμα ανοικτού κώδικα της Google. Βασικός σκοπός ενός Kubernetes cluster είναι η ανάπτυξη (deployment), η διαχείριση και η εύρυθμη λειτουργία των microservices που βρίσκονται σε containers. Επίσης, δίνει τη δυνατότητα αναβαθμίσεων των υπηρεσιών με zero-downtime, χρησιμοποιώντας τεχνικές όπως blue, green και canary deployments.

Το Kubernetes μπορεί να διαχειρίζεται τα pods, τα οποία αποτελούνται από containers και βρίσκονται μέσα σε ένα κόμβο (node). Δίνεται η δυνατότητα ελέγχου της κατάστασης ενός pod και διαχείρισης των στιγμιοτύπων του, ανάλογα με το φόρτο εργασίας, πραγματοποιώντας οριζόντιο autoscaling ή manual scaling. Επίσης παρέχεται η δυνατότητα παραμετροποίησης και βελτιστοποίησης των πόρων του κόμβου (cluster), χρησιμοποιώντας ποσόστωση στη χρήση CPU και μνήμης από κάθε node ή group από nodes.

Μια ακόμα βασική λειτουργία του Kubernetes είναι η παροχή service discovery των pods και των containers που βρίσκονται σε λειτουργία μέσα στον κόμβο. Τα Service objects του Kubernetes, μπορούν να δρομολογούν κλήσεις, προς το εσωτερικό των pods και να παρέχουν εξισορρόπηση φόρτου. Επίσης, το Ingress object μπορεί να δρομολογεί κλήσεις που προέρχονται από έξω, προς το εσωτερικό ενός κόμβου (node).

Στο εσωτερικό ενός cluster, το Kubernetes παρέχει δίκτυο με IPs, όπου το κάθε pod δεσμεύει τη δική του διεύθυνση, ώστε να μπορεί να επικοινωνήσει με τα υπόλοιπα pods, ανεξάρτητα από τον κόμβο (node) στον οποίο βρίσκεται.

Helm charts

Το Helm είναι ένα package manager ανοικτού κώδικα για την διαχείριση των αρχείων .yml που χρησιμοποιούνται για την δημιουργία και παραμετροποίηση εφαρμογών σε περιβάλλον Kubernetes.

Η ανάπτυξη (deployment) μιας εφαρμογής που αποτελείται από microservices στο Kubernetes, απαιτεί τη δημιουργία αρχείων manifest που να δηλώνουν την επιθυμητή κατάσταση των Service και Deployment objects, όπως επίσης και αρχεία ρυθμίσεων ConfigMaps και Secrets. Η λειτουργία της δήλωσης της επιθυμητής κατάστασης είναι πολύ χρήσιμη, διότι μεταθέτει την ευθύνη στο Kubernetes, ώστε να ενεργήσει αυτοματοποιημένα προκειμένου να φέρει την πραγματική κατάσταση της εφαρμογής, όσο πιο κοντά είναι εφικτό, στην επιθυμητή κατάσταση. Για παράδειγμα αν έχει δηλωθεί ως επιθυμητή κατάσταση, ένα pod να διαθέτει τρία instances και το ένα σταματήσει να αποκρίνεται, τότε το Kubernetes θα προσπαθήσει αυτόματα, να δημιουργήσει ένα νέο instance.

Η δημιουργία και διατήρηση των αρχείο δήλωσης επιθυμητής κατάστασης, μπορεί να γίνει αρκετά πολύπλοκη διαδικασία, ιδιαίτερα όταν αυξάνεται το πλήθος των microservices. Η χρήση των Helm Charts διευκολύνει τους προγραμματιστές, παρέχοντας templates με πρότυπα αρχεία .yml και προκαθορισμένες τιμές για όλες τις κατηγορίες ρυθμίσεων. Οι τιμές μπορούν στη συνέχεια να τροποποιηθούν ανάλογα με τις προδιαγραφές κάθε εφαρμογής.

Τα Helm Charts έχουν προκαθορισμένη δομή και περιγράφουν τον τρόπο με τον οποίο θα αναπτυχθούν όλα τα στοιχεία που αποτελούν την εφαρμογή σε διαφορετικά περιβάλλοντα, όπως development, testing και production. Το Helm, διαθέτει επίσης τη δυνατότητα διατήρησης ιστορικού των αλλαγών και επαναφορά των ρυθμίσεων σε προηγούμενες εκδόσεις.

Service Mesh - Istio

Το service mesh είναι ένα επίπεδο υποδομής (infrastructure layer) που ελέγχει και παρακολουθεί την επικοινωνία μεταξύ των υπηρεσιών σε περιβάλλον Kubernetes. Οι δυνατότητες που παρέχει το service mesh είναι η ασφάλεια, επιβολή πολιτικής (policy enforcement), παρατηρησιμότητα (observability), ανθεκτικότητα (resilience) και διαχείριση της κίνησης (traffic management). Οι παραπάνω δυνατότητες υλοποιούνται με τον έλεγχο και παρακολούθηση όλων των εσωτερικών επικοινωνιών μέσα στο service mesh, όπως οι επικοινωνίες μεταξύ των microservices.

Ένα από τα βασικά στοιχεία του service mesh, είναι ένας ελαφρύς διακομιστής proxy που τοποθετείται σε κάθε microservice. Όλη η κίνηση των δεδομένων, από και προς το microservice, διέρχεται από τον διακομιστή proxy. Τα δεδομένα συλλέγονται από ένα επίπεδο ελέγχου του service mesh, το οποίο συλλέγει επίσης, δεδομένα τηλεμετρίας που χρησιμοποιούνται για την οπτικοποίηση της ροής δεδομένων μέσα στο service mesh.

Το Istio είναι μια υλοποίηση του service mesh και παρέχει δυνατότητες όπως Ingress gateway, το οποίο ρυθμίζει την εισερχόμενη και εξερχόμενη κίνηση από το service mesh, Virtual Services για τον ορισμό κανόνων δρομολόγησης από το Ingress Istio gateway προς τα Kubernetes services και μεταξύ των services. Όπως επίσης, διαθέτει δυνατότητες για Destination Rules, Peer Authentication (mutual TLS), Request Authentication και Authorization με χρήση JSON Web token σε πρότυπα Oauth2.0 και OpenID Connect. Όλες οι παραπάνω λειτουργίες, ελέγχονται μέσω της CLI kube-injected εντολής istioctl. Τέλος, το Istio διαθέτει δυνατότητα distributed tracing που ονομάζεται Jaeger και observability tool, το οποίο διαθέτει γραφικό περιβάλλον και ονομάζεται Kiali.

Συμπεράσματα

Η παρούσα μεταπτυχιακή διατριβή είχε ως στόχο την ανάπτυξη ολοκληρωμένης εφαρμογής που ενσωματώνει τεχνολογία Blockchain και βασίζεται στην αρχιτεκτονική των μικροϋπηρεσιών, με χρήση του προγραμματιστικού μοντέλου reactive programming. Η εφαρμογή υλοποιήθηκε λαμβάνοντας υπόψη βέλτιστες πρακτικές και μοτίβα σχεδίασης μικροϋπηρεσιών και δόθηκε ιδιαίτερη έμφαση στο θέμα της ασφάλειας.

Πραγματοποιήθηκε αναφορά στις δυσκολίες που προκύπτουν με τη χρήση του μοντέλου των μικροϋπηρεσιών, καθώς και στα σχεδιαστικά μοτίβα που επιλύουν πολλά από τα ζητήματα που προκύπτουν. Επίσης, έγινε αναλυτική παρουσίαση των τεχνολογιών και εργαλείων που χρησιμοποιήθηκαν, καθώς και ανάλυση του σχεδιασμού και της υλοποίησης της εφαρμογής. Τέλος, παρουσιάστηκε ο τρόπος με τον οποίο μπορεί να πραγματοποιηθεί το deployment της εφαρμογής με χρήση της τεχνολογίας Docker και docker compose.

Βιβλιογραφία - Αναφορές

1. Richardson, C. (2019). *Microservices patterns: with examples in Java*. Shelter Island, New York: Manning Publications.
2. Larsson, M. (2021). *Microservices with Spring Boot and Spring Cloud : build resilient and scalable microservices using Spring Cloud, Istio, and Kubernetes*. Birmingham: Packt Publishing.
3. Marinescu, D.C. (2018). *Cloud computing : theory and practice*. 2nd ed. Cambridge, Ma: Elsevier.
4. spring.io. (n.d.). *Spring microservices.* [online] Available at: https://spring.io/microservices.
5. docs.spring.io. (n.d.). *Web on Reactive Stack*. [online] Available at: https://docs.spring.io.
6. docs.spring.io. (n.d.). *Spring Cloud Gateway*. [online] Available at: https://docs.spring.io.
7. spring.io. (2019). *Securing Services with Spring Cloud Gateway*. [online] Available at: https://spring.io.
8. spring.io. (n.d.). *Service Registration and Discovery*. [online] Available at: https://spring.io.
9. Richardson, C. (2017). *Microservices.io*. [online] microservices.io. Available at: https://microservices.io.
10. projectreactor.io. (n.d.). *reactor-core 3.4.19*. [online] Available at: https://projectreactor.io.
11. www.rabbitmq.com. (n.d.). *Documentation: Table of Contents — RabbitMQ*. [online] Available at: https://www.rabbitmq.com.
12. Apache Kafka. (n.d.). *Apache Kafka*. [online] Available at: https://kafka.apache.org.
13. www.keycloak.org. (n.d.). *Keycloak*. [online] Available at: https://www.keycloak.org.
14. koserwal, A. (2021). *Running Keycloak using PostgreSQL database*. [online] Keycloak. Available at: https://medium.com.
15. Almeida, D. (2020). *Creating a File Upload component with Angular and RxJS*. [online] Medium. Available at: https://medium.com.
16. Zoysa, D.D. (2020). *Securing Spring Boot REST APIs with Keycloak*. [online] DevOps Dudes. Available at: https://medium.com.
17. Shoshi, E. (2021). *Step by Step Procedure of Spring Webflux multipart file upload and read each line without saving it*. [online] Medium. Available at: https://medium.com.
18. Ch, K. and rakant (2020). *Concurrency in Spring WebFlux | Baeldung*. [online] www.baeldung.com. Available at: https://www.baeldung.com.
19. baeldung (2018). *Spring Security OAuth Login with WebFlux | Baeldung*. [online] www.baeldung.com. Available at: https://www.baeldung.com.
20. Auth0 (n.d.). *Authorization Code Flow*. [online] Auth0 Docs. Available at: https://auth0.com.
21. piotr.minkowski (2020). *Spring Cloud Gateway OAuth2 with Keycloak*. [online] Piotr's TechBlog. Available at: https://piotrminkowski.com.
22. developer.okta.com. (n.d.). *OAuth 2.0 and OpenID Connect Overview | Okta Developer*. [online] Available at: https://developer.okta.com.
23. Codersee | Kotlin, Ktor, Spring. (2020). *Keycloak with Spring Boot and Kotlin- Introduction*. [online] Available at: https://codersee.com.

24. Codersee | Kotlin, Ktor, Spring. (2020). *How to Set Up Keycloak Admin Client with Spring Boot and Kotlin?* [online] Available at: https://codersee.com.
25. www.elastic.co. (n.d.). *Running the Elastic Stack ('ELK') on Docker | Getting Started [8.2] | Elastic*. [online] Available at: https://www.elastic.co.
26. zipkin.io. (n.d.). *OpenZipkin · A distributed tracing system*. [online] Available at: https://zipkin.io.
27. Sr, R.K. (2021). *If You Want To Understand Blockchain, See This Java Implementation*. [online] Medium. Available at: https://betterprogramming.pub.
28. dzone.com. (n.d.). *Blockchain and Distributed Ledger Technology for Documents - DZone Refcardz*. [online] Available at: https://dzone.com.
29. www.mongodb.com. (n.d.). *MongoDB Java Reactive Streams — MongoDB Drivers*. [online] Available at: https://www.mongodb.com.
30. www.mongodb.com. (n.d.). *GridFS — Java*. [online] Available at: https://www.mongodb.com.
31. IBM Cloud Education (2022). *What Is Optical Character Recognition (OCR)?* [online] www.ibm.com. Available at: https://www.ibm.com.
32. GitHub. (2022). *Jib*. [online] Available at: https://github.com.
33. Docker Documentation. (2019). *Docker Documentation*. [online] Available at: https://docs.docker.com.