ΠΑΝΕΠΙΣΤΗΜΙΟ ΜΑΚΕΔΟΝΙΑΣ
ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΤΜΗΜΑΤΟΣ ΕΦΑΡΜΟΣΜΕΝΗΣ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΣΧΕΔΙΑΣΗ ΚΑΙ ΥΛΟΠΟΙΗΣΗ ΣΥΣΤΗΜΑΤΟΣ ΜΙΚΡΟΫΠΗΡΕΣΙΩΝ

Διπλωματική Εργασία

του

Δημητρακόπουλου Δημήτριου

Θεσσαλονίκη, Νοέμβριος 2022

ΣΧΕΔΙΑΣΗ ΚΑΙ ΥΛΟΠΟΙΗΣΗ ΣΥΣΤΗΜΑΤΟΣ ΜΙΚΡΟΫΠΗΡΕΣΙΩΝ

Δημητρακόπουλος Δημήτριος

Πτυχίο Μηχανικών Πληροφορικής Τ.Ε., ΤΕΙ Δυτικής Μακεδονίας, 2017

Διπλωματική Εργασία

υποβαλλόμενη για τη μερική εκπλήρωση των απαιτήσεων του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΤΙΤΛΟΥ ΣΠΟΥΔΩΝ ΣΤΗΝ ΕΦΑΡΜΟΣΜΕΝΗ ΠΛΗΡΟΦΟΡΙΚΗ

Επιβλέπων Καθηγητής
Μαργαρίτης Κωνσταντίνος

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 11/11/2022

Μαργαρίτης Κωνσταντίνος	Κασκάλης Θεόδωρος	Σακελλαρίου Ηλίας
....	....	....

Δημητρακόπουλος Δημήτριος

....

Περίληψη

Όλες οι εφαρμογές που διακρίνονται για την πολύ υψηλή επισκεψιμότητα τους έχουν υιοθετήσει την έννοια των μικροϋπηρεσιών στην υλοποίηση των προϊόντων τους. Η επιλογής της συγκεκριμένης αρχιτεκτονικής δεν είναι τυχαία, καθώς κατά αυτόν τον τρόπο επιτυγχάνεται η οριζόντια κλιμάκωση, η κλιμάκωση ανάλογα τις ανάγκες της κάθε μικροϋπηρεσίας, η ικανότητα να χρησιμοποιήσουμε διαφορετικά πλαίσια και γλώσσες προγραμματισμού ανάλογα με την ανάγκη της κάθε μικροϋπηρεσίας, όπως επίσης και τον διαχωρισμό του προσωπικού δυναμικού σε ανεξάρτητες ομάδες ανάπτυξης.

Η συγκεκριμένη εργασία δίνει έμφαση στην σχεδίαση και υλοποίηση ενός συστήματος μικροϋπηρεσιών. Οι υπηρεσίες επικοινωνούν μεταξύ τους μέσω κανάλι μηνυμάτων, GRPC και REST APIs. Ως κανάλι μηνυμάτων χρησιμοποιήθηκε η λύση της RabbitMQ. Η κάθε υπηρεσία έχει υλοποιηθεί με την χρήση του προγραμματιστικού πλαισίου .Net Core και τη γλώσσα προγραμματισμού C#. Επίσης η κάθε υπηρεσία ακολουθεί τη σχεδίαση βάσει τομέα/Domain-Driven Design ως μέθοδο υλοποίησης. Τέλος η κάθε υπηρεσία έχει τη δική της ανεξάρτητη SQL Server βάση δεδομένων.

Λέξεις Κλειδιά: Microservices, Domain-Driven Design, .Net Core, SQL Server, RabbitMQ, GRPC, REST API

Abstract

All applications that stand out for their very high traffic have adopted the concept of microservices architecture on their products. The choice of the specific architecture achieves horizontal scaling, scaling according to the needs of each microservice, the ability to use different frameworks and programming languages according to the needs of each microservice, as well as separation of staff in independent development teams.

This particular work emphasizes the design and implementation of a microservices system. Services communicate with each other via message channels, GRPC and REST APIs. The RabbitMQ solution was used as the message channel. Each service has been implemented using the .Net Core programming framework and the C# programming language. Also, each service follows Domain-Driven Design as an implementation method. Finally, each service has its own independent SQL Server database.

Keywords: Microservices, Domain-Driven Design, .Net Core, SQL Server, RabbitMQ, GRPC, REST API

Πρόλογος – Ευχαριστίες

Πρωτίστως θα ήθελα να δηλώσω τις βαθιές μου ευχαριστίες στον κ. Μαργαρίτη για τις πολύτιμες παρατηρήσεις και παροτρύνσεις περί του θέματος που θα διαπραγματευόταν η συγκεκριμένη εργασία. Την πολύτιμη βοήθεια που μου μετέδωσε, την αμέριστη προθυμία του να με βοηθήσει και κυρίως τη κατανόηση και υπομονή που έδειξε κατά τη διάρκεια της.

Ύστερα θα ήθελα να ευχαριστήσω όλους του καθηγητές του Πανεπιστημίου Μακεδονίας για τις γνώσεις που μου χάρισαν κατά τη διάρκεια των σπουδών μου.

Τέλος θα ήθελα να εκφράσω την βαθιά μου αγάπη και τις ταπεινές ευχαριστίες μου στους γονείς μου. Ότι έχω καταφέρει, και ότι θα καταφέρω στην πορεία της ζωής μου το χρωστάω κατά ένα πολύ μεγάλο ποσοστό σε αυτούς. Ήταν πάντοτε δίπλα μου, στυλοβάτες και υποστηρικτές σε όλα τα βήματα της προσωπικής και ακαδημαϊκής ζωής μου και τους διαβεβαιώνω ότι δεν θα το ξεχάσω ποτέ.

Περιεχόμενα

1 Εισαγωγή 1
1.1 Πρόβλημα – Σημαντικότητα του θέματος 1
1.2 Σκοπός – Στόχοι 1
1.3 Διάρθρωση της μελέτης 1
2 Θεωρητικό Υπόβαθρο 2
2.1 Αρχιτεκτονική Μικροϋπηρεσίας 2
2.1.1 Σχεδιασμός βάσει τομέα – Domain Driven Design 2
2.1.2 DDD Mind Map 4
2.1.3 Κύρια συστατικά του Mind Map 4
2.2 Αρχιτεκτονική συστήματος Μικροϋπηρεσιών 14
2.2.1 Πύλη API (API Gateway) 14
2.2.2 Ξεχωριστή βάση δεδομένων ανά μικροϋπηρεσία 15
2.2.3 Event driven – Publish/Subscribe 16
2.2.4 CQRS – Command and Query Responsibility Segregation 17
2.2.5 Μοτίβο διαμεσολαβητή – Mediator Pattern 21
2.3 Τεχνολογίες 22
2.3.1 HTTP 22
2.3.2 REST API 23
2.3.3 GRPC 24
2.3.4 SQL Server 25
2.3.5 RabbitMQ 25
2.3.6 Docker 25
3 Σχεδίαση 26
3.1 Καταγραφή απαιτήσεων 26
3.2 Αρχιτεκτονική συστήματος 27
3.2.1 Πύλη API (API Gateway) 27
3.2.2 Μικροϋπηρεσία Persons 28
3.2.3 Μικροϋπηρεσία PeopleRelations 31
4 Υλοποίηση 37
4.1 Υπηρεσίες συστήματος 37
4.1.1 HTTP API Gateway 37
4.1.2 Μικροϋπηρεσία Persons 45
4.1.3 Μικροϋπηρεσία PeopleRelations 46
4.2 Διαγράμματα ροής 47
4.2.1 Προσθήκη ατόμου στο σύστημα 47
4.2.2 Αποστολής αιτήματος φιλίας 51
4.2.3 Αποδοχή αιτήματος φιλίας και δημιουργία φιλίας ανάμεσα στους χρήστες 53
5 Επίλογος 56
5.1 Σύνοψη και συμπεράσματα 56
5.2 Όρια και περιορισμοί της έρευνας 56
5.3 Μελλοντικές επεκτάσεις 57

Κατάλογος Εικόνων

Εικόνα 2-1 : DDD Mind Map ... 4
Εικόνα 2-2 : DDD Mind Map ... 4
Εικόνα 2-3 : Οι μεταβλητές τις οντότητας FriendRequest ... 5
Εικόνα 2-4 : Οι μέθοδοι της οντότητας FriendRequest ... 6
Εικόνα 2-5 : Τα σύνολα της υπηρεσίας PeopleRelations, FriendRequest και Person ... 7
Εικόνα 2-6: Απόσπασμα από το PersonAggregate ... 8
Εικόνα 2-7: Τα bounded contexts της υλοποίησης μας ... 9
Εικόνα 2-8: Ορισμός από τον Martin Fowler ... 10
Εικόνα 2-9 : Διάγραμμα ροής των λειτουργιών αποστολής και αποδοχής αιτήματος φιλίας ... 10
Εικόνα 2-10: Δημιουργία integration event κατά τη δημιουργία ατόμου στην εφαρμογή ... 11
Εικόνα 2-11: Το PersonRepository της υλοποίησης μας ... 12
Εικόνα 2-12: Τα τρία επίπεδα στην μικροϋπηρεσία Persons ... 13
Εικόνα 2-13: Εξαρτήσεις ανάμεσα στα επίπεδα ... 14
Εικόνα 2-14 : Πύλη API ... 15
Εικόνα 2-15 : Διάγραμμα βάσης δεδομένων ... 16
Εικόνα 2-16 : Κανάλι μηνυμάτων ... 17
Εικόνα 2-17: Υλοποίηση Στοίβας ... 18
Εικόνα 2-18 : CQRS και CQS ... 18
Εικόνα 2-19 : Διαφορές CQS και CQRS ... 19
Εικόνα 2-20 : Συχνότητα λειτουργιών της εφαρμογής ... 20
Εικόνα 2-21 : Τεχνικές βελτιστοποίησης των αναγνώσεων σε ένα σύστημα ... 21
Εικόνα 2-22 : Σχέσεις μεταξύ των στοιχείων του συστήματος χωρίς Mediator ... 21
Εικόνα 2-23 : Σχέσεις μεταξύ των στοιχείων του συστήματος με Mediator ... 22
Εικόνα 3-1: Αρχιτεκτονική της πιλοτικής εφαρμογής ... 27
Εικόνα 3-3-2: Post /Persons ... 29
Εικόνα 3-3-3: Get /Persons ... 30
Εικόνα 3-3-4: Οι οντότητες και τα σύνολα της μικροϋπηρεσίας Persons ... 30
Εικόνα 3-3-5: Διάγραμμα βάσης δεδομένων της μικροϋπηρεσίας Persons ... 31
Εικόνα 3-3-6: API της μικροϋπηρεσίας PeopleRelations ... 32
Εικόνα 3-3-7: Η Get /Persons ... 33
Εικόνα 3-3-8: Η Post /FriendRequests ... 33
Εικόνα 3-3-9: Η Get /FriendRequests ... 34
Εικόνα 3-3-10: Η Put /FriendRequests ... 35
Εικόνα 3-3-11: Οντότητες και σύνολα της μικροϋπηρεσίας PeopleRelations ... 35
Εικόνα 3-3-12: Διάγραμμα βάσης δεδομένων της μικροϋπηρεσίας PeopleRelations ... 36

1 Εισαγωγή

1.1 Πρόβλημα – Σημαντικότητα του θέματος

Τα τελευταία χρόνια το μέγεθος, η επισκεψιμότητα και η πολυπλοκότητα των εφαρμογών αυξάνεται. Οι ανάγκες για συνεχείς αλλαγές, υψηλή διαθεσιμότητα και οριζόντια κλιμάκωση δεν μπορούν να καλυφθούν από τη μονολιθική προσέγγιση. Ως ακόλουθο των παραπάνω, η αρχιτεκτονική μικροϋπηρεσιών κερδίζει συνεχώς έδαφος. Ένα πλήθος αυτόνομων μικροϋπηρεσιών, οι οποίες δουλεύουν παράλληλα και κατανεμημένα για την επίτευξη ενός συνολικού σκοπού, δίνουν την δυνατότητα για συνεχείς αλλαγές, κλιμάκωση ανάλογα τις ανάγκες της κάθε μικροϋπηρεσίας, οριζόντια κλιμάκωση, καθιστούν τη συγκεκριμένη αρχιτεκτονική τη πλέον ενδεδειγμένη για εφαρμογές που διακρίνονται για την υψηλή τους επισκεψιμότητα.

Ωστόσο η αρχιτεκτονική μικροϋπηρεσιών έρχεται με πολλές προκλήσεις και η υλοποίηση της θα πρέπει να σχεδιαστεί με τέτοιο τρόπο ώστε να επιτρέπει τα οφέλη που παρατηρήσαμε.

1.2 Σκοπός – Στόχοι

Σκοπός της παρούσας διπλωματικής είναι η σχεδίαση και υλοποίηση ενός συστήματος που χρησιμοποιεί μοντέρνες μεθοδολογίες υλοποίησης και απαρτίζεται από αυτόνομες μικροϋπηρεσίες, οι οποίες με τη σειρά τους αναπτύσσονται και διατηρούνται ανεξάρτητα.

Θα εξερευνήσουμε την θεωρία και τις βέλτιστες πρακτικές με σκοπό να σχεδιάσουμε και να εφαρμόσουμε ένα σύστημα μικροϋπηρεσιών που θα αντιμετωπίζει όλες τις προκλήσεις της συγκεκριμένης αρχιτεκτονικής και θα μπορεί να θεωρηθεί έτοιμη για ένα υψηλής επισκεψιμότητας παραγωγικό περιβάλλον.

1.3 Διάρθρωση της μελέτης

Το υπόλοιπο της διπλωματικής αποτελείται από τέσσερα κεφάλαια.

Το κεφάλαιο 2 αποτελεί την ανάλυση του θεωρητικού και τεχνολογικού υποβάθρου που είναι αναγκαίο για τη παρακολούθηση της εργασίας.

Το κεφάλαιο 3 εστιάζει στην καταγραφή των απαιτήσεων, στην αρχιτεκτονική του συστήματος και της κάθε μικροϋπηρεσίας.

Το κεφάλαιο 4 επικεντρώνεται στην υλοποίηση του συστήματος και της κάθε μικροϋπηρεσίας ξεχωριστά.

Τέλος, στο κεφάλαιο 5 σημειώνονται τα συμπεράσματα της ανάπτυξης και τυχόν μελλοντικές επεκτάσεις.

2 Θεωρητικό Υπόβαθρο

2.1 Αρχιτεκτονική Μικροϋπηρεσίας

2.1.1 Σχεδιασμός βάσει τομέα – Domain Driven Design

Ο όρος σχεδιασμός βάσει τομέα (DDD) επινοήθηκε από το Eric Evans στο ομότιτλο βιβλίο του που δημοσιεύθηκε το 2003 [1]. Σύμφωνα με τον Martin Fowler [1], έναν εκ των κορυφαίων προγραμματιστών που συνεισφέραν στην διάδοση του, η συγκεκριμένη προσέγγιση στην συγγραφή λογισμικού επικεντρώνεται στην ανάπτυξη ενός προγραμματιστικού μοντέλου που κατανοεί πλήρως τις διαδικασίες και τους κανόνες ενός λογικού τομέα.

Δύο βασικοί όροι που θα μας βοηθήσουν να κατανοήσουμε το συγκεκριμένο μοντέλο:

- **Τομέας (Domain):** Ένας τομέας είναι η σφαίρα γνώσης και δραστηριότητας από την οποία περιστρέφεται η λογική της εφαρμογής.
- **Μοντέλο (Model):** Ένα σύστημα αφαιρέσεων που περιγράφει επιλεγμένες πτυχές ενός τομέα και μπορεί να χρησιμοποιηθεί για την επίλυση προβλημάτων που σχετίζονται με τον συγκεκριμένο τομέα.

Ένα από τα βασικά χαρακτηριστικά του DDD αποτελεί η επικοινωνία των προγραμματιστών με τους ειδικούς του εκάστοτε τομέα. Για το λόγο αυτό, απαιτείται η συγγραφή ενός λεξιλογίου που αποτελείται από επιχειρηματικούς όρους και την απλούστερη επεξήγηση τους. Το συγκεκριμένο λεξιλόγιο αποτελεί και αυτό ένα από τα βασικά συστατικά του DDD και ονομάζεται Ubiquitous Language.

Πλεονεκτήματα του Σχεδιασμού βάσει τομέα (DDD)

Τα 6 βασικά πλεονεκτήματα του DDD

[1] Martin Fowler (software engineer) - Wikipedia

- **Ευελιξία (Flexible):** Επειδή το DDD εστιάζει στην ανάπτυξη μικρών, μεμονωμένων και σχεδόν αυτόνομων κομματιών του τομέα, οι διαδικασίες και το λογισμικό που προκύπτει είναι ευέλικτο. Η μετακίνηση ή τροποποίηση κομματιών του λογισμικού μπορεί ναι γίνει εύκολα, με ελάχιστες ή μηδαμινές παρενέργειές. Επιπλέον επιτρέπει την ευελιξία με τους πόρους του έργου καθ' όλη την διάρκεια της ανάπτυξης.
- **Όραμα/Προοπτική πελάτη για τη επίλυση του προβλήματος**: Η στενή συνεργασία με τους ειδικούς του τομέα σε όλη τη διάρκεια ανάπτυξης, έχει ως αποτέλεσμα το λογισμικό που προκύπτει να κατανοεί και να λύνει τα προβλήματα προς επίλυση που έθεσε ο εκάστοτε πελάτης.
- **Καθαρή διαδρομή μέσα σε ένα περίπλοκο πρόβλημα:** Το DDD δίνει μια σαφή και διαχειρίσιμη διαδρομή σε ένα περίπλοκο πρόβλημα. Οι απαιτήσεις εξηγούνται πάντα από την οπτική γωνία του λογικού τομέα.
- **Καλά οργανωμένος και εύκολα ελεγμένος κώδικας:** Η δομή που απαιτείται για την ανάπτυξη μέσω του παραδείγματος του DDD, έχει ως αποτέλεσμα ο κώδικας να διαχωρίζεται σε ξεκάθαρα επίπεδα ως προς τη λογική και την ύπαρξη τους. Να θεωρείται καλογραμμένος και να είναι εύκολα ελέγξιμος, καθώς διευκολύνει την ύπαρξή πολλών unit tests.
- **Η επιχειρηματική λογική ζει σε ένα μέρος:** Η επιχειρηματική λογική ζει σε ένα μέρος, συγκεκριμένα στο επίπεδο Τομέα.
- **Βελτιωμένα μοτίβα:** Το DDD προσφέρει τις αρχές και τα πρότυπα για την επίλυση δύσκολων προβλημάτων στο λογισμικό.

Προβληματισμοί ως προς την χρήση του Σχεδιασμού βάσει τομέα (DDD)

Ενώ ο σχεδιασμός βάσει τομέα παρέχει πολλά τεχνικά πλεονεκτήματα, όπως η εκτεταμένη δυνατότητα συντήρησης, θα πρέπει να εφαρμόζεται μόνο σε σύνθετους τομείς.

Το παραπάνω απόσπασμα από το βιβλίο του Eric Evans [1], δηλώνει ρητά ότι το DDD δεν είναι κατάλληλο για προβλήματα όταν υπάρχει σημαντική τεχνική πολυπλοκότητα, αλλά μικρή πολυπλοκότητα επιχειρηματικού τομέα. Η χρήση του DDD είναι ωφέλιμη όταν η πολυπλοκότητα του τομέα καθιστά δύσκολο για τους ειδικούς του τομέα να επικοινωνήσουν τις ανάγκες τους στους προγραμματιστές λογισμικού. Επενδύοντας τον χρόνο και τη προσπάθεια στη μοντελοποίηση του τομέα και καταλήγοντας σε ένα σύνολο ορολογίας(Ubiquitus language), η διαδικασία κατανόησης και επίλυσης του προβλήματος γίνεται πολύ πιο απλή και ομαλή.

2.1.2 DDD Mind Map

Εικόνα 2-1 [2]: DDD Mind Map

Εικόνα 2-2 [2]: DDD Mind Map

2.1.3 Κύρια συστατικά του Mind Map

Για να κατανοήσουμε καλύτερα τα βασικά συστατικά του DDD, θα τα παρουσιάσουμε με τη βοήθεια της υλοποίησης της εφαρμογής μας. Περιληπτικά, καθώς θα αναλυθεί εκτενέστερα στα κεφάλαια 3 και 4, η εφαρμογή μας αποτελείται από δύο

[2] Πηγή: Tackling Complexity in the Heart of DDD (vladikk.com)

υπηρεσίες. Την υπηρεσία Persons, που διαχειρίζεται την εγγραφή των χρηστών στο σύστημα και την υπηρεσία PeopleRelations (PR) που δημιουργεί αιτήματα φιλίας και φιλίες μεταξύ των χρηστών.

Οντότητα(Entity): Μια οντότητα τομέα στο DDD εφαρμόζει τη λογική ή τη συμπεριφορά τομέα που σχετίζεται με τα δεδομένα της οντότητας. Για παράδειγμα, στην εφαρμογή μας, ως μέρος μιας κλάσης οντότητας ενός αιτήματος φιλίας, πρέπει να περιέχεται όλη την επιχειρηματική λογική και οι λειτουργίες, ως μέθοδοι για τις εργασίες όπως την αποδοχή ενός αιτήματος. Οι μέθοδοι της οντότητας φροντίζουν για τις μεταβλητές και τους κανόνες της οντότητας, αντί αυτή η επιχειρηματική λογική να είναι διασκορπισμένη στο συνολικό επίπεδο της εφαρμογής.

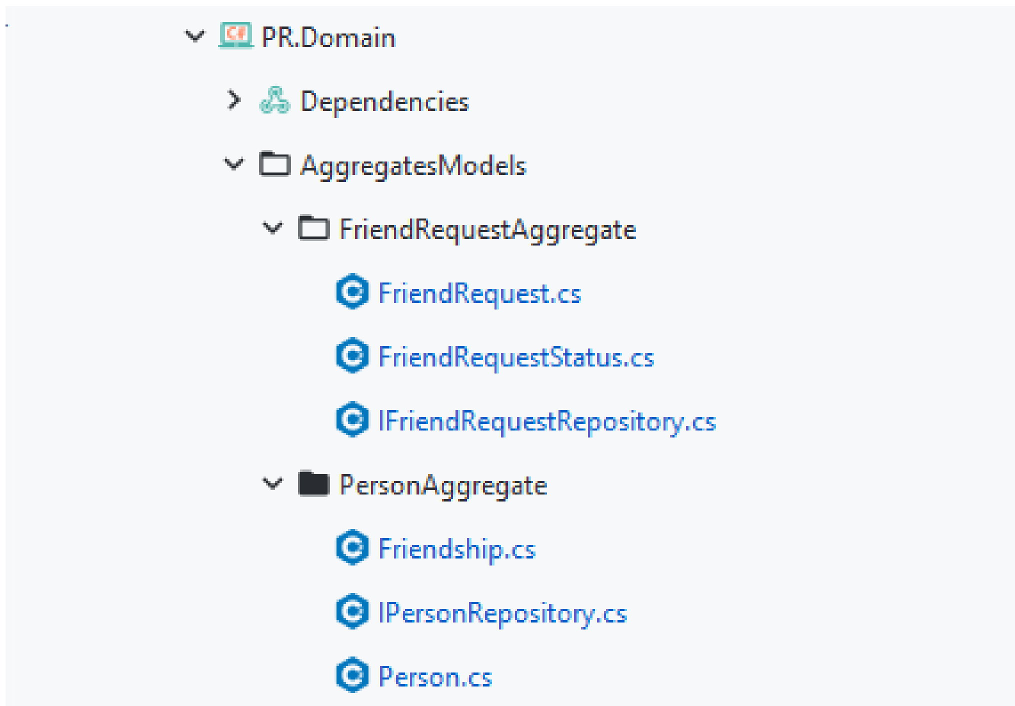
```
21 usages   DimitrisDimitrako +1   5 exposing APIs
public class FriendRequest : Entity, IAggregateRoot
{
    DimitrisDimitrako
    public FriendRequest()
    {
    }

    8 usages
    public int SenderPersonId { get; }
    7 usages
    public int ReceiverPersonId { get; }
    3 usages
    public DateTimeOffset CreatedDate { get; }
    3 usages
    public int Modifier { get; }
    3 usages
    public DateTimeOffset? ModifiedDate { get; }
    2 usages
    public FriendRequestStatus FriendRequestStatus { get; private set; }
    private int _friendRequestStatusId;
```

Εικόνα 2-3 : Οι μεταβλητές τις οντότητας FriendRequest

```
1 usage   DimitrisDimitrako +1
public bool IsEqualTo(int senderId, int receiverId)
{
    return SenderPersonId == senderId
           && ReceiverPersonId == receiverId;
}

1 usage   DimitrisDimitrako
public void SetAcceptedFriendRequestStatus()
{
    if (_friendRequestStatusId != FriendRequestStatus.AwaitingConfirmation.Id)
    {
        StatusChangeException(FriendRequestStatus.Confirmed);
    }

    _friendRequestStatusId = FriendRequestStatus.Confirmed.Id;
    AddDomainEvent(new FriendRequestAcceptedDomainEvent(this));
}
```

Εικόνα 2-4 : Οι μέθοδοι της οντότητας FriendRequest

Σύνολο(Aggregate): Ένα σύμπλεγμα συσχετισμένων αντικειμένων και οντοτήτων που αντιμετωπίζονται ως μονάδα για τους σκοπούς των αλλαγών των δεδομένων. Με απλά λόγια, ένα αντικείμενο σύνολο αποτελείται από μεταβλητές, αντικείμενα και οντότητες.



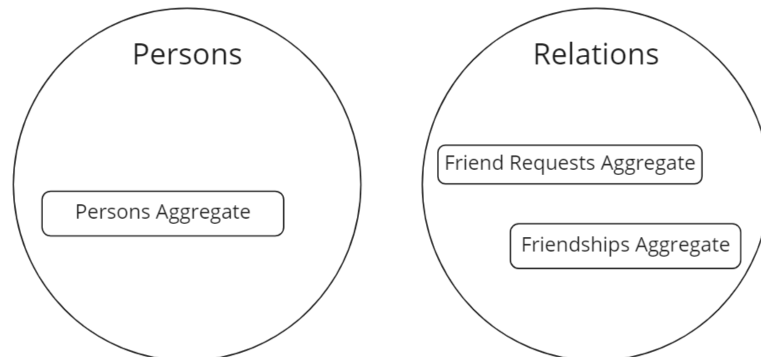
Εικόνα 2-5 : Τα σύνολα της υπηρεσίας PeopleRelations, FriendRequest και Person

Ένα χαρακτηριστικό παράδειγμα συνόλου στην εφαρμογή μας αποτελεί το PersonAggregate, το οποίο φιλοξενεί επίσης την οντότητα Friendship. Όπως παρατηρούμε στην εικόνα 2-5, το συγκεκριμένο σύνολο περιέχει επίσης την επιχειρηματική λογική της προσθήκης μιας φίλιας σε ένα άτομο, στην υλοποίηση της μεθόδου AddFriendship.

```
private int _id;
8 usages
public string IdentityGuid { get; }

private readonly List<Friendship> _friendshipsSent;
private readonly List<Friendship> _friendshipsReceived;

3 usages  DimitrisDimitrako
public IReadOnlyCollection<Friendship> FriendshipsSent => _friendshipsSent;
3 usages  DimitrisDimitrako
public IReadOnlyCollection<Friendship> FriendshipsReceived => _friendshipsReceived;

1 usage  DimitrisDimitrako +1
private Person()
{
    _friendshipsReceived = new List<Friendship>();
    _friendshipsSent = new List<Friendship>();
}

3 usages  DimitrisDimitrako +1
public Person(string identityGuid) : this()
{
    IdentityGuid = !string.IsNullOrEmpty(identityGuid)
        ? identityGuid
        : throw new ArgumentNullException(nameof(identityGuid));
}

1 usage  DimitrisDimitrako
public void AddFriendship(int senderGuid, int receiverGuid)
{
    var friendshipToAdd = new Friendship(senderGuid, receiverGuid);
    _friendshipsSent.Add(friendshipToAdd);
}
```

Εικόνα 2-6: Απόσπασμα από το PersonAggregate

Bounded Context: Το Bounded Context αποτελεί ένα κεντρικό μοτίβο στο DDD. Μπορεί να θεωρηθεί το επίκεντρο του τμήματος στρατηγικού σχεδιασμού του DDD, καθώς διαιρεί και συγκεντρώνει μεγάλα μοντέλα - σύνολα σε ομάδες. Η κάθε μία ομάδα αποτελεί ένα ενοποιημένο μοντέλο για τους σκοπούς των αλλαγών των δεδομένων, και προσπαθεί να επιλύσει ένα συγκεκριμένο πρόβλημα στο επιλεγμένο επιχειρηματικό τομέα. Συνήθως το κάθε bounded context αποτελεί μια ξεχωριστή μικροϋπηρεσία στο σύστημα[2].



Εικόνα 2-7: Τα bounded contexts της υλοποίησης μας

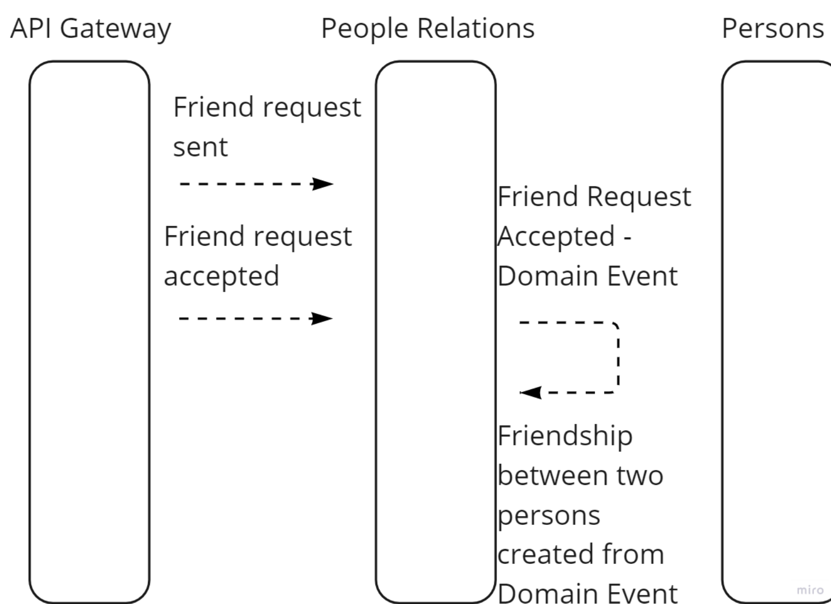
Στην υλοποίηση μας, υπάρχουν δυο bounded context, ακριβώς όσες και οι μικροϋπηρεσίες:

1. **Persons**: Το οποίο αποτελείται από το PersonsAggregate και περικλείει την λογική και τις λειτουργείες που είναι συνυφασμένες με την λογική οντότητα των ατόμων/χρηστών στο σύστημα μας.
2. **People Relations (PR):** Το οποίο αποτελείται από δύο σύνολα. Το Friend Requests, που αποτελεί τη λογική γύρω από τα αιτήματα φιλίας και το Friendships, το οποίο διαχειρίζεται τη σύνδεση μέσω φιλίας μεταξύ των ατόμων του συστήματος.

Domain Event: Τα domain events χρησιμοποιούνται για να επιτρέψουν στα σύνολα ενός συγκεκριμένου bounded context, να αντιδράσουν σε μία αλλαγή από άλλο σύνολο του ίδιου bounded context, χωρίς όμως να συνδέονται ρητά. Με το συγκεκριμένο τρόπο πετυχαίνουμε τη χαλαρή σύζευξη μεταξύ των συνόλων του ίδιου bounded context.

A Domain Event captures the memory of something interesting which affects the domain — Martin Fowler

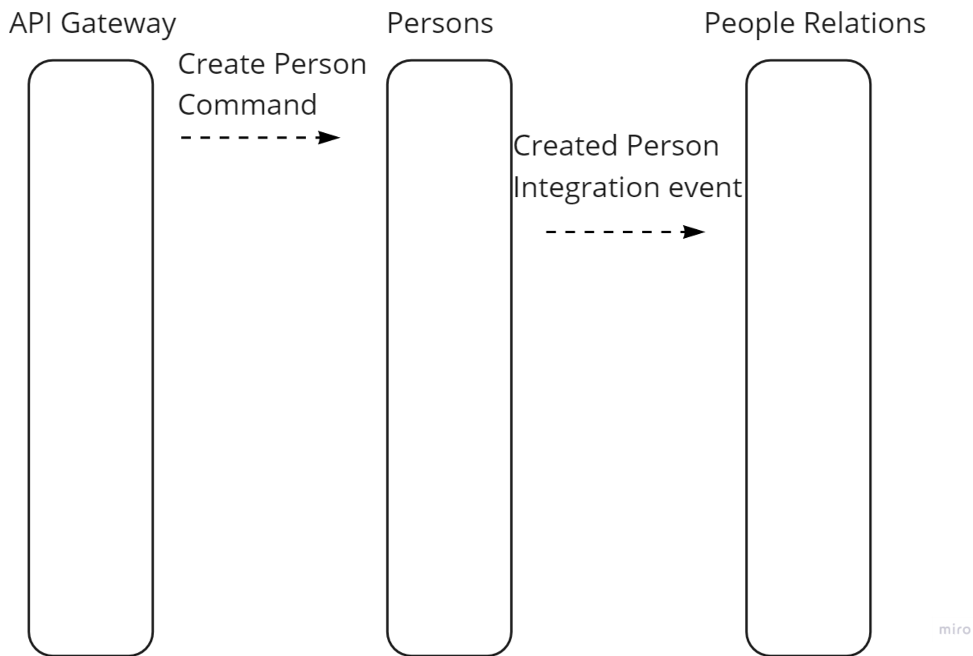
Εικόνα 2-8: Ορισμός από τον Martin Fowler



Εικόνα 2-9 : Διάγραμμα ροής των λειτουργιών αποστολής και αποδοχής αιτήματος φιλίας

Ένα χαρακτηριστικό παράδειγμα από την υλοποίηση της εφαρμογής μας, αποτελεί η λειτουργία της αποδοχής ενός αιτήματος φιλίας. Η λειτουργία της αποδοχής βρίσκεται στο PeopleRelations bounded context και στο σύνολο, FriendRequest. Μόλις εκτελεστεί, το σύστημα δημιουργεί το event FriendRequestAcceptedDomainEvent, που έχει ως στόχο να ενημερώσει κάποιο άλλο σύνολο στο ίδιο bounded context. Το FrienshipAggregate, μέσω του AddPersonFriendshipRequestAcceptedEventHandler, ενημερώνεται για το παραπάνω event, και με τη σειρά του δημιουργεί μια καινούργια οντότητα φιλίας στο σύστημα.

Integration Event: Τη διαφορά μεταξύ των integration events και των domain events αποτελεί ότι το integration event επιτρέπει την επικοινωνία διαφορετικών bounded context - μικροϋπηρεσιών μεταξύ τους, ενώ τα domain events επιτρέπουν την επικοινωνία των συνόλων ενός συγκεκριμένου bounded context.



Εικόνα 2-10: Δημιουργία integration event κατά τη δημιουργία ατόμου στην εφαρμογή

Στην υλοποίηση μας, το CreatedPersonIntegrationEvent δημιουργείται από το Persons bounded context καθώς μια οντότητα ατόμου έχει δημιουργηθεί. Στο bounded context PersonsRelations, υπάρχει ο PersonCreatedIntegrationEventHandler, που ενημερώνεται για καινούργια CreatedPersonEvents και με τη σειρά του εκτελεί κάποιες λειτουργίες που αφορούν το συγκεκριμένο λογικό τομέα.

Repository: Το repository στο DDD χρησιμοποιείται για πρόσβαση σε δεδομένα από μόνιμες πηγές αποθήκευσης, όπως οι βάσεις δεδομένων. Σε αντίθεση με πολλές βιβλιοθήκες που χρησιμοποιούνται για την επικοινωνία με τις βάσεις δεδομένων και χρησιμοποιούν τις γλώσσες προγραμματισμού των βάσεων, τα repositories χρησιμοποιούν μια μεταφορική έννοια των όρων.

Database	Repository
Save	Add
Delete	Remove
Load	Get/Find

```
10 usages  1 inheritor  DimitrisDimitrako
public interface IPersonRepository : IRepository<Person>
{
    1 usage  1 implementation  DimitrisDimitrako
    Person Add(Person person);
    2 usages  1 implementation  DimitrisDimitrako
    Task<Person?> FindAsync(string personIdentityGuid);
    1 usage  1 implementation  DimitrisDimitrako
    Task<Person?> FindWithFriendShipsNoTrackingAsync(string personIdentityGuid);
    3 usages  1 implementation  DimitrisDimitrako
    Task<Person?> FindByIdAsync(int id);
}
```

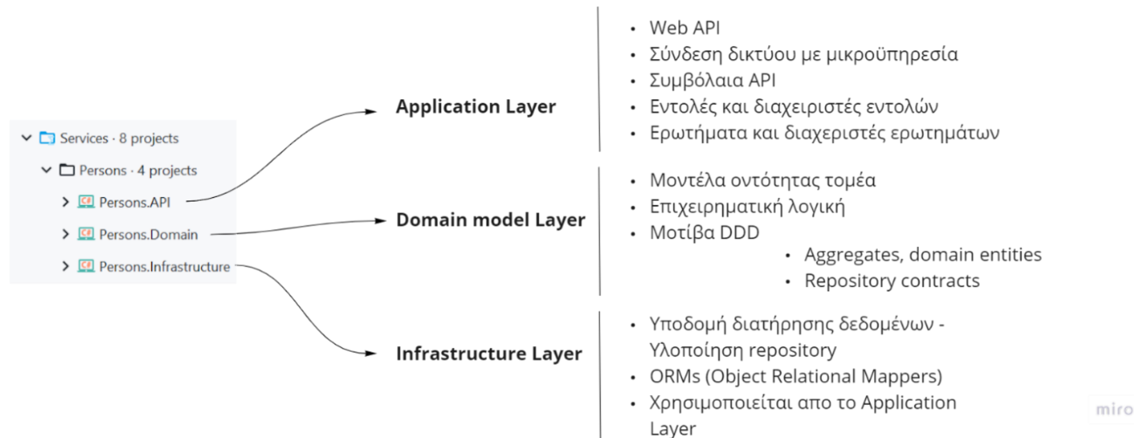
Εικόνα 2-11: Το PersonRepository της υλοποίησης μας

Δεν χρειαζόμαστε repository για κάθε οντότητα του συστήματος. Αντίθετα, τα repositories διαχειρίζονται ομάδες αντικειμένων, τα προαναφερθέντα Aggregates.

Επίπεδα(Layers): Οι περισσότερες εταιρικές εφαρμογές με σημαντική επιχειρηματική και τεχνική πολυπλοκότητα ορίζονται από πολλαπλά επίπεδα. Τα επίπεδα είναι ένα λογικό τεχνούργημα και δεν συσχετίζονται με την ανάπτυξη της υπηρεσίας. Υπάρχουν για να βοηθήσουν τους προγραμματιστές να διαχειριστούν την πολυπλοκότητα του κώδικα. Διαφορετικά επίπεδα (όπως το επίπεδο μοντέλου τομέα έναντι του επιπέδου παρουσίασης) μπορεί να έχουν διαφορετικές τύπους μοντέλων, οι οποίοι επιβάλλουν μετατροπές μεταξύ αυτών των τύπων.

Για παράδειγμα, μια οντότητα θα μπορούσε να φορτωθεί από τη βάση δεδομένων. Στη συνέχεια, μέρος αυτών των πληροφοριών μπορεί να σταλεί στη διεπαφή χρήστη του πελάτη μέσω ενός REST Web API. Το θέμα εδώ είναι ότι η οντότητα τομέα περιέχεται στο επίπεδο μοντέλου τομέα και δεν πρέπει να διαδίδεται σε άλλες περιοχές στις οποίες δεν ανήκει, όπως στο επίπεδο παρουσίασης. Επομένως οι οντότητες δεν θα πρέπει να δεσμεύονται σε προβολές πελατών. Αυτός είναι ο λόγος για το οποίο χρησιμεύουν τα ViewModels ή τα Data Transfer Objects(DTO). Το ViewModel είναι ένα μοντέλο δεδομένων αποκλειστικά για τις ανάγκες του επιπέδου παρουσίασης. Οι οντότητες τομέα δεν ανήκουν απευθείας στο ViewModel. Αντίθετα, πρέπει να γίνει μετατροπή μεταξύ των ViewModels και των οντοτήτων τομέα και αντίστροφα. Ενώ τα ViewModels αφορούν αποκλειστικά το επίπεδο παρουσίασης, τα DTOs μετατρέπουν τις οντότητες των άλλων επιπέδων στο εκάστοτε επίπεδο.

Για την αντιμετώπιση της πολυπλοκότητας, είναι σημαντικό το μοντέλο τομέα να ελέγχεται από ένα σύνολο, το οποίο διασφαλίζει ότι όλοι οι κανόνες που σχετίζονται με αυτήν την ομάδα οντοτήτων (Aggregate) εκτελούνται μέσω ενός σημείου εισόδου.



Εικόνα 2-12: Τα τρία επίπεδα στην μικροϋπηρεσία Persons

Επίπεδο μοντέλου τομέα(Domain Layer): Το επίπεδο μοντέλου τομέα είναι το σημείο που εκφράζεται η επιχείρηση. Είναι το επίπεδο όπου περιλαμβάνονται όλοι οι επιχειρηματικοί κανόνες που σχετίζονται με το πρόβλημα που πρέπει να επιλυθεί. Σε αυτό το επίπεδο εμπεριέχονται οι οντότητες και τα σύνολα, δηλαδή η επιχειρηματική λογική της εφαρμογής.

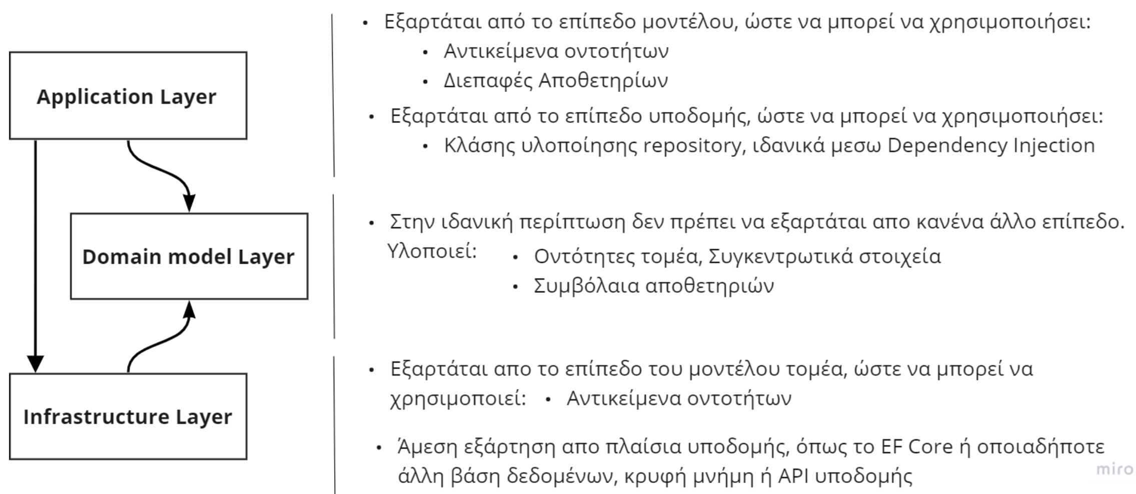
Το επίπεδο μοντέλου τομέα δεν θα πρέπει να έχει καμία άμεση εξάρτηση από οποιοδήποτε άλλο επίπεδο. Η λογική στο επίπεδο του μοντέλου τομέα, τα μοντέλα δεδομένων και οι επιχειρηματικοί κανόνες πρέπει να είναι εντελώς ανεξάρτητοι από τα επίπεδα παρουσίασης, υποδομής και εφαρμογής.

Επίπεδο εφαρμογής (Application Layer): Το επίπεδο εφαρμογής καθορίζει τις εργασίες που υποτίθεται ότι πρέπει να κάνει το λογισμικό και κατευθύνει τα αντικείμενα τομέα να επιλύσουν τα 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.

Δεν περιέχει επιχειρηματικούς κανόνες ή λογική, αλλά συντονίζει και αναθέτει εργασίες σε συνεργασία με τα αντικείμενα τομέα στο κάτω επίπεδο. Μπορεί να περιέχει καταστάσεις που αντικατοπτρίζουν την πρόοδο μιας εργασίας για το χρήστη αλλά

αντίθετα δεν περιέχει ποτέ καταστάσεις που αντιπροσωπεύουν την κατάσταση της επιχειρηματικής οντότητας/λογικής.

Επίπεδο υποδομής (Infrastructure Layer): Το επίπεδο υποδομής είναι το επίπεδο με το οποίο τα δεδομένα που διατηρούνται αρχικά στη μνήμη, αποθηκεύονται μόνιμα σε βάσεις δεδομένων ή σε άλλο μόνιμο χώρο αποθήκευσης. Όπως αναφέρθηκε προηγουμένως, οι κλάσεις οντοτήτων του μοντέλου τομέα πρέπει να παραμείνουν άγνωστες με την υποδομή που χρησιμοποιείται για τη διατήρηση των δεδομένων. Έτσι, τα υπόλοιπα επίπεδα θα πρέπει τελικά να εξαρτώνται από το επίπεδο μοντέλου τομέα και όχι το αντίστροφο, όπως φαίνεται στη παρακάτω εικόνα 2-13.



Εικόνα 2-13: Εξαρτήσεις ανάμεσα στα επίπεδα

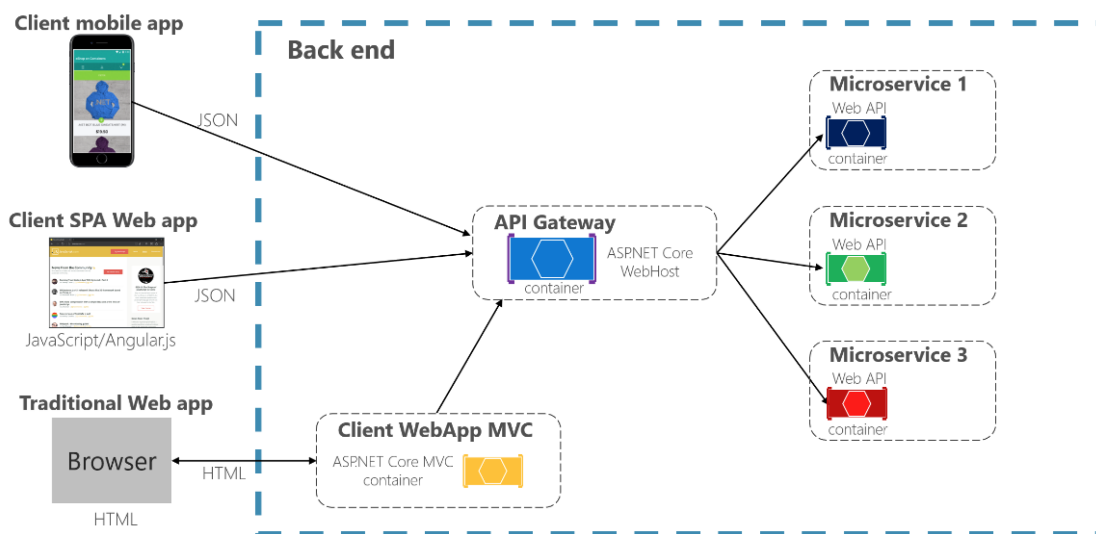
Το επίπεδο εφαρμογής εξαρτάται από το μοντέλο τομέα και την υποδομή, η υποδομή εξαρτάται από το μοντέλο τομέα, αλλά το μοντέλο τομέας δεν εξαρτάται από κανένα επίπεδο. Αυτός ο σχεδιασμός επιπέδων θα πρέπει να είναι ανεξάρτητος για κάθε μικροϋπηρεσία. [3][4][5]

2.2 Αρχιτεκτονική συστήματος Μικροϋπηρεσιών

2.2.1 Πύλη API (API Gateway)

Σε μία αρχιτεκτονική μικροϋπηρεσιών, οι εφαρμογές-πελάτες για να χρησιμοποιήσουν μια λειτουργικότητα του συστήματος, συνήθως χρειάζεται να

επικοινωνήσουν με παραπάνω από μία μικροϋπηρεσία. Ο χειρισμός τόσων πολλών τελικών σημείων από τις εφαρμογές-πελάτες μπορεί να αποδειχθεί εξαιρετικά πολύπλοκος. Το αρχιτεκτονικό πρότυπο σχεδίασης, API Gateway μπορεί να απλοποιήσει την παραπάνω πολύπλοκη κατάσταση. Αυτό το πρότυπο είναι μια υπηρεσία που παρέχει ένα σημείο πρόσβασης για ορισμένες ομάδες μικροϋπηρεσιών. Είναι παρόμοιο με το πρότυπο σχεδίασης Facade[3] στον αντικειμενοστραφή προγραμματισμό, αλλά σε αυτήν την περίπτωση είναι μέρος ενός κατανεμημένου συστήματος. Τέλος, πολλές φορές το συγκεκριμένο πρότυπο αποκαλείται το backend των frontend υπηρεσιών, καθώς αποτελεί για τα frontend συστήματα το μόνο τρόπο επικοινωνίας με το εκάστοτε backend σύστημα.



Εικόνα 2-14 : Πύλη API [4]

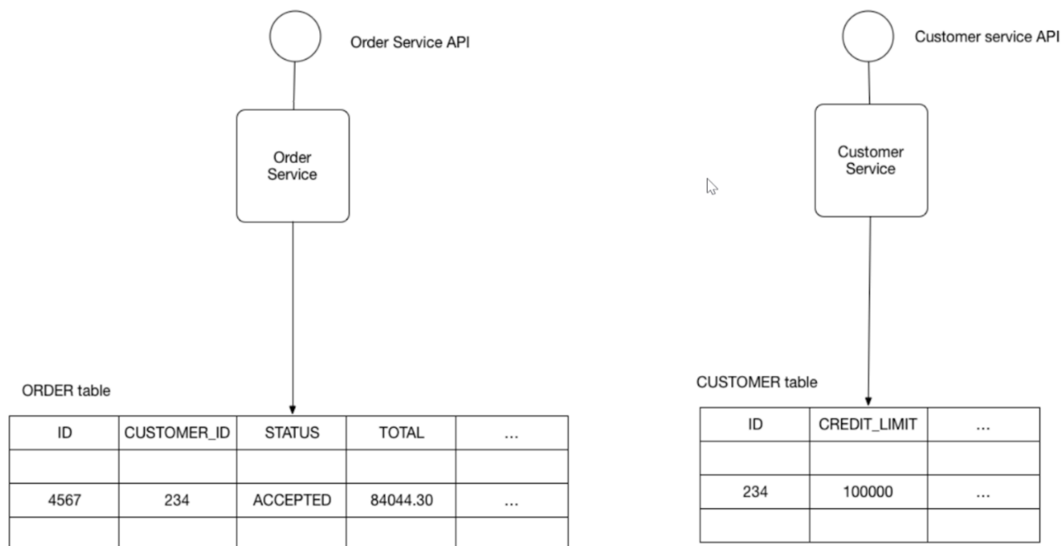
2.2.2 Ξεχωριστή βάση δεδομένων ανά μικροϋπηρεσία

Ένα βασικό χαρακτηριστικό της αρχιτεκτονικής μικροϋπηρεσιών είναι ότι οι υπηρεσίες είναι χαλαρά συζευγμένες και επικοινωνούν μέσω των APIs ή συμβάντων. Ένας τρόπος για να επιτευχθεί χαλαρή σύζευξη είναι η κάθε υπηρεσία να έχει το δικό της χώρο αποθήκευσης δεδομένων.

Σε ένα ηλεκτρονικό κατάστημα για παράδειγμα, η υπηρεσία 'Παραγγελίας' έχει μια βάση δεδομένων που περιλαμβάνει τον πίνακα Παραγγελίες, ενώ η υπηρεσία 'Εξυπηρέτηση πελατών' έχει τη βάση δεδομένων της, η οποία περιλαμβάνει τον πίνακα

[3] Βλέπε: Facade pattern - Wikipedia
[4] Πηγή: The API gateway pattern versus the direct client-to-microservice communication | Microsoft Docs

Πελάτες. Κατά το χρόνο εκτέλεσης, οι υπηρεσίες είναι απομονωμένες μεταξύ τους, οπότε μια υπηρεσία δεν θα αποκλειστεί ποτέ επειδή μια άλλη υπηρεσία διατηρεί ένα κλείδωμα στην βάση δεδομένων.[6]



Εικόνα 2-15 [5]: Διάγραμμα βάσης δεδομένων

Το συγκεκριμένο μοτίβο, πέρα από τη χαλαρή σύζευξη, μας δίνει τη δυνατότητα να αναπτύξουμε, να δημοσιεύσουμε και να κλιμακώσουμε τη κάθε υπηρεσία ανεξάρτητα. Καλούμαστε όμως να αντιμετωπίσουμε κάποιες προκλήσεις. Πέρα από τη ανάπτυξη και συντήρηση αρκετών βάσεων δεδομένων, τα δεδομένα μας διακατέχονται από την έννοια της τελικής συνέπειας. Αυτό σημαίνει ότι οι ενημερώσεις ανάμεσα στις υπηρεσίες θα χρειαστούν κάποιο χρόνο ώστε να διαδοθούν και να συγχρονιστούν τελικά τα δεδομένα.

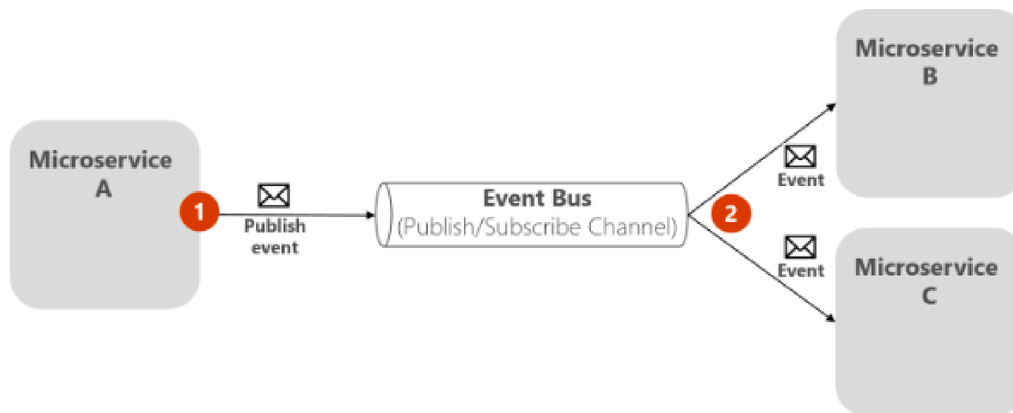
2.2.3 Event driven – Publish/Subscribe

Μία μικροϋπηρεσία δημοσιεύει συμβάντα(events) σε ένα κανάλι μηνυμάτων, από το οποίο πολλές μικροϋπηρεσίες μπορούν να εγγραφούν ώστε με την είσοδο ενός συμβάντος θα μπορέσουν να ειδοποιηθούν και να ενεργήσουν. Με τον παραπάνω τρόπο επιτυγχάνετε η ασύγχρονη επικοινωνία ανάμεσα στις μικροϋπηρεσίες.

Event bus

Το event bus επιτρέπει την επικοινωνία τύπου publish/subscribe μεταξύ μικροϋπηρεσιών χωρίς να απαιτείται από τα στοιχεία να γνωρίζουν ρητά το καθένα.

[5] Πηγή: Database per service (microservices.io)



Εικόνα 2-16 [6]: Κανάλι μηνυμάτων

Παράδειγμα: Όπως βλέπουμε στη εικόνα 2-16, μια μικροϋπηρεσία δημοσιεύει ένα συμβάν όταν συμβαίνει κάτι αξιοσημείωτο, όπως όταν ενημερώνεται μια επιχειρηματική οντότητα. Οι άλλες μικροϋπηρεσίες, που έχουν εγγραφεί σε αυτό το κανάλι συμβάντων, μόλις λάβουν την ειδοποίηση από το κανάλι για ένα καινούργιο συμβάν, μπορούν να ενημερώσουν και τις δικές τους επιχειρηματικές οντότητες, κάτι που φυσικά μπορεί να οδηγήσει στη δημοσίευση περισσοτέρων συμβάντων.

2.2.4 CQRS – Command and Query Responsibility Segregation

Το Command and Query Responsibility Segregation (CQRS) παρουσιάστηκε στο ομώνυμο βιβλίο του Greg Young το 2010.[7] Ο Young άντλησε την ιδέα από την αρχή του διαχωρισμού εντολής-ερώτησης (CQS) που επινοήθηκε από τον Bertrand Meyer [7].

Η αρχή διαχωρισμού του ερωτήματος και των εντολών, γνωστό ως CQS, δηλώνει ότι κάθε μέθοδος πρέπει είτε να είναι μια εντολή που εκτελεί μια ενέργεια ή ένα ερώτημα που επιστρέφει δεδομένα, αλλά όχι και τα δύο. Εν συντομία, το να κάνεις μια ερώτηση δεν πρέπει να αλλάζει την απάντηση. Για την σωστή ακολουθία της παραπάνω αρχής, πρέπει να βεβαιωθεί ότι εάν μια μέθοδος αλλάξει κάποια κατάσταση, αυτή η μέθοδος θα πρέπει να είναι πάντα τύπου void, διαφορετικά θα πρέπει να επιστρέψει κάποια δεδομένα. Αυτό επιτρέπει την αύξηση της αναγνωσιμότητας και τη μείωση της πολυπλοκότητας του κώδικα.

[6] Πηγή: Implementing event-based communication between microservices (integration events) | Microsoft Docs
[7] Βλέπε: Bertrand Meyer - Wikipedia

Περιορισμοί CQS

Η συγκεκριμένη αρχή δεν είναι δυνατόν να ακολουθείται σε όλες τις περιπτώσεις. Πάντα θα υπάρχουν καταστάσεις όπου θα ήταν λογικό μια μέθοδος να είναι εντολή αλλά και να επιστρέφει δεδομένα. [8] Ένα χαρακτηριστικό παράδειγμα, όπως φαίνεται στην εικόνα 2-5, αποτελεί η γνωστή δομή δεδομένων, στοίβα:

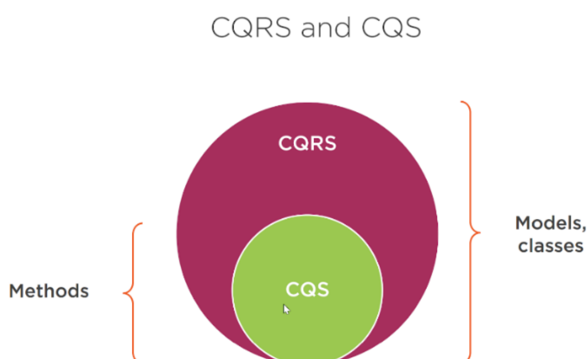
```
var stack = new Stack<string>();
stack.Push("value");              // Command
string value = stack.Pop();       // Both query and command
```

Εικόνα 2-17: Υλοποίηση Στοίβας

Η μέθοδος Pop αφαιρεί το στοιχείο που εισήχθη στην στοίβα τελευταίο και το επιστρέφει. Αυτή η μέθοδος παραβιάζει την αρχή CQS, αλλά ταυτόχρονα, δεν υπάρχει νόημα να διαχωρίσουμε αυτές τις ευθύνες σε δύο διαφορετικές λειτουργίες.

CQS και CQRS

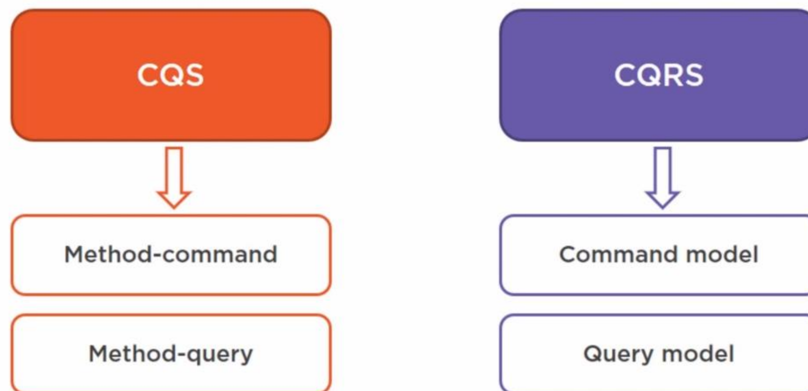
Το CQRS παίρνει την ίδια ιδέα και την επεκτείνει σε υψηλότερο επίπεδο. Αντί για μεθόδους όπως στο CQS, το CQRS εστιάζει στις κλάσεις του μοντέλου και στη συνέχεια εφαρμόζει τις ίδιες αρχές σε αυτά.



Εικόνα 2-18 [8]: CQRS και CQS

[8] Πηγή: [CQRS Course | Pluralsight]

Ακριβώς όπως το CQS ενθαρρύνει το διαχωρισμό μιας μεθόδου σε δύο, μια εντολή και ένα ερώτημα, το CQRS ενθαρρύνει τον διαχωρισμό του ενιαίου μοντέλου σε δύο μοντέλα, ένα για το χειρισμό εντολών/εγγραφών και το άλλο για το χειρισμό των ερωτημάτων, δηλαδή τις αναγνώσεις.

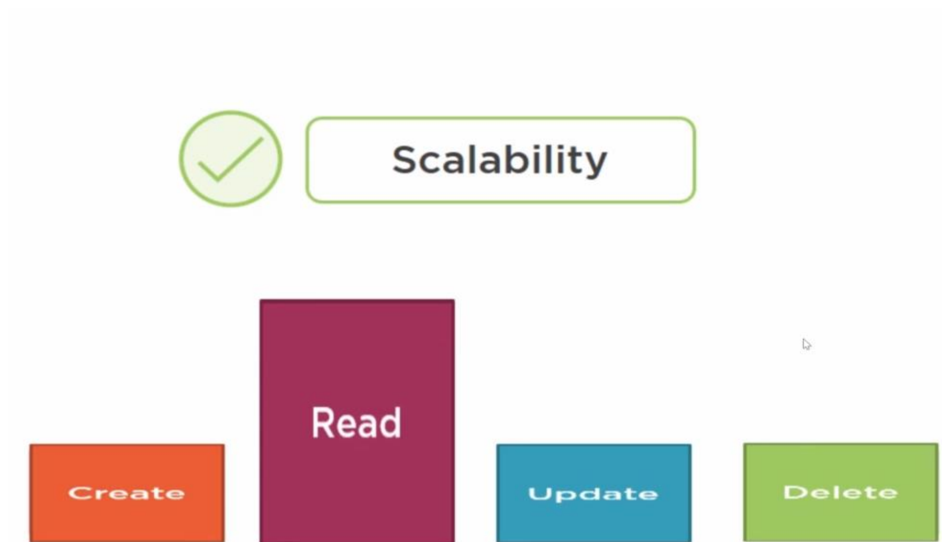


Εικόνα 2-19 [6] : Διαφορές CQS και CQRS

Πλεονεκτήματα του CQRS

Όπως ειπώθηκε παραπάνω, η αρχή του CQRS είναι εξαιρετικά απλή, ωστόσο συνεπάγεται με πολύ ενδιαφέροντα πλεονεκτήματα.

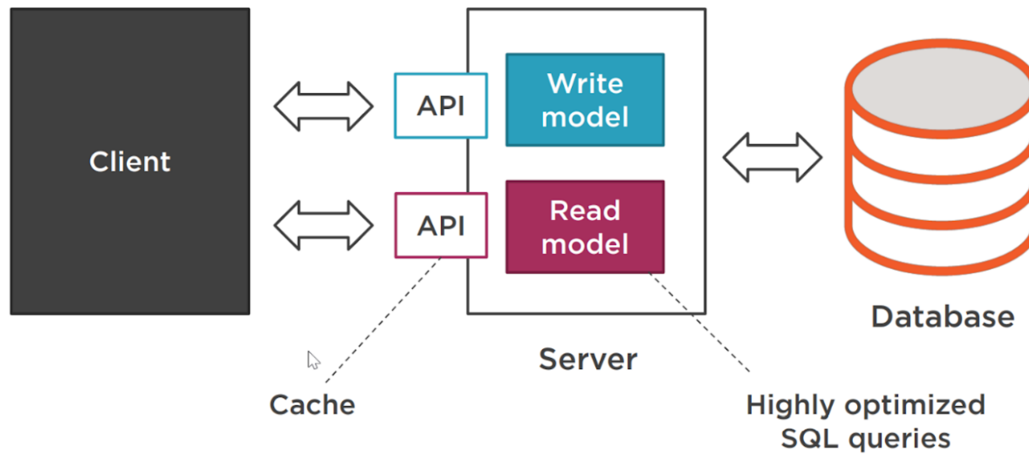
Το πρώτο πλεονέκτημα αποτελεί η επεκτασιμότητα. Σε μία τυπική επιχειρηματική εφαρμογή, παρατηρείται ότι μεταξύ όλων των λειτουργιών που δημιουργούν, διαβάζουν, ενημερώνουν και διαγράφουν δεδομένα, συνήθως η λειτουργία της ανάγνωσης των δεδομένων χρησιμοποιείται περισσότερο. Υπάρχουν περισσότερες αναγνώσεις από εγγραφές σε ένα τυπικό σύστημα, επομένως είναι πολύ σημαντική η δυνατότητα περαιτέρω κλιμάκωσης μόνο της λειτουργίας ανάγνωσης.



Εικόνα 2-20 [6] : Συχνότητα λειτουργιών της εφαρμογής

Η απόδοση του συστήματος αποτελεί το δεύτερο πλεονέκτημα. Με την διαφοροποίηση της ανάγνωσης και της εγγραφής, μπορούν να εφαρμοστούν τεχνικές βελτιστοποίησης που δεν θα ήταν δυνατές σε ένα ενιαίο μοντέλο. Όπως φαίνεται στην εικόνα 2-21, η διαφοροποίηση των ερωτημάτων από τις εντολές επιτρέπει την δημιουργία μια προσωρινής μνήμης (cache) για το συγκεκριμένο τμήμα της εφαρμογής. Επίσης επιτρέπει να χρησιμοποιήσουμε εξαιρετικά εξελιγμένα SQL ερωτήματα για την ανάγνωση δεδομένων από τη βάση δεδομένων, αφήνοντας την πλευρά τον εντολών να το χειριστεί το εκάστοτε ORM [9] της επιλογής μας.

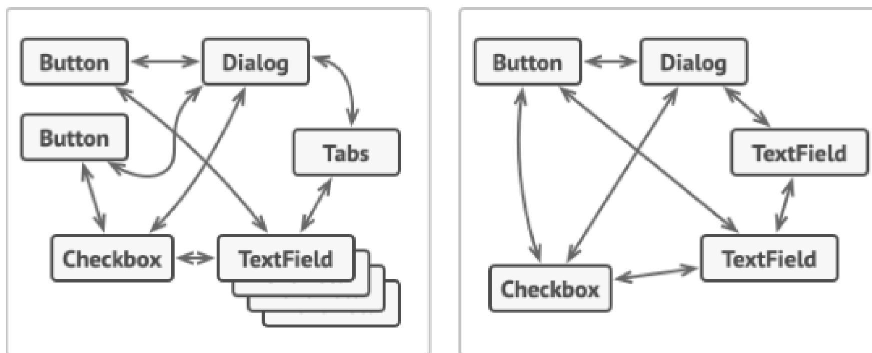
[9] Βλέπε: Object–relational mapping - Wikipedia



Εικόνα 2-21 [6] : Τεχνικές βελτιστοποίησης των αναγνώσεων σε ένα σύστημα

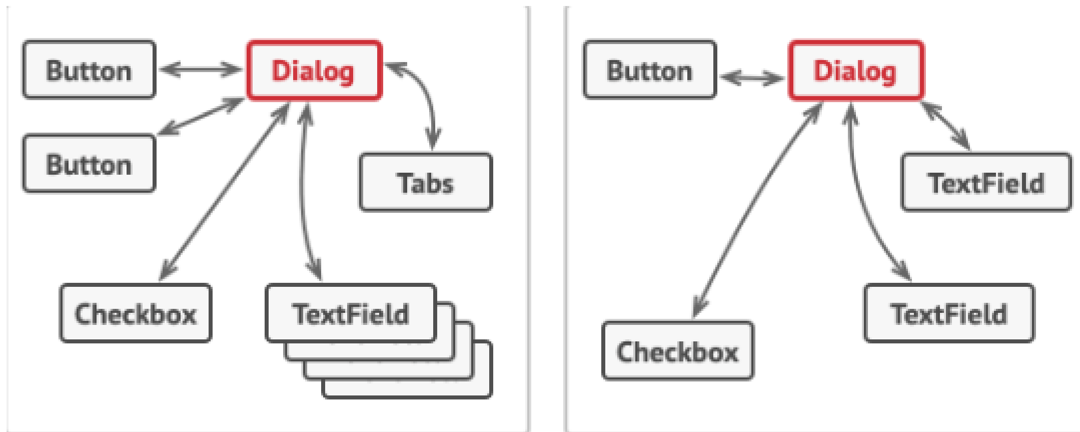
2.2.5 Μοτίβο διαμεσολαβητή – Mediator Pattern

Ένα μοτίβο διαμεσολαβητή περιλαμβάνει τον τρόπο με τον οποίο τα αντικείμενα αλληλοεπιδρούν μεταξύ τους. Αυτό το μοτίβο προωθεί την χαλαρή σύζευξη των αντικειμένων για να αποφευχθεί ένα πολύπλοκο γράφημα εξάρτησης. Το μοτίβο Διαμεσολαβητή αποφεύγει την απευθείας αναφορά μεταξύ των αντικειμένων, ενθυλακώνοντας την επικοινωνία σε ένα κεντρικό αντικείμενο διαμεσολαβητή.



Εικόνα 2-22 [10]: Σχέσεις μεταξύ των στοιχείων του συστήματος χωρίς Mediator

[10] Πηγή: Mediator (refactoring.guru)



Εικόνα 2-23 [7]: Σχέσεις μεταξύ των στοιχείων του συστήματος με Mediator

Σε ένα σύστημα χιλίων αντικειμένων, το οποίο θα πρέπει το καθένα να αναφέρετε και να επικοινωνεί με το άλλο, το γράφημα εξάρτησης θα είναι εξαιρετικά πολύπλοκο. Με το μοτίβο διαμεσολαβητή, δημιουργούμε ένα κεντρικό αντικείμενο διαμεσολαβητή. Αυτό το μεμονωμένο αντικείμενο έχει την αποκλειστική ευθύνη για την διατήρηση των αναφορών στα αντικείμενα του συστήματος μας. Είναι επίσης υπεύθυνο για τη μετάδοση οποιωνδήποτε μηνυμάτων επικοινωνίας μεταξύ και προς αυτά τα αντικείμενα. Θα μπορούσαμε να παραλληλίσουμε έναν διαμεσολαβητή σαν ένα κόμβο επικοινωνίας.

Στα αρνητικά του συγκεκριμένου μοτίβου μπορούμε να συμπεριλάβουμε ότι το συγκεκριμένο αντικείμενο αποτελεί ένα god object [11]. Η λύση για το συγκεκριμένο antipattern είναι η χρήση της ‘Αρχή της Ενιαίας Ευθύνης’ από τις αρχές SOLID[12]. Η συγκεκριμένη αρχή δηλώνει ότι μια κλάση πρέπει να έχει μόνο μια ευθύνη. Ακολουθώντας την προσέγγιση ‘διαίρει και βασίλευε’, θα πρέπει να διαχωρίζουμε τα αντικείμενα διαμεσολαβητή, δημιουργώντας ομάδες κλάσεων γύρω από την ίδια συμπεριφορά.

2.3 Τεχνολογίες

2.3.1 HTTP

Το Πρωτόκολλο Μεταφοράς Υπερκειμένου (HyperText Transfer Protocol, HTTP) είναι ένα πρωτόκολλο επικοινωνίας. Αποτελεί το κύριο πρωτόκολλο που

[11] Βλέπε: God object - Wikipedia
[12] Βλέπε: SOLID - Wikipedia

χρησιμοποιείται στους φυλλομετρητές του Παγκοσμίου Ιστού για να μεταφέρει δεδομένα ανάμεσα σε έναν διακομιστή (server) και έναν πελάτη (client).

Αν και το HTTP πρωτόκολλο σχεδιάστηκε για χρήση στον Ιστό, υποστηρίζει λειτουργίες που είναι πιο γενικές απ' ότι απαιτείται. Οι λειτουργίες αυτές ονομάζονται μέθοδοι.

Παρακάτω παρουσιάζονται συνοπτικά οι βασικές μέθοδοι αίτησης του πρωτοκόλλου HTTP:

GET: Η μέθοδος GET ζητά από το διακομιστή να στείλει τη σελίδα. Η σελίδα κωδικοποιείται κατάλληλα σε μορφή MIME.

POST: Η μέθοδος POST χρησιμοποιείται κατά την υποβολή φορμών. Όπως και η μέθοδος GET, η POST περιέχει μια διεύθυνση URL αλλά αντί να ανακτά απλώς τη σελίδα μεταφέρει δεδομένα στον διακομιστή όπως για παράδειγμα τα περιεχόμενα της φόρμας. Έπειτα ο διακομιστής κάνει κάτι με αυτά τα δεδομένα ανάλογα με το URL.Τέλος, η μέθοδος επιστρέφει μια σελίδα που δείχνει το αποτέλεσμα.

PUT: Η μέθοδος PUT είναι η αντίστροφη της GET, δηλαδή αντί να διαβάζει τη σελίδα, γράφει τη σελίδα. Η μέθοδος αυτή κάνει εφικτή την κατασκευή μιας συλλογής ιστοσελίδων σε έναν απομακρυσμένο διακομιστή. Το σώμα της αίτησης περιέχει τη σελίδα. Μπορεί να κωδικοποιείται μέσω του MIME, οπότε οι γραμμές που ακολουθούν την PUT μπορεί να περιέχουν κεφαλίδες Content-Type και πιστοποίησης ταυτότητας, ώστε να αποδείξουν ότι ο αιτών έχει πραγματικά την άδεια να εκτελέσει τη ζητούμενη λειτουργία.

DELETE: Η μέθοδος DELETE καταργεί τη σελίδα ή τουλάχιστον δηλώνει ότι ο διακομιστής Ιστού έχει συμφωνήσει να καταργήσει τη σελίδα. Όπως και με την PUT και σε αυτή τη μέθοδο παίζουν μεγάλο ρόλο η πιστοποίηση της ταυτότητας και της άδειας εκτέλεσης της λειτουργίας. [9]

2.3.2 REST API

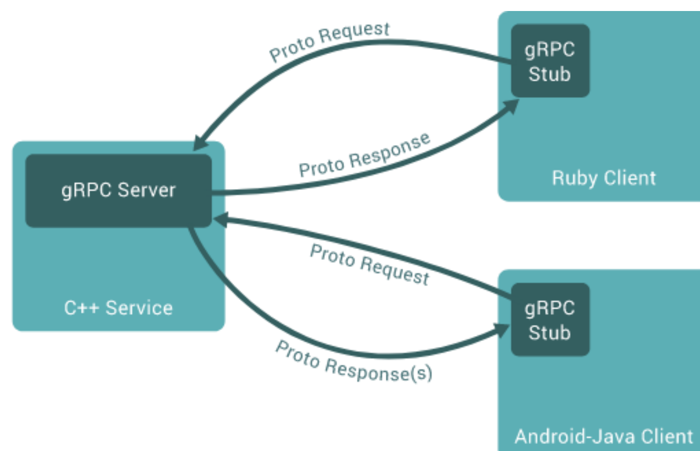
Το REST API (γνωστό και ως RESTful API) είναι μια διεπαφή προγραμματισμού εφαρμογών (API ή web API) που συμμορφώνεται με τους περιορισμούς του αρχιτεκτονικού στυλ REST και επιτρέπει την αλληλεπίδραση με τις υπηρεσίες ιστού RESTful. Το REST σημαίνει μεταφορά κατάστασης αναπαράστασης και δημιουργήθηκε από τον επιστήμονα υπολογιστών Roy Fielding.

Το REST είναι ένα σύνολο αρχιτεκτονικών περιορισμών, όχι ένα πρωτόκολλο ή ένα πρότυπο. Οι προγραμματιστές API μπορούν να εφαρμόσουν το REST με διάφορους τρόπους.

Όταν ένα αίτημα πελάτη γίνεται μέσω ενός RESTful API, μεταφέρει μια αναπαράσταση της κατάστασης του πόρου στον αιτούντα ή στο τελικό σημείο. Αυτές οι πληροφορίες, ή αναπαράσταση, παραδίδονται σε μία από τις διάφορες μορφές μέσω HTTP: JSON (Javascript Object Notation), HTML, XLT, Python, PHP ή απλό κείμενο. Το JSON είναι η πιο δημοφιλής μορφή αρχείου που χρησιμοποιείται γενικά, επειδή, παρά το όνομά του, είναι αγνωστική ως προς τη γλώσσα, καθώς και αναγνώσιμο τόσο από ανθρώπους όσο και από μηχανήματα. [10]

2.3.3 GRPC

Στο gRPC, μια εφαρμογή πελάτη μπορεί να καλέσει απευθείας μια μέθοδο σε μια εφαρμογή διακομιστή σε διαφορετικό μηχάνημα σαν να ήταν ένα τοπικό αντικείμενο, διευκολύνοντας τη δημιουργία κατανεμημένων εφαρμογών και υπηρεσιών. Όπως σε πολλά συστήματα RPC, το gRPC βασίζεται στην ιδέα του ορισμού μιας υπηρεσίας, καθορίζοντας τις μεθόδους που μπορούν να κληθούν εξ αποστάσεως με τις παραμέτρους και τους τύπους επιστροφής. Από την πλευρά του διακομιστή, ο διακομιστής υλοποιεί αυτήν τη διεπαφή και εκτελεί έναν διακομιστή gRPC για τη διαχείριση κλήσεων πελατών. Στην πλευρά του πελάτη, ο υπολογιστής-πελάτης έχει ένα στέλεχος (αναφέρεται απλώς ως πελάτης σε ορισμένες γλώσσες) που παρέχει τις ίδιες μεθόδους με τον διακομιστή.



Εικόνα 2-23: Παράδειγμα GRPC

Από προεπιλογή, το gRPC χρησιμοποιεί Protocol Buffers, τον ώριμο μηχανισμό ανοιχτού κώδικα της Google για σειριοποίηση δομημένων δεδομένων (αν και μπορεί να χρησιμοποιηθεί με άλλες μορφές δεδομένων όπως το JSON). Ακολουθεί μια γρήγορη εισαγωγή για το πώς λειτουργεί. Εάν είστε ήδη εξοικειωμένοι με τα buffer πρωτοκόλλου, μη διστάσετε να μεταβείτε στην επόμενη ενότητα. [11]

2.3.4 SQL Server

Ο SQL Server είναι μια σχεσιακή βάση δεδομένων, η οποία αναπτύσσεται από τη Microsoft. Οι κύριες γλώσσες που χρησιμοποιούνται είναι η T-SQL και η ANSI SQL. Ο SQL Server βγήκε για πρώτη φορά στην αγορά το 1989 σε συνεργασία με την Sybase.

Η κύρια μονάδα αποθήκευσης στοιχείων είναι μια βάση δεδομένων, η οποία αποτελείται από μια συλλογή πινάκων και κώδικα. [12]

2.3.5 RabbitMQ

Το RabbitMQ είναι ένα λογισμικό αναμονής μηνυμάτων, γνωστό και ως μεσίτης μηνυμάτων ή διαχειριστής ουρών αναμονής. Με απλά λόγια, πρόκειται για λογισμικό όπου ορίζονται ουρές, στις οποίες συνδέονται εφαρμογές για να μεταφέρουν ένα μήνυμα ή μηνύματα.

Ένα μήνυμα μπορεί να περιλαμβάνει κάθε είδους πληροφορία. Θα μπορούσε, για παράδειγμα, να περιέχει πληροφορίες σχετικά με μια διεργασία ή εργασία που πρέπει να ξεκινήσει σε μια άλλη εφαρμογή (η οποία θα μπορούσε να βρίσκεται ακόμη και σε άλλο διακομιστή), ή θα μπορούσε να είναι ένα απλό μήνυμα κειμένου. Το λογισμικό διαχείρισης ουράς αποθηκεύει τα μηνύματα μέχρι να συνδεθεί μια εφαρμογή λήψης και να αφαιρέσει ένα μήνυμα από την ουρά. Στη συνέχεια, η λαμβάνουσα εφαρμογή επεξεργάζεται το μήνυμα. [13]

2.3.6 Docker

Το Docker είναι μια ανοιχτή πλατφόρμα για την ανάπτυξη, αποστολή και εκτέλεση εφαρμογών. Το Docker επιτρέπει το διαχωρισμό των εφαρμογών από την υποδομή, ώστε η γρήγορη παράδοση του λογισμικού να είναι εφικτή. Με το Docker, μπορούμε να διαχειριστούμε την υποδομή μας με τους ίδιους τρόπους διαχείρισης των

εφαρμογών μας. Αξιοποιώντας τις μεθοδολογίες του Docker για τη γρήγορη αποστολή, δοκιμή και ανάπτυξη κώδικα, μπορούμε να μειώσουμε σημαντικά την καθυστέρηση μεταξύ της συγγραφής κώδικα και της εκτέλεσής του στην παραγωγή. [14]

3 Σχεδίαση

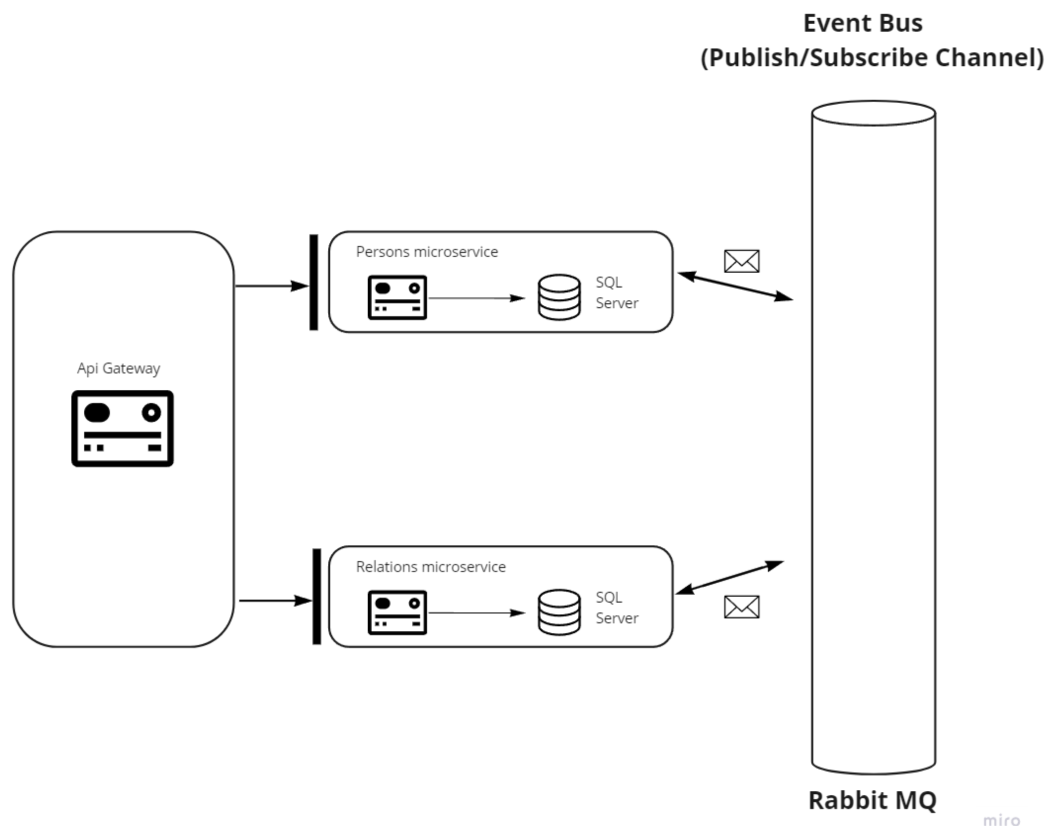
Στην πρώτη ενότητα του κεφαλαίου παρουσιάζονται τα βασικά χαρακτηριστικά της πιλοτικής εφαρμογής. Στη δεύτερη ενότητα γίνεται ανάλυση της αρχιτεκτονικής του συστήματος, των τριών βασικών υπηρεσιών και των βάσεων δεδομένων ανά υπηρεσία.

3.1 Καταγραφή απαιτήσεων

Οι κύριες απαιτήσεις του συστήματος που σχεδιάστηκε αφορούν τα εξής σημεία:

- Λειτουργία του συστήματος σε μεγάλη κλιμάκωση
- Οριζόντια κλιμάκωση
- Υψηλή διαθεσιμότητα
- Αυτόνομες μικροϋπηρεσίες
- Υλοποίηση μικροϋπηρεσιών με την προσέγγιση του domain driven design(DDD)

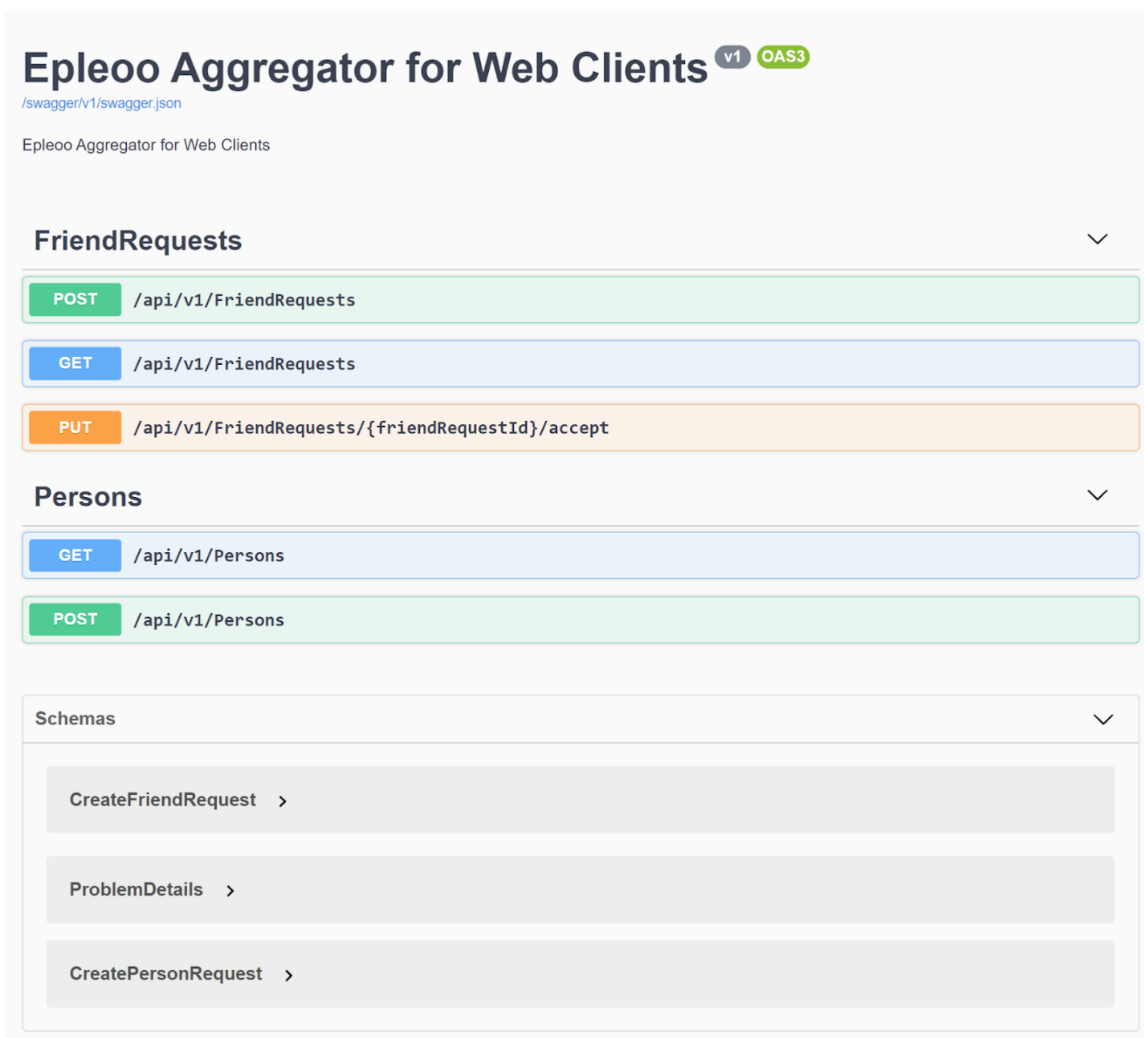
3.2 Αρχιτεκτονική συστήματος



Εικόνα 3-1: Αρχιτεκτονική της πιλοτικής εφαρμογής

3.2.1 Πύλη API (API Gateway)

Όπως αναφέρθηκε στο κεφάλαιο 2.2.1, η πύλη API αποτελεί ένα σημείο πρόσβασης για ορισμένες ομάδες μικροϋπηρεσιών. Ακολούθως, η πύλη API του πιλοτικού συστήματος, αποτελεί το μοναδικό σημείο πρόσβασης για τους πελάτες στις μικροϋπηρεσίες Persons και PeopleRelations. Η πύλη σχεδιάστηκε με γνώμονα την εξυπηρέτηση web πελατών. Οι web πελάτες μπορούν να επικοινωνήσουν με την πύλη μέσω HTTP/REST καθώς η πύλη αποτελεί ένα HTTP API. Εάν οι πελάτες αποτελούσαν κινητές συσκευές, θα είχαν την ανάγκη να επικοινωνήσουν με διαφορετικό πρωτόκολλο ή κάποιες διαφορετικές παραμέτρους. Η συγκεκριμένη αρχιτεκτονική μας επιτρέπει τη δημιουργία μιας επιπρόσθετης πύλης, υλοποιημένης με διαφορετικό πρωτόκολλο ή HTTP με διαφορετικές παραμέτρους, υλοποιημένη με γνώμονα τους πελάτες κινητών συσκευών.



Εικόνα 3-2: API της πύλης API

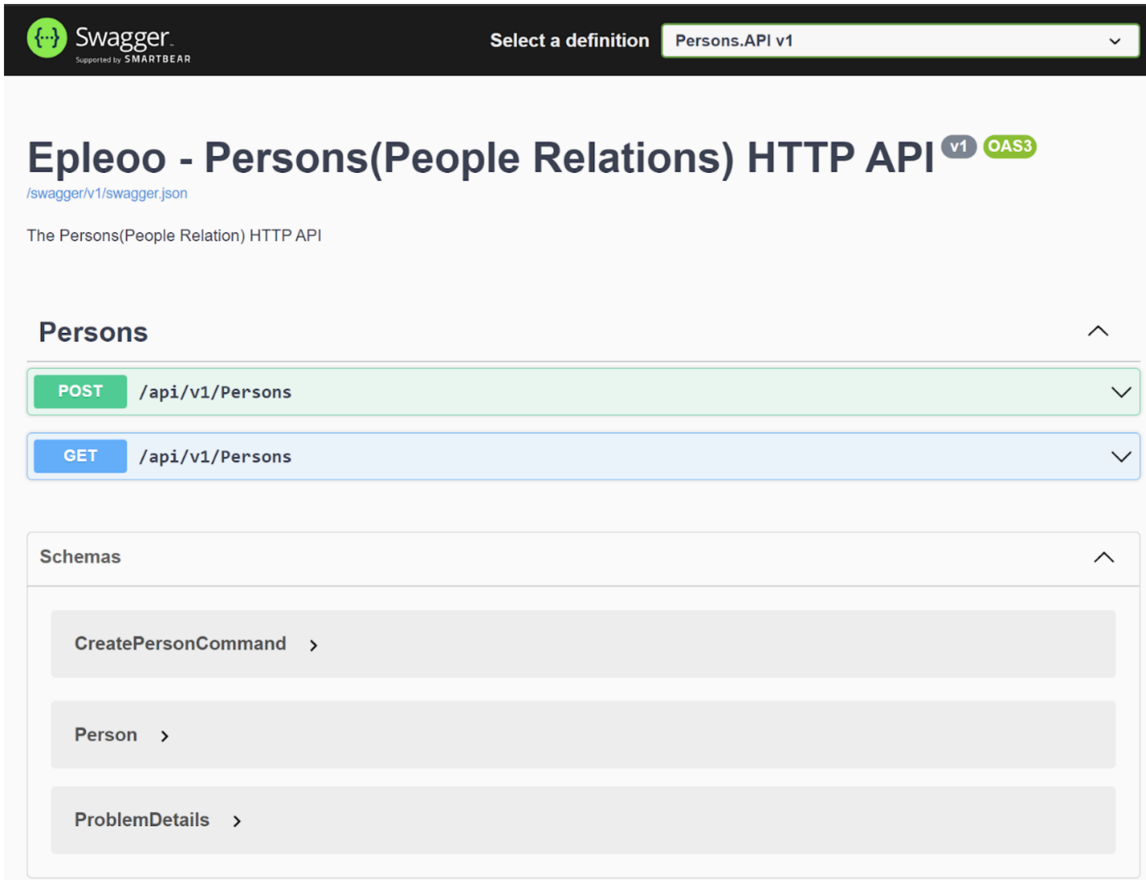
Τέλος, η επικοινωνία μεταξύ της πύλης και των μικροϋπηρεσιών γίνεται με τα πρωτόκολλα HTTP/REST και GRPC.

3.2.2 Μικροϋπηρεσία Persons

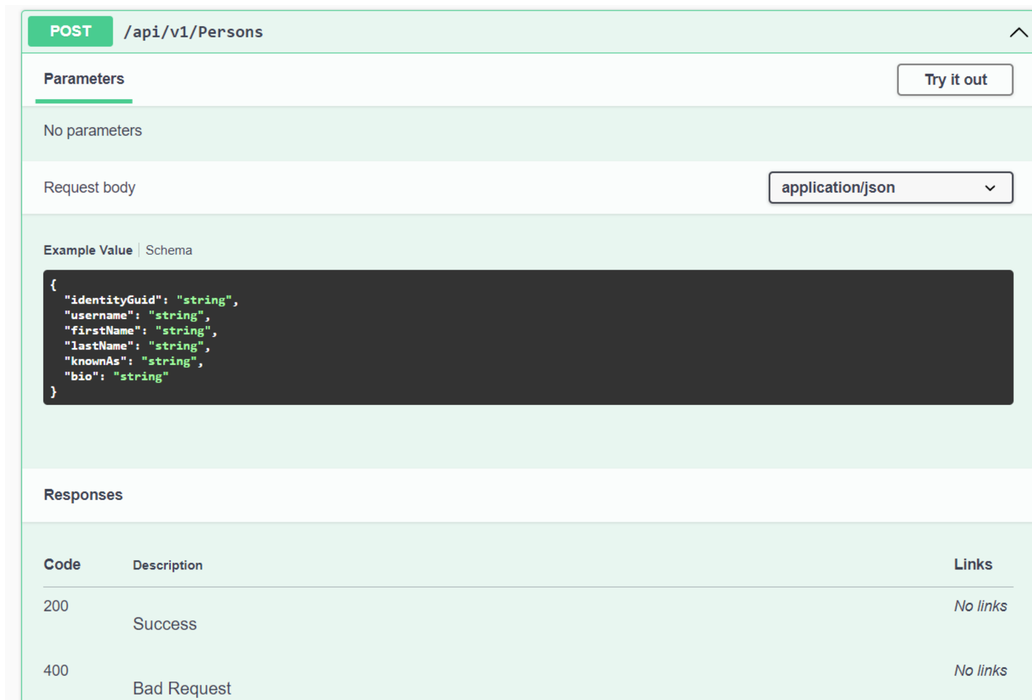
Η υπηρεσία Persons είναι υπεύθυνη στο σύστημα για τη λογική του τομέα των χρηστών της εφαρμογής. Διαχειρίζεται τις εγγραφές των χρηστών και τη μόνιμη αποθήκευση τους στη βάση δεδομένων.

3.2.2.1 API

Το API της μικροϋπηρεσίας Persons, εικόνα 3-3, αποτελείται από δύο τελικά σημεία HTTP. Το POST /Persons δημιουργεί ένα καινούργιο άτομο στο σύστημα. Στην εικόνα 3-3-2 φαίνονται οι λεπτομέρειες της κλήσης.

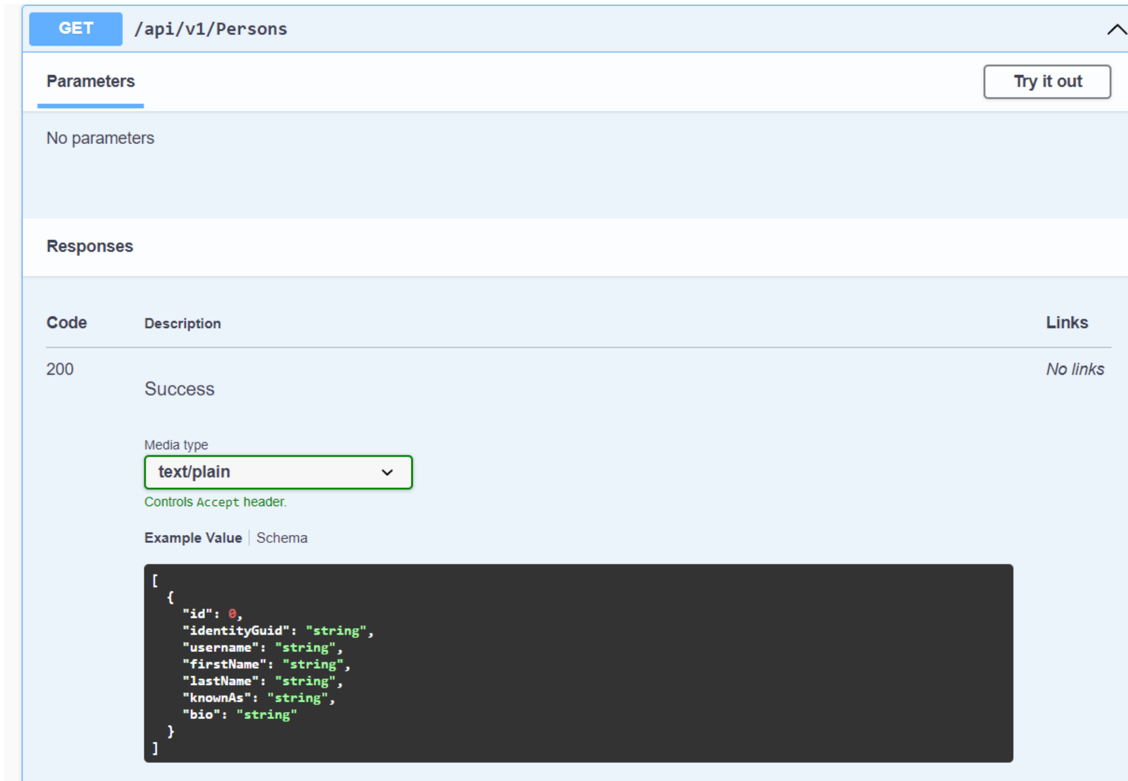


Εικόνα 3-3: API της μικροϋπηρεσίας Persons



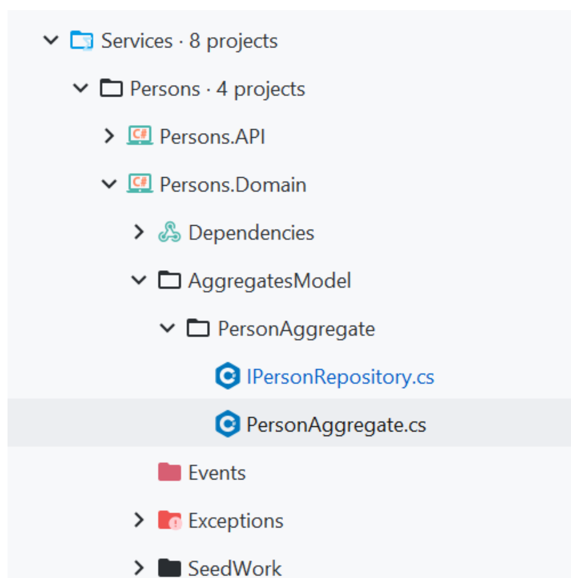
Εικόνα 3-3-2: Post /Persons

Η GET /Persons καλεί όλα τα άτομα που έχουν εγγραφεί στο σύστημα, όπως φαίνεται στη παρακάτω εικόνα 3-3-3.



Εικόνα 3-3-3: Get /Persons

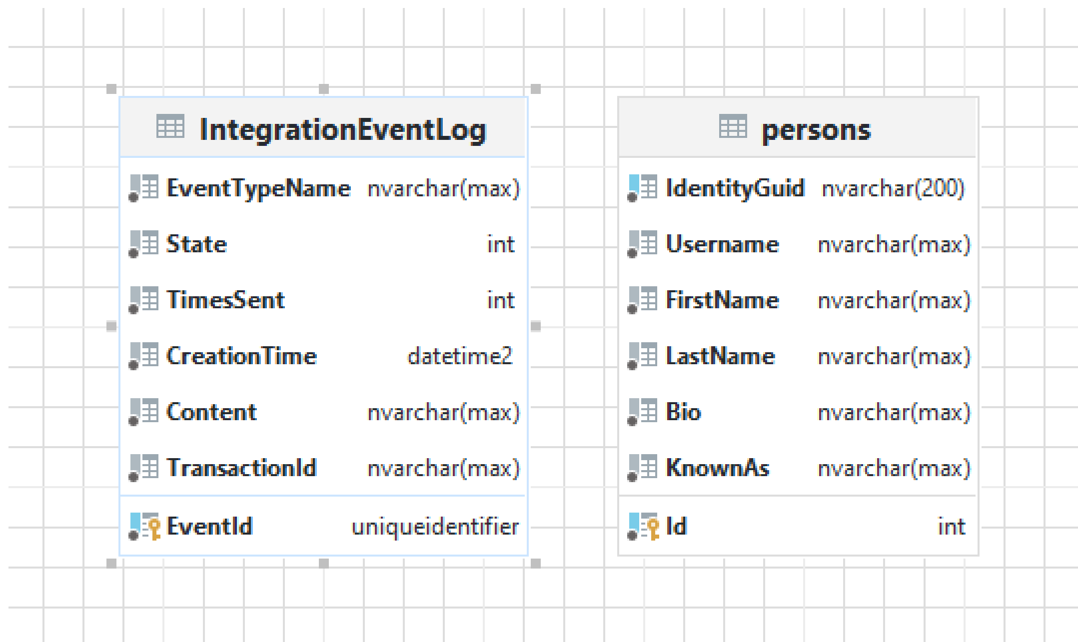
3.2.2.2 Οντότητες και σύνολα



Εικόνα 3-3-4: Οι οντότητες και τα σύνολα της μικροϋπηρεσίας Persons

3.2.2.3 Σχεδίαση βάσης δεδομένων

Καθώς η υπηρεσία Persons αποτελείται από μια οντότητα, συνεπώς η βάση δεδομένων αποτελείται και από ένα πίνακα, το persons. Επίσης, όπως σε όλες τις υπηρεσίες, υπάρχει ο πίνακας IntegrationEventLog, ο οποίος αποθηκεύει τα integration events που δημιουργούνται από τη συγκεκριμένη υπηρεσία προς τις άλλες.



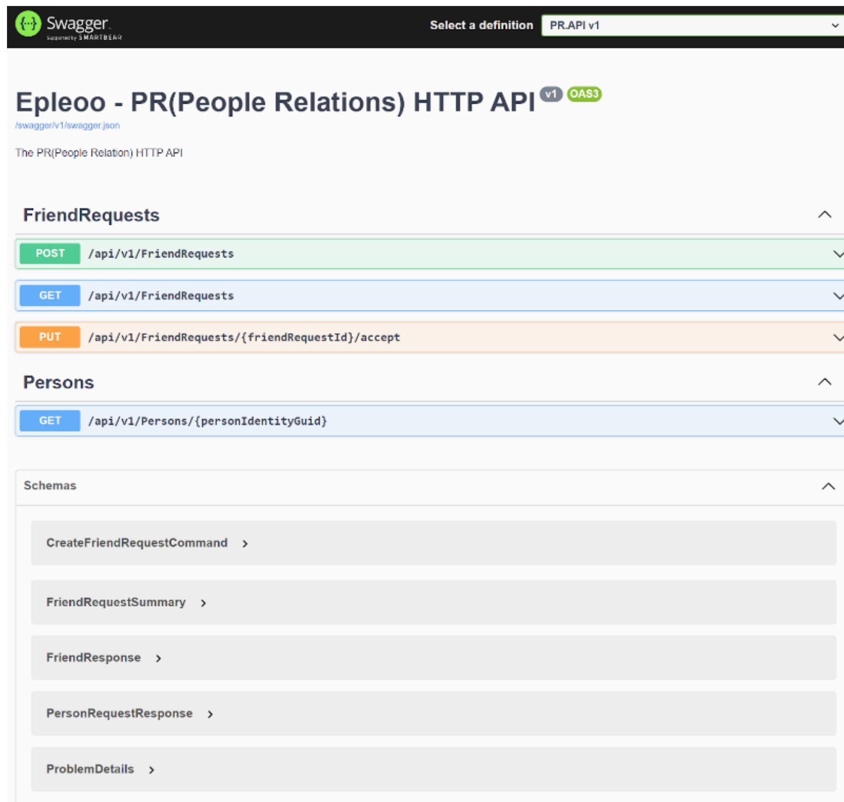
Εικόνα 3-3-5: Διάγραμμα βάσης δεδομένων της μικροϋπηρεσίας Persons

3.2.3 Μικροϋπηρεσία PeopleRelations

Η υπηρεσία Persons Relations είναι υπεύθυνη στο σύστημα για τη λογική του τομέα των σχέσεων φιλίας μεταξύ των χρηστών της εφαρμογής. Δημιουργεί και διαχειρίζεται τη δημιουργία αιτημάτων φιλίας από έναν χρήστη προς ένα άλλον, την αποδοχή του αιτήματος και τη δημιουργία φιλίας μεταξύ τους.

3.2.3.1 API

Το API της μικροϋπηρεσίας PeopleRelations (PR) αποτελείται από 4 τελικά σημεία HTTP, τα οποία ανήκουν σε δύο κατηγορίες.



Εικόνα 3-3-6: API της μικροϋπηρεσίας PeopleRelations

Τη κατηγορία Persons, που διαθέτει ένα τελικό σημείο, το /Persons/{personIdentityGuid} και είναι ενημερωμένο και συγχρονισμένο με το πίνακα Persons της μικροϋπηρεσίας Persons. Δέχεται ως παράμετρο το IdentityGuid κάποιου χρήστη και επιστρέφει τους φίλους του, όπως φαίνεται στη παρακάτω εικόνα 3-3-7.

GET /api/v1/Persons/{personIdentityGuid}

Parameters | Try it out

Name	Description
personIdentityGuid * required string *(path)*	personIdentityGuid

Responses

Code	Description	Links
200	Success	*No links*

Media type: text/plain

Controls Accept header.

Example Value | Schema

```
{
  "friends": [
    {
      "personId": 0,
      "personIdentityGuid": "string"
    }
  ]
}
```

Εικόνα 3-3-7: Η Get /Persons

Η κατηγορία FriendRequests, όπου διαθέτει τρία τελικά σημεία HTTP.

Τη POST /FriendRequests η οποία δημιουργεί ένα αίτημα φιλίας μεταξύ των χρηστών.

POST /api/v1/FriendRequests

Parameters | Try it out

No parameters

Request body: application/json

Example Value | Schema

```
{
  "senderPersonIdentityGuid": "string",
  "receiverPersonIdentityGuid": "string"
}
```

Responses

Code	Description	Links
200	Success	*No links*
400	Bad Request	*No links*

Media type: text/plain

Example Value | Schema

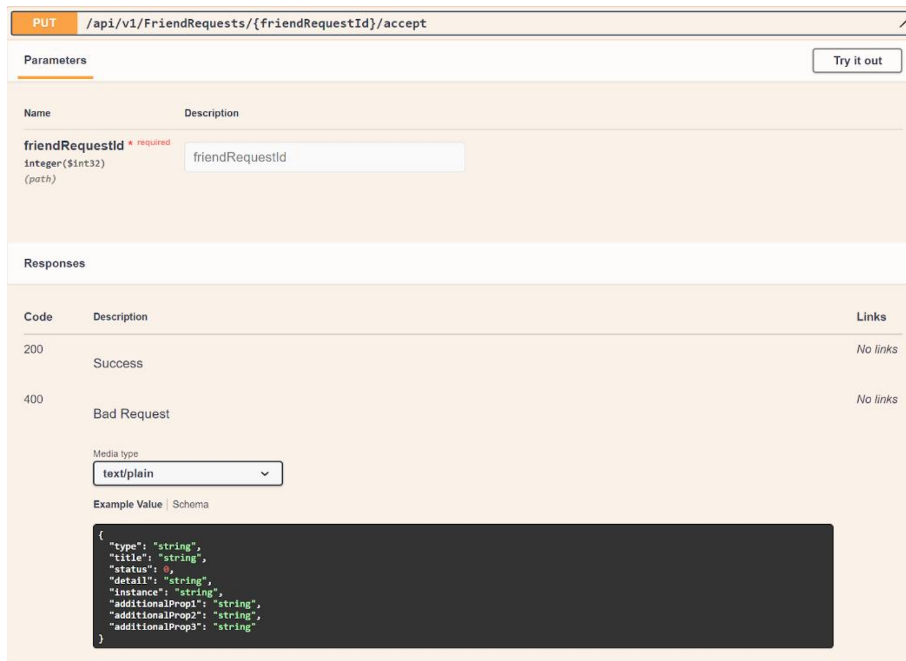
```
{
  "type": "string",
  "title": "string",
  "status": 0,
  "detail": "string",
  "instance": "string",
  "additionalProp1": "string",
  "additionalProp2": "string",
  "additionalProp3": "string"
}
```

Εικόνα 3-3-8: Η Post /FriendRequests

Η GET /FriendRequests, η οποία επιστρέφει τα friendRequests που έχει στείλει ή δεχθεί ένας χρήστης. Δέχεται δύο προαιρετικές παραμέτρους, το IdentityGuid του χρήστη που έχει στείλει το αίτημα ή το IdentityGuid του χρήστη που έχει δεχθεί αίτημα.

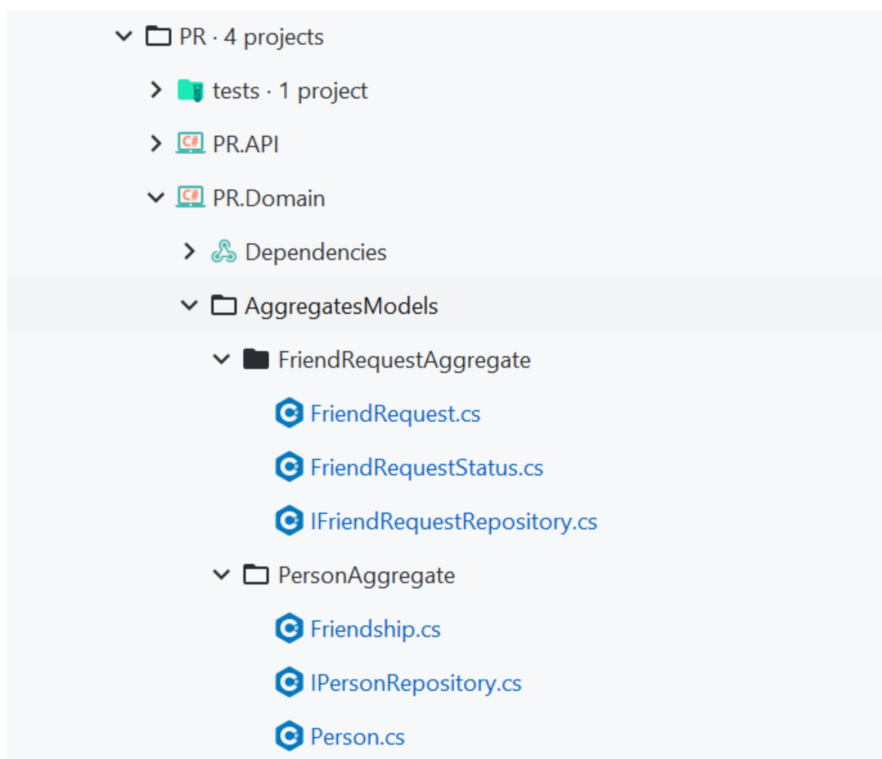
Εικόνα 3-3-9: Η Get /FriendRequests

Η PUT /FriendRequests/{FriendRequestId}/accept, με την οποία ένας χρήστης αποδέχεται ένα αίτημα φιλίας. Δέχεται ως παράμετρο το Id του αιτήματος φιλίας.



Εικόνα 3-3-10: Η Put /FriendRequests

3.2.3.2 Οντότητες και σύνολα



Εικόνα 3-3-11: Οντότητες και σύνολα της μικροϋπηρεσίας PeopleRelations

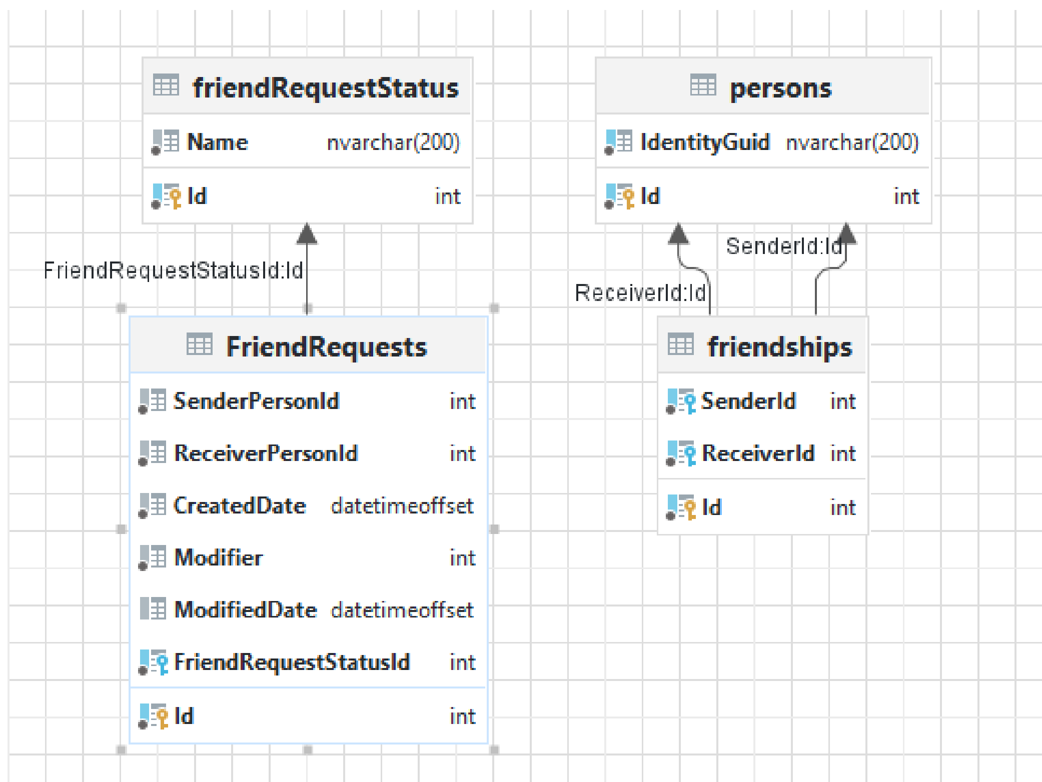
Η υπηρεσία PeopleRelations, αποτελείται από δύο σύνολα:

- Το σύνολο FriendRequestAggregate, το οποίο αποτελείται από τις οντότητες FriendRequest και FriendRequestStatus.

- Το σύνολο PersonAggregate, το οποίο αποτελείται από τις οντότητες Friendship και Person.

3.2.3.3 Σχεδίαση βάσης δεδομένων

Η υπηρεσία PeopleRelations αποτελείται από τέσσερις πίνακες. Το κάθε σύνολο διαθέτει από δύο. Στο παρακάτω σχεδιάγραμμα 3-3-12, μπορούμε να διακρίνουμε ότι υπάρχουν σχέσεις μόνο μεταξύ των πινάκων του ίδιου συνόλου και όχι το αντίθετο. Για την επικοινωνία μεταξύ των συνόλων, χρησιμοποιούνται τα domain events. Επίσης, αξίζει να αναφερθεί, ότι ο πίνακας persons αποτελεί ένα συγχρονισμένο αντίγραφο, μέσω integration event, με το πίνακα persons της υπηρεσίας Persons. Η μόνη διαφορά προκύπτει στο σύνολο των πληροφοριών, τις στήλες του κάθε πίνακα. Στην υπηρεσία Persons, αποθηκεύουμε όλα τα στοιχεία ενός χρήστη, όπως username και ονοματεπώνυμο, ενώ στην υπηρεσία Persons Relations μόνο το IdentityGuid, καθώς ο λογικός τομέας του δεν χρειάζεται τις περαιτέρω πληροφορίες. Τους κρατάμε συγχρονισμένους για να επιτύχουμε συνέπεια των δεδομένων ανάμεσα στις υπηρεσίες. Τέλος ο συγχρονισμένος πίνακας μας δίνει τη δυνατότητα να πραγματοποιήσουμε συγκεκριμένους έλεγχους στην υπηρεσία PeopleRelations, ώστε να μην χρειάζεται κάθε φορά να καλεί την υπηρεσία Persons.



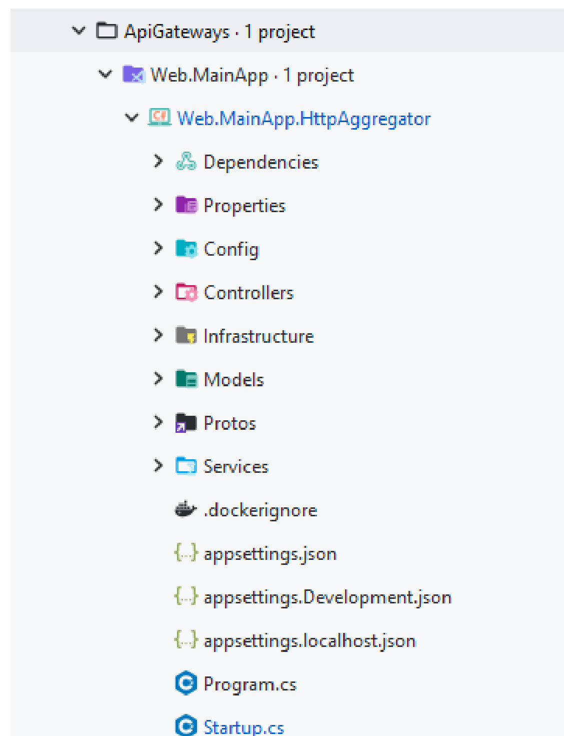
Εικόνα 3-3-12: Διάγραμμα βάσης δεδομένων της μικροϋπηρεσίας PeopleRelations

4 Υλοποίηση

Όπως ήδη αναφέρθηκε στο προηγούμενο κεφάλαιο, η εφαρμογή αποτελείται από δύο υπηρεσίες και μία εξωτερική πύλη HTTP που επικοινωνούν μεταξύ τους. Αυτό το κεφάλαιο περιγράφει την υλοποίηση της κάθε υπηρεσίας ξεχωριστά.

4.1 Υπηρεσίες συστήματος

4.1.1 HTTP API Gateway



Εικόνα 4-1: Δομή του project HTTP Gateway

Η υπηρεσία της πύλης αποτελεί μια κλασσική εφαρμογή Model-View-Controller(MVC). Παρακάτω γίνεται ανάλυση των φακέλων με τη σειρά που φαίνεται στην εικόνα 4-1. [15]

Properties: Περιέχει το αρχείο launchSettings.json, το οποίο εμπεριέχει όλες τις πληροφορίες που απαιτούνται για την εκτέλεση του προγράμματος: λεπτομέρειες διαμόρφωσης σχετικά με τις ενέργειες που πρέπει να διεκπεραιωθούν κατά την εκτέλεση, λεπτομέρειες όπως ρυθμίσεις web διακομιστών, διευθύνσεις URL εφαρμογών και λεπτομέρειες θυρών SSL.

Config: Περιέχει τη κλάση UrlConfig.cs, η οποία εμπεριέχει τα τελικά σημεία HTTP της κάθε μικροϋπηρεσίας.

Controllers: Περιέχει τους controllers. Οι controllers διαχειρίζονται όλα τα εισερχόμενα αιτήματα και είναι υπεύθυνοι για το χειρισμό της αλληλεπίδρασης του τελικού χρήστη, το χειρισμό του μοντέλου και την επιλογή μιας προβολής για την εμφάνιση της διεπαφής του χρήστη.

```
[Route("api/v1/[controller]")]
[ApiController]
public class FriendRequestsController : ControllerBase
{
   private readonly IPrApiClient _client;

   public FriendRequestsController(IPrApiClient client)
   {
       _client = client;
   }

   [HttpPost]
   [ProducesResponseType((int)HttpStatusCode.BadRequest)]
   [ProducesResponseType((int)HttpStatusCode.Created)]
   public async Task<IActionResult>
CreateFriendRequest(CreateFriendRequest request)
   {
       var response = await _client.CreateFriendRequest(request);

       if (response.IsFailure)
           return BadRequest();

       return Created(string.Empty, string.Empty);
   }

   [HttpGet]
   [ProducesResponseType((int)HttpStatusCode.BadRequest)]
   [ProducesResponseType((int)HttpStatusCode.OK)]
   public async Task<IActionResult> CreatePersonAsync()
   {
       var response = await _client.GetFriendRequests();

       return Ok(response);
   }

   [Route("{friendRequestId}/accept")]
   [HttpPut]
   [ProducesResponseType((int)HttpStatusCode.BadRequest)]
   [ProducesResponseType((int)HttpStatusCode.OK)]
   public async Task<IActionResult> AcceptFriendRequest(int
friendRequestId)
   {
```

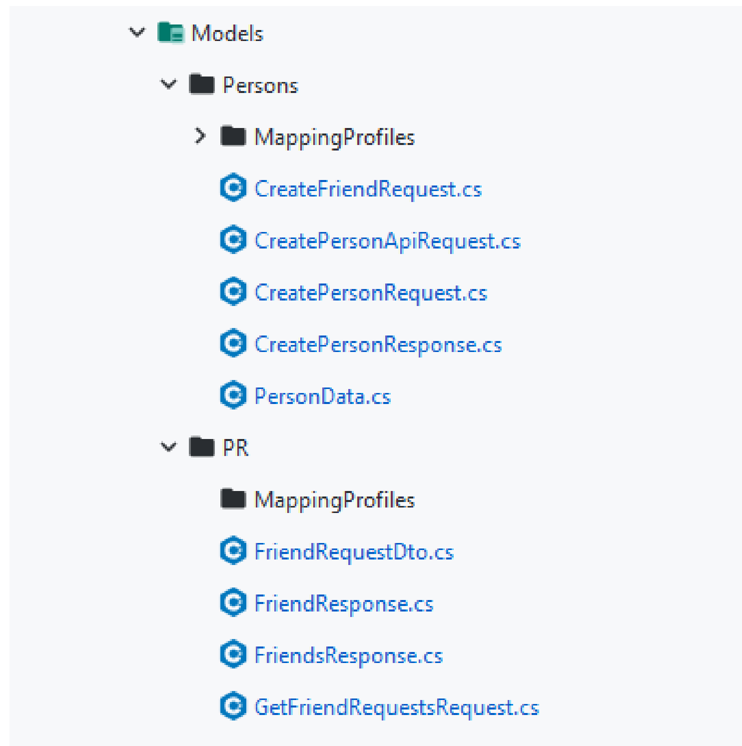
```
        var response = await
_client.AcceptFriendRequest(friendRequestId);

        if (response.IsFailure)
            return BadRequest();

        return Ok();
    }
}
```

```
[Route("api/v1/[controller]")]
[ApiController]
public class PersonsController : ControllerBase
{
  private readonly IPersonApiClient _client;

  public PersonsController(IPersonApiClient client)
  {
     _client = client;
  }

  [HttpGet]
  [ProducesResponseType((int)HttpStatusCode.BadRequest)]
  [ProducesResponseType((int)HttpStatusCode.OK)]
  public async Task<IActionResult> GetPersons()
  {
     var response = await _client.GetPersons();

     return Ok(response);
  }

  [HttpPost]
  [ProducesResponseType((int)HttpStatusCode.BadRequest)]
  [ProducesResponseType((int)HttpStatusCode.Created)]
  public async Task<IActionResult> CreatePersonAsync(CreatePersonRequest
person)
  {
     var response = await _client.CreatePersonAsync(person);

     if (response.IsFailure)
        return BadRequest();

     return Created(string.Empty, string.Empty);
  }
}
```

Infrastructure: Αποτελείται από ένα αρχείο, το GrpcExceptionInterceptor. Χρησιμοποιείται κατά την επικοινωνία με gRPC της πύλης με τις υπηρεσίες, με στόχο να διαχειριστεί το επιτυχημένο ή αποτυχημένο αίτημα gRPC. Σε περίπτωση αποτυχίας, αποθηκεύει το λόγο αυτής σε σύστημα καταγραφής αποτυχιών.

```
public class GrpcExceptionInterceptor : Interceptor
{
   private readonly ILogger<GrpcExceptionInterceptor> _logger;

   public GrpcExceptionInterceptor(ILogger<GrpcExceptionInterceptor> logger)
   {
       _logger = logger;
   }

   public override AsyncUnaryCall<TResponse> AsyncUnaryCall<TRequest,
TResponse>(
       TRequest request,
       ClientInterceptorContext<TRequest, TResponse> context,
       AsyncUnaryCallContinuation<TRequest, TResponse> continuation)
   {
       var call = continuation(request, context);

       return new
AsyncUnaryCall<TResponse>(HandleResponse(call.ResponseAsync),
call.ResponseHeadersAsync, call.GetStatus, call.GetTrailers, call.Dispose);
   }

   private async Task<TResponse> HandleResponse<TResponse>(Task<TResponse>
t)
   {
       try
       {
           var response = await t;
           return response;
       }
       catch (RpcException e)
       {
           _logger.LogError("Error calling via grpc: {Status} -
{Message}",     e.Status, e.Message);
           throw new Exception(e.Message);
       }
   }
}
```

Models: Ένα μοντέλο αντιπροσωπεύει τα δεδομένα της εφαρμογής και ένα ViewModel/Dto αντιπροσωπεύει δεδομένα που θα εμφανίζονται στη διεπαφή χρήστη. Ο φάκελος μοντέλα περιέχει όλες τις κατηγορίες των οντοτήτων, διαχωρισμένες ανά μικροϋπηρεσία.



Εικόνα 4-2: Τα μοντέλα του HTTP Gateway

Ενδεικτικά ο κώδικας μερικών Dtos και ViewModels:

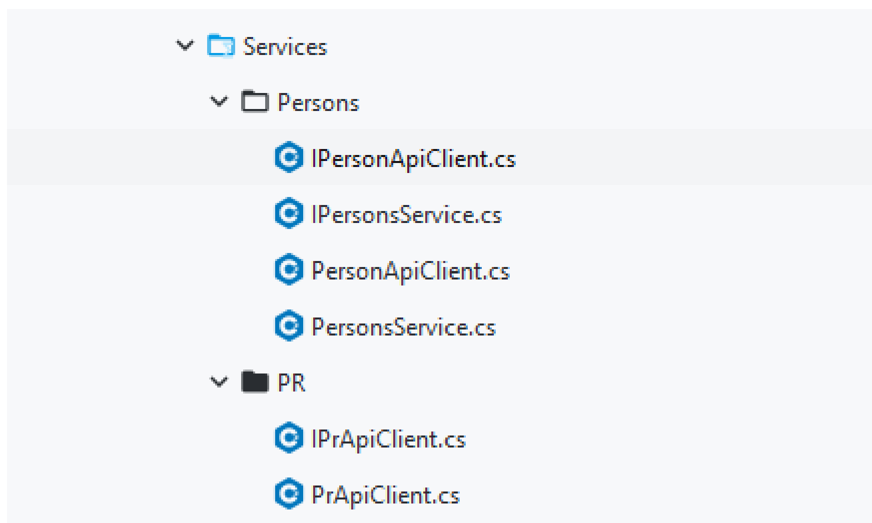
```
public class CreateFriendRequest
{
   public string SenderPersonIdentityGuid { get; set; }
   public string ReceiverPersonIdentityGuid { get; set; }
}


public class CreatePersonRequest
{
  public string Username { get; set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public string KnownAs { get; set; }
  public string Bio { get; set; }
}
```

Protos: Ο φάκελος που περιέχει τα Protocol Buffers του GRPC.

```
syntax = "proto3";

option csharp_namespace = "GrpcPersons";

package PersonsApi;

service PersonsGrpc {
 rpc CreatePerson(CreatePersonCommand) returns (CreatedPersonDto) {}
}

message CreatePersonCommand {
 string IdentityGuid = 1;
```

```
 string Username = 2;
 string FirstName = 3;
 string LastName = 4;
 string KnownAs = 5;
 string Bio = 6;
}

message CreatedPersonDto {
 string IdentityGuid = 1;
 string Username = 2;
 string FirstName = 3;
 string LastName = 4;
 string KnownAs = 5;
 string Bio = 6;
}
```

Services: Περιέχει τις διεπαφές και τις υλοποιήσεις τους, που απαιτούνται για την επικοινωνία μέσω HTTP της πύλης με τις μικροϋπηρεσίες.



Εικόνα 4-3: Τα services του API Gateway

Ενδεικτικά, ο κώδικας των PersonApiClient.cs και PersonsService. Είναι άξιο αναφοράς ότι αυτές οι δύο κλάσεις υλοποιούνται κατά το πρότυπο facade[13].

```
public class PersonApiClient : IPersonApiClient
{
   private readonly IMapper _mapper;
   private readonly IPrApiClient _prApiClient;
   private readonly HttpClient _httpClient;

   public PersonApiClient(IHttpClientFactory httpClientFactory, IMapper
mapper, IPrApiClient prApiClient)
   {
       _mapper = mapper;
       _prApiClient = prApiClient;
```

[13] Βλέπε: Facade pattern - Wikipedia

```
        _httpClient = httpClientFactory.CreateClient("Persons");
    }

    public async Task<Result> CreatePersonAsync(CreatePersonRequest request)
    {
        try
        {
            var internalRequest =
_mapper.Map<CreatePersonApiRequest>(request);
            var jsonBody = JsonSerializer.Serialize(internalRequest);
            var internalRequestBody = new StringContent(jsonBody,
                Encoding.UTF8, MediaTypeNames.Application.Json);

            var result = await
_httpClient.PostAsync(UrlsConfig.PersonsOperations.Base,
internalRequestBody);

            return !result.IsSuccessStatusCode ? Result.Failure("Error
Creating Person") : Result.Success();
        }
        catch (Exception e)
        {
            Console.WriteLine(e);
            throw;
        }
    }

    public async Task<IList<PersonData>> GetPersons()
    {
        var getPersonsResponseMessage = await
_httpClient.GetAsync(UrlsConfig.PersonsOperations.Base);

        if (!getPersonsResponseMessage.IsSuccessStatusCode)
            return new List<PersonData>();

        await using var contentStream =
            await getPersonsResponseMessage.Content.ReadAsStreamAsync();

        var persons = await
JsonSerializer.DeserializeAsync<IList<PersonData>>(contentStream);
        if (persons == null)
            return new List<PersonData>();
        foreach (var person in persons)
        {
            person.Friends = await
_prApiClient.GetFriendshipsAsync(person.IdentityGuid);
        }

        return persons;
    }

    public Task<PersonData> GetPersonAsync(PersonData personData)
    {
        throw new System.NotImplementedException();
    }
}
```

```
public class PersonsService : IPersonsService
{
  private readonly PersonsGrpc.PersonsGrpcClient _personsGrpcClient;
  private readonly ILogger<PersonsService> _logger;

  //private persons
  public PersonsService(PersonsGrpc.PersonsGrpcClient personsGrpcClient,
ILogger<PersonsService> logger)
  {
     _personsGrpcClient = personsGrpcClient;
     _logger = logger;
  }

  public async Task<PersonData> CreatePersonAsync(PersonData personData)
  {
     _logger.LogDebug("Grpc creating person {@PersonData}", personData);
     var createPersonCommand = MapToCreatePersonCommand(personData);
     var createdPersonDto = await
_personsGrpcClient.CreatePersonAsync(createPersonCommand);
     _logger.LogDebug("Grpc create person request {@Response}",
createdPersonDto);
     var response = new PersonData
     {
        IdentityGuid = createdPersonDto.IdentityGuid
     };
     return response;
  }

  private CreatePersonCommand MapToCreatePersonCommand(PersonData
personData)
  {
     var command = new CreatePersonCommand();
     command.Bio = personData.Bio;
     command.Username = personData.Username;
     command.FirstName = personData.FirstName;
     command.LastName = personData.LastName;
     command.IdentityGuid = personData.IdentityGuid;
     command.KnownAs = personData.KnownAs;

     return command;
  }
}
```

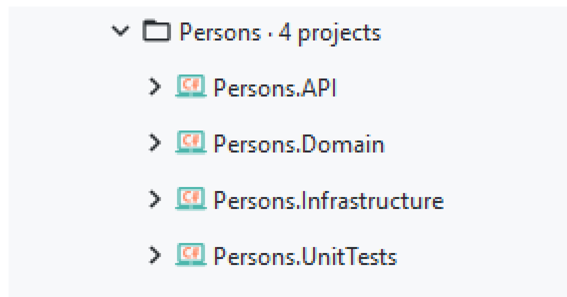
Program.cs: Αυτή η κλάση είναι το σημείο εισόδου της εφαρμογής. Κατασκευάζει τον κεντρικό διακομιστή και εκτελεί τη μέθοδο εκτέλεσης.

Startup.cs: Η κλάση αποτελεί το σημείο εισόδου στην εφαρμογή, ρυθμίζοντας τις υπηρεσίες διαμόρφωσης που θα χρησιμοποιεί η εφαρμογή. Επίσης αποτελεί το σημείο εισόδου υπηρεσιών στον IoC container [14].

[14] Βλέπε: Dependency injection - .NET | Microsoft Docs

4.1.2 Μικροϋπηρεσία Persons

Η μικροϋπηρεσία Persons, όπως ειπώθηκε στα προηγούμενα κεφάλαια, υλοποιήθηκε σύμφωνα με τις αρχές του Domain-Driven Design.



Εικόνα 4-4: Τα projects της μικροϋπηρεσίας Persons

Το επίπεδο διεπαφής (API) αποτελεί το HTTP REST API και το GRPC API.

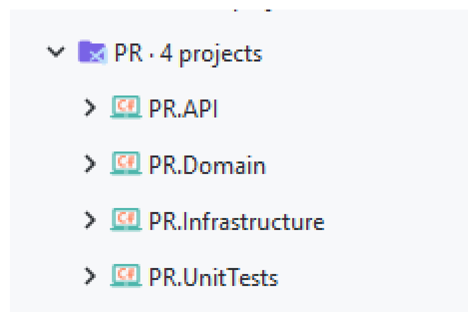
Ενδεικτικά, για λόγους πρακτικότητας, παρακάτω παρουσιάζεται ο κώδικας του συνόλου Persons.

```
public class PersonAggregate : Entity, IAggregateRoot
{
  public override int Id
  {
     get => _id;
  }

  private int _id;
  public string IdentityGuid { get; private set; }
  public string Username { get; }
  public string FirstName { get; }
  public string LastName { get; }
  public string KnownAs { get; }
  public string Bio { get; }

  public PersonAggregate()
  {
  }



  public PersonAggregate(string identityGuid, string username, string firstName,
string lastName, string knownAs, string bio) : this()
  {
     IdentityGuid = identityGuid;
     Username = !string.IsNullOrEmpty(username) ? username : throw
new      ArgumentNullException(nameof(username));
     FirstName = !string.IsNullOrEmpty(firstName) ? firstName : throw new
ArgumentNullException(nameof(firstName));
     LastName = !string.IsNullOrEmpty(lastName) ? lastName : throw new
ArgumentNullException(nameof(lastName));
     KnownAs = knownAs;
     Bio = bio;
  }

  public void SetIdentityGuid(string newIdentityGuid)
  {
     IdentityGuid = newIdentityGuid;
```

```
    }
}
```

4.1.3 Μικροϋπηρεσία PeopleRelations

Η μικροϋπηρεσία PeopleRelations(PR), όπως ειπώθηκε στα προηγούμενα κεφάλαια, υλοποιήθηκε σύμφωνα με τις αρχές του Domain-Driven Design.



Εικόνα 4-5: Τα projects της μικροϋπηρεσίας PeopleRelations

Ενδεικτικά, για λόγους πρακτικότητας, παρακάτω παρουσιάζεται ο κώδικας του συνόλου FriendRequestAggregate.

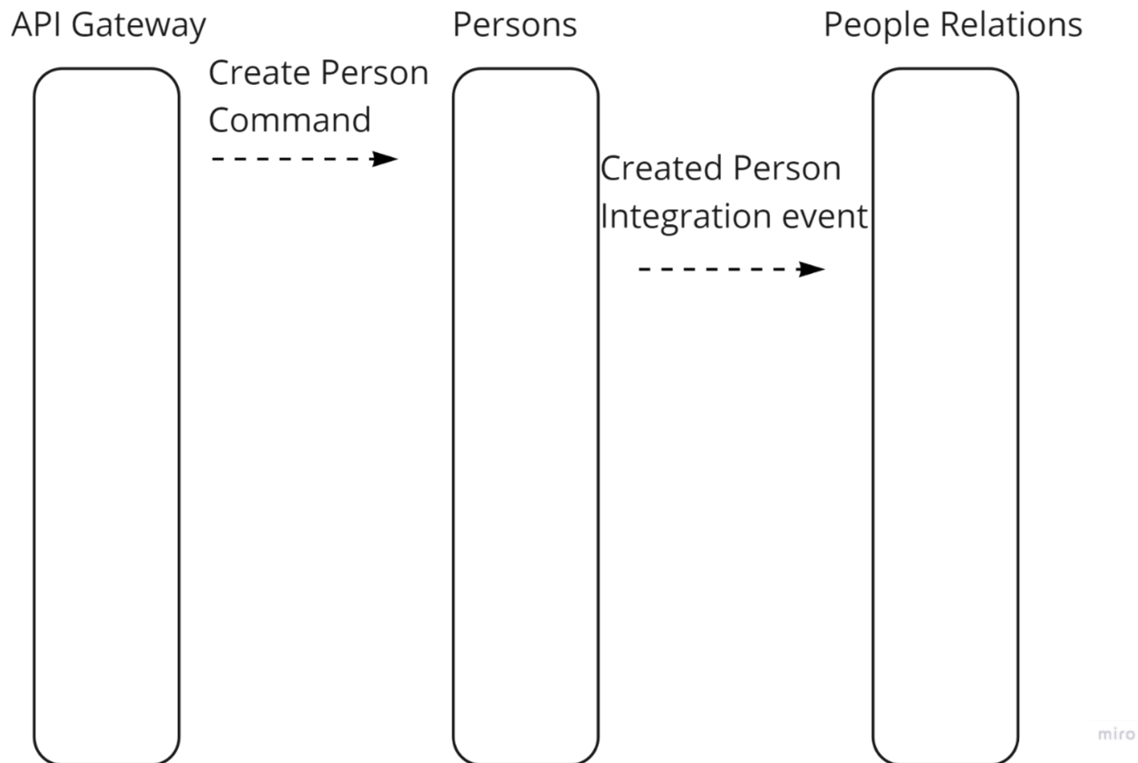
```
public class FriendRequest : Entity, IAggregateRoot
{
  public FriendRequest()
  {
  }

  public int SenderPersonId { get; }
  public int ReceiverPersonId { get; }
  public DateTimeOffset CreatedDate { get; }
  public int Modifier { get; }
  public DateTimeOffset? ModifiedDate { get; }
  public FriendRequestStatus FriendRequestStatus { get; private set; }
  private int _friendRequestStatusId;

  public FriendRequest(int senderIdentityId, int receiverIdentityId, int statusId)
  {
     SenderPersonId = senderIdentityId > 0
        ? senderIdentityId
        : throw new PRDomainException(nameof(senderIdentityId));
     ReceiverPersonId = receiverIdentityId > 0
        ? receiverIdentityId
        : throw new PRDomainException(nameof(receiverIdentityId));
     CreatedDate = DateTimeOffset.Now;
     Modifier = senderIdentityId;
     ModifiedDate = DateTimeOffset.Now;
     _friendRequestStatusId = statusId;
  }

  public bool IsEqualTo(int senderId, int receiverId)
  {
     return SenderPersonId == senderId
            && ReceiverPersonId == receiverId;
  }
```

```
  public void SetAcceptedFriendRequestStatus()
  {
     if (_friendRequestStatusId != FriendRequestStatus.AwaitingConfirmation.Id)
     {
        StatusChangeException(FriendRequestStatus.Confirmed);
     }

     _friendRequestStatusId = FriendRequestStatus.Confirmed.Id;
     AddDomainEvent(new FriendRequestAcceptedDomainEvent(this));
  }

  private void StatusChangeException(FriendRequestStatus friendRequestStatusToChange)
  {
     throw new PRDomainException(
        $"Is not possible to change the friend request status from
{FriendRequestStatus.Name} to {friendRequestStatusToChange.Name}.");
  }
}


public class FriendRequestStatus : Enumeration
{

  public static FriendRequestStatus AwaitingConfirmation =
     new FriendRequestStatus(1, nameof(AwaitingConfirmation).ToLowerInvariant());

  public static FriendRequestStatus Confirmed = new FriendRequestStatus(2,
nameof(Confirmed).ToLowerInvariant());
  public static FriendRequestStatus Removed = new FriendRequestStatus(3,
nameof(Removed).ToLowerInvariant());
  public static FriendRequestStatus Cancelled = new FriendRequestStatus(4,
nameof(Cancelled).ToLowerInvariant());

  public FriendRequestStatus(int id, string name) : base(id, name)
  {
  }
}
```

4.2 Διαγράμματα ροής

Σε αυτό το κεφάλαιο θα αναλυθούν τεχνικά οι τρεις βασικές λειτουργίες της πιλοτικής μας εφαρμογής.

4.2.1 Προσθήκη ατόμου στο σύστημα



Εικόνα 4-6: Διάγραμμα ροής της λειτουργίας προσθήκης ατόμου

Για τη δημιουργία ενός χρήστη στο σύστημα:

1. Ο πελάτης καλεί από την εξωτερική πύλη το HTTP POST τελικό σημείο /api/v1/Persons.

```
[HttpPost]
[ProducesResponseType((int)HttpStatusCode.BadRequest)]
[ProducesResponseType((int)HttpStatusCode.Created)]
public async Task<IActionResult> CreatePersonAsync(CreatePersonRequest person)
{
  var response = await _client.CreatePersonAsync(person);

  if (response.IsFailure)
     return BadRequest();

  return Created(string.Empty, string.Empty);
}
```

2. Η πύλη με τη σειρά της επικοινωνεί μέσω της εντολής GRPC CreatePersonCommand με την υπηρεσία Persons.

```
syntax = "proto3";

option csharp_namespace = "GrpcPersons";

package PersonsApi;

service PersonsGrpc {
 rpc CreatePerson(CreatePersonCommand) returns (CreatedPersonDto) {}
}
```

```
message CreatePersonCommand {
 string IdentityGuid = 1;
 string Username = 2;
 string FirstName = 3;
 string LastName = 4;
 string KnownAs = 5;
 string Bio = 6;
}

message CreatedPersonDto {
 string IdentityGuid = 1;
 string Username = 2;
 string FirstName = 3;
 string LastName = 4;
 string KnownAs = 5;
 string Bio = 6;
}
```

3. Καταχωρείται ο χρήστης στον πίνακα Persons της υπηρεσίας Persons.

```
/// <summary>
/// Create person command handler
/// </summary>
public class CreatePersonCommandHandler : IRequestHandler<CreatePersonCommand,
Result<string>>
{
  private readonly IPersonRepository _personRepository;
  private readonly ILogger<CreatePersonCommandHandler> _logger;
  private readonly IPersonIntegrationEventService _personIntegrationEventService;

  public CreatePersonCommandHandler(
     IPersonRepository personRepository,
     ILogger<CreatePersonCommandHandler> logger,
     IPersonIntegrationEventService personIntegrationEventService
     )
  {
     _personRepository = personRepository ?? throw new
ArgumentNullException(nameof(personRepository));
     _logger = logger ?? throw new ArgumentNullException(nameof(logger));
     _personIntegrationEventService = personIntegrationEventService;
  }


  public async Task<Result<string>> Handle(CreatePersonCommand request,
CancellationToken cancellationToken)
  {
     try
     {
        var person = new PersonAggregate(request.IdentityGuid, request.Username,
request.FirstName, request.LastName,
           request.KnownAs, request.Bio);

        if (string.IsNullOrEmpty(request.IdentityGuid))
           person.SetIdentityGuid(Guid.NewGuid().ToString());

        _logger.LogInformation("----- Creating Person: {@Person}", person);

        _personRepository.Add(person);

        var result = await
_personRepository.UnitOfWork.SaveEntitiesAsync(cancellationToken);

        if (!result) return Result.Failure<string>(string.Empty);

        var personCreatedIntegrationEvent = new
PersonCreatedIntegrationEvent(person.IdentityGuid);
```

```
            await
    _personIntegrationEventService.AddAndSaveEventAsync(personCreatedIntegrationEvent);

            return Result.Success(person.IdentityGuid);
        }
        catch (Exception e)
        {
            _logger.LogError(e, nameof(CreatePersonCommandHandler));
            Console.WriteLine(e);
            throw;
        }

    }
```

4. Εφόσον δημιουργηθεί επιτυχώς ο χρήστης, η υπηρεσία Persons δημοσιεύει ένα integration event, το CreatedPersonIntegrationEvent.

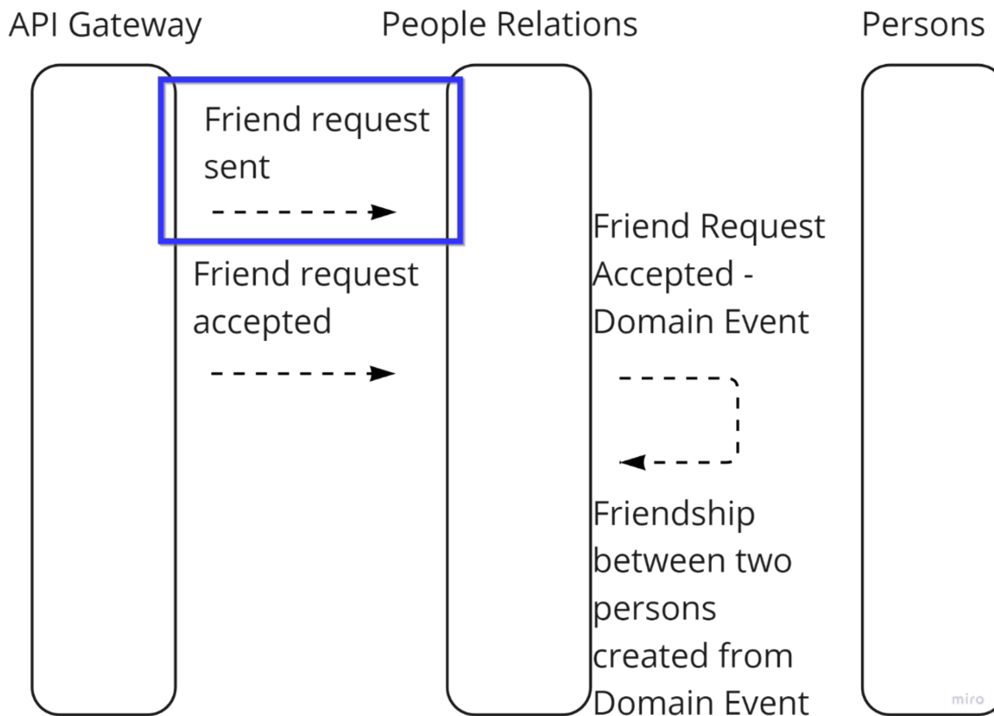
```
_personIntegrationEventService.AddAndSaveEventAsync(personCreatedIntegrationEve
nt);
```

5. Η υπηρεσία PeopleRelations, έχει εγγραφεί στο κανάλι μηνυμάτων ώστε να αντιλαμβάνεται και να αντιδράει στα καινούργια events CreatedPersonIntegrationEvent. Ακολούθως, μόλις εμφανιστεί το συγκεκριμένο event, προσθέτει με τη σειρά της τον καινούργιο χρήστη στο πίνακα Persons της υπηρεσίας.

```
public class PersonCreatedIntegrationEventHandler :
IIntegrationEventHandler<PersonCreatedIntegrationEvent>
{
  private readonly IMediator _mediator;
  private readonly ILogger<PersonCreatedIntegrationEventHandler> _logger;

  public PersonCreatedIntegrationEventHandler(
     IMediator mediator,
     ILogger<PersonCreatedIntegrationEventHandler> logger
  )
  {
     _mediator = mediator;
     _logger = logger;
  }

  public async Task Handle(PersonCreatedIntegrationEvent @event)
  {
     using (LogContext.PushProperty("IntegrationEventContext", $"{@event.Id}-
{Program.AppName}"))
     {
        _logger.LogInformation("----- Handling integration event: {IntegrationEventId}
at {AppName} - ({@IntegrationEvent})", @event.Id, Program.AppName, @event);

        var command = new CreatePersonCommand(@event.PersonId);

        _logger.LogInformation(
           "----- Sending command: {CommandName} - {IdProperty}: ({@Command})",
           command.GetGenericTypeName(),
           nameof(command.PersonId),
           command);
```

```
        await _mediator.Send(command);
    }
  }
}
```

4.2.2 Αποστολής αιτήματος φιλίας



Εικόνα 4-7: Διάγραμμα ροής λειτουργίας αποστολής αιτήματος φιλίας

Για τη δημιουργία ενός αιτήματος φιλίας στο σύστημα:

1. Ο πελάτης καλεί από την εξωτερική πύλη το HTTP POST τελικό σημείο /api/v1/FriendRequests.

```
[HttpPost]
[ProducesResponseType((int)HttpStatusCode.BadRequest)]
[ProducesResponseType((int)HttpStatusCode.Created)]
public async Task<IActionResult> CreateFriendRequest(CreateFriendRequest
request)
{
   var response = await _client.CreateFriendRequest(request);

   if (response.IsFailure)
       return BadRequest();

   return Created(string.Empty, string.Empty);
}
```

2. Η πυλή καλεί μέσω HTTP την υπηρεσία People Relations.

```
[HttpPost]
[ProducesResponseType((int)HttpStatusCode.OK)]
[ProducesResponseType((int)HttpStatusCode.BadRequest)]
public async Task<IActionResult> CreateFriendRequestAsync(
  [FromBody] CreateFriendRequestCommand createFriendRequestCommand)
```

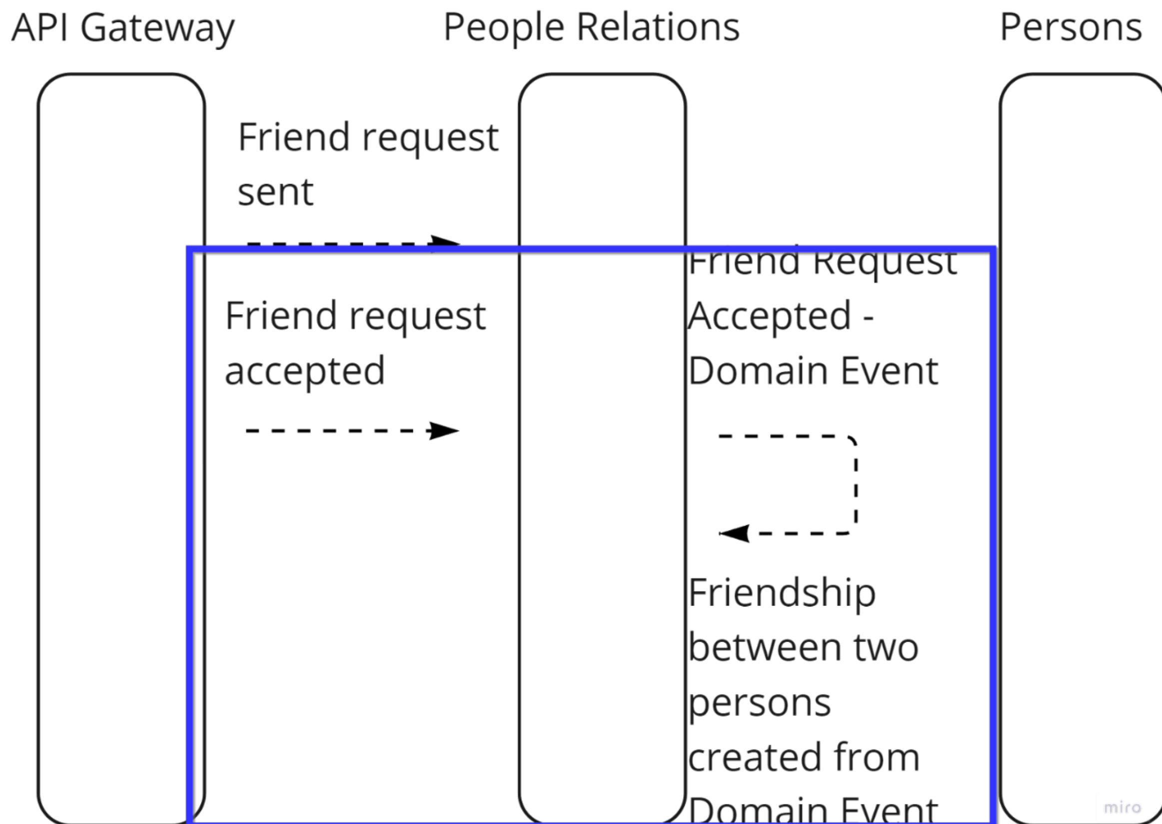
```
{
  _logger.LogInformation(
     "----- Sending command: {CommandName} - {IdProperty}: {CommandId} ({@Command})",
     createFriendRequestCommand.GetGenericTypeName(),
     nameof(createFriendRequestCommand.ReceiverPersonIdentityGuid),
     createFriendRequestCommand.SenderPersonIdentityGuid,
     createFriendRequestCommand);

  var result = await _mediator.Send(createFriendRequestCommand);
  if (result)
     return Ok();

  return BadRequest();
}
```

3. Στην υπηρεσία Person Relations, δημιουργείται στον πίνακα FriendRequests μια καινούργια εγγραφή με Status AwaitingConfirmation

```
public class CreateFriendRequestCommandHandler :
IRequestHandler<CreateFriendRequestCommand, bool>
{
  private readonly IFriendRequestRepository _friendRequestRepository;
  private readonly IPersonRepository _personRepository;
  private readonly ILogger<CreateFriendRequestCommandHandler> _logger;

  public CreateFriendRequestCommandHandler(IFriendRequestRepository
friendRequestRepository,
     IPersonRepository personRepository,
     ILogger<CreateFriendRequestCommandHandler> logger)
  {
     _friendRequestRepository =
        friendRequestRepository ?? throw new
ArgumentNullException(nameof(friendRequestRepository));
     _personRepository = personRepository ?? throw new
ArgumentNullException(nameof(personRepository));
     ;
     _logger = logger ?? throw new ArgumentNullException(nameof(logger));
  }

  /// <summary>
  /// Handler which process then command when person sends a friend request to
another person
  /// </summary>
  /// <param name="request"></param>
  /// <param name="cancellationToken"></param>
  /// <returns></returns>
  /// <exception cref="NotImplementedException"></exception>
  public async Task<bool> Handle(CreateFriendRequestCommand request,
CancellationToken cancellationToken)
  {
     try
     {
        var sender = await
_personRepository.FindAsync(request.SenderPersonIdentityGuid);
        var receiver = await
_personRepository.FindAsync(request.ReceiverPersonIdentityGuid);

        if (sender == null)
           return false;
        if (receiver == null)
           return false;
```

```
            var exists = await _friendRequestRepository.Exists(sender.Id, receiver.Id);
            if (exists)
                return false;

            var friendRequest =
                new Domain.AggregatesModel.FriendRequestAggregate.FriendRequest(sender.Id,
                    receiver.Id, FriendRequestStatus.AwaitingConfirmation.Id);

            _logger.LogInformation("----- Creating FriendRequest - FriendRequest:
{@FriendRequest}", friendRequest);

            _friendRequestRepository.Add(friendRequest);

            return await
_friendRequestRepository.UnitOfWork.SaveEntitiesAsync(cancellationToken);
        }
        catch (Exception e)
        {
            Console.WriteLine(e);
            throw;
        }
    }
}
```

4.2.3 Αποδοχή αιτήματος φιλίας και δημιουργία φιλίας ανάμεσα στους χρήστες



Εικόνα 4-8: Διάγραμμα ροής λειτουργίας αποδοχή αιτήματος και δημιουργία φιλίας

Για την αποδοχή ενός αιτήματος φιλίας και τη δημιουργία φιλίας ανάμεσα στους χρήστες:

1. Ο πελάτης καλεί απο την εξωτερική πύλη το HTTP PUT τελικό σημείο /api/v1/FriendRequests/{friendRequestId}/accept.

```
[Route("{friendRequestId}/accept")]
[HttpPut]
[ProducesResponseType((int)HttpStatusCode.BadRequest)]
[ProducesResponseType((int)HttpStatusCode.OK)]
public async Task<IActionResult> AcceptFriendRequest(int friendRequestId)
{
   var response = await _client.AcceptFriendRequest(friendRequestId);

   if (response.IsFailure)
       return BadRequest();

   return Ok();
}
```

2. Η πύλη καλεί μέσω HTTP την υπηρεσία People Relations.

```
[Route("{friendRequestId}/accept")]
[HttpPut]
[ProducesResponseType((int)HttpStatusCode.BadRequest)]
[ProducesResponseType((int)HttpStatusCode.OK)]
public async Task<IActionResult> AcceptFriendRequest(int friendRequestId)
{
   var response = await _client.AcceptFriendRequest(friendRequestId);

   if (response.IsFailure)
       return BadRequest();

   return Ok();
}
```

3. Αν το FriendRequestStatus είναι AwaitingConfirmation, αποδέχεται το αίτημα. Η αποδοχή του αιτήματος σημαίνει ότι το FriendRequestStatus περνάει στη κατάσταση Confirmed.

```
public class AcceptFriendRequestCommandHandler :
IRequestHandler<AcceptFriendRequestCommand, bool>
{
  private readonly IFriendRequestRepository _friendRequestRepository;
  private readonly ILogger<CreateFriendRequestCommandHandler> _logger;
  private readonly IMediator _mediator;

  public AcceptFriendRequestCommandHandler(IFriendRequestRepository
friendRequestRepository,
     ILogger<CreateFriendRequestCommandHandler> logger,
     IMediator mediator)
  {
     _friendRequestRepository =
        friendRequestRepository ?? throw new
ArgumentNullException(nameof(friendRequestRepository));
     _logger = logger ?? throw new ArgumentNullException(nameof(logger));
     _mediator = mediator ?? throw new ArgumentNullException(nameof(mediator));
  }
```

```
  public async Task<bool> Handle(AcceptFriendRequestCommand request,
CancellationToken cancellationToken)
  {
     try
     {
        var friendRequestToAccept = await
_friendRequestRepository.FindByIdAsync(request.FriendRequestId);
        if (friendRequestToAccept == null) return false;

        friendRequestToAccept.SetAcceptedFriendRequestStatus();
        var result =  await
_friendRequestRepository.UnitOfWork.SaveEntitiesAsync(cancellationToken);
        return result;
     }
     catch (Exception e)
     {
        Console.WriteLine(e);
        throw;
     }
  }
}
```

4. Η υπηρεσία People Relations και συγκεκριμένα το σύνολο PersonAggregate δημοσιεύει ένα καινούργιο Domain Event FriendRequestAcceptedEvent ώστε να επικοινωνήσει με το έτερο σύνολο PersonAggregate της υπηρεσίας, με σκοπό τη δημιουργία της φιλίας ανάμεσα στους χρήστες.

```
public void SetAcceptedFriendRequestStatus()
{
  if (_friendRequestStatusId != FriendRequestStatus.AwaitingConfirmation.Id)
  {
     StatusChangeException(FriendRequestStatus.Confirmed);
  }

  _friendRequestStatusId = FriendRequestStatus.Confirmed.Id;
  AddDomainEvent(new FriendRequestAcceptedDomainEvent(this));
}
```

5. Το σύνολο Person διαχειρίζεται το παραπάνω event μέσω του διαχειριστήAddPersonFriendshipWhenFriendshipRequestAcceptedEventHandler, δημιουργώντας μια καινούργια εγγραφή στο πίνακα Friendships.

```
public class AddPersonFriendshipWhenFriendshipRequestAcceptedEventHandler :
INotificationHandler<FriendRequestAcceptedDomainEvent>
{
  private readonly IPersonRepository _personRepository;
  private readonly ILoggerFactory _logger;

  public AddPersonFriendshipWhenFriendshipRequestAcceptedEventHandler()
  {
  }
```

```
  public
AddPersonFriendshipWhenFriendshipRequestAcceptedEventHandler(IPersonRepository
personRepository, ILoggerFactory logger)
  {
     _personRepository = personRepository ?? throw new
ArgumentNullException(nameof(personRepository));
     _logger = logger;
  }

  public async Task Handle(FriendRequestAcceptedDomainEvent notification,
CancellationToken cancellationToken)
  {
     var sender = await
_personRepository.FindByIdAsync(notification.FriendRequest.SenderPersonId);
     if (sender == null) throw new ArgumentNullException(nameof(notification));

     sender.AddFriendship(notification.FriendRequest.SenderPersonId,
notification.FriendRequest.ReceiverPersonId);

      _logger.CreateLogger<AddPersonFriendshipWhenFriendshipRequestAcceptedEventHandl
er>()
         .LogTrace("Person with Id: {Id} has been successfully create a friendship
with person {Id2}",
            notification.FriendRequest.SenderPersonId,
notification.FriendRequest.ReceiverPersonId);
  }
}
```

5 Επίλογος

5.1 Σύνοψη και συμπεράσματα

Στη παρούσα διπλωματική εργασία διαπιστώσαμε και επιλύσαμε κάποιες από τις προκλήσεις της αρχιτεκτονικής μικροϋπηρεσιών, όπως η απρόκλητη επικοινωνία μεταξύ των στοιχείων του συστήματος, ο συγχρονισμός και η τελική συνέπεια των δεδομένων τους. Στη συνέχεια ερευνήσαμε τη σχεδίαση της κάθε μικροϋπηρεσίας με τη προσέγγισης του σχεδιασμού βάσει τομέα (Domain-Driven Design) με απώτερο στόχο να ακολουθήσουμε τον όρο, κατά τον Robert C. Martin, της 'αρχή της ενιαίας ευθύνης'. Η αρχή δηλώνει: «συγκεντρώστε εκείνα τα πράγματα που αλλάζουν για τον ίδιο λόγο και διαχωρίστε αυτά που αλλάζουν για διαφορετικούς λόγους». Με το διαχωρισμό των λογικών σημείων της εφαρμογής σε επιμέρους αυτόνομες μικροϋπηρεσίες, καταφέραμε οι υπηρεσίες αυτές να είναι αυτόνομες, εύκολες στη κατανόηση και πλήρως ελεγμένες.

5.2 Όρια και περιορισμοί της έρευνας

Η πιλοτική εφαρμογή, καθώς υλοποιήθηκε με εκπαιδευτικό γνώμονα, έχει αρκετά περιθώρια βελτίωσης. Οι λειτουργίες είναι απλές, καθώς είχαν ως στόχο την εξερεύνηση της αρχιτεκτονικής των μικροϋπηρεσιών. Επίσης δεν ερευνήθηκαν λύσεις

που αφορούν το παραγωγικό περιβάλλον, όπως κάποιος container orchestrator [15], καθώς θεωρήθηκαν εκτός θέματος.

5.3 Μελλοντικές επεκτάσεις

Στην παρούσα εργασία δόθηκε βαρύτητα στην ανάπτυξη της εφαρμογής σε τοπικό περιβάλλον. Ως μελλοντική επέκταση προκύπτει η μεταφορά της σε διαδικτυακό απομακρυσμένο παραγωγικό περιβάλλον. Τέλος, θα μπορούσαμε να επεκτείνουμε το εύρος των λειτουργιών της εφαρμογής, προσθέτοντας και άλλες λειτουργίες.

[15] Βλέπε: What is container orchestration? (redhat.com)

Βιβλιογραφία

[1] Domain-Driven Design – Tackling Complexity in the Hearth of Software. Eric Evans, 2003.
[2] BoundedContext (martinfowler.com)
[3] Clean Architecture: A craftsman's Guide to Software Structure and Design, Robert C. Martin (Uncle Bob), 2008.
[4] Patterns of Enterprise Application Architecture, Martin Fowler, 2002.
[5] Implementing Domain-Driven Design, Vaugn Vernon, 2013
[6] Shared database (microservices.io)
[7] https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf CQRS - Greg Young, 2010.
[8] CQRS Course | Pluralsight CQRS in Practice, Introduction, Vladimir Khorikov.
[9] Πρωτόκολλο Μεταφοράς Υπερκειμένου - Βικιπαίδεια (wikipedia.org)
[10] What is a REST API? (redhat.com)
[11] gRPC
[12] Microsoft SQL Server - Βικιπαίδεια (wikipedia.org)
[13] Part 1: RabbitMQ for beginners - What is RabbitMQ? - CloudAMQP
[14] Docker overview | Docker Documentation
[15] Folder Structure Of ASP.NET Core MVC 6.0 Project (c-sharpcorner.com)