Πανεπιστήμιο Πειραιώς – Τμήμα Πληροφορικής
Πρόγραμμα Μεταπτυχιακών Σπουδών
«Πληροφορική»

Μεταπτυχιακή Διατριβή

Τίτλος Διατριβής	Ανάπτυξη εφαρμογής Android για την αξιοποίηση αισθητήρων και δημιουργία microservices για αποθήκευση και απεικόνιση Big Data Android application development for mobile sensor utilization and development of microservices for storing and visualization of Big Data
Ονοματεπώνυμο Φοιτητή	Μαρία Κακογιάννου
Πατρώνυμο	Παναγιώτης
Αριθμός Μητρώου	ΜΠΠΛ/17015
Επιβλέπων	Ευθύμιος Αλέπης, Αναπληρωτής Καθηγητής

Ημερομηνία Παράδοσης Δεκέμβριος 2021

Τριμελής Εξεταστική Επιτροπή

(υπογραφή)	(υπογραφή)	(υπογραφή)
Ευθύμιος Αλέπης Αναπληρωτής Καθηγητής	Μαρία Βίρβου Καθηγήτρια	Κωνσταντίνος Πατσάκης Αναπληρωτής Καθηγητής

Περιεχόμενα

Πίνακας Εικόνων

Περίληψη

Καθώς η τεχνολογία εξελίσσεται και η νέα εποχή χαρακτηρίζεται από αυξημένες απαιτήσεις από ταχύτητα και ευελιξία, η μονολιθική αρχιτεκτονική προσέγγιση της κατασκευής ψηφιακών εφαρμογών έχει αποδειχθεί πλέον αναποτελεσματική και ξεπερασμένη. Τα αρχιτεκτονικά παραδείγματα στην ανάπτυξη λογισμικού αλλάζουν με την πάροδο του χρόνου. Οι απαιτήσεις για νέες τεχνολογικές προσεγγίσεις εξελίσσονται συνεχώς για να αντιμετωπίσουν το νέο σύνολο επιχειρηματικών προκλήσεων.

Ο σκοπός αυτής της εργασίας είναι να αξιολογήσει την προσέγγιση με ένα πείραμα στο σχεδιασμό ενός συστήματος microservice. Η διατριβή αυτή έχει ως σκοπό να αναλύσει τι είναι τα microservices, ποια είναι τα βασικά σημεία υλοποίησης, τη διαφορά τους από τη μονολιθική αρχιτεκτονική και εντέλει μέσω ανάπτυξης λογισμικού πως αυτά μπορούν να χρησιμοποιηθούν για να υποστηρίξουν Big Data Analytics. Τέλος, θα προταθούν προοπτικές για περαιτέρω μελέτη.

Η συγκεκριμένη αρχιτεκτονική δίνει λύση σε συστήματα όπου η πολυπλοκότητα ξεπερνά τη συνηθισμένη. Όσο μία εφαρμογή μεγαλώνει σε μέγεθος αλλά και σε πολυπλοκότητα η ανάπτυξη νέων feature αλλά και το debugging των ήδη υπαρχόντων γίνεται όλο και πιο δύσκολη και σε κάποιες περιπτώσεις αδύνατη. Σε τέτοιες συνθήκες έρχονται συχνά τα microservices να δώσουν λύσεις.

Abstract

As technology evolves and the new era is marked by increased demands for speed and flexibility, the monolithic architectural approach to digital application construction has now proved ineffective and outdated. Architectural examples in software development change over time. The demands for new technological approaches are constantly evolving to meet the new set of business challenges.

The purpose of this thesis is to evaluate the approach with an experiment in the design of a microservice system. This dissertation aims to analyze what microservices are, what their main points of implementation are, their difference from monolithic architecture and finally through software development how they can be used to support Big Data Analytics. Finally, prospects for further study will be proposed.

This architecture provides a solution to systems where the complexity exceeds the usual. As an application grows in size (Urdhwareshe, 2016) but also in complexity, the development of new features and the debugging of existing ones becomes more and more difficult and, in some cases, impossible. In such conditions, microservices often come to provide solutions.

1 Εισαγωγή

Τα microservices με την πάροδο του χρόνου εξελίσσονται και χρησιμοποιούνται όλο και περισσότερο και σε πολλές περιπτώσεις έρχονται να αντικαταστήσουν τις μονολιθικές αρχιτεκτονικές. Σε αυτό το κεφάλαιο, θα γίνει απόπειρα να συγκριθούν οι δυο αρχιτεκτονικές (μονολιθική και microservices) παραθέτοντας τα πλεονεκτήματα και τα μειονεκτήματα τους.
Έχουν υπάρξει διάφορες μελέτες μέχρι και σήμερα, οι οποίες υποστηρίζουν ότι ίσως τα microservices είναι απλά μια παραλλαγή της SOA (Service Oriented Architecture), που διορθώνει τις αδυναμίες και τις ελλείψεις της και δεν αποτελεί μία νέα παραλλαγή (Zimmermann, 2016), είτε μελετώνται οι ανάγκες οι οποίες οδήγησαν στην ανάπτυξη αυτής της αρχιτεκτονικής, καθώς και μελέτες για την αποδοτικότητά τους.
Παρόλα αυτά δεν υπάρχει μια ολοκληρωμένη έρευνα που να αποδεικνύει τις δυνατότητες των mcicroservices έναντι της SOA όσον αφορά την κλιμάκωση, την ελαστικότητα, τον χρόνο απόκρισης και την διέλευση των αιτημάτων κ.α.. Ταυτόχρονα δεν υπάρχει σχετική εμπειρική μελέτη ή έρευνα που να καταδεικνύει το καταλληλότερο πρωτόκολλο μεταφοράς για τα microservices, καθώς βάσει αρχιτεκτονικής είναι συμβατό με σχεδόν όλα τα πρωτόκολλα.

1.1 Μονολιθικές Εφαρμογές

Η αρχιτεκτονική SOA αναφέρεται από πολλούς επιστήμονες ως μονολιθική ή ημιμονολιθική αρχιτεκτονική. Ένα σύστημα λογισμικού ονομάζεται «μονολιθικό» εάν διαθέτει μονολιθική αρχιτεκτονική, στην οποία λειτουργικά διακριτές πτυχές (Urdhwareshe, 2016), για παράδειγμα είσοδος και έξοδος δεδομένων, επεξεργασία δεδομένων, χειρισμός σφαλμάτων και διεπαφή χρήστη, δεν είναι αρχιτεκτονικά ξεχωριστά στοιχεία αλλά όλα είναι συνυφασμένα.
Μια μονολιθική αρχιτεκτονική είναι το παραδοσιακό ενοποιημένο μοντέλο για το σχεδιασμό ενός προγράμματος λογισμικού και των μέσων που συνθέτουν όλα σε ένα κομμάτι. Το μονολιθικό λογισμικό είναι σχεδιασμένο να είναι αυτόνομο. Τα στοιχεία του προγράμματος είναι διασυνδεδεμένα και αλληλοεξαρτώμενα και όχι τόσο χαλαρά συνδεδεμένα όπως συμβαίνει με τα δομημένα προγράμματα λογισμικού (Havey, 2008) . Σε μια αυστηρά συζευγμένη αρχιτεκτονική, κάθε στοιχείο και τα συναφή συστατικά του (dependencies) πρέπει να είναι παρόντα για να εκτελεστεί ή να συνταχθεί κώδικας.
Στόχος του κάθε προγραμματιστή είναι να δημιουργήσει μία εφαρμογή που θα λειτουργεί και θα αναπτύσσεται για ένα μεγάλο χρονικό διάστημα. Παρόλα αυτά, όταν το μέγεθος μίας εφαρμογής είναι πολύ μεγάλο, η περαιτέρω ανάπτυξη της δυσχεραίνεται λόγω πολυπλοκότητας. Σαν αποτέλεσμα η επίλυση προβλημάτων αλλά και η ανάπτυξη νέων λειτουργιών είναι δύσκολη, επίπονη και χρονοβόρα.
Πιο αναλυτικά τα προβλήματα αυτά, αφορούν αρκετά κομβικούς τομείς της ανάπτυξης μίας εφαρμογής. Για αρχή, το compile, το start up και το deployment είναι ανάλογο του μεγέθους της εφαρμογής, δηλαδή όσο μεγαλύτερη είναι η εφαρμογή, τόσο μεγαλύτερος και ο χρόνος των δύο παραπάνω διαδικασιών. Επιπλέον, φυσική απόρροια μίας μεγάλης εφαρμογής είναι ότι δεν είναι εύκολα scalable. Για παράδειγμα ένα module που θέλει να έχει πρόσβαση σε μια βάση χρειάζεται μεγάλες ταχύτητες I/O. Ένα άλλο module που υλοποιεί πολύπλοκο business logic μπορεί να θέλει ιδιαίτερα καλό επεξεργαστή. Όμως εφόσον η εφαρμογή γίνεται deploy στο ίδιο υπολογιστικό σύστημα, οι δυνατότητες της είναι περιορισμένες.
Ένα ακόμα αρνητικό της μονολιθικής εφαρμογής είναι η αξιοπιστία. Καθώς όλα τα επιμέρους τμήματά της συνυπάρχουν σε ένα υπολογιστικό σύστημα και τρέχουν σαν μία εφαρμογή είναι αναπόφευκτο λάθη του ενός να επηρεάζουν το άλλο. Ένα καταστροφικό bug σε ένα module θα ρίξει ολόκληρη την εφαρμογή. Ένα memory leak θα απορροφήσει όλη τη μνήμη. Ένα deadlock θα σταματήσει όλους τους υπολογισμούς.

1.2 Microservices

Εν είδει επίλυσης των παραπάνω προβλημάτων, δημιουργήθηκε η αρχιτεκτονική των microservices. Η αρχιτεκτονική των microservices (MSA) έχει εξελιχθεί ως λύση σε πολλά από τα προβλήματα που σχετίζονται με τις μεγάλες, μονολιθικές εφαρμογές. Είναι το πρώτο

αρχιτεκτονικό στυλ της εποχής της DevOps και υιοθετεί πλήρως τις πρακτικές συνεχούς παράδοσης λογισμικού (continuous delivery). Είναι επίσης ένα παράδειγμα εξελικτικής αρχιτεκτονικής, που υποστηρίζει την αύξηση των αλλαγών ως πρώτη αρχή.
Η αύξηση χρήσης των microservices υπήρξε αξιοσημείωτη. Με τα microservices, μία εφαρμογή αναπτύσσεται ή αναδιαμορφώνεται σε ξεχωριστές υπηρεσίες που επικοινωνούν η μία με την άλλη με έναν καθορισμένο τρόπο μέσω των APIs. Κάθε microservice είναι αυτοδύναμο διατηρώντας τη δική του βάση δεδομένων, κάτι το οποίο έχει σημαντικά οφέλη και μπορεί να ενημερώνεται ανεξάρτητα από τις άλλες εφαρμογές.

1.3 Πλεονεκτήματα MSA

Ένα από τα πιο σημαντικά πλεονεκτήματα των microservices είναι ότι αρχιτεκτονικά προσφέρει τη λύση της αποκέντρωσης των υπηρεσιών (decentralization) και έτσι αποδομείται η αυξανόμενη πολυπλοκότητα ενός συστήματος. Αυτό σημαίνει ότι η εφαρμογή χωρίζεται με τέτοιο τρόπο, ώστε οι υπηρεσίες και οι ενότητες να είναι πιο σαφή αλλά και loosely coupled, επιτυγχάνοντας έτσι ευκολότερη και ταχύτερη ανάπτυξη.
Επιπλέον, αφού έχει επιτευχθεί το παραπάνω, απελευθερώνονται οι επιλογές για τη χρήση της τεχνολογίας που θα χρησιμοποιηθεί για την ανάπτυξη του κάθε service. Αυτό συνεπάγεται, ότι μπορεί να χρησιμοποιηθεί οποιαδήποτε γλώσσα προγραμματισμού σε συνδυασμό με οποιαδήποτε τεχνολογία, αφού ο μόνος περιορισμός για την επικοινωνία των δύο υπηρεσιών είναι να υποστηρίζουν το API.
Ένα ακόμα πλεονέκτημα αυτής της αρχιτεκτονικής σε σύγκριση με τη μονολιθική είναι οι μειωμένοι χρόνοι του deployment. Εφόσον, η κάθε υπηρεσία είναι πλέον ανεξάρτητη της άλλης, μπορεί να βγει παραγωγή (live production) αμέσως αφού περάσει testing, χωρίς να εξαρτάται από τα υπόλοιπα.
Όπως αναφέρθηκε, ένα ιδιαίτερα σημαντικό πρόβλημα μιας ώριμης μονολιθικής εφαρμογής είναι το scaling. Με τα microservices μπορούμε να κάνουμε scale όποιο microservice χρειάζεται μόνο και στο βαθμό που χρειάζεται επίσης. Είναι σύνηθες όταν ο orchestrator (πχ kubernetes) βλέπει κάποιο microservice ότι φτάνει στα «όριά του» να κάνει spin up περισσότερα instances. Επίσης μπορούν διαφορετικά microservices να τρέχουν σε διαφορετικό hardware αναλόγως με τις ανάγκες και τις ιδιαιτερότητες του καθένα.

1.4 Μειονεκτήματα MSA

Το decentralization του συστήματος μπορεί να επιφέρει αρκετά πλεονεκτήματα, παρόλα αυτά καταλήγει να προσθέτει ένα overhead στον προγραμματιστή, εφόσον έχει να διαχειριστή τις διαφορετικές επικοινωνίες μεταξύ των microservices, προσθέτοντας στην ανάπτυξη κώδικα περισσότερο error handling αλλά και άυξηση των api calls, ως σύνολο.
Όπως μπορεί κάποιος να αντιληφθεί το decentralization δεν αφορά μόνο τις υπηρεσίες αλλά και τις βάσεις των υπηρεσιών αυτών. Στη συγκεκριμένη αρχιτεκτονική, κάθε υπηρεσία χρησιμοποιεί τη δική βάση, σε σύγκριση με τη μονολιθική αρχιτεκτονική στην οποία χρησιμοποιείται μία καθολική βάση. Το παραπάνω σημαίνει ότι τα transactions αυξάνονται σε πολλές βάσεις και αυτό πρέπει να διαχειριστεί από τον προγραμματιστή.
Στην περίπτωση του testing προστίθεται μία πολυπλοκότητα ακόμα στο orchestration. Εφόσον δεν τρέχει όλη η εφαρμογή ενιαία, πρέπει ο tester να σιγουρευτεί ότι τρέχουν όλα τα microservices και επικοινωνούν μεταξύ τους ή να γίνουν stubs.
Επιπλέον, όταν γίνονται αλλαγές σε ένα microservice (προστίθενται λειτουργίες ή αλλάζει το API) και αυτό επηρεάζει κάποιο άλλο microservice με το οποίο επικοινωνεί, είναι απαραίτητο να γίνουν οι απαραίτητες αλλαγές σε όλες τις υπηρεσίες που επηρεάζονται αλλά και να σχεδιαστεί πως θα γίνει το roll out, ώστε να ικανοποιούνται τα dependencies. Σε αυτό το σημείο να διευκρινιστεί ότι όλα αυτά επιλύονται με ροές fallback, όπου όμως και εκεί προστίθεται overhead στον προγραμματιστή.
Το deployment της εφαρμογής ταυτόχρονα με το να γίνει πιο ευέλικτο και δυνατό έγινε και περισσότερο πολύπλοκο. Σε μία κλασική μονολιθική εφαρμογή έχουμε απλά έναν server. Σε περίπτωση που έχουμε και scaling έχουμε απλά πανομοιότυπους servers. Στην περίπτωση των

microservices όμως έχουμε πολλές διαφορετικές εφαρμογές σε διαφορετικά VMs/dockers σε διαφορετικά HWs. Όλο αυτό το σύστημα είναι σίγουρα πιο ευέλικτο αλλά πολύ πιο δύσκολο να οργανωθεί.

2 Υλοποίηση Εφαρμογής

Στα πλαίσια της παρούσης μεταπτυχιακής εργασίας αναπτύχθηκε ένα σύστημα από microservices, το οποίο συλλέγει δεδομένα από αισθητήρες κινητών μέσω μίας android εφαρμογής, τα αποθηκεύει σε μία βάση δεδομένων και γίνεται επεξεργασία των δεδομένων αυτών με σκοπό την εξαγωγή στατιστικών δεδομένων.

2.1 Big Data

Ενώ αρχικά χρησιμοποιούταν σαν συνώνυμο πολύ μεγάλων όγκων δεδομένων, τα οποία δεν ήταν εύκολα διαχειρίσιμα από σχεσιακές βάσεις δεδομένων και τεχνολογίες παλαιότερης εποχής, στη σημερινή εποχή τα Big Data έρχονται να καλύψουν μία ποικιλία προηγμένων χαρακτηριστικών, τεχνολογιών, μεθόδων και μοτίβων. Με την πάροδο του χρόνου τα Big Data έχουν περάσει από σημαντικές αλλαγές για να καταλήξουν να είναι μία από τις πιο ανοδικές τεχνολογίες.

Παρά την μακροχρόνια ωρίμανση και την εξαιρετικά ενεργή ερευνητική κοινότητα, δε βρέθηκε ένας διακριτός και καθολικά εφαρμοσμένος ορισμός που περιγράφει με ακρίβεια αυτόν τον όρο. Παρ 'όλα αυτά, σύμφωνα με έναν από τους πιο ευρέως διαδεδομένους ορισμούς, τα Big Data «αποτελούνται από εκτεταμένα σύνολα δεδομένων -κυρίως στα χαρακτηριστικά του όγκου, ποικιλία, ταχύτητα και/ή μεταβλητότητα -που απαιτούν ευμετάβλητη αρχιτεκτονική για αποτελεσματική αποθήκευση, εκκαθάριση και ανάλυση». Ένας άλλος ορισμός που μπορεί να χρησιμοποιηθεί είναι ότι τα Big Data είναι ένα πεδίο που ασχολείται συστηματικά, εξάγει πληροφορίες ή βρίσκει τρόπους ανάλυσης των όγκων δεδομένων που είναι για παράδειγμα πολύ περίπλοκα, πολύ γρήγορα κινούμενα, πολύ μεγάλα ή πολύ αδύναμα δομημένα για ανάλυση χρησιμοποιώντας χειροκίνητες και συμβατικές μεθόδους επεξεργασίας δεδομένων.

2.2 Big Data & Microservices

Τα πλεονεκτήματα του να χτίζεις εφαρμογές Big Data με microservices είναι αξιοσημείωτα, αν και κάποιος μπορεί να πει ότι δε συνδέονται άμεσα.
Εκτός από τα ζητήματα οπτικοποίησης δεδομένων και τις λειτουργικές προκλήσεις του σχεδιασμού και της εφαρμογής λύσεων Big Data, τα microservices μπορούν να βοηθήσουν στην αντιμετώπιση - σε κάποιο βαθμό - ορισμένων από τα εμπόδια που αντιμετωπίζουν συχνά οι εταιρείες. Ακολουθούν μερικά από τα πιο σημαντικά ζητήματα που σχετίζονται με πρωτοβουλίες Big Data και πώς μπορεί να βοηθήσει η αρχιτεκτονική των microservices.
Ένα από τα μεγαλύτερα οφέλη από τη χρήση της αρχιτεκτονικής microservices για εφαρμογές Big Data είναι το scalability που παρέχει. Παρόλο που δεν είναι απαραίτητα συνώνυμο του cloud computing, είναι σχετικά συνηθισμένο να χρησιμοποιούνται και τα δύο μαζί. Μια παραδοσιακή μονολιθική εφαρμογή δεν έχει την ευελιξία εφαρμογών που βασίζονται σε microservices. Με κάθε υπηρεσία να λειτουργεί ανεξάρτητα, οι διακομιστές στους οποίους βρίσκονται μπορούν να αυξάνονται και να μειώνονται σε πόρους ανάλογα με τις ανάγκες. Αυτό είναι ιδιαίτερα σημαντικό για τα συστήματα Big Data, τα οποία τείνουν να καταναλώνουν πολλούς πόρους καθώς χειρίζονται δεδομένα σε υψηλό όγκο και ταχύτητα. (Aptude, 2016)
Καθώς τα δεδομένα ρέουν μέσω της διαδικασίας απορρόφησης, υπάρχουν πολλά σημεία αποτυχίας που μπορεί να προκύψουν και η ποιότητα των δεδομένων είναι μία από τις μεγαλύτερες προκλήσεις που σχετίζονται με τις εφαρμογές των Big Data. Ενώ η ποιότητα των δεδομένων (data quality) είναι ένα πολύπλευρο ζήτημα, που περιλαμβάνει πολλές λειτουργικές πτυχές, τα microservices μπορούν επίσης να βοηθήσουν σε αυτό το θέμα. Οι ομάδες ανάπτυξης λογισμικού που δημιουργούν τα microservices έχουν ένα πολύ πιο εστιασμένο έργο. Δεδομένου ότι κάθε υπηρεσία έχει έναν συγκεκριμένο σκοπό για τον οποίο έχει αναπτυχθεί, διευκολύνει τη δημιουργία, τη συντήρηση και τον έλεγχο δεδομένων καθώς τα δεδομένα επικοινωνούνται μεταξύ των υπηρεσιών.

3 Τεχνολογίες

3.1 Java

Η Java είναι μια γλώσσα προγραμματισμού που προέρχεται από τα εργαστήρια της Sun Microsystems και πατέρας της είναι ο James Gosling. Παρουσιάστηκε το 1995 σαν το κύριο κομμάτι της Java Platform της Sun Microsystems. Οι αρχικοί java compilers, virtual machines και βιβλιοθήκες αναπτύχθηκαν από τη Sun, η οποία όμως το 2007 τους κυκλοφόρησε υπό την άδεια GNU GPL. Σαν γλώσσα κληρονομεί πολλά στοιχεία από το συντακτικό της C και της C++ αλλά σε πολλά σημεία μένει σε υψηλότερο επίπεδο αρχιτεκτονικά από αυτές. Οι εφαρμογές που είναι γραμμένες σε java τυπικά γίνονται compile σε bytecode που στη συνέχεια μπορεί να τρέξει (θεωρητικά) σε κάθε java virtual machine (JVM) ασχέτως αρχιτεκτονικής. Είναι δηλαδή μια γλώσσα γενικού σκοπού, concurrent (δηλαδή τα προγραμμάτά της μπορούν να σχεδιαστούν να και να υλοποιηθούν σαν τμήματα που μπορούν να εκτελεστούν παράλληλα), αντικειμενοστραφής και class-based (δηλαδή η κληρονομικότητα βασίζεται σε κλάσεις αντικειμένων και όχι στα ίδια τα αντικείμενα). Έχει σχεδιαστεί ειδικά για να έχει όσο το δυνατόν λιγότερες εξαρτήσεις όσον αφορά την υλοποίηση. Σκοπός της είναι «να γράφεται μία φορά και να τρέχει παντού», δηλαδή ο ίδιος κώδικας να τρέχει σε διαφορετικές πλατφόρμες .
Επιγραμματικά αναφέρουμε κάποια βασικά πλεονεκτήματά της:

- platform independent (write once, run everywhere)
- simple (ο προγραμματιστής δεν χρειάζεται να ασχοληθεί με pointers)
- object oriented (inheritance, encapsulation, polymorphism, dynamic binding)
- robust (διαχείριση μνήμης, garbage collection, exception handling, type checking)
- secure (όλα τα προγράμματα τρέχουν μέσα σε μιας μορφής "sandbox")
- multithreaded

3.2 Spring boot

Το Spring framework χρησιμοποιεί νέες τεχνικές όπως Aspect Oriented programming (AOP), Plain Old Java Object (POJO), και dependency injection για την ανάπτυξη enterprice εφαρμογών, αφαιρώντας έτσι τις δυσκολίες και τις πολυπλοκότητες που έχουν άλλες αντίστοιχες τεχνολογίες (όπως τα enterprice java beans EJB). Το Spring είναι ένα ελαφρύ σχετικά framework ανοιχτού κώδικα που δίνει στους προγραμματιστές τη δυνατότητα να αναπτύξουν απλές, αξιόπιστες και scalable enterprise εφαρμογές. Κυρίως επικεντρώνεται στο να προσφέρει διάφορους τρόπους διαχείρισης των διαφορετικών business object. Με αυτές τις τεχνικές έκανε ευκολότερη την ανάπτυξη διαδικτυακών εφαρμογών από τα παραδοσιακά frameworks όπως JDBC, JSP, Java Servlets. Το Spring framework μπορεί να θεωρηθεί μία ομπρέλα κάτω από την οποία υπάρχουν διάφορα άλλα μικρότερα frameworks που μπορούν να λειτουργήσουν συνδυαστικά. Αυτά τα υπο-frameworks ονομάζονται layers και ενδεικτικά είναι το Spring Web MVC, Spring Data, Spring AOP, Spring ORM, Spring Web Flow κ.ά (Spring Boot, 2021)
Κάποια από τα βασικότερα χαρακτηριστικά του είναι τα εξής:

- **Inversion of Control Container**: Είναι το βασικό container στο οποίο με τη χρήση inversion of control / dependency injection για να μπορέσει να παρέχει ένα reference ενός object σε μια κλάση σε runtime χρόνο
- **Data access framework**: Δίνει τη δυνατότητα χρήσης APIs όπως το JDBC / Hibernate για την αποθήκευση δεδομένων λύνοντας προβλήματα όπως database interaction, exception handling, transaction handling κ.ά.
- **Spring MVC framework**: Επιτρέπει την ανάπτυξη διαδικτυακών εφαρμογών με την αρχιτεκτονική MVC

- **Transaction Management**: Βοηθά στην διαχείριση transaction τόσο global (μέσω ενός application server) αλλά και τοπικών (μέσω JDBC Hibernate κτλ)
- **JDBC Abstraction Layer**: βοηθά στην αντιμετώπιση λαθών με εύκολο τρόπο στην αλληλεπίδραση με JDBC
- **Spring Test Context**: βοηθά στο unit testing και integration testing των spring εφαρμογών.

Δυστυχώς όμως χρειαζόταν (αρχικά τουλάχιστον) πολύ και πολύπλοκο configuration μέσω xml αρχείων και trial and error. Η λύση σε αυτό ήρθε από την δημιουργό εταιρία Pivotal με το Spring Boot. Το Spring Boot είναι ένα framework βασισμένο σε microservices που δίνει τη δυνατότητα να αναπτυχθούν production-ready εφαρμογές σε πολύ λίγο χρόνο. Βασίζεται στην έννοια του convention over configuration έχοντας προ-παραμετροποιημένες πολλές λειτουργίες και επιλογές σύμφωνα με τη γνώμη των δημιουργών του. Επίσης έρχεται με configuration για πολλά 3rd party libraries που συχνά καλείται να κάνει integrate ο προγραμματιστής

3.3 Βάση Δεδομένων MongoDB

Η πλατφόρμα MongoDB είναι ένα πρόγραμμα ανοικτού κώδικα, βάσης δεδομένων μη σχεσιακού χαρακτήρα και για το λόγο αυτό κατατάσσεται στην κατηγορία 'NoSQL'. Η συγκεκριμένη πλατφόρμα εκτός από τα πλεονεκτήματα που χαρακτηρίζουν τις 'NoSQL' βάσεις δεδομένων έχει και όλη τη δύναμη μιας σχεσιακής βάσης δεδομένων. Η MongoDB αποθηκεύει δεδομένα σε έγγραφα τύπου JSON ('open Standard file format' - ISO/IEC 21778:2017), τα οποία παρέχουν πολύ περισσότερες δυνατότητες από τα παραδοσιακά μοντέλα σχεσιακών. Παρέχει μια ισχυρή γλώσσα ανάπτυξης ερωτημάτων που επιτρέπει το φιλτράρισμα και ταξινόμηση ανά πεδίο, ανεξάρτητα από το πόσο σύνθετο μπορεί να είναι ένα 'JSON' έγγραφο. Υποστηρίζει σύγχρονες υπηρεσίες επεξεργασίας δεδομένων , όπως συσσωμάτωση 'aggregation' , αναζήτηση σε γεωγραφικού τύπου δεδομένων , γράφους και κείμενο (Mongo DB, 2021).
Παρακάτω παρουσιάζονται τα βασικά χαρακτηριστικά των δυνατοτήτων τους.

- **Αυτοσχέδια ερωτήματα (Ad-hoc queries) –** Το MongoDB υποστηρίζει αναζητήσεις πεδίων 'field', εύρους τομέα 'range queries' αλλά και έκφρασης 'regular-expression'. Επιπρόσθετα στις αναζητήσεις και συγκεκριμένα στα αποτελέσματα αυτών, μπορούν να εφαρμοστούν και λειτουργίες JavaScript. Επίσης υπάρχει η δυνατότητα ρύθμισης των αποτελεσμάτων, ώστε να επιστρέφουν ένα τυχαίο δείγμα αποτελεσμάτων ενός δεδομένου μεγέθους.
- **Ευρετήρια (Indexing) –** Υποστηρίζει δημιουργία ευρετηρίου 'indexing' και με πρωτεύων αλλά και δευτερεύων δεικτών.
- **Αντίγραφα (Replication) –** Το MongoDB παρέχει υψηλή διαθεσιμότητα με σετ αντιγράφων. Ένα σύνολο αντιγράφων αποτελείται από δύο ή περισσότερα αντίγραφα των δεδομένων. Κάθε αντιγραφόμενο μέλος έχει τη δυνατότητα να ενεργεί σε ρόλο είτε πρωτεύοντος είτε δευτερεύοντος αντιγράφου ανά πάσα στιγμή. Όλες οι εγγραφές και οι αναγνώσεις γίνονται από το πρωτότυπο αντίγραφο από προεπιλογή. Τα δευτερεύοντα αντίγραφα διατηρούν ένα αντίγραφο των δεδομένων του πρωτεύοντος χρησιμοποιώντας μια ενσωματωμένη εφαρμογή 'build in'. Όταν αποτυγχάνει ένα πρωτότυπο αντίγραφο, το σετ αντιγράφων πραγματοποιεί αυτόματα μια διαδικασία εκλογής για να προσδιορίσει το δευτερεύων αντίγραφό και να χαρακτηριστεί ως κύριο – πρωτεύων. Τα δευτερεύων αντίγραφα μπορούν προαιρετικά να χρησιμοποιηθούν στα πλαίσια λειτουργιών ανάγνωσης για την προβολή τους. Κατά συνέπεια, μια εξειδικευμένη διανομή MongoDB απαιτεί τουλάχιστον τρεις ξεχωριστούς διακομιστές, ακόμη και για τις περιπτώσεις απλού σχεδιασμού με πρωτεύων και δευτερεύων αντιγράφων .
- **Εξισορρόπηση φορτίου (Load balancing) –** Υποστηρίζει υπηρεσία οριζόντιας κλιμάκωσης 'Cluster' χρησιμοποιώντας 'shard' κλειδιά. Αυτά τα κλειδιά είναι με τέτοιο τρόπο σχεδιασμένα , ώστε να παρέχουν με άμεση σχέση στην αποτελεσματικότητα της

κατανομής των δεδομένων. Με αυτό τον τρόπο τα δεδομένα χωρίζονται σε περιοχές, βάσει 'shard' κλειδιού και κατανέμονται στα αντίστοιχα τμήματα. Η MongoDB έχοντας την δυνατότητα να τρέξει σε πολλούς διακομιστές, μπορεί να εξισορροπεί το φορτίο , αντιγράφοντας τα δεδομένα με τέτοιο τρόπο ώστε να εξασφαλίζει και περιπτώσεις βλάβης υλικού.

- **Σύστημα αρχείων αποθήκευσης (File storage) –** Η MongoDB έχει τη δυνατότητα να χρησιμοποιηθεί ως σύστημα αρχείων, 'GridFS', με δυνατότητες εξισορρόπησης φορτίου και αναπαραγωγής δεδομένων σε πολλά μηχανήματα για την αποθήκευση αρχείων. Αυτή η λειτουργία διαιρεί ένα αρχείο σε τμήματα ή κομμάτια και αποθηκεύει κάθε ένα από αυτά τα κομμάτια ως ξεχωριστό έγγραφο. Διαθέτει επιπρόσθετες λειτουργίες για τη διαχείριση των αρχείων αλλά και το περιεχόμενο τους όπως τα βοηθητικά προγράμματα mongofiles , Nginx και lighttpd
- **Συσσωμάτωση Aggregation –** Η MongoDB παρέχει τρεις τρόπους υπολογισμού των δεδομένων , μέσα από τεχνικές 'pipeline aggregations' , από 'MapReduce' εφαρμογές αλλά και απλές 'single purpose' μεθόδους. Ο κάθε τρόπος εξυπηρετεί και ένα σκοπό, ωστόσο, η τεχνική των 'pipeline aggregations' παρέχει την καλύτερη απόδοση για τις περισσότερες λειτουργίες.
- **Server-side JavaScript execution –** Υποστηρίζει JavaScript λειτουργίες αναζήτησης αλλά και επεξεργασίας.
- **Κλειστές συλλογές (Capped collections) –** υποστηρίζει την διαχείριση συλλογών σταθερού μεγέθους που ονομάζονται 'capped collections'. Αυτός ο τύπος διαχείρισης διατηρεί τη σειρά εισαγωγής και, μόλις επιτευχθεί το προκαθορισμένο μέγεθος, προχωράει με κυκλική ουρά.
- **Συναλλαγές (Transactions) -** Δεν υποστηρίζει συναλλαγές ACID πολλαπλών εγγράφων καθώς ο συγκεκριμένος ισχυρισμός από την κυκλοφορία της έκδοσης 4.0 βρέθηκε να μην είναι αληθής, παραβιάζοντας την λειτουργία 'snapshot isolation'.

3.4 Git

Το Git είναι ένα DVCS (distributed version control system) με πρωταρχικό στόχο την ταχύτητα. Ο αρχικός σχεδιασμός και η ανάπτυξη έγινε από τον Linus Torvalds για να χρησιμοποιηθεί για revision control στο Linux Kernel, αλλά από τότε χρησιμοποιείται ευρέως σε πληθώρα open source project, αλλά και σε εταιρικό επίπεδο. Διανέμεται υπό την άδεια ανοιχτού λογισμικού GNU General Public License V2.

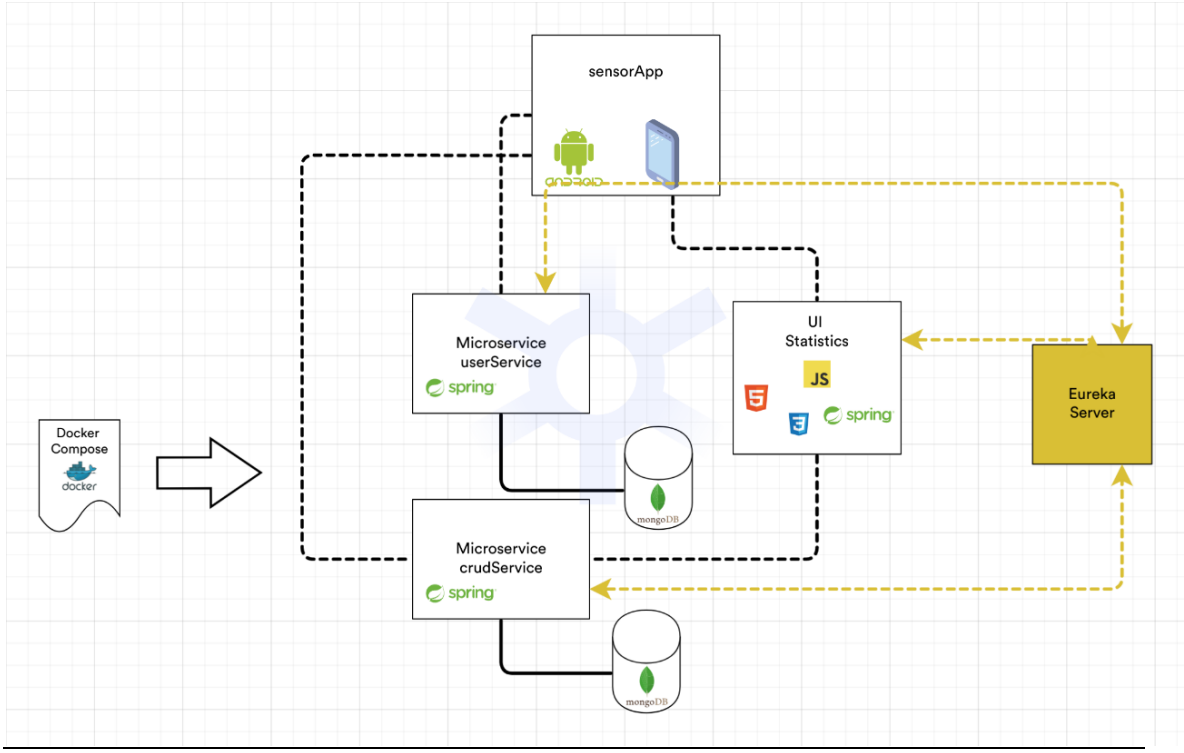
Το Git, υποστηρίζει πολύ γρήγορη δημιουργία branch αλλά και merge περιλαμβάνοντας τα κατάλληλα εργαλεία ώστε να μπορεί ο developer να εργαστεί αλλά και να περιηγηθεί στον κώδικα μη-γραμμικά. Βασικό αξίωμα κατά τον σχεδιασμό του git ήταν ότι μία αλλαγή γίνεται merge περισσότερες φορές από όσες γράφεται και συνεπώς η δημιουργία branch και είναι πολύ γρήγορη αλλά και το ίδιο το branch πολύ ελαφρύ καθώς αποτελεί απλά μία αναφορά σε ένα commit. Όπως προείπαμε, το git είναι DVCS, distributed, δηλαδή διανεμημένο. Αυτό σημαίνει ότι κάθε developer κατέχει ένα πλήρες τοπικό αντίγραφο του repository με ολόκληρη την ιστορία όλων των αλλαγών σε όλα τα branches. Οι αλλαγές που επιτελεί στον κώδικα αποθηκεύονται πρώτα στο τοπικό repository (local repository) και μετά εάν το επιθυμεί προωθούνται στο απομακρυσμένο. Επειδή δημιουργήθηκε για να χρησιμοποιηθεί από το linux kernel, είναι ιδιαίτερα αποτελεσματικό στα μεγάλα projects. Η ύπαρξη του local repository το κάνει να είναι σε άλλη τάξη μεγέθους από άλλα vcs. Δεν γίνεται δηλαδή αργότερο όσο το code base και το project μεγαλώνουν. Το data model που χρησιμοποιεί το git εξασφαλίζει την κρυπτογραφική ακεραιότητα (cryptographic integrity) κάθε bit του project. Για κάθε αρχείο και commit που γίνεται, υπολογίζεται το checksum του, και με αυτό ακριβώς το checksum ανακτώνται από το repository. Γι αυτό το λόγο είναι αδύνατο ο χρήστης να πάρει κάτι διαφορετικό από αυτό ακριβώς που έβαλε. Και γι αυτό είναι αδύνατο να γίνει οποιαδήποτε αλλαγή χωρίς να αλλάξουν τα IDs των πάντων από το σημείο αυτό και μετά. Οπότε εάν ο χρήστης κατέχει το ID ενός συγκεκριμένου commit, εξασφαλίζει όχι μόνο ότι το project είναι ακριβώς το ίδιο με όταν έγινε το commit, αλλά και ότι τίποτα στο παρελθόν δεν είχε αλλάξει.

3.5 Android Studio

Το Android Studio είναι το επίσημο περιβάλλον ανάπτυξης για εφαρμογές Android. Πρωτοεμφανίστηκε 16 Μαΐου 2003 σε συνέδριο της Google και είναι βασισμένο στο λογισμικό JetBrains-Intelli J IDEA ενώ είναι κατασκευασμένο σε Java. Επίσης εμπεριέχει το Android SDK το οποίο περιλαμβάνει τις βιβλιοθήκες που χρειάζονται για την ανάπτυξη των εφαρμογών. Το πρόγραμμα αυτό είναι διαθέσιμο δωρεάν για Windows, Mac OS και Linux, και αντικατέστησε το Eclipse Android Development Tools (ADT), το οποίο μέχρι το 2014 αποτελούσε το κύριο IDE για κατασκευή λογισμικού Android. Η κυκλοφορία αυτού του νέου προγραμματιστικού περιβάλλοντος ανακοινώθηκε από την Google στις 16 Μαΐου 2013, στο Συνέδριο Google I/O, ενώ η κατασκευή του βρισκόταν ακόμα σε πρώιμο στάδιο. Η πρώτη σταθερή έκδοση του Android Studio ήταν διαθέσιμη για το κοινό το Δεκέμβριο του 2014 (έκδοση 1.0).
Το Android Studio προσφέρει διάφορες υπηρεσίες στου χρήστες, κάποιες από αυτές είναι οι εξής:

- Ένα ευέλικτο Cradle-based σύστημα κατασκευής.
- Ένα γρήγορο και με πολλές λειτουργίες εξομοιωτή (emulator).
- Δυνατότητα ανάπτυξης εφαρμογών για όλες τις συσκευές Android (Smartphones, Smart watches, Smart TV's).
- Περιέχει εργαλεία ελέγχου της απόδοσης , της χρηστικότητας και και έλεγχο συμβατότητας των εφαρμογών. •
- Υποστήριξη C++
- Υποστήριξή Google Cloud Platform.

4 Αρχιτεκτονική Εφαρμογής



Διάγραμμα 4-1: Απεικόνιση της αρχιτεκτονικής

Το παραπάνω σχήμα δείχνει την αρχιτεκτονική της εφαρμογής. Έχει δημιουργηθεί ένα αρχείο που ονομάζεται docker-compose, το οποίο όταν τρέχει κάνει spin up όλη τη διασυνδεδεμένη εφαρμογή. Τα microservices που την απαρτίζουν επιγραμματικά είναι τα εξής:

- **Eureka Service**: Υλοποίηση του cloud eureka server που βοηθά στο Service Registration και το Service Discovery.
- **CRUD Service**. Spring boot microservice, που διαχειρίζεται τα στοιχεία της βάσης.
- **User Service**: Spring boot microservice που υλοποιεί τη σύνδεση και την εγγραφή χρηστών.
- **UIStatistics**: Spring boot microservice που απεικονίζει τα στατιστικά στοιχεία που υπολογίζονται στο StatisticsService
- **MongoDB**: Το container με την εκάστοτε βάση στην οποία γράφει το UserService και το CrudService.

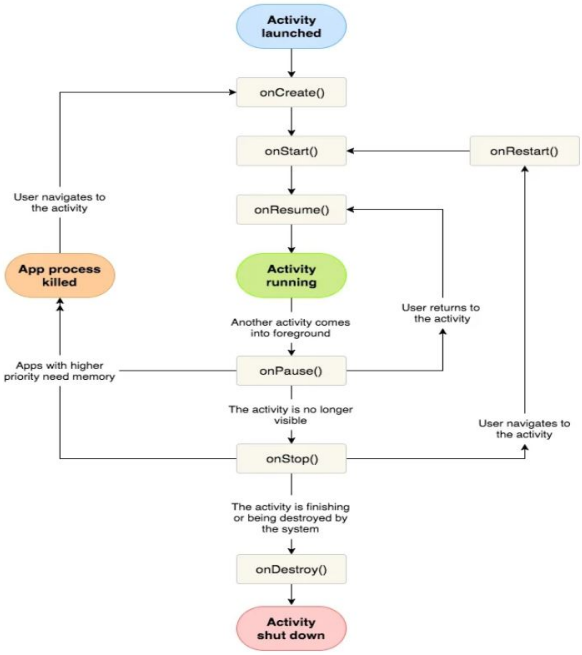
Η επιλογή των microservices αυτών έγινε ώστε να έχουμε μία μικρογραφία ενός distributed microservices ecosystem όπου κάθε εφαρμογή μπορεί να υπάρξει σε περισσότερα από ένα container χωρίς να δημιουργεί πρόβλημα αλλά και, το κυριότερο, να μην είναι ορατό από κάποιον τρίτο. Δηλαδή όταν κάποιος αλληλοεπιδρά με το σύστημα δεν έχει γνώση ούτε αντιλαμβάνεται αν είναι μία μονολιθική εφαρμογή ή αν αποτελείται από περισσότερα microservices και πόσα είναι αυτά. Η επιτυχία της σχεδίασης μιας εφαρμογής με microservices είναι όταν κάποιος τρίτος παρατηρητής δεν μπορεί να αντιληφθεί την αρχιτεκτονική αυτή, αλλά το μόνο που αντιλαμβάνεται είναι το availability και η σταθερότητά της. Στο κεφάλαιο Παρουσίαση Microservice6 αναλύονται τα παραπάνω με μεγαλύτερη λεπτομέρεια.

5 Εφαρμογή Android

Τα σημαντικότερα δομικά στοιχεία του πλαισίου εφαρμογών είναι:

- Σύστημα προβολών (View System): αποτελεί ένα εκτενές σύνολο από αντικείμενα GUI τα οποία μπορούν να χρησιμοποιηθούν κατά το σχεδιασμό μιας εφαρμογής. Παραδείγματα προβολών είναι οι λίστες (listView), το πλέγμα (GridView), πεδία εισαγωγής κειμένου, κουμπιά, κλπ
- Πάροχος Περιεχομένου (Content Provider): δίνει τη δυνατότητα στις εφαρμογές να μοιράζονται ή να ανταλλάσσουν δεδομένα μιας συγκεκριμένης μορφής η οποία ορίζεται από τον πάροχο. Παραδείγματα δεδομένων, είναι οι επαφές χρήστη και οι βάσεις δεδομένων των εφαρμογών.
- Διαχειριστής Πόρων (Resource Manager): παρέχει πρόσβαση σε υλικό το οποίο δεν είναι σε μορφή κώδικα όπως πχ, εικόνες, αρχεία xml, πίνακες χαρακτήρων, κλπ
- Διαχειριστής Ειδοποιήσεων (Notification Manager): δίνει στις εφαρμογές πρόσβαση στις υπηρεσίες ειδοποιήσεων χρήστη. Τέτοιες είναι οι ειδοποιήσεις στη notification bar, τα toast μηνύματα στο κάτω μέρος της οθόνης, η δόνηση του κινητού και η ενεργοποίηση της ◦ οθόνης, κλπ
- Διαχειριστής Τοποθεσίας (Location Manager): δίνει στις εφαρμογές τη δυνατότητα εντοπισμού της τοποθεσίας μη τη χρήση του GPS η των κεραιών των κυψελωτών δικτύων.
- Διαχειριστής Δραστηριοτήτων (Activity Manager): διαχειρίζεται τον κύκλο ζωής των δραστηριοτήτων και παρέχει δυνατότητα πλοήγησης από δραστηριότητα σε δραστηριότητα κρατώντας αποθηκευμένη στη μνήμη τη σειρά εκτέλεσης αυτών.

Στο παρακάτω σχεδιάγραμμα φαίνεται λεπτομερώς ο κύκλος ζωής κάθε δραστηριότητας:



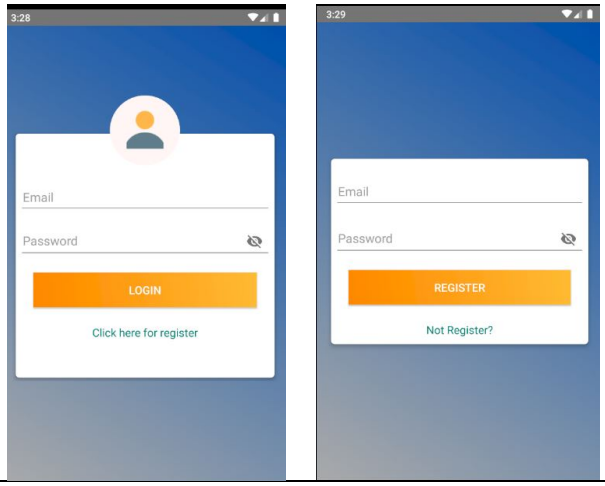
Εικόνα 5-1: Κύκλος ζωής activity (Ζωγράφου, 2021)

5.1 Login & Register Service

Προϋπόθεση για την χρήση της εφαρμογής είναι η εγγραφή του χρήστη στο σύστημα. Την πρώτη φορά που θα ανοίξει ο χρήστης την εφαρμογή θα αντικρίσει μια οθόνη για την ταυτοποίηση του . Οι χρήστες που δεν έχουν εγγραφεί ακόμα μπορούν να επιλέξουν την κατηγορία χρήστη που προτρέπει στην οθόνη εγγραφής. Σε περίπτωση που ο χρήστης έχει ήδη λογαριασμό, για να

συνδεθεί στην εφαρμογή, πρέπει να εισάγει το email και το password και στη συνέχεια να πατήσει το κουμπί Login. Ο αλγόριθμος κατά την υποβολή της εγγραφής προχωράει ως εξής:
Γίνεται ο έλεγχος των πεδίων για τυχόν μη αποδεκτές τιμές (validation). Αν τα πεδία είναι αποδεκτά τότε ο αλγόριθμος αποστέλλει το αίτημα στον εξυπηρετητή. Στην περίπτωση σφάλματος εμφανίζεται ανάλογο μήνυμα στον χρήστη.

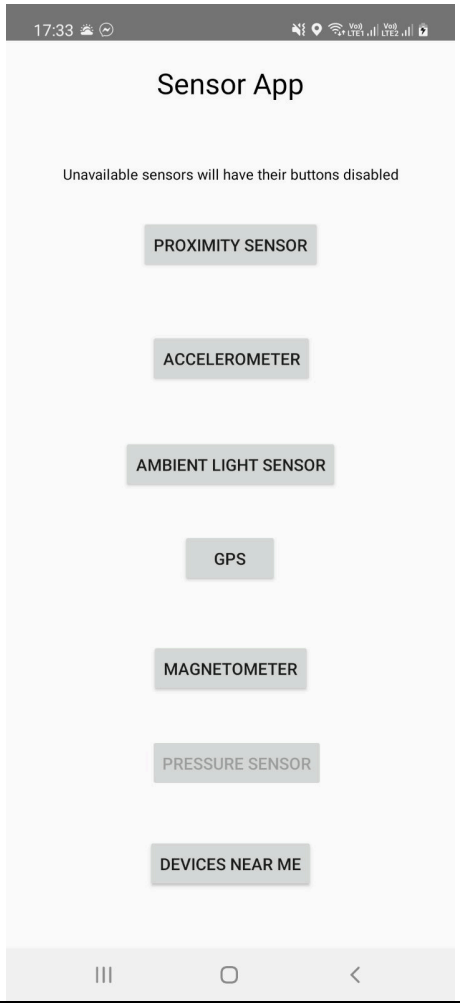
Σε αυτή τη λειτουργία η εφαρμογή επικοινωνεί με το User Service και με τον τρόπο αυτό είτε δημιουργείται ο χρήστης είτε επιτρέπεται η περιήγηση στην εφαρμογή. Η ταυτοποίηση του χρήστη είναι απαραίτητη για τη δημιουργία δεδομένων με το crud service και εν συνεχεία για την απεικόνισή τους.



Εικόνα 5-2: Login & Register οθόνες της εφαρμογής

5.2 Activities

Μετά τη σύνδεση του χρήστη, εκείνος περιηγείται στην επόμενη οθόνη της εφαρμογής στην οποία καταγράφονται οι επιλογές των αισθητήρων του κινητού. Η εφαρμογή ενημερώνει τον χρήστη ότι οι μη διαθέσιμοι αισθητήρες θα έχουν το αντίστοιχο κουμπί της λίστας απενεργοποιημένο.



Εικόνα 5-3: Βασικό UI εφαρμογής

5.2.1 Proximity Sensor

Παρακάτω παρατίθεται ο κώδικας που αναπτύχθηκε στα πλαίσια της εργασίας αυτής για λήψη των δεδομένων από τον proximity sensor. Όταν ο χρήστης πατήσει το κουμπί από το UI για τον αισθητήρα αυτόν τότε ξεκινάει το lifecycle του Activity, με τις αντίστοιχες συναρτήσεις να φαίνονται στις παρακάτω εικόνες που αποτελούν απόσπασμα του κώδικα της android εφαρμογής.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_proximity);

    //retrieve widget
    proximitySensorText = findViewById(R.id.proximityText);

    //define instances
    sensorManager = (SensorManager)getSystemService(Context.SENSOR_SERVICE);
    proximitySensor = sensorManager.getDefaultSensor(Sensor.TYPE_PROXIMITY);
}

//register the listener once the activity starts
@Override
protected void onStart() {
    super.onStart();

    if(proximitySensor != null) {

        sensorManager.registerListener( listener: this, proximitySensor, sensorManager.SENSOR_DELAY_NORMAL);
    }
}

//stop the sensor when the activity stops to reduce battery usage
@Override
protected void onStop() {
    super.onStop();

    sensorManager.unregisterListener(this);
}
```

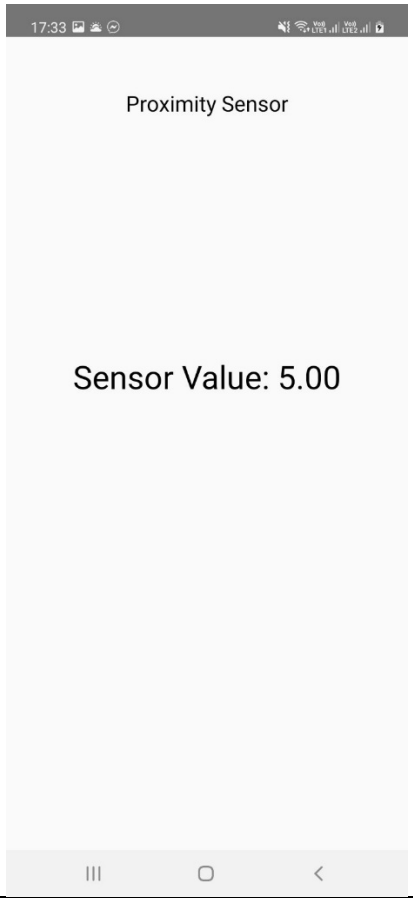
Εικόνα 5-4: Κώδικας συναρτήσεων onStart, onCreate, onStop για proximity sensor

Στην Εικόνα 5-5 ο κώδικας δείχνει πως λαμβάνονται οι τιμές και παρουσιάζονται στην οθόνη του χρήστη καθώς και η κλήση που εκτελείται στο crud service για τη δημιουργία των δεδομένων στη βάση.

Στην Εικόνα 5-6 παρουσιάζεται στιγμιότυπο του UI την ώρα που έχει καταγραφεί μία από τις μετρήσεις του proximity sensor.

```
@Override
public void onSensorChanged(SensorEvent sensorEvent) {

    session = new SessionManager(getApplicationContext());
    final HashMap<String, String> user = session.getUserDetails();
    final String username = user.get(SessionManager.KEY_username);


    //retrieve the current value of the proximity sensor
    float currentValue = sensorEvent.values[0];

    //display the retrieved value onto the textView
    proximitySensorText.setText("Sensor Value: {currentValue}");

    try {
        JSONObject jsonBody = new JSONObject();
        jsonBody.put( name: "proximity", currentValue);
        jsonBody.put( name: "dataType",  value: "PROXIMITY_SENSOR");
        jsonBody.put( name: "username", username);
        final String requestBody = jsonBody.toString();

        CreateDataApi.postData(getApplicationContext(), requestBody);
    } catch (JSONException e) {
        e.printStackTrace();
    }
}
```

Εικόνα 5-5: Κώδικας συνάρτησης onSensorChanged για proximity sensor



Εικόνα 5-6: Στιγμιότυπο από μέτρηση του Proximity Sensor

5.2.2 Accelerometer Sensor

Παρακάτω παρατίθεται ο κώδικας που αναπτύχθηκε στα πλαίσια της εργασίας αυτής για λήψη των δεδομένων από τον accelerometer sensor. Όταν ο χρήστης πατήσει το κουμπί από το UI για τον αισθητήρα αυτόν τότε ξεκινάει το lifecycle του Activity, με τις αντίστοιχες συναρτήσεις να φαίνονται στις παρακάτω εικόνες που αποτελούν απόσπασμα του κώδικα της android εφαρμογής.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_accelerometer);

    //retrieve widgets
    xValue = findViewById(R.id.xValue);
    yValue = findViewById(R.id.yValue);
    zValue = findViewById(R.id.zValue);

    //define instances
    sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
    accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
}

//register the listener once the activity starts
@Override
protected void onStart() {
    super.onStart();

    if (accelerometer != null) {

        sensorManager.registerListener( listener: this, accelerometer, sensorManager.SENSOR_DELAY_NORMAL);
    }
}

//stop the sensor when the activity stops to reduce battery usage
@Override
protected void onStop() {
    super.onStop();

    sensorManager.unregisterListener(this);
}
```

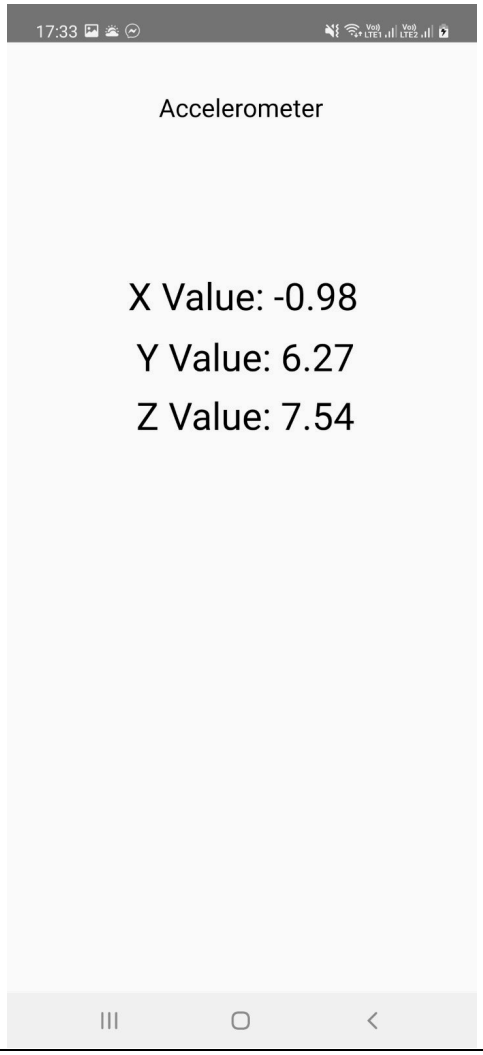
Εικόνα 5-7: Κώδικας συναρτήσεων onStart, onCreate, onStop για accelerometer sensor

Στην Εικόνα 5-8 ο κώδικας δείχνει πως λαμβάνονται οι τιμές και παρουσιάζονται στην οθόνη του χρήστη καθώς και η κλήση που εκτελείται στο crud service για τη δημιουργία των δεδομένων στη βάση.

Στην Εικόνα 5-9 παρουσιάζεται στιγμιότυπο του UI την ώρα που έχει καταγραφεί μία από τις μετρήσεις του accelerometer sensor.

```
public void onSensorChanged(SensorEvent sensorEvent) {

    session = new SessionManager(getApplicationContext());
    final HashMap<String, String> user = session.getUserDetails();
    final String username = user.get(SessionManager.KEY_username);

    //get the current values of the accelerometer for each axis
    float current_xValue = sensorEvent.values[0];
    float current_yValue = sensorEvent.values[1];
    float current_zValue = sensorEvent.values[2];

    //display the current values of the  accelerometer for each axis onto the
    //textView widgets
    xValue.setText("X Value: {current_xValue}");
    yValue.setText("Y Value: {current_yValue}");
    zValue.setText("Z Value: {current_zValue}");

    try {
        JSONObject jsonBody = new JSONObject();
        jsonBody.put( name: "latitude", current_xValue);
        jsonBody.put( name: "longitude", current_yValue);
        jsonBody.put( name: "height", current_zValue);
        jsonBody.put( name: "dataType",  value: "ACCELEROMETER_SENSOR");
        jsonBody.put( name: "username", username);
        final String requestBody = jsonBody.toString();

        CreateDataApi.postData(getApplicationContext(), requestBody);
    } catch (JSONException e) {
        e.printStackTrace();
    }
}
```

Εικόνα 5-8: Κώδικας συνάρτησης onSensorChanged για accelerometer sensor



Εικόνα 5-9: Στιγμιότυπο από μέτρηση του Accelerometer Sensor

5.2.3 Ambient Light Sensor

Παρακάτω παρατίθεται ο κώδικας που αναπτύχθηκε στα πλαίσια της εργασίας αυτής για λήψη των δεδομένων από τον accelerometer sensor. Όταν ο χρήστης πατήσει το κουμπί από το UI για τον αισθητήρα αυτόν τότε ξεκινάει το lifecycle του Activity, με τις αντίστοιχες συναρτήσεις να φαίνονται στις παρακάτω εικόνες που αποτελούν απόσπασμα του κώδικα της android εφαρμογής.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_light);

    //retrieve widgets
    lightSensorText = findViewById(R.id.lightSensorText);

    //define instances
    sensorManager = (SensorManager)getSystemService(Context.SENSOR_SERVICE);
    lightSensor = sensorManager.getDefaultSensor(Sensor.TYPE_LIGHT);
}

//register the listener once the activity starts
@Override
protected void onStart() {
    super.onStart();

    if(lightSensor != null) {

        sensorManager.registerListener( listener: this, lightSensor, sensorManager.SENSOR_DELAY_NORMAL);
    }
}

//stop the sensor when the activity stops to reduce battery usage
@Override
protected void onStop() {
    super.onStop();

    sensorManager.unregisterListener(this);
}
```

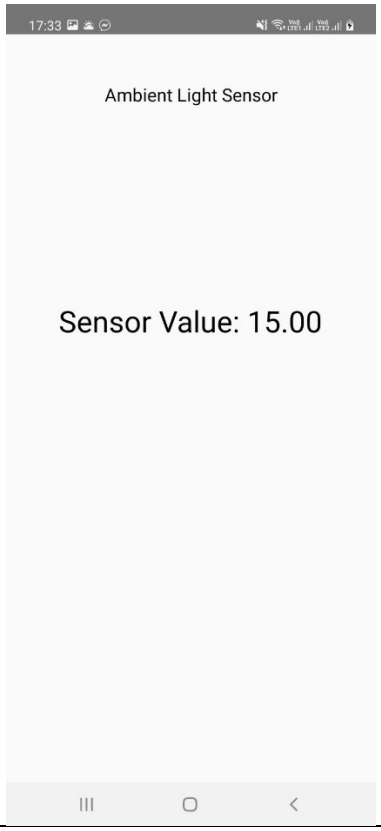
Εικόνα 5-10: Κώδικας συναρτήσεων onStart, onCreate, onStop για light sensor

Στην Εικόνα 5-11 ο κώδικας δείχνει πως λαμβάνονται οι τιμές και παρουσιάζονται στην οθόνη του χρήστη καθώς και η κλήση που εκτελείται στο crud service για τη δημιουργία των δεδομένων στη βάση.

Στην Εικόνα 5-9 παρουσιάζεται στιγμιότυπο του UI την ώρα που έχει καταγραφεί μία από τις μετρήσεις του accelerometer sensor.

```
@Override
public void onSensorChanged(SensorEvent sensorEvent) {

    session = new SessionManager(getApplicationContext());
    final HashMap<String, String> user = session.getUserDetails();
    final String username = user.get(SessionManager.KEY_username);

    //retrieve the current value of the light sensor
    float currentValue = sensorEvent.values[0];

    try {
        JSONObject jsonBody = new JSONObject();
        jsonBody.put( name: "light", currentValue);
        jsonBody.put( name: "dataType",  value: "LIGHT_SENSOR");
        jsonBody.put( name: "username", username);
        final String requestBody = jsonBody.toString();

        CreateDataApi.postData(getApplicationContext(), requestBody);
    } catch (JSONException e) {
        e.printStackTrace();
    }

    //display the retrieved values onto the textView
    lightSensorText.setText("Sensor Value: {currentValue}");
}
```

Εικόνα 5-11: Κώδικας συνάρτησης onSensorChanged για light sensor



Εικόνα 5-12: Στιγμιότυπο λήψης τιμής από τον Light Sensor

5.2.4 GPS

Στις παρακάτω εικόνες (Εικόνα 5-13) παρουσιάζεται ο κώδικας που αναπτύχθηκε για τον GPS Sensor. Βλέπουμε ότι ο κάτωθι κώδικας παρουσιάζει στον χρήστη και δημιουργεί καταχώρηση στη βάση με τα πεδία latitude, longitude, altitude, bearing, speed, location.
Στην Εικόνα 5-14 παρουσιάζεται στιγμιότυπο λήψης των τιμών από το αισθητήρα GPS και παρουσιάζονται οι προαναφερθείσες τιμές.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_gps);

    //retrieve widgets
    GPSBtn = findViewById(R.id.GPSBtn);
    latText = findViewById(R.id.latText);
    longText = findViewById(R.id.longText);
    altitudeText = findViewById(R.id.altitudeText);
    bearingText = findViewById(R.id.bearingText);
    speedText = findViewById(R.id.speedText);
    locationText = findViewById(R.id.address);

    session = new SessionManager(getApplicationContext());
    final HashMap<String, String> user = session.getUserDetails();
    final String username = user.get(SessionManager.KEY_username);

    //request permission for GPS access
    ActivityCompat.requestPermissions( activity: GPSActivity.this,
            new String[]{Manifest.permission.ACCESS_FINE_LOCATION},
            LOCATION_REQUEST);
```

```
//set listener for GPS button widget
GPSBtn.setOnClickListener((view) -> {

        //get instance of GPSTracker class in order to ask the user for
        //location access
        GPSTracker GPSTracker = new GPSTracker(getApplicationContext());
        Location location = GPSTracker.getLocation();

        if (location != null) {

            //get current values for latitude and longitude
            double latValue = location.getLatitude();
            double longValue = location.getLongitude();
            double altitude = location.getAltitude();
            float bearing = location.getBearing();
            float speed = location.getSpeed();

            //display the values retrieved onto the textView widgets
            latText.setText(getResources().getString(R.string.GPS_lat_value, latValue));
            longText.setText("Longitude: {longValue}");
            altitudeText.setText("Altitude: {altitude}");
            bearingText.setText("Bearing: {bearing}");
            speedText.setText("Speed: {speed}");


            JSONObject jsonBody = new JSONObject();

            Geocoder geocoder = new Geocoder(getApplicationContext(), Locale.getDefault());
    Geocoder geocoder = new Geocoder(getApplicationContext(), Locale.getDefault());
    try {
        List<Address> addresses = geocoder.getFromLocation(latValue, longValue,  maxResults: 1);
        String fullAddress = addresses.get(0).getAddressLine( index: 0);
        locationText.setText("Location:\n" +" " + fullAddress);
        String admin = addresses.get(0).getAdminArea();
        String subadmin = addresses.get(0).getSubAdminArea();
        String city = addresses.get(0).getLocality();
        String area =addresses.get(0).getThoroughfare();
        String countryName = addresses.get(0).getCountryName();
        String countryCode = addresses.get(0).getCountryCode();
        String postalCode = addresses.get(0).getPostalCode();

        jsonBody.put( name: "fullAddress", fullAddress);
        jsonBody.put( name: "admin", admin);
        jsonBody.put( name: "subadmin", subadmin);
        jsonBody.put( name: "city", city);
        jsonBody.put( name: "countryName", countryName);
        jsonBody.put( name: "countryCode", countryCode);
        jsonBody.put( name: "postalCode", postalCode);
        jsonBody.put( name: "area", area);
        jsonBody.put( name: "latitude", latValue);
        jsonBody.put( name: "longitude", longValue);
        jsonBody.put( name: "altitude", altitude);
        jsonBody.put( name: "bearing", bearing);
        jsonBody.put( name: "speed", speed);
        jsonBody.put( name: "dataType",  value: "GPS_SENSOR");
        jsonBody.put( name: "username", username);
        final String requestBody = jsonBody.toString();
        Log.i( tag: "requestBody: ", requestBody);
        CreateDataApi.postData(getApplicationContext(), requestBody);
    } catch (IOException | JSONException e) {
        e.printStackTrace();
    }
```

Εικόνα 5-13: Κώδικας συνάρτησης onCreate για τον GPS sensor

Εικόνα 5-14: Στιγμιότυπο από μέτρηση του GPS Sensor

5.2.5 Magnetometer Sensor

Παρακάτω παρατίθεται ο κώδικας που αναπτύχθηκε στα πλαίσια της εργασίας αυτής για λήψη των δεδομένων από τον magnetometer sensor. Όταν ο χρήστης πατήσει το κουμπί από το UI για τον αισθητήρα αυτόν τότε ξεκινάει το lifecycle του Activity, με τις αντίστοιχες συναρτήσεις να φαίνονται στις παρακάτω εικόνες που αποτελούν απόσπασμα του κώδικα της android εφαρμογής.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_magnetometer);

    //retrieve widgets
    xValue = findViewById(R.id.xValue);
    yValue = findViewById(R.id.yValue);
    zValue = findViewById(R.id.zValue);

    //define instances
    sensorManager = (SensorManager)getSystemService(Context.SENSOR_SERVICE);
    magnetometer = sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);
}

//register the listener once the activity starts
@Override
protected void onStart() {
    super.onStart();

    if(magnetometer != null) {

        sensorManager.registerListener( listener: this, magnetometer, sensorManager.SENSOR_DELAY_NORMAL);
    }
}

//stop the sensor when the activity stops to reduce battery usage
@Override
protected void onStop() {
    super.onStop();

    sensorManager.unregisterListener(this);
}
```

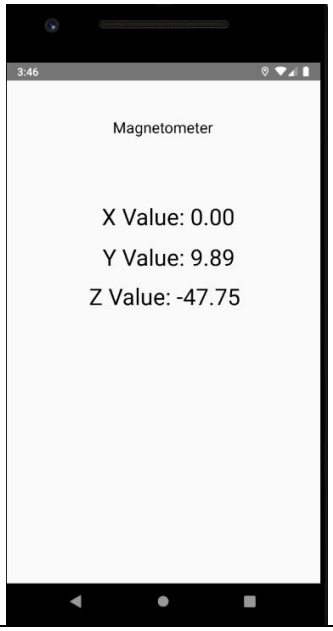
Εικόνα 5-15: Κώδικας συναρτήσεων onStart, onCreate, onStop για magnetometer sensor

Στην Εικόνα 5-16 ο κώδικας δείχνει πως λαμβάνονται οι τιμές και παρουσιάζονται στην οθόνη του χρήστη καθώς και η κλήση που εκτελείται στο crud service για τη δημιουργία των δεδομένων στη βάση.

Στην Εικόνα 5-17 παρουσιάζεται στιγμιότυπο του UI την ώρα που έχει καταγραφεί μία από τις μετρήσεις του magnetometer sensor.

```
@Override
public void onSensorChanged(SensorEvent sensorEvent) {

    session = new SessionManager(getApplicationContext());
    final HashMap<String, String> user = session.getUserDetails();
    final String username = user.get(SessionManager.KEY_username);

    //retrieve the current values of the magnetometer for each axis
    float current_xValue = sensorEvent.values[0];
    float current_yValue = sensorEvent.values[1];
    float current_zValue = sensorEvent.values[2];

    //display each value onto its corresponding textView
    xValue.setText("X Value: {current_xValue}");
    yValue.setText("Y Value: {current_yValue}");
    zValue.setText("Z Value: {current_zValue}");

    try {
        JSONObject jsonBody = new JSONObject();
        jsonBody.put( name: "latitude", current_xValue);
        jsonBody.put( name: "longitude", current_yValue);
        jsonBody.put( name: "height", current_zValue);
        jsonBody.put( name: "dataType", value: "MAGNETOMETER_SENSOR");
        jsonBody.put( name: "username", username);
        final String requestBody = jsonBody.toString();

        CreateDataApi.postData(getApplicationContext(), requestBody);
    } catch (JSONException e) {
        e.printStackTrace();
    }
}
```

Εικόνα 5-16: Κώδικας συναρτήσεων onSensorChanged για magnetometer sensor



Εικόνα 5-17: Στιγμιότυπο από μέτρηση του Magnetometer Sensor

5.2.6 Pressure Sensor

Παρακάτω παρατίθεται ο κώδικας που αναπτύχθηκε στα πλαίσια της εργασίας αυτής για λήψη των δεδομένων από τον pressure sensor. Όταν ο χρήστης πατήσει το κουμπί από το UI για τον αισθητήρα αυτόν τότε ξεκινάει το lifecycle του Activity, με τις αντίστοιχες συναρτήσεις να φαίνονται στις παρακάτω εικόνες που αποτελούν απόσπασμα του κώδικα της android εφαρμογής.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_pressure);

    //retrieve widget
    pressureSensorText = findViewById(R.id.pressureSensorText);

    //define instances
    sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
    pressureSensor = sensorManager.getDefaultSensor(Sensor.TYPE_PRESSURE);
}

//register the listener once the activity starts
@Override
protected void onStart() {
    super.onStart();

    if (pressureSensor != null) {

        sensorManager.registerListener( listener: this, pressureSensor, sensorManager.SENSOR_DELAY_NORMAL);
    }
}

//stop the sensor when the activity stops to reduce battery usage
@Override
protected void onStop() {
    super.onStop();

    sensorManager.unregisterListener(this);
}
```

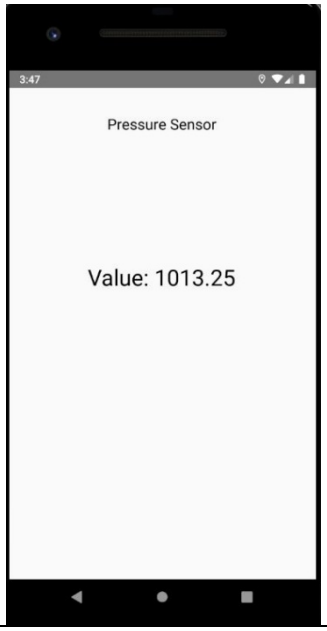
Εικόνα 5-18: Κώδικας συναρτήσεων onStart, onCreate, onStop για pressure sensor

Στην Εικόνα 5-19 ο κώδικας δείχνει πως λαμβάνονται οι τιμές και παρουσιάζονται στην οθόνη του χρήστη καθώς και η κλήση που εκτελείται στο crud service για τη δημιουργία των δεδομένων στη βάση.

Στην Εικόνα 5-20 παρουσιάζεται στιγμιότυπο του UI την ώρα που έχει καταγραφεί μία από τις μετρήσεις του pressure sensor.

```
@Override
public void onSensorChanged(SensorEvent sensorEvent) {

    session = new SessionManager(getApplicationContext());
    final HashMap<String, String> user = session.getUserDetails();
    final String username = user.get(SessionManager.KEY_username);

    //retrieve the current value of the pressure sensor
    float currentValue = sensorEvent.values[0];

    //display the retrieved value onto the textView
    pressureSensorText.setText("Value: {currentValue}");

    try {
        JSONObject jsonBody = new JSONObject();
        jsonBody.put( name: "pressure", currentValue);
        jsonBody.put( name: "dataType", value: "PRESSURE_SENSOR");
        jsonBody.put( name: "username", username);
        final String requestBody = jsonBody.toString();

        CreateDataApi.postData(getApplicationContext(), requestBody);
    } catch (JSONException e) {
        e.printStackTrace();
    }
}
```

Εικόνα 5-19: Κώδικας συναρτήσεων onSensorChanged για pressure sensor



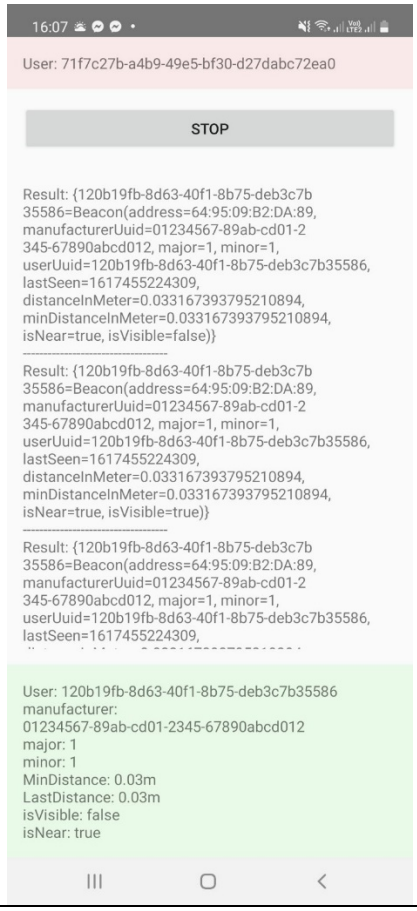
Εικόνα 5-20: Στιγμιότυπο από μέτρηση του Pressure Sensor

5.2.7 Devices Near Me

Για τη λειτουργία αυτή έχει ενσωματωθεί στην εφαρμογή η βιβλιοθήκη Close To Me η οποία δημιουργήθηκε από τον Mohsen Mirhoseini, την περίοδο της μεγάλης έξαρσης του κορονοϊού. Σκοπός αυτής της βιβλιοθήκης είναι να εντοπίσει όλες τις Bluetooth συσκευές χρησιμοποιώντας ένα BLE API για να εκθέσει ένα beacon και συγχρόνως να σκανάρει για άλλα beacon στην περιβάλλουσα περιοχή (Mirhoseini, 2021).

Προϋπόθεση για να δουλέψει η συγκεκριμένη βιβλιοθήκη είναι να έχουν και άλλοι χρήστες την ίδια εφαρμογή στο κινητό τους που θα χρησιμοποιεί τεχνολογία beacon έτσι ώστε να μπορεί να σκαναριστεί και εν συνεχεία να εντοπιστεί.

Όπως φαίνεται και στην παρακάτω εικόνα η βιβλιοθήκη δεν παραθέτει προσωπικά στοιχεία των χρηστών, αλλά τους δίνει ένα μοναδικό αναγνωριστικό (uuid), εμφανίζει την απόσταση που έχει η συσκευή τους από τη συσκευή του χρήστη και βάση ακτίνας αποφασίζει αν θεωρείται κοντινή η απόσταση ή όχι. Πατώντας το STOP η εφαρμογή σταματάει να εκθέτει το beacon του χρήστη και να σκανάρει την περιβάλλουσα περιοχή.



Εικόνα 5-21: Στιγμιότυπο σκαναρίσματος συσκευών από τη βιβλιοθήκη closeToMe

6 Παρουσίαση Microservice

6.1 Eureka Service

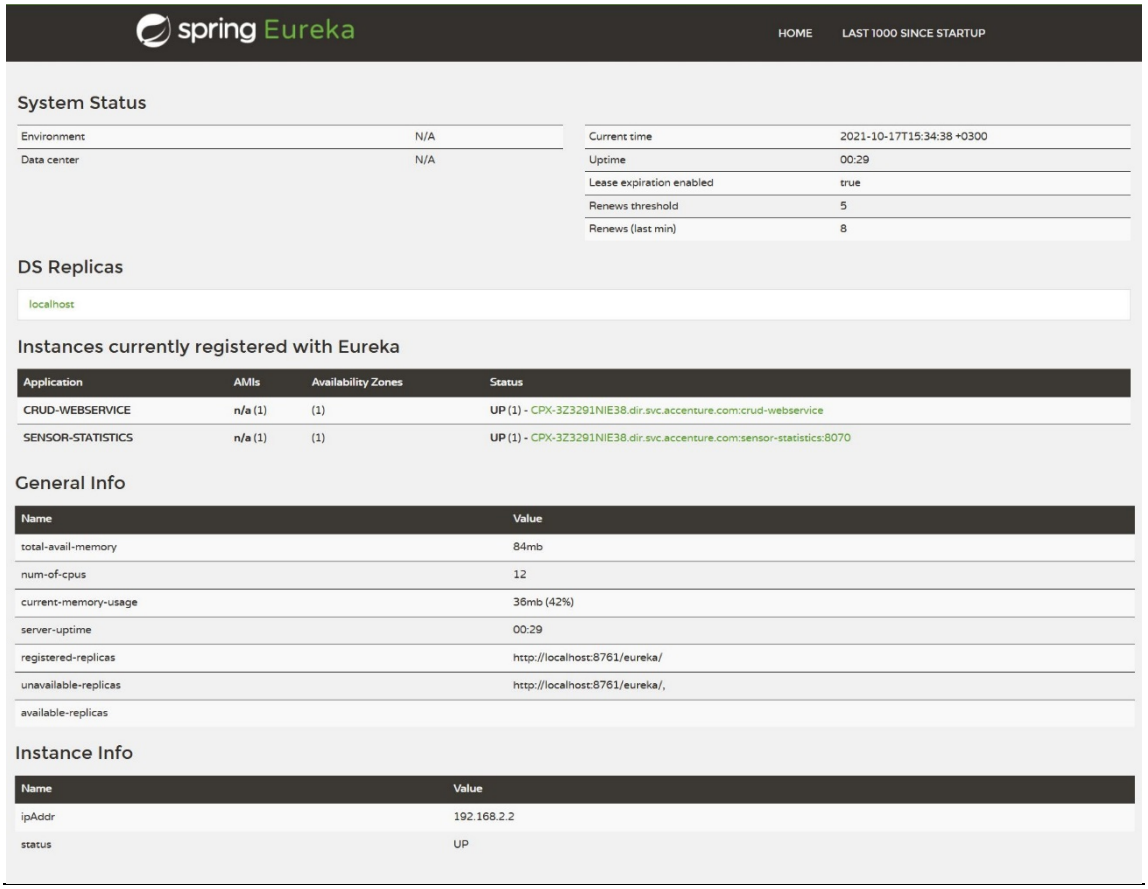
Όπως ήδη έχει περιγραφεί παραπάνω κάθε υλοποίηση με microservices υπάρχει ένα οικοσύστημα εφαρμογών, οι οποίες επικοινωνούν μεταξύ τους. Το Eureka έρχεται να λύση το πρόβλημα του service discovery αλλά και των κάτωθι προβλημάτων, τα οποία δεν υπάρχουν στην ανάπτυξη μίας μονολιθικής εφαρμογής:

- High availability: Σε περιπτώσεις πολλών instances πρέπει επικοινωνίες προς ένα προβληματικό instance να μεταδίδονται σε άλλα υγιή
- load balancing: σε περιπτώσεις υψηλού φόρτου αυτό πρέπει να μοιράζεται μεταξύ των instances
- resiliency: πρέπει να υποστηρίζει local caching
- fault tolerance: όταν κάποιο instance δεν είναι available πρέπει να αφαιρείται από τη λίστα των available

Μία τέτοια υλοποίηση είναι το Netflix Eureka. Μία open source εφαρμογή ανεπτυγμένη από το Netflix για να λύσουν προβλήματα service discovery. Συγκεκριμένα λύνει τα εξής θέματα:

- service registration: επιτρέπει σε services να "εγγραφούν" σε ένα κατάλογο με διαθέσιμα services σε ένα κεντρικό server
- client lookup of servive address: ο client μπορεί να αναζητήσει τη διεύθυνση για κάποιο συγκεκριμένο service από αυτό τον κατάλογο
- information sharing: διαμοιράζεται την πληροφορία για τα διάφορα services μεταξύ των nodes
- health monitoring: επιβλέπει τα διάφορα services και αν κάποιο από αυτά δεν είναι "υγειές" το αφαιρεί από τον κατάλογο.

Για να δουλέψει το Netflix Eureka χρειάζεται ένας server και ένας αριθμός από clients. Στην παρούσα εφαρμογή ο server είναι το microservice "eurekaserver" (Kothagal, 2019).
Στον client ενεργοποιείται το service registration με το όνομα spring.application.name. Επίσης επιλέγεται το local caching του καταλόγου των services θέτοντας το eureka.client.fetchRegistry:true και τέλος ορίζεται σε πιο συγκεκριμένο zone θα γίνει register με το eureka.client.serviceUrl.defaultZone Το eureka server όταν τρέχει σηκώνει και ένα web ui στο οποίο μπορεί ο χρήστης να δει τα διάφορα services που έχουν γίνει register:



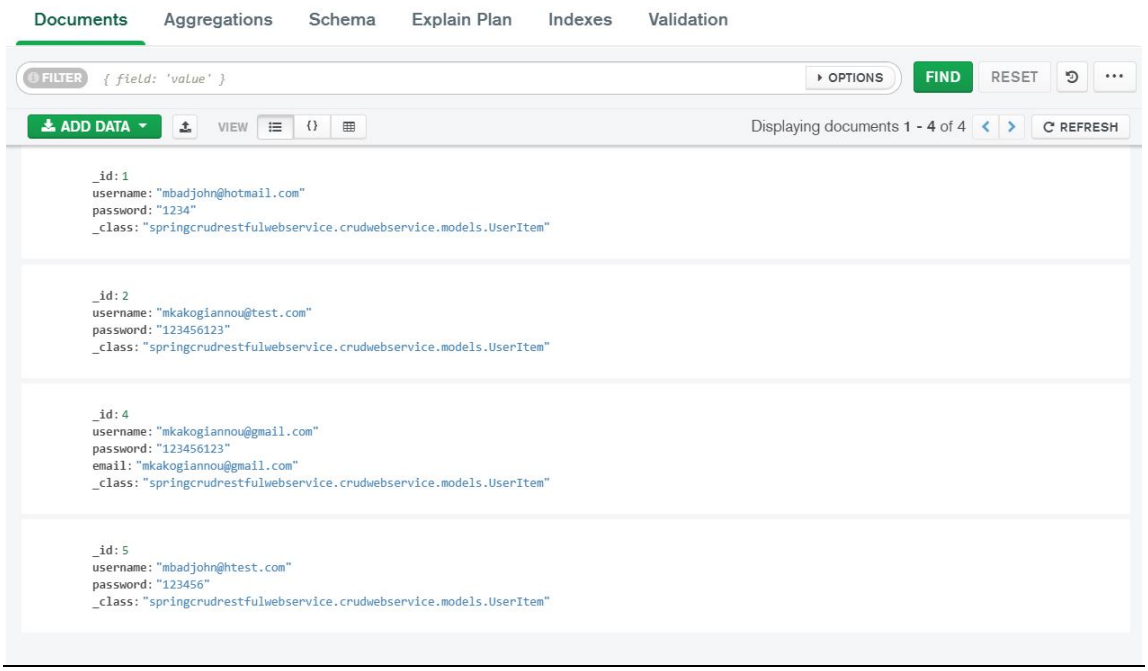
Εικόνα 6-1: UI Eureka Service με running instances

6.2 User Service

Το user service είναι εκείνο το service με το οποίο επικοινωνεί η android εφαρμογή για να εγγραφούν οι χρήστες είτε να συνδεθούν. Όπως απεικονίζεται παρακάτω συνδέεται με τη δική του βάση Mongo DB και έχει δύο endpoints.

6.2.1 Mongo DB

Η βάση που επιλέχθηκε ήταν η Mongo DB. Για αυτό το service δημιουργήθηκε ένα collection στη βάση και καταγράφονται 3 πεδία. Το id, username και password του χρήστη που απαιτώνται για την ταυτοποίηση του χρήστη.



Εικόνα 6-2: Στιγμιότυπο από τη βάση Users

6.2.2 Endpoints

Παρακάτω φαίνονται τα endpoints που αναπτύχθηκαν στα πλαίσιο του συγκεκριμένου service.

Path	Http Command	Λειτουργία
/users	GET	Ανακτά όλους τους χρήστες
/users/register	POST	Εγγραφή των χρηστών στην εφαρμογή
/users/login	POST	Σύνδεση ήδη εγγεγραμμένου χρήστη
/users/{username}/changepwd	PUT	Αλλαγή κωδικού πρόσβασης

Αυτές οι λειτουργίες δεν εκτελούνται απευθείας από τον controller για να υπάρχει ο απαραίτητος διαχωρισμός, έτσι ώστε ο controller να παίζει αποκλειστικά το ρόλο της «πόρτας» προς τα υπόλοιπα services. Αυτές οι λειτουργίες εκτελούνται από το userService δημιουργώντας με το @Autowired, dependency injection. Αυτό απεικονίζεται στο στιγμιότυπο κώδικα παρακάτω:

```java
@RestController
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping(path = "/users")
    public List<UserItem> getAllUsers() { return userService.getAllUsers(); }

    @PostMapping(path = "/users/register")
    public void register(@RequestBody UserItem user) {

        userService.registerUser(user);
    }

    @PostMapping(path = "/users/login")
    public UserItem login(@RequestBody UserItem user) {
        return userService.loginUser(user);
    }

    @PutMapping(path = "/users/{username}/changepwd")
    public UserItem changePassword(@PathVariable(value = "username") String username,@RequestBody UserItem user) {
        return userService.changeUserPassword(username, user);
    }

}
```

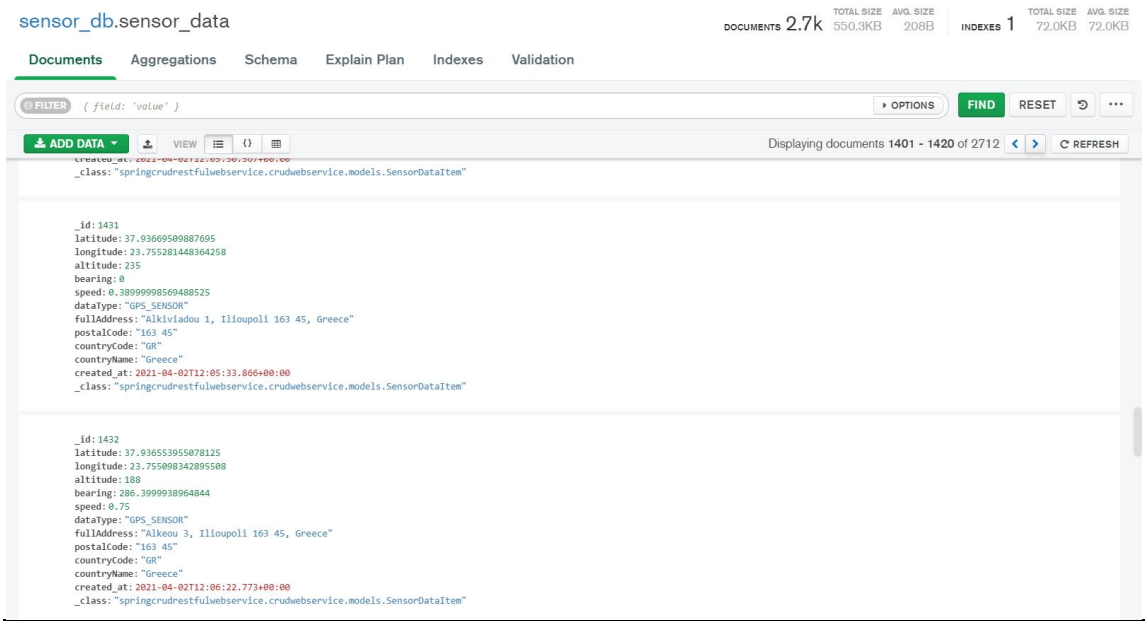
Εικόνα 6-3: Στιγμιότυπο κώδικα από το UserService

6.3 Crud Service

Το Crud Service είναι εκείνο το service το οποίο πάλι καλείται από την εφαρμογή android αλλά και από το Statistics UI όπως αυτό θα περιγραφεί παρακάτω. Δημιουργεί, διαβάζει και διαχειρίζεται όλα τα δεδομένα των sensors. Για το service αυτό επιλέχθηκε πάλι mongo DB βάση δεδομένων και τα endpoints θα περιγραφούν στο αντίστοιχο υποκεφάλαιο.

6.3.1 Mongo DB

Η βάση δεδομένων που επιλέχθηκε για το Crud Service είναι η mongo DB. Δημιουργήθηκε ένα collection που κάθε document αναπαριστά κάποια καταγραφή δεδομένων ενός sensor. Τα πεδία διαφοροποιούνται ανάλογα με τον sensor. Σε κάθε document αναγράφεται και το username του χρήστη, έτσι ώστε να μπορούν να εξαχθούν και στοιχεία ανά λογαριασμό αλλά και για περαιτέρω ανάλυση στατιστικών δεδομένων.



Εικόνα 6-4: Στιγμιότυπο από τη βάση που αντιστοιχεί στο CRUD Service

6.3.2 Endpoints

Παρακάτω φαίνονται όλα τα endpoints που δημιουργήθηκαν στο πλαίσιο αυτού του service.

Path	Http Command	Λειτουργία
/data	GET	Φέρνει όλα τα δεδομένα
/data/{id}	GET	Φέρνει τα δεδομένα με το ID
/data/type/{dataType}	GET	Φέρνει όλα τα δεδομένα ανάλογα με τον sensor
/data/type/{dataType}/{username}	GET	Φέρνει όλα τα δεδομένα ανάλογα με τον sensor και τον χρήστη
/data/user/{username}	GET	Φέρνει όλα τα δεδομένα ενός χρήστη
/data/avg	GET	Φέρνει τους μέσους όρους των δεδομένων ανά sensor
/data	POST	Εγγραφή δεδομένων στη βάση από την εφαρμογή android
/data/{id}	PUT	Αλλαγή δεδομένου στη βάση
/data/{id}	DELETE	Διαγραφή δεδομένου από με βάση το Id της εγγραφής
/data/date/{dataType}/{username}/	POST	Ανάκτηση δεδομένων με βάση τον sensor, user και ημερομηνία (από, μέχρι)

Συγκεκριμένα για τη λειτουργία η οποία ανακτά τους μέσους όρους των δεδομένων ανά αισθητήρα, απαιτήθηκε η διαδικασία ανάπτυξης aggregation query στη Mongo DB, όπως αυτό απεικονίζεται παρακάτω:

```
public interface SensorDataSearchRepository extends MongoRepository<SensorDataItem, String> {
    @Query(value="{ 'dataType' : ?0 }")
    List<SensorDataItem> findByDataType(String dataType);

    @Query(value="{ 'username' : ?0 }")
    List<SensorDataItem> findByUsername(String username);

    @Query(value = "{ 'username' : ?0, 'dataType' : ?1 }")
    List<SensorDataItem> findSensorDataItemByFilters(String username, String dataType);

    @Query(value = "{ 'created_at' : { $gt: ?0, $lt: ?1 }, 'dataType': ?2, 'username': ?3}")
    List<SensorDataItem> findByDateBetween(LocalDateTime created_atGT, LocalDateTime created_atLT, String dataType, String username);

    @Aggregation("{ '$group': { '_id' : '$dataType', 'light' : { $avg: '$light' }, " +
            "'avgX' : { $avg: '$latitude' }, 'avgY' : { $avg: '$longitude' }, " +
            "'avgZ' : { $avg: '$altitude' }, 'speed' : { $avg: '$speed' }, " +
            "'bearing' : { $avg: '$bearing' }, 'height' : { $avg: '$height' }, " +
            "'pressure' : { $avg: '$pressure' }, 'proximity' : { $avg: '$proximity' } } }")
    List<AverageDataItem> calculateAvgValue();


}
```

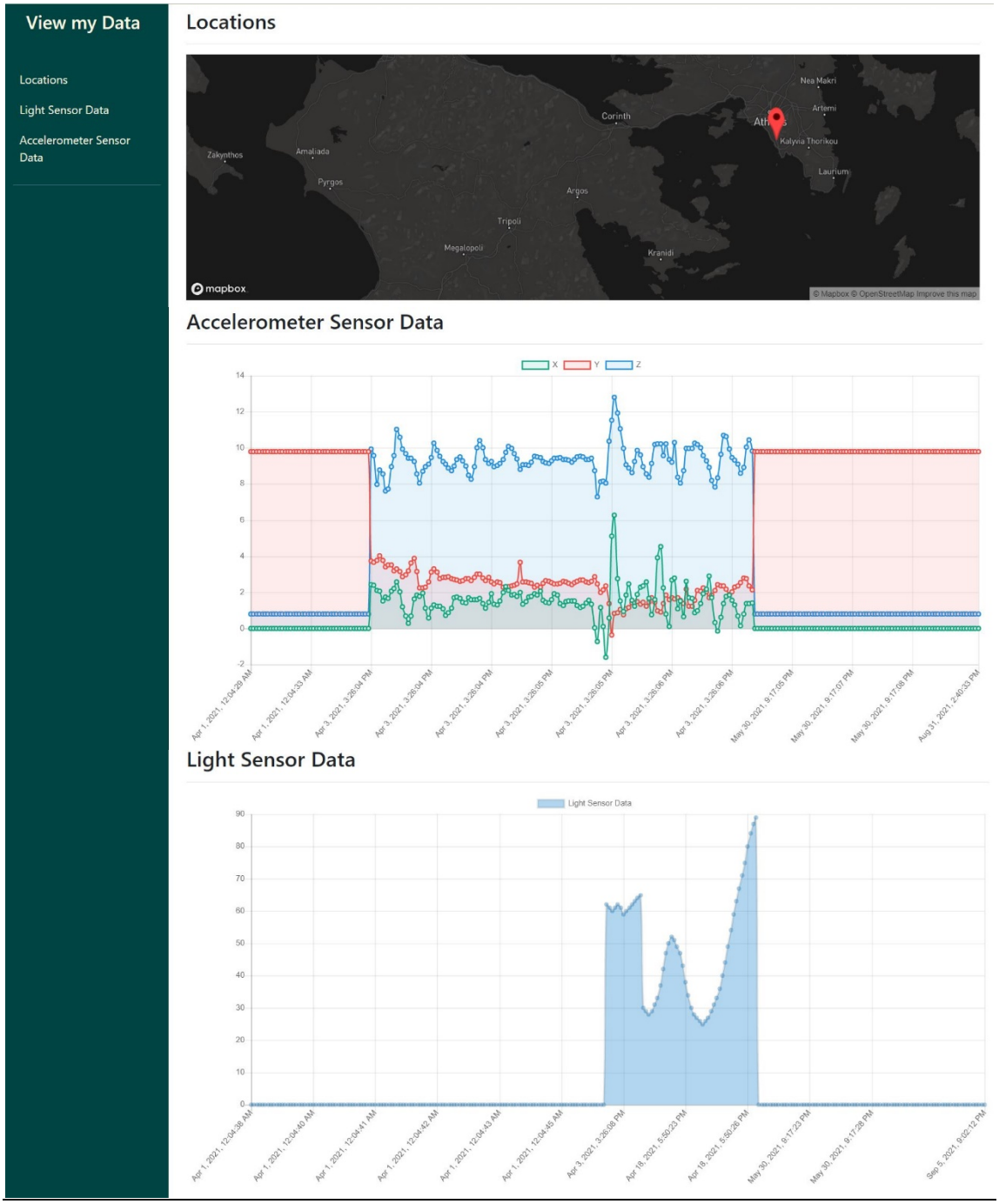
Εικόνα 6-5: Στιγμιότυπο από το Data Search Repository

6.4 Statistics Service

Το microservice Statistics Service είναι εκείνο που αναλαμβάνει να κάνει την απεικόνιση των δεδομένων. Έχει αναπτυχθεί με Spring Boot και το UI είναι γραμμένο σε freemarker, με χρήση html, css, bootstrap και javascript. Έχουν δημιουργήθει δύο σελίδες. Η μία είναι εκείνη που απεικονίζει τα δεδομένα ανά χρήστη και η άλλη είναι εκείνη που απεικονίζει τα συγκεντρωτικά δεδομένα όλων των χρηστών.

6.5 Απεικόνιση Δεδομένων ανά Χρήστη

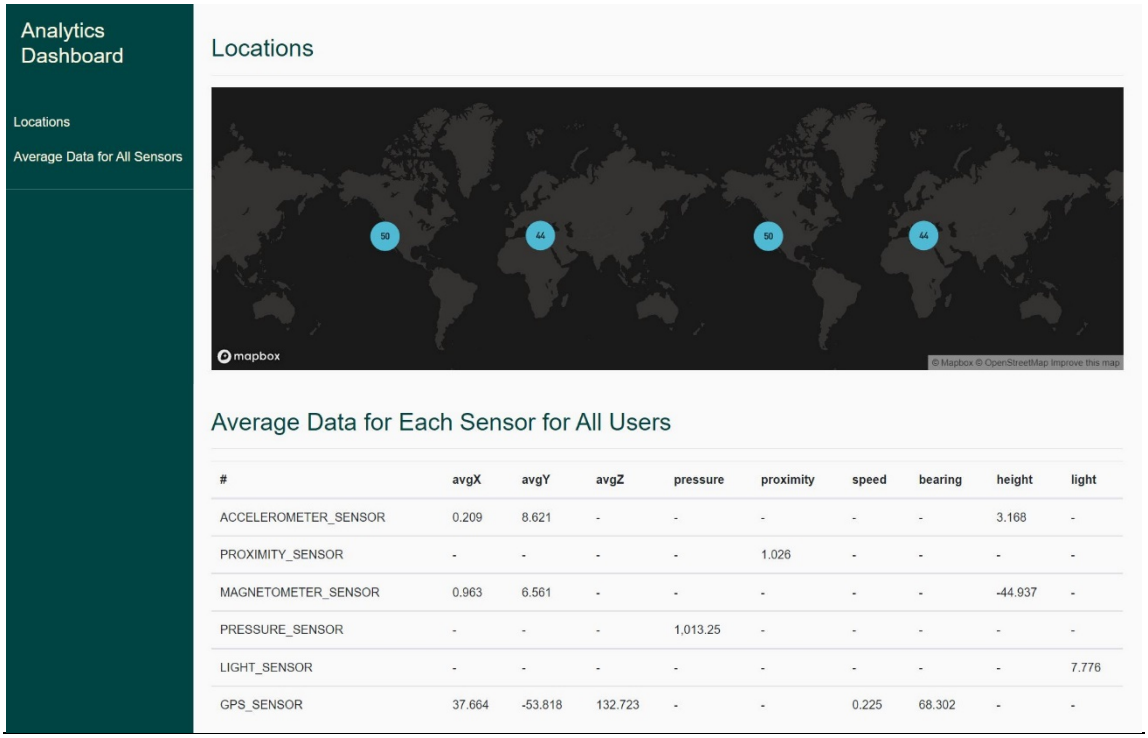
Στη σελίδα αυτή έχει αναπτυχθεί το UI στο οποίο απεικονίζονται τα δεδομένα ανά χρήστη. Εμφανίζονται pins με τις τοποθεσίες του στον χάρτη και γραφήματα με τα δεδομένα από τους υπόλοιπους αισθητήρες.



Εικόνα 6-6: User's UI

6.6 Απεικόνιση Συνολικών Δεδομένων

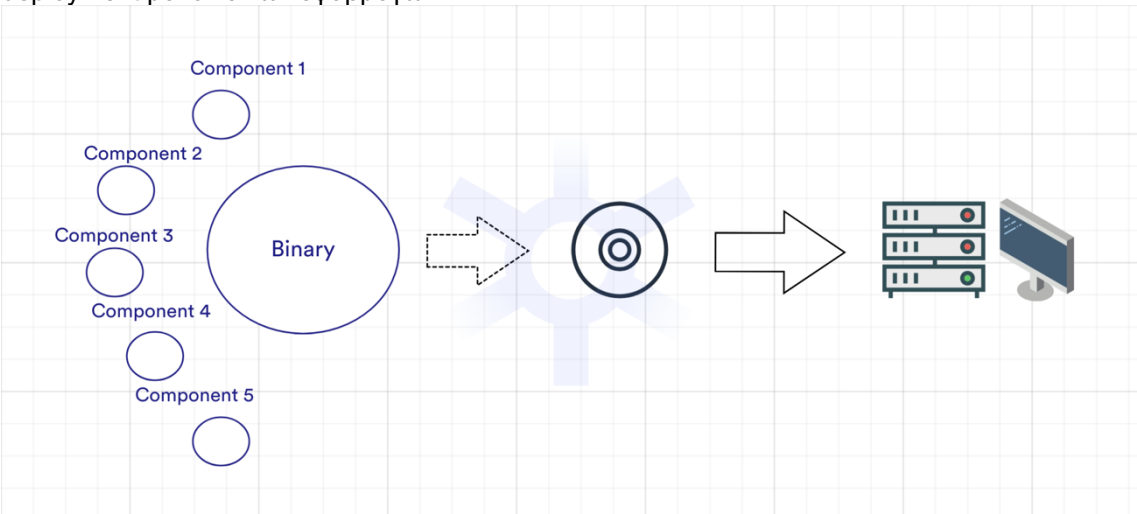
Στο παρακάτω UI απεικονίζονται τα συνολικά δεδομένα όλων των χρηστών της εφαρμογής. Γίνεται clustering των τοποθεσιών για να διευκολυνθεί η απεικόνιση των δεδομένων και παρακάτω απεικονίζονται οι μέσοι όροι όλων αισθητήρων σε μορφή πίνακα.



Εικόνα 6-7: Analytics Dashboard

7 Πρόταση Deployment

Το στάδιο που ακολουθεί μετά την ανάπτυξη του κώδικα είναι εκείνο του deployment. Έχουν αναπτυχθεί έως τώρα τα αρνητικά των μονολιθικών εφαρμογών όσον αφορά την ανάπτυξη, εν προκειμένω όμως το βήμα του deployment. Στην Εικόνα 7-1 απεικονίζεται η διαδικασία του deployment μονολιθικών εφαρμογών.



Εικόνα 7-1: Deployment Μονολιθικής Εφαρμογής

Στις μονολιθικές εφαρμογές όλα τα components είναι απλά τμήματα της ίδιας εφαρμογής και το μόνο που χρειάζεται είναι το compilation και το packaging σε μία μορφή (jar, war) αρχείου που ο server θα μπορέσει να εκτελέσει. Στη συνέχεια το deployment είναι απλά μία μεταφορά και εκτέλεση αρχείου στον εκάστοτε server. Όπως μπορεί να γίνει αντιληπτό, κάτι ανάλογο δεν μπορεί να γίνει με την αρχιτεκτονική του microservice. Κάθε εφαρμογή είναι διαφορετική και μπορεί να τρέχει με ένα ή περισσότερα instances, σε διαφορετικούς servers ή και στον ίδιο.

7.1 Docker

Το Docker είναι μια πλατφόρμα ανοιχτού κώδικα που παρέχει εικονοποίηση (virtualization) σε επίπεδο λειτουργικού συστήματος έτσι ώστε να διευκολύνει την παράδοση λογισμικού σε πακέτα που ονομάζονται containers. Τα πακέτα αυτά είναι απομονωμένα το ένα από το άλλο καθώς και από το φυσικό μηχάνημα στο οποίο εκτελούνται γεγονός που καθιστά εύκολη και επαναλήψιμη την διαδικασία εγκατάστασης του λογισμικού. Τα containers εκτελούνται από ένα λειτουργικό σύστημα, και για τον λόγο αυτό απαιτούν λιγότερους πόρους από τις κλασσικές εικονικές μηχανές (virtual machines) (Docker, 2021).

7.2 Docker Containers

Έχοντας δημιουργήσει τα docker containers και έχοντας τα στο docker registry (ή στο development machine) το μόνο που μένει είναι το τελικό orchestration. Η τελική σύνδεση των containers ώστε να συνδεθούν τα κομμάτια τα οποία θα απαρτίζουν τη συνολική εφαρμογή. Ένας από τους τρόπους που μπορεί να γίνει αυτό και αυτός που προτείνεται για την εφαρμογή είναι το docker compose.

Το Docker αντίθετα από ένα virtual machine, αντί να δημιουργήσει ένα ολόκληρο εικονικό λειτουργικό σύστημα, επιτρέπει στις εφαρμογές να χρησιμοποιούν τον ίδιο πυρήνα Linux με το σύστημα στο οποίο εκτελούν και απαιτεί μόνο να γίνονται deployed εφαρμογές που δεν εκτελούνται ήδη στον κεντρικό υπολογιστή. Αυτό δίνει σημαντική ενίσχυση της απόδοσης και μειώνει το μέγεθος της εφαρμογής (Αγγελόπουλος, 2020).

7.3 Docker Registry

Αφού δημιουργηθούν τα containers, σειρα έχει το deployment το οποίο όμως θα γίνει σε διαφορετικούς servers. Μία λύση θα ήταν να αντιγραφούν τα containers στους servers που θα γίνει το deployment, αλλά ο πιο δόκιμος τρόπος είναι το docker registry, που αποτελεί ένα stateless highly scalable server-side application που δίνει τη δυνατότητα να γίνονται distribute docker images. Φυσικά εκτός του docker registry υπάρχουν και άλλες παρόμοιες εφαρμογές που έχουν αναπτυχθεί για αυτόν τον σκοπό, όπως το Harbor, quay.io, είτε το Azure Container Registry. Κάποια εταιρία/οργανισμός έχει τις εξής επιλογές όσον αφορά το docker registry:

- να κάνει host το δικό του docker registry
- να χρησιμοποιήσει κάποιο paid plan στο Docker Hub
- να χρησιμοποιήσει κάποιο free plan στο Docker Hub (μόνο για public repositories)

7.4 Dockerfiles

Σε όλα τα microservices θα πρέπει να οριστεί στο path ../src/main/docker ένα αρχείο dockerfile. Αυτό περιλαμβάνει τα βήματα με τα οποία θα κατασκευαστεί το docker container για το service αυτό. Παρακάτω παρατίθεται το dockerfile ενός εκ των microservices που αναπτύχθηκαν αυτού του crud:

```
FROM openjdk:8-jdk-alpine
RUN apk update && apk upgrade && apk add netcat-openbsd
RUN mkdir -p /usr/local/crudservice
ADD @project.build.finalName@.jar /usr/local/crudservice
ADD run.sh run.sh
RUN chmod +x run.sh
CMD ./run.sh
```

Εικόνα 7-2: Dockerfile

Αυτές είναι docker εντολές και κάνουν τα εξής:

- **FROM openjdk:8-jdk-alpine**: Δηλώνεται από πιο base container θα ξεκινήσει. Έχει επιλεγεί ένα alpine linux με openjdk toolkit για java 8. Το alpine linux είναι ένα μικρό linux distribution για να κρατηθεί και αντίστοιχα το μέγεθος μικρό. Το image αυτό θα το κατεβάσει από το docker hub.
- **RUN apk update && apk upgrade && apk add netcat-openbsd**: με την εντολή αυτή γίνονται update τα πακέτα του alpine και προστίθεται το netcat-openbsd. Όπως προαναφέρθηκε το alpine έχει όσο το δυνατόν λιγότερα πακέτα by default
- **RUN mkdir -p /usr/local/crudservice**: δημιουργείται μέσα στον linux το φάκελο `/usr/local/crudservice`
- **`ADD @project.build.finalName@.jar /usr/local/crudservice/** : προστίθεται το .jar αρχείο της εφαρμογής στο φάκελο που μόλις δημιουργήθηκε. Το @project.build.finalName@ είναι ένα placeholder για το maven plugin ώστε το όνομα αρχείου να μπορεί να γίνει παραμετροποιήσιμο.
- **ADD run.sh run.sh**: προστίθεται το αρχείο run.sh που βρίσκεται στον ίδιο φάκελο

- **RUN chmod +x run.sh**: παραχωρούνται στο run.sh permissions εκτελέσιμου
- **CMD ./run.sh**: εκτελείται το run.sh

Σκοπός όπως μπορεί κάποιος να παρατηρήσει είναι να δημιουργηθεί το κατάλληλο περιβάλλον για να τρέξει το service, να εισαχθεί το .jar του service και να εκτελεστεί με το run.sh.

Εικόνα 7-3: script run.sh

Όπως παρατηρείται, το run.sh είναι ένα bash script το οποίο περιμένει να αντιληφθεί ότι έχουν ήδη σηκωθεί τα services Eureka Server και Database πριν εκτελέσει το java -jar. Το Crud service χρειάζεται αυτά τα services να έχουν σηκωθεί πριν μπορέσει να λειτουργήσει και έτσι χρησιμοποιείται για να είναι βέβαιο ότι τα services θα τρέξουν με τη σωστή σειρά.

7.5 Docker compose

Έχοντας δημιουργήσει τα docker containers και έχοντας τα στο docker registry (ή στο development machine) το μόνο που μένει είναι το τελικό orchestration. Ένας από τους τρόπους που μπορεί να γίνει αυτό είναι το docker compose.

Το Compose είναι ένα εργαλείο για τον καθορισμό και την εκτέλεση εφαρμογών Docker πολλαπλών κοντέινερ. Με το Compose, ο developer χρησιμοποιεί ένα αρχείο YAML για να διαμορφώσει τα services της εφαρμογής. Στη συνέχεια, με μία μόνο εντολή, δημιουργεί και ξεκινά όλα τα services από το configuration. Η χρήση του Compose είναι βασικά μια διαδικασία τριών βημάτων:

- Ορισμός του περιβάλλοντος της εφαρμογής με ένα αρχείο Docker, ώστε να μπορεί να αναπαραχθεί οπουδήποτε
- Καθορισμός των υπηρεσιών που συνθέτουν την εφαρμογή στο docker-compose.yml έτσι ώστε να μπορούν να λειτουργούν μαζί σε ένα απομονωμένο περιβάλλον.
- Εκτέλεση της εντολής docker-compose up και το Compose εκκινεί και εκτελεί ολόκληρη την εφαρμογή.

Τέλος, εκτελώντας το αρχείο yaml που απαρτίζεται από όλα τα environmental variables που είναι απαραίτητα, ολοκληρώνεται η διαδικασία του deployment.

8 Συμπεράσματα & Προτάσεις

8.1 Συμπεράσματα

Τα microservices όσο περνάει ο καιρός διαδίδονται όλο και περισσότερο και η ενιαία μονολιθική εφαρμογή παραγκωνίζεται σιγά σιγά. Η αποσύνθεση ενός μεγάλου service σε πολλά μικρότερα, κάνει την τελική εφαρμογή πιο ευέλικτη σε αλλαγές και ανάπτυξη, χωρίς αυτό να σημαίνει ότι έρχεται χωρίς δυσκολίες.
Ο developer που θα αναλάβει να αναπτύξει microservices χρειάζεται να αντιμετωπίσει προβλήματα που σε μια παραδοσιακή μονολιθική εφαρμογή δεν υπάρχουν: Το πρόβλημα του service discovery, πώς δηλαδή θα «ανακαλύπτει» το ένα service τα υπόλοιπα. Επίσης πώς θα επικοινωνεί εύκολα με τα υπόλοιπα services. Μαζί με αυτά τα προβλήματα θα πρέπει να αρχίσει να αντιλαμβάνεται και να χρησιμοποιεί patterns που στη μονολιθική εφαρμογή δεν υπάρχουν. Πράγματα που τα θεωρούσε δεδομένα (διευθύνσεις, instances, χρόνος απόκρισης) πλέον είναι ευμετάβλητα.
Κάθε φορά που χρειάζεται να επικοινωνήσει με κάποιο άλλο microservice χρειάζεται να το αντιμετωπίζει σαν call σε κάποιο 3rd party service σε κάποιο άλλο δίκτυο για το οποίο δεν γνωρίζει σχεδόν τίποτα. Σε όλες αυτές τις αλλαγές και τις δυσκολίες έρχεται και ένα νέο σύστημα deployment καθώς χρειάζεται να διαχειριστούμε και να κάνουμε deploy πολλές εφαρμογές ταυτόχρονα.
Σκοπός αυτής της εφαρμογής ήταν να παρουσιάσει βασικούς τρόπους να αντιμετωπιστούν αυτά τα προβλήματα. Να δώσει στον developer εργαλεία για εύκολο service discovery. Να προσφέρει τρόπους για service communication χρησιμοποιώντας resilience design patterns με κατάλληλες βιβλιοθήκες. Και τέλος να δείξει πώς μπορεί ένα containerized deployment σύστημα να ενσωματωθεί με το development.

8.2 Προτάσεις

Όταν κάποιος αναφέρεται σε κάποια εφαρμογή big data σε συνδυασμό με microservices τότε από τα πιο σημαντικά στοιχεία είναι το Logging mechanism το οποίο πρέπει να αναπτυχθεί. Μία αρκετά ολοκληρωμένη λύση είναι το ELK stack. Πρόκειται για το Elasticsearch, Logstash και Kibanna.
Επιπλέον, η δυνατότητα αυτοματοποιημένου scaling αυτών των εφαρμογών είναι υψίστης σημασίας. Μία περαιτέρω διερεύνηση θα μπορούσε να είναι το orchestration να γίνει με Kubernetes αντί του docker compose, να χρησιμοποιηθεί κάποιο queueing mechanism και βάσει αυτού να γίνουν ρυθμίσεις χρησιμοποιώντας το application του KEDA (Kubernetes Event-Driven Autoscaling). Μία πρόταση θα ήταν να ρυθμίζεται ο αριθμός των pod ανάλογα με τα πόσα μηνύματα περιμένουν στην ουρά για να διαβαστούν.
Τέλος, θα ήταν προτιμότερο για την εξαγωγή στατιστικών στοιχείων αλλά και γενικότερα στις περιπτώσεις ανάπτυξης λογισμικού για Big Data να χρησιμοποιείται pySpark.

Βιβλιογραφία

Aptude. (2016). *Benefits and Challenges of Using Microservices with Big Data Applications*. Ανάκτηση από Aptude: https://aptude.com/blog/entry/benefits-and-challenges-of-using-microservices-with-big-data-applications/

Docker. (2021). Ανάκτηση από https://docs.docker.com/

Havey, M. (2008). *SOA Cookbook.*

Kothagal, K. (2019). Ανάκτηση από https://github.com/koushikkothagal/spring-boot-microservices-workshop/tree/master/discovery-server

Mirhoseini, M. (2021). *https://github.com/mohsenoid/close-to-me/*.

Mongo DB. (2021). Ανάκτηση από https://www.mongodb.com/

Spring Boot. (2021). Ανάκτηση από https://spring.io/microservices

Urdhwareshe, R. (2016). *Why to Evolve from Monolithic to Microservices Architecture*. Ανάκτηση από http://www.cuelogic.com/blog/why-to-evolve-from-monolithic-to-microservices-architecture

Zimmermann, O. (2016). Service cutter: A systematic approach to service decomposition. *Service cutter: A systematic approach to service decomposition.*

Αγγελόπουλος, Σ. (2020). *Τεχνικές για ανάπτυξη εφαρμογής με χρήση microservices* . Πανεπιστήμιο Πειραιά.

Ζωγράφου, Μ. -Α. (2021). *Ανάπτυξη Android εφαρμογής Happy Hour Offers.* Πανεπιστήμιο Πειραιώς.