Πανεπιστήμιο Πειραιώς – Τμήμα Πληροφορικής

Πρόγραμμα Μεταπτυχιακών Σπουδών

«Προηγμένα Συστήματα Πληροφορικής - Ανάπτυξη Λογισμικού και Τεχνητής Νοημοσύνης»

Μεταπτυχιακή Διατριβή

Τίτλος Διατριβής	**Υλοποίηση ενός συστήματος υγείας με την χρήση της αρχιτεκτονικής των microservices.** **Development of a Healthcare System using microservices architecture.**
Ονοματεπώνυμο Φοιτητή	**Ωνάσης Παναγιώτης**
Πατρώνυμο	**Κωνσταντίνος**
Αριθμός Μητρώου	**ΜΠΣΠ/ 18028**
Επιβλέπων	**Ευθύμιος Αλέπης, Αναπληρωτής καθηγητής**

Ημερομηνία Παράδοσης: Νοέμβριος 2022

Τριμελής Εξεταστική Επιτροπή

(υπογραφή)	(υπογραφή)	(υπογραφή)
Μαρία Βίρβου Καθηγήτρια	Ευθύμιος Αλέπης Αναπληρωτής καθηγητής	Ευάγγελος Σακκόπουλος Αναπληρωτής καθηγητής

Περιεχόμενα

1. Εισαγωγή – Σύντομη Περιγραφή ... 6
2. Αρχιτεκτονική microservices και μονόλιθοι ... 8
   - 2.1 Ορισμός των Microservices ... 8
   - 2.2 Μονολιθική αρχιτεκτονική ... 8
   - 2.3 Προβλήματα μονολιθικών εφαρμογών ... 8
   - 2.4 Microservices έναντι μονολιθικών εφαρμογών ... 9
3. Τεχνολογίες και σχεδιαστικά πρότυπα ... 11
   - 3.1 Περίληψη Κεφαλαίου ... 11
   - 3.2 Visual Studio 2022 ... 11
   - 3.3 .NET 5.0 ... 11
   - 3.4 Microsoft SQL Server ... 12
   - 3.5 Entity Framework Core ... 13
   - 3.6 Σχεδιαστικά πρότυπα ... 13
   - 3.6.1 Domain Driven Design ... 13
   - 3.6.2 CQRS Pattern ... 19
   - 3.6.3 Retry Pattern ... 21
   - 3.6.4 Gateway Aggregation Pattern ... 23
   - 3.6.5 Circuit Breaker Pattern ... 24
4. Υλοποίηση και Σχεδιασμός Συστήματος ... 25
   - 4.1 Περίληψη Κεφαλαίου ... 25
   - 4.2 Βασικές Λειτουργίες Συστήματος ... 25
   - 4.3 Αρχιτεκτονική συστήματος ... 26
   - 4.4 User Microservice ... 27
   - 4.4.1 Αρχιτεκτονική ... 27
   - 4.4.2 Λειτουργίες και API Requests ... 32
   - 4.4.3 Σχεδιασμός Domain επιπέδου ... 35
   - 4.4.4 Βάση Δεδομένων ... 37
   - 4.5 Patient Microservice ... 38
   - 4.5.1 Αρχιτεκτονική ... 38
   - 4.5.2 Λειτουργίες και API Requests ... 39
   - 4.5.3 Σχεδιασμός Domain επιπέδου ... 41
   - 4.5.4 Βάση Δεδομένων ... 42
   - 4.6 Personnel Microservice ... 45
   - 4.6.1 Αρχιτεκτονική ... 45
   - 4.6.2 Λειτουργίες και API Requests ... 46

4.6.3 Σχεδιασμός Domain επιπέδου ... 49
4.6.4 Βάση Δεδομένων ... 52
4.7 Lis Microservice ... 54
4.7.1 Αρχιτεκτονική ... 54
4.7.2 Λειτουργίες και API Requests ... 55
4.7.3 Σχεδιασμός Domain επιπέδου ... 58
4.7.4 Βάση Δεδομένων ... 61
4.8 Clinical Information Microservice ... 63
4.8.1 Αρχιτεκτονική ... 63
5. Εκτέλεση και χρήση microservices ... 64
5.1 Περίληψη Κεφαλαίου ... 64
5.2 Εκτέλεση Microservice Medbook ... 64
5.3 Περιγραφή σεναρίων χρήσης ... 66
6. Συμπεράσματα και πιθανές επεκτάσεις ... 74
6.1 Συμπεράσματα ... 74
6.2 Μελλοντικές επεκτάσεις ... 74
Βιβλιογραφία ... 77

Περίληψη

Τα τελευταία χρόνια η συνεχόμενη αύξηση της εξάρτησης των ανθρώπων γύρω από τις ηλεκτρονικές συσκευές έχει οδηγήσει στην εμφάνιση ολοένα και περισσότερο απαιτητικών εφαρμογών, οι οποίες χρειάζονται όλο και περισσότερους υπολογιστικούς πόρους αλλά και την εύκολη δυνατότητα επεκτασιμότητας και λειτουργίας τους σε μεγάλη κλίμακα. Το πρόβλημα αυτό λοιπόν προσπαθήσει να επιλύσει η αρχιτεκτονική των microservices. Ο όρος microservices αναφέρθηκε πρώτη φορά σε ένα συνέδριο από τον Dr. Peter Rodgers το 2005 και παρουσιάστηκε με τον όρο «Micro-Web-Services». Σκοπός της συγκεκριμένης αρχιτεκτονικής ήταν η διάσπαση μεμονωμένων μεγάλων μονολιθικών εφαρμογών σε πολλαπλά ανεξάρτητα μικροσυστήματα, καθιστώντας έτσι τον κώδικα πιο απλό , εύκολα διαχειρίσιμο και επεκτάσιμο. Στην παρούσα διπλωματική εργασία λοιπόν θα εξετάσουμε την ανάπτυξη και την υλοποίηση ενός back-end συστήματος με την χρήση microservices, το συγκεκριμένο σύστημα θα αναπτυχθεί πάνω στο τομέα της Υγείας (Ιατρικός Φάκελος) ο οποίος αποτελεί ένα πολύ εύστοχο και σύνθετο σενάριο εφαρμογής ως προς την μοντελοποίηση του σε πολλαπλά microservices. Το συγκεκριμένο σύστημα θα ονομάζεται **Medbook** και θα αποτελείται από πέντε αυτόνομα και ανεξάρτητα microservices, όπου το καθένα θα εξυπηρετεί ένα συγκεκριμένο και μοναδικό σκοπό σε επίπεδο business logic και θα επικοινωνούν μεταξύ τους . Τα microservices αυτά έχουν τις εξής ονομασίες: User, Personel, PatientDemographics, Lis (Laboratory Information System) και ClinicalInformation. Πιο συγκεκριμένα μέσω των συγκεκριμένων microservices θα μπορεί να γίνει η δημιουργία και η διαχείριση ενός Ιατρικού Φακέλου για έναν ασθενή, ο οποίος θα περιέχει όλα τις απαραίτητες λειτουργικότητες για να χρησιμοποιηθεί από ένα η πολλαπλά νοσοκομεία όπως ιστορικό, επισκέψεις σε εξωτερικά η εσωτερικά ιατρεία νοσοκομείων, διεξαγωγή εξετάσεων, καταγραφή αποτελεσμάτων και χορήγηση φαρμάκων.

Abstract

In recent years, the continuous increase of people's dependence around electronic devices has led to the appearance of increasingly demanding applications, which need more and more computing resources but also the easy possibility of their scalability and operation on a large scale. So, the microservices architecture tries to solve this problem. The term microservices was first mentioned in a conference by Dr. Peter Rodgers in 2005 and introduced as "Micro-Web-Services". The purpose of this particular architecture was to break down individual large monolithic applications into multiple independent microsystems, thus making the code simpler, easier to manage, and more extensible. In this thesis, we will examine the development and implementation of a back-end system called Medbook, which will implement a specific business logic, specifically in the field of Healthcare (Patient Medical File) using the microservices architecture. The specific system will consist of five autonomous and independent microservices, where each will serve a specific and unique purpose at the business logic level and will communicate with each other. These microservices have the following names: User, Personel, PatientDemographics, Lis (Laboratory Information System) and ClinicalInformation. More specifically, through the specific microservices, it will be possible to create and manage a Medical File for a patient, which will contain all the necessary functionalities to be used by one or multiple hospitals, such as history, visits to external or internal hospital clinics, conducting examinations, recording results and administering medicines.

1. Εισαγωγή – Σύντομη Περιγραφή

Ο τομέας της υγείας πάντα αποτελούσε ένα θεμελιώδη λίθο μέσα σε μία κοινωνία και αυτό έγινε ακόμα περισσότερο αντιληπτό από την πλειοψηφία των ανθρώπων τα τελευταία χρόνια με την εμφάνιση της πανδημίας του Covid-19. Πλέον η σωστή λειτουργία και παράδοση υπηρεσιών από ένα σύστημα υγείας σε μία χώρα δεν πρέπει απλά να παρέχετε αλλά να αποτελεί βασική προϋπόθεση για να μπορεί η κοινωνία να προχωρήσει μπροστά. Ειδικά κατά την διάρκεια αλλά και μετά την πανδημία οι κυβερνήσεις σχεδόν σε όλο τον κόσμο έδωσαν ιδιαίτερη έμφαση στην ανάπτυξη και στην αναβάθμιση του τομέα της υγείας και από άποψη υποδομών αλλά και από άποψη τεχνολογίας, με αποτέλεσμα τόσο τα δημόσια αλλά και τα ιδιωτικά νοσοκομεία να αναθεωρήσουν τους τρόπους παροχής υπηρεσιών και κοινωνικής φροντίδας. Αυτό είχε ως επακόλουθο οι τεχνολογικές εταιρείες του κλάδου της Υγείας να αναπτύξουν νέα αλλά και να βελτιώσουν υπάρχοντα συστήματα τα οποία θα εξυπηρετούν τις τρέχουσες ανάγκες της κοινωνίας. Παρότι μπορεί να μοιάζει ως κάτι απλό δεν είναι. Ο συγκεκριμένος τομέας περιέχει ένα αρκετά περίπλοκο σύστημα προς μοντελοποίηση έτσι οι εταιρείες που απλά διατήρησαν την τρέχουσα μονολιθική αρχιτεκτονική τους και δεν προσπάθησαν την υλοποίηση ενός συστήματος με microservices θα αντιμετώπισαν πληθώρα προβλημάτων τα οποία θα αναφέρουμε εκτενέστερα αργότερα.

Το αντικείμενο λοιπόν της παρούσας διπλωματικής είναι η σχεδίαση και η υλοποίηση ενός συστήματος υγείας που θα αποτελούσε πρότυπο για τον διαχωρισμό του συγκεκριμένου τομέα στην αρχιτεκτονική των microservices. Για την παρούσα διπλωματική επιλέχθηκαν συγκεκριμένα αλλά και κύρια κομμάτια της λειτουργίας ενός συστήματος υγείας και τα οποία θα προσφέρουν τις κύριες λειτουργικότητες ενός ιατρικού φακέλου ασθενή. Τα microservices αυτά θα αφορούν τις εξής λειτουργίες – δυνατότητες.

- User Authentication (User Api): Διαχειρίζεται την ασφάλεια του συστήματος, την είσοδο των χρηστών και τα δικαιώματα εισόδου σε συγκεκριμένα συστήματα πληροφοριών.
- Personnel Information Management (Personnel Api): Υπεύθυνο για τα στοιχεία του προσωπικού ενός ιατρικού κέντρου για παράδειγμα διαχείριση ιατρών, ωράρια ιατρών και ραντεβού
- Patient Demographics Management (PatientDemographics Api): Υπεύθυνο για την διαχείριση των στοιχείων των ασθενών, τις επισκέψεις τους και το ιστορικό τους.
- LIS (Lis Api): Τα αρχικά σημαίνουν Laboratory Information Systems και χρησιμοποιείται για την εισαγωγή παραγγελιών και αποτελεσμάτων των εξετάσεων.
- Clinical Information Management (ClinicalInfromation Api): Σύστημα το οποίο συνδυάζει τις πληροφορίες από άλλα συστήματα (LIS, Personnel etc) και παρέχει την τελική πληροφορία στον χρήστη

Αυτά θα αποτελούν τα microservices που θα παρουσιάσει η συγκεκριμένη διπλωματική εργασία αλλά η δομή τους θα επιτρέπει την εύκολη προσθήκη νέων και ακόμα περισσότερων microservices για μελλοντική επέκταση.

Η συγκεκριμένη εργασία αποτελείται από πέντε κεφάλαια. Το πρώτο κεφάλαιο αφορά μία γενική εισαγωγή στην αρχιτεκτονική των microservice και τα υπέρ και κατά που έχει να προσφέρει σε σχέση με ένα μονολιθικό σύστημα. Το δεύτερο κεφάλαιο αναλύει και εξηγεί τις τεχνολογίες που χρησιμοποιήθηκαν για την συγκεκριμένη υλοποιήσει και τα σχεδιαστικά πρότυπα που εκτελέστηκαν. Στο τρίτο κεφάλαιο θα γίνει μία αναλυτική επεξήγηση του επιμέρους microservice και του domain στο οποίο αναφαίρετε και του σκοπούς που θα εξυπηρετεί στο συγκεκριμένο σύστημα.

Το τέταρτο κεφάλαιο θα περιέχει παραδείγματα χρήσης του συστήματος τόσο για την πραγματικά σενάρια χρήσης όσο και για τρόπο επικοινωνίας των microservices μέσα στο ενοποιημένο ιατρικό σύστημα που φτιάχτηκε. Το πέμπτο και τελευταίο κεφάλαιο παρουσιάζει τα τελικά συμπεράσματα αλλά και πιθανές μελλοντικές επεκτάσεις ή βελτιώσεις πάνω στο συγκεκριμένο σύστημα microservices που υλοποιήθηκε.

2. Αρχιτεκτονική microservices και μονόλιθοι

2.1 Ορισμός των Microservices

Αδιαμφισβήτητα τα microservices γίνονται όλο και πιο δημοφιλή τα τελευταία χρόνια τόσο από τις μεγάλες εταιρείες αλλά και από τους ίδιους τους προγραμματιστές. Ας ξεκινήσουμε λοιπόν με το τι σημαίνει ακριβώς ο όρος microservice και σε τι απευθύνεται. Microservice με λίγα λόγια είναι η απλοποίηση ενός σύνθετου ενοποιημένου συστήματος (Μονόλιθος) σε ένα σύνολο περισσότερων αλλά και απλούστερων εφαρμογών (services) οι οποίες επικοινωνούν μεταξύ τους μέσω ενός κοινού πρωτοκόλλου όπως είναι το HTTP, WebSocket ή AMQP. Όπως καταλαβαίνεται αυτό σημαίνει πως η αρχιτεκτονική αυτή απευθύνεται κυρίως στο back-end των εφαρμογών και όχι τόσο στο front-end. Κάθε microservice είναι υπεύθυνο να εφαρμόζει και να υλοποιεί μία δικιά του επιχειρησιακή ικανότητα (domain model) και να μπορεί να αναπτυχθεί αυτόνομα και ανεξάρτητα. Επίσης κάθε microservice συνηθίζεται να ζει σε έναν ξεχωριστό μηχάνημα και να έχει την δικιά του βάση δεδομένων η γενικά το δικό του σύστημα διατήρησης δεδομένων (SQL, NoSQL, Redis) . Φυσικά η συγκεκριμένη αρχιτεκτονική δεν πρόκειται για μία λύση που απευθύνεται στην ανάπτυξη κάθε πιθανής εφαρμογής αλλά κυρίως προορίζεται για εφαρμογές στις οποίες το “business logic” είναι αρκετά περίπλοκο, έτσι διαχωρίζοντας το σε επιμέρους microservices θα απλοποιήσει την δομή και τον χρόνο υλοποίησης της εφαρμογής.

2.2 Μονολιθική αρχιτεκτονική

Πριν προχωρήσουμε στην περαιτέρω ανάλυση της αρχιτεκτονικής των microservices ας πούμε δύο λόγια και για τις μονολιθικές εφαρμογές και την αρχιτεκτονική αυτή. Μονολιθική εφαρμογή λοιπόν σημαίνει μία εφαρμογή η οποία έχει ως βάση έναν ενιαίο κώδικα ο οποίος όμως αποτελείται από πολλαπλές ενότητες . Οι μονολιθικές εφαρμογές έχουν σχεδιαστεί έτσι ώστε να είναι αυτόνομες και οι λειτουργίες τους είναι στενά συνδεδεμένες σε αντίθεση με τα microservices. Επίσης οι εφαρμογές που είναι μονολιθικές είναι «Single-tiered» πράγμα που σημαίνει ότι πολλαπλά εξαρτήματα συνδυάζονται σε μία μεγάλη εφαρμογή με πολλαπλές γραμμές κώδικα ανά αρχείο και τείνουν να μεγαλώνουν και να δυσκολεύουν ακόμα περισσότερο με τον χρόνο. Γενικότερα η συγκεκριμένη αρχιτεκτονική είναι ευρέως διαδεδομένη όχι μόνο για την απλή υλοποίηση της αλλά διότι τα περισσότερα εργαλεία ανάπτυξης κώδικα έχουν κατασκευαστεί με στόχο την δημιουργία μίας ενιαίας συνολικής εφαρμογής. Επίσης ως μία μεμονωμένη εφαρμογή, η δημιουργία τεστ είτε χειροκίνητων είτε αυτοματοποιημένων μέσω εργαλείων όπως το selenium για παράδειγμα είναι κάτι πολύ εύκολο. Τέλος ακόμα και όταν χρειαστεί τέτοιες εφαρμογές να γίνουν scale λόγω μεγάλου φόρτου εργασίας το μόνο που χρειάζεται είναι η αντιγραφή της εφαρμογής σε πολλαπλά instance τα οποία θα τρέχουν παράλληλα και απλά θα υπάρχει ένας load balancer ο οποίος θα διανέμει τον φόρτο εργασίας σε αυτά.

2.3 Προβλήματα μονολιθικών εφαρμογών

Δυστυχώς όπως καταλαβαίνετε η παραπάνω αρχιτεκτονική έχει τους περιορισμούς της, έτσι ας δούμε λοιπόν μερικούς από τους λόγους που οδήγησαν στην ανάγκη της αρχιτεκτονικής των microservices. Αρχικά όπως είναι λογικό οι επιτυχημένες εφαρμογές τείνουν να μεγαλώνουν και να επεκτείνονται με την πάροδο του χρόνου, κάτι το οποίο οδηγεί τις μονολιθικές εφαρμογές από μικρές και εύκολα επεκτάσιμες σε μονόλιθους τεράστιων διαστάσεων και αρχεία κώδικα εκατοντάδων γραμμών.

Αυτό λοιπόν έχει σαν αποτέλεσμα να δυσκολεύει τους προγραμματιστές καθώς η ανάπτυξη νέων χαρακτηριστικών ή η διόρθωση κενών και σφαλμάτων γίνεται από πολύ δύσκολη ως και αδύνατη και

χρονοβόρα. Επίσης η διόρθωση προβλημάτων και η ανάπτυξη νέων λειτουργιών κάνει τον πηγαίο κώδικα τόσο πολύπλοκο όπου δυστυχώς θέλοντας η μη δημιουργούνται νέα προβλήματα ή δίνονται λύσεις οι οποίες δεν είναι σωστές. Βέβαια τα προβλήματα δεν σταματάνε εδώ, ο μεγάλος αυτός όγκος δημιουργεί καθυστερήσεις και στην διαδικασία του debugging διότι ένας τέτοιος μεγάλος μονόλιθος θα χρειαστεί ίσως και ολόκληρα λεπτά για να γίνει compile και να ξεκινήσει να τρέχει. Αυτό έχει ως αποτέλεσμα να μεγαλώνει ακόμα περισσότερο ο χαμένος χρόνος του προγραμματιστή καθώς το compile-debugging της εφαρμογής μπορεί να χρειαστεί να γίνει πολλές φορές μέσα σε μία μέρα, μειώνοντας έτσι την παραγωγικότητα του προγραμματιστή.

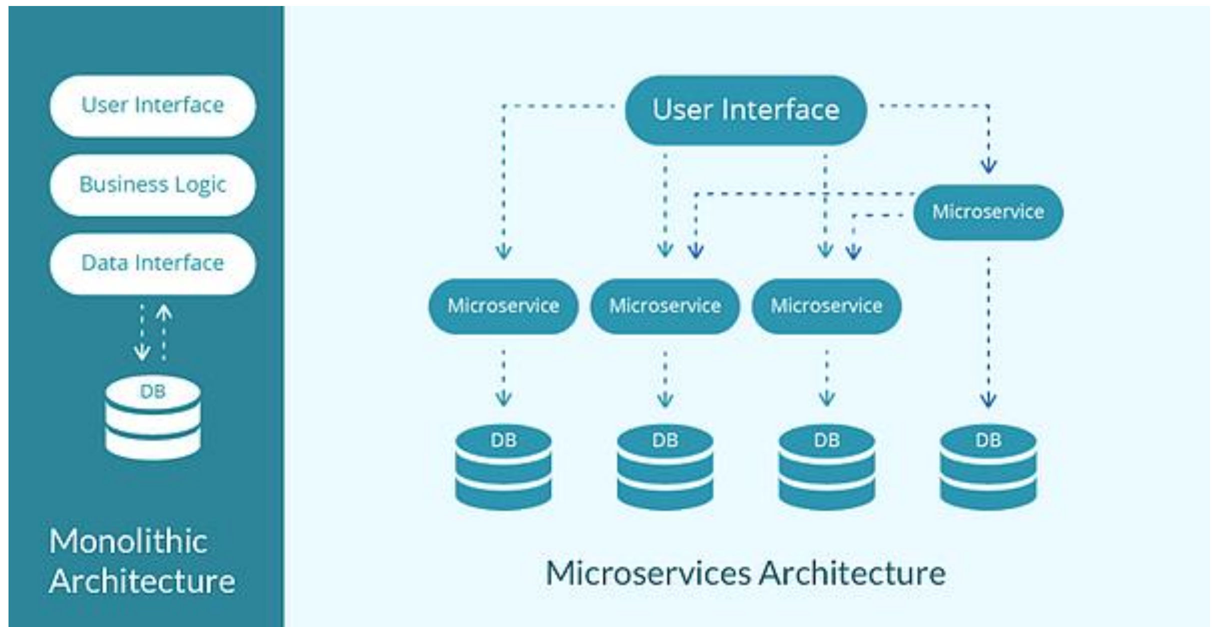
Πριν αν θυμάστε αναφέραμε το πόσο εύκολο είναι το scaling μίας τέτοιας εφαρμογής τι συμβαίνει όμως όταν αυτή γίνει αρκετά περίπλοκη και μεγάλη. Ένα από τα άσχημα σενάρια θα ήταν πως ένα συγκεκριμένο κομμάτι της χρησιμοποιεί πολλούς πόρους με αποτέλεσμα την χρήση μεγάλης επεξεργαστικής ισχύς πράγμα το οποίο άλλα κομμάτια της εφαρμογής να μην χρειάζονται, ένα memory leak θα μπορούσε να γεμίζει την μνήμη ολόκληρου του server η ένας ατέρμονας βρόγχος να ρίξει ολόκληρη την εφαρμογή.

Τέλος η χρήση ενός τέτοιου μονόλιθου περιορίζει ακόμα και τις ίδιες τις ομάδες προγραμματιστών να κάνουν scale τόσο σε επίπεδο υλοποιήσεων αλλά και σε επίπεδο τεχνολογιών. Πιο συγκεκριμένα σε μια μονολιθική εφαρμογή δεν θα μπορούσαμε να σπάσουμε τα νέα χαρακτηριστικά σε πολλαπλές ομάδες ώστε η κάθε μία να αναπτύξει την δικιά της λογική και υλοποίηση πράγμα το οποίο θα ανέβαζε πολύ την παραγωγικότητα. Αλλά ακόμα και αν υπήρχε ένας πρώτος διαχωρισμός και πάλι όλες οι ομάδες θα ήταν δεσμευμένες να χρησιμοποιούν τις ίδιες γλώσσες, τεχνολογίες και frameworks. Το τελικό συμπέρασμα λοιπόν είναι πως όσο η πολυπλοκότητα αυξάνεται στην συγκεκριμένη αρχιτεκτονική τόσο και τα προβλήματα – μειονεκτήματα αυξάνονται.

2.4 Microservices έναντι μονολιθικών εφαρμογών

Ήρθε η ώρα λοιπόν να επικεντρωθούμε στο σήμερα και στα πλεονεκτήματα που μας προσφέρει η αρχιτεκτονική των microservices έναντι των μονόλιθων, πράγμα το οποίο θα είναι και το κύριο θέμα της παρούσας διπλωματικής. Όπως βλέπετε και στην εικόνα 1.1 πιο κάτω από την μία μεριά έχουμε έναν μονόλιθο ο οποίος χρησιμοποιεί την ίδια τεχνολογική υποδομή για όλα τα χαρακτηριστικά του. Από την άλλη μεριά βλέπουμε πως τα microservices έχουν τελείως ξεχωριστές τεχνολογίες και υποδομές το ένα με το άλλο και μπορούν να τρέχουν σε τελείως ξεχωριστά περιβάλλοντα.

Ένα από τα σημαντικότερα πλεονεκτήματα της αρχιτεκτονικής των microservices είναι ότι προσφέρει μία λύση στην αυξανόμενη πολυπλοκότητα ενός συστήματος. Χτίζοντας ένα σύστημα με microservices ο προγραμματιστής αναγκάζεται να χωρίσει την εφαρμογή σε ξεχωριστά services τα οποία όμως έχουν ένα δικό τους domain με σαφή όρια. Με αυτό τον τρόπο μπορούμε να έχουμε εφαρμογές που είναι πολύ πιο σαφή και πολύ περισσότερο «loosely coupled» και μπορούν να αναπτυχθούν ευκολότερα και ταχύτερα.



Εικόνα 1.1 Μονολιθική αρχιτεκτονική έναντι της αρχιτεκτονικής των microservices

Ένα άλλο σημαντικό πλεονέκτημα είναι ότι με αυτή την αρχιτεκτονική κάθε microservice μπορεί να αναπτυχθεί σε οποιαδήποτε γλώσσα και χρησιμοποιώντας οποιαδήποτε τεχνολογία. Το μόνο σταθερό είναι το Restful API που πρέπει να υλοποιεί. Έτσι οι προγραμματιστές μπορούν να επιλέξουν διαφορετικές γλώσσες/τεχνολογίες αναλόγως με τα ιδιαίτερα χαρακτηριστικά που έχει κάθε microservice. Έτσι δίνεται η δυνατότητα σε νέα microservices να χρησιμοποιηθούν τεχνολογίες που δεν υπήρχαν όταν δημιουργήθηκαν προηγούμενα. Τέλος καθώς τα microservices είναι σχετικά μικρά, υπάρχει η δυνατότητα ακόμα και για καθολικής ανάπτυξης από την αρχή με κάποια πιο νέα τεχνολογία.

Σε αντίθεση με τη μονολιθική εφαρμογή όπου το deployment ήταν μία επίπονη και πολύπλοκη διαδικασία, πλέον κάθε microservice γίνεται deployed μόνο του. Το καθένα ακολουθεί το δικό του ρυθμό ανάπτυξης και μπορεί να έχει τα δικά του αυτοματοποιημένα η μη τεστ χωρίς να επηρεάζει τις υλοποιήσεις κάποιου άλλου συστήματος. Σταματά δηλαδή το ένα τμήμα της εφαρμογής να δημιουργεί εμπόδια στο άλλο και όλα να περιμένουν κάποιο τρίτο που ήταν κλασικό σενάριο παλιότερα.

Όπως αναφέραμε, 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 microservices 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.

3. Τεχνολογίες και σχεδιαστικά πρότυπα

3.1 Περίληψη Κεφαλαίου

Στο κεφάλαιο αυτό αναλύονται οι τεχνολογίες και τα σχεδιαστικά πρότυπα που χρησιμοποιηθήκαν για την ανάπτυξη των πέντε microservices που φτιάχτηκαν για την δημιουργία του Medbook συστήματος. Πιο συγκεκριμένα η γλώσσα που επιλέχτηκε είναι η C# και το IDE που χρησιμοποιήθηκε είναι το **Visual Studio 2022**. Επίσης χρησιμοποιήθηκαν οι τεχνολογίες **.NET 5.0** , Entity Framework Core (EFCore), και για την αποθήκευση των δεδομένων χρησιμοποιήθηκε η σχεσιακή βάση δεδομένων της Microsoft, η **MS SQL** και το IDE που προσφέρει για την διαχείριση των βάσεων της το **MS SQL Server Management Studio**. Επιπρόσθετα θα αναφερθούν μερικά από τα σχεδιαστικά πρότυπα που χρησιμοποιήθηκαν όπως CQRS(Command and Query Responsibility Segregation), Retry, Singleton, DDD(Domain Driven Design), HealthChecks αλλά και μια γενικότερη αρχιτεκτονική του συστήματος πέραν από αυτήν των microservices.

3.2 Visual Studio 2022

Το Visual Studio 2022 είναι ένα IDE ειδικά σχεδιασμένο για προγραμματισμό σε Windows. Αναπτύσσεται από την Microsoft και η κύρια εστίασή της είναι η ανάπτυξη σε C++ και C#. Ως IDE το οποίο είναι κυρίως ανεπτυγμένο για την δημιουργία εφαρμογών που χρησιμοποιούν την C++ ή την C# ως γλώσσες ανάπτυξης είναι πολύ λογικό να χρησιμοποιηθεί για την συγκεκριμένη διπλωματική.

3.3 .NET 5.0

Το .NET 5 είναι ένα open-source, cross-platform .NET Framework, το οποίο θα αντικαταστήσει τα .Net Framework, .Net Core και Xamarin με μια ενιαία πλατφόρμα. Το .NET 5 είναι μια ενιαία πλατφόρμα που μπορείτε να χρησιμοποιήσετε για όλους τους σύγχρονους κώδικες .NET με τα πιο πρόσφατα API.

Με το .NET 5, η Microsoft στοχεύει να βελτιώσει το .NET με τους εξής τρόπους:

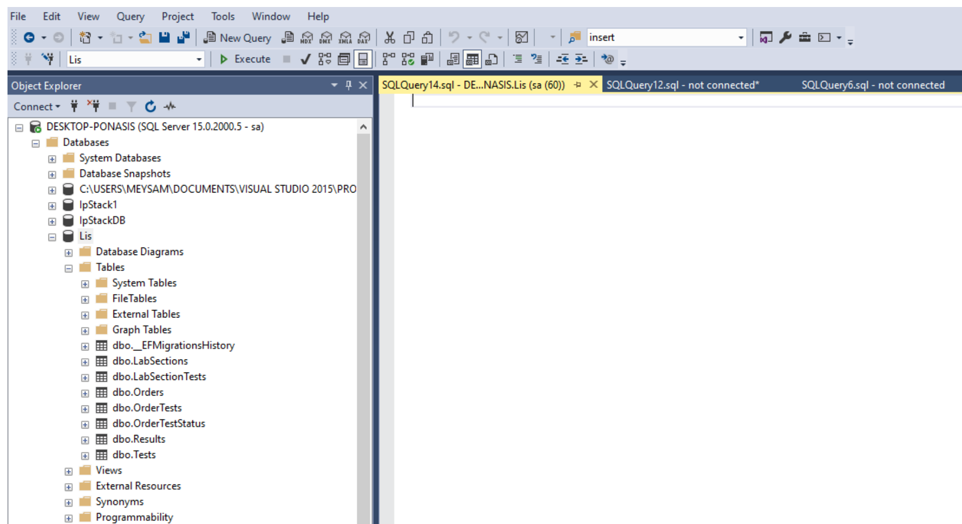
- Να δημιουργήσει ένα ενιαίο πλαίσιο χρόνου εκτέλεσης .NET που θα μπορεί να χρησιμοποιηθεί παντού και θα έχει ομοιόμορφες συμπεριφορές χρόνου εκτέλεσης και εμπειρίες προγραμματιστών.
- Να επεκτείνει τις δυνατότητες του .NET αξιοποιώντας τα καλύτερα των .NET Core, .NET Framework, Xamarin και Mono.
- Να δημιουργήσει αυτό το προϊόν από μια ενιαία βάση κώδικα που οι προγραμματιστές (η Microsoft και η κοινότητα) μπορούν να εργαστούν μαζί και να επεκτείνουν για να βελτιώσουν όλα τα σενάρια.

Εικόνα 2 .Net 5.0

3.4 Microsoft SQL Server

Microsoft SQL Server είναι ένα από τα κύρια συστήματα διαχείρισης σχεσιακών βάσεων δεδομένων στην αγορά που εξυπηρετεί ένα ευρύ φάσμα εφαρμογών λογισμικού για επιχειρηματική ευφυΐα και ανάλυση σε εταιρικά περιβάλλοντα. Βασισμένο στη γλώσσα Transact-SQL, ενσωματώνει ένα σύνολο επεκτάσεων προγραμματισμού τυπικής γλώσσας και η εφαρμογή του είναι διαθέσιμη για χρήση τόσο σε φυσικά μηχανήματα όσο και στο cloud.

Η συγκεκριμένη βάση δεδομένων είναι ιδανικός για την αποθήκευση όλων των επιθυμητών πληροφοριών σε σχεσιακές βάσεις δεδομένων, καθώς και για την διαχείριση τέτοιων δεδομένων χωρίς επιπλοκές, χάρη στο IDE που διαθέτει αλλά και τα εργαλεία αυτού. Παρακάτω έχουμε μία εικόνα απεικόνισης του συγκεκριμένου IDE.



3.5 Entity Framework Core

Για τον σχεδιασμό τη δημιουργία και οποιαδήποτε πράξη ανάγνωσης / εγγραφής προς τη βάση δεδομένων χρησιμοποιήθηκε το πακέτο Entity Framework Core. Πρόκειται για ένα πακέτο ανοικτού κώδικα που λειτουργεί πάνω από την πλατφόρμα του .Net Core. Παρέχει στον χρήστη τη δυνατότητα να αντιστοιχίζει αντικειμένων του κώδικα της εφαρμογής με πίνακες της βάσης δεδομένων (object – relational mapping). Η αντιστοίχιση μεταξύ μοντέλων της βάσης και κλάσεων του κώδικα μπορεί να γίνει και προς τις δύο κατευθύνσεις. Έτσι προκύπτουν δύο βασικοί τρόποι χρήσης του πακέτου.

Database First ονομάζεται η αντιστοίχιση κατά την οποία ο χρήστης δημιουργεί την βάση, τους πίνακες και τις συσχετίσεις μεταξύ των πινάκων με κώδικα SQL απευθείας στην βάση.

Έπειτα συνδέοντας το Entity Framework Core στην έτοιμη πλέον βάση δεδομένων δημιουργούνται αυτόματα οι κλάσεις που χρειάζονται ώστε να αποτυπωθεί το πλήρες σχήμα της βάσης σε επίπεδο κώδικα.

Code First ονομάζεται η προσέγγιση εκείνη κατά την οποία ο χρήστης δημιουργεί τις κλάσεις και τις συσχετίσεις μεταξύ των κλάσεων στον κώδικα του και έπειτα με χρήση του Entity Framework Core σχηματίζονται στη βάση τόσο οι πίνακες όσο και οι συσχετίσεις μεταξύ τους. Αυτή η προσέγγιση χρησιμοποιήθηκε και στην παρούσα διπλωματική εργασία έτσι κάθε εφαρμογή ελέγχει την κατάσταση της βάσης δεδομένων της και την φέρνει στην επιθυμητή κατάσταση, τόσο όσον αφορά τους πίνακες όσο και όσον αφορά στα δεδομένα, κατά την εκκίνηση του.

.

3.6 Σχεδιαστικά πρότυπα

Τα microservices όντας μία πολύ σύγχρονη αρχιτεκτονική η οποία προσφέρει την δημιουργία back-end εφαρμογών οι οποίες είναι ασφαλής, επεκτάσιμες, απλές, αξιόπιστες και υποστηρίξιμες στο cloud είναι πλέον απαιτούμενο αλλά και λογικό να ακολουθούν κάποια πολύ γνωστά “Design Patterns” τα οποία προωθεί ακόμα και η ίδια η Microsoft. Αυτά τα σχεδιαστικά πρότυπα όχι μόνο λύνουν αρκετά αρχιτεκτονικά αλλά και πρακτικά θέματα που ίσως προκύψουν κατά την υλοποίηση αλλά προσφέρουν και έναν πιο ευανάγνωστο κώδικα για τους προγραμματιστές. Έτσι λοιπόν για την ακριβέστερη προσομοίωση μιας αρχιτεκτονικής από microservice στην εργασία αυτή χρησιμοποιήθηκαν αρκετές τέτοιες πρακτικές προτύπων οι οποίες θα αναφερθούν και θα αναλυθούν σε αυτό το κεφάλαιο.

3.6.1 Domain Driven Design

Το σχεδιαστικό πρότυπο **Domain Driven Design (DDD)** υποστηρίζει τη μοντελοποίηση με βάση την επιχειρηματική πραγματικότητα ως σχετική με τις περιπτώσεις χρήσης. Στο πλαίσιο της δημιουργίας εφαρμογών, το DDD μιλάει για προβλήματα ως τομείς (Domains) . Περιγράφει τις ανεξάρτητες προβληματικές περιοχές ως **Bounded Contexts** (κάθε bounded context συσχετίζεται με ένα microservice) και δίνει έμφαση σε μια κοινή γλώσσα για να μιλήσουμε για αυτά τα προβλήματα. Προτείνει επίσης πολλές τεχνικές έννοιες και μοτίβα, όπως domain entities με πλούσια μοντέλα (no anemic-domain model), value objects , aggregates και root aggregates (root entity) για την υποστήριξη της εσωτερικής υλοποίησης. Αυτή η ενότητα εισάγει τον σχεδιασμό και την υλοποίηση αυτών των εσωτερικών προτύπων του DDD.

Μερικές φορές αυτοί οι τεχνικοί κανόνες και μοτίβα DDD γίνονται αντιληπτοί ως εμπόδια που έχουν μια απότομη καμπύλη μάθησης για την εφαρμογή προσεγγίσεων DDD. Αλλά το σημαντικό μέρος δεν είναι τα ίδια τα μοτίβα, αλλά η οργάνωση του κώδικα ώστε να ευθυγραμμίζεται με τα επιχειρηματικά προβλήματα και να χρησιμοποιεί τους ίδιους επιχειρηματικούς όρους (ubiquitous language). Επιπλέον, οι προσεγγίσεις DDD θα πρέπει να εφαρμόζονται μόνο εάν εφαρμόζετε πολύπλοκα microservices με σημαντικούς επιχειρηματικούς κανόνες. Οι απλούστερες ευθύνες, όπως μια ένα CRUD API, μπορούν να αντιμετωπιστούν με απλούστερες προσεγγίσεις και χωρίς την χρήση του DDD.

Το βασικό καθήκον κατά το σχεδιασμό και τον καθορισμό ενός microservice είναι που θα θέσουμε τα όρια. Τα μοτίβα DDD σάς βοηθούν να κατανοήσετε την πολυπλοκότητα στον τομέα. Για το Domain επίπεδο για κάθε Bounded Context, προσδιορίζετε και ορίζετε τις οντότητες(entities), τα αντικείμενα τιμών(value objects) και τα συγκεντρωτικά στοιχεία(aggregates) που μοντελοποιούν τον τομέα σας. Δημιουργείτε και τελειοποιείτε ένα μοντέλο τομέα που περιέχεται σε ένα όριο που καθορίζει το περιβάλλον σας. Και αυτό είναι ρητό με τη μορφή του microservice. Τα στοιχεία εντός αυτών των ορίων καταλήγουν να είναι τα δικά σας microservices, αν και σε ορισμένες περιπτώσεις μια BC ή μια επιχείρηση με microservices μπορεί να αποτελείται από πολλές φυσικές υπηρεσίες. Το DDD αφορά τα όρια και το ίδιο ισχύει και για τα microservices.

Εδώ ας κάνουμε μία μικρή επεξήγηση στα τρία είδη των οντοτήτων που υπάρχουν μέσα σε ένα Domain επίπεδο και τα οποία αναφέρθηκαν ονομαστικά πριν από λίγο.

- Entities
- Value objects
- Aggregates & Roots

Entities

Όπως αναφέρει και ο Eric Evans (συγγραφέας του βιβλίου Domain-Driven Design: Tackling Complexity in the Heart of Software) “πολλά αντικείμενα δεν ορίζονται θεμελιωδώς από τα χαρακτηριστικά τους, αλλά μάλλον από ένα νήμα συνέχειας και ταυτότητας”. Entity επομένως είναι ένα αντικείμενο που ορίζεται κυρίως από την ταυτότητά του και που έχει σημασία (π.χ. Πελάτης) στο σύστημα πωλήσεων είναι μια οντότητα και μπορεί να αλλάξει με την πάροδο του χρόνου.

Value Objects

Συνεχίζοντας με τα Value Objects εδώ ο Eric Evans λέει “Πολλά αντικείμενα δεν έχουν εννοιολογική ταυτότητα. Αυτά τα αντικείμενα περιγράφουν χαρακτηριστικά ενός πράγματος” . Τα Value Objects είναι αντικείμενα που είναι γνωστά μόνο από τις ιδιότητες και τις τιμές τους. Για παράδειγμα, η "Διεύθυνση πελάτη" μπορεί να σχεδιαστεί ως Value Object. Τα Value Objects μπορούν να εκχωρηθούν σε διαφορετικές οντότητες και συνήθως υλοποιούνται ως immutable (π.χ. ημερομηνία, διεύθυνση)

Aggregates & Roots

Τέλος μένουν τα Aggregates, τα οποία σχεδιάζουν ένα όριο γύρω από μία ή περισσότερες οντότητες. Ένα Aggregate επιβάλλει αμετάβλητες για όλες τις οντότητες του για οποιαδήποτε λειτουργία υποστηρίζει. Κάθε Aggregate έχει ένα Root Entity, το οποίο είναι το μόνο μέλος του Aggregate στο οποίο επιτρέπεται να έχει αναφορά σε οποιοδήποτε αντικείμενο εκτός του Aggregate. Όπως ο Eric Evans ανέφερε στο βιβλίο του, οι κανόνες που πρέπει να επιβάλουμε περιλαμβάνουν:

- Το Root Entity έχει παγκόσμια ταυτότητα και είναι τελικά υπεύθυνη για τον έλεγχο των αναλλοίωτων (invariants)
- Το Root Entity έχει παγκόσμια ταυτότητα. Τα Entities εντός του ορίου έχουν τοπική ταυτότητα, μοναδική μόνο εντός του Aggregate.
- Τίποτα έξω από το όριο του Aggregate δεν μπορεί να περιέχει αναφορά σε οτιδήποτε μέσα, εκτός από το Root Entity. Το Root Entity μπορεί να παραδώσει αναφορές στις εσωτερικά Entities σε άλλα αντικείμενα, αλλά μπορεί να τις χρησιμοποιήσει μόνο παροδικά (μέσα σε μία μόνο μέθοδο).
- Μόνο τα Aggregate Roots μπορούν να ληφθούν απευθείας με ερωτήματα βάσης

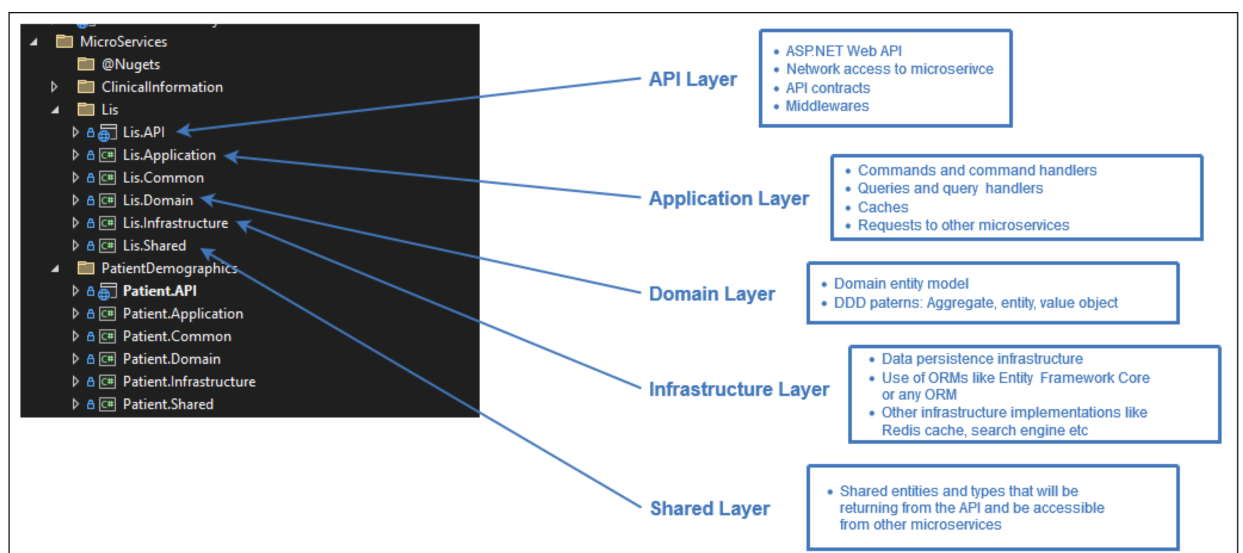
δεδομένων (database queries). Όλα τα άλλα πρέπει να γίνονται μέσω διέλευσης δια μέσω του Root Aggregate.

- Τα αντικείμενα μέσα σε ένα Aggregate μπορούν να περιέχουν αναφορές σε άλλα Aggregate Roots.
- Μια λειτουργία διαγραφής πρέπει να καταργήσει τα πάντα εντός του ορίου ενός Aggregate ταυτόχρονα.
- Όταν δεσμεύεται μια αλλαγή σε οποιοδήποτε αντικείμενο εντός του ορίου του Aggregate, πρέπει να ικανοποιούνται όλα τα αμετάβλητα ολόκληρου του Aggregate.

Αφού λοιπόν δόθηκαν κάποιες λεπτομέρειες για τους τύπους των οντοτήτων μέσα στο Domain επίπεδο θα συνεχίσουμε με την περιγραφή των επίπεδων στα οποία χωρίζει το DDD ένα microservice και συγκεκριμένα πως έγινε στην παρούσα διπλωματική εργασία. Αρχικά οι περισσότερες εταιρικές εφαρμογές με σημαντική επιχειρηματική και τεχνική πολυπλοκότητα ορίζονται από πολλαπλά επίπεδα. Τα επίπεδα είναι ένα λογικό τεχνούργημα και δεν σχετίζονται με την ανάπτυξη της υπηρεσίας. Υπάρχουν για να βοηθήσουν τους προγραμματιστές να διαχειριστούν την πολυπλοκότητα στον κώδικα. Διαφορετικά επίπεδα (όπως το domain και το presentation επίπεδο κ.λπ.) μπορεί να έχουν διαφορετικούς τύπους, οι οποίοι επιβάλλουν μεταφράσεις μεταξύ αυτών των τύπων. Για παράδειγμα, μια οντότητα θα μπορούσε να φορτωθεί από τη βάση δεδομένων. Στη συνέχεια, μέρος αυτών των πληροφοριών ή μια συγκέντρωση πληροφοριών που περιλαμβάνει πρόσθετα δεδομένα από άλλες οντότητες, μπορεί να σταλεί στη διεπαφή χρήστη του πελάτη μέσω ενός REST Web API. Το θέμα εδώ είναι ότι η domain οντότητα περιέχεται στο domain επίπεδο και δεν πρέπει να διαδίδεται σε άλλες περιοχές στις οποίες δεν ανήκει, όπως στο presentation layer.

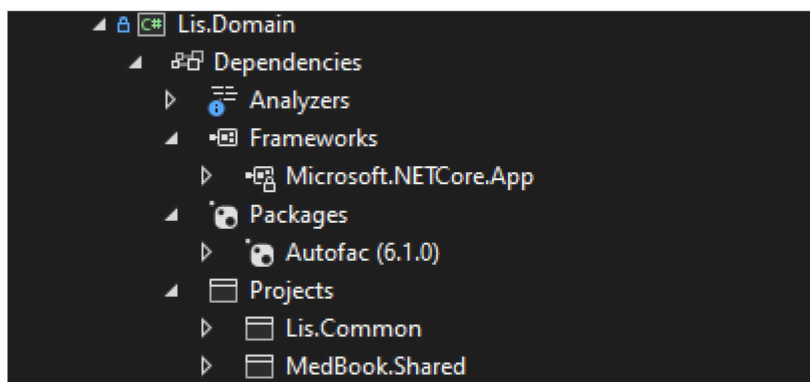
Επιπλέον, πρέπει να υπάρχουν πάντα έγκυρες οντότητες που ελέγχονται από aggregate root (root entities). Επομένως, οι οντότητες δεν θα πρέπει να δεσμεύονται σε προβολές πελατών(client views), επειδή σε επίπεδο διεπαφής χρήστη ορισμένα δεδομένα ενδέχεται να μην έχουν ακόμη επικυρωθεί. Αυτός είναι ο λόγος για τον οποίο χρησιμεύει το ViewModel. Το ViewModel είναι ένα μοντέλο δεδομένων αποκλειστικά για ανάγκες επιπέδου παρουσίασης(presentation layer). Οι οντότητες τομέα(domain layer) δεν ανήκουν απευθείας στο ViewModel. Αντίθετα, πρέπει να κάνετε μετάφραση μεταξύ ViewModels και οντοτήτων τομέα και αντίστροφα.

Όταν αντιμετωπίζετε την πολυπλοκότητα, είναι σημαντικό να έχετε ένα domain model που ελέγχεται από aggregate roots και διασφαλίζει ότι όλες οι μεταβλητές και οι κανόνες που σχετίζονται με αυτήν την ομάδα οντοτήτων (aggregate) εκτελούνται μέσω ενός μόνο σημείου εισόδου ή πύλης, του aggregate root.



Εικόνα 3. Επίπεδα της αρχιτεκτονικής DDD στο LIS microservice

Στην Εικόνα 3 λοιπόν διακρίνετε ο διαχωρισμός του LIS microservice του συστήματος Medbook σε 5 επίπεδα. Όπως φαίνεται κάθε επίπεδο είναι ένα DLL του Visual Studio, τα βασικά επίπεδα που αναφέρει και το DDD είναι Lis.API (API layer), Lis.Domain (Domain layer), Lis.Infrastructure (Infrastructure layer) όμως εκτός από αυτά έχει γίνει ένας επιπλέον διαχωρισμός και έχουν δημιουργηθεί και τα Lis.Application και Lis.Shared. Είναι σημαντικό να σημειωθεί πως το κάθε επίπεδο μπορεί να επικοινωνεί μόνο με ορισμένα επίπεδα και όχι με όλα και για αυτό το λόγω η λογική αυτή έχει σπάσει σε διαφορετικές βιβλιοθήκες (DLL) και όχι σε απλούς φακέλους για παράδειγμα, ώστε δηλαδή να υπάρχει σαφήνεια ποιο DLL εξαρτάται από ποιο (μέσω των dependencies που διαθέτει κάθε DLL Εικόνα 4.)



Εικόνα 4. Βιβλιοθήκες που υπάρχουν στο Domain Layer

Επιστρέφοντας ξανά λοιπόν στα επίπεδα ας δούμε μερικά παραδείγματα εξάρτησης (reference) , το επίπεδο του Domain (Lis.Domain) δεν πρέπει να εξαρτάται από καμία άλλη προσαρμοσμένη βιβλιοθήκη μέσα στο Lis microservice (API ή Infrastructure Layer) κάτι το οποίο συμβαίνει όπως βλέπουμε και στην Εικόνα 5. (εξαιρούνται τα Lis.Common & Medbook.Shared διότι παρέχουν επεκτάσεις βοηθητικών κλάσεων) αλλά μόνο από το .NetCore.nuget . Αντιθέτως η εσωτερική βιβλιοθήκη ή επίπεδο Lis.Infrastructure χρειάζεται και πρέπει να χρησιμοποιεί το Domain επίπεδο κάτι το οποίο είναι υποχρεωτικό και αναγκαίο.

Κλείνοντας με την συγκεκριμένη αρχιτεκτονική θα γίνει μία σύντομη περιγραφή για το τι θα πρέπει να περιέχουν τα τρία βασικά επίπεδα του προτύπου αυτού δηλαδή το Domain, Infrastructure , Application.

Domain Επίπεδο

Υπεύθυνο για την αναπαράσταση εννοιών της επιχείρησης, πληροφορίες σχετικά με την κατάσταση της επιχείρησης και επιχειρηματικούς κανόνες. Η κατάσταση που αντικατοπτρίζει την επιχειρηματική κατάσταση ελέγχεται και χρησιμοποιείται εδώ, παρόλο που οι τεχνικές λεπτομέρειες της αποθήκευσής της ανατίθενται στο Infrastructure. Αυτό το επίπεδο είναι η καρδιά του επιχειρηματικού λογισμικού. Το Domain επίπεδο είναι το σημείο όπου εκφράζεται η επιχείρηση. Όταν υλοποιείτε ένα Domain layer για ένα microservice στο .NET, αυτό το επίπεδο κωδικοποιείται ως βιβλιοθήκη κλάσεων με τις οντότητες του Domain που καταγράφουν δεδομένα συν τη συμπεριφορά (methods with logic). Ακολουθώντας τις αρχές του Persistence Ignorance και της Infrastructure Ignorance, αυτό το επίπεδο πρέπει να αγνοεί εντελώς τις λεπτομέρειες της αποθήκευσης δεδομένων. Αυτές οι εργασίες αποθήκευσης θα πρέπει να εκτελούνται από το Infrastructure επίπεδο. Επομένως, αυτό το επίπεδο δεν πρέπει να έχει άμεσες εξαρτήσεις από το Infrastructure, πράγμα που σημαίνει ότι ένας σημαντικός κανόνας είναι ότι οι κλάσεις οντοτήτων του Domain μας πρέπει να είναι POCOs(Plain Old CLR Objects).

Οι οντότητες ενός Domain δεν πρέπει να έχουν καμία άμεση εξάρτηση (όπως να προέρχονται από μια βασική κλάση) από οποιοδήποτε πλαίσιο υποδομής πρόσβασης δεδομένων όπως το Entity Framework ή το NHibernate. Στην ιδανική περίπτωση, οι οντότητες του Domain δεν θα πρέπει να προέρχονται από ή να εφαρμόζουν οποιονδήποτε τύπο που ορίζεται σε οποιοδήποτε πλαίσιο του Infrastructure. Τα περισσότερα σύγχρονα πλαίσια ORM όπως το Entity Framework Core επιτρέπουν αυτήν την προσέγγιση, έτσι ώστε οι κατηγορίες μοντέλων τομέα σας να μην συνδέονται με το Infrastructure. Ωστόσο, η ύπαρξη οντοτήτων POCO δεν είναι πάντα δυνατή όταν χρησιμοποιείτε ορισμένες βάσεις δεδομένων και πλαίσια NoSQL, όπως Actors and Reliable Collections στο Azure Service Fabric.

Ακόμη και όταν είναι σημαντικό να ακολουθήσετε την αρχή της Άγνοιας Εμμονής για το μοντέλο του Domain σας, δεν πρέπει να αγνοήσετε τους προβληματισμούς σχετικά με την επιμονή. Είναι ακόμα σημαντικό να κατανοήσετε το μοντέλο φυσικών δεδομένων και τον τρόπο με τον οποίο αντιστοιχίζεται στο μοντέλο αντικειμένου της οντότητάς σας. Διαφορετικά μπορείτε να δημιουργήσετε αδύνατα σχέδια. Επίσης, αυτή η πτυχή δεν σημαίνει ότι μπορείτε να πάρετε ένα μοντέλο σχεδιασμένο για μια σχεσιακή βάση δεδομένων και να το μετακινήσετε απευθείας σε μια βάση δεδομένων NoSQL ή προσανατολισμένη σε έγγραφα. Σε ορισμένα μοντέλα οντοτήτων, το μοντέλο μπορεί να ταιριάζει, αλλά συνήθως δεν ταιριάζει. Εξακολουθούν να υπάρχουν περιορισμοί στους οποίους πρέπει να τηρεί το μοντέλο της οντότητάς σας, με βάση τόσο την τεχνολογία αποθήκευσης όσο και την τεχνολογία ORM.

Application Επίπεδο

Καθορίζει τις εργασίες που υποτίθεται ότι πρέπει να κάνει το λογισμικό και κατευθύνει τα αντικείμενα του Domain για να επιλύσουν προβλήματα. Οι εργασίες για τις οποίες είναι υπεύθυνο αυτό το επίπεδο είναι σημαντικές για την επιχείρηση ή απαραίτητες για την αλληλεπίδραση με τα επίπεδα εφαρμογής άλλων συστημάτων. Αυτό το στρώμα διατηρείται λεπτό. Δεν περιέχει επιχειρηματικούς κανόνες ή γνώσεις, αλλά συντονίζει μόνο εργασίες και αναθέτει εργασίες σε αντικείμενα στο επίπεδο του Domain. Δεν έχει κατάσταση που αντικατοπτρίζει την κατάσταση της επιχείρησης, αλλά μπορεί να έχει κατάσταση που αντικατοπτρίζει την πρόοδο μιας εργασίας για τον χρήστη ή το πρόγραμμα.

Το Application επίπεδο ενός microservice στο .NET κωδικοποιείται συνήθως ως έργο ASP.NET Core Web API. Το έργο υλοποιεί την αλληλεπίδραση του microservice, την απομακρυσμένη πρόσβαση στο δίκτυο και τα εξωτερικά API Web που χρησιμοποιούνται από το περιβάλλον χρήστη ή τις εφαρμογές πελάτη. Περιλαμβάνει ερωτήματα εάν χρησιμοποιείται μια προσέγγιση CQRS, εντολές που γίνονται αποδεκτές από το microservice, ακόμη και την επικοινωνία που βασίζεται σε συμβάντα μεταξύ των microservice (συμβάντα ενοποίησης). Το ASP.NET Core Web API το οποίο αντικατοπτρίζει το Application επίπεδο πρέπει να συντονίζει μόνο εργασίες και δεν πρέπει να διατηρεί ή να ορίζει καμία κατάσταση τομέα (Domain Layer). Αναθέτει την εκτέλεση επιχειρηματικών κανόνων στις ίδιες τις κλάσεις του Domain (Aggregate Roots & domain entities), οι οποίες τελικά θα ενημερώσουν τα δεδομένα σε αυτές τις οντότητες του Domain.

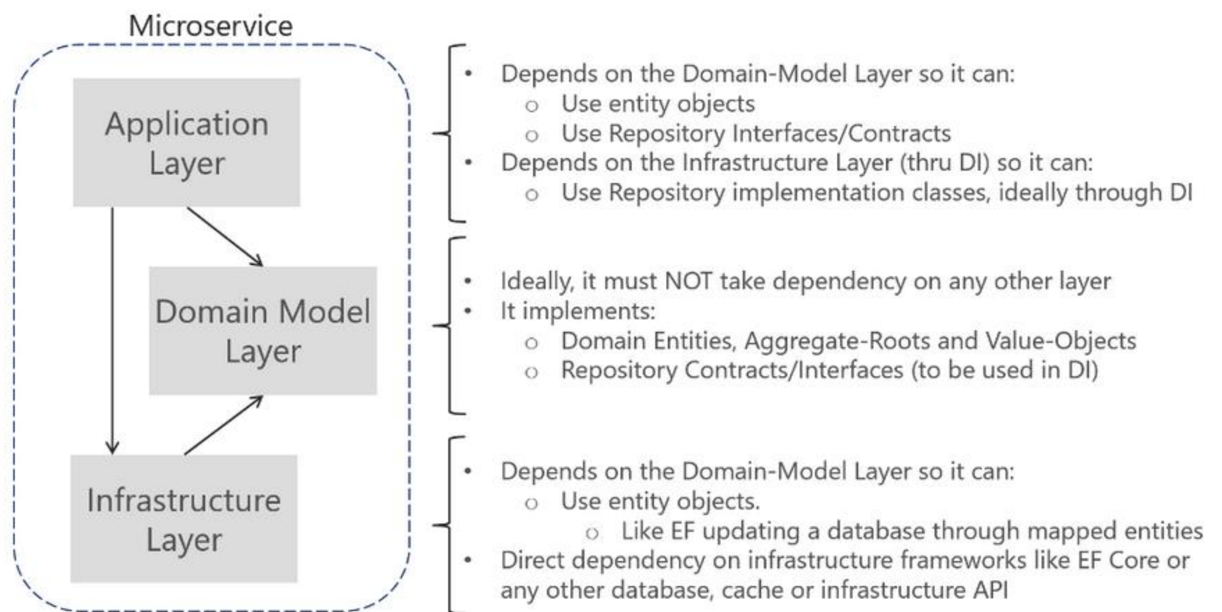
Βασικά, η λογική του Application Layer υλοποιεί όλες τις περιπτώσεις χρήσης που εξαρτώνται από μια δεδομένη διεπαφή. Για παράδειγμα, η υλοποίηση που σχετίζεται με μια υπηρεσία Web API.

Ο στόχος είναι ότι το Domain επίπεδο, τα μοντέλα του Domain, τα αμετάβλητά του, το μοντέλο δεδομένων και οι σχετικοί επιχειρηματικοί κανόνες πρέπει να είναι εντελώς ανεξάρτητοι από τα επίπεδα παρουσίασης και εφαρμογής(Representation and application layer). Πάνω απ 'όλα, το επίπεδο του Domain δεν πρέπει να εξαρτάται άμεσα από το επίπεδο του Infrastructure.

Infrastructure Επίπεδο

Κλείνοντας το επίπεδο υποδομής(Infrastructure) είναι ο τρόπος με τον οποίο τα δεδομένα που διατηρούνται αρχικά σε Domain οντότητες (στη μνήμη) διατηρούνται σε βάσεις δεδομένων ή σε άλλο μόνιμο χώρο αποθήκευσης. Ένα παράδειγμα είναι η χρήση του Entity Framework Core για την υλοποίηση των κλάσεων μοτίβων αποθετηρίου(Repository Pattern) που χρησιμοποιούν ένα DBCcontext για να διατηρηθούν δεδομένα σε μια σχεσιακή βάση δεδομένων.

Σύμφωνα με τις αρχές που αναφέρθηκαν προηγουμένως επίμονη άγνοια και άγνοια υποδομής, το επίπεδο υποδομής δεν πρέπει να "μολύνει" το επίπεδο μοντέλου τομέα(Domain Layer). Πρέπει να διατηρήσετε τις κλάσεις οντοτήτων μοντέλου τομέα (Domain Models) αγνωστικιστές από την υποδομή(Infrastructure Layer) που χρησιμοποιείτε για την διατήρηση δεδομένων (EF ή οποιοδήποτε άλλο πλαίσιο) μη λαμβάνοντας σκληρές εξαρτήσεις από πλαίσια. Η βιβλιοθήκη του Domain Layer θα πρέπει να έχει μόνο τον κωδικό που αφορά το Domain σας, απλώς κλάσεις οντοτήτων POCO που υλοποιούν την καρδιά του λογισμικού σας και πλήρως αποσυνδεδεμένες από τεχνολογίες υποδομής.



Εικόνα 5. Εξαρτήσεις μεταξύ επιπέδων στο DDD

3.6.2 CQRS Pattern

Το CQRS σημαίνει Command and Query Responsibility Segregation, ένα μοτίβο που διαχωρίζει τις λειτουργίες ανάγνωσης και ενημέρωσης για ένα χώρο αποθήκευσης δεδομένων. Η εφαρμογή CQRS στην εφαρμογή σας μπορεί να μεγιστοποιήσει την απόδοση, την επεκτασιμότητα και την ασφάλειά της. Η ευελιξία που δημιουργείται με τη μετάβαση στο CQRS επιτρέπει σε ένα σύστημα να εξελίσσεται καλύτερα με την πάροδο του χρόνου και αποτρέπει τις εντολές ενημέρωσης από το να προκαλούν συγκρούσεις συγχώνευσης σε επίπεδο Domain.

Πιο αναλυτικά το CQRS διαχωρίζει τις αναγνώσεις και τις εγγραφές σε διαφορετικά μοντέλα, χρησιμοποιώντας εντολές(Commands) για την ενημέρωση δεδομένων και ερωτήματα(Queries) για την ανάγνωση δεδομένων.

- Οι εντολές(Commands) θα πρέπει να βασίζονται σε εργασίες και όχι σε δεδομένα. ("Book hotel room", not "set ReservationStatus to Reserved ").
- Οι εντολές(Commands) μπορούν να τοποθετηθούν σε μια ουρά για ασύγχρονη επεξεργασία, αντί να υποβάλλονται σε σύγχρονη επεξεργασία.
- Τα ερωτήματα (Queries) δεν τροποποιούν ποτέ τη βάση δεδομένων. Ένα ερώτημα επιστρέφει ένα DTO που δεν ενσωματώνει καμία γνώση στο Domain.

Η ύπαρξη ξεχωριστών μοντέλων ερωτημάτων και ενημερώσεων απλοποιεί τη σχεδίαση και την υλοποίηση. Ωστόσο, ένα μειονέκτημα είναι ότι ο κώδικας CQRS δεν μπορεί να δημιουργηθεί αυτόματα από ένα σχήμα βάσης δεδομένων χρησιμοποιώντας μηχανισμούς αυτοματοποίησης, όπως εργαλεία O/RM.

Για μεγαλύτερη απομόνωση, μπορείτε να διαχωρίσετε φυσικά τα δεδομένα ανάγνωσης από τα δεδομένα εγγραφής. Σε αυτήν την περίπτωση, η βάση δεδομένων ανάγνωσης μπορεί να χρησιμοποιήσει το δικό της σχήμα δεδομένων που είναι βελτιστοποιημένο για ερωτήματα. Για παράδειγμα, μπορεί να αποθηκεύσει μια υλοποιημένη προβολή των δεδομένων, προκειμένου να αποφευχθούν πολύπλοκες ενώσεις ή σύνθετες αντιστοιχίσεις O/RM. Μπορεί ακόμη και να χρησιμοποιεί διαφορετικό τύπο αποθήκευσης δεδομένων. Για παράδειγμα, η βάση δεδομένων εγγραφής μπορεί να είναι σχεσιακή, ενώ η βάση δεδομένων ανάγνωσης είναι μια βάση δεδομένων εγγράφων.

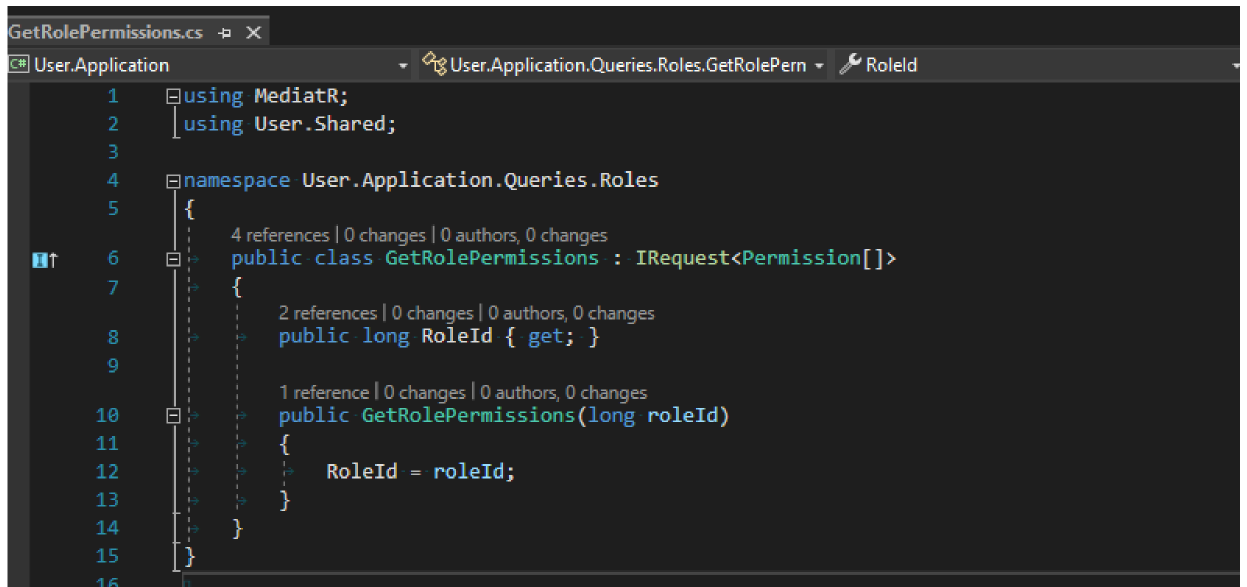
Εάν χρησιμοποιούνται ξεχωριστές βάσεις δεδομένων ανάγνωσης και εγγραφής, πρέπει να διατηρούνται συγχρονισμένες. Συνήθως αυτό επιτυγχάνεται βάζοντας το μοντέλο εγγραφής να δημοσιεύει ένα συμβάν κάθε φορά που ενημερώνει τη βάση δεδομένων.

Μερικά από τα πλεονεκτήματα που προσφέρει το σχεδιαστικό πρότυπο CQRS είναι:

- Ανεξάρτητη κλιμάκωση. Το CQRS επιτρέπει στους φόρτους εργασίας ανάγνωσης και εγγραφής να κλιμακώνονται ανεξάρτητα και μπορεί να οδηγήσει σε λιγότερες διενέξεις κλειδώματος
- Βελτιστοποιημένα σχήματα δεδομένων. Η πλευρά ανάγνωσης μπορεί να χρησιμοποιεί ένα σχήμα που είναι βελτιστοποιημένο για ερωτήματα, ενώ η πλευρά εγγραφής χρησιμοποιεί ένα σχήμα που είναι βελτιστοποιημένο για ενημερώσεις.
- Ασφάλεια. Είναι πιο εύκολο να διασφαλίσετε ότι μόνο οι κατάλληλες οντότητες τομέα εκτελούν εγγραφή στα δεδομένα.
- Διαχωρισμός ανησυχιών. Ο διαχωρισμός των πλευρών ανάγνωσης και εγγραφής μπορεί να οδηγήσει σε μοντέλα που είναι πιο συντηρήσιμα και ευέλικτα. Το μεγαλύτερο μέρος της πολύπλοκης επιχειρηματικής λογικής πηγαίνει στο μοντέλο εγγραφής. Το μοντέλο ανάγνωσης μπορεί να είναι σχετικά απλό.
- Πιο απλά ερωτήματα. Αποθηκεύοντας μια υλοποιημένη προβολή στη βάση

δεδομένων ανάγνωσης, η εφαρμογή μπορεί να αποφύγει πολύπλοκες συνδέσεις κατά την υποβολή ερωτημάτων.

Τέλος θα δούμε πως το συγκεκριμένο πρότυπο χρησιμοποιήθηκε μέσα στο σύστημα Medbook και στα microservices. Αρχικά για την υλοποίηση του βοήθησε μια .Net βιβλιοθήκη η οποία ονομάζεται Mediator και μας παρέχει κάποια Interfaces οι οποίες μας βοηθάνε στην υλοποίηση του CQRS. Ξεκινώντας από την Εικόνα 6. βλέπουμε μία κλάση η οποία θα αποτελεί το μοντέλο ανάγνωσης, η συγκεκριμένη κλάση ονομάζεται GetRolePermissions και περιέχει μέσα το RoleId το οποίο θα χρησιμοποιηθεί για να πάρουμε τα συγκεκριμένα RolePermissions. Έπειτα με την βοήθεια του IRequest interface (Mediator Intreface) δηλώνουμε τον τύπο απάντησης για το συγκεκριμένο Query, έτσι μέχρι τώρα έχουμε καταφέρει μέσα σε μία πολύ απλή κλάση να ορίσουμε το αντικείμενο που ψάχνουμε αλλά και τα δεδομένα τα οποία χρειαζόμαστε για να το βρούμε.



Εικόνα 6. Query Model

Αυτό που λείπει τώρα λοιπόν από την υλοποίηση μας είναι για το που θα βρίσκεται η υλοποίηση του κώδικα η οποία θα ψάχνει με το παραπάνω μοντέλο. Η απάντηση αυτή βρίσκεται στην Εικόνα 7, όπως φαίνεται η κλάση GetRolePermissionsHandler αποτελεί των κώδικα ο οποίος θα διαχειριστεί το παραπάνω μοντέλο όταν σταλεί από το API. Για να γνωρίζει ο Handler ποιο Query υλοποιεί θα πρέπει να κληρονομήσει και αυτός με την σειρά του από ένα Interface το οποίο ονομάζεται IRequestHandler και αποτελεί και αυτό κομμάτι της βιβλιοθήκης του Mediator. Εδώ παρατηρείτε όμως πως μόνο αυτό δεν φτάνει αλλά θα πρέπει να δηλωθούν συγκεκριμένα ο τύπος του Query Model (GetRolePermissions) αλλά και η κλάση η οποία θα επιστρέφει μέσα της τα δεδομένα από το αποτέλεσμα του QueryHandler. Το τελικό βήμα μετά είναι η υλοποίηση του κώδικα που θα μας επιστρέψει τα δεδομένα που θέλαμε μέσα στο Handle method που αναγκάστηκα πρέπει να υλοποιηθεί λόγω κληρονομικότητας.

```
using MediatR;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using User.Application.Infrastructure.Repositories;
using User.Shared;

namespace User.Application.Queries.Roles
{
    1 reference | 0 changes | 0 authors, 0 changes
    class GetRolePermissionsHandler : IRequestHandler<GetRolePermissions, Permission[]>
    {
        private readonly IPermissionRepository _permissionRepository;

        0 references | 0 changes | 0 authors, 0 changes
        public GetRolePermissionsHandler(IPermissionRepository permissionRepository)
        {
            _permissionRepository = permissionRepository;
        }

        0 references | 0 changes | 0 authors, 0 changes
        public async Task<Permission[]> Handle(GetRolePermissions request, CancellationToken cancellationToken)
        {
            var permissions = await _permissionRepository.GetPermissionsByRoleId(request.RoleId);
            return permissions
                .Select(permission => new Permission(
                    id: permission.Id,
                    sysname: permission.Sysname,
                    name: permission.Sysname,
                    description: permission.Description))
                .ToArray();
        }
    }
}
```

Εικόνα 7. Query Handler

3.6.3 Retry Pattern

Το Retry Pattern κάνει μια εφαρμογή να χειρίζεται παροδικές αποτυχίες όταν προσπαθεί να συνδεθεί σε μια υπηρεσία ή έναν πόρο δικτύου, δοκιμάζοντας ξανά με διαφάνεια μια αποτυχημένη λειτουργία. Αυτό μπορεί να βελτιώσει τη σταθερότητα της εφαρμογής. Στο cloud και ειδικά μεταξύ της επικοινωνίας πολλαπλών microservices είναι αρκετά συχνό κάποιες από αυτές τις επικοινωνίες να αποτυγχάνουν είτε λόγω θεμάτων δικτύου, είτε λόγω προβλήματος σε κάποιο microservice παρόλα αυτά το αποτέλεσμα είναι το ίδιο δηλαδή σφάλμα στην επικοινωνία. Για αυτό λοιπόν τα microservices πρέπει να είναι έτοιμα να διαχειριστούν τέτοιες καταστάσεις εδώ έρχεται το Retry pattern, το οποίο ελαχιστοποιεί τα προβλήματα και τις επιπτώσεις σε μία επιχείρηση η προϊόν το οποίο βρεθεί σε μία τέτοια κατάσταση.

Εάν λοιπόν μια εφαρμογή εντοπίσει μια αποτυχία όταν προσπαθεί να στείλει ένα αίτημα σε μια απομακρυσμένη υπηρεσία, μπορεί να χειριστεί την αποτυχία χρησιμοποιώντας τις ακόλουθες στρατηγικές:

- Ματαίωση(Cancel). Εάν το σφάλμα υποδεικνύει ότι η αποτυχία δεν είναι παροδική ή είναι απίθανο να είναι επιτυχής εάν επαναληφθεί, η εφαρμογή θα πρέπει να ακυρώσει τη λειτουργία και να αναφέρει μια εξαίρεση. Για παράδειγμα, μια αποτυχία ελέγχου ταυτότητας που προκαλείται από την παροχή μη έγκυρων διαπιστευτηρίων δεν είναι πιθανό να επιτύχει, όσες φορές κι αν επιχειρηθεί.

- Ξαναδοκιμάσετε(Retry). Εάν το συγκεκριμένο σφάλμα που αναφέρεται είναι ασυνήθιστο ή σπάνιο, μπορεί να προκλήθηκε από ασυνήθιστες περιστάσεις, όπως ένα πακέτο δικτύου που καταστράφηκε κατά τη μετάδοσή του. Σε αυτήν την περίπτωση, η εφαρμογή θα μπορούσε να δοκιμάσει ξανά το αίτημα που αποτυγχάνει ξανά αμέσως, επειδή η ίδια αποτυχία είναι απίθανο να επαναληφθεί και το αίτημα θα είναι πιθανώς επιτυχές.
- Προσπαθήστε ξανά μετά από καθυστέρηση(retry after delay). Εάν το σφάλμα προκαλείται από μία από τις πιο συνηθισμένες αποτυχίες συνδεσιμότητας ή αστοχίες απασχολημένου, το δίκτυο ή η υπηρεσία ενδέχεται να χρειαστεί ένα σύντομο χρονικό διάστημα μέχρι να διορθωθούν τα προβλήματα συνδεσιμότητας ή να εκκαθαριστεί το ανεκτέλεστο έργο. Η εφαρμογή θα πρέπει να περιμένει για έναν κατάλληλο χρόνο πριν δοκιμάσει ξανά το αίτημα.

Για τις πιο συνηθισμένες παροδικές αποτυχίες, η περίοδος μεταξύ των επαναλήψεων θα πρέπει να επιλέγεται για να κατανέμονται τα αιτήματα από πολλές περιπτώσεις της εφαρμογής όσο το δυνατόν πιο ομοιόμορφα. Αυτό μειώνει την πιθανότητα να συνεχίσει να υπερφορτώνεται μια πολυάσχολη υπηρεσία. Εάν πολλές περιπτώσεις μιας εφαρμογής κατακλύζουν συνεχώς μια υπηρεσία με αιτήματα επανάληψης, θα χρειαστεί περισσότερος χρόνος για την ανάκτηση της υπηρεσίας.

Εάν το αίτημα εξακολουθεί να αποτύχει, η εφαρμογή μπορεί να περιμένει και να κάνει άλλη μια προσπάθεια. Εάν είναι απαραίτητο, αυτή η διαδικασία μπορεί να επαναληφθεί με αυξανόμενες καθυστερήσεις μεταξύ των προσπαθειών επανάληψης, έως ότου επιχειρηθεί κάποιος μέγιστος αριθμός αιτημάτων. Η καθυστέρηση μπορεί να αυξηθεί σταδιακά ή εκθετικά, ανάλογα με τον τύπο της αστοχίας και την πιθανότητα να διορθωθεί κατά τη διάρκεια αυτής της περιόδου. Το παρακάτω διάγραμμα απεικονίζει την επίκληση μιας λειτουργίας σε μια φιλοξενούμενη υπηρεσία χρησιμοποιώντας αυτό το μοτίβο. Εάν το αίτημα είναι ανεπιτυχές μετά από έναν προκαθορισμένο αριθμό προσπαθειών, η εφαρμογή θα πρέπει να αντιμετωπίσει το σφάλμα ως εξαίρεση και να το χειριστεί αναλόγως. Στην παρούσα εργασία για την υλοποίηση του συγκεκριμένου πρότυπου χρησιμοποιήθηκε η βιβλιοθήκη Polly.

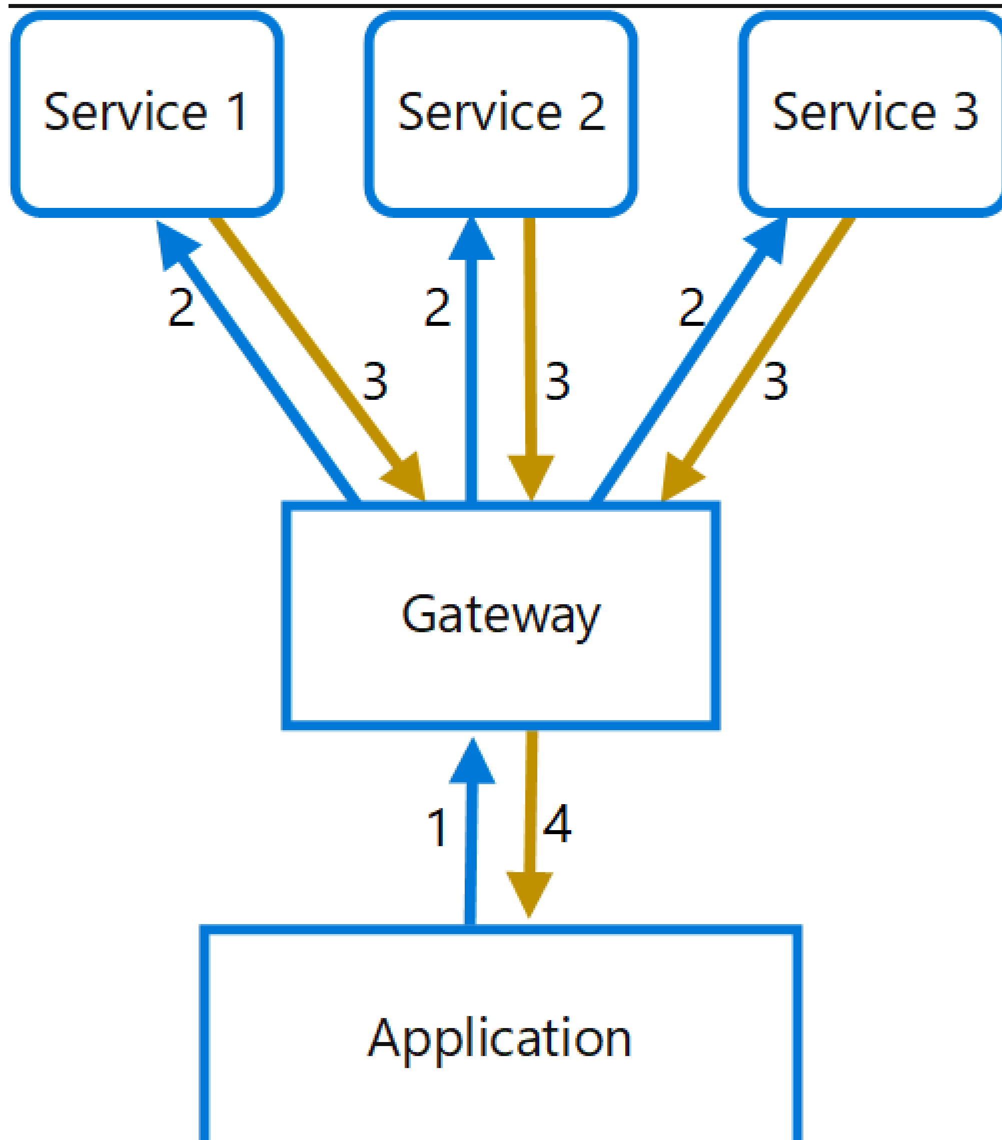
1: Application invokes operation on hosted service. The request fails, and the service host responds with HTTP response code 500 (internal server error).
2: Application waits for a short interval and tries again. The request still fails with HTTP response code 500.
3: Application waits for a longer interval and tries again. The request succeeds with HTTP response code 200 (OK).

Εικόνα 8. Παράδειγμα Retry Pattern

3.6.4 Gateway Aggregation Pattern

Το συγκεκριμένο πρότυπο χρησιμεύει ώστε να χρησιμοποιηθεί μια πύλη για να συγκεντρωθούν πολλαπλά μεμονωμένα αιτήματα σε ένα μόνο αίτημα. Αυτό το μοτίβο είναι χρήσιμο όταν ένας πελάτης πρέπει να πραγματοποιήσει πολλές κλήσεις σε διαφορετικά συστήματα υποστήριξης για να εκτελέσει μια λειτουργία. Αναλυτικότερα όταν ένα σύστημα όπως και το Medbook στην συγκεκριμένη περίσταση αποτελείται από πολλαπλά microservice θα υπάρξουν σίγουρα ανάγκες στις οποίες θα χρειαστεί να αντληθεί πληροφορία για ένα αίτημα από παραπάνω από ένα microservices, στην περίπτωση αυτή κάποιοι developers ίσως απλά έφτιαχναν το front-end App να στέλνει τρία διαφορετικά Http Requests και έπειτα να μαζεύει τις πληροφορίες και να τις συνδυάζει κάτι το οποίο δεν θεωρείται ούτε καθαρή αλλά ούτε βέλτιστη λύση.

Στην περίπτωση μας λοιπόν και με την βοήθεια της θεωρίας πίσω από το συγκεκριμένο μοτίβο υλοποιήθηκε ένας διαμεσολαβητής που έχουμε αναφέρει και νωρίτερα το ClinicalInformation microservice, αυτό λοιπόν το σύστημα είναι υπεύθυνο να στείλει πολλαπλά requests προς τα core microservices του Medbook και να συλλέξει όλες τις απαραίτητες πληροφορίες ώστε να επιστρέψει μια ολοκληρωμένη απάντηση στον αιτούντα πελάτη η οποία εν τέλη θα φανεί στο front-end App. Το διάγραμμα στην Εικόνα 9 είναι ένα ωραίο παράδειγμα για την υλοποίηση που μόλις αναφέρθηκε.



Εικόνα 9. Σχεδιάγραμμα Gateway

3.6.5 Circuit Breaker Pattern

Το μοτίβο Circuit Breaker, που διαδόθηκε από τον Michael Nygard στο βιβλίο του, Release It, μπορεί να αποτρέψει μια εφαρμογή από το να προσπαθεί επανειλημμένα να εκτελέσει μια λειτουργία που είναι πιθανό να αποτύχει. Επιτρέποντάς του να συνεχίσει χωρίς να περιμένει να διορθωθεί το σφάλμα ή να σπαταλήσει τους κύκλους της CPU, ενώ προσδιορίζει ότι το σφάλμα είναι μακροχρόνιο. Το μοτίβο Circuit Breaker επιτρέπει επίσης σε μια εφαρμογή να ανιχνεύσει εάν το σφάλμα έχει επιλυθεί. Εάν το πρόβλημα φαίνεται να έχει επιλυθεί, η εφαρμογή μπορεί να προσπαθήσει να επικαλεστεί τη λειτουργία.

Για να μην υπάρξει σύγχυσει ας αναφερθεί εδώ πως, ο σκοπός του μοτίβου Circuit Breaker είναι διαφορετικός από το μοτίβο Retry. Το μοτίβο Retry επιτρέπει σε μια εφαρμογή να δοκιμάσει ξανά μια λειτουργία με την προσδοκία ότι θα πετύχει. Το μοτίβο Circuit Breaker εμποδίζει μια εφαρμογή να εκτελέσει μια λειτουργία που είναι πιθανό να αποτύχει. Μια εφαρμογή μπορεί να συνδυάσει αυτά τα δύο μοτίβα χρησιμοποιώντας το μοτίβο Επανάληψη για να ενεργοποιήσει μια λειτουργία μέσω ενός διακόπτη κυκλώματος. Ωστόσο, η λογική επανάληψης δοκιμής θα πρέπει να είναι ευαίσθητη σε τυχόν εξαιρέσεις που επιστρέφονται από τον διακόπτη κυκλώματος και να εγκαταλείπει τις προσπάθειες επανάληψης εάν ο διακόπτης κυκλώματος υποδεικνύει ότι ένα σφάλμα δεν είναι παροδικό.

Ένας διακόπτης κυκλώματος λειτουργεί ως πληρεξούσιος για λειτουργίες που ενδέχεται να αποτύχουν. Ο διακομιστής μεσολάβησης θα πρέπει να παρακολουθεί τον αριθμό των πρόσφατων αποτυχιών που έχουν συμβεί και να χρησιμοποιεί αυτές τις πληροφορίες για να αποφασίσει εάν θα επιτρέψει τη συνέχιση της λειτουργίας ή απλώς να επιστρέψει αμέσως μια εξαίρεση.

Ο διακομιστής μεσολάβησης μπορεί να υλοποιηθεί ως μηχάνημα κατάστασης με τις ακόλουθες καταστάσεις που μιμούνται τη λειτουργικότητα ενός διακόπτη ηλεκτρικού κυκλώματος:

- Closed: Το αίτημα από την εφαρμογή δρομολογείται στη λειτουργία. Ο διακομιστής μεσολάβησης διατηρεί μια καταμέτρηση του αριθμού των πρόσφατων αποτυχιών και εάν η κλήση στη λειτουργία δεν είναι επιτυχής, ο διακομιστής μεσολάβησης αυξάνει αυτή τη μέτρηση. Εάν ο αριθμός των πρόσφατων αποτυχιών υπερβεί ένα καθορισμένο όριο εντός μιας δεδομένης χρονικής περιόδου, ο διακομιστής μεσολάβησης τοποθετείται στην κατάσταση Ανοιχτό. Σε αυτό το σημείο ο διακομιστής μεσολάβησης ξεκινά ένα χρονόμετρο λήξης και όταν λήξει αυτός ο χρονοδιακόπτης, ο διακομιστής μεσολάβησης τοποθετείται στην κατάσταση Half-Open.
- Open: Το αίτημα από την εφαρμογή αποτυγχάνει αμέσως και μια εξαίρεση επιστρέφεται στην εφαρμογή.
- Half-Open: Ένας περιορισμένος αριθμός αιτημάτων από την εφαρμογή επιτρέπεται να περάσουν και να καλέσουν τη λειτουργία. Εάν αυτά τα αιτήματα είναι επιτυχή, θεωρείται ότι το σφάλμα που προκαλούσε προηγουμένως την αστοχία έχει διορθωθεί και ο διακόπτης κυκλώματος μεταβαίνει στην κατάσταση Κλειστό (ο μετρητής αστοχίας επαναφέρεται). Εάν οποιοδήποτε αίτημα αποτύχει, ο διακόπτης κυκλώματος υποθέτει ότι το σφάλμα εξακολουθεί να υπάρχει, επομένως επιστρέφει στην κατάσταση Ανοιχτό και επανεκκινεί το χρονόμετρο λήξης για να δώσει στο σύστημα μια επιπλέον χρονική περίοδο για να ανακάμψει από την αστοχία.

4. Υλοποίηση και Σχεδιασμός Συστήματος

4.1 Περίληψη Κεφαλαίου

Στο κεφάλαιο αυτό θα αναλυθεί ο τρόπος υλοποίησης του back-end συστήματος του Medbook, πως δομήθηκε και γιατί πάρθηκαν κάποιες σχεδιαστικές αποφάσεις αλλά και μία πιο αναλυτική παρουσίαση του κάθε microservice που το αποτελεί και για ποιο σκοπό συγκεκριμένα δημιουργήθηκε αλλά και τις δυνατότητες που προσφέρει στο σύνολο του συστήματος.

4.2 Βασικές Λειτουργίες Συστήματος

Όπως αναφέρθηκε και σε προηγούμενα κεφάλαια η συγκεκριμένη διπλωματική εργασία παρουσιάζει ένα Ιατρικό σύστημα το οποίο έχει αναπτυχθεί με την αρχιτεκτονική των microservices και θα είναι υπεύθυνο να εκτελεί μερικές από τις βασικές λειτουργίες ενός ιατρικού φακέλου ασθενούς. Ας αναφέρουμε μερικές λειτουργίες ώστε να γίνει πιο κατανοητός ο λόγος διαχωρισμός των microservices με τον τρόπο τον οποίο συνέβη.

Αρχικά η δημιουργία ενός τέτοιου συστήματος έχει ως ανάγκη την ασφάλεια των προσωπικών δεδομένων των ασθενών και των ιατρικών αποτελεσμάτων τους αυτό λοιπόν δημιουργεί την ανάγκη για έναν τρόπο ταυτοποίησης για κάθε ένα άτομο που θέλει να κάνει είσοδο στο σύστημα είτε είναι ιατρικό προσωπικό είτε είναι ένας απλός ασθενής. Οπότε δημιουργία λογαριασμό χρηστών οι οποίοι θα μπορούν να διαχειριστούν από τον κάθε χρήστη και να δημιουργούνται από το ιατρικό προσωπικό. Επίσης σε ένα ιατρικό σύστημα θα πρέπει να υπάρχει έλεγχος στο επίπεδο το οποίο κάθε μέλος του ιατρικού προσωπικού έχει πρόσβαση για παράδειγμα ένας ιατρός ο οποίος ανήκει στο Παιδιατρικό τμήμα ενός νοσοκομείου δεν θα μπορεί να προσθέσει μικροβιολογικά αποτελέσματα αλλά θα μπορούσε να έχει πρόσβαση να τα διαβάσει ως ιστορικό από τον ασθενή. Επομένως η ανάγκη εδώ είναι η προσθήκη ρόλων στους χρήστες του συστήματος.

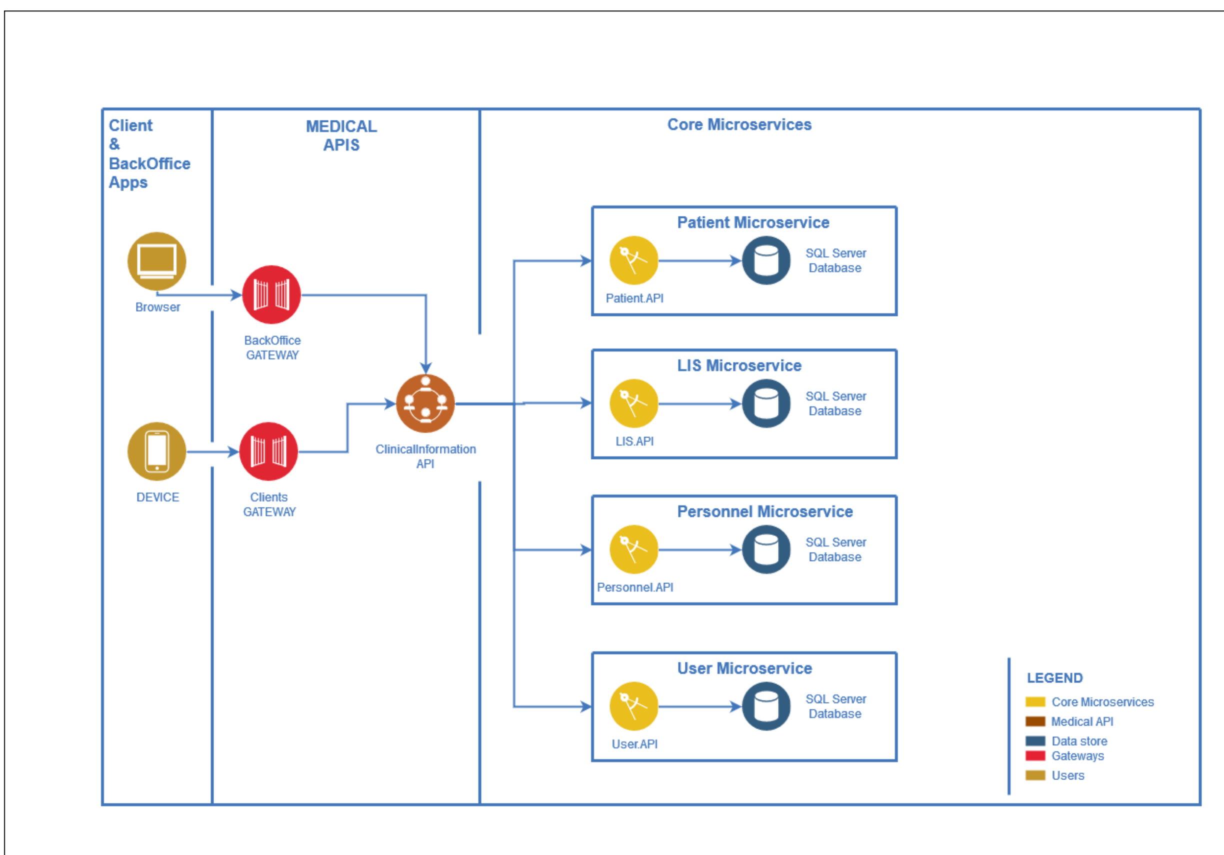
Έπειτα θα συνεχίσουμε με τις ανάγκες και τις λειτουργίες που θα πρέπει να διαθέτει ένα τέτοιο σύστημα σε επίπεδο ιατρικού προσωπικού. Σε αυτό το τμήμα το σύστημα θα πρέπει να μπορεί να διαβάζει, να διαχειρίζεται και να προσθέτει δεδομένα για ιατρικό προσωπικό του. Για παράδειγμα από τα στοιχεία ενός ιατρού τηλέφωνο, όνομα , ιδιότητα ιατρού , το νοσοκομείο στο οποίο ανήκει καθώς μπορεί να είναι ένα συγκρότημα νοσοκομείων αλλά και τα ραντεβού που τυχόν μπορεί να έχει ή την διαθεσιμότητα ενός ιατρού για να κλείσει κάποιος ένα ραντεβού.

Όπως καταλαβαίνετε το ίδιο μοτίβο θα πρέπει να ακολουθηθεί και για τους ασθενείς. Το σύστημα θα πρέπει να έχει την δυνατότητα να αποθηκεύσει τα στοιχεία ενός ασθενή, να προσθέσει κάποιο ιατρικό ιστορικό, φάρμακα που τυχόν παίρνει ο ασθενής και το είδος της επίσκεψης ενός ασθενή αν δηλαδή έγινε εισαγωγή στο νοσοκομείο ή έγινε απλή επίσκεψη σε κάποιο ιατρείο.

Τέλος μία από τις σημαντικότερες λειτουργίες του Medbook είναι η διαχείριση και η καταγραφή των εξετάσεων των ασθενών. Η συγκεκριμένη ανάγκη αποτελεί και την πιο βασική στο σύστημα καθώς είναι και ο λόγος για τον οποίο ένας ασθενής χρειάζεται τον ιατρικό αυτό φάκελο. Στο κομμάτι αυτό λοιπόν το σύστημα θα διαχωρίζει σε ποιο κομμάτι του νοσοκομείου ο ασθενής έκανε κάποιο τεστ (παθολογικό, ουρολογικό, μικροβιολογικό) θα πρέπει να μπορεί να προσθέτει την παραγγελία της εξέτασης και έπειτα να δέχεται και το αντίστοιχο αποτέλεσμα. Αφού λοιπόν αναφέρθηκαν κάποιες από τις βασικές λειτουργίες του συστήματος ήρθε η ώρα να προχωρήσουμε στην ανάλυση του κάθε τμήματος του ξεχωριστά.

4.3 Αρχιτεκτονική συστήματος

Όπως γνωρίζετε στόχος της συγκεκριμένης διπλωματικής είναι η δημιουργία ενός Ιατρικού συστήματος με την αρχιτεκτονική των microservices, πιο συγκεκριμένα στόχος της είναι ο σωστός διαχωρισμός των υποσυστημάτων και η σύνδεση μεταξύ αυτών. Εδώ λοιπόν θα γίνει μια συνοπτική παρουσίαση του συστήματος που σχεδιάστηκε για την δημιουργία της Ιατρικής εφαρμογής Medbook.



Εικόνα 10. Medbook Microservices

Όπως βλέπετε στην Εικόνα 10 το σύστημα αποτελείται από τέσσερα Core Microservices τα οποία είναι τα εξής User, Patient, Personnel και Lis microservice, κάθε ένα από αυτά τα microservice έχει την δικιά του βάση δεδομένων στην οποία έχει πρόσβαση μόνο αυτό. Για παράδειγμα θα μπορούσε κάθε ένα από αυτά να ζει στο δικό του container μαζί με την δικιά του βάση δεδομένων και για να μιλήσει κάποιος μαζί του θα το κάνει αποκλείστηκα μέσω του API που διαθέτει το συγκεκριμένο microservice (HTTP requests). Επίσης είναι σημαντικό να σημειωθεί πως αυτά τα microservices δεν είναι public στον έξω κόσμο αλλά μπορούν μόνο από εσωτερικά συστήματα. Μετά την δημιουργία των βασικών αυτών Core Microservice παρατηρήθηκε η ανάγκη για ένα microservice το οποίο δεν θα διαθέτει κάποια δικιά του βάση δεδομένων όπως φαίνεται και στην εικόνα, αλλά θα είναι υπεύθυνο για να επικοινωνεί με όλα τα Core Microservices να παίρνει τα απαραίτητα δεδομένα από το κάθε επιμέρους σύστημα και να τα συνδυάζει σε μία ενοποιημένη απάντηση. Το συγκεκριμένο microservice ονομάζεται ClinicalInformation και εκτός από την συνδυαστική επικοινωνία με τα επιμέρους microservice είναι υπεύθυνο για να διαχειριστή τα για το ποιος χρήστης έχει πρόσβαση σε συγκεκριμένα δεδομένα ή όχι.

Τέλος το ClinicalInformation microservice θα είναι ο διαμεσολαβητής μεταξύ του έξου κόσμου και των Core Microservices, πιο συγκεκριμένα θα τα δύο Gateways που βλέπουμε στην Εικόνα 3 θα είναι δύο API τα οποία θα είναι public ώστε να μπορούν οι εξωτερικές συσκευές των ασθενών αλλά και το προσωπικό από τα το εσωτερικό Website να χτυπάνε τα δύο Gateways αυτά με την σειρά τους αφού ελέγξουν τα δικαιώματα του χρήστη θα προωθούν τις κλήσεις στο ClinicalInformation microservice.

Πριν προχωρήσουμε στην περαιτέρω ανάλυση του συστήματος και του κάθε ενός microservice ξεχωριστά πρέπει να αναφερθούν κάποιες κοινές βιβλιοθήκες που χρησιμοποιήθηκαν σε όλα τα microservices και οι οποίες είναι:

- **Nlog**: Πρόκειται για μία δωρεάν πλατφόρμα καταγραφής μηνυμάτων(Logging) για .Net εφαρμογές. Μέσω του Nlog γίνεται εύκολη παραγωγή και διαχείριση αρχείων καταγραφής υψηλής ποιότητας για την εφαρμογή μας, ανεξάρτητα από το μέγεθος ή την πολυπλοκότητά της.
- **Swagger**: Είναι μία βιβλιοθήκη για .Net εφαρμογές η οποία προσφέρει έναν εύκολο τρόπο για την απεικόνιση των Http Requests & Controller που χρησιμοποιεί μία .Net Core εφαρμογή (API).Έτσι όταν γίνει η αρχικοποίηση του, το Swagger μας παρέχει μια διαδραστική σελίδα στην διεύθυνση της εφαρμογής μας η οποία μας επιτρέπει να δούμε αλλά και να δοκιμάσουμε τις διάφορες κλήσεις της εφαρμογής μας με τη χρήση οποιουδήποτε προγράμματος περιήγησης διαδικτύου (browser).
- **Autofac**: Το Autofac είναι ένα IoC container για .Net εφαρμογές. Διαχειρίζεται τις εξαρτήσεις μεταξύ των κλάσεων, έτσι ώστε οι εφαρμογές να αλλάζουν εύκολα καθώς μεγαλώνουν σε μέγεθος και πολυπλοκότητα.

Στη συνέχεια θα αναλύσουμε ένα προς ένα τα επιμέρους microservices του συστήματος αυτού, δίνοντας λεπτομέρειες για την υλοποίηση τους και την λογική την οποία περιέχουν. Θα αναφερθούμε στο Domain Layer και τα Entities που περιέχουν για κάθε ένα από αυτά, την βάση που χρησιμοποιούν και τα Http Requests τα οποία έχουν διαθέσιμα μέσω του API τους.

4.4 User Microservice

Στο κεφάλαιο αυτό θα μιλήσουμε για το User microservice για το ποιες είναι αναλυτικά οι ευθύνες του. Αυτό το microservice τις εξής βασικές λειτουργικότητες οι οποίες είναι, η ταυτοποίηση ενός χρήστη ο οποίος προσπαθεί να κάνει είσοδο στο Medbook είτε σαν ασθενής είτε ως προσωπικό του νοσοκομείου, η παροχή πληροφοριών για του χρήστες αυτούς όπως ονόματα, διευθύνσεις και άλλα. Η διαχείριση των στοιχείων των χρηστών όπως η αλλαγή κωδικού ή αλλαγή ονόματος, η εξακρίβωση των δικαιωμάτων και πιο συγκεκριμένα των ρόλων και των αδειών που διαθέτει ένας χρήστης και χρησιμεύουν στο να ξέρει το σύστημα ποιος έχει ή δεν έχει πρόσβαση σε συγκεκριμένα τμήματα του συστήματος (πχ ένας ασθενής δεν μπορεί να διαβάσει τον ιατρικό φάκελο άλλου ασθενή). Προφανώς τα δικαιώματα και οι ρόλοι είναι και αυτές δύο οντότητες οι οποίες μπορούν να διαχειριστούν όπως ο χρήστης. Δηλαδή μπορούμε να αφαιρέσουμε δικαιώματα από έναν ρόλο ή να προσθέσουμε νέα, επίσης μπορούμε να προσθέσουμε νέους ρόλους και να τους δώσουμε στους χρήστες.

Από την παραπάνω περιγραφή γίνεται κατανοητό πως το κύριο μοντέλο ασχολίας αυτού του microservice θα είναι ο User και η ταυτοποίηση εισόδου του στο σύστημα. Ας δούμε λοιπόν πιο συγκεκριμένα τις δυνατότητες και τις λειτουργίες του συγκεκριμένου συστήματος ξεκινώντας με την αρχιτεκτονική του, τον τρόπο υλοποίησης κάποιον βασικών λειτουργιών και των κλήσεων που διαθέτει.

4.4.1 Αρχιτεκτονική

Το User microservice είναι υλοποιημένο με C# και με την χρήση του framework .Net 5.0 που προσφέρει η Microsoft και αποτελείται από έξι βιβλιοθήκες(DLL) οι οποίες είναι Users.API, Users.Application, Users.Domain, Users.Common, Users.Infrastructure & Users.Shared.

Από αυτές τις έξι αυτή η οποία θα τρέξει με την βοήθεια ενός Docker ή μέσα στον IIS ενός Windows Server είναι το Users.API το οποίο αποτελεί το ASP.NET Core Application το οποίο περιέχει όλους τους Controllers του User microservice που ο καθένας περιέχει τα επιμέρους HTTP Requests που προσφέρει το συγκεκριμένο service.

Ένα βασικό στοιχείο στο συγκεκριμένο Api αποτελεί η κλάση **ExceptionsMiddleware** η οποία είναι ένας διαμεσολαβητής των εισερχόμενων Http Request και είναι υπεύθυνη να διαχειρίζεται τα εσωτερικά σφάλματα που ίσως προκύψουν από κάποιες κλήσεις ώστε είτε να γυρίσει κάποιο συγκεκριμένο μήνυμα ή HttpStatusCode (Unathorized, NotImplemented) και στο τέλος να καταγράψει το σφάλμα με την βοήθεια του Nlog(βιβλιοθήκη η οποία καταγράφη Logs) σε ένα αρχείο txt .

Έπειτα έρχεται το Users.Application, η συγκεκριμένη βιβλιοθήκη αποτελεί βασικό κομμάτι του microservice καθώς μέσα σε αυτό είναι υλοποιημένο το CQRS πρότυπο που αναφέραμε σε προηγούμενο κεφάλαιο. Πιο συγκεκριμένα το API περιέχει απλά τους Controllers οι οποίοι με την χρήση του Mediator ενεργοποιούν τους Command ή Query Handlers που βρίσκονται μέσα στο User.Application και είναι υπεύθυνοι για τον συντονισμό των διεργασιών που πρέπει να γίνουν σε επίπεδο Domain & Infrastructure. Παράδειγμα του παραπάνω σχεδιασμού και η υλοποίηση του φαίνεται στις εικόνες 11 & 12.

```
namespace User.API.Controllers
{
    [Authorize]
    [ApiController]
    [Route("api/[controller]")]
    1 reference | 0 changes | 0 authors, 0 changes
    public class UsersController : ControllerBase
    {
        private readonly IMediator _mediator;

        0 references | 0 changes | 0 authors, 0 changes
        public UsersController(IMediator mediator)
        {
            _mediator = mediator;
        }

        Get

        [AllowAnonymous]
        [HttpPost("RegisterUser")]
        [ProducesResponseType(typeof(UserModel), StatusCodes.Status200OK)]
        0 references | 0 changes | 0 authors, 0 changes
        public async Task<IActionResult> RegisterUser([FromBody] UserRegisterPayload payload)
        {
            UserModel response = await _mediator.Send(new RegisterUser(payload));
            return Ok(response);
        }
```

Εικόνα 11. Αποστολή του RegisterUser Command , απο το API στο Application με την χρήση του Mediator

```
[UnsafeCommandHandler]
1 reference | 0 changes | 0 authors, 0 changes
class RegisterUserHandler : CommandHandler<RegisterUser, UserModel>
{
    private readonly IUserRepository _userRepository;
    private readonly IPasswordService _passwordService;

    0 references | 0 changes | 0 authors, 0 changes
    public RegisterUserHandler(IDomainEventsHub eventsHub, IUserRepository userRepository, IPasswordService passwordService) : base(eventsHub)
    {
        _userRepository = userRepository;
        _passwordService = passwordService;
    }

    2 references | 0 changes | 0 authors, 0 changes
    protected override async Task<NotifiableResponse> HandleInternal(RegisterUser request, CancellationToken cancellationToken)
    {
        var payload = request.Payload;
        var addModification = Domain.Aggregates.User.Users.RegisterNew(
            username: payload.Username,
            password: _passwordService.Hash(payload.Password),
            email: payload.Email,
            firstName: payload.FirstName,
            lastName: payload.LastName,
            sex: payload.Sex,
            mobileNumber: payload.Phone,
            doctorId: payload.DoctorId,
            patientId: payload.PatientId);

        await _userRepository.RegisterUser(addModification);

        var registeredUser = new UserModel(
            userId: addModification.User.Id,
            username: payload.Username,
            email: payload.Email,
            firstName: payload.FirstName,
            lastName: payload.LastName,
            phone: payload.Phone,
            roles: Array.Empty<Role>(),
            doctorId: payload.DoctorId,
            patientId: payload.PatientId);

        return new NotifiableResponse(registeredUser, addModification.DomainEvents);
    }
}
```

Εικόνα 12. RegisterUserHandler, εδώ εκτελείται η υλοποίηση του Command ο οποίος δημιουργεί το Domain Entity και το σώζει στην βάση μέσω του IUserRepository

Άλλο ένα σημαντικό κομμάτι υλοποιήσεις για το οποίο είναι υπεύθυνο το συγκεκριμένο microservice θα είναι ο έλεγχος των στοιχείων ενός χρήστη, κάτι το οποίο αποτελεί μία πολύ σημαντική λειτουργία καθώς είναι μέρος της ασφάλειας του συστήματος.

Εδώ ακολουθήθηκε ένα ευρέως γνωστό πρότυπο που είναι υπεύθυνο για την αυθεντικοποίηση ενός χρήστη, το **JWT(Json Web Token) Authentication**. Το JWT ορίζει έναν συμπαγή και αυτόνομο τρόπο για την ασφαλή μετάδοση δεδομένων μεταξύ δύο ενδιαφερόμενων, ως ένα αντικείμενο JSON. Αυτές οι πληροφορίες μπορούν να επαληθευτούν και να είναι έμπιστες επειδή είναι ψηφιακά υπογεγραμμένες. Τα JWT μπορούν να υπογραφούν χρησιμοποιώντας ένα μυστικό κλειδί (με τον αλγόριθμο HMAC) ή ένα ζεύγος δημόσιου/ιδιωτικού κλειδιού χρησιμοποιώντας RSA ή ECDSA. Έτσι λοιπόν ακολουθώντας το συγκεκριμένο πρότυπο δημιουργήθηκε μία κλάση που ονομάστηκε **TokenProviderService** και είναι υπεύθυνη να δημιουργήσει αυτό το μοναδικό κλειδί. Στην περίπτωση μας χρησιμοποιήθηκε ο αλγόριθμος **HmacSha56** και παράγει ένα κλειδί το οποίο είναι έγκυρο για δέκα λεπτά (αυτό είναι παραμετρικό και μπορεί να αλλάξει). Το κλειδί αυτό θα περιέχει το μοναδικό Id του χρήστη , το όνομα, το επώνυμο, το email και τέλος τα δικαιώματα τα οποία διαθέτει εκείνη την συγκεκριμένη στιγμή. Να προστεθεί πως σχεδόν όλα τα κομμάτια πλοήγησης στην εφαρμογή χρειάζονται αυτό το Token αλλιώς θα συμβαίνει ένα UnAuthorized σφάλμα.

Το τελευταίο αλλά εξίσου σημαντικό κομμάτι στο Application DLL είναι ο τρόπος που θα αποθηκεύεται ένας κωδικός στην βάση και στο πως θα ελέγχεται ώστε να μην μπορεί κάποιος να επιχειρήσει εύκολα μία επίθεση στο σύστημα μας με την οποία θα μπορούσε να κλέψει προσωπικά δεδομένα ασθενών.

Για την συγκεκριμένη υλοποίηση φτιάξαμε την κλάση **PasswordService** η οποία ακολουθεί την την προσέγγιση **"Hash and Salt"** . Τα παλαιότερα χρόνια για την αποθήκευση ενός κωδικού συνήθως κατακερμάτιζαν(Hashing) τον κωδικό αυτό με κάποιον αλγόριθμο(MD5, SHA) μετατρέποντας έτσι ένα απλό κείμενο σε ένα κρυπτογραφημένο κείμενο (ciphertext) το οποίο έκρυβε μέσα του τον αρχικό κωδικό και το έσωζαν στην βάση.

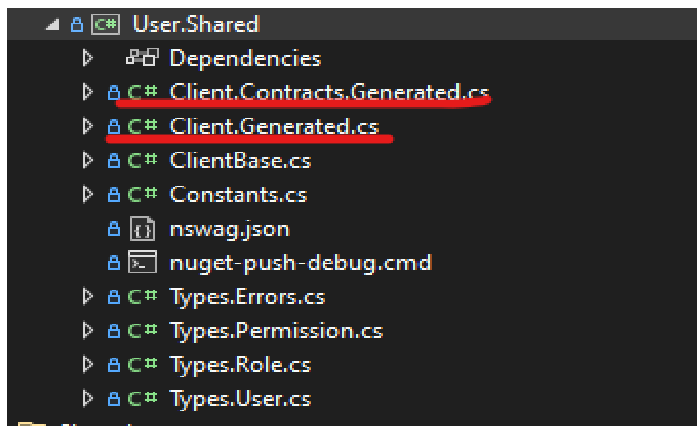
Προφανώς ο κατακερματισμός δεν μπορεί να αντιστραφεί οπότε για να γίνει η ταυτοποίηση ενός χρήστη, κατακερματίζανε και πάλι τον κωδικό και βλέπανε αν ταιριάζει με αυτόν που έχει σωθεί εξαρχής στην βάση. Πλέον αυτή η μέθοδος δεν θεωρείται η βέλτιστη για αυτό δημιουργήθηκε το αλάτι. Το Salting όπως και ονομάζεται κάνει το εξής, δημιουργεί ένα τυχαίο κείμενο για κάθε έναν μοναδικό χρήστη έπειτα συνδυάζει αυτό το κείμενο με τον κωδικό και έπειτα εκτελείται ο κατακερματισμός του συνδυαστικού κειμένου. Όπως καταλαβαίνεται για να γίνει η ταυτοποίηση του κωδικού δεν φτάνει μόνο το ciphertext αλλά πρέπει κάθε φορά να κρατάμε και το αλάτι του συγκεκριμένου χρήστη. Στην παρακάτω εικόνα φαίνεται η υλοποίησης του **“Hash and Salt”** μέσα στο User microservice.

```
class PasswordService : IPasswordService
{
    3 references | 0 changes | 0 authors, 0 changes
    public Password Hash(string password)
    {
        if (password == null)
            throw new ArgumentNullException("password");

        string saltStr;
        string passwordHash;

        Random rnd = new Random();
        var saltBytes = new byte[16];
        rnd.NextBytes(saltBytes);
        saltStr = Convert.ToBase64String(saltBytes);

        // Get derived bytes from the combined salt and password, using the specified number of iterations.
        using (var derivedBytes = new Rfc2898DeriveBytes(password, saltBytes, 1000))
        {
            var hashBytes = derivedBytes.GetBytes(32);
            passwordHash = Convert.ToBase64String(hashBytes);
        }

        return new Password(passwordHash: passwordHash, passwordSalt: saltStr);
    }

    3 references | 0 changes | 0 authors, 0 changes
    public bool Validate(string password, string salt, string hash)
    {
        if (password == null)
            throw new ArgumentNullException("password");

        if (salt == null)
            throw new ArgumentNullException("salt");

        if (hash == null)
            throw new ArgumentNullException("hash");

        var saltBytes = Convert.FromBase64String(salt);
        var hashLength = Convert.FromBase64String(hash).Length;
        using (var derivedBytes = new Rfc2898DeriveBytes(password, saltBytes, 1000))
        {
            var hashBytes = derivedBytes.GetBytes(hashLength);
            var newHash = Convert.ToBase64String(hashBytes);
            return newHash == hash;
        }
    }
```

Στην συνέχεια έχουμε το Users.Domain το οποίο περιέχει την λογική της επιχείρησης σε μοντέλα όπως Aggregates & Entities. Το συγκεκριμένο κομμάτι όμως θα αναφερθεί αναλυτικότερα λίγο παρακάτω. Οπότε σειρά έχει το Infrastructure, σε αυτό το κομμάτι του microservice όπως και σε όλα τα microservice αυτού του συστήματος υπάρχει η βιβλιοθήκη Entity Framework Core η οποία μας βοηθάει να μετατρέψουμε τα μοντέλα του Domain σε πίνακες βάσεων δεδομένων στην SQL.

Επομένως εδώ υπάρχει το Database Configuration της βάσης αλλά και κλάσεις οι οποίες είναι υπεύθυνες για την εκτέλεση ερωτήσεων, εισαγωγή εγγραφών και ενημέρωση υπάρχων δεδομένων στην βάση.

Τέλος το Users.Shared είναι το DLL που περιέχει όλα τα “Contracts” τα οποία θα γυρνάει το Users.API μέσω των απαντήσεων του στα HTTP Requests. Επίσης μέσα σε αυτό το DLL θα υπάρχει ένα cmd αρχείο το οποίο θα τρέχει όταν θέλουμε να παράγουμε μία νέα έκδοση της βιβλιοθήκης του User.Shared και των κλήσεων του, οι οποίες θα πρέπει να χρησιμοποιηθούν από κάποιο άλλο microservice. Πιο συγκεκριμένα με την βοήθεια της βιβλιοθήκης Swagger γίνεται η παραγωγή δύο αρχείων όπως φαίνεται και στην Εικόνα13 τα οποία θα γίνουν export σε ένα nuget package το οποίο θα είναι διαθέσιμο να χρησιμοποιηθεί από άλλα microservices. Με πιο απλά λόγια φέρνει τα μοντέλα και τα Http Requests του συγκεκριμένου microservice διαθέσιμα προς χρήση μέσα σε ένα άλλο (δίαυλος επικοινωνίας) .

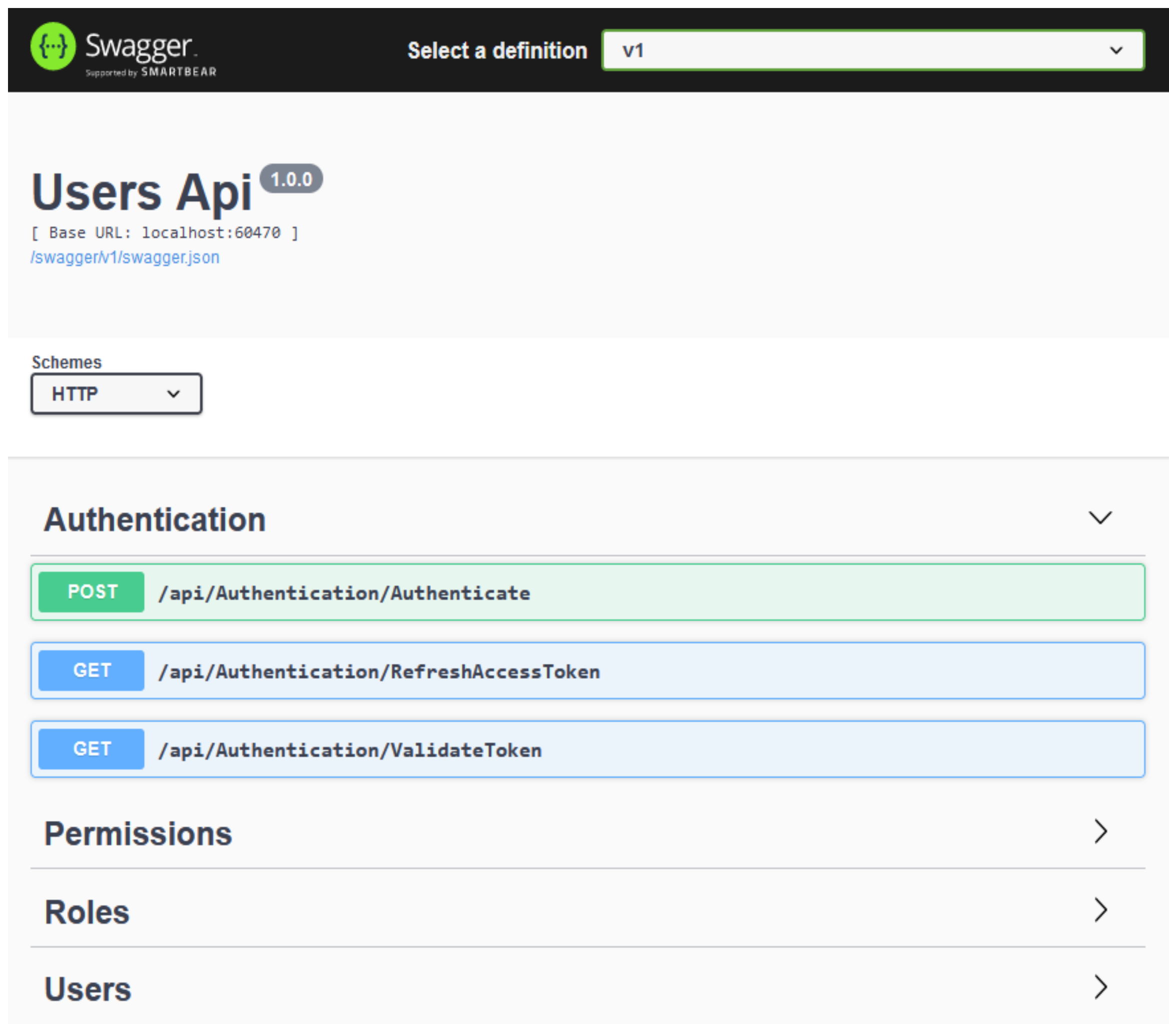


Εικόνα 13. Αρχεία παραγωγής από το Swagger, που περιέχουν τις κλήσεις και τα μοντέλα του API

4.4.2 Λειτουργίες και API Requests

Για την πιο εύκολη παρουσίαση των λειτουργιών και των Http Requests που διαθέτει το συγκεκριμένο microservice έχει γίνει η χρήση της βιβλιοθήκη Swagger όπως αναφέρθηκε και προηγουμένως.

Όπως φαίνεται και στην Εικόνα 14 παρακάτω μπορούμε να διακρίνουμε τους διαθέσιμους Controllers του συγκεκριμένου microservice οι οποίοι είναι οι Authentication, Permissions, Roles, Users. Τώρα αν ανοίξουμε το μενού σε κάθε έναν από αυτούς τους Controllers μπορούμε να δούμε τις συγκεκριμένες HTTP κλήσεις τις οποίες διαχειρίζεται και δρομολογούνται μέσω του εκάστοτε Controller.



Εικόνα 14. Απεικόνιση διεπαφής Users.Api μέσω της βιβλιοθήκης Swagger.

Authentication Controller

Ξεκινώντας θα εξηγήσουμε τις κλήσεις που διαθέτει ο Authentication Controller. Οι κλήσεις που δρομολογούνται μέσω του συγκεκριμένου αντικειμένου είναι οι εξής:

api/Authentication/Authenticate (POST): Η συγκεκριμένη κλήση δέχεται ως παραμέτρους το AuthenticatePayload το οποίο περιέχει το username & password τα οποία θα έχει εισάγει ένας χρήστης. αν αυθεντικοποίηση πετύχει τότε η συγκεκριμένη κλήση θα επιστρέψει Status 200 μαζί με ένα «AccessToken» που αναφέραμε προηγουμένως.

api/Authentication/RefreshAccessToken (POST): Η κλήση αυτή δέχεται ένα ήδη ενεργό Access Token και επιστρέφει ένα καινούργιο με νέα ημερομηνία, αυτή η κλήση χρειάζεται σε περιπτώσης ανανέωσης του Token όταν λήγει ο χρόνος χρήσης του και ώστε να μην χρειαστεί να κάνει ο χρήστης εκ νέου είσοδο.

Permissions Controller

Εδώ θα αναφερθούν οι κλήσεις του Permissions Controller, οι κλήσεις της συγκεκριμένης κλάσης είναι οι εξής:

api/Permissions/GetAll (GET): Η κλήση επιστρέφει όλα τα permissions που υπάρχουν στο σύστημα γυρνώντας το Id, Name, Description

/**api/Permissions/AddPermission (POST):** Η κλήση δέχεται ένα PermissionInfoPayload για να προσθέσει ένα νέο Permision.

/api/Permissions/UpdatePermission (GET): Η κλήση δέχεται ως παράμετρο ένα PermissionId για το οποίο θα ανανεώσει τα πεδία του Sysname, Name, Description.

api/Permissions/DeletePermissions (Delete): Η κλήση δέχεται ως παράμετρο ένα PermissionId ώστε να εκτελέσει την διαγραφή του.

Roles Controller

Εδώ θα αναφερθούν οι κλήσεις του Roles Controller, οι κλήσεις της συγκεκριμένης κλάσης είναι οι εξής:

api/Roles/GetAll (GET): Η κλήση επιστρέφει όλα τους ρόλους που υπάρχουν στο σύστημα γυρνώντας το Id, Name, Description

/**api/ Roles /GetRolePermissions (GET):** Η κλήση δέχεται ένα RoleId και επιστρέφει όλα τα Permissions του συγκεκριμένου ρόλου.

/api/ Roles /AddRole (POST): Η κλήση δέχεται ως παράμετρο ένα RolePayload για το οποίο θα προσθέσει έναν νέο ρόλο χωρίς Permissions.

api/ Roles /AddRolePermissions (POST): Η κλήση δέχεται ως παράμετρο ένα RoleID και μία λίστα από PermissionPayload ώστε να προσθέσει συγκεκριμένα permissions σε έναν ρόλο.

api/ Roles /RemoveRolePermissions (POST): Η κλήση δέχεται ως παράμετρο ένα RoleID και μία λίστα από PermissionPayload ώστε να αφαιρέσει συγκεκριμένα permissions από έναν ρόλο

/**api/ Roles /UpdateRole (POST):** Η κλήση δέχεται ως παράμετρο ένα RoleID και ένα RolePayload ώστε να ανανεώσει τα πεδία του συγκεκριμένου ρόλου.

api/ Roles /DeleteRoles (Delete): Η κλήση δέχεται ως παράμετρο μία λίστα με RoleID ώστε να εκτελέσει την διαγραφή τους.

Users Controller

Εδώ θα αναφερθούν οι κλήσεις του User Controller, οι κλήσεις της συγκεκριμένης κλάσης είναι οι εξής:

api/Users/GetAll (GET): Η κλήση επιστρέφει όλα τους χρήστες που υπάρχουν στο σύστημα.

/api/Users/GetUser (GET): Η κλήση δέχεται ένα UserId και επιστρέφει τα δεδομένα του συγκεκριμένου χρήστη.

/api/ Users /GetUserRoles (GET): Η κλήση δέχεται ως παράμετρο ένα UserId ώστε να επιστρέψει τους ρόλους του χρήστη αυτού.

api/ Users /SearchUser (POST): Η κλήση δέχεται ως παράμετρο ένα SearchUserPayload, το οποίο θα ψάξει να βρει χρήστες που τηρούν τα συγκεκριμένα κριτήρια που περιέχει το Payload.

api/ Users /RegisterUser (POST): Η κλήση δέχεται ως παράμετρο το UserRegisterPayload και θα πάει να δημιουργήσει έναν νέο χρήστη με αυτά τα στοιχεία και θα επιστρέψει το Id του.

/**api/ Users /AddUserRoles (POST):** Η κλήση δέχεται ως παράμετρο ένα UserId και μία λίστα από UserRolePayload ώστε να προσθέσει τους ρόλους αυτούς στον συγκεκριμένο χρήστη.

/**api/ Users /RemoveUserRoles (POST):** Η κλήση δέχεται ως παράμετρο ένα UserId και μία λίστα από UserRolePayload ώστε να αφαιρέσει τους ρόλους αυτούς από τον συγκεκριμένο χρήστη.

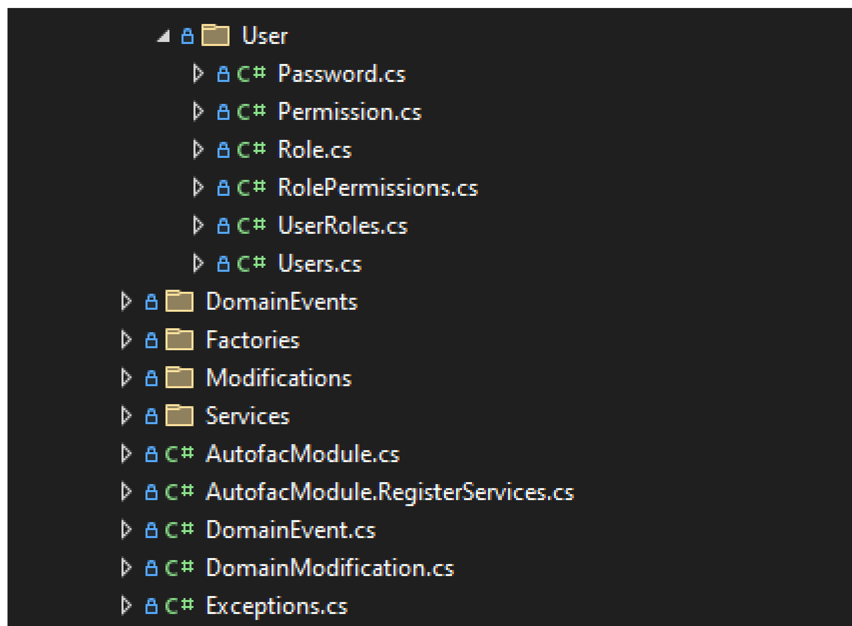
api/ Users /UpdatePassword (POST): Η κλήση δέχεται ως παράμετρο το UserId και τον παλαιό και καινούργιο κωδικό ώστε να ανανεώσει τον κωδικό του χρήστη.

api/ Users /UpdateUserInfo (POST): Η κλήση δέχεται ως ένα UserId και το UpdateUserInfoPayload ώστε να ανανεώσει κάποια από τα στοιχεία του χρήστη.

4.4.3 Σχεδιασμός Domain επιπέδου

Στο σημείο αυτό ήρθε η ώρα να γίνει μία μικρή επεξήγηση για το Domain επίπεδο του User microservice, δηλαδή για το ποια πράγματα μέσα σε αυτό αποτελούν Aggregates, Entities και ποια Value Objects και γενικότερα για το σχεδιασμό αυτού του επιπέδου.

Βλέποντας την εικόνα 15 καταλαβαίνουμε πως αυτό ο τομέας έχει συνολικά έξι οντότητες. Ας ξεκινήσουμε λέγοντας τα Aggregates του domain στο User microservice, τα οποία είναι τα Users, Role και Permission. Έπειτα έχουμε τρία Value Object τα Password, RolePermissions & UserRoles , επομένως εδώ δεν υπάρχουν καθόλου Entities.



Εικόνα 15. Το Domain Επίπεδο και οι κλάσεις του

Ξεκινώντας από τις τρεις βασικές οντότητες σε αυτόν τον τομέα, καλό είναι να αναφερθεί πως μέσα από αυτές και μόνο αυτές θα εκτελούνται αλλαγές ώστε να αποθηκευτούν στην βάση. Το Permission είναι το πιο απλό από όλα καθώς δεν διαθέτει πολλές και περίπλοκες μεθόδους παρά μόνο ένα Update, οπότε δεν χρειάζεται να πούμε κάτι περισσότερο.

Στην συνέχεια έχουμε τον χρήστη (User), ο οποίος ίσως και η πιο βασική οντότητα, όπως φαίνεται και στην παρακάτω εικόνα (16) ο χρήστης περιέχει μέσα του δύο Value Object το Password και μία λίστα με UserRoles. Η απόφαση πως αυτές οι δύο οντότητες έπρεπε να είναι Value Object πάρθηκε όταν κατανοήθηκε πως δεν γίνεται να υπάρχουν χωρίς τον User. Δεν έχει νόημα να κρατάς τα στοιχεία κωδικών και τους ρόλους ενός χρήστη χωρίς να υπάρχει ο χρήστης ταυτόχρονα όμως είναι πληροφορίες οι οποίες έπρεπε να υπάρχουν σε έναν διαφορετικό πίνακα για αυτούς τους λόγους λοιπόν έγιναν διαφορετικές οντότητες, το ίδιο ισχύει και για το Role Aggregate και τα RolePermissions.

Επομένως όταν ένας χρήστης για παράδειγμα θέλει να προσθέσει έναν ρόλο ή να αλλάξει κωδικό θα πρέπει να γεμίσουμε το User Aggregate με τα στοιχεία του συγκεκριμένου χρήστη και να καλέσουμε αντίστοιχα τις μεθόδους AddRoles ή UpdatePassword. Όπως βλέπεται και στην εικόνα ο μόνος τρόπος να αλλάξεις κάποια ιδιότητα της κλάσης αυτής είναι μέσω των μεθόδων της και όχι αλλιώς, αφού λοιπόν γίνει αυτό για να διαχωριστεί ξεκάθαρα το επίπεδο του Domain & Infrastructure δημιουργήσαμε κλάσεις που ονομάσαμε Modifications (πχ UpdateUserPasswordModification) κάθε μία από αυτές τις κλάσεις αντιπροσωπεύει μία συγκεκριμένη αλλαγή στην βάση και περιέχει τον χρήστη για τον οποίο γίνεται η αλλαγή αλλά και τα νέα δεδομένα που θα σωθούν στη βάση.

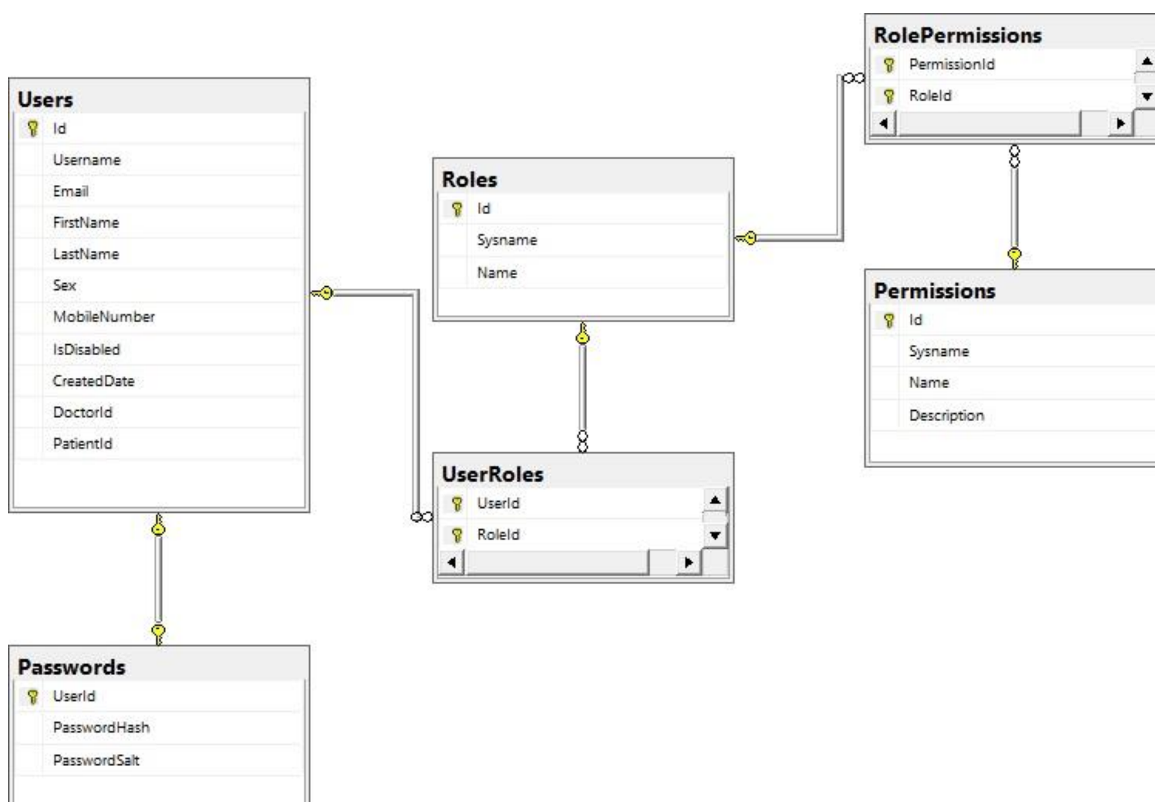
Έπειτα το συγκεκριμένο Modification που θα παραχθεί θα σταλθεί σε ένα Repository το οποίο ζει στο Infrastructure επίπεδο και εκεί θα γίνει Persist στην βάση , διαχωρίζοντας έτσι τελείως αυτά τα δύο επίπεδα του microservice.

```
[Serializable]
46 references | 0 changes | 0 authors, 0 changes
public class Users : RootEntity<long>
{
    private readonly List<UserRoles> _roles;

    7 references | 0 changes | 0 authors, 0 changes
    public string Username { get; private set; }
    9 references | 0 changes | 0 authors, 0 changes
    public Password Password { get; private set; }
    8 references | 0 changes | 0 authors, 0 changes
    public string Email { get; private set; }
    9 references | 0 changes | 0 authors, 0 changes
    public string FirstName { get; private set; }
    9 references | 0 changes | 0 authors, 0 changes
    public string LastName { get; private set; }
    3 references | 0 changes | 0 authors, 0 changes
    public string Sex { get; private set; }
    4 references | 0 changes | 0 authors, 0 changes
    public string MobileNumber { get; private set; }
    2 references | 0 changes | 0 authors, 0 changes
    public bool? IsDisabled { get; private set; }
    2 references | 0 changes | 0 authors, 0 changes
    public DateTime CreatedDate { get; private set; }
    3 references | 0 changes | 0 authors, 0 changes
    public long? DoctorId { get; private set; }
    3 references | 0 changes | 0 authors, 0 changes
    public long? PatientId { get; private set; }
    10 references | 0 changes | 0 authors, 0 changes
    public IReadOnlyCollection<UserRoles> Roles => _roles.AsReadOnly();

    0 references | 0 changes | 0 authors, 0 changes
    private Users() ...

    1 reference | 0 changes | 0 authors, 0 changes
    public Users(long id, UserRoles[] roles, string username, Password password,

    1 reference | 0 changes | 0 authors, 0 changes
    public static RegisterUserModification RegisterNew(string username, Password
        string mobileNumber, long? doctorId, long? patientId)...

    1 reference | 0 changes | 0 authors, 0 changes
    public UpdateUserInfoModification UpdateUserInfo(string username, string ema

    1 reference | 0 changes | 0 authors, 0 changes
    public UpdateUserPasswordModification UpdatePassword(Password password)...

    1 reference | 0 changes | 0 authors, 0 changes
    public UpdateUserRolesModification AddRoles(UserRoles[] roles)...

    1 reference | 0 changes | 0 authors, 0 changes
    public UpdateUserRolesModification RemoveRoles(UserRoles[] roles)...
```

Εικόνα 16. Οντότητα User

4.4.4 Βάση Δεδομένων

Για την αποθήκευση των δεδομένων του το συγκεκριμένο microservice χρησιμοποίησε την σχεσιακή βάση MS SQL η οποία είναι πρόσβαση μόνο από αυτό το microservice και από κανένα άλλο μέσα στο σύστημα Medbook. Στην Εικόνα φαίνονται οι πίνακες που δημιουργήθηκαν για το microservice αυτό αλλά και οι σχέσεις που υπάρχουν μεταξύ τους.



Εικόνα 17. Σχήμα βάσης δεδομένων User Microservice

Εδώ ας δώσουμε μία μικρή επεξήγηση για την βάση δεδομένων που βλέπουμε από πάνω. Ο πίνακας Users περιέχει όλους τους χρήστες οι οποίοι έχουν εγγραφή στο σύστημα μας και έχει μερικά από τα βασικά στοιχεία τους όπως φαίνεται και στην παραπάνω εικόνα. Έπειτα δύο πίνακες συνδέονται με τον πίνακα Users οι οποίοι έχουν ως Foreign Key το UserId. Αυτοί οι πίνακες είναι ο Passwords και ο UserRoles, ο πρώτος αποθηκεύει το γνωστό “Salt & Hash” τα οποία πρακτικά είναι ο κωδικός του χρήστη κρυπτογραφημένα ενώ ο δεύτερος περιέχει τους ρόλους ενός χρήστη οι οποίοι μπορεί να είναι περισσότεροι από ένας.

Συνεχίζοντας υπάρχει ο πίνακας Roles ο οποίος περιέχει τους ρόλους των χρηστών και στην αρχικοποίηση του συστήματος περιέχει κάποιους βασικούς ρόλους όπως Admin, Doctor, Patient. Από αυτόν τον πίνακα εξαρτώνται δύο άλλοι ο UserRoles που ήδη αναφέραμε αλλά και ο RolePermissions ο οποίος είναι υπεύθυνος για την αποθήκευση των Permission σε κάθε ένα ρόλο όπου και εδώ κάθε ρόλος μπορεί να περιέχει πολλά Permissions.

Τέλος μένει ο πίνακας Permissions ο οποίος γεμίζει την πρώτη φορά που θα τρέξει η εφαρμογή και περιέχει όλα τα επιτρεπόμενα Permissions που μπορούν να δοθούν σε έναν ρόλο.

4.5 Patient Microservice

Στο κεφάλαιο αυτό θα μιλήσουμε για το Patient microservice για το ποιες είναι αναλυτικά οι ευθύνες του. Αυτό το microservice διαθέτει τις εξής λειτουργίες:

- Δημιουργία ενός νέου ασθενή
- Εύρεση ασθενών μέσω διαφόρων κριτηρίων όπως μοναδικό αριθμό μητρώου, όνομα, επίθετο, ημερομηνία γέννησης, ΑΜΚΑ, τηλέφωνο
- Ανανέωση στοιχείων ενός ασθενή όπως τηλέφωνο και διεύθυνση
- Καταχώρηση ιατρικών συνταγών για χορήγηση φαρμάκων, αποθηκεύοντας λεπτομέρειες όπως την περίοδο χορήγησης, την ποσότητα, το είδος του φαρμάκου και οδηγίες χορήγησης
- Αποθήκευση και δημιουργία ιατρικού ιστορικού του ασθενή, αποθηκεύοντας παλαιότερες διαγνώσεις που ίσως έχουν γίνει ή δημιουργώντας νέες
- Καταγραφή αφίξεων των ασθενών στο νοσοκομείο είτε είναι στα εξωτερικά ιατρεία είτε είναι για την εσωτερική νοσηλεία στο νοσοκομείο ώστε να υπάρχει ιστορικό επισκέψεων αλλά και για το σύνολο των ασθενών μέσα στο νοσοκομείο.
- Καταγραφή εξετάσεων όπως αξονικές, ακτινογραφίες και άλλες οι οποίες δεν χρειάζονται να περάσουν από κάποιο εργαστηριακό τμήμα όπως αιματολογικές, ουρολογικές και άλλες. Επίσης περιέχουν το στάδιο στο οποίο βρίσκονται και την γνωμάτευση η οποία έχει προκύψει από τον ιατρό

Από την παραπάνω περιγραφή γίνεται κατανοητό πως το κύριο μοντέλο ασχολίας αυτού του microservice θα είναι ο ασθενής και η δημιουργία ενός αναλυτικού φακέλου για αυτόν. Ας δούμε λοιπόν πιο συγκεκριμένα τις δυνατότητες και τις λειτουργίες του συγκεκριμένου συστήματος ξεκινώντας με την αρχιτεκτονική του, τον τρόπο υλοποίησης κάποιον βασικών λειτουργιών και των κλήσεων που διαθέτει.

4.5.1 Αρχιτεκτονική

Το Patient microservice είναι υλοποιημένο με C# και με την χρήση του framework .Net 5.0 που προσφέρει η Microsoft και αποτελείται από έξι βιβλιοθήκες(DLL) οι οποίες είναι Patient.API, Patient.Application, Patient.Domain, Patient.Common, Patient.Infrastructure & Patient.Shared.

Από αυτές τις έξι αυτή η οποία θα τρέξει με την βοήθεια ενός Docker ή μέσα στον IIS ενός Windows Server είναι το Patient.API το οποίο αποτελεί το ASP.NET Core Application το οποίο περιέχει όλους τους Controllers του User microservice που ο καθένας περιέχει τα επιμέρους HTTP Requests που προσφέρει το συγκεκριμένο service.

Ένα βασικό στοιχείο στο συγκεκριμένο Api αποτελεί η κλάση **ExceptionsMiddleware** η οποία είναι ένας διαμεσολαβητής των εισερχόμενων Http Request και είναι υπεύθυνη να διαχειρίζεται τα εσωτερικά σφάλματα που ίσως προκύψουν από κάποιες κλήσεις ώστε είτε να γυρίσει κάποιο συγκεκριμένο μήνυμα ή HttpStatusCode (Unathorized, NotImplemented) και στο τέλος να καταγράψει το σφάλμα με την βοήθεια του Nlog(βιβλιοθήκη η οποία καταγράφη Logs) σε ένα αρχείο txt .

Έπειτα έρχεται το Patient.Application, η συγκεκριμένη βιβλιοθήκη αποτελεί βασικό κομμάτι του microservice καθώς μέσα σε αυτό είναι υλοποιημένο το CQRS πρότυπο που αναφέραμε σε προηγούμενο κεφάλαιο. Πιο συγκεκριμένα το API περιέχει απλά τους Controllers οι οποίοι με την χρήση του Mediator ενεργοποιούν τους Command ή Query Handlers που βρίσκονται μέσα στο Patient.Application και είναι υπεύθυνοι για τον συντονισμό των διεργασιών που πρέπει να γίνουν σε επίπεδο Domain & Infrastructure και δώσαμε και παρόμοιο παράδειγμα στην προηγούμενη ενότητα.

Στην συνέχεια έχουμε το Patient.Domain το οποίο περιέχει την λογική της επιχείρησης σε μοντέλα όπως Aggregates & Entities. Το συγκεκριμένο κομμάτι όμως θα αναφερθεί αναλυτικότερα λίγο παρακάτω. Οπότε σειρά έχει το Infrastructure, σε αυτό το κομμάτι του microservice όπως και σε όλα τα microservice αυτού του συστήματος υπάρχει η βιβλιοθήκη Entity Framework Core η οποία μας βοηθάει να μετατρέψουμε τα μοντέλα του Domain σε πίνακες βάσεων δεδομένων στην SQL. Επομένως εδώ υπάρχει το Database Configuration της βάσης αλλά και κλάσεις οι οποίες είναι υπεύθυνες για την εκτέλεση ερωτήσεων, εισαγωγή εγγραφών και ενημέρωση υπάρχων δεδομένων στην βάση.

Τέλος το Patient.Shared είναι το DLL που περιέχει όλα τα “Contracts” τα οποία θα γυρνάει το Patient.API μέσω των απαντήσεων του στα HTTP Requests. Επίσης μέσα σε αυτό το DLL θα υπάρχει ένα cmd αρχείο το οποίο θα τρέχει όταν θέλουμε να παράγουμε μία νέα έκδοση της βιβλιοθήκης του Patient.Shared και των κλήσεων του, οι οποίες θα πρέπει να χρησιμοποιηθούν από κάποιο άλλο microservice. Πιο συγκεκριμένα με την βοήθεια της βιβλιοθήκης Swagger γίνεται η παραγωγή δύο αρχείων τα οποία θα γίνουν export σε ένα nuget package το οποίο θα είναι διαθέσιμο να χρησιμοποιηθεί από άλλα microservices. Με πιο απλά λόγια φέρνει τα μοντέλα και τα Http Requests του συγκεκριμένου microservice διαθέσιμα προς χρήση μέσα σε ένα άλλο (δίαυλος επικοινωνίας) .

4.5.2 Λειτουργίες και API Requests

Όπως και στο προηγούμενο microservice χρησιμοποιήσαμε και πάλι την βιβλιοθήκη swagger για την παρουσίαση των Controller & Endpoint του Patient microservice

Όπως φαίνεται και στην Εικόνα 18 παρακάτω μπορούμε να διακρίνουμε τους διαθέσιμους Controllers του συγκεκριμένου microservice οι οποίοι είναι οι Patients, Medicalrecords, Visits. Τώρα αν ανοίξουμε το μενού σε κάθε έναν από αυτούς τους Controllers μπορούμε να δούμε τις συγκεκριμένες HTTP κλήσεις τις οποίες διαχειρίζεται και δρομολογούνται μέσω του εκάστοτε Controller.

MedicalRecords

Patients

GET /api/Patients/GetPatient

POST /api/Patients/SearchPatients

POST /api/Patients/RegisterPatient

POST /api/Patients/UpdatePatientAddress

POST /api/Patients/UpdatePatientFullname

POST /api/Patients/UpdatePatientExtraInfo

Visits

Εικόνα 18. Απεικόνιση διεπαφής Patient.Api μέσω της βιβλιοθήκης Swagger.

Patients Controller

Ξεκινώντας θα εξηγήσουμε τις κλήσεις που διαθέτει ο Patients Controller. Οι κλήσεις που δρομολογούνται μέσω του συγκεκριμένου αντικειμένου είναι οι εξής:

/api/Patients/GetPatient (GET): Η κλήση δέχεται το Id ενός ασθενή και επιστρέφει τα στοιχεία του

/**api/ Patients /SearchPatients (POST):** Η κλήση δέχεται ένα SearchPatientPayload και αναλόγως των κριτηρίων που έχουν συμπληρωθεί στο Payload γυρνάει τους ασθενείς που τα ικανοποιούν

/api/ Patients /RegisterPatient (POST): Η κλήση προσθέτει έναν νέο ασθενή με τα βασικά του στοιχεία

api/ Patients /UpdatePatientAddress (POST): Η κλήση δέχεται ως παράμετρο ένα PatientId και ένα UpdatePatientAddressPayload και ανανεώνει την διεύθυνση του συγκεκριμένου ασθενή

api/ Patients /UpdatePatientExtraInfo (POST): Η κλήση δέχεται ως παράμετρο ένα PatientId και ένα UpdatePatientExtraInfoPayload και ανανεώνει τα έξτρα στοιχεία του συγκεκριμένου ασθενή

Visits Controller

Εδώ θα αναφερθούν οι κλήσεις του Visits Controller, οι κλήσεις της συγκεκριμένης κλάσης είναι οι εξής:

/api/Visits/RegisterVisit (POST): Η κλήση προσθέτει μία νέα επίσκεψη ενός ασθενή

/**api/ Visits /AddVisitExaminations (POST):** Η κλήση δέχεται ένα VisitId και μία λίστα από AddVisitExaminationsPayload για να προσθέσει εξετάσεις πάνω σε μία επίσκεψη ενός ασθενή

/api/ Visits /RemoveVisitExaminations (POST): Η κλήση δέχεται ένα VisitId και μία λίστα από AddVisitExaminationsPayload για να αφαιρέσει εξετάσεις από μία επίσκεψη ενός ασθενή

api/ Visits /UpdateExaminationStatus (POST): Η κλήση δέχεται ως παράμετρο ένα VisitId, ExaminationId και ένα κείμενο το οποίο αποτελεί το αποτλέσμα της συγκεκριμένης εξέτασης.

api/ Visits /UpdateExaminationDiagnosis (POST): Η κλήση δέχεται ως παράμετρο ένα VisitId, ExaminationId και ένα ExaminationStatus μίας εξέτασης και θα κάνει update το Status.

api/ Visits /GetPatientVisits (GET): Η κλήση δέχεται ως παράμετρο ένα PatientId και επιστρέφει όλες τις επισκέψεις του συγκεκριμένου ασθενή

MedicalRecords Controller

Εδώ θα αναφερθούν οι κλήσεις του Medicalrecords Controller, οι κλήσεις της συγκεκριμένης κλάσης είναι οι εξής:

/api/MedicalRecords/RegisterMedicalRecord (POST): Η κλήση δέχεται ένα RegisterMedicalRecordsPayload και αποθηκεύει το συγκεκριμένο MedicalRecord του ασθενή

/api/MedicalRecords/UpdateMedicalRecord (POST): Η κλήση δέχεται ένα MedicalRecordsUpdatePayload και ανανεώνει το MedicalRecord του ασθενή

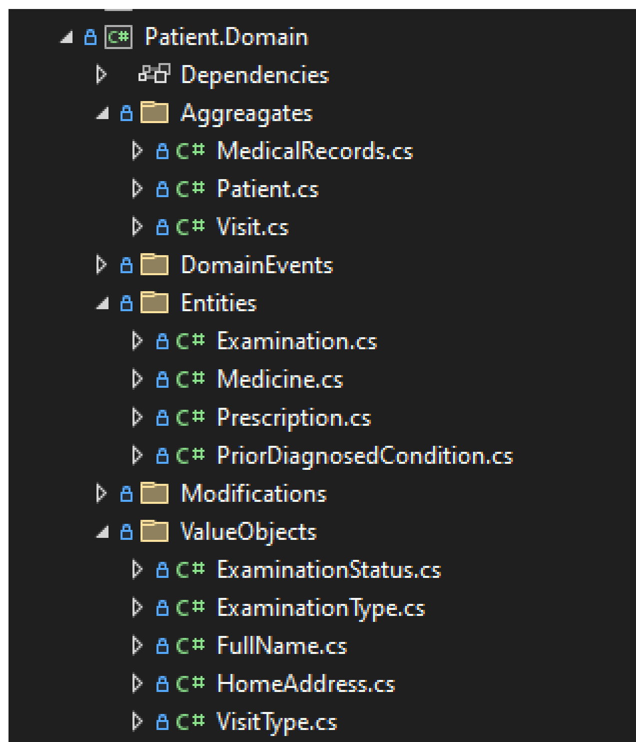
/api/ MedicalRecords /AddPrescription (POST): Η κλήση δέχεται ως παράμετρο ένα MedicalRecordsId και ένα PrescriptionPayload και προσθέτει τα συγκεκριμένα φάρμακα στον φακελο του ασθενή

api/ MedicalRecords /AddPriorDiagnosedConditions (POST): Η κλήση δέχεται ως παράμετρο ένα MedicalRecordsId και ένα AddPriorDiagnosedConditionsPayload και προσθέτει τις συγκεκριμένες γνωματεύσεις στον φάκελο του ασθενή

api/ MedicalRecords /GetPatientRecords (GET): Η κλήση δέχεται ως παράμετρο ένα PatientId και επιστρέφει το Medical Record του ασθενή

4.5.3 Σχεδιασμός Domain επιπέδου

Στο σημείο αυτό ήρθε η ώρα να γίνει μία μικρή επεξήγηση για το Domain επίπεδο του Patient microservice, δηλαδή για το ποια πράγματα μέσα σε αυτό αποτελούν Aggregates, Entities και ποια Value Objects και γενικότερα για το σχεδιασμό αυτού του επιπέδου.
Το συγκεκριμένο επίπεδο τομέα αποτελείται από τα εξής Aggregates, Patient, MedicalRecords και Visit. Αυτές οι τρεις οντότητες θα αποτελέσουν το κορμό της λογικής του συγκεκριμένου microservice, μιας και είναι ένα service που θα αναφέρεται κυρίως σε πληροφορίες που είναι γύρω από τον ασθενή τα στοιχεία του και γενικά τις εξετάσεις του. Έπειτα έχουμε τα Entities είναι τα εξής, Examination, Medicine, Perscription και PriorDiagnosedCondition και τα ValueObjects FullName, HomeAddress, VisiType, ExaminationStatus και ExaminationType. Όλα τα παραπάνω φαίνονται και στην εικόνα 19 από κάτω.



Εικόνα 19. Patient Microservice Domain

Η σκέψη για το συγκεκριμένο microservice ήταν πως χρειαζόμαστε ένα κομμάτι του συστήματος να είναι υπεύθυνο για τον ασθενή και μόνον αυτόν, και να διαθέτει τις βασικές πληροφορίες του. Έτσι λοιπόν δημιουργήθηκε ένα επίπεδο το οποίο θα περιέχει το Patient Aggregate το οποίο θα έχει commands όπως ανανέωση στοιχείων κυρίως.

```
1 reference | 0 changes | 0 authors, 0 changes
public static RegisterPatientModification RegisterNew(FullName fullName, HomeAddress homeAddress,
    long? medicalRecordsId, string email, string phone, DateTime birthDate,
    string gender, string hisId, string amka, string nationality,
    string insuranceProvider)[...]

1 reference | 0 changes | 0 authors, 0 changes
public UpdatePatientFullnameModification UpdatePatientFullname(string firstname, string lastname, string middlename, string fathername)[...]

1 reference | 0 changes | 0 authors, 0 changes
public UpdatePatientAddressModification UpdatePatientAddress(string country, string city, string street, string number, string zipCode)[...]

1 reference | 0 changes | 0 authors, 0 changes
public UpdatePatientExtraInfoModification UpdatePatientExtraInfo(long? medicalRecordsId, string email, string phone, DateTime birthDate,
    string gender, string hisId, string amka, string nationality, string insuranceProvider)[...]
```

Εικόνα 20. Patient μέθοδοι

Όμως δεν έφτανε μόνο αυτό για να γεμίσουμε την λειτουργικότητα που θέλαμε επομένως δημιουργήθηκαν δύο οντότητες που θα έχουν τις επισκέψεις και το ιατρικό φάκελο του ασθενή ο οποίος θα έχει φάρμακα που έχει πάρει, παλαιότερες ασθένειες που έχει διαγνωσθεί αλλά και αποτελέσματα από επισκέψεις σε ιατρεία του νοσοκομείου (Medicalrecords & Visit).

Όπως θα δείτε και στις παρακάτω εικόνες το MedicalRecords περιέχει μεθόδους για την προσθήκη φαρμάκων και προηγούμενων διαγνώσεων και την αφαίρεση τους φυσικά. Να αναφερθεί πως κάθε ασθενής αντιστοιχεί αυστηρά σε ένα MedicalRecord. Έπειτα έχουμε την οντότητα Visit η οποία περιέχει μεθόδους για την προσθήκη εξετάσεων από Ιατρούς και την ανανέωση του αποτελέσματος και της συνθήκης που βρίσκονται οι εξετάσεις αυτές.

```
1 reference | 0 changes | 0 authors, 0 changes
public UpdateMedicalRecordsModification UpdateMedicalRecords(string height, string weight, string bloodTypeSysname, string allergies, string condition,
    DateTime? dateOfDesease)[...]

1 reference | Panos Onasis, 18 days ago | 1 author, 1 change
public UpdatePrescriptionModification AddPrescriptions(Prescription[] prescriptions)[...]

0 references | Panos Onasis, 18 days ago | 1 author, 1 change
public UpdatePrescriptionModification RemovePrescriptions(Prescription[] prescriptions)[...]

1 reference | 0 changes | 0 authors, 0 changes
public UpdatePriorDiagnosedConditionsModification AddPriorDiagnosedConditions(PriorDiagnosedCondition[] priorDiagnosedConditions)[...]

0 references | 0 changes | 0 authors, 0 changes
public UpdatePriorDiagnosedConditionsModification RemovePriorDiagnosedConditions(PriorDiagnosedCondition[] priorDiagnosedConditions)[...]
```

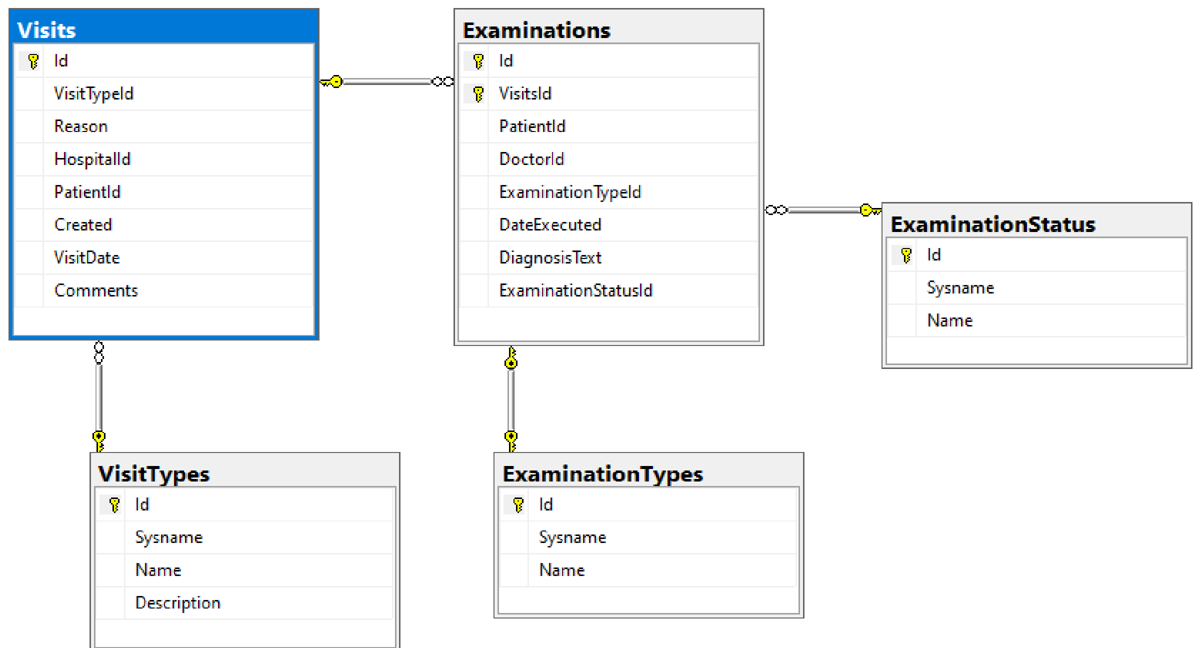
Εικόνα 21. MedicalRecord μέθοδοι

```
1 reference | 0 changes | 0 authors, 0 changes
public UpdateVisitExaminationsModification AddExaminations(Examination[] examinations)[...]

1 reference | 0 changes | 0 authors, 0 changes
public UpdateVisitExaminationsModification RemoveExaminations(Examination[] examinations)[...]

1 reference | 0 changes | 0 authors, 0 changes
public UpdateExaminationStatusModification UpdateExaminationStatus(long examinationIdForUpdate, ExaminationStatus newExaminationStatus)[...]

1 reference | 0 changes | 0 authors, 0 changes
public UpdateExaminationDiagnosisModification UpdateExaminationDiagnosis(long examinationIdForUpdate, string diagnosisText)[...]
```

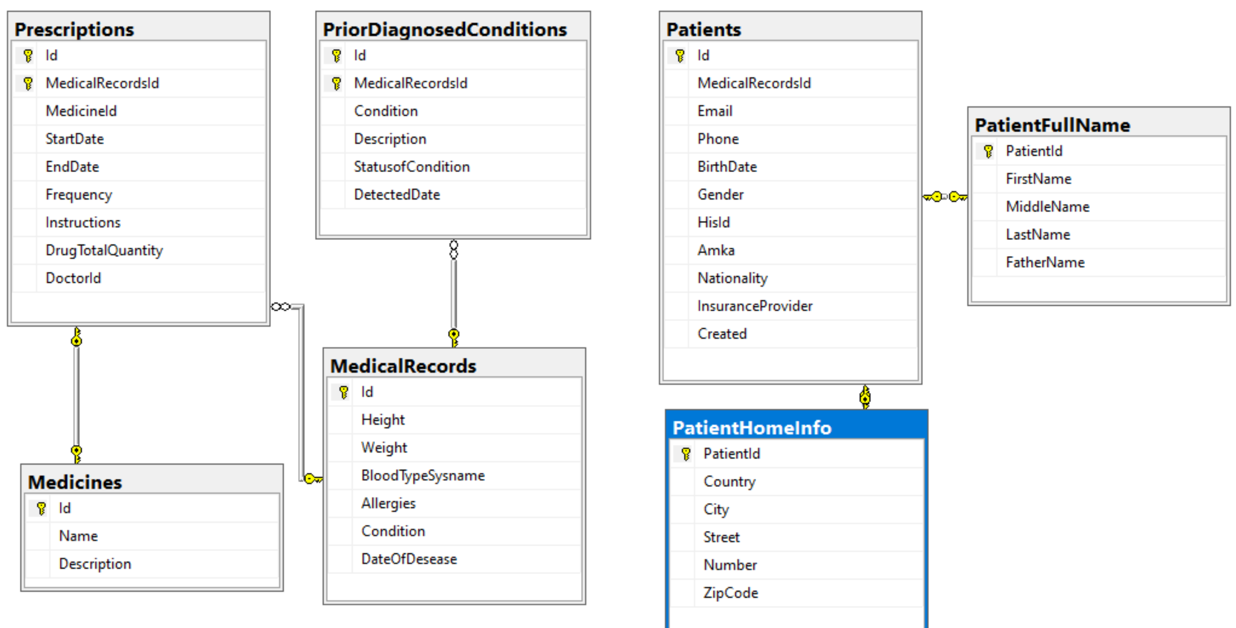
Εικόνα 22. Visit μεθόδοι

4.5.4 Βάση Δεδομένων

Για την αποθήκευση των δεδομένων του το συγκεκριμένο microservice χρησιμοποίησε την σχεσιακή βάση MS SQL στην οποία έχει πρόσβαση μόνο από αυτό το microservice και από κανένα άλλο μέσα στο σύστημα Medbook. Στις εικόνες από κάτω φαίνονται οι πίνακες της βάσης Patient που δημιουργήθηκαν για το microservice αυτό αλλά και οι σχέσεις που υπάρχουν μεταξύ τους. Ας δώσουμε μία εξήγηση για τις παρακάτω εικόνες.



Στην εικόνα από πάνω διακρίνονται πέντε πίνακες όπου οι τέσσερις από αυτούς βασίζονται σε έναν τον πίνακα Examinations. Ο συγκεκριμένος πίνακας περιέχει τις εξετάσεις καταγράφοντας το Id του γιατρού που τις καταχώρισε, την ημερομηνία διεξαγωγής της εξέτασης, ένα κείμενο με την απάντηση της εξέτασης, το Id του ασθενή, τον τύπο της εξέτασης που όπως βλέπουμε θα είναι και το Foreign Key με τον πίνακα ExaminationStatus πρακτικά ο πίνακας αυτός περιέχει όλα τα είδη των εξετάσεων από τα οποία ένας γιατρός μπορεί να επιλέξει. Άλλο ένα Id που περιέχει ο πίνακας Examinations είναι το VisitId το οποίο είναι Foreign Key με τον πίνακα και αναφέρετε στον τύπο της επίσκεψης που έκανε ο ασθενής όταν καταγράφηκε η συγκεκριμένη εξέταση, δηλαδή αν κάποιος είναι εξωτερικός ή εσωτερικός ασθενής. Τέλος ο πίνακας ExaminationStatus περιέχει όλα τα επιτρεπόμενα στάδια από τα οποία μπορεί να περάσει μία εξέταση όπως Pending, Approved, Released, Cancelled.



Οι τελευταίοι πίνακες και σχέσεις της βάσης Patient φαίνονται στην παραπάνω εικόνα. Εδώ διακρίνουμε δύο βασικούς πίνακες τον Patients και MedicalRecords, οι δύο αυτοί πίνακες πάιζουν πολύ σημαντικό ρόλο σε αυτό το microservice. O Patient είναι αυτός που κατέχει όλη την πληροφορία για έναν ασθενή και για να μην υπάρχουν πολλές στήλες στον πίνακα τον σπάσαμε και στους PatientHomeInfo & PatientFullName, ο πρώτος έχει τα στοιχεία της κατοικίας του ασθενή και ο δεύτερος τα ονομαστικά στοιχεία του. Κάποια βασικά στοιχεία όπως Ids, τηλέφωνο, ημερομηνία γέννησης παραμένουν στον κεντρικό πίνακα Patients.

Στην συνέχεια έχουμε το MedicalRecords πίνακα ο οποίος περιέχει κάποια βασικά στοιχεία του ασθενή όπως ύψος, βάρος, ομάδα αίματος. Έπειτα τον πίνακα Prescriptions ο οποίος περιέχει τις συνταγογραφήσεις των φαρμάκων ενός ασθενή περιέχοντας την ποσότητα, ποιο φάρμακο είναι (MedicineId Foreign Key με τον πίνακα Medicine), οδηγίες, περίοδο χρήσης του φαρμάκου. Τέλος υπάρχει ο πίνακας PriorDiagnosedConditions ο οποίος έχει παλαιότερες παθήσεις τις οποίες είχε η έχει διαγνωσθεί ένας ασθενής.

4.6 Personnel Microservice

Στο κεφάλαιο αυτό θα μιλήσουμε για το Personnel microservice για το ποιες είναι αναλυτικά οι ευθύνες του. Αυτό το microservice διαθέτει τις εξής λειτουργίες:

- Δημιουργία ενός νέου ιατρού με όλα τα στοιχεία του όπως ονοματεπώνυμο, στοιχεία κατοικίας , στοιχεία του νοσοκομείου και ιατρείου και ειδικότητας
- Δημιουργία διαθεσιμότητας ραντεβού με ημερομηνίες και ώρες για την προσθήκη ραντεβού
- Δυνατότητα αλλαγής ειδικότητας ενός υπάρχοντα ιατρού
- Δυνατότητα αλλαγής διαθεσιμότητας ιατρού για ένα διάστημα
- Εύρεση ιατρών ή ιατρού που τηρεί συγκεκριμένα κριτήρια τα οποία θα συμπληρώσει ο ασθενής ή γραμματέας όπως όνομα, ειδικότητα ιατρού, νοσοκομείο στο οποίο ανήκει και συγκεκριμένο διάστημα διαθεσιμότητας
- Δημιουργία ενός νέου νοσοκομείου στο σύστημα καθώς κάθε ιατρός ανήκει σε συγκεκριμένο νοσοκομείο
- Εύρεση όλων των διαθέσιμων νοσοκομείων του συστήματος
- Προσθήκη ραντεβού σε έναν ιατρό από συγκεκριμένο ασθενή
- Αλλαγή ώρας συγκεκριμένου ραντεβού
- Ακύρωση ραντεβού ιατρού
- Εύρεση όλων των ραντεβού που έχει ένα συγκεκριμένος ιατρός

Όπως φαίνεται και από την παραπάνω περιγραφή των λειτουργιών, το συγκεκριμένο microservice ασχολείται με τον Ιατρό και γενικά το ιατρικό προσωπικό ενός νοσοκομείου και τις δυνατότητες που προσφέρει μέσα σε ένα ιατρικό σύστημα. Είναι υπεύθυνο για την διαχείριση όλων των ενεργειών που αφορούν έναν γιατρό, το νοσοκομείο αλλά και των ραντεβού που μπορεί να έχει με διάφορους ασθενείς. Ας δούμε λοιπόν πιο συγκεκριμένα τις δυνατότητες και τις λειτουργίες του συγκεκριμένου συστήματος ξεκινώντας με την αρχιτεκτονική του, τον τρόπο υλοποίησης κάποιον βασικών λειτουργιών και των κλήσεων που διαθέτει.

4.6.1 Αρχιτεκτονική

Το Personnel microservice είναι υλοποιημένο με C# και με την χρήση του framework .Net 5.0 που προσφέρει η Microsoft και αποτελείται από έξι βιβλιοθήκες(DLL) οι οποίες είναι Personnel.API, Personnel.Application, Personnel.Domain, Personnel.Common, Personnel.Infrastructure & Personnel.Shared.

Από αυτές τις έξι αυτή η οποία θα τρέξει με την βοήθεια ενός Docker ή μέσα στον IIS ενός Windows Server είναι το Personnel.API το οποίο αποτελεί το ASP.NET Core Application το οποίο περιέχει όλους τους Controllers του Personnel microservice που ο καθένας περιέχει τα επιμέρους HTTP Requests που προσφέρει το συγκεκριμένο service.

Ένα βασικό στοιχείο στο συγκεκριμένο Api αποτελεί η κλάση **ExceptionsMiddleware** η οποία είναι ένας διαμεσολαβητής των εισερχόμενων Http Request και είναι υπεύθυνη να διαχειρίζεται τα εσωτερικά σφάλματα που ίσως προκύψουν από κάποιες κλήσεις ώστε είτε να γυρίσει κάποιο συγκεκριμένο μήνυμα ή HttpStatusCode (Unathorized, NotImplemented) και στο τέλος να καταγράψει το σφάλμα με την βοήθεια του Nlog(βιβλιοθήκη η οποία καταγράφη Logs) σε ένα αρχείο txt .

Έπειτα έρχεται το Personnel.Application, η συγκεκριμένη βιβλιοθήκη αποτελεί βασικό κομμάτι του microservice καθώς μέσα σε αυτό είναι υλοποιημένο το CQRS πρότυπο που αναφέραμε σε προηγούμενο κεφάλαιο. Πιο συγκεκριμένα το API περιέχει απλά τους Controllers οι οποίοι με την χρήση του Mediator ενεργοποιούν τους Command ή Query Handlers που βρίσκονται μέσα στο Personnel.Application και είναι υπεύθυνοι για τον συντονισμό των διεργασιών που πρέπει να γίνουν σε επίπεδο Domain & Infrastructure και δώσαμε και παρόμοιο παράδειγμα στην προηγούμενη ενότητα.

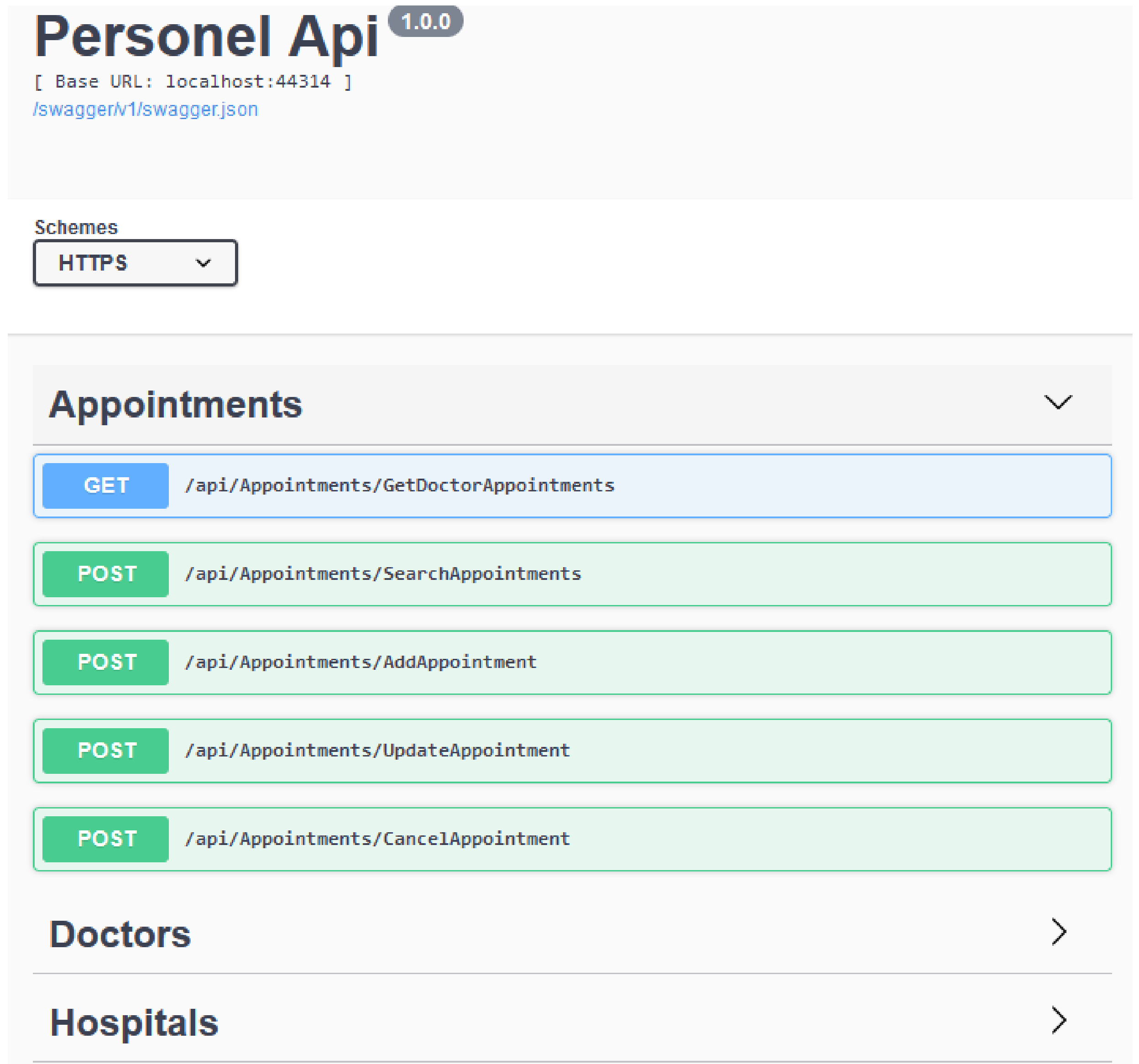
Στην συνέχεια έχουμε το Personnel.Domain το οποίο περιέχει την λογική της επιχείρησης σε μοντέλα όπως Aggregates & Entities. Το συγκεκριμένο κομμάτι όμως θα αναφερθεί αναλυτικότερα λίγο παρακάτω. Οπότε σειρά έχει το Infrastructure, σε αυτό το κομμάτι του microservice όπως και σε όλα τα microservice αυτού του συστήματος υπάρχει η βιβλιοθήκη Entity Framework Core η οποία μας βοηθάει να μετατρέψουμε τα μοντέλα του Domain σε πίνακες βάσεων δεδομένων στην SQL. Επομένως εδώ υπάρχει το Database Configuration της βάσης αλλά και κλάσεις οι οποίες είναι υπεύθυνες για την εκτέλεση ερωτήσεων, εισαγωγή εγγραφών και ενημέρωση υπάρχων δεδομένων στην βάση.

Τέλος το Personnel.Shared είναι το DLL που περιέχει όλα τα “Contracts” τα οποία θα γυρνάει το Personnel.API μέσω των απαντήσεων του στα HTTP Requests. Επίσης μέσα σε αυτό το DLL θα υπάρχει ένα cmd αρχείο το οποίο θα τρέχει όταν θέλουμε να παράγουμε μία νέα έκδοση της βιβλιοθήκης του Personnel.Shared και των κλήσεων του, οι οποίες θα πρέπει να χρησιμοποιηθούν από κάποιο άλλο microservice. Πιο συγκεκριμένα με την βοήθεια της βιβλιοθήκης Swagger γίνεται η παραγωγή δύο αρχείων τα οποία θα γίνουν export σε ένα nuget package το οποίο θα είναι διαθέσιμο να χρησιμοποιηθεί από άλλα microservices. Με πιο απλά λόγια φέρνει τα μοντέλα και τα Http Requests του συγκεκριμένου microservice διαθέσιμα προς χρήση μέσα σε ένα άλλο (δίαυλος επικοινωνίας).

4.6.2 Λειτουργίες και API Requests

Όπως και στο προηγούμενο microservice χρησιμοποιήσαμε και πάλι την βιβλιοθήκη swagger για την παρουσίαση των Controller & Endpoint του Personnel microservice

Όπως φαίνεται και στην Εικόνα 14 παρακάτω μπορούμε να διακρίνουμε τους διαθέσιμους Controllers του συγκεκριμένου microservice οι οποίοι είναι οι Doctors, Appointments, Hospitals. Τώρα αν ανοίξουμε το μενού σε κάθε έναν από αυτούς τους Controllers μπορούμε να δούμε τις συγκεκριμένες HTTP κλήσεις τις οποίες διαχειρίζεται και δρομολογούνται μέσω του εκάστοτε Controller.



Εικόνα 23. Απεικόνιση διεπαφής Personnel.Api μέσω της βιβλιοθήκης Swagger.

Hospitals Controller

Ξεκινώντας θα εξηγήσουμε τις κλήσεις που διαθέτει ο Hospitals Controller. Οι κλήσεις που δρομολογούνται μέσω του συγκεκριμένου αντικειμένου είναι οι εξής:

/api/Hospital/GetAll (GET): Η κλήση επιστρέφει όλα τα νοσοκομεία που υπάρχουν στο Medbook σύστημα μάζι με κάποιες πληροφορίες όπως όνομα, πόλη και οδό

/api/ Hospital /AddHospital (POST): Η κλήση δέχεται ένα HospitalPayload για την προσθήκη ενός νέου νοσοκομείου

/api/ Hospital /UpdateHospitalInfo (POST): Η κλήση δέχεται ένα HospitalId και ένα HospitalInfoPayload για την ανανέωση στοιχείων του συγκεκριμένου νοσοκομείου

Doctors Controller

Εδώ θα αναφερθούν οι κλήσεις του Doctors Controller, οι κλήσεις της συγκεκριμένης κλάσης είναι οι εξής:

/api/Doctors/GetAll (GET): Η κλήση επιστρέφει μία λίστα από DoctorModel το οποίο έχει πληροφορίες για τον εκάστοτε γιατρό, όπως όνομα, επίθετο τηλέφωνο.

/**api/ Doctors /Get (GET):** Η κλήση δέχεται ένα DoctorId και επιστρέφει ένα DoctorModel του συγκεκριμένου γιατρού.

/api/ Doctors /GetDoctorAvailability (GET): Η κλήση δέχεται ένα DoctorId και επιστρέφει μία λίστα από DoctorAvailabilityModel η οποία περιέχει τα ωράρια των επισκέψεων του συγκεκριμένου ιατρού και τις κλεισμένες ώρες.

api/ Doctors /RegisterDoctor (POST): Η κλήση δέχεται ως παράμετρο ένα RegisterDoctorPayload για την δημιουργία ενός νέου ιατρού στο σύστημα.

api/ Doctors /AddDoctorAvailability (POST): Η κλήση δέχεται ως παράμετρο ένα DoctorId,

api/ Doctors /UpdateAvailableHours (POST): Η κλήση δέχεται ως παράμετρο ένα AddAvailabilityPayload για την προσθήκη ενός νέου ωραρίου επισκέψεων για έναν συγκεκριμένο ιατρό.

api/ Doctors /UpdateAppointmentDuration (POST): Η κλήση δέχεται ως παράμετρο ένα VisitId, ExaminationId και ένα κείμενο το οποίο αποτελεί το αποτλέσμα της συγκεκριμένης εξέτασης.

api/ Doctors /UpdateHospitalInfo (POST): Η κλήση δέχεται ως παράμετρο ένα DoctorId και ένα UpdateDoctorHomeInfoPayload για την ανανέωση των στοιχείων του ιατρού στο νοσοκομείο.

api/ Doctors /UpdateHomeInfo (POST): Η κλήση δέχεται ως παράμετρο ένα DoctorId και ένα UpdateDoctorHomeInfoPayload για την ανανέωση των προσωπικών στοιχείων του ιατρού.

Appointments Controller

Εδώ θα αναφερθούν οι κλήσεις του Appointments Controller, οι κλήσεις της συγκεκριμένης κλάσης είναι οι εξής:

/api/Appointments/GetDoctorAppointments (GET): Η κλήση δέχεται ένα DoctorId και επιστρέφει όλα ραντεβού του ιατρού

/**api/ Appointments /AddAppointment (POST):** Η κλήση δέχεται ένα AddAppointmentPayload για την προσθήκη ενός νέου ραντεβού για έναν ασθενή.

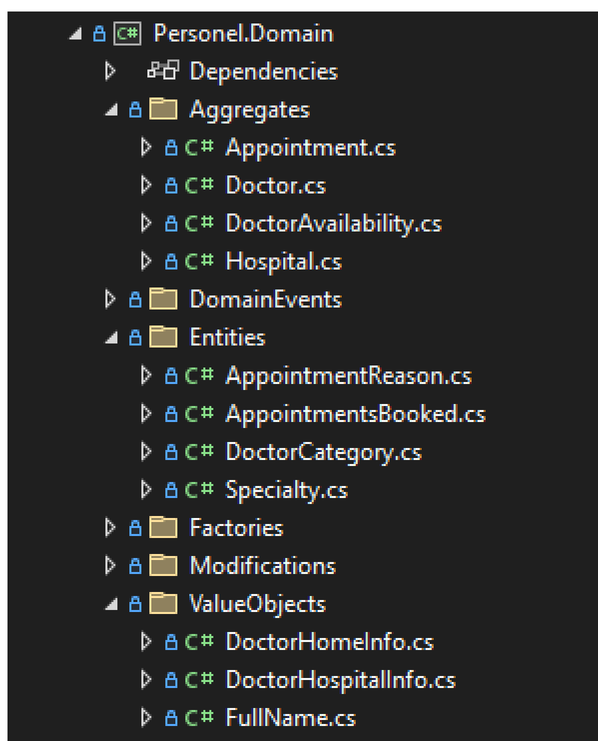
/api/ Appointments /SearchAppointment (POST): Η κλήση δέχεται ένα SearchAppointmentPayload και επιστρέφει όλα τα ραντεβού που βρέθηκαν με τα συγκεκριμένα κριτήρια.

api/ Appointments /CancelAppointment (POST): Η κλήση δέχεται ως παράμετρο AppointmentId και ακυρώνει το συγκεκριμένο ραντεβού του ασθενή.

4.6.3 Σχεδιασμός Domain επιπέδου

Στο σημείο αυτό ήρθε η ώρα να γίνει μία μικρή επεξήγηση για το Domain επίπεδο του Personnel microservice, δηλαδή για το ποια πράγματα μέσα σε αυτό αποτελούν Aggregates, Entities και ποια Value Objects και γενικότερα για το σχεδιασμό αυτού του επιπέδου.

Το συγκεκριμένο επίπεδο τομέα αποτελείται από τα εξής Aggregates, Hospital, Doctor, DoctorAvailability και Appointments. Αυτές οι τρεις οντότητες θα αποτελέσουν το κορμό της λογικής του συγκεκριμένου microservice, μιας και είναι ένα service που θα αναφέρεται κυρίως σε πληροφορίες που είναι γύρω από το νοσοκομείο και το ιατρικό προσωπικό του όπως είναι οι ιατροί και τα ραντεβού τους. Έπειτα έχουμε τα Entities είναι τα εξής, AppointmentReason, AppointmentBooked, DoctorCategory και Specialty και τα ValueObjects FullName, DoctorHomeInfo, και DoctorHospitalInfo. Όλα τα παραπάνω φαίνονται και στην εικόνα 24 από κάτω.



Εικόνα 24. Personnel Microservice Domain

Η σκέψη για το συγκεκριμένο microservice ήταν πως χρειαζόμαστε ένα κομμάτι του συστήματος να είναι υπεύθυνο για τον ιατρικό προσωπικό και κάποιες βασικές λειτουργίες του νοσοκομείου όπως τα ραντεβού και την διαθεσιμότητα ενός ιατρού. Έτσι λοιπόν δημιουργήθηκε ένα επίπεδο το οποίο θα περιέχει το Doctor Aggregate το οποίο θα έχει commands όπως ανανέωση στοιχείων κυρίως, αλλαγή νοσοκομειακών στοιχείων, αλλαγή ειδικότητας και αλλαγή βαθμίδας ιατρού (εκπαιδευόμενος, διευθυντής) τα οποία φαίνονται και στην παρακάτω εικόνα.

```
1 reference | Panos Onasis, 22 days ago | 1 author, 1 change
public static RegisterDoctorModification RegisterNew(FullName fullName, DoctorHomeInfo homeInfo, DoctorHospitalInfo hospitalInfo, DateTime dateOfBirth,
    string gender, string comments, Specialty specialty, DoctorCategory doctorCategory)[...]

1 reference | Panos Onasis, 22 days ago | 1 author, 1 change
public UpdateDoctorInfoModification UpdateFullname(FullName fullName)[...]

1 reference | Panos Onasis, 22 days ago | 1 author, 1 change
public UpdateDoctorInfoModification UpdateHomeInfo(DoctorHomeInfo doctorHomeInfo)[...]

1 reference | Panos Onasis, 22 days ago | 1 author, 1 change
public UpdateDoctorInfoModification UpdateHospitalInfo(DoctorHospitalInfo doctorHospitalInfo)[...]

1 reference | Panos Onasis, 22 days ago | 1 author, 1 change
public UpdateDoctorInfoModification UpdateDoctorCategory(DoctorCategory doctorCategory)[...]

0 references | Panos Onasis, 22 days ago | 1 author, 1 change
public UpdateDoctorInfoModification UpdateSepcialty(Specialty specialty)[...]
```

Στην συνέχεια έχουμε την οντότητα Appointment, το συγκεκριμένο Aggregate είναι αρκετά απλό καθώς περιέχει την δημιουργία ενός νέου ραντεβού για έναν ιατρό, την αλλαγή της ώρας και ημέρας και την ακύρωση του. Μέσα στο συγκεκριμένο μοντέλο υπάρχει και το AppointmentReason, το οποίο όπως αναφέραμε και πριν είναι ένας τύπος Entity στο Domain Layer του Personnel και περιγράφει τον λόγω του συγκεκριμένου ραντεβού δηλαδή (οφθαλμολογική επίσκεψη, γενικές εξετάσεις, ανάλυση ακτινογραφίας).

Έπειτα έχουμε το DoctorAvailability, το συγκεκριμένο Aggregate είναι υπεύθυνο για το ωράριο διαθεσιμότητας ενός ιατρού. Μέσω αυτού μπορούν να προστεθούν νέα ωράρια, να γίνουν αλλαγές στις ώρες ή στο διάστημα που κρατάει ένα ραντεβού και τέλος όταν προστίθεται ένα ραντεβού όπως αναφέραμε προηγουμένως θα πρέπει να καλεστεί η AddBookedAppointments του συγκεκριμένου μοντέλου ώστε να ενημερωθεί η διαθεσιμότητα του ιατρού. Παρατίθενται φωτογραφίες της υλοποίησης των δύο οντοτήτων παρακάτω.

```
public class Appointment : RootEntity<long>
{
    11 references | 0 changes | 0 authors, 0 changes
    public AppointmentReason AppointmentReason { get; private set;  }
    6 references | 0 changes | 0 authors, 0 changes
    public DateTime StartDateTime { get; private set; }
    5 references | 0 changes | 0 authors, 0 changes
    public DateTime? EndDateTime { get; private set; }
    4 references | 0 changes | 0 authors, 0 changes
    public long PatientId { get; }
    5 references | 0 changes | 0 authors, 0 changes
    public long DoctorId { get; }
    7 references | 0 changes | 0 authors, 0 changes
    public bool? IsCancelled { get; private set; }

    0 references | 0 changes | 0 authors, 0 changes
    private Appointment() [...]

    1 reference | 0 changes | 0 authors, 0 changes
    public Appointment(long id, AppointmentReason appointmentReason, DateTime startDateTime, DateTime? endDateTime,

    1 reference | 0 changes | 0 authors, 0 changes
    public static CreateAppointmentModification CreateNew(AppointmentReason appointmentReason, DateTime startDateTi
        long patientId, long doctorId)[...]

    1 reference | 0 changes | 0 authors, 0 changes
    public UpdateAppointmentModification Update(AppointmentReason appointmentReason, DateTime startDateTime, DateTi

    1 reference | 0 changes | 0 authors, 0 changes
    public CancelAppointmentModification Cancel()[...]
}
```

```
1 reference | 0 changes | 0 authors, 0 changes
public DoctorAvailability(long id,
    AppointmentsBooked[] booked,
    long doctorId,
    DateTime dateFrom,
    DateTime dateTo,
    DateTime availableHoursFrom,
    DateTime availableHoursTo,
    TimeSpan appointmentDuration)...

1 reference | 0 changes | 0 authors, 0 changes
public static CreateDoctorAvailabilityModification RegisterNew(long doctorId,
    DateTime dateFrom,
    DateTime dateTo,
    DateTime availableHoursFrom,
    DateTime availableHoursTo,
    TimeSpan appointmentDuration)...

1 reference | 0 changes | 0 authors, 0 changes
public UpdateAvailabilityModification UpdateAvalableHours(DateTime availableHoursFrom, DateTime availableHoursTo)...

1 reference | 0 changes | 0 authors, 0 changes
public UpdateAvailabilityModification UpdateDuration(TimeSpan appointmentDuration)...

1 reference | 0 changes | 0 authors, 0 changes
public UpdateAppointmentsBookedModification AddBookedAppointments(AppointmentsBooked[] appointmentsBooked)...

0 references | 0 changes | 0 authors, 0 changes
public UpdateAppointmentsBookedModification RemoveBookedAppointments(AppointmentsBooked[] appointmentsBooked)...
```

Τέλος έμεινε το Hospital Aggregate, όπως είναι αναμενόμενο το συγκεκριμένο αφορά το νοσοκομείο και περιέχει μεθόδους για την ανανέωση στοιχείων ενός νοσοκομείου και την προσθήκη ενός καινούργιου.

4.6.4 Βάση Δεδομένων

Για την αποθήκευση των δεδομένων του το συγκεκριμένο microservice χρησιμοποίησε την σχεσιακή βάση MS SQL στην οποία έχει πρόσβαση μόνο από αυτό το microservice και από κανένα άλλο μέσα στο σύστημα Medbook. Στις εικόνες από κάτω φαίνονται οι πίνακες της βάσης Personnel που δημιουργήθηκαν για το microservice αυτό αλλά και οι σχέσεις που υπάρχουν μεταξύ τους. Ας δώσουμε μία εξήγηση για τις παρακάτω εικόνες.

Στην παραπάνω εικόνα διακρίνουμε μερικούς από του πίνακες της βάσης του Personnel microservice. Οι συγκεκριμένοι πίνακες επικεντρώνονται και συνδέονται με τον πίνακα που αφορά τους ιατρούς (Doctor) και είναι οι DoctorHomeInfo, DoctorFullName, DoctorCategory, Specialy, DoctorHospitalInfo, Hospitals. Κατά την διάρκεια του σχεδιασμού της συγκεκριμένης βάσης έγινε κατανοητό πως ένας ιατρός αποτελείται από πολλά στοιχεία και πληροφορίες για αυτό έπρεπε να σχεδιαστεί και διαχωριστεί όσο πιο σωστά γίνεται σε επιμέρους πίνακες. Έτσι δημιουργήθηκε αρχικά ο πίνακας DoctosFullName έχει σχέση 1-1 με τον πίνακα Doctors και αποθηκεύει το ονοματεπώνυμο ενός ιατρού, αντίστοιχα ο πίνακας DoctorsHomeInfo και DoctorsHospitalInfo περιέχουν δεδομένα για την οικεία και στοιχεία επικοινωνίας για το γραφείου του ιατρού στο νοσοκομείο. Τέλος έχουμε τους

πίνακες DoctorCategory και Specialty όπου είναι πίνακες που απλά περιέχουν τις ιεραρχικές θέσεις που έχει το νοσοκομείο και τις ειδικότητες των ιατρών αντίστοιχα. Οι συγκεκριμένοι πίνακες γεμίζουν με δεδομένα κατά την εκτέλεση του συστήματος την πρώτη φορά και δεν ανανεώνονται έπειτα. Ο πίνακας του ιατρού μετά απλά κρατάει τα Ids των συγκεκριμένων πινάκων ώστε να ξέρουμε σε ποια κατηγορία.

Τέλος υπάρχουν άλλοι τέσσερις πίνακες όπως βλέπουμε και στην εικόνα από πάνω οι οποίοι είναι οι DoctorAvailabilty , AppointementsBooked, Appointment και AppointmentReason. Οι πρώτοι δύο είναι υπεύθυνοι για την αποθήκευση των διαθέσιμων ωρών και ημερών για κλείσιμο ραντεβού για έναν ιατρό και αν αυτό επιβεβαιωθεί τότε αποθηκεύεται στον AppointementsBooked. Επίσης αποθηκεύεται και στον Appoinment πίνακα ώστε να υπάρχει μία συγκεντρωτική εικόνα με όλα τα ραντεβού του νοσοκομείου. Ο AppointmentReason πίνακας διαθέτει όλους τους λόγους για τους οποίους κάποιος ιατρός μπορεί να κλείσει ένα ραντεβού όπως γενική εξέταση, οφθαλμολογική εξέταση και γεμίζει κατά την αρχικοποίησή της εφαρμογής.

4.7 Lis Microservice

Στο κεφάλαιο αυτό θα μιλήσουμε για το Lis microservice για το ποιες είναι αναλυτικά οι ευθύνες του. Πριν αναλύσουμε την υλοποίηση και την τις λειτουργίες του συγκεκριμένου microservice στο σύστημα Medbook ας δώσουμε μία γενική περιγραφή για το τι σημαίνει η λέξη Lis για ένα ιατρικό σύστημα. Το Lis αποτελεί ένα σύστημα πληροφοριών εργαστηρίου (Laboratory Information System), είναι μία λύση λογισμικού υγειονομικής περίθαλψης που επεξεργάζεται, αποθηκεύει και διαχειρίζεται δεδομένα ασθενών που σχετίζονται με εργαστηριακές διαδικασίες και δοκιμές. Οι πάροχοι και οι επαγγελματίες του εργαστηρίου χρησιμοποιούν συστήματα πληροφοριών εργαστηρίου για να συντονίσουν τη ροή εργασίας και τον ποιοτικό έλεγχο των ιατρικών δοκιμών εσωτερικού και εξωτερικού, συμπεριλαμβανομένης της αιματολογίας, της χημείας, της ανοσολογίας, της μικροβιολογίας, της τοξικολογίας, της δημόσιας υγείας και άλλων εργαστηριακών τομέων. Τα συστήματα πληροφοριών εργαστηρίου παρακολουθούν, αποθηκεύουν και ενημερώνουν κλινικές λεπτομέρειες σχετικά με έναν ασθενή κατά τη διάρκεια μιας επίσκεψης και αποθηκεύουν τις πληροφορίες στη βάση δεδομένων του για μελλοντική αναφορά.

Αφού αναλύσαμε το τι σημαίνει ένα σύστημα Lis για τον έξω κόσμο ας δούμε τις λειτουργίες και τις δυνατότητες που έχει στο σύστημα Medbook:

- Δημιουργία ιατρικού εργαστηρίου όπως Βιοχημικό εργαστήριο, Μικροβιολογικό, Αιματολογικό, Ανοσολογικό και άλλα.
- Προσθήκη εξετάσεων σε συγκεκριμένο εργαστήριο οι οποίες θα χρησιμοποιηθούν για προσθήκη παραγγελιών.
- Προσθήκη διευθυντή ιατρού σε ένα εργαστήριο.
- Εμφάνιση όλων των εξετάσεων που διαθέτει ένα εργαστήριο.
- Προσθήκη μίας νέας παραγγελίας η οποία αποτελείται από ένα δείγμα (αίμα, ιστός) πάνω στην οποία μπορεί να έχουν περαστεί παραπάνω από μία εξετάσεις προς εκτέλεση.
- Ανανέωση κατάστασης μίας παραγγελίας εξετάσεων, οι πιθανές καταστάσεις είναι Pending, Released, Cancelled, Approved.
- Αναζήτηση παραγγελιών ενός συγκεκριμένου ασθενή ή αναζήτηση παραγγελιών με βάση κριτήριων όπως ημερομηνία παραγγελίας, κατάσταση παραγγελίας, είδος εξετάσεων που περιέχει.
- Προσθήκη αποτελεσμάτων μίας παραγγελίας, ανανέωση αποτελεσμάτων για την καταγραφή της τελικής γνωμάτευσης ή αποτελέσματος της εξέτασης.
- Αναζήτηση αποτελεσμάτων ασθενών με βάση κριτηρίων όπως αναφέρθηκε και πριν ή με βάση το κωδικό συγκεκριμένου ασθενή.

Όπως ορίστηκε και στην εισαγωγή για την περιγραφή του συστήματος LIS αλλά και μετά την αναφορά των βασικών λειτουργιών στο σύστημα Medbook, πρόκειται για ένα πολύ σημαντικό microservice καθώς θα είναι υπεύθυνο για την αποθήκευση και την διαχείριση εργαστηριακών εξετάσεων και αποτελεσμάτων για τους ασθενής ενός νοσοκομείου. Ας δούμε λοιπόν πιο συγκεκριμένα τις δυνατότητες και τις λειτουργίες του συγκεκριμένου συστήματος ξεκινώντας με την αρχιτεκτονική του, τον τρόπο υλοποίησης κάποιον βασικών λειτουργιών και των κλήσεων που διαθέτει.

4.7.1 Αρχιτεκτονική

Το Lis microservice είναι υλοποιημένο με C# και με την χρήση του framework .Net 5.0 που προσφέρει η Microsoft και αποτελείται από έξι βιβλιοθήκες(DLL) οι οποίες είναι Lis.API, Lis.Application, Lis.Domain, Lis.Common, Lis.Infrastructure & Lis.Shared.

Από αυτές τις έξι αυτή η οποία θα τρέξει με την βοήθεια ενός Docker ή μέσα στον IIS ενός Windows Server είναι το Lis.API το οποίο αποτελεί το ASP.NET Core Application το οποίο περιέχει όλους τους Controllers του Lis microservice που ο καθένας περιέχει τα επιμέρους HTTP Requests που προσφέρει το συγκεκριμένο service.

Ένα βασικό στοιχείο στο συγκεκριμένο Api αποτελεί η κλάση **ExceptionsMiddleware** η οποία είναι ένας διαμεσολαβητής των εισερχόμενων Http Request και είναι υπεύθυνη να διαχειρίζεται τα εσωτερικά σφάλματα που ίσως προκύψουν από κάποιες κλήσεις ώστε είτε να γυρίσει κάποιο συγκεκριμένο μήνυμα ή HttpStatusCode (Unathorized, NotImplemented) και στο τέλος να καταγράψει το σφάλμα με την βοήθεια του Nlog(βιβλιοθήκη η οποία καταγράφη Logs) σε ένα αρχείο txt .

Έπειτα έρχεται το Personnel.Application, η συγκεκριμένη βιβλιοθήκη αποτελεί βασικό κομμάτι του microservice καθώς μέσα σε αυτό είναι υλοποιημένο το CQRS πρότυπο που αναφέραμε σε προηγούμενο κεφάλαιο. Πιο συγκεκριμένα το API περιέχει απλά τους Controllers οι οποίοι με την χρήση του Mediator ενεργοποιούν τους Command ή Query Handlers που βρίσκονται μέσα στο Personnel.Application και είναι υπεύθυνοι για τον συντονισμό των διεργασιών που πρέπει να γίνουν σε επίπεδο Domain & Infrastructure και δώσαμε και παρόμοιο παράδειγμα στην προηγούμενη ενότητα.

Στην συνέχεια έχουμε το Lis.Domain το οποίο περιέχει την λογική της επιχείρησης σε μοντέλα όπως Aggregates & Entities. Το συγκεκριμένο κομμάτι όμως θα αναφερθεί αναλυτικότερα λίγο παρακάτω. Οπότε σειρά έχει το Infrastructure, σε αυτό το κομμάτι του microservice όπως και σε όλα τα microservice αυτού του συστήματος υπάρχει η βιβλιοθήκη Entity Framework Core η οποία μας βοηθάει να μετατρέψουμε τα μοντέλα του Domain σε πίνακες βάσεων δεδομένων στην SQL. Επομένως εδώ υπάρχει το Database Configuration της βάσης αλλά και κλάσεις οι οποίες είναι υπεύθυνες για την εκτέλεση ερωτήσεων, εισαγωγή εγγραφών και ενημέρωση υπάρχων δεδομένων στην βάση.

Τέλος το Lis.Shared είναι το DLL που περιέχει όλα τα "Contracts" τα οποία θα γυρνάει το Lis.API μέσω των απαντήσεων του στα HTTP Requests. Επίσης μέσα σε αυτό το DLL θα υπάρχει ένα cmd αρχείο το οποίο θα τρέχει όταν θέλουμε να παράγουμε μία νέα έκδοση της βιβλιοθήκης του Lis.Shared και των κλήσεων του, οι οποίες θα πρέπει να χρησιμοποιηθούν από κάποιο άλλο microservice. Πιο συγκεκριμένα με την βοήθεια της βιβλιοθήκης Swagger γίνεται η παραγωγή δύο αρχείων τα οποία θα γίνουν export σε ένα nuget package το οποίο θα είναι διαθέσιμο να χρησιμοποιηθεί από άλλα microservices. Με πιο απλά λόγια φέρνει τα μοντέλα και τα Http Requests του συγκεκριμένου microservice διαθέσιμα προς χρήση μέσα σε ένα άλλο (δίαυλος επικοινωνίας).

4.7.2 Λειτουργίες και API Requests

Όπως και στο προηγούμενο microservice χρησιμοποιήσαμε και πάλι την βιβλιοθήκη swagger για την παρουσίαση των Controller & Endpoint του Lis microservice

Όπως φαίνεται και στην Εικόνα 25 παρακάτω μπορούμε να διακρίνουμε τους διαθέσιμους Controllers του συγκεκριμένου microservice οι οποίοι είναι οι Labsection, Order, Result. Τώρα αν ανοίξουμε το μενού σε κάθε έναν από αυτούς τους Controllers μπορούμε να δούμε τις συγκεκριμένες HTTP κλήσεις τις οποίες διαχειρίζεται και δρομολογούνται μέσω του εκάστοτε Controller.

Lis Api 1.0.0

[Base URL: localhost:44304]
/swagger/v1/swagger.json

Schemes
HTTPS

LabSection

Order

Result

GET /api/Result/GetResultsByOrderId

GET /api/Result/GetResultsByPatientId

POST /api/Result/SearchResult

POST /api/Result/AddResult

POST /api/Result/UpdateResult

Εικόνα 25. Απεικόνιση διεπαφής Lis.Api μέσω της βιβλιοθήκης Swagger.

Labsections Controller

Εδώ θα αναφερθούν οι κλήσεις του Labsections Controller, οι κλήσεις της συγκεκριμένης κλάσης είναι οι εξής:

/api/Labsections/GetAll (GET): Η κλήση επιστρέφει μία λίστα από LabsectionModel η οποία περιέχει τα δεδομένα για όλα τα εργαστήρια του συστήματος και τις εξετάσεις που περιέχουν.

/**api/ Labsections /GetAllTests (GET):** Η κλήση επιστρέφει μία λίστα από TestModel η οποία έχει όλα τα αποθηκευμένα τεστ.

/api/ Labsections /GetLabsectionById (GET): Η κλήση δέχεται ένα LabsectionId και επιστρέφει το συγκεκριμένο εργαστήριο μαζί με τις εξετάσεις που περιέχει.

api/ Labsections /AddLasection (POST): Η κλήση δέχεται ένα AddLabsectionPayload το οποίο προσθέτει το εργαστήριο στο σύστημα.

api/ Labsections /AddLabsectionsTests (POST): Η κλήση δέχεται ένα LabsectionId και μία λίστα από TestIds τα οποία τα συνδέει στο συγκεκριμένο εργαστήριο.

api/ Labsections /UpdateLabsectionDoctor (POST): Η κλήση δέχεται ένα LabsectionId και ένα DoctorId για να ανανεώσει τον υπεύθυνο ιατρό στο συγκεκριμένο εργαστήριο.

Orders Controller

Εδώ θα αναφερθούν οι κλήσεις του Orders Controller, οι κλήσεις της συγκεκριμένης κλάσης είναι οι εξής:

/api/ Orders/GetOrdersById (GET): Η κλήση δέχεται ένα OrderId και επιστρέφει ένα OrderModel το οποίο περιέχει πληροφορίες για την συγκεκριμένη παραγγελία όπως PatientId, SampleDate, DoctorId, VisitId, SampleType.

/**api/ Orders/GetOrdersByPatientId (GET):** Η κλήση δέχεται ένα PatientId και επιστρέφει μία λίστα από OrderModel τα όποια είναι όλες οι παραγγελίες για τον συγκεκριμένο ασθενή.

/api/ Orders/SearchOrders (GET): Η κλήση δέχεται ένα OrderSearchPayload το οποίο περιέχει κριτήρια για την εύρεση παραγγελίων που τα τηρούν όπως DoctorId, PatientId, TestId, SampleDaterange.

/api/ Orders/SearchOrdersTests (GET): Η κλήση δέχεται ένα ένα OrderTestSearchPayload το οποίο περιέχει κριτήρια για την εύρεση OrderTest τα οποία είναι εξετάσεις που αφορούν συγκεκριμένη παραγγελία.

api/ Orders/AddOrder (POST): Η κλήση δέχεται ως παράμετρο ένα AddOrderPayload για την δημιουργία μίας νέας παραγγελίας στο σύστημα Lis

api/ Orders/AddOrderTests (POST): Η κλήση δέχεται ως παράμετρο ένα AddOrderTestPayload για την δημιουργία ενός OrderTest στο σύστημα Lis

api/ Orders/UpdateOrder (POST): Η κλήση δέχεται ως ένα OrderId και ένα UpdateOrderPayload σύμφωνα με το οποίο θα γίνει ανανέωση κάποιων στοιχείων της συγκεκριμένης παραγγελίας

api/ Orders/UpdateOrderTestStatus (POST): Η κλήση δέχεται ως παράμετρο ένα OrderId, TestId και Status (Pending, InProgress, Approved) , η συγκεκριμένη βρίσκει την συγκεκριμένη παραγγελία με την συγκεκριμένη εξέταση και ορίζει το status σε αυτό που ήρθε από το Http Request.

Results Controller

Εδώ θα αναφερθούν οι κλήσεις του Results Controller, οι κλήσεις της συγκεκριμένης κλάσης είναι οι εξής:

/api/Results/GetResultsByOrderId (GET): Η κλήση δέχεται ένα OrderId και επιστρέφει όλα τα αποτελέσματα για την συγκεκριμένη παραγγελία.

/**api/ Results/GetResultsByPatientId (GET):** Η κλήση δέχεται ένα PatientId και επιστρέφει όλα τα αποτελέσματα του συγκεκριμένου ασθενή.

/api/ Results/SearchResult (Post): Η κλήση δέχεται SearchResultPayload το οποίο περιέχει κριτήρια για την εύρεση αποτελεσμάτων όπως OrderId, OrderTestId, PatientId.

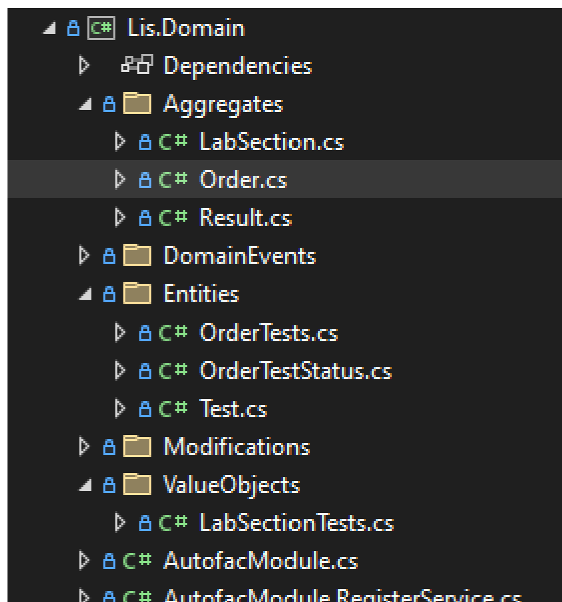
api/ Results/AddResult (POST): Η κλήση δέχεται ως παράμετρο ένα OrderId και ένα CreateResultPayload για την δημιουργία ενός αποτελέσματος για συγκεκριμένη παραγγελία.

api/ Results/UpdateResult (POST): Η κλήση δέχεται ως παράμετρο ένα resultId και ένα UpdateResultPayload για την ανανέωση ενός αποτελέσματος.

4.7.3 Σχεδιασμός Domain επιπέδου

Στο σημείο αυτό ήρθε η ώρα να γίνει μία μικρή επεξήγηση για το Domain επίπεδο του Lis microservice, δηλαδή για το ποια πράγματα μέσα σε αυτό αποτελούν Aggregates, Entities και ποια Value Objects και γενικότερα για το σχεδιασμό αυτού του επιπέδου.

Το συγκεκριμένο επίπεδο τομέα αποτελείται από τα εξής Aggregates, Order, Result και Labsection. Αυτές οι τρεις οντότητες θα αποτελέσουν το κορμό της λογικής του συγκεκριμένου microservice, μιας και είναι ένα service που θα αναφέρεται κυρίως σε πληροφορίες που είναι γύρω από το εργαστηριακά αποτελέσματα και παραγγελίες αυτών. Έπειτα έχουμε τα Entities τα οποία είναι τα εξής, OrderTests, Test, OrderTestStatus και το LabsectionTests που είναι το μοναδικό ValueObject. Όλα τα παραπάνω φαίνονται και στην εικόνα 24 από κάτω.



Εικόνα 26. Επίπεδο Domain του Lis Microservice

Όπως αναφέραμε και στην αρχή του κεφαλαίου αυτού το συγκεκριμένο microservice είναι υπεύθυνο για την καταγραφή, επεξεργασία και διαχείριση εργαστηριακών εξετάσεων ενός ασθενή. Η πρώτη οντότητα που σχεδιάστηκε λοιπόν ήταν αυτή του εργαστηρίου (Labsection) η οποία είναι ένα Aggregate μέσα σε αυτό το Domain και επομένως διαθέτει μεθόδους εκτέλεσης (Commands). Μέσω του Labsection λοιπόν μπορούμε να δημιουργήσουμε ένα νέο εργαστήριο, να προσθέσουμε εξετάσεις οι οποίες θα είναι διαθέσιμες προς παραγγελία μέσω του εργαστηρίου αυτού ή ακόμα και να αφαιρέσουμε κάποιες , τέλος μπορούμε να ορίσουμε υπεύθυνο γιατρό εργαστηρίου. Αυτές είναι κάποιες από τις βασικές λειτουργίες που θεωρήσαμε πως μία τέτοια οντότητα χρειάζεται μέσα σε ένα Lis microservice και φαίνονται και στην παρακάτω εικόνα.

```
public class LabSection : RootEntity<long>
{
    private readonly List<LabSectionTests> _labSectionTests;

    5 references | 0 changes | 0 authors, 0 changes
    public string Description { get; private set; }
    5 references | 0 changes | 0 authors, 0 changes
    public string Phone { get; private set; }
    6 references | 0 changes | 0 authors, 0 changes
    public long? ChiefDoctorId { get; private set; }

    5 references | 0 changes | 0 authors, 0 changes
    public IReadOnlyCollection<LabSectionTests> LabSectionTests => _labSectionTests.AsReadOnly();

    0 references | 0 changes | 0 authors, 0 changes
    private LabSection() ...

    16 references | 0 changes | 0 authors, 0 changes
    public LabSection(long id, LabSectionTests[] labSectionTests, string description, string phone, long? c

    1 reference | 0 changes | 0 authors, 0 changes
    public static CreateLabsectionModification CreateNew(string description, string phone, long? chiefDocto

    1 reference | 0 changes | 0 authors, 0 changes
    public UpdateLabSectionChiefDoctorModification UpdateChiefDoctor(long doctorId)...

    1 reference | 0 changes | 0 authors, 0 changes
    public UpdateLabSectionTestsModification AddLabSectionTests(LabSectionTests[] labSectionTests)...

    1 reference | 0 changes | 0 authors, 0 changes
    public UpdateLabSectionTestsModification RemoveLabSectionTests(LabSectionTests[] labSectionTests)...
```

Οι επόμενες σημαντικές οντότητες που δημιουργήθηκαν και αποτελούν Root Entities στο Domain του Lis είναι το Order και Result και μαζί με το OrderTests entity δημιουργούν μία σύνδεση την οποία θα εξηγήσουμε αμέσως. Η ανάγκη λοιπόν ενός εργαστηριακού συστήματος για την επεξεργασία και διαχείριση εξετάσεων είναι σχετικά απλή, η ανάγκη είναι η προσθήκη εξετάσεων για έναν ασθενή και εν τέλη η προσθήκη απάντησης στις εξετάσεις αυτές. Αυτό οδήγησε λοιπόν στην αρχική δημιουργία του μοντέλου Order αλλά έφτανε μόνο αυτό; η απάντηση είναι όχι, όπως έχουμε ξαναπεί σε ένα microservice το Domain πρέπει να είναι απλό, επεκτάσιμο και εύκολα κατανοητό για νέους προγραμματιστές επομένως αν καταλήγαμε με ένα μόνο Aggregate μάλλον θα καταλήγαμε με ένα αρχείο που θα έκανε πάρα πολλά πράγματα. Εδώ επομένως έγινε ο εξής διαχωρισμός, χρειαζόμαστε έναν τρόπο να προσθέτουμε πολλαπλές εξετάσεις σε μία παραγγελία και όχι να βάζουμε κάθε εξέταση σε μία νέα παραγγελία. Για αυτό το λόγω δημιουργήθηκε το OrderTest entity και το οποίο ζει μέσα στο Order και συγκεκριμένα μία λίστα από OrderTests. Με αυτόν τον τρόπο το επίπεδο της παραγγελίας διαχωρίστηκε με αποτέλεσμα να μπορούμε να έχουμε πολλαπλές εξετάσεις και η κάθε μία από αυτές να έχει το δικό της OrderTestStatus. Συγκεντρωτικά έχουμε λοιπόν την δημιουργία παραγγελίας, ανανέωση OrderTestStatus μίας εξέτασης παραγγελίας, προσθήκη νέων εξετάσεων, αφαίρεση εξετάσεων και επεξεργασία παραγγελίας. Όλες οι διαδικασίες που αναφέρθηκαν φαίνονται και στην παρακάτω εικόνα του Order Aggregate.

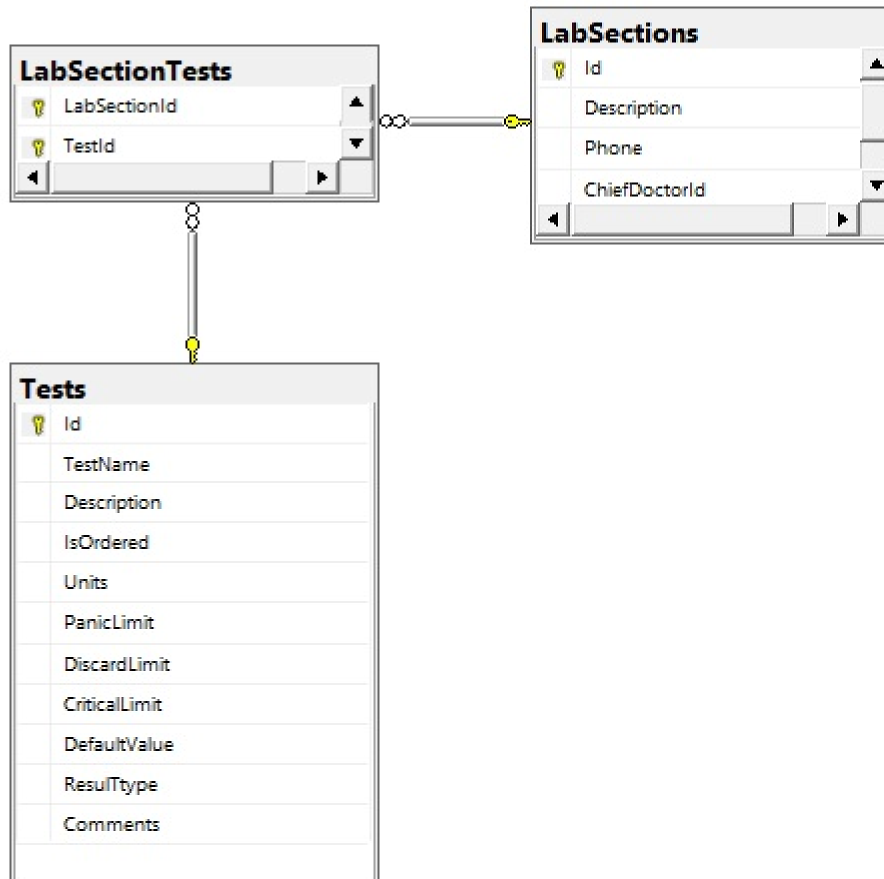
```
6 references | 0 changes | 0 authors, 0 changes
public IReadOnlyCollection<OrderTests> OrderTests => _orderTests.AsReadOnly();

0 references | Panos Onasis, 23 days ago | 1 author, 1 change
private Order() ...

1 reference | Panos Onasis, 23 days ago | 1 author, 1 change
public Order(long id, OrderTests[] orderTests, long[] resultIds, long? patientId, DateTime o

1 reference | Panos Onasis, 23 days ago | 1 author, 1 change
public static CreateOrderModification CreateNew(long? patientId, DateTime orderDate,
    DateTime sampleDate, long? sampleSeqNum, string sampleType,
    long? doctorId, string comments, long? hospitalId, long? visitId)...

1 reference | Panos Onasis, 23 days ago | 1 author, 1 change
public UpdateOrderModification UpdateOrder(long? patientId, DateTime orderDate, DateTime sam
    string? sampleType, long? doctorId, string comments)...

0 references | Panos Onasis, 23 days ago | 1 author, 1 change
public UpdateOrderTestModification UpdateOrderTest(long testId, OrderTests orderTest)...

1 reference | Panos Onasis, 23 days ago | 1 author, 1 change
public UpdateOrderTestStatusModification UpdateOrderTestsStatus(long testId, OrderTestStatus

1 reference | Panos Onasis, 23 days ago | 1 author, 1 change
public UpdateOrderTestsModification AddOrderTests(OrderTests[] orderTests)...

1 reference | Panos Onasis, 23 days ago | 1 author, 1 change
public UpdateOrderTestsModification RemoveOrderTests(OrderTests[] orderTests)...
```

Τέλος έχουμε το τελευταίο Aggregate του Lis Domain το οποίο είναι το Result. Όπως φαίνεται και στην εικόνα από κάτω πρακτικά με την δημιουργία ενός OrderTest θα πρέπει να δημιουργείται και ένα άδειο με αποτελέσματα Result. Πρακτικά το Result έχει μόνο μία σημαντική εκτέλεση μεθόδου να κάνει και αυτή είναι η ανανέωση του αποτελέσματος μίας συγκεκριμένης εξέτασης σε μία συγκεκριμένη παραγγελία για έναν ασθενή. Μέσω αυτού δηλαδή θα έχουμε την τελική εικόνα για το αποτέλεσμα μιας παραγγελίας εξετάσεων σε ένα σύστημα πληροφοριών εργαστηρίου.

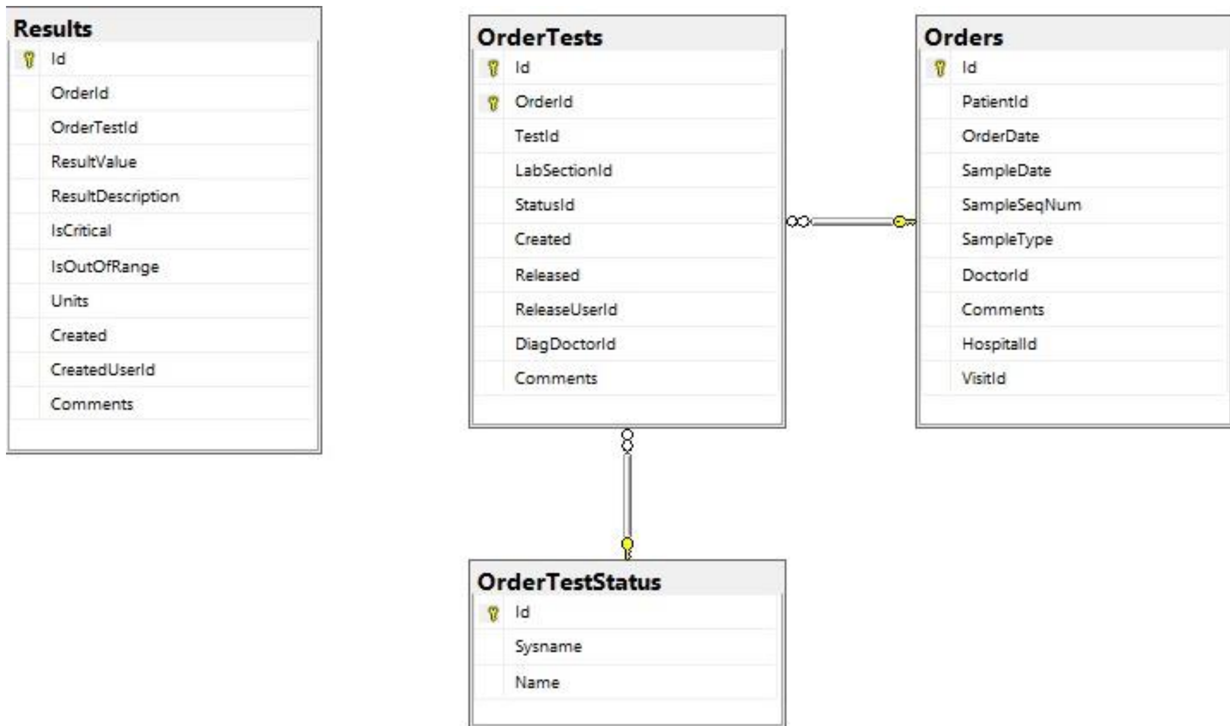
```
    0 references | 0 changes | 0 authors, 0 changes
    private Result() ...

    2 references | 0 changes | 0 authors, 0 changes
    public Result(long id, long orderId, long orderTestId, string resultValue, string resultDescriptio

    1 reference | 0 changes | 0 authors, 0 changes
    public static CreateResultModification CreateNew(long orderTestId, long orderId,string units, long

    1 reference | 0 changes | 0 authors, 0 changes
    public UpdateResultModification UpdateResult(string? resultvalue, string? resultDesc, bool? isCrit
}
```

4.7.4 Βάση Δεδομένων

Για την αποθήκευση των δεδομένων του το συγκεκριμένο microservice χρησιμοποίησε την σχεσιακή βάση MS SQL στην οποία έχει πρόσβαση μόνο από αυτό το microservice και από κανένα άλλο μέσα στο σύστημα Medbook. Στις εικόνες από κάτω φαίνονται οι πίνακες της βάσης Lis που δημιουργήθηκαν για το microservice αυτό αλλά και οι σχέσεις που υπάρχουν μεταξύ τους. Ας δώσουμε μία εξήγηση για τις παρακάτω εικόνες.



Ξεκινώντας από τους παραπάνω πίνακες Labsections, LabsectionTests και Tests παρατηρούμε πως αναφέρονται στα εργαστήρια και τις εξετάσεις που περιέχουν αυτά. Πιο συγκεκριμένα στον πίνακα Labsections είναι αποθηκευμένα τα εργαστήρια έχοντας ένα όνομα, περιγραφή και υπεύθυνο ιατρό. Έπειτα λόγω της ανάγκης του κάθε εργαστηρίου να περιέχει πολλαπλές εξετάσεις δημιουργήθηκε ένας ενδιάμεσος πίνακας ο LabsectionTests ο οποίος συνδέει ένα εργαστήριο με μία εξέταση (κάθε εργαστήριο μπορεί να έχει πολλές εξετάσεις). Τέλος έχουμε τον πίνακα Tests ο οποίος περιέχει ολές τις πληροφορίες που διαθέτει μία εξέταση (Σιδηρου, χοληστερίνης, καλίου) για παράδειγμα κάθε εξέταση θα έχει όνομα, περιγραφή, τύπο αποτελεσμάτων και όρια τιμών.



Στην από πάνω εικόνα βλέπουμε τους τέσσερις πιο βασικούς πίνακες βάσεις του Lis microservice, ας ξεκινήσουμε με τον Orders. Ο συγκεκριμένος πίνακας είναι υπεύθυνος να διατηρεί τις παραγγελίες εργαστηριακών εξετάσεων με την διαφορά πως μία παραγγελία ενός ασθενή μπορεί να πολλαπλές εργαστηριακές εξετάσεις πάνω στο συγκεκριμένο δείγμα εξέτασης όπως για παράδειγμα μία γενική αίματος. Για τον λόγο της σύνδεσης αυτής αλλά και της δυνατότητας υπάρχει και ο πίνακας OrderTests ο οποίος αναφέρετε στις εξετάσεις μίας συγκεκριμένης παραγγελίας εργαστηριακών εξετάσεων. Ο πρώτος περιέχει δεδομένα όπως αριθμό ασθενή, ιατρού, επίσκεψης, τύπο δείγματος και ημερομηνία παραγγελίας ενώ ο δεύτερος περιέχει τον αριθμό εξέτασης, την κατάσταση της συγκεκριμένης εξέτασης, τον γιατρό διάγνωσης και αποτελέσματος. Τέλος έχουμε τον πίνακα Results ο οποίος περιέχει τα αποτελέσματα ενός συγκεκριμένου OrderTest συγκεκριμένα περιέχει τον ακριβή αριθμό, επεξήγηση του αποτελέσματος, αν πρόκειται για αποτέλεσμα εκτός ορίων, πότε καταχωρήθηκε και την μονάδα μέτρησης του αποτελέσματος.

4.8 Clinical Information Microservice

Κάπου εδώ βρισκόμαστε στο τελευταίο microservice του συστήματος Medbook που δημιουργήθηκε για την παρούσα διπλωματική εργασία. Το συγκεκριμένο microservice είναι ίσως λίγο πιο εύκολο από όσα έχουμε δει ως τώρα και έχει και κάτι ξεχωριστό σε σχέση με τα άλλα. Η σημαντική διαφορά είναι πως δεν διαθέτει ένα επίπεδο Domain όπως τα υπόλοιπα microservice που είδαμε ως τώρα. Αυτό συμβαίνει διότι το Clinical Information Microservice δεν υλοποιεί κάποια ξεχωριστή λογική στο ιατρικό παρόν ιατρικό σύστημα που φτιάξαμε, όμως αυτό δεν σημαίνει πως δεν έχει έναν δικό του σημαντικό ρόλο.

Ο ρόλος αυτός λοιπόν είναι να προσφέρει μία πιο σφαιρική εικόνα και να συλλέξει πολλαπλά δεδομένα από τα επιμέρους συστήματα του Medbook δηλαδή τα Core Microservices (User, Patient,Personnel, Lis) και να τα συνδυάσει. Όπως θα έχετε καταλάβει ως τώρα ένα σημαντικό κομμάτι της αρχιτεκτονικής των microservice είναι η διάσπαση της λογικής και της πληροφορίας σε επιμέρους συστήματα και για λόγους ασφαλείας αλλά και για λόγους επεκτασιμότητας, βελτίωσης και άλλων δυνατοτήτων. Αυτά όμως οδηγούν στο ότι μία front-end εφαρμογή θα χρειαστεί να ξέρει και να καλέσει όλα αυτά τα microservices και εν τέλη να συνδυάσει όλη αυτήν την πληροφορία, όπως είπαμε όμως και στα εισαγωγικά κεφάλαια κάτι τέτοιο θα ήταν τελείως λάθος. Για αυτόν τον λόγω λοιπόν δημιουργήθηκε ένας ενδιάμεσος κρίκος, ο οποίος όχι μόνο θα επικοινωνεί με τα επιμέρους χωρισμένα συστήματα αλλά θα κάνει συνδυαστικές Http κλήσεις τις οποίες θα επεξεργάζεται και θα τις συνδυάζει σε ένα ενιαίο μοντέλο το οποίο θα είναι και αυτό εν τέλη το οποίο θα βλέπει ένας χρήστης στην δ επαφή του είτε είναι ένα website ή μέσω ενός κινητού.

4.8.1 Αρχιτεκτονική

Το ClinicalInformation microservice είναι υλοποιημένο με C# και με την χρήση του framework .Net 5.0 που προσφέρει η Microsoft και αποτελείται από έξι βιβλιοθήκες(DLL) οι οποίες είναι ClinicalInformation.API, ClinicalInformation.Application, ClinicalInformation.Common & ClinicalInformation.Shared.

Από αυτές τις έξι αυτή η οποία θα τρέξει με την βοήθεια ενός Docker ή μέσα στον IIS ενός Windows Server είναι το Lis.API το οποίο αποτελεί το ASP.NET Core Application το οποίο περιέχει όλους τους Controllers του Lis microservice που ο καθένας περιέχει τα επιμέρους HTTP Requests που προσφέρει το συγκεκριμένο service.

Ένα βασικό στοιχείο στο συγκεκριμένο Api αποτελεί η κλάση **ExceptionsMiddleware** η οποία είναι ένας διαμεσολαβητής των εισερχόμενων Http Request και είναι υπεύθυνη να διαχειρίζεται τα εσωτερικά σφάλματα που ίσως προκύψουν από κάποιες κλήσεις ώστε είτε να γυρίσει κάποιο συγκεκριμένο μήνυμα ή HttpStatusCode (Unathorized, NotImplemented) και στο τέλος να καταγράψει το σφάλμα με την βοήθεια του Nlog(βιβλιοθήκη η οποία καταγράφη Logs) σε ένα αρχείο txt .

Έπειτα έρχεται το ClinicalInformation.Application, η συγκεκριμένη βιβλιοθήκη αποτελεί βασικό κομμάτι του microservice καθώς μέσα σε αυτό είναι υλοποιημένο το CQRS πρότυπο που αναφέραμε σε προηγούμενο κεφάλαιο. Πιο συγκεκριμένα το API περιέχει απλά τους Controllers οι οποίοι με την χρήση του Mediator ενεργοποιούν τους Command ή Query Handlers που βρίσκονται μέσα στο ClinicalInformation.Application και είναι υπεύθυνοι για τον συντονισμό των διεργασιών και την επικοινωνία με τα επιμέρους Core Microservices. Όπως καταλαβαίνετε το συγκεκριμένο microservice δεν θα περιέχει Domain και Infrastructure επίπεδο καθώς δεν επικοινωνεί με καμία βάση δεδομένων και δεν περιέχει καμία λογική πάνω στο Medbook σύστημα παρά μόνο τον συντονισμό και την οργάνωση κλήσεων από τα Core Microservices.

Τέλος το Lis.Shared είναι το DLL που περιέχει όλα τα “Contracts” τα οποία θα γυρνάει το ClinicalInformation.API μέσω των απαντήσεων του στα HTTP Requests. Επίσης μέσα σε αυτό το DLL θα υπάρχει ένα cmd αρχείο το οποίο θα τρέχει όταν θέλουμε να παράγουμε μία νέα έκδοση της βιβλιοθήκης του ClinicalInformation.Shared και των κλήσεων του, οι οποίες θα πρέπει να χρησιμοποιηθούν από κάποιο άλλο microservice. Πιο συγκεκριμένα με την βοήθεια της βιβλιοθήκης Swagger γίνεται η παραγωγή δύο αρχείων τα οποία θα γίνουν export σε ένα nuget package το οποίο θα είναι διαθέσιμο να χρησιμοποιηθεί από άλλα microservices. Με πιο απλά λόγια φέρνει τα μοντέλα και τα Http Requests του συγκεκριμένου microservice διαθέσιμα προς χρήση μέσα σε ένα άλλο (δίαυλος επικοινωνίας).

5. Εκτέλεση και χρήση microservices

5.1 Περίληψη Κεφαλαίου

Στην συγκεκριμένη ενότητα της διπλωματικής εργασίας θα εκτελέσουμε όλα τα microservice που υπάρχουν στο σύστημα του Medbook και θα δείξουμε κάποια πραγματικά σενάρια χρήσης που θα είχε το συγκεκριμένο σύστημα μέσα σε ένα νοσοκομείο.

Πριν ξεκινήσουμε την περιγραφή των σεναρίων ας αναφερθούμε σε κάποια εργαλεία που βοήθησαν σε αυτό. Για την εκτέλεση των microservices χρησιμοποιήθηκε ο γνωστός IIS (Internet Information Services) ο οποίος αποτελεί έναν web server ο οποίος είναι της Microsoft και τρέχει στα Windows. Επίσης δεν δημιουργήθηκε κάποιο User Interface για την εκτέλεση του προγράμματος οπότε για την αποτελεσματικότερη παρουσίαση των παραδειγμάτων εκτέλεσης και των πιθανών σεναρίων που θα δούμε χρησιμοποιήσαμε ένα πολύ γνωστό εργαλείο για την πραγματοποίηση των Http Request στα microservices το Postman. Το Postman αποτελεί μία εφαρμογή την οποία μπορεί να εγκαταστήσει ο καθένας στον υπολογιστή του και είναι υπεύθυνή για τις δοκιμές σε ένα API, μέσω αυτού μπορούμε να εκτελέσουμε HTTP Requests μέσω ενός μιας πολύ φιλικής διεπαφής προς τον χρήστη και να δούμε και τα αποτελέσματα των κλήσεων αυτών.

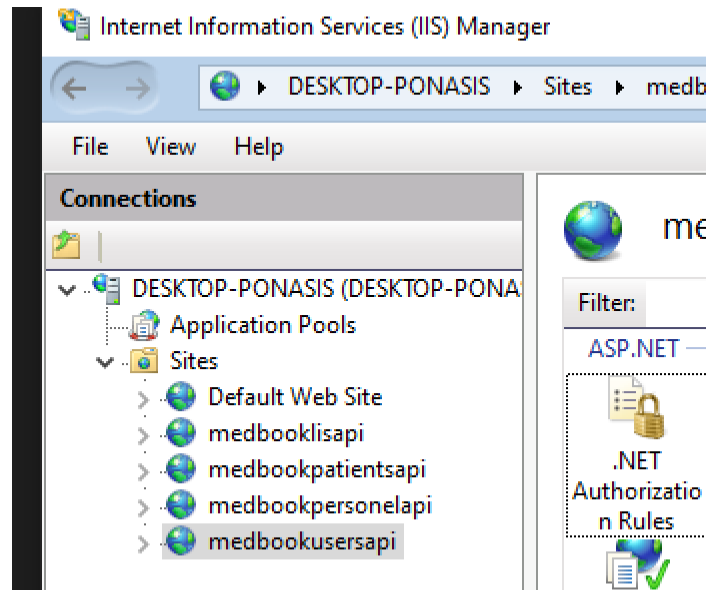
5.2 Εκτέλεση Microservice Medbook

Πριν ξεκινήσει η ανάπτυξη και παρουσίαση των σεναρίων χρήσης στο σύστημα Medbook θα δώσουμε μία σύντομη περιγραφή για το τι χρειάστηκε ακριβώς για να στηθούν και να τρέξουν στον IIS ενός υπολογιστή με λογισμικό Windows. Αρχικά για κάθε ένα microservice κάναμε publish το API μέσω του IDE του Visual Studio, το οποίο παράγει έναν φάκελο με όλα τα απαραίτητα αρχεία και DLL που χρειάζεται η συγκεκριμένη εφαρμογή για να τρέξει σε ένα server ή Docker. Μέσα σε αυτά τα αρχεία είναι και το appsettings.json το οποίο είναι ένα αρχείο το οποίο περιέχει παραμέτρους οι οποίες χρειάζεται για να διαθέτει το microservice κάποιες συγκεκριμένες πληροφορίες όπως για παράδειγμα το μέρος που βρίσκεται η βάση δεδομένων ή το κλειδί που χρησιμοποιεί ο αλγόριθμος που παράγει τα unique authentication token των χρηστών. Παρακάτω δίνεται ένα παράδειγμα ενός τέτοιου αρχείου (Εικόνα 27).

```
{
  "instance": {
    "cacheType": "Redis", //InMemory|Redis
    "databaseType": "SqlServer", // InMemory|SqlServer
    "applyDatabaseMigrations": true
  },
  "connectionStrings": {
    "medBookUsers": "server=localhost;database=Users;uid=sa;password=1234;trusted_connection=false"
  },
  "JwtAuthentication": {
    "Key": "MIIBOQIBAAJBAMHtk951jZ0pdrGIZAHINitYK9mR8bCtjAq1TG7b1cl24lic0ZTb3U1/Sh+cdGnxEllyH00G2z+v/2GaNATxS7
    "Issuer": "MedBook.gr"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

Εικόνα 27. Παράδειγμα αρχείου AppSettings.json

Αφού λοιπόν τοποθετήθηκαν όλες οι παράμετροι που χρειαζόταν το κάθε microservice, πήγαμε και τοποθετήσαμε το κάθε ξεχωριστό φάκελο για κάθε ένα από αυτά μέσα στον local IIS server.Όπως φαίνεται και στην παρακάτω εικόνα μετά την χορήγηση κάποιων δικαιωμάτων και την δημιουργία application pool για κάθε microservice, φαίνονται πλέον όλα τα microservice μέσα στον IIS server και είναι έτοιμα για να ξεκινήσουν. Με την πρώτη εκκίνηση του κάθε microservice δημιουργούνται όλοι οι απαραίτητοι SQL πίνακες μέσα στην βάση δεδομένων που έχουμε ορίσει στα παραμετρικά αρχεία που αναφέραμε πριν.



Τέλος μόλις όλα τα microservices τρέξουν αν πάμε στο Clinical Information microservice και προηγηθούμε στην σελίδα /healthchecks , θα παρατηρήσουμε αν όλα τα microservices είναι πάνω και είναι «Healthy» (Εικόνα 28). Να θυμίσουμε πως το Clinical Information microservice είναι ο κεντρικός κόμβος επικοινωνίας με όλα τα microservice και για αυτό η συγκεκριμένη σελίδα βρίσκεται σε αυτόν.

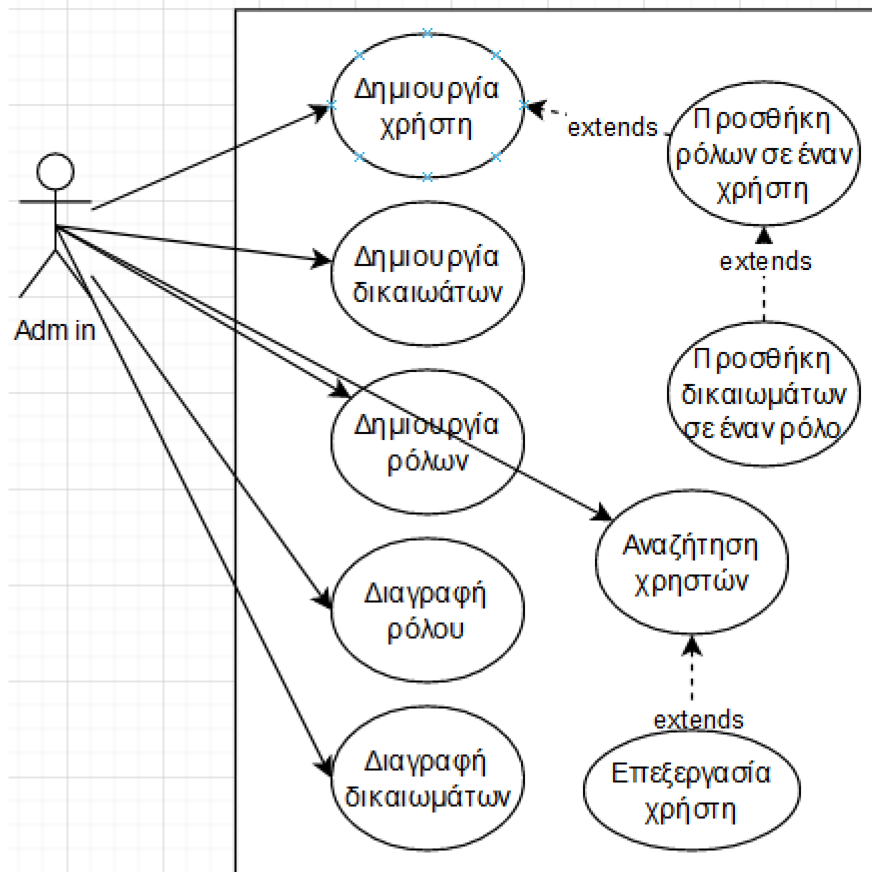
Application Health Overview

NAME	TAGS	HEALTH
Disk storage		Healthy
Memory allocated		Healthy
Service.Users		Healthy
Service.Personel		Healthy
Service.Patients		Healthy
Service.Lis		Healthy

Εικόνα 28.Microservices HealthCheck UI

5.3 Περιγραφή σεναρίων χρήσης

Αφού η εκτέλεση όλων των microservices ολοκληρώθηκε ας περάσουμε στην παρουσίαση μερικών περιπτώσεων χρήσης του συστήματος του Medbook, όπως είπαμε και πριν αυτό θα γίνει με την χρήση του εργαλείου Postman. Οι περιπτώσεις χρήσης θα χωριστούν σε δύο κομμάτια τις περιπτώσεις χρήσης από την μεριά του ιατρικού προσωπικού του νοσοκομείου όπως η γραμματεία , ένας ιατρός ή ένας διαχειριστής συστήματος και υπηρεσιών (admin) και από την μεριά του ασθενή.
Ας ξεκινήσουμε με τις δυνατότητες και τις περιπτώσεις από την μεριά του ιατρικού προσωπικού συγκεκριμένα του admin, αφού στην αρχή του συστήματος ο μόνος που θα έχει κωδικούς χρήσης και δικαίωμα για είσοδο θα είναι αυτός. Όπως βλέπουμε και στην εικόνα από κάτω οι βασικές ενέργειες που θα έχει ένας admin στο σύστημα μας θα είναι η διαχείριση των χρηστών σε χαμηλό επίπεδο.



Εικόνα 29. Διάγραμμα χρήσης του Medbook από έναν admin

Αρχικά το πρώτο Http Request θα είναι για την είσοδο του στο σύστημα και φαίνεται στην παρακάτω εικόνα. Όπως βλέπουμε αν το username και το password είναι σωστά θα επιστραφεί ένα μοναδικό «Authorization Token» το οποίο θα χρησιμοποιηθεί για όλες τις επόμενες κλήσεις που θα γίνουν. Αν δεν το χρησιμοποιήσουμε τότε θα δούμε ένα «Authorization Error» το οποίο μας λέει πως πρακτικά δεν έχουμε δικαίωμα για την εκτέλεση της συγκεκριμένης κλήσης, το ίδιο θα συνέβαινε αν στέλναμε Token το οποίο δεν περιέχει τα κατάλληλα «Access Rights».

/ api / Authentication / **Authentication Authenticate**

POST {{baseUrl}}/api/Authentication/Authenticate

Params Authorization Headers (9) **Body** Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON


```
1 {
2     "username": "admin",
3     "password": "admin"
4 }
```


Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON


```
1 {
2     "$type": "User.Shared.SuccessAuthenticationResultModel, User.Shared",
3     "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
          eyJVU0VSSUQiOiI4Njg1MDI2OTk1MDU2MTQ4NDgiLCJGSVJTVE5BTUUiOiJHaW9yZ29zIiwiTEFTVE5BT
          L21kZW50aXR5L2NsYW1tcy9yb2xlIjpbIkFETU1OSVNUUkFUSU9OU19BQ0NFU1MiLCJET0NUT1JfQUNDR
          Q0NFU1MiLCJQYXRpZW50X0R1bW9ncmFwaGljX0FjY2VzcyIsI1BBVEhPTE9HWV9BQ0NFU1MiLCJUZXN0c
          4jhg4_TaWXSc4CMdK-4zE4STtR6O_mqqeIrX_1Kl7jE",
4     "refreshToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJVU0VSSUQiOiI4Njg1MDI2OTk1MDU
          cAjKISFMrv7r5jEPW4ZVyj2GCkVQSvEDQJS_ikypx-g"
5 }
```


Έπειτα εφόσον ο admin έχει κάνει την είσοδο του στο σύστημα θα μπορεί να προσθέσει νέους χρήστες οι οποίοι θα αποτελούν ασθενείς ή ιατρικό προσωπικό. Με την προσθήκη ενός νέου χρήστη θα επιστραφεί το μοναδικό αναγνωριστικό του το οποίο θα χρησιμοποιήσει ο admin για να του προσθέσει ρόλους. Στην παρακάτω εικόνα φαίνεται η απάντηση της συγκεκριμένης κλήσης μετά την προσθήκη ενός νέου ασθενή. Όπως φαίνεται ο συγκεκριμένος χρήστης ακόμα δεν έχει κάποιο ρόλο. Ας εκτελέσουμε την επόμενη ενέργεια λοιπόν του admin η οποία είναι η προσθήκη και η αφαίρεση δικαιωμάτων.

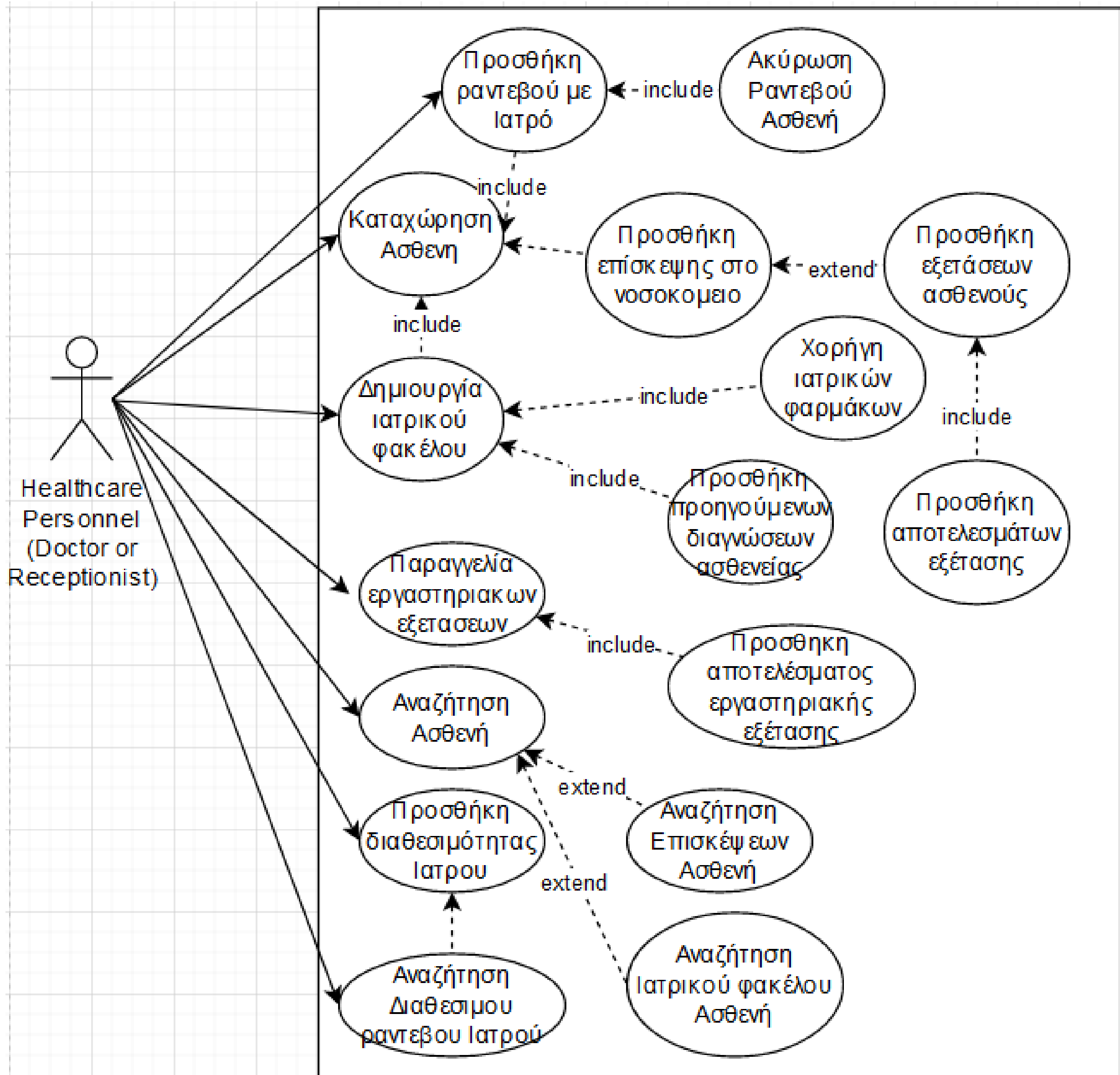

```
1  {
2      "$type": "User.Shared.UserModel, User.Shared",
3      "userId": 1041397522888654848,
4      "username": "patient1",
5      "email": "testPatient@gamil.com",
6      "firstName": "testPatient",
7      "lastName": "testPatient",
8      "phone": "1231131231",
9      "roles": [],
10     "doctorId": null,
11     "patientId": null
12 }
```

Αφού λοιπόν ο admin εκτελέσει το AddUserRoles request και έπειτα ξαναζητήσει να δει τα στοιχεία του συγκεκριμένου ασθενή η παραπάνω εικόνα θα έχει αλλάξει. Στην νέα απάντηση λοιπόν της συγκεκριμένης κλήσης βλέπουμε πως πλέον ο χρήστης του ασθενή TestPatient έχει έναν νέο ρόλο(Patient) ο οποίος μέσα του περιέχει ένα δικαίωμα (Patient_Access).


```
"$type": "User.Shared.UserModel, User.Shared",
"userId": 1041397522888654848,
"username": "patient1",
"email": "testPatient@gamil.com",
"firstName": "testPatient",
"lastName": "testPatient",
"phone": "1231131231",
"roles": [
    {
        "$type": "User.Shared.Role, User.Shared",
        "roleId": 3,
        "sysname": "PATIENT",
        "name": "Patient",
        "permissions": [
            {
                "$type": "User.Shared.Permission, User.Shared",
                "id": 4,
                "sysname": "PATIENT_ACCESS",
                "name": "Patient",
                "description": ""
            }
```


Οι συγκεκριμένες ενέργειες θα είναι μερικές από τις βασικές περιπτώσεις χρήσης του συστήματος Medbook από την μεριά του διαχειριστή του συστήματος. Ας αναφέρουμε όμως μερικές ακόμα χωρίς την χρήση του Postman, δημιουργία και διαγραφή ρόλων, δημιουργία και διαγραφή δικαιωμάτων διαγραφή χρηστών και γενική επεξεργασία στοιχείων ενός χρήστη σε περίπτωση αλλαγής τηλεφώνου ή άλλων στοιχείων.

Συνεχίζοντας με τις περιπτώσεις χρήσης του συστήματος μας θα αναφερθούμε στις ενέργειες τις οποίες θα μπορεί να εκτελέσει ένας ιατρός ή γενικά το ιατρικό προσωπικό όπως η γραμματεία για παράδειγμα. Όπως βλέπουμε και στην εικόνα από κάτω οι βασικές ενέργειες που θα έχει ένα τέτοιος ρόλος αφορούνε την διαχείριση του ιατρικού φακέλου ενός ασθενή στο σύστημα Medbook. Ας αναλύσουμε λίγο το διάγραμμα περιπτώσεων που βλέπουμε από κάτω.



Εικόνα 30. Διάγραμμα χρήσης του Medbook από ιατρικό προσωπικό

Μία από τις βασικότερες λειτουργικότητες είναι η προσθήκη ενός νέου ασθενή στο σύστημα Medbook το οποίο είναι ένα αναγκαστικό βήμα το οποίο πρέπει να γίνει αλλιώς δεν θα μπορούν να εκτελεστούν έξτρα ενέργειες πάνω σε έναν ασθενή. Μόλις λοιπόν ένας χρήστης από το ιατρικό προσωπικό συμπληρώσει τα στοιχεία του ασθενή και τα καταχωρίσει στην συνέχεια θα μπορεί να ανοίξει και τον ιατρικό φάκελο του ασθενή. Μέσω του ιατρικού φακέλου του ασθενή το προσωπικό θα είναι σε θέση να χορηγήσει φάρμακα σε έναν ασθενή και να τα καταχωρήσει στο σύστημα, να παραγγείλει μία ανάλυση εργαστηριακών εξετάσεων όπως γενική αίματος, να κλείσει ένα ραντεβού με κάποιον ιατρό στα εξωτερικά ή εσωτερικά ιατρεία, προσθήκη εισαγωγής στο νοσοκομείο και προσθήκη αποτελεσμάτων στις εξετάσεις είτε είναι εργαστηριακές είτε αξονικές. Αλλά εκτός από τις προσθήκες νέων εξετάσεων και επισκέψεων το προσωπικό έχει και την δυνατότητα αναζήτηση ενός ασθενή μέσω διαφόρων κριτηρίων όπως όνομα, επίθετο, ηλικία ή αναζήτηση επισκέψεων ενός ασθενή. Τέλος ένας χρήστης από το προσωπικό θα μπορεί να προσθέτει και να διαχειρίζεται τις διαθέσιμες ώρες ραντεβού ενός ιατρού και να τις αλλάζει. Ας δούμε μερικές από τις παραπάνω κλήσεις με την χρήση του Postman εργαλείου.

Στην παρακάτω εικόνα βλέπουμε την εκτέλεση της αναζήτησης ενός ασθενή με βάση κριτηρίων, στην συγκεκριμένη περίπτωση ψάχνουμε ασθενείς που έχουν το όνομα “Kostas” , όπως φαίνεται στα αποτελέσματα της κλήσης βρέθηκαν δύο ασθενείς με το όνομα αυτό και έχουμε πλέον τις πληροφορίες τους.

POST {{baseUrl}}/api/Patients/SearchPatients

Params Authorization Headers (10) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1  {
2    "firstname": "Kostas",
3    "lastname": "",
4    "hisId": "",
5    "amka": "",
6    "email": "",
7    "phone": "",
8    "birthdateFrom": "",
9    "birthdateTo": ""
10 }
```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize

[{"$type":"Patient.Shared.PatientModel,
Patient.Shared","id":1041741784742363136,"firstName":"Kostas","middlename":"","lastName":"Onasis",
03-04T04:44:05.179Z","gender":"male","hisId":"856305893","amka":"123746327","nationality":"Greek",
{"$type":"Patient.Shared.PatientModel,
Patient.Shared","id":1041742156315754496,"firstName":"Kostas","middlename":"Papadopoulos","lastNa
03-04T04:44:05.179Z","gender":"male","hisId":"8512305893","amka":"1237945327","nationality":"Greel
11","number":null,"zipCode":null}]

Άλλο ένα παράδειγμα των παραπάνω λειτουργιών που αναφέρθηκαν φαίνεται στην από κάτω εικόνα, στην οποία εκτελείται μία αναζήτηση παραγγελιών στις εργαστηριακές εξετάσεις του νοσοκομείου. Όπως φαίνεται και από την εικόνα έχουμε την δυνατότητα να συμπληρώσουμε κωδικό ασθενή, κωδικό ιατρού, κωδικό επίσκεψης , κωδικό εξέτασης εργαστηρίου και άλλα χαρακτηριστικά που αφορούν τις παραγγελίες. Αφού λοιπόν ο χρήστης επιλέξει τα κριτήρια αναζήτησης και εκτελέσει την αναζήτηση θα εμφανιστούν όλες οι παραγγελίες όλων των ασθενών που αντιστοιχούν στους συγκεκριμένους όρους. Παρακάτω έχουμε ζητήσει να δούμε όλες τις παραγγελίες που περιέχουν την εξέταση με κωδικό 2, όπως φαίνεται βρέθηκαν τρεις τέτοιες παραγγελίες διαφορετικών ασθενών.

POST {{baseUrl}}/api/Order/SearchOrders

Params Authorization Headers (10) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

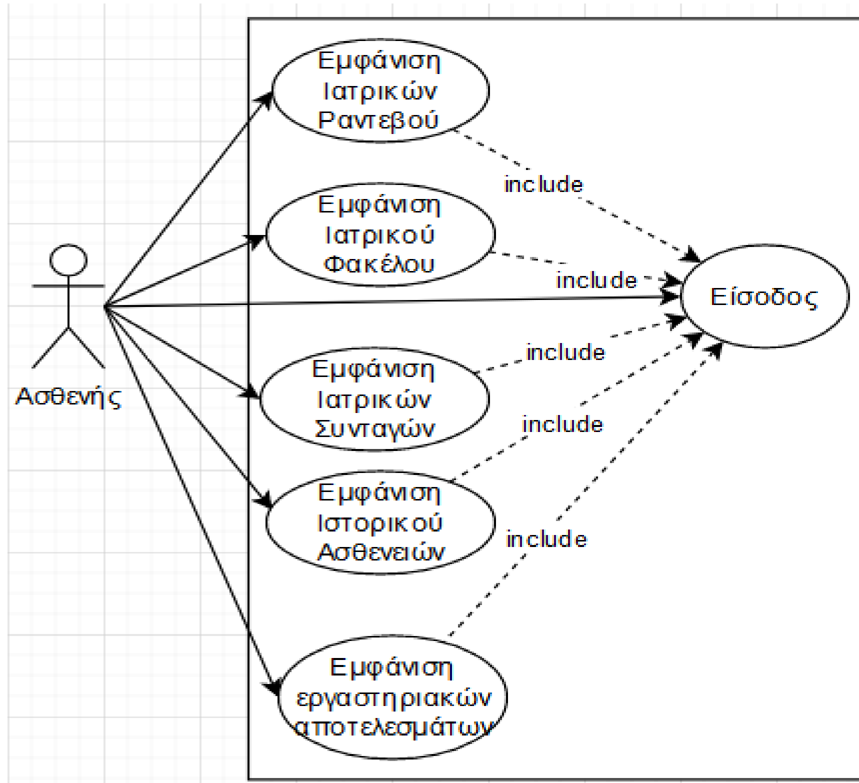

```
1  {
2    "patientId": null,
3    "doctorId": null,
4    "hospitalId": null,
5    "visitId": null,
6    "orderId": null,
7    "testId": 2,
8      "orderDateFrom": "",
9     "orderDateTo": "",
10    "sampleDateFrom": "",
11    "sampleDateTo": ""
```


Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize


```
[{"$type":"Lis.Shared.OrderModel, Lis.Shared","id":1041051025986813952,"patientId":10315679990997
"sampleSeqNum":123123,"sampleType":"blood","doctorId":881570150975799296,"comments":"test order",
"id":1041752438559735808,"patientId":1031567999099731968,"orderDate":"2022-06-10T17:36:41.4Z","sa
"doctorId":881570150975799296,"comments":"test order","hospitalId":1,"visitId":104104305817236275
"patientId":1041741496631427072,"orderDate":"2022-06-10T17:36:41.4Z","sampleDate":"2022-05-11T20:
"comments":"test order","hospitalId":1,"visitId":null}]
```


Συνεχίζοντας με τις περιπτώσεις χρήσης του συστήματος μας θα αναφερθούμε στον τελευταίο πιθανό ρόλο που προσφέρει το συγκεκριμένο ιατρικό σύστημα και είναι ο ρόλος του ασθενή. Στο συγκεκριμένο σενάριο θα παρατηρήσουμε πως ο ασθενής έχει πολύ βασικές δυνατότητες και επιλογές που κυρίως αφορούν την ανάγνωση και την εμφάνιση πληροφοριών για τον φάκελο του και τις εξετάσεις του. Όπως φαίνεται και στην εικόνα 31 από κάτω ο ασθενής για να εκτελέσει οποιαδήποτε ενέργεια που αφορά τον ιατρικό του φάκελο θα πρέπει να εισαχθεί στο σύστημα του Medbook με τους μοναδικούς κωδικούς που πήρε από το σύστημα μας κατά την εγγραφή του στο νοσοκομειακό σύστημα Medbook. Αφού λοιπόν γίνει η είσοδος στο σύστημα οποιαδήποτε από τις πληροφορίες που βλέπει θα αφορούν μόνο το δικό του φάκελο και όχι άλλου ασθενή, επομένως θα έχει την δυνατότητα να δει τα αποτελέσματα εργαστηριακών εξετάσεων που έχει παραγγείλει, ιατρικές συνταγές που έχουν γραφτεί για αυτών και αφορούν φάρμακα, πιθανά ραντεβού που ίσως έχει προγραμματίσει με κάποιον ιατρό αλλά δεν θα μπορεί να τα επεξεργαστεί και τέλος μία σελίδα με όλα τον ιατρικό φάκελο αναλυτικά που θα περιέχει και παλαιότερες διαγνώσεις ασθενειών και εισαγωγών στο νοσοκομείο.



Εικόνα 31. Διάγραμμα χρήσης του Medbook από έναν ασθενή

Ας δούμε μερικές από τις παραπάνω συγκεντρωτικές κλήσεις του ιατρικού φακέλου του ασθενή με την χρήση του Postman εργαλείου. Στις επόμενες δύο εικόνες λοιπόν θα δούμε μία κλήση για την προβολή των εργαστηριακών αποτελεσμάτων και παραγγελιών που έχει ένας ασθενής, πιο συγκεκριμένα φαίνονται όλα τα στοιχεία του συγκεκριμένου αποτελέσματος όπως αποτέλεσμα, περιγραφή αποτελέσματος, σχόλια και άλλα.

GET {{baseUrl}}/api/Patients/SearchPatientTestResults?patientId=1031567999099731968&labsectionIds=null&testIds=null&resultDateFrom=null&r

Params Authorization Headers (8) Body Pre-request Script Tests Settings

Query Params

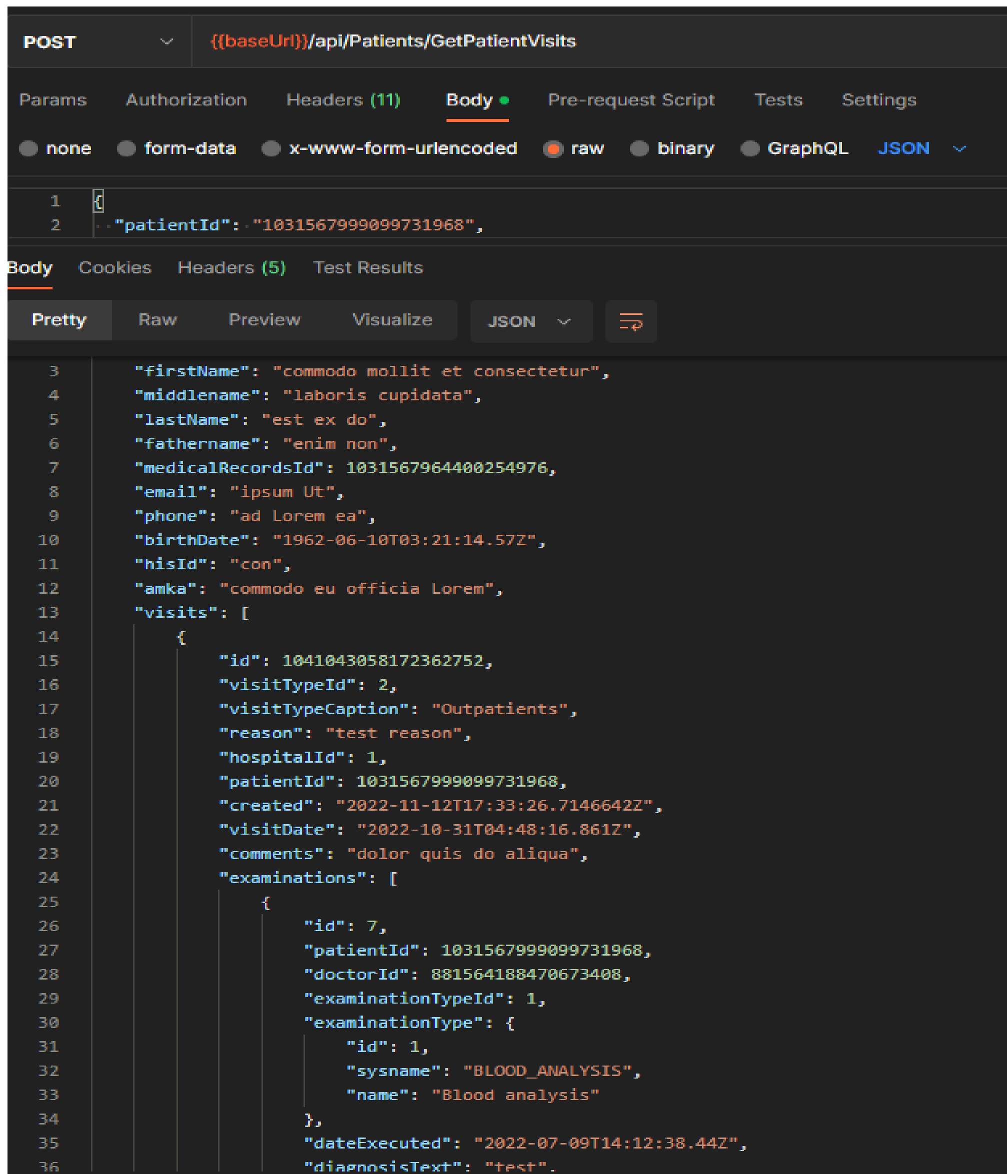
KEY	VALUE	DESC
patientId	1031567999099731968	

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON


```
{
    "patientId": 1031567999099731968,
    "testResults": [
        {
            "labsectionId": 1,
            "labsectionName": "",
            "testId": 1,
            "testname": "",
            "result": {
                "id": 1041100362783653888,
                "orderId": 1041051025986813952,
                "orderTestId": 1,
                "resultValue": "test result",
                "resultDescription": "test desc",
                "isCritical": null,
                "isOutOfRange": null,
                "units": "incididunt cupidatat",
                "created": "2022-11-12T21:21:09.1971753Z",
                "createdUserId": 1,
                "comments": "Excepteur fugiat"
            }
        },
        {
```

Η δεύτερη εικόνα τώρα όπως βλέπουμε από κάτω μας δείχνει τις συνολικές επισκέψεις ενός ασθενή στο νοσοκομείο και τις εξετάσεις που έγιναν σε κάθε μία από τις επισκέψεις αυτές αν υπάρχουν. Επίσης σε κάθε επίσκεψη καταγράφονται οι πιθανές εξετάσεις που μπορεί να γίνουν αλλά και τα φάρμακα που μπορεί να χορηγηθούν στον ασθενή. Στην εικόνα για παράδειγμα βλέπουμε πως στην επίσκεψη με κωδικό 1041043058172362752 έχει γίνει μία ανάλυση αίματος στον συγκεκριμένο ασθενή.



6. Συμπεράσματα και πιθανές επεκτάσεις

6.1 Συμπεράσματα

Η παρούσα διατριβή είχε ως στόχο την ανάλυση ενός πολύπλοκου μονολιθικού συστήματος, στην περίπτωση μας ενός ιατρικού συστήματος το οποίο θα αποτελούσε πρότυπο για την υλοποίηση και τον σχεδιασμό για την νέα αρχιτεκτονική των microservice.

Η συγκεκριμένη αρχιτεκτονική αποτελεί πλέον ένα ευρέως γνωστό πρότυπο που όλο και περισσότερες εταιρείες τείνουν να ακολουθούν, αυτό όμως δεν σημαίνει ότι μπορεί να χρησιμοποιηθεί παντού και σε όλες τις περιπτώσεις. Όπως έχουμε ήδη αναφέρει η κάθε αρχιτεκτονική έχει τα θετικά και τα αρνητικά της, συγκεκριμένα οι ομάδες προγραμματιστών που θα εφαρμόσουν για πρώτη ένα τέτοιο πρότυπο θα αντιμετωπίσουν δυσκολία στην ανάπτυξη αλλά μεγάλη ευκολία στις αλλαγές και στην ευελιξία των microservices. Όπως και στην παρούσα εργασία θα χρειαστούν να σκεφτούν με έναν τελείως διαφορετικό τρόπο και να θέσουν καινούργια ερωτήματα μεταξύ τους, όπως ποιες είναι οι βασικές λειτουργίες του συστήματος μας και πόσα διαφορετικά Domain μπορούν να δημιουργηθούν ώστε να υπάρχει ένας σωστός και λογικός χωρισμός της επιχειρηματικής λογικής χωρίς υπερβολές.

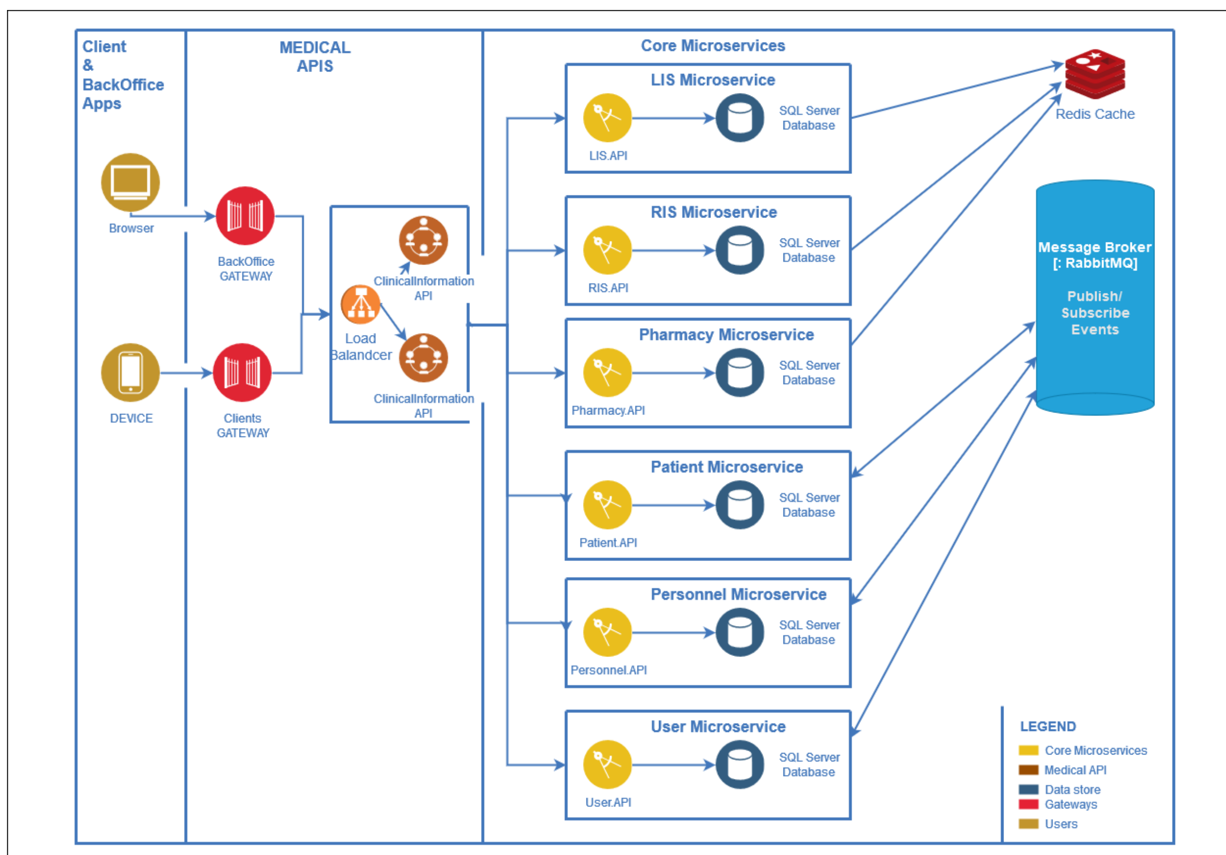
Έπειτα ακόμα και όταν ο διαχωρισμός γίνει και η ομάδα καταλήξει στο πόσα και ποια θα είναι τα microservice του συστήματος της θα πρέπει να σκεφτεί και να υλοποιήσει νέα τεχνικά πρότυπα μέσα στο κάθε microservice όπως CQRS, Event Sourcing, Retry Pattern. Εκτός από τα πρότυπα υπάρχουν και άλλα πράγματα προς σκέψη και ανάλυση που σε ένα μονόλιθο δεν χρειάστηκε ποτέ να σκεφτούμε, όπως ποιο microservice θα επικοινωνεί με ποιο, και με ποιον τρόπο με Http Requests, ή Message Queues . Επίσης χρειάστηκε να σκεφτούμε την δημιουργία ενός ενιαίου συστήματος καταγραφής για Logs καθώς πλέον δεν έχουμε μία back-end εφαρμογή αλλά 5 microservices τα οποία έχουν τα δικά τους ξεχωριστά logs είτε είναι Errors ή Infos .

Σκοπός λοιπόν της παρούσας διπλωματικής ήταν να παρουσιάσει και να δώσει λύσεις σε μερικά από τα παραπάνω προβλήματα, να δώσει ιδέες και μια διαφορετική οπτική γωνία για τον τρόπο υλοποίησης ενός τέτοιου συστήματος από πολλαπλά microservices, κάτι και το οποίο έκανε. Αναλύθηκαν αρκετά σχεδιαστικά πρότυπα τα οποία γίνανε και πράξη, αναφέρθηκαν χρήσιμες βιβλιοθήκες που χρειάζονται και είναι απαραίτητες μέσα σε ένα τέτοιο σύστημα αλλά και τις δυνατότητες και προοπτικές που προσφέρει στο τέλος της ημέρας τόσο σε έναν προγραμματιστή αλλά και ολόκληρες ομάδες. Τέλος παρακάτω θα αναφερθούν μερικές πιθανές επεκτάσεις και βελτιώσεις που θα ανέβαζαν ακόμα περισσότερο την ποιότητα και την επεκτασιμότητα του Medbook.

6.2 Μελλοντικές επεκτάσεις

Η υλοποίηση της παρούσας διπλωματικής εργασίας αποτελεί μία εφαρμογή για την επίδειξη των δυνατοτήτων της αρχιτεκτονικής των microservice σε ένα περίπλοκο πληροφοριακό σύστημα, στην περίπτωση μας ένα σύστημα υγείας. Για να φτάσει όμως ένα τέτοιο σύστημα να μπορεί να ανταπεξέλθει στις πραγματικές συνθήκες του έξω κόσμου θα πρέπει να γίνουν μερικές επεκτάσεις στο σύστημα που ως τώρα σχεδιάστηκε.

Για τις επεκτάσεις αυτές ανανεώθηκε το υπάρχον σχεδιάγραμμα που είχαμε φτιάξει το οποίο παρουσίαζε την αρχιτεκτονική του συστήματος. Όπως φαίνεται στο νέο σχεδιάγραμμα από κάτω προστέθηκαν μερικά νέα microservices αλλά και κάποιες νέες τεχνολογίες λογισμικού οι οποίες θα μας βοηθήσουν στην ανάπτυξη του συστήματος, ας μιλήσουμε για κάθε ένα από αυτά αναλυτικά.



Ξεκινώντας βλέπουμε πως σε ένα από τα αρχικά επίπεδα του συστήματος μας έχει προστεθεί ένας Load Balancer. Πρόκειται για ένα σύστημα εξισορρόπησης του φόρτου εργασίας ενός συστήματος, είναι υπεύθυνο για την αποτελεσματική ίση κατανομή των εισερχόμενων Http Request στην ομάδα διακομιστών υποστήριξης που είναι γνωστή και ως σύμπλεγμα διακομιστών (server farm). Με πιο απλά λόγια θα χωρίσει ισότιμα τις εισερχόμενες κλήσεις προς όλα τα διαθέσιμα ClinicalInformation.API , στην περίπτωση μας έχουμε δύο αλλά θα μπορούσαμε να έχουμε ακόμα περισσότερα εξαρτάται από τα πόσα Requests περιμένουμε να έχουμε κατά μέσο όρο ανά ημέρα. Με την συγκεκριμένη επέκταση λοιπόν δεν χρειάζεται να γίνει αναβάθμιση των μηχανημάτων για την υποστήριξη και την εξυπηρέτηση των χιλιάδων μηνυμάτων το οποίο θα είχε και πολύ κόστος από άποψη Infrastructure (Server) αλλά θα δημιουργούσαμε πολλαπλές οντότητες του ίδιου microservice στις οποίες τα μηνύματα θα εξισορροπούνταν μέσω του Load Balancer (nginx).

Αφού αναφέραμε τον βασικό τρόπο ο οποίος βοηθάει στην επεκτασιμότητα του συστήματος μας θα αναφέρουμε ένα τρόπο που θα βοηθήσει ακόμα περισσότερο στην γρηγορότερη εξυπηρέτηση των λειτουργιών του συστήματος χωρίς την έξτρα επιβάρυνση στην μνήμη του μηχανήματος που ένα microservice ζει. Συχνά στα microservices για να μειωθεί όσο το περισσότερο η συχνή επικοινωνία με άλλα microservices αποθηκεύουμε κάποια πράγματα στην μνήμη (Caching) όμως ο συγκεκριμένος τρόπος έχει κάποια όρια. Για αυτόν τον λόγω όμως έχει δημιουργηθεί η Redis (Remote Dictionary Server), η Redis λοιπόν είναι μία open source βιβλιοθήκη η οποία αποθηκεύει δεδομένα στην μνήμη και πιο συγκεκριμένα στην δικιά της κατανεμημένη βάση δεδομένων. Η Redis υποστηρίζει διαφορετικού τρόπους αποθήκευσης δεδομένων όπως λίστες, χάρτες, σύνολα. Όπως βλέπουμε και στην νέα αρχιτεκτονική από πάνω τα Lis, Ris και Pharmacy microservice θα ήταν αυτά τα οποία θα αποθήκευαν πλέον δεδομένα στην Redis και όχι στην μνήμη.

Τα τρία αυτά microservice θα έχουν αρκετό φορτίο να διαχειριστούν για αυτό θα είναι πάρα πολύ σημαντικό ,μερικό από αυτό το φορτίο δεδομένων να αποθηκεύεται στην Redis ώστε να μην χρειάζεται το microservice να διαβάζει συνέχεια από την βάση. Σαφέστατα το να διαβάζει κάποιος από ένα key-value Database είναι πολύ πιο γρήγορο από το να τρέξει ένα περίπλοκο Query στην βάση.

Έπειτα άλλη μία νέα βιβλιοθήκη που προστέθηκε και φαίνεται και στο παραπάνω διάγραμμα είναι

ένα σύστημα δημοσιοποίησης και απορρόφησης μηνυμάτων. Η συγκεκριμένη βιβλιοθήκη στην περίπτωση μας είναι το RabbitMQ, το οποίο εφαρμόζει ένα Advanced Message Queuing Protocol με πιο απλά λόγια μέσω του RabbitMQ μπορούμε να δημιουργήσουμε ουρές(FIFO) στις οποίες ένα microservice θα κάνει Publish μηνύματα και ένα άλλο θα τα κάνει Consume. Είναι ένας τρόπος να εκτελέσουμε κάποια δευτερεύοντα σενάρια και χωρίς την χρήση Http Request. Μία τέτοια επικοινωνία στο σύστημα Medbook θα ήταν λογική μεταξύ των User, Patient, Personnel microservice όπως φαίνεται και στο διάγραμμα παραπάνω. Ένα παράδειγμα είναι ότι στην δημιουργία ενός νέου ασθενή αντί να γίνονται εσωτερικές κλήσεις για την δημιουργία λογαριασμού και την αποστολή email στην διεύθυνση του όλη αυτή η λογική να υλοποιηθεί μέσω μηνυμάτων μέσα σε μία ουρά, γλιτώνοντας έτσι μια περίπλοκη υλοποίηση που ίσως στο μέλλον θα χρειαζόταν να προστεθούν επιπλέον κλήσεις.

Τέλος μία τελευταία μελλοντική επέκταση θα ήταν η ακόμα μεγαλύτερη διαίρεση της αρχιτεκτονικής του Medbook σε ακόμα περισσότερα microservices και για την διάσπαση της λογική σε πιο συγκεκριμένα συστήματα αλλά και για την επεκτασιμότητα. Μερικά από αυτά τα microservices θα μπορούσαν να είναι το Ris.API (Radiology Information System) , το οποίο είναι ένα σύστημα πληροφοριών ακτινολογίας και χρησιμοποιείται για την εισαγωγή παραγγελιών, τον προγραμματισμό και την διαχείριση της λίστας εργασιών σε ένα ακτινολογικό τμήμα. Ένα άλλο θα ήταν το Pharmacy microservice το οποίο θα διαχειρίζεται το απόθεμα των φαρμάκων στο νοσοκομείο, τις παραγγελίες και την παράδοση των φαρμάκων στους εσωτερικούς ασθενείς του νοσοκομείου.

Βιβλιογραφία

R. Hill, D. Shadija, and M. Rezai. Enabling community health care with mi-croservices. In Ubiquitous Computing and Communications (ISPA/IUCC), 2017

Mulesoft. Microservices vs Monolithic Architecture, 2018. Ανακτήθηκε https://www.mulesoft.com/resources/api/microservices-vs-monolithic.

N. Patel. Five things to know about the future of Microservices and IoT, 2017. [Blog] Διαθέσιμο στο https://bit.ly/2AYrQ9P.

Redhat. What are microservices?, 2017. Ανακτήθηκε από https://www.redhat.com/en/topics/microservices/what-are-microservices.

C. Richardson. Pattern: Microservice Architecture, 2017. Ανακτήθηκε από https://microservices.io/patterns/microservices.html.

N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Musta_n, and L. Sana. Microservices: yesterday, today, and tomorrow. In Present and Ulterior Software Engineering.

Eric Evans, 2003. Domain Driven Design: Tackling Complexity in the Heart of Software.

Mittal, A.(2013) Understanding Multilayered Architecture in .Net. Ανακτήθηκε 21 Σεπτεμβρίου 2020 από https://codeteddy.com/2013/04/01/understanding-multilayered-architecture-in-net/.

Microsoft (2019). Common web application architectures. Ανακτήθηκε από https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures.

Microsoft (2019). Command and Query Responsibility Segregation (CQRS) pattern. Ανακτήθηκε από https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs.

Microsoft (2018). Design a DDD-oriented microservice. Ανακτήθηκε από https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/ddd-oriented-microservice.

Pablo Pazos Gutierrez 2019. Microservice architectures and open platforms for health care information systems. Ανακτήθηκε από https://cabolabs.com/blog/article/microservice_architectures_and_open_platforms_for_health_car_information_systems-5cab9ab60d88a.html

Gary Scialdone, 2002. Information Technologies for the Healthcare Delivery System. Ανακτήθηκε από https://scholarworks.rit.edu/cgi/viewcontent.cgi?referer=https://www.google.com/&httpsredir=1&article=8573&context=theses

Aloni Shanne Black, 2018. E-Health As A Service: A Service Based Design Approach for Large Scale E-Health Architecture. Ανακτήθηκε από https://eprints.qut.edu.au/