Πανεπιστήμιο Πειραιώς – Τμήμα Πληροφορικής

Πρόγραμμα Μεταπτυχιακών Σπουδών

«Προηγμένα Συστήματα Πληροφορικής»

**Μεταπτυχιακή Διατριβή**

| | |
|---|---|
| Τίτλος Διατριβής | **Κεντρικοποιημένο σύστημα αναζήτησης προϊόντων σε κρυπτογραφημένες εγγραφές.**<br><br>**Centralized product search system on hashed records.** |
| Ονοματεπώνυμο Φοιτητή | **Αρώνης Κωνσταντίνος-Ανέστης** |
| Πατρώνυμο | **Αλέξανδρος** |
| Αριθμός Μητρώου | **ΜΠΣΠ/ 18002** |
| Επιβλέπων | **Ευάγγελος Σακκόπουλος, Επίκουρος καθηγητής** |

Ημερομηνία Παράδοσης: Οκτώβριος 2020

Τριμελής Εξεταστική Επιτροπή

| | | |
|---|---|---|
| (υπογραφή) | (υπογραφή) | (υπογραφή) |
| Μαρία Βίρβου<br>Καθηγήτρια | Ευθύμιος Αλέπης<br>Αναπληρωτής καθηγητής | Ευάγγελος Σακκόπουλος<br>Επίκουρος καθηγητής |

# Περιεχόμενα


1 Εισαγωγή – Σύντομη Περιγραφή ..... 6

2. Θεωρητικό υπόβαθρο ..... 7

2.1 Συναρτήσεις Κατακερματισμού (Hash Functions). ..... 7

2.2 Κατακερματισμός ευαίσθητος στην τοποθεσία - Locality-Sensitive Hashing ..... 8

2.3 MinHash ..... 8

2.4 Δείκτης Jaccard ..... 9

2.5 Σταθμισμένο (Weighted) LSH ..... 9

3. Τεχνολογίες και Αρχιτεκτονική Συστήματος ..... 11

3.1 Περίληψη Κεφαλαίου ..... 11

3.2 .Net Core ..... 11

3.3 ASP .Net Core ..... 12

3.4 Windows Presentation Foundation ..... 12

3.5 Docker ..... 13

3.6 Entity Framework Core ..... 14

3.7 Αρχιτεκτονική συστήματος ..... 15

3.8 Αρχιτεκτονική βάσεων δεδομένων ..... 17

4. Υλοποίηση Συστήματος ..... 18

4.1 Περίληψη κεφαλαίου ..... 18

4.2 Υλοποίηση LSH (LSHDotNet) ..... 18

4.2.1 MinHasher ..... 18

4.2.2 LSHSearch ..... 20

4.2.3 Παράδειγμα χρήσης της βιβλιοθήκης ..... 23

4.3 EncryptedAuctionDatatypes ..... 24

4.4 EncryptedAuctionAggregator ..... 25

4.4.1 Μοντέλα & βάση δεδομένων ..... 25

4.4.2 Λειτουργικότητα ..... 29

4.5 EncryptedAuctionStore ..... 34

4.5.1 Μοντέλα & Βάση Δεδομένων ..... 35

4.5.2 Λειτουργικότητα ..... 41

4.6 EncryptedAuctionClient ..... 44

4.6.1 Εικαστικός σχεδιασμός ..... 44

4.6.2 Λειτουργικότητα ..... 45

5. Εκτέλεση και χρήση συστήματος ..... 49

5.1 Εκτέλεση υπηρεσιών ..... 49

5.2 Εκτέλεση αναζήτησης μέσω της εφαρμογής ..... 50

6. Συμπεράσματα και πιθανές επεκτάσεις ..... 57
6.1 Συμπεράσματα ..... 57
6.2 Μελλοντικές επεκτάσεις ..... 57
6.3 Γνώσεις που αποκτήθηκαν ..... 58
**Βιβλιογραφία** ..... 59

## Περίληψη

Στην παρούσα εργασία παρουσιάζεται ένα web-based κεντρικοποιημένο σύστημα αναζήτησης προϊόντων το οποίο παρέχει ανωνυμία όσων αφορά στις αναζητήσεις του χρήστη και τα αποτελέσματα αυτών. Πιο συγκεκριμένα περιφερειακά μικρά ή μεγαλύτερα μαγαζιά (EncryptedAuctionStore) καταχωρούν τα προϊόντα τους, σε κατακερματισμένη (hashed) μορφή, στη βάση δεδομένων μίας κεντρικής υπηρεσίας (EncryptedAuctionAggregator). Κατόπιν, πελάτες που θέλουν να αναζητήσουν ένα προϊόν στέλνουν μέσω του προγράμματος αναζήτησης (EncryptedAuctionClient) σε κατακερματισμένη μορφή τους όρους αναζήτησής τους. Για την λειτουργία της αναζήτησης σε κατακερματισμένες εγγραφές υλοποιήθηκε χρησιμοποιήθηκε η αλγοριθμική τεχνική Locality Sensitivity Hashing.

## Abstract

This postgraduate thesis presents a web-based centralized product search system which provides anonymity on user’s searches and the corresponding results. Small stores (EncryptedAuctionStore) can register their products, in a hashed form, in the database of a central service (EncryptedAuctionAggregator). Then, customers interested in searching for a product can, using the client application (EncryptedAuctionClient), post a hashed search query. Locality Sensitivity Hashing has been implemented and used for the function of searching on hashed records.

# 1 Εισαγωγή – Σύντομη Περιγραφή

Αντικείμενο της παρούσας διατριβής ήταν η εξερεύνηση των δυνατοτήτων που μας δίνει ο αλγόριθμος του LSH (Locality Sensitivity Hashing) στην συσσώρευση εγγραφών από διαφορετικές πηγές σε ένα κεντρικό υπολογιστικό σύστημα και στην συσχέτιση-αναζήτηση αυτών από χρήστες διατηρώντας παράλληλα αδιάβλητα τα δεδομένα. Θέλαμε επίσης να αντιμετωπίσουμε το πρόβλημα της εσφαλμένης αναζήτησης του χρήστη, τις περιπτώσεις δηλαδή που ο χρήστης είτε δεν γνωρίζει ακριβώς το όνομα, είτε κάνει τυπογραφικό λάθος κατά την αναζήτηση.

Έτσι περιγράφουμε και υλοποιούμε το σκελετό ενός συστήματος στο οποίο τον ρόλο των δεδομένων έχουν τα προϊόντα, τον ρόλο των πηγών δεδομένων έχουν τα μαγαζιά, τον ρόλο των χρηστών έχουν οι πελάτες.

Η αρχιτεκτονική του συστήματος μας παρέχει πολλαπλά πλεονεκτήματα, τόσο για τα μαγαζιά όσο και για τους χρήστες-καταναλωτές.

- Ανωνυμία της αναζήτησης του χρήστη. Ο κεντρικός κόμβος δεν γνωρίζει και δεν θα μπορούσε να γνωρίζει τι προϊόντα αναζητά ο κάθε χρήστης
- Ανωνυμία των προσφερόμενων προϊόντων κάθε μαγαζιού. Στον κεντρικό κόμβο αποθηκεύεται μονάχα η κατακερματισμένη (hashed) εκδοχή του προϊόντος
- Πιθανή διόρθωση λαθών των όρων αναζήτησης του χρήστη. Καθώς ο αλγόριθμος δεν λειτουργεί με απόλυτο ταίριασμα εγγραφών αλλά με δείκτη ομοιότητας μεταξύ αναζήτησης και εγγραφών.
- Δυνατότητα αποθήκευσης μεγάλου όγκου προϊόντων από μικρό σχετικά μαγαζί. Καθώς η επεξεργαστικά απαιτητική λειτουργία της αναζήτησης έχει μετατεθεί στον κεντρικό κόμβο, τα επιμέρους μαγαζιά μπορούν με ελάχιστους υπολογιστικούς πόρους να εκθέσουν μια μεγάλη ποσότητα προϊόντων προς αναζήτηση.

Η εργασία χωρίζεται σε πέντε κεφάλαια. Το πρώτο κεφάλαιο αφορά το θεωρητικό υπόβαθρο των αλγορίθμων που χρησιμοποιήθηκαν, με κεντρικό τον αλγόριθμο LSH. Το δεύτερο κεφάλαιο αναλύει και εξηγεί την αρχιτεκτονική του συστήματος καθώς και τις τεχνολογίες και εργαλεία που χρησιμοποιήθηκαν (.Net core, docker). Στο τρίτο κεφάλαιο παρουσιάζονται στοιχεία της υλοποίησης τόσο τού αλγόριθμου LSH όσο και του συστήματος με παραδείγματα κώδικα και διαγραμμάτων κλάσεων, στο τέλος του κεφαλαίου θα παρουσιαστεί συνοπτικά η μεθοδολογία για εκτέλεση του συστήματος. Το τέταρτο κεφάλαιο περιέχει τις βασικές περιπτώσεις χρήσης, τόσο σε μορφή διαγράμματος όσο και σαν παραδείγματα εκτέλεσης του προγράμματος καθώς και τις επικοινωνίες μεταξύ των υπηρεσιών. Το πέμπτο και τελευταίο κεφάλαιο παρουσιάζει τα τελικά συμπεράσματα αλλά και πιθανές μελλοντικές επεκτάσεις καθώς και εναλλακτικούς τρόπους χρήσης τής ίδιας αρχιτεκτονικής.

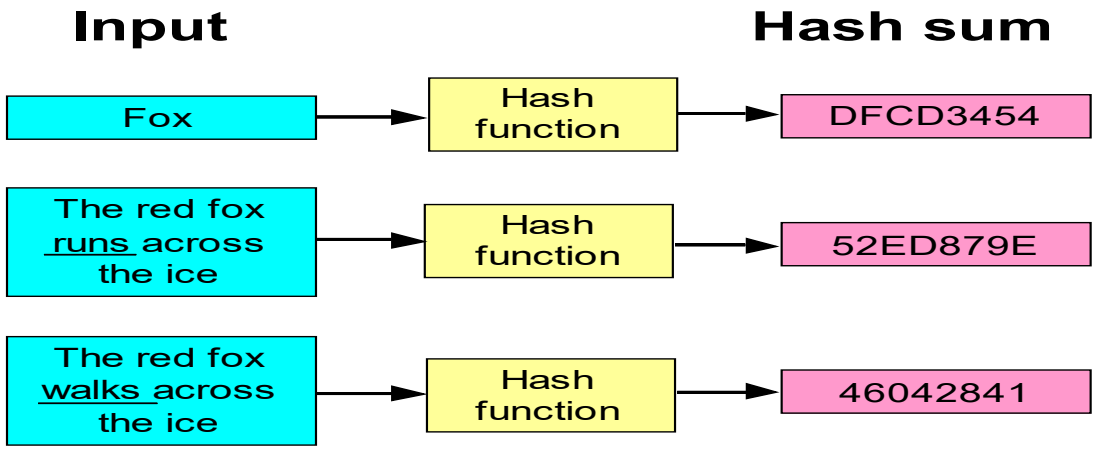
# 2. Θεωρητικό υπόβαθρο

## 2.1 Συναρτήσεις Κατακερματισμού (Hash Functions).

Συναρτήσεις κατακερματισμού ονομάζουμε συναρτήσεις πού δεχόμενες ως είσοδο δεδομένα τυχαίου μεγέθους παράγουν ως αποτέλεσμα μια αναπαράσταση των δεδομένων με προκαθορισμένο μέγεθος. Το αποτέλεσμα μιας τέτοιας συνάρτησης ονομάζεται κατακερματισμός ή hash όπως έχει επικρατήσει. Οι συναρτήσεις αυτές είναι μονόδρομες καθώς δεν γίνεται από τα αποτελέσματα (hashes) να προκύψουν οι αρχικές τιμές των δεδομένων.

Μια καλή συνάρτηση κατακερματισμού πρέπει ταυτόχρονα να είναι γρήγορη στην επεξεργασία και να παράγει ελάχιστες «συγκρούσεις». Σύγκρουση ονομάζουμε την παραγωγή ίδιου αποτελέσματος για περισσότερες τις μίας τιμές εισόδου.

Οι συναρτήσεις αυτές λόγω των χαρακτηριστικών τους βρίσκουν εφαρμογή στην κρυπτογραφία, στην επιβεβαίωση ακεραιότητας δεδομένων και προγραμμάτων στην δημιουργία μοναδικών κλειδιών (hash tables) και σε πολλούς άλλους τομείς της σύγχρονης πληροφορικής.



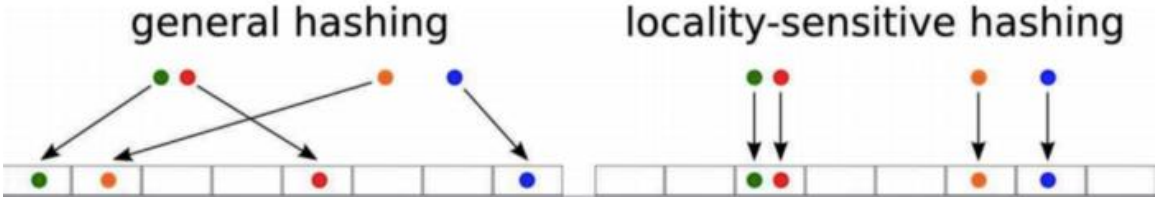

*Εικόνα 1. Απεικόνιση παραδειγμάτων εισόδου - εξόδου συνάρτησης κατακερματισμού.*

## 2.2 Κατακερματισμός ευαίσθητος στην τοποθεσία - Locality-Sensitive Hashing

Τοπικά-ευαίσθητο κατακερματισμό (locality-sensitive hashing) (Piotr & Rajeev, 1988) ονομάζουμε την αλγοριθμική τεχνική με την οποία δεδομένα εισόδου που είναι μεταξύ τους όμοια θα παράγουν εξίσου όμοια «δεδομένα» (hashes) εξόδου. (Jure, et al., 2010). Σε αντίθεση με τους κλασσικούς αλγορίθμους κατακερματισμού που προσπαθούν να αυξήσουν την απόσταση μεταξύ των παραγόμενων αποτελεσμάτων για αποφυγή «συγκρούσεων» οι αλγόριθμοι LSH μειώνουν την απόσταση των αποτελεσμάτων ώστε τα «κοντινά δεδομένα» να παραμείνουν κοντά.

Οι τεχνικές LSH χρησιμοποιούνται ως επί το πλείστων για να επιτρέψουν την αναζήτηση σε δεδομένα πολλών διαστάσεων ή μεγάλου όγκου. Οι εφαρμογές της τεχνικής περιλαμβάνουν:

- Αναζητήσεις ομοιοτήτων σε γενεαλογικά δεδομένα (Dumitru, et al., 2010) (Konstantin, et al., 2015)
- Αναζητήσεις ομοιοτήτων-αντιγραφών σε κείμενα (Jure, et al., 2010)
- Συστήματα προτάσεων (recommendation systems) (Lianyong, et al., 2017)
- Αναζήτηση ομοιοτήτων σε εικόνες (Yushi & Shumeet, 2008)




*Εικόνα 2.Διαγραμματική απεικόνιση διαφοράς κλασσικών αλγορίθμων hashing με το LSH.*

Πέρα από την ελαχιστοποίηση των διαστάσεων των δεδομένων που ήταν και ο αρχικός στόχος της έρευνας γύρω από το θέμα η τεχνική αυτή μας δίνει και άλλες δυνατότητες. Όπως η δυνατότητα να εντοπίσουμε «κοντινότερους γείτονες» στα παραγόμενα hashes χωρίς να έχουμε πρόσβαση στα πραγματικά δεδομένα. Αυτό παρέχει στα συστήματα που χρησιμοποιούν την τεχνική την απαραίτητη ανωνυμία στα δεδομένα ώστε να τα καθιστά κατάλληλα για διαχείριση ευαίσθητων δεδομένων, όπως προσωπικά, ιατρικά και οικονομικά.

Έχουν προταθεί διάφορες μεθοδολογίες υλοποίησης του LSH, στην παρούσα εργασία υλοποιήθηκε και χρησιμοποιήθηκε μια απλή υλοποίηση της πιο διαδεδομένης μεθοδολογίας, του Min-Hash (Broder, 1997)

## 2.3 MinHash

Η τεχνική MinHash χρησιμοποιείται για γρήγορο υπολογισμό της ομοιότητας μεταξύ δύο συνόλων. Αναπτύχθηκε για να χρησιμοποιηθεί από την μηχανή αναζήτησης AltaVista για τον εντοπισμό διπλότυπων ιστοσελίδων.

Για τον υπολογισμό του hash μιας τιμής με την τεχνική του MinHash ακολουθούνται τα εξής βήματα:

- Ορίζονται k διαφορετικές συναρτήσεις κατακερματισμού.
- Η τιμή που θέλουμε να κατακερματίσουμε χωρίζεται σε σύμβολα ( λέξεις ή χαρακτήρες, ανάλογα με την εφαρμογή)
- Για κάθε σύμβολο υπολογίζουμε το hash που παράγεται από κάθε μία από τις k συναρτήσεις.
- Αποθηκεύουμε την χαμηλότερη τιμή που παράχθηκε από οποιαδήποτε σύμβολο για κάθε από τις συναρτήσεις
- Το τελικό hash είναι η αλληλουχία των χαμηλότερων παραχθέντων τιμών από οποιοδήποτε σύμβολο. Έτσι το τελικό μέγεθος κάθε τιμής είναι k ακέραιοι.

Για τον υπολογισμό της ομοιότητας μεταξύ δύο hashes, και άρα κατά συνέπεια και της ομοιότητας μεταξύ των αρχικών τιμών χρησιμοποιούνται μέτρα ομοιότητας. Στην παρούσα υλοποίηση έχει χρησιμοποιηθεί ο δείκτης Jaccard.

## 2.4 Δείκτης Jaccard

Γνωστό και ως συντελεστής ομοιότητας Jaccard, είναι ένα στατιστικό μέτρο που υπολογίζει την ομοιότητα μεταξύ δύο συνόλων. Για τον υπολογισμό του δείκτη Jaccard μεταξύ δύο συνόλων διαιρούμε την τομή με την ένωση των συνόλων όπως φαίνεται και στον παρακάτω τύπο.

$$J(A,B) = \frac{|\ A\ \cap\ B\ |}{|\ A\ \cup\ B\ |}$$

*Εικόνα 3. Δείκτης Jaccard*

## 2.5 Σταθμισμένο (Weighted) LSH

Μια πλειάδα τεχνικών έχουν αναπτυχθεί για την εισαγωγή βαρών στον υπολογισμό του MinHash. Στην παρούσα εργασία έχουμε αποδώσει διαφορετικό βάρος σε κάθε χαρακτηριστικό των προϊόντων. Θεωρούμε για παράδειγμα το μοντέλο ενός προϊόντος σημαντικότερο από την μάρκα του προϊόντος κατά την αναζήτηση. Γενικεύοντας για **n** χαρακτηριστικά.

Για να αποθηκεύσουμε τα δεδομένα:

- Αποδίδουμε ένα βάρος W σε κάθε χαρακτηριστικό ενός δεδομένου μας.
- Το σύνολο των βαρών δεν πρέπει να ξεπερνάει τη μονάδα.
- Υπολογίζουμε ξεχωριστά το hash για κάθε χαρακτηριστικό. Δημιουργούνται έτσι n διαφορετικά hashes.
- Αποθηκεύουμε τα hashes διατηρώντας επίσης τα αντίστοιχα βάρη.

Όταν θέλουμε να βρούμε δεδομένα όμοια με μία αναζήτηση:

- Υπολογίζουμε το hash των όρων αναζήτησης.
- Για κάθε προϊόν:
  - Υπολογίζουμε τον δείκτη Jaccard μεταξύ του hash της αναζήτησης και κάθε ενός από τα hashes των χαρακτηριστικών ξεχωριστά.
  - Πολλαπλασιάζουμε τον παραγόμενο δείκτη Jaccard με το αντίστοιχο βάρος.
  - Αθροίζουμε τις τιμές που προέκυψαν από τους διαφορετικούς δείκτες.

Έτσι ο δείκτης ομοιότητας μας μεταβάλλεται στον παρακάτω τύπο όπου $W_1$ – $W_n$ είναι τα βάρη για τα χαρακτηριστικά 1-n, $A_1$ – $A_n$ είναι τα σύνολα που παράχθηκαν από το MinHash των χαρακτηριστικών 1 – n και B είναι το σύνολο που παράχθηκε από το MinHash των όρων αναζήτησης.

$$J(A,B) = W_1 \times \frac{|\ A_1\ \cap\ B\ |}{|\ A_1\ \cup\ B\ |} + W_2 \times \frac{|\ A_2\ \cap\ B\ |}{|\ A_2\ \cup\ B\ |} + \ldots + W_n \times \frac{|\ A_n\ \cap\ B\ |}{|\ A_n\ \cup\ B\ |}$$

*Εικόνα 4. Σταθμισμένος δείκτης Jaccard*

Ο δείκτης αυτός παρέχει μεγαλύτερα περιθώρια παραμετροποίησης της αλγοριθμικής λύσης για εφαρμογές πέραν αυτής που παρουσιάζεται στην παρούσα εργασία.

# 3. Τεχνολογίες και Αρχιτεκτονική Συστήματος

## 3.1 Περίληψη Κεφαλαίου

Στο κεφάλαιο αυτό αναλύονται οι τεχνολογίες που χρησιμοποιήθηκαν για την ανάπτυξη του συστήματος καθώς και η αρχιτεκτονική αλλά και τα υποσυστήματα που το απαρτίζουν.

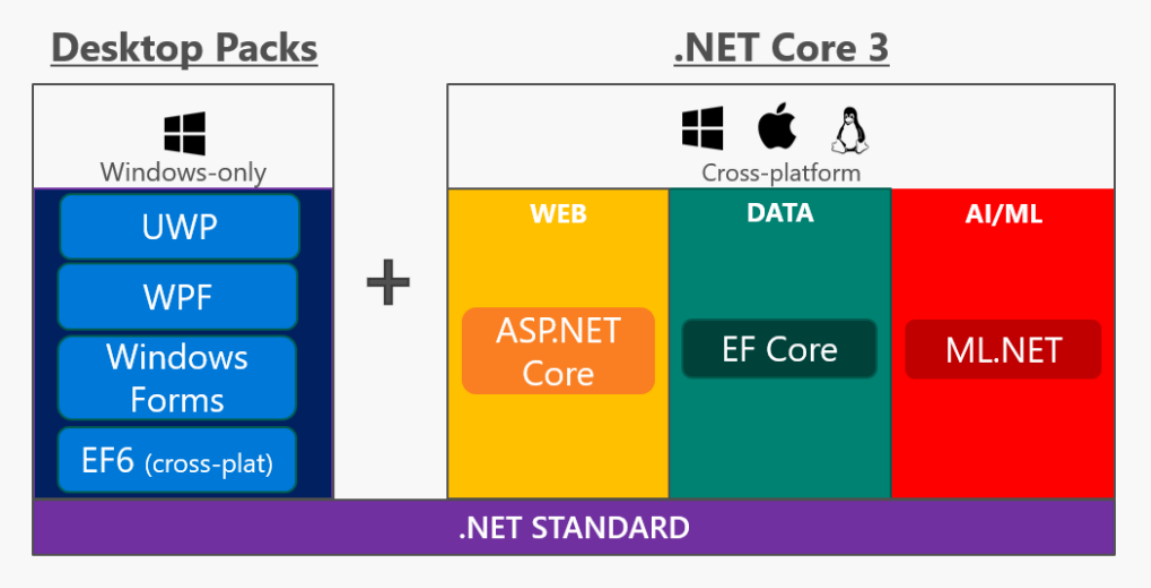
Θα γίνει αρχικά μια σύντομη παρουσίαση της πλατφόρμας .Net Core, το κύριο εργαλείο που χρησιμοποιήθηκε για την ανάπτυξη του συστήματος καθώς και οι τεχνολογίες ASP.NetCore, Entity Framework Core (EFCore) και Windows Presentation Foundation (WPF). Επίσης θα παρουσιαστεί η πλατφόρμα docker που χρησιμοποιήθηκε για την εκτέλεση και οργάνωση των υποσυστημάτων σε ένα ενιαίο σύστημα. Έπειτα θα παρουσιαστεί η γενικότερη αρχιτεκτονική του συστήματος και οι διασυνδέσεις μεταξύ των υποσυστημάτων.

## 3.2 .Net Core

Η πλατφόρμα .Net Core είναι ένα framework προγραμματισμού το οποίο δημιούργησε αρχικά η Microsoft αλλά δέχεται συνεισφορές σε επίπεδο κώδικα από όλη την κοινότητα ανοιχτού κώδικα. Οι κύριες γλώσσες προγραμματισμού τις οποίες μπορεί κάποιος να χρησιμοποιήσει με το .Net Core είναι C#, F# και Visual Basic. Για λήψη βιβλιοθηκών, είτε επίσημων είτε από άλλες πηγές, οι χρήστες μπορούν να χρησιμοποιήσουν το διαχειριστή πακέτων (package manager) NuGet. Οι παραγόμενες εφαρμογές μπορούν να τρέξουν στα τρία βασικά λειτουργικά συστήματα, Windows, Linux και MacOS.

Υποστηρίζει τέσσερεις βασικές κατηγορίες εφαρμογών:

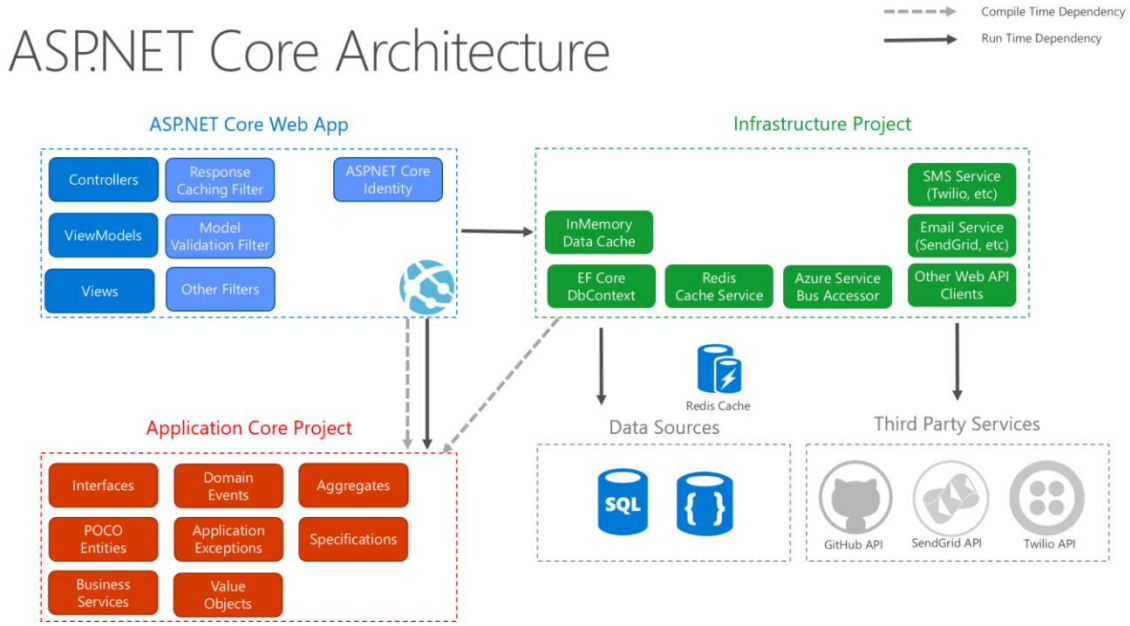
- Web εφαρμογές με χρήση του framework ASP .Net Core
- Εφαρμογές κονσόλας
- Εφαρμογές γραφικού περιβάλλοντος, με χρήση WPF ή UWP.
- Βιβλιοθήκες (.dll)




*Εικόνα 5. .Net Core*

### 3.3 ASP .Net Core

Το ASP .Net Core πλατφόρμα ανοιχτού κώδικα για σχεδιασμό και υλοποίηση ιστοσελίδων, υπηρεσιών ιστού και εφαρμογών διαδικτύου. Η πλατφόρμα είναι χτισμένη χρησιμοποιόντας το περιβάλλον εκτέλεσης του .Net Core και επιτρέπει στον χρήστη να γράψει κώδικα σε όλες τις γλώσσες που υποστηρίζει το .Net Core.

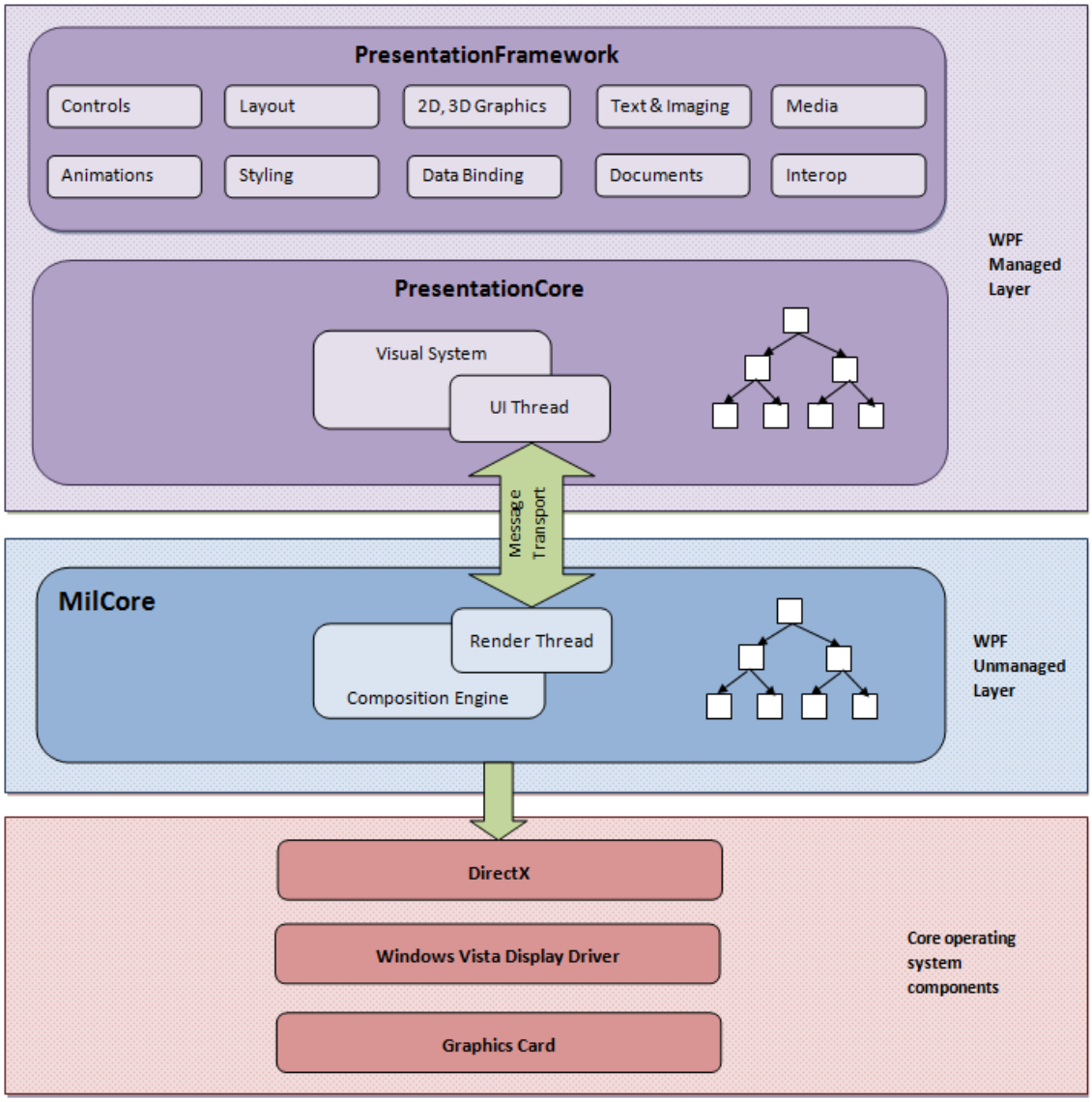



*Εικόνα 6. Γενική αρχιτεκτονική εφαρμογής ASP.Net Core.*

### 3.4 Windows Presentation Foundation

Το Windows Presentation Foundation είναι πλατφόρμα ανοιχτού κώδικα για σχεδιασμό και υλοποίηση εφαρμογών με γραφικό περιβάλλον διεπαφής χρήστη. Διαχωρίζει την λειτουργικότητα της εφαρμογής από τον σχεδιασμό της παρουσίασης των γραφικών της. Για το σχεδιασμό του περιβάλλοντος διεπαφής χρησιμοποιούνται αντικείμενα στην γλώσσα xaml, μια γλώσσα σήμανσης παρόμοια με την γνωστή xml. Για την υλοποίηση των λειτουργιών της εφαρμογής χρησιμοποιούνται οι συμβατές με το .Net Core γλώσσες, όπως C# F# και Visual Basic.

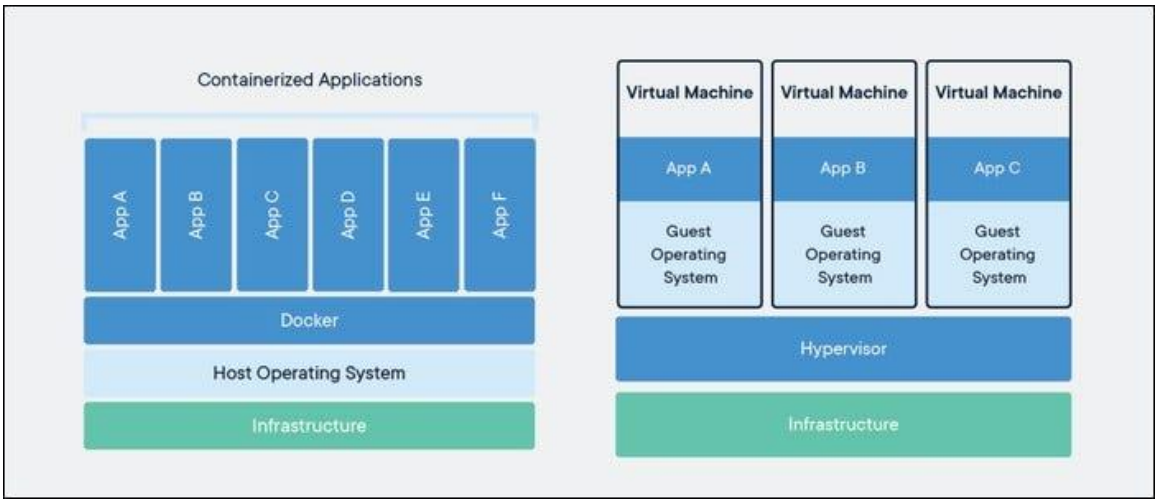
Στο διάγραμμα αρχιτεκτονικής που ακολουθεί ο χρήστης γράφει κώδικα μόνο για το ανώτερο επίπεδο χρησιμοποιώντας το PresentationFramework και τα εργαλεία του WPF αναλαμβάνουν τα υπόλοιπα βήματα για σχεδιασμό της διεπαφής στην οθόνη του χρήστη




*Εικόνα 7. Αρχιτεκτονική WPF*

## 3.5 Docker

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




*Εικόνα 8. Διαφορές docker containers με εικονικές μηχανές.*

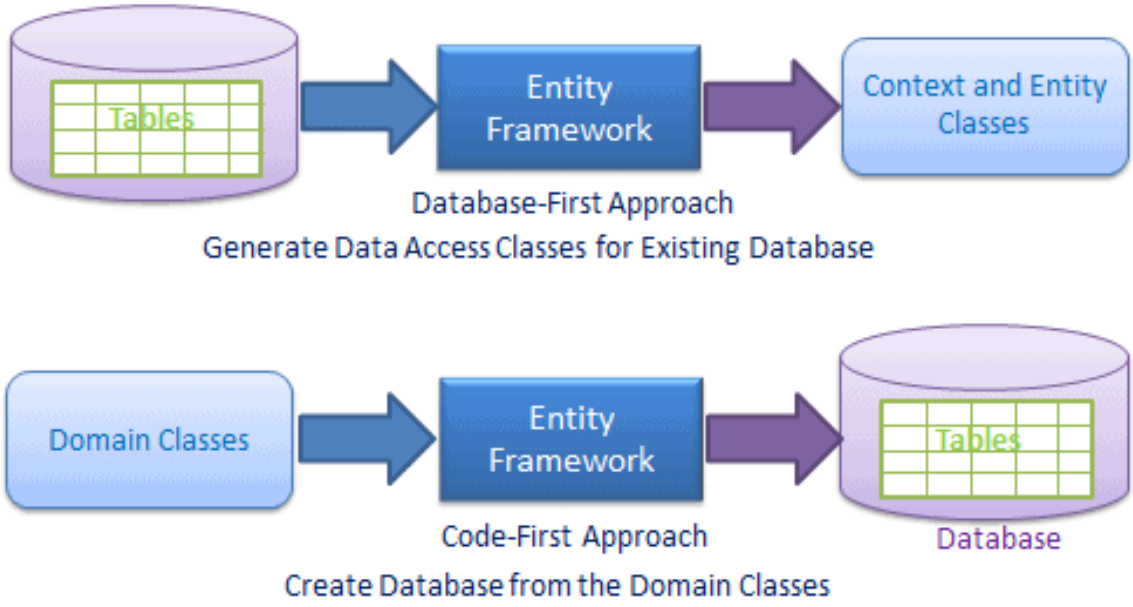
Το εργαλείο docker-compose που επίσης χρησιμοποιήθηκε στην παρούσα εργασία λειτουργεί σαν ενορχηστρωτής πολλών docker container. Χρησιμοποιώντας ένα αρχείο ρυθμίσεων εγκαθιστά και διασυνδέει μεταξύ τους τα container σε εσωτερικό υποδίκτυο. Αυτό μας επιτρέπει να περιγράψουμε και να εκτελέσουμε εύκολα το σύνολο της εφαρμογής. Επίσης μας παρέχει με εύκολο τρόπο την διαγραμματική απεικόνιση των container που χρησιμοποιήθηκαν με χρήση του προγράμματος compose_plantuml. Το διάγραμμα αυτό αποτελεί αποτύπωση της αρχιτεκτονικής της εφαρμογής και παρατίθεται στο κεφάλαιο 3.7.

## 3.6 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 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.




*Εικόνα 9. Entity Framework Core*

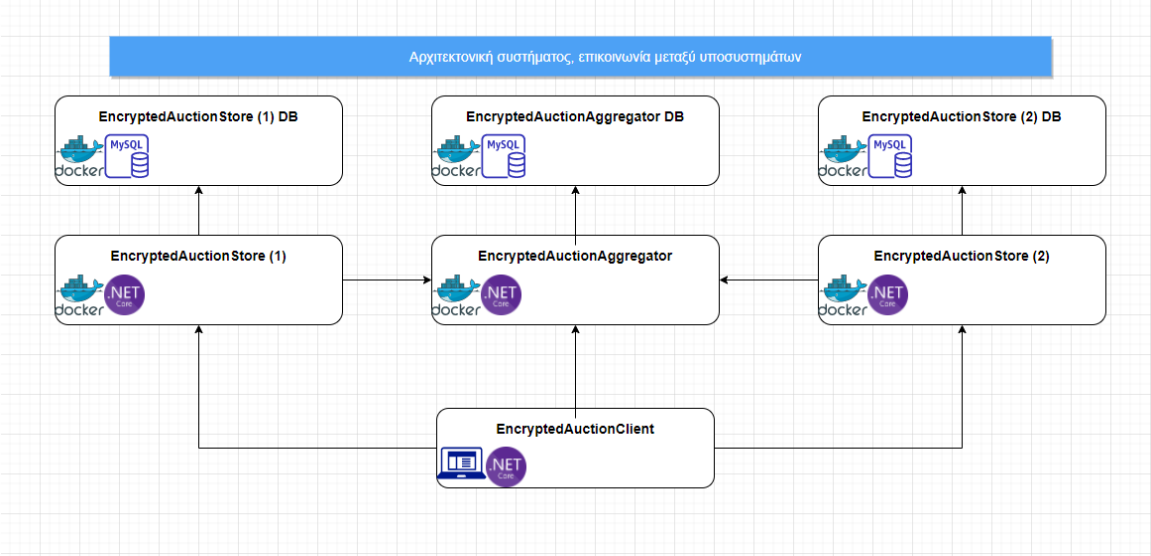
## 3.7 Αρχιτεκτονική συστήματος

Κεντρικός στόχος της εργασίας ήταν περισσότερο η έρευνα γύρω από την αρχιτεκτονική και την συνδεσμολογία μεταξύ των υποσυστημάτων παρά η αρτιότητα των επί μέρους υποσυστημάτων. Αυτό συνεπάγεται ότι τα υποσυστήματα είναι απλοποιημένες και περιορισμένες λειτουργικά αναπαραστάσεις του αντίστοιχου υποσυστήματος σε περίπτωση που μια τέτοια αρχιτεκτονική παρουσιαζόταν σε επίπεδο παραγωγής.

Το σύστημα που παρουσιάζεται στην παρούσα εργασία αποτελείται από 3 ξεχωριστά υποσυστήματα. Το κατάστημα (EncryptedAuctionStore) προσομοιάζει ένα κατάστημα ηλεκτρονικού εμπορίου το οποίο διαθέτει προς πώληση κάποια προϊόντα. Η κεντρική υπηρεσία αναζήτησης (EncryptedAuctionAggregator) προσομοιάζει έναν κεντρικό κόμβο στον οποία πολλά μαγαζιά μπορούν να καταχωρήσουν τα προϊόντα τους (υπό μορφή weighted hash). Είναι το κεντρικό σημείο του συστήματος και το σημείο στο οποίο γίνεται η μεγαλύτερη επεξεργασία των δεδομένων. Το πρόγραμμα πελάτη (EncryptedAuctionClient) προσομοιάζει το περιβάλλον διεπαφής του χρήση – αγοραστή με το σύστημα, μέσα από το οποίο μπορεί να κάνει αναζητήσεις και να πάρει προσφορές από τα καταστήματα για τα προϊόντα πού τον ενδιαφέρουν.

Κάθε κατάστημα καθώς και το κεντρικό σημείο αναζήτησης αποθηκεύει τα δεδομένα του σε αυτόνομη MySQL βάση η οποία εκτελείται σαν ξεχωριστό docker container.

Μια διαγραμματική απεικόνιση του συστήματός και τής επικοινωνίας μεταξύ των τμημάτων του εμφανίζεται στο παρακάτω διάγραμμα.




*Εικόνα 10. Αρχιτεκτονικό διάγραμμα υποσυστημάτων της εφαρμογής.*

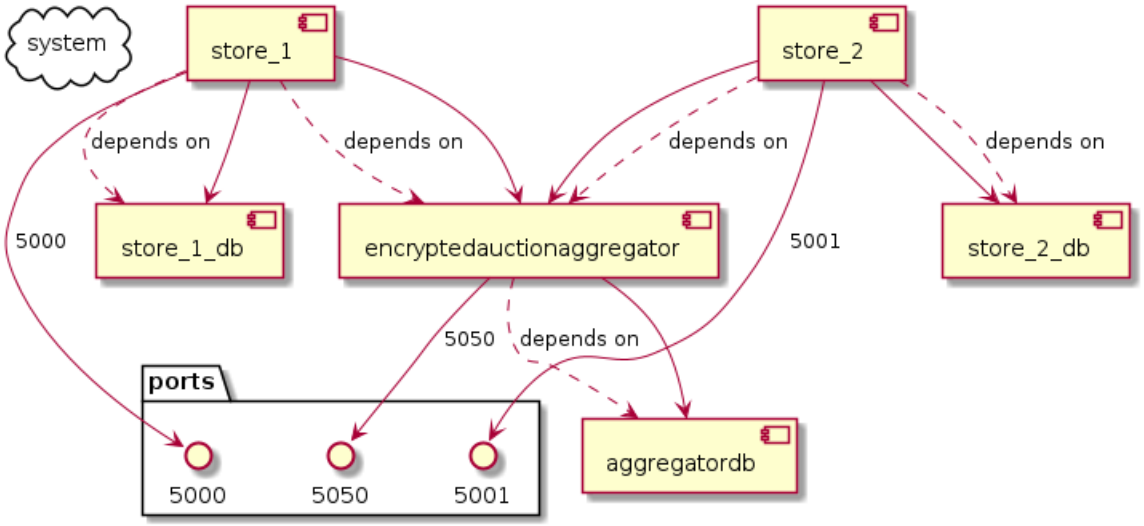
Η αρχιτεκτονική των υπηρεσιών απεικονίζεται πιο αναλυτικά στο παρακάτω διάγραμμα που έχει εξαχθεί από το αρχείο παραμετροποίησης docker-compose.yml. Παρατηρουμε στο διάγραμμα αυτό έξι συνολικά docker container:

- Έναν κεντρικό κόμβο (encryptedauctionaggregator)
- Δύο καταστήματα (store_1, store_2)
- Τρείς βάσεις δεδομένων (store_1_db, store_2_db, store_3_db

Από το διάγραμμα γίνεται επίσης εμφανές ότι οι βάσεις κάθε υποσυστήματος είναι αποκλειστικά συνδεδεμένες με το υποσύστημα χωρίς να υπάρχει τρόπος να αποκτήσει πρόσβαση σε αυτές κάποιο άλλο υποσύστημα.

Παρατηρούμε επίσης τόσο τις εσωτερικές συνδέσεις μεταξύ των καταστημάτων και του κεντρικού κόμβου, όσο και τις δικτυακές πόρτες (ports) που το κάθε υποσύστημα κρατάει ανοικτές για επικοινωνία με την εφαρμογή EncryptedAuctionClient.

Ιδιαίτερη, τέλος, σημασία παρουσιάζουν οι σχέσεις εξάρτησης μεταξύ των εφαρμογών που απεικονίζονται με τα διακεκομμένα βέλη. Οι σχέσεις αυτές καθορίζουν στο εργαλείο docker-compose την σειρά με την οποία θα εκτελέσει τα container του συστήματός μας. Έτσι βλέπουμε ότι κάθε εφαρμογή εξαρτάται από τη βάση της. Άρα οι 3 βάσεις δεδομένων θα είναι τα πρώτα container που θα λειτουργήσουν. Επίσης βλέπουμε ότι τα καταστήματα εξαρτώνται από τον κεντρικό κόμβο. Έτσι ξέρουμε ότι τα καταστήματα θα αρχικοποιηθούν εφόσον έχει αρχικοποιηθεί ο κεντρικός κόμβος.




*Εικόνα 11. Αρχιτεκτονική docker container που χρησιμοποιήθηκαν*

## 3.8 Αρχιτεκτονική βάσεων δεδομένων

Στο σύστημα χρησιμοποιήθηκαν δύο σχήματα βάσης δεδομένων:

Η κεντρική βάση δεδομένων του EncryptedAuctionAggregator εκτελείται μία και μόνη φορά και περιέχει:

- Το κλειδί του αλγορίθμου LSH που μοιράζεται όλο το σύστημα για τη λειτουργία του
- Πληροφορίες για τα καταστήματα
- Κατακερματισμένες μορφές των προϊόντων κάθε καταστήματος

Η βάση των καταστημάτων εκτελείται μία φορά για κάθε κατάστημα που συμμετέχει στο σύστημα και περιέχει αναλυτικές πληροφορίες για τα προϊόντα του καταστήματος.

# 4. Υλοποίηση Συστήματος

## 4.1 Περίληψη κεφαλαίου

Το κεφάλαιο ξεκινάει με την υλοποίηση του βασικού αλγορίθμου που χρησιμοποιήθηκε και μελετάμε, του Locality – Sensitive Hashing σε προγραμματιστικό περιβάλλον .Net Core. Έπειτα θα παρουσιαστούν οι υλοποιήσεις και οι λειτουργίες των τριών υποσυστημάτων που προαναφέρθηκαν. Τέλος θα παρουσιαστεί ο τρόπος με τον οποίο εγκαθίσταται και λειτουργεί το σύστημα με χρήση του εργαλείου docker compose.

## 4.2 Υλοποίηση LSH (LSHDotNet)

Ο αλγόριθμος LSH υλοποιήθηκε σε C# με χρήση του περιβάλλοντος .Net Core 3.1 σε μορφή βιβλιοθήκης κώδικα (.dll). Η βιβλιοθήκη είναι διαθέσιμη τόσο σε επίπεδο κώδικα στο αποθετήριο Github όσο και σαν πακέτο στο NuGet για άμεση χρήση σε άλλες εφαρμογές. Παρόλο που υλοποιήθηκε επί της ουσίας για να καλύψει το μικρό εύρος λειτουργιών του συστήματος, ο στόχος είναι να παρέχει όσο πιο γενικευμένες διεπαφές για άλλες πιθανές μετέπειτα χρήσεις.

Οι δύο βασικές κλάσεις που παρέχονται από την βιβλιοθήκη παρουσιάζονται παρακάτω.

### 4.2.1 MinHasher

Η κύρια λειτουργία της κλάσης MinHasher είναι να δημιουργεί MinHash για τα δεδομένα εισόδου.

Για τη δημιουργία των συναρτήσεων κατακερματισμού που αναφέρθηκαν στο κεφάλαιο 1.3 χρησιμοποιεί εσωτερικά έναν πίνακα από ζεύγη ακεραίων. Το μήκος του πίνακα (k) είναι που καθορίζει το τελικό μήκος του δημιουργούμενου MinHash. Οι τιμές αυτές αναφέρονται στον κώδικα ως _minHashSeeds και _signatureSize αντίστοιχα.

```
private int _signatureSize;
private int[][] _minHashSeeds;
```

Η βιβλιοθήκη μας δίνει δύο τρόπους να δημιουργήσουμε ένα νέο αντικείμενο MinHasher. Είτε ορίζοντας τον πίνακα minHashSeeds είτε ορίζοντας το μήκος του hash που θέλουμε να παράγεται. Στην δεύτερη περίπτωση μέσω της συνάρτησης CreateMinHashSeeds δημιουργείται ένας νέος πίνακας με τυχαία ζεύγη ακεραίων. Η συνάρτηση CreateMinHashSeeds

```
public MinHasher (int SignatureSize)
{
    _signatureSize = SignatureSize;
    _minHashSeeds = CreateMinHashSeeds(_signatureSize);
}
public MinHasher(int[][] minHashes)
{
    _signatureSize = minHashes.Length;
    _minHashSeeds = minHashes;
}
```

Το σύνολο των συναρτήσεων που είναι διαθέσιμες για τον χρήστη παρουσιάζονται στο παρακάτω *Interface*.

```
namespace LSHDotNet
{
    public interface IMinHasher<IdType>
    {
        int[] GetMinHashSignature(string value);
        int[] GetMinHashSignature(string[] value);
        Dictionary<IdType, int[]> CreateMinhashCollection(Dictionary
<IdType, string> documents);
        Dictionary<IdType, int[]> CreateMinhashCollection(Dictionary
<IdType, ILSHashable> documents);
        Dictionary<IdType, List<int[]>> CreateMinhashCollection(Dict
ionary<IdType, ILSHWeightedHashable> documents);
    }
}
```

Ο γενικός τύπος IdType δίνεται από τον χρήστη και αντιπροσωπεύει την μοναδική τιμή που θα χρησιμοποιείται για αναγνώριση των δεδομένων όταν επιστρέφονται αποτελέσματα αναζητήσεων. Στην δική μας υλοποίηση χρησιμοποιήθηκε ο τύπος Guid ( globally unique identifier ) αλλά η βιβλιοθήκη δέχεται κάθε τύπο που υλοποιεί η C#.

Οι βασική συνάρτηση κατακερματισμού GetMinHashSignature δέχεται ως όρισμα μια συμβολοσειρά και επιστρέφει το MinHash της ως πίνακα ακεραίων αφού πρώτα μετατρέψει τη συμβολοσειρά σε σύμβολα.

```
public int[] GetMinHashSignature(string[] tokens)
{
    int[] minHashValues = Enumerable.Repeat(int.MaxValue, _signature
Size).ToArray();

    HashSet<string> skipDups = new HashSet<string>();
    foreach (var token in tokens)
    {
        if (skipDups.Add(token))
        {
            for (int i = 0; i < _signatureSize; i++)
            {
                int[] seed = _minHashSeeds[i];
                int currentHashValue=LSHHash(token,seed[0],seed[1]);
                if (currentHashValue < minHashValues[i])
                    minHashValues[i] = currentHashValue;
            }
        }
    }
    return minHashValues;
}
```

Η συνάρτηση GetMinHashSignature εκτελεί τα εξής βήματα

- Δημιουργεί έναν πίνακα (minHashValues) μήκους _signatureSize και τον γεμίζει με τη μέγιστη τιμή ακεραίου
- Για κάθε σύμβολο της συμβολοσειράς εισόδου
  - Για κάθε θέση i του πίνακα _minHashSeeds
    - Εκτελεί την συνάρτηση κατακερματισμού LSHHash με το σύμβολο και την δυάδα τιμών του πίνακα _minHashSeeds σαν όρισμα
    - Αν η τιμη που παράγεται είναι μικρότερη από την τιμή του πίνακα minHashValues στη θέση i αντικαθιστά την τιμή του πίνακα με το αποτέλεσμα της συνάρτησης LSHHash

Η συνάρτηση LSHHash δέχεται σαν όρισμα συμβολοσειρά και δύο ακεραίους και επιστρέφει έναν ακέραιο.

```
private static int LSHHash(string inputData,int seedOne,int seedTwo)
{
    unchecked
    {
        int hash = (int)2166136261;
        hash = hash * 16777619 ^ seedOne;
        hash = hash * 16777619 ^ seedTwo;
        hash = hash * 16777619 ^ GetSimpleHash(inputData);
        return hash;
    }
}
```

Η μέθοδος κατακερματισμού που χρησιμοποιήθηκε είναι γνωστή ως FNV (Fowler/Noll/Vo) hash ή οποία χρησιμοποιεί μετατόπιση bit αποκλειστικού «ή» (xor bit shifting) για να δημιουργήσει κατακερματισμένες τιμές με χαμηλή πιθανότητα σύγκρουσης. Η μέθοδος αυτή δουλεύει επαρκώς για τις ανάγκες της εργασίας και για απόδειξη της καταλληλότητας του αλγορίθμου για το σύστημα στο οποίο χρησιμοποιήθηκε. Η ακρίβεια των αποτελεσμάτων όμως θα αυξανόταν με τη χρήση μιας συνάρτησης που θα δημιουργούσε περισσότερες συγκρούσεις.

Οι συναρτήσεις CreateMinhashCollection είναι απλές βοηθητικές συναρτήσεις που δέχονται συλλογές από δεδομένα και επιστρέφουν συλλογές από τις κατακερματισμένες μορφές των δεδομένων.

### 4.2.2 LSHSearch

Η κλάση LSHSearch υλοποιεί την αναζήτηση στα δεδομένα με χρήση του LSH.

Για τη σύγκριση μεταξύ δύο δεδομένων χρησιμοποιείται ένα μέτρο ομοιότητας που δίνεται από τον χρήστη της βιβλιοθήκης κατά τη δημιουργία ενός νέου αντικειμένου LSHSearch.

```
public LSHSearch(MinHasher<IdType> minHasher, ISimilarity similarity
Measure)
{
    this.minHasher = minHasher;
    this.similarityMeasure = similarityMeasure;
}
```

Όπως φαίνεται και από τον κώδικα ό τύπος LSHSearch είναι και αυτός γενικευμένος ως προς τον τύπο με τον οποίον θα ταυτοποιεί τις εγγραφές (IdType).

Τα μέτρα ομοιότητας περιγράφονται από τον τύπο ISimilarity, είναι συναρτήσεις που δέχονται ως ορίσματα πίνακες ακεραίων (hash) και επιστρέφουν έναν δεκαδικό ( από 0 εώς 1 ) που αντιπροσωπεύει την ομοιότητα των δύο αντικειμένων.

```
public delegate double ISimilarity(int[] setA, int[] setB);
```

Στο σύστημα της παρούσας εργασίας χρησιμοποιήθηκε ο δείκτης Jaccard σαν μέτρο ομοιότητας ο οποίος υλοποιείται από την παρακάτω συνάρτηση.

```
public static double Jaccard(int[] setA, int[] setB)
{
    int intersection = setA.Intersect(setB).Count();
    return intersection / (double)setA.Length;
}
```

Η αναζήτηση γίνεται με την χρήση κάποιας από τις παραλλαγές της συνάρτησης GetClosest όπως περιγράφονται στο παρακάτω *Interface*.

```
namespace LSHDotNet
{
    public interface ILSHSearch<IdType>
    {
        List<IdType> GetClosest<IdType>(Dictionary<IdType, int[]> se
archSpaceHashes, int[] searchMinHash, int count);

        List<Result<IdType>> GetClosest(Dictionary<IdType, string> s
earchSpace, string searchString, int count, double threshold = 0.5);

        List<Result<IdType>> GetClosest(Dictionary<IdType, ILSHashab
le> searchSpace, string searchString, int count, double threshold =
0.5);

        List<Result<IdType>> GetClosest(Dictionary<IdType, ILSHWeigh
tedHashable> searchSpace, string searchString, int count, double thr
eshold = 0.5);

        List<Result<IdType>> GetClosest(Dictionary<IdType, IWeighted
Hashed> hashedSearchSpace, int[] hashedSearchString, int count, doub
le threshold = 0.5);

    }
}
```

Όπως φαίνεται από τον κώδικα ο χρήστης της βιβλιοθήκης ορίζει τόσο τον μέγιστο αριθμό των αποτελεσμάτων που θέλει να λάβει, όσο και την ελάχιστη ομοιότητα που θεωρεί αποδεκτή για την αναζήτησή του με χρήση των παραμέτρων count και threshold αντίστοιχα. Οι διαφορετικές παραλλαγές της συνάρτησης GetClosest επιτρέπουν στον χρήστη της βιβλιοθήκης να πραγματοποιήσει αναζητήσεις είτε μεταξύ συμβολοσειρών, με εσωτερική μετατροπή τους σε hashes, είτε μεταξύ hashes είτε με χρήση ενός από τα *Interface* που παρέχονται από την βιβλιοθήκη για ευκολότερη ενσωμάτωση της βιβλιοθήκης σε μελλοντικές εφαρμογές. Για τον ίδιο σκοπό, ο τύπος εξόδου Result χρησιμοποιεί εσωτερικά το *Interface* BaseHashable.

```
namespace LSHDotNet
{
    public interface BaseHashable { }
    public interface ILSHashable : BaseHashable
    {
        public string GetStringToHash();
    }

    public interface ILSHWeightedHashable : BaseHashable
    {
        public List<Tuple<string,double>> GetWeightedStringsToHash()
;
    }

    public interface IWeightedHashed : BaseHashable
    {
        public List<Tuple<int[], double>> GetWeightedHashes();
    }
}
```

Τα *Interfaces* εισόδου στην αναζήτηση περιγράφονται στον παραπάνω κώδικα απαιτούν την υλοποίηση μόνο μίας συνάρτησης. Δίνουν έτσι στον χρήστη την ευελιξία να ορίσει ο ίδιος είτε:

- Ποια συμβολοσειρά από κάθε δεδομένο του θα χρησιμοποιηθεί είτε για την αναζήτηση του hash, με τη χρήση του ILSHHashable
- Ποιες συμβολοσειρές και με τί συντελεστή βαρύτητας η κάθε μία θα χρησιμοποιηθούν για την αναζήτηση με την χρήση του ILSHWeightedHashable
- Ποια hashes, υπολογισμένα εκ των προτέρων και με τι συντελεστή βαρύτητας το κάθε ένα θα χρησιμοποιηθούν για την αναζήτηση με την χρήση του ILSHWeightedHashed

Ο τύπος των αποτελεσμάτων της αναζήτησης περιέχει εκτός από τον κωδικό ταυτοποίησης (Id) και την ομοιότητα του αποτελέσματος με τους όρους της αναζήτησης και την αναπαράσταση του δεδομένου στην μορφή που δόθηκε με την χρήση του κενού *Interface* BaseHashable που «υλοποιείται» από όλα τα προαναφερθέντα *Interfaces.*

```
namespace LSHDotNet
{
    public class Result<T> : IComparable<Result<T>>
    {
        public T Id { get; set; }
        public double Similarity { get; set; }
        public BaseHashable Document { get; set; }
        public int CompareTo(Result<T> that)
        {
            if (this.Similarity > that.Similarity) return -1;
            if (this.Similarity == that.Similarity) return 0;
            return 1;
        }
    }
}
```

### 4.2.3 Παράδειγμα χρήσης της βιβλιοθήκης

Στο παρακάτω τμήμα κώδικα παρουσιάζουμε μια απλή περίπτωση χρήσης της βιβλιοθήκης για αναζήτηση σε δεδομένα τύπου Product που όπως φαίνεται και από τον κώδικα υλοποιούνε το *Interface* ILSHWeightedHashable, δηλαδή η αναζήτηση σε αυτά θα είναι σταθμισμένη.

```
public class Product : ILSHWeightedHashable
{
    private string Brand;
    private string Model;
    public Product(string brand, string model)
    {
        this.Brand = brand;
        this.Model = model;
    }
    public List<Tuple<string, double>> GetWeightedStringsToHash()
    {
        return new List<Tuple<string, double>>()
        {
            new Tuple<string, double>(Brand, 0.2),
            new Tuple<string, double>(Model, 0.8),
        };
    }
}
```

Στην κλάση αυτή φαινεται η ευελιξία που δίνει η βιβλιοθήκη στον χρήστη της για πειραματισμό και βελτιστοποίηση της αναζήτησης στα δεδομένα του με βάση το ποια χαρακτηριστικά των δεδομένων κρίνονται σημαντικότερα. Έτσι στο συγκεκριμένο παράδειγμα βλέπουμε ότι το

μοντέλο του προιόντος συμμετέχει στην αναζήτηση με συντελλεστή βαρύτητας τεσσερις φορές μεγαλύτερο από την μάρκα.

Ακολουθεί ο κώδικας της αναζήτησης με σχόλια επεξήγησης για κάθε εντολή:

```
public void DemoSearch()
{
    //Δημιουργούμε ένα αντικείμενο τύπου MinHasher με μήκος signatur
e τους 50 χαρακτήρες και τύπο ταυτοποίησης Guid
    var minHasher = new MinHasher<Guid>(50);
    //Δημιουργούμε ένα αντικείμενο τύπου LSHSearch με μέτρο ομοιότητ
ας τον συντελεστή Jaccard
    var lshSearcher = new LSHSearch<Guid>(minHasher, SimilarityMeasu
res.Jaccard);
    //Δημιουργούμε και γεμίζουμε την λίστα τών προιόντων
    var productList = new Dictionary<Guid, ILSHWeightedHashable>();
    productList.Add(Guid.NewGuid(), new Product("Samsung", "C24F390F
HU"));
    //Ορίζουμε τον όρο της αναζήτησης
    var searchString = "Samsung C24F390FU";
    //Πραγματοποιούμε την αναζήτηση και παίρνουμε τα πιθανά αποτελέσ
ματα.
    //Έχουμε θέσει το μέγιστο αριθμό αποτελεσμάτων να είναι 5 και τη
ν ελάχιστη επιθυμητή ομοιότητα στο 50%.
    var results = lshSearcher.GetClosest(productList, searchString,
5, 0.5);
}
```

### 4.3 EncryptedAuctionDatatypes

Για να υπάρχει ευκολότερη ανταλλαγή πληροφορίας μεταξύ των υποσυστημάτων δημιουργήθηκε μια βιβλιοθήκη κώδικα σε c# .Net Core η οποία περιέχει μόνο δηλώσεις κλάσεων δεδομένων. Έτσι όλα τα υποσυστήματα ανταλλάσσουν μεταξύ τους πληροφορία κάνοντας χρήση των κοινών κλάσεων και αν χρειαστεί περισσότερη πληροφορία ή κάποια λειτουργικότητα για εσωτερική χρήση σε κάποια από τις εφαρμογές, τότε η εφαρμογή αυτή επεκτείνει κάποια από τις κλάσεις κληρονομώντας από αυτήν για τη δημιουργία μιας υπερ-κλάσης.

```
namespace EncryptedAuctionDatatypes
{
    public class StoreBase
    {
        public virtual Guid Id { get; set; }
        public string Name { get; set; }
        public string ApiUrl { get; set; }
    }
}
```

```
namespace EncryptedAuctionAggregator.Database.DBModels
{
    public class Store : StoreBase
    {
        [Key]
        public override Guid Id { get; set; }
        public List<HashedProduct> HashedProducts { get; set; }
    }
}
```

Έτσι έχουμε για παράδειγμα την κλάση Store που βρίσκεται στον κώδικα του EncryptedAuctionAggregator που κληρονομεί από την βασική κλάση StoreBase που βρίσκεται στην κοινή βιβλιοθήκη EncryptedAuctionDatatypes. Η συγκεκριμένη επέκταση έγινε ώστε η κλάση Store να χρησιμοποιήθει για συσχέτιση καταστημάτων με hashed προϊόντα στην βάση δεδομένων του EncryptedAuctionAggregator.

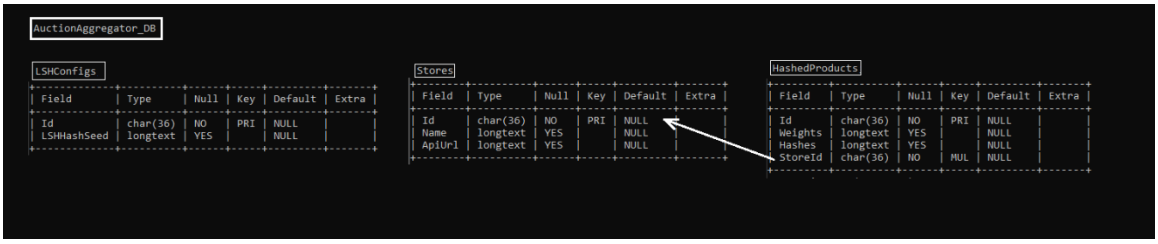
## 4.4 EncryptedAuctionAggregator

Πρόκειται για μία εφαρμογή διαδικτύου που αποτελεί το κεντρικό υποσύστημα της εφαρμογής. Υλοποίημένο με χρήση των προαναφερθέντων τεχνολογίων .Net Core, ASP.Net Core και Entity Framework Core.

Επικοινωνεί με τα υπόλοιπα τμήματα του συστήματος με κλήσεις HTTP μέσω των Χειριστών (Controllers) του. Controller ονομάζουμε μια κλάση που διαχειρίζεται ένα σύνολο κλήσεων HTTP σχετικών, συνήθως, με ένα μοντέλο ή μια λειτουργία του συστήματος.

### 4.4.1 Μοντέλα & βάση δεδομένων

Για την αποθήκευση των δεδομένων της, η εφαρμογή χρησιμοποιεί μια σχεσιακή βάση MySQL η οποία εκτελείται σε ξεχωριστό docker container. Οι πίνακες που υποστηρίζουν την αποθήκευση των δεδομένων καθώς και οι σχέσεις μεταξύ τους εμφανίζονται στο παρακάτω διάγραμμα.




*Εικόνα 12. Σχήμα βάσης δεδομένων EncryptedAuctionAggregator*

- Ο πίνακας LSHConfigs περιέχει στη φυσική λειτουργία της εφαρμογής μόνο μία εγγραφή δεδομένων η οποία δημιουργείται στην πρώτη εκτέλεση της εφαρμογής. Το πεδίο LSHHashSeed είναι το «κλειδί» το οποίο όλοι οι χρήστες της εφαρμογής, είτε πρόκειται για καταστήματα είτε για πελάτες χρησιμοποιούν για κατακερματισμό LSH των δεδομένων τους.
- Ο πίνακας Stores αποθηκεύει τα καταστήματα (EncryptedAuctionStores) τα οποία έχουν καταχωρήσει προϊόντα τους στη βάση του EncryptedAuctionAggregator. Το πεδίο Name περιέχει το όνομα του καταστήματος ενώ το πεδίο ApiUrl περιέχει την ηλεκτρονική διεύθυνση του καταστήματος ώστε η εφαρμογή του πελάτη (EncryptedAuctionClient) να κάνει κλήσεις για να πάρει την συνολική πληροφορία για τα προϊόντα που επέστρεψε η έρευνα.

- Ο πίνακας HashedProducts περιέχει την κατακερματισμένη μορφή των προϊόντων όπως αυτά καταχωρήθηκαν από τα αντίστοιχα καταστήματα. Το πεδίο StoreId είναι το μοναδικό κλειδί του καταστήματος ώστε να γίνεται εύκολα η συσχέτιση του αποτελέσματός με το κατάστημα ώστε να επιστραφεί στον χρήστη. Τα πεδία Weights και Hashes αποθηκεύουν πίνακες υπό μορφή συμβολοσειρών. Για παράδειγμα αν ένα προϊόν έχει καταχωρηθεί ως 3 διαφορετικά σταθμισμένα hashes η εγγραφή του στα παραπάνω πεδία θα είναι ως εξής:
  Weights: “0.2,0.8”
  Hashes: “8,3,12,42,11;99,42,69,420,11”
  Η μετατροπή των δεδομένων από συμβολοσειρές σε πίνακες μιας η δύο διαστάσεων γίνεται αυτόματα από το Entity Framework Core με χρήση συναρτήσεων μετατροπής που εμείς παρέχουμε.

```
public static class Converters
{
    public static ValueConverter IntArrayArrayConverter = new ValueC
onverter<int[][], string>(
    a => string.Join(";", a.Select(vv => string.Join(',', vv))),
    s => s.Split(";", StringSplitOptions.RemoveEmptyEntries).Select(
vv => vv.Split(",", StringSplitOptions.RemoveEmptyEntries)).Select(v
vv => vvv.Select(vvvv => int.Parse(vvvv)).ToArray()).ToArray());

    public static ValueConverter IntArrayConverter = new ValueConver
ter<int[], string>(
            a => string.Join(",", a),
            s => s.Split(",", StringSplitOptions.RemoveEmptyEntries)
.Select(vv => int.Parse(vv)).ToArray());

    public static ValueConverter DoubleArrayConverter = new ValueCon
verter<double[], string>(
    a => string.Join(",", a),
    s => s.Split(",", StringSplitOptions.RemoveEmptyEntries).Select(
vv => double.Parse(vv)).ToArray());
}
```

Οι στατικές συναρτήσεις τής κλάσης Converters χρησιμοποιούνται, όπως θα δούμε παρακάτω, από το Entity Framework Core για μετατροπή τιμών κατά την αποθήκευση ή ανάκτηση πληροφορίας από τη βάση δεδομένων.

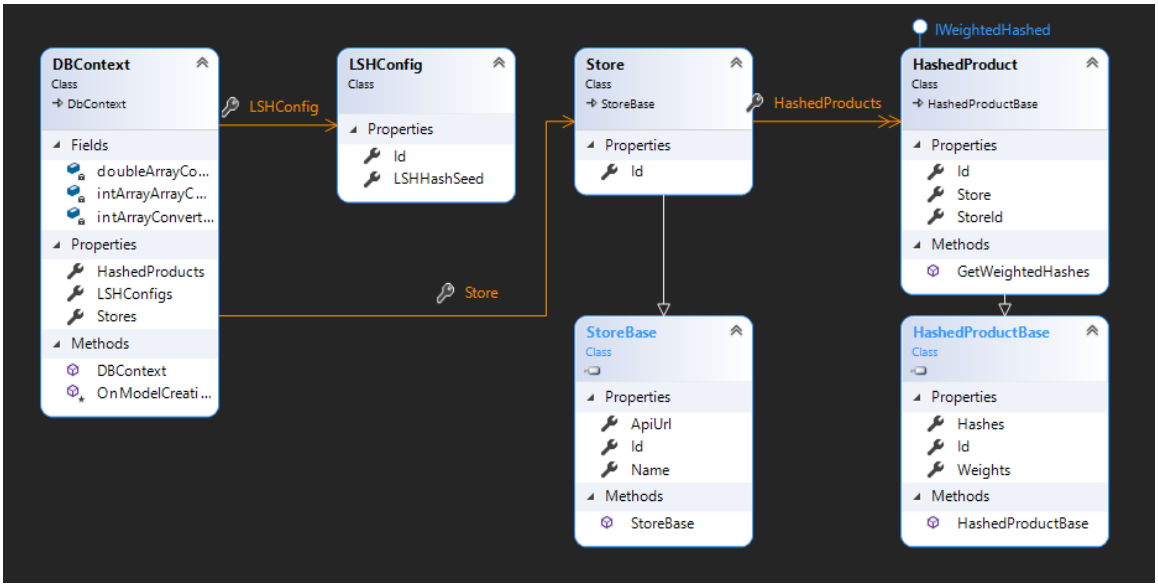
Η βάση αυτή παράχθηκε με χρήση των εργαλείων Code First του Entity Framework Core. Για τη δημιουργία της βάσης χρησιμοποιήθηκε η κλάση DBContext στην οποία περιγράφεται το σύνολο των οντοτήτων της εφαρμογής καθώς και οι συσχετίσεις μεταξύ τους.

```
public class DBContext : DbContext
{
    public DBContext(DbContextOptions<DBContext> dbContextOptions) :
 base(dbContextOptions) { }
    public DbSet<Store> Stores { get; set; }
```

```
    public DbSet<HashedProduct> HashedProducts { get; set; }
    public DbSet<LSHConfig> LSHConfigs { get; set; }
    public LSHConfig LSHConfig
    {
        get
        {
            return LSHConfigs.FirstOrDefault();
        }
    }
    protected override void OnModelCreating(ModelBuilder modelBuilde
r)
    {
        modelBuilder.Entity<Store>()
            .HasMany(s => s.HashedProducts)
            .WithOne(p => p.Store)
            .OnDelete(DeleteBehavior.Cascade);
        modelBuilder.Entity<LSHConfig>()
            .Property(l => l.LSHHashSeed)
            .HasConversion(Converters.IntArrayArrayConverter);
        modelBuilder.Entity<HashedProduct>()
            .Property(h => h.Hashes)
            .HasConversion(Converters.IntArrayArrayConverter);
        modelBuilder.Entity<HashedProduct>()
            .Property(h => h.Weights)
            .HasConversion(Converters.DoubleArrayConverter);
    }
}
```

Αναλυτικότερα βλέπουμε τα 3 DbSet που ορίζουν τους 3 πίνακες που είδαμε παραπάνω Stores, HashedProducts και LSHConfigs. Βλέπουμε επίσης την παράμετρο LSHConfig που εκφράζει την μοναδική εγγραφή στον πίνακα LSHConfigs για ευκολότερη χρήση από την εφαρμογή.

Στην συνάρτηση OnModelCreating βλέπουμε τις συσχετίσεις που θα κληθεί να δημιουργήσει το Entity Framework Core στη βάση δεδομένων. Οι συσχετίσεις εκφράζονται με αναλυτικό τρόπο, με κλήσεις δηλαδή σε συναρτήσεις HasMany().WithOne() που εκφράζουν μια σχέση ένα προς πολλά μεταξύ των καταστημάτων και των προϊόντων τους. Στην ίδια συνάρτηση επίσης ορίζονται οι συναρτήσεις που κάνουν την μετατροπή δεδομένων για τα πεδία που κωδικοποιούν περίπλοκές δομές δεδομένων σε απλές συμβολοσειρές. Αυτό γίνεται με την χρήση της συνάρτησης Property().HasConversion().




*Εικόνα 13. Διαγραμμα κλάσεων EncryptedAuctionAggregator*

Όπως φαίνεται από το παραπάνω διάγραμμα κλάσεων η διαχείριση όλων των μοντέλων – κλάσεων γίνεται μέσω της κεντρικής κλάσης DBContext η οποία αντιπροσωπεύει την σύνδεση της εφαρμογής με την βάση δεδομένων. Επίσης από το διάγραμμα φαίνεται η κληρονομικότητα των κλάσεων της εφαρμογής μας από τις γενικευμένες κλάσεις της κοινής βιβλιοθήκης που προαναφέραμε. Ακόμα βλέπουμε πώς η κλάση HashedProduct που αντιπροσωπεύει την κατακερματισμένη μορφή των προϊόντων υλοποιεί το Interface IWeightedHashed που χρησιμοποιείται εσωτερικά από την κλάση LSHSearcher για σταθμισμένη αναζήτηση LSH.

Για την αρχικοποίηση των δεδομένων στη βάση χρησιμοποιείται η στατική μέθοδος Seedit της κλάσης Seeder.

```
public static class Seeder
{
    public static void Seedit(string jsonData, DBContext context)
    {
        if (!context.LSHConfigs.Any())
        {
            context.LSHConfigs.Add(new LSHConfig()
            {
                Id = Guid.NewGuid(),
                LSHHashSeed = LSHDotNet.MinHasher<Guid>.CreateMinHas
hSeeds(100)
            });
        }
        List<Store> stores = JsonConvert.DeserializeObject<List<Stor
e>>(jsonData);
        if (!context.Stores.Any())
        {
            context.AddRange(stores);
            context.SaveChanges();
        }
```

```
    }
}
```

Η μέθοδος αυτή εκτελείται μόνο κατά την πρώτη εκτέλεση του προγράμματος και δημιουργεί:

- Ένα τυχαίο κλειδί που θα χρησιμοποιηθεί από το σύνολο του συστήματος για τη δημιουργία LSH hashes και αποθηκεύεται στον πίνακα LSHConfigs.
- Μία σειρά από εγγραφές καταστημάτων δεδομένα για τις οποίες διαβάζει από ένα αρχείο τύπου .json με όνομα seed.json ένα παράδειγμα του αρχείου παρουσιάζεται παρακάτω.

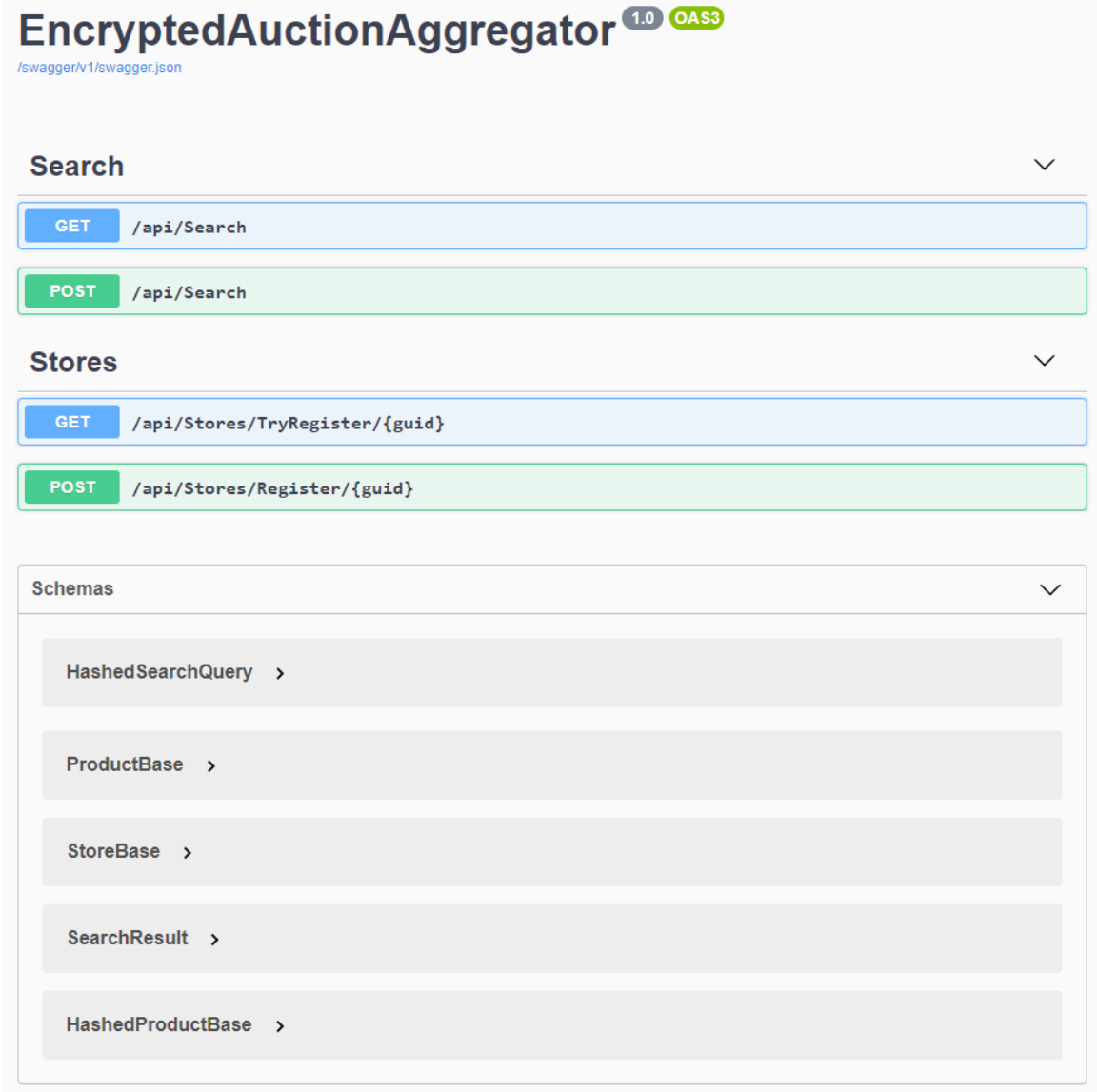
```
[
  {
    "Id": "c388b9a1-e6e3-4733-a070-feeca5618816",
    "Name": "FirstStore",
    "ApiUrl": "http://localhost:5000"
  },
  {
    "Id": "96fb9df1-3e8c-42e3-a3dd-7a0181555ec3",
    "Name": "SecondStore",
    "ApiUrl": "http://localhost:5001"
  }
]
```

Το παραπάνω αρχείο θα δημιουργήσει στην βάση δεδομένων δύο εγγραφές τύπου Store.

### 4.4.2 Λειτουργικότητα

Η υπηρεσία EncryptedAuctionAggregator παρέχει λειτουργίες αναζήτησης προϊόντων στους χρήστες της εφαρμογής «πελάτη» καθώς και για την καταχώρηση νέων καταστημάτων και την αποθήκευση των προϊόντων τους. Παρέχει αυτές τις λειτουργίες εκθέτοντας μια διαδικτυακή διεπαφή (Web API) με την χρήση του ASP.Net Core. Μια συνοπτική παρουσίαση των λειτουργιών όπως υλοποιούνται.

Ένας εύκολος τρόπος να απεικονίσουμε την διεπαφής δικτυακών εφαρμογών που χρησιμοποιούν .Net Core είναι η χρήση της βιβλιοθήκης swagger. Έπειτα από την 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).




*Εικόνα 14. Απεικόνιση διεπαφής EncryptedAuctionAggregator μέσω της βιβλιοθήκης swagger.*

Στην παραπάνω εικόνα σημειώνονται τόσο οι κλήσεις του συστήματος όσο και οι τύποι δεδομένων που ανταλλάσσονται μεταξύ της υπηρεσίας και των εφαρμογών που την χρησιμοποιούν κατά τη λειτουργία της.

Για την αναζήτηση προϊόντων από τους πελάτες χρησιμοποιείται η κλάση SearchController, η οποία διαχειρίζεται τις HTTP κλήσεις που δρομολογούνται στην διεύθυνση /api/Search.

Η κλήση GET επιστρέφει στην εφαρμογή που την καλεί το κλειδί του LSH ώστε να κωδικοποιήσει τους όρους αναζήτησής του χρήστη.

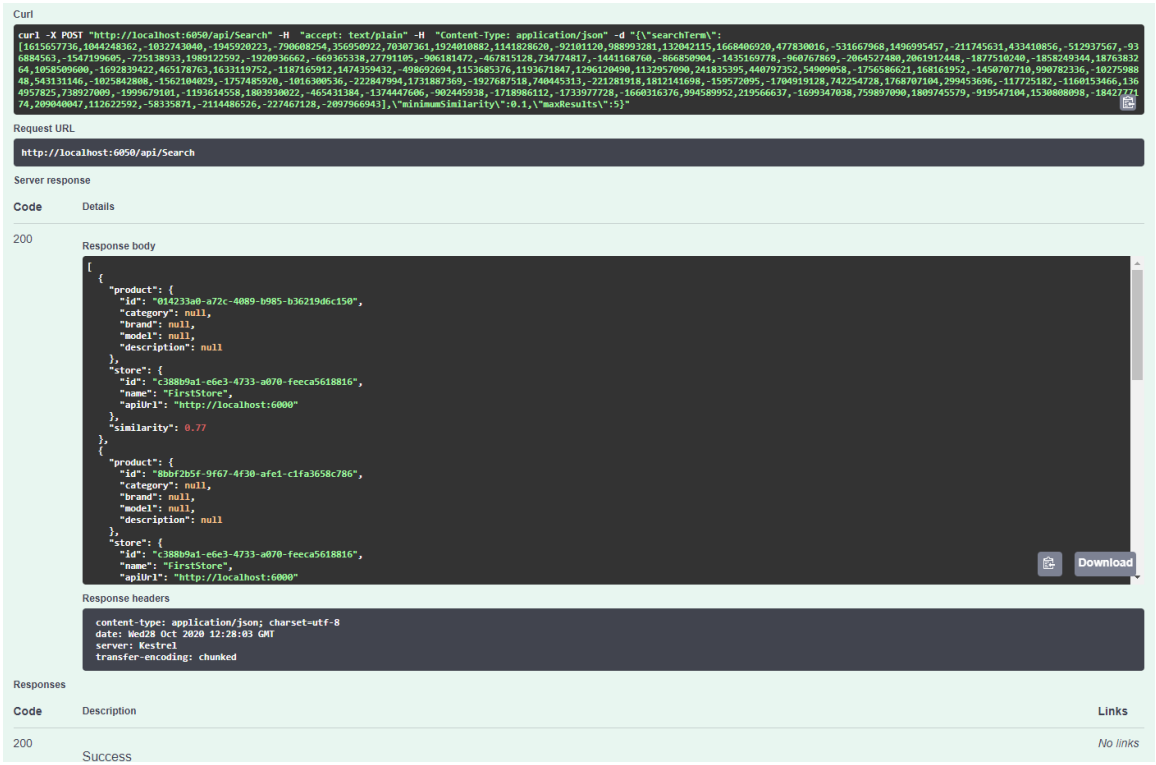
Στην κλήση POST η εφαρμογή δέχεται στο σώμα του HTTP μηνύματος ένα αντικείμενο τύπου HashedSearchQuery και επιστρέφει στο σώμα της απάντησης μια λίστα από αντικείμενα τύπου SearchResult.

```
[Route("api/[controller]")]
[ApiController]
public class SearchController : ControllerBase
{
    private ILogger<SearchController> _logger;
    private DBContext _db;
    public SearchController(ILogger<SearchController> logger, DBCont
ext dbContext)
    {
        _logger = logger;
        _db = dbContext;
    }
    [HttpGet]
    public int[][] GetHashSeed()
    {
        return _db.LSHConfig.LSHHashSeed;
    }
    [HttpPost]
    public List<SearchResult> Search(HashedSearchQuery search)
    {
        var minHasher = new MinHasher<Guid>(_db.LSHConfig.LSHHashSee
d);
        var lshSearcher = new LSHSearch<Guid>(minHasher, SimilarityM
easures.Jaccard);
        var searchSpace = _db.HashedProducts
            .ToDictionary(p => p.Id, p => (IWeightedHashed)p);
        var res = lshSearcher.GetClosest(searchSpace, search.SearchT
erm, search.MaxResults, search.MinimumSimilarity);
        var results = res.Select(r =>
            new SearchResult
            {
                Product = new ProductBase()
                {
                    Id = r.Id
                },
                Similarity = r.Similarity,
                Store = _db.HashedProducts
                    .Include(p => p.Store)
                    .FirstOrDefault(p => p.Id == r.Id).Store
            }
        ).ToList();
        return results;
    }
}
```

Οι δύο συναρτήσεις της κλάσης χρησιμοποιούνε το Entity Framework Core για να ανακτήσουν δεδομένα από την βάση δεδομένων.

Η συνάρτηση GetHashSeed διαχειρίζεται την κλήση GET: /api/Search. Ανασύρει την μοναδική εγγραφή του πίνακα LSHConfigs και επιστρέφει στον χρήστη το κλειδί για την κωδικοποίηση της αναζήτησης.

Η συνάρτηση Search διαχειρίζεται την κλήση POST: /api/Search. Ανασύρει το σύνολο των προϊόντων από τη βάση και πραγματοποιεί αναζήτηση σε αυτά συγκρίνοντας την κατακερματισμένη μορφή των όρων αναζήτησης που στάλθηκαν από τον χρήστη με τις κατακερματισμένες μορφές των προϊόντων ένα προς ένα. Οι όροι τής αναζήτησης καθώς και η ελάχιστη ομοιότητα και ο μέγιστος αριθμός των αποτελεσμάτων ορίζονται από τον χρήστη μέσω του προγράμματος «πελάτη».




*Εικόνα 15. Παράδειγμα κλήσης Search.*

Στο παραπάνω παράδειγμα εκτέλεσης της κλήσης Search βλέπουμε ότι η μόνη πληροφορία που στέλνει ο χρήστης στον EncryptedAuctionAggregator σχετικά με την αναζήτησή του είναι το LSH hash που δημιουργήθηκε από την εφαρμογή EncryptedAuctionClient. Έπειτα η απάντηση του EncryptedAuctionAggregator είναι μια λίστα από αντικείμενα που ανταποκρίνονται περισσότερο στην κωδικοποιημένη αναζήτηση του χρήστη. Για κάθε αντικείμενο μας δίνεται το μοναδικό Id του καθώς και η διεύθυνση του καταστήματος store.apiUrl από όπου προήλθε ώστε σε δεύτερο χρόνο να λάβουμε τις βασικές του παραμέτρους, όπως κατηγορία, μάρκα και μοντέλο. Παρατηρούμε επίσης ότι σε κάθε αντικείμενο έχει δοθεί μία δεκαδική τιμή similarity (από το 0 ως το 1) που αντιπροσωπεύει την ομοιότητα του αντικειμένου με τους όρους της αναζήτησης.

Βλέπουμε ότι για την αναζήτηση LSH χρησιμοποιούνται από την συνάρτηση οι δύο βασικές κλάσεις της βιβλιοθήκης LSHDotNet, MinHasher και LSHSearch. Για την αρχικοποίηση του MinHasher χρησιμοποιείται το κοινό κλειδί που χρησιμοποιούν όλες οι εφαρμογές του συστήματος. Αυτό συμβαίνει γιατί μόνο έτσι μπορούν να γίνουν συγκρίσεις μεταξύ των hashes. Για την αρχικοποίηση του LSHSearch χρησιμοποιείται ο δείκτης ομοιότητας Jaccard.

Αυτή η μέθοδος αναζήτησης είναι επαρκής για τις ανάγκες της εργασίας, αλλά θα παρουσίαζε κακή απόδοση σε μεγάλης κλίμακας υλοποίηση καθώς το σύνολο των δεδομένων αποθηκεύεται, έστω και πρόσκαιρα, στην μνήμη ram.

Για την καταχώρηση των προϊόντων των καταστημάτων χρησιμοποιούνται οι HTTP κλήσεις που δρομολογούνται στις διευθύνσεις /api/Stores/TryRegister και /api/Stores/Register.

Η κλήση GET δέχεται το αναγνωριστικό ID ενός καταστήματος και επιστρέφει στο κατάστημα που την καλεί το κλειδί του LSH ώστε να κωδικοποιήσει τα προϊόντα του.

Στην κλήση POST η εφαρμογή δέχεται το αναγνωριστικό ID ενός καταστήματος και μια λίστα από αντικείμενα HashedProductBase και επιστρέφει μήνυμα επιτυχούς ή ανεπιτυχούς κατάστασης.

```
[Route("api/[controller]")]
[ApiController]
public class StoresController : ControllerBase
{
    private readonly ILogger<StoresController> _logger;
    private readonly DBContext _db;
    public StoresController(ILogger<StoresController> logger, DBCont
ext dbContext)
    {
        _logger = logger;
        _db = dbContext;
    }
    [HttpGet("TryRegister/{guid}")]
    public ActionResult<int[][]> TryRegister([FromRoute] Guid guid)
    {
        if (_db.Stores.Any(s => s.Id == guid))
        {
            return Ok(_db.LSHConfig.LSHHashSeed);
        }
        else
        {
            return BadRequest();
        }
    }
    [HttpPost("Register/{guid}")]
    public ActionResult Register([FromRoute] Guid guid, [FromBody] L
ist<HashedProductBase> products)
    {
        if (_db.Stores.Any(s => s.Id == guid))
        {
            try
            {
                _db.HashedProducts.RemoveRange(_db.HashedProducts.Wh
ere(p => p.StoreId == guid));
```

```
                var dbProducts = products.Select(p => new HashedProd
uct()
                {
                    Id = p.Id,
                    Weights = p.Weights,
                    Hashes = p.Hashes,
                    Store = null,
                    StoreId = guid
                });
                _db.HashedProducts.AddRange(dbProducts);
                _db.SaveChanges();
                return Ok();
            }
            catch(Exception ex)
            {
                throw ex;
            }
        }
        else
        {
            return BadRequest();
        }
    }
}
```

Η κλάση StoresController διαχειρίζεται τις κλήσεις GET και POST /api/Stores με τις συναρτήσεις TryRegister και Register αντίστοιχα.

Και οι δύο συναρτήσεις δέχονται ως όρισμα το μοναδικό ID του καταστήματος και ελέγχουν στη βάση τους εάν είναι καταχωρημένο. Καταστήματα που δεν είναι καταχωρημένα δεν μπορούν να καταχωρήσουν προϊόντα τους στη βάση δεδομένων του EncryptedAuctionAggregator. Η καταχώρηση καταστημάτων γίνεται από το αρχείο seed.js κατά την πρώτη εκκίνηση του EncryptedAuctionAggregator.

Η συνάρτηση TryRegister επιστρέφει στα καταχωρημένα καταστήματα το κλειδί του LSH έτσι ώστε να παράγουν κατακερματισμένες μορφές των προϊόντων τους.

Η συνάρτηση Register δέχεται κατακερματισμένα προϊόντα και ανανεώνει τον κατάλογο του καταστήματος. Αρχικά αναζητά και διαγράφει το σύνολο των προηγουμένως καταχωρημένων προϊόντων του καταστήματος. Έπειτα αποθηκεύει τα νέα προϊόντα δημιουργώντας ταυτόχρονα συσχετίσεις μεταξύ των νέων προϊόντων και του καταστήματος.

## 4.5 EncryptedAuctionStore

Πρόκειται για μία εφαρμογή διαδικτύου που αποτελεί το κεντρικό υποσύστημα της εφαρμογής. Υλοποιημένο με χρήση των προαναφερθέντων τεχνολογιών .Net Core, ASP.Net Core και Entity Framework Core.

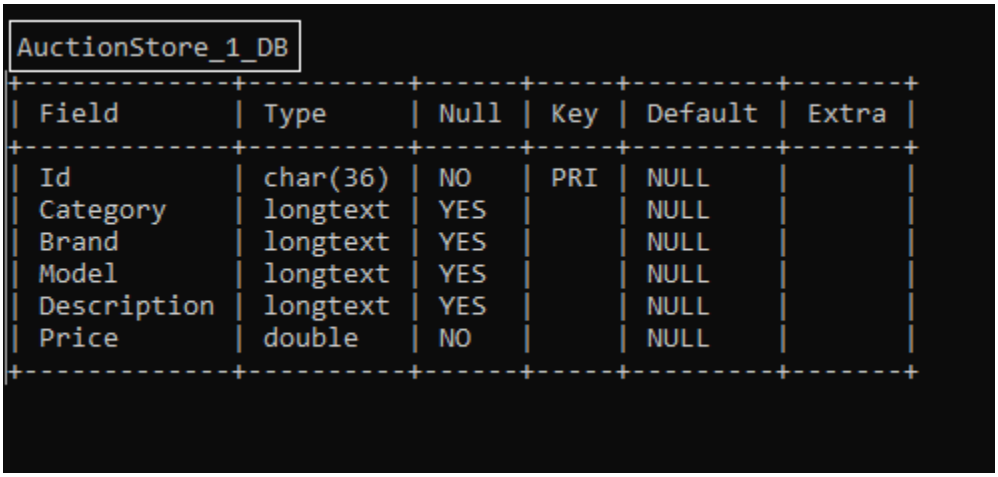
Επικοινωνεί με τα υπόλοιπα τμήματα του συστήματος με κλήσεις HTTP μέσω των Χειριστών (Controllers) του. Controller ονομάζουμε μια κλάση που διαχειρίζεται ένα σύνολο κλήσεων HTTP σχετικών, συνήθως, με ένα μοντέλο ή μια λειτουργία του συστήματος. Επίσης η εφαρμογή

λειτουργεί ως χρήστης της εφαρμογής EncryptedAuctionAggregator μέσω κλήσεων στην διεύθυνση /api/Stores για καταχώρηση των προϊόντων του καταστήματος.

Η αρχιτεκτονική του συστήματος προβλέπει μεγάλο αριθμό καταστημάτων να λειτουργούν ταυτόχρονα και να συνδέονται παράλληλα σε ένα κεντρικό EncryptedAuctionAggregator.

### 4.5.1 Μοντέλα & Βάση Δεδομένων

Για την αποθήκευση των δεδομένων της, κάθε ξεχωριστό κατάστημα χρησιμοποιεί μια σχεσιακή βάση MySQL η οποία εκτελείται σε ξεχωριστό docker container. Κάθε κατάστημα αποθηκεύει τα δεδομένα του σε έναν πίνακα Products χωρίς συσχετίσεις με άλλους πίνακες. Ο πίνακας εμφανίζεται στην παρακάτω εικόνα.



AuctionStore_1_DB

| Field | Type | Null | Key | Default | Extra |
|---|---|---|---|---|---|
| Id | char(36) | NO | PRI | NULL | |
| Category | longtext | YES | | NULL | |
| Brand | longtext | YES | | NULL | |
| Model | longtext | YES | | NULL | |
| Description | longtext | YES | | NULL | |
| Price | double | NO | | NULL | |

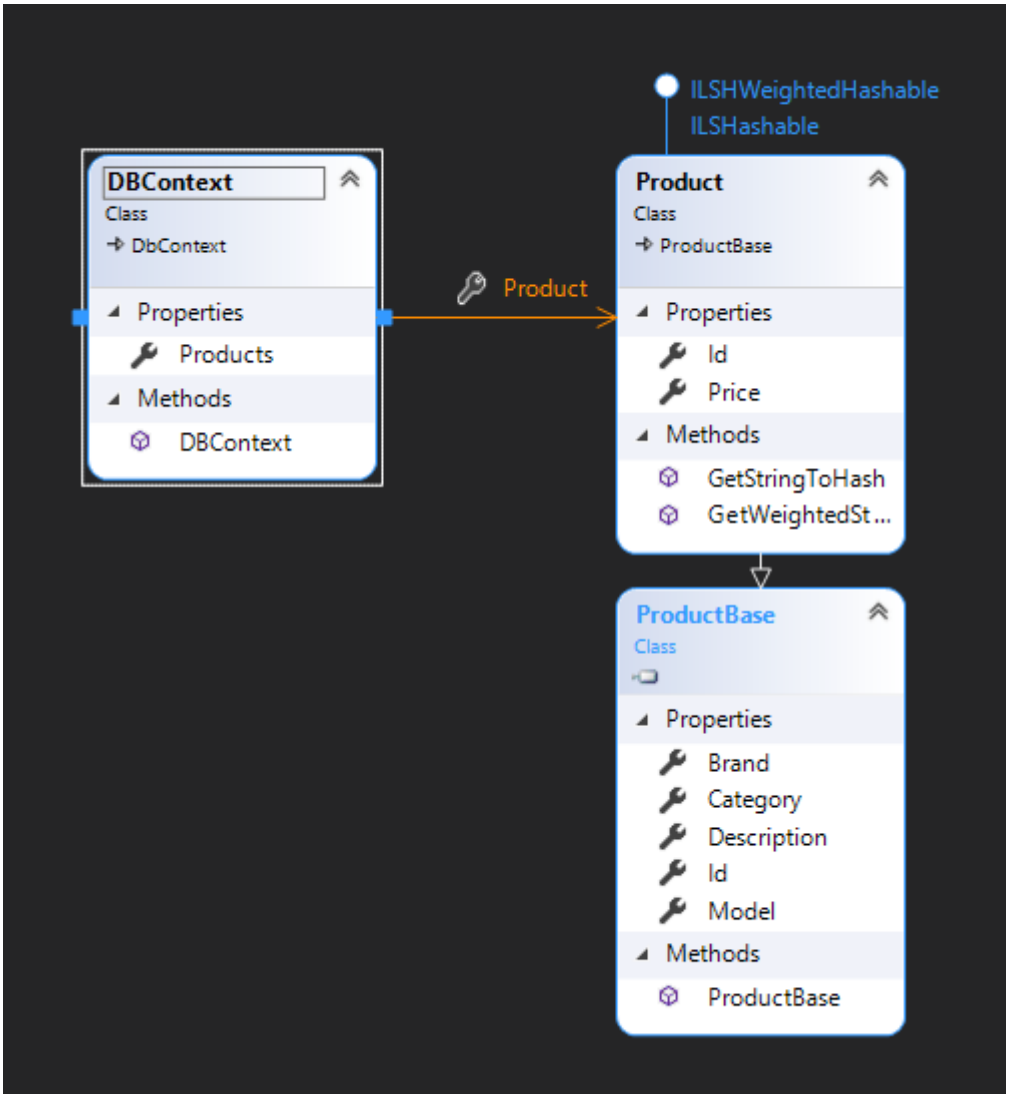
*Εικόνα 16. Σχήμα της βάσης δεδομένων για την εφαρμογή EncryptedAuctionStore.*

Το πεδίο Id που βλέπουμε στον παραπάνω πίνακα αποτελεί το ξεχωριστό κλειδί κάθε προϊόντος και αποθηκεύεται στην βάση δεδομένων χρησιμοποιώντας τον τύπο char(36) ενώ στην εφαρμογή μας χρησιμοποιείται ο τύπος Guid, όπως προαναφέρθηκε. Τα υπόλοιπα πεδία πέραν τις τιμής αποτελούνε περιγραφικά στοιχεία του προϊόντος τα οποία, μετά τον κατακερματισμό και την απόδοση σε αυτά βαρών, χρησιμοποιούνται στην αναζήτηση προϊόντων από τον EncryptedAuctionAggregator.

Η βάση αυτή, παρόλη την απλότητά της παράχθηκε με χρήση των εργαλείων Code First του Entity Framework Core έτσι ώστε να έχουμε καλύτερο έλεγχο των πράξεων στα δεδομένα της καθώς και ευκολότερη ενσωμάτωσή της με την εφαρμογή.

```
public class DBContext : DbContext
{
    public DBContext(DbContextOptions<DBContext> dbContextOptions)
        : base(dbContextOptions)
    { }
    public DbSet<Product> Products { get; set; }
}
```

Η κλάση DBContext η οποία είναι υπεύθυνη τόσο για τον σχηματισμό της βάσης όσο και για την ανάγνωση και εγγραφή προϊόντων στον πίνακα Products.




*Εικόνα 17. Διάγραμμα κλάσεων του EncryptedAuctionStore.*

Στο διάγραμμα κλάσεων βλέπουμε την κλάση DBContext καθώς και τη συσχέτιση της με την κλάση Products.

Η κλάση Products κληρονομεί από την κλάση ProductBase από την κοινή βιβλιοθήκη κλάσεων και την επεκτείνει χρησιμοποιώντας το Interface ILSHWeightedHashable ώστε να επιτρέψει την λειτουργικότητα της σταθμισμένης LSH αναζήτησης ανάμεσα σε προϊόντα. Βλέπουμε επίσης ότι η κλάση Product περιέχει την παράμετρο Price που υποδηλώνει την τιμή του προϊόντος, πληροφορία που είναι διαθέσιμη μόνο στο πλαίσιο του EncryptedAuctionStore και δεν μεταφέρεται στο EncryptedAuctionAggregator.

Για την αρχικοποίηση των δεδομένων της εφαρμογής χρησιμοποιούνται, κατά την πρώτη εκτέλεση, οι κλάσεις Seeder και Registrator.

```
public static class Seeder
{
    public static void Seedit(Registrator reg, string jsonData, DBCo
ntext context)
```

```
    {
        List<Product> products = JsonConvert.DeserializeObject<List<
Product>>(jsonData);
        if (!context.Products.Any())
        {
            try
            {
                context.AddRange(products);
                var tries = 3;
                while (tries > 0)
                {
                    tries -= 1;
                    try
                    {
                        reg.Register(products);
                        context.SaveChanges();
                        return;
                    }
                    catch(Exception ex)
                    {
                        Debug.WriteLine(ex.Message);
                    }
                }
            }
            catch
            {
                throw;
            }
        }
    }
}
```

Η κλάση Seeder παίρνει τα προϊόντα του καταστήματος διαβάζοντας τα από ένα αρχείο τύπου json το οποίο πρέπει να βρίσκεται στον υποκατάλογο της εφαρμογής με όνομα seed.json. Ένα μικρό παράδειγμα της μορφής του αρχείου εμφανίζεται παρακάτω.

```
[
  {
    "Id": "10d86b2a-80f8-449d-bbf9-e0bba37dfda8",
    "Brand": "Dell",
    "Model": "S2719DGF",
    "Price": 323.69,
    "Description": "Dell S2719DGF",
    "Category": "Monitor"
  },
  {
    "Id": "a64cfc4b-979a-44f6-b8b0-95bcf5438988",
```

```
    "Brand": "Samsung",
    "Model": "C24F390FHU",
    "Price": 106.25,
    "Description": "Samsung C24F390FHU",
    "Category": "Monitor"
  }
]
```

Το συγκεκριμένο αρχείο περιγράφει δύο προϊόντα. Για καλύτερη λειτουργία της αναζήτησης και των συγκρίσεων προτείνεται να χρησιμοποιηθούν περισσότερα προϊόντα. Στον κώδικα της εφαρμογής χρησιμοποιούνται 240 προϊόντα ανά κατάστημα.

Η κλάση Seeder αφού ανασύρει τα προϊόντα από το αρχείο, τα αποθηκεύει στην βάση δεδομένων της εφαρμογής και έπειτα, με εσωτερική χρήση της κλάσης Registrator καταχωρεί τις κατακερματισμένες μορφές τους στην υπηρεσία EncryptedAuctionAggregator.

```
public class Registrator
{
    private StoreBase storeObj;
    private RestClient _client;
    IConfiguration configuration { get; set; }
    public Registrator(IConfiguration conf, StoreBase storeObj)
    {
        this.configuration = conf;
        var aggregatorUrl = conf.GetValue<string>("aggregatorUrl");
        this.storeObj = storeObj;
        _client = new RestClient(aggregatorUrl);
    }
    private int[][] TryRegister()
    {
        var req = new RestRequest($"/api/stores/TryRegister/{storeOb
j.Id}");
        try
        {
            var res = _client.Get<int[][]>(req);
            if (res.StatusCode == System.Net.HttpStatusCode.OK)
            {
                var seed = JsonConvert.DeserializeObject<int[][]>(re
s.Content);
                return seed;
            }
            else
            {
                if (res.ErrorException != null)
                {
                    throw res.ErrorException;
                }
```

```
                throw new Exception(res.StatusDescription);
            }
        }
        catch
        {
            throw;
        }
    }
    public void Register(List<Product> products)
    {
        try
        {
            var hashSeed = TryRegister();
            var minHasher = new MinHasher<Guid>(hashSeed);
            var hashedProducts = new List<HashedProductBase>();

            foreach (var product in products)
            {
                var id = product.Id;
                var hashedProduct = new HashedProductBase()
                {
                    Id = id
                };
                var weightsL = new List<double>();
                var hashL = new List<int[]>();
                var toHash = product.GetWeightedStringsToHash();
                foreach(var sth in toHash)
                {
                    var weight = sth.Item2;
                    var str = sth.Item1;
                    var hash = minHasher.GetMinHashSignature(str);
                    weightsL.Add(weight);
                    hashL.Add(hash);
                    hashedProduct.Hashes = hashL.ToArray();
                    hashedProduct.Weights = weightsL.ToArray();
                }
                hashedProducts.Add(hashedProduct);
            }
            var req = new RestRequest($"/api/stores/Register/{storeO
bj.Id}");
            req.AddJsonBody(hashedProducts);
            var res = _client.Post(req);
            if(res.StatusCode == System.Net.HttpStatusCode.OK)
            {
                return;
```

```
            }
            else
            {
                throw new Exception(res.StatusDescription);
            }
        }
        catch
        {
            throw;
        }
    }
}
```

Η κλάση Registrator καταχωρεί τα προϊόντα του καταστήματος κάνοντας χρήση της των WebAPI του EncryptedAuctionAggregator /api/stores/TryRegister και /api/Stores/Register.

- Αρχικά με την συνάρτηση TryRegister πραγματοποιεί μια HTTP κλήση η οποία παίρνει ως απάντηση το LSH κλειδί του EncryptedAuctionAggregator.
- Έπειτα χρησιμοποιεί αυτό το κλειδί στην συνάρτηση Register ώστε να κατακερματίσει τα προϊόντα που η κλάση Seeder τοποθέτησε στην βάση δεδομένων. Έτσι για κάθε αντικείμενο τύπου Product δημιουργεί το αντίστοιχο αντικείμενο τύπου HashedProductBase. Για να κάνει την μετατροπή χρησιμοποιεί το μοναδικό Id του Product και την συνάρτηση GetWeightedStringsToHash του Interface ILSHWeightedHashable που ικανοποιεί ο τύπος Product.
- Τέλος πραγματοποιεί μια HTTP Post κλήση προς τον EncryptedAuctionAggregator στη διεύθυνση /api/Stores/Register μεταφέροντας τη λίστα των αντικειμένων HashedProductBase στο σώμα της κλήσης.

```
public class Product : ProductBase, ILSHWeightedHashable
{
    [Key]
    public override Guid Id { get; set; }
    [JsonIgnore]
    public double Price { get; set; }
    public List<Tuple<string, double>> GetWeightedStringsToHash()
    {
        return new List<Tuple<string, double>>()
        {
            new Tuple<string, double>(Brand, 0.1),
            new Tuple<string, double>(Model, 0.4),
            new Tuple<string, double>(Brand + " " + Description, 0.5),
        };
    }
}
```

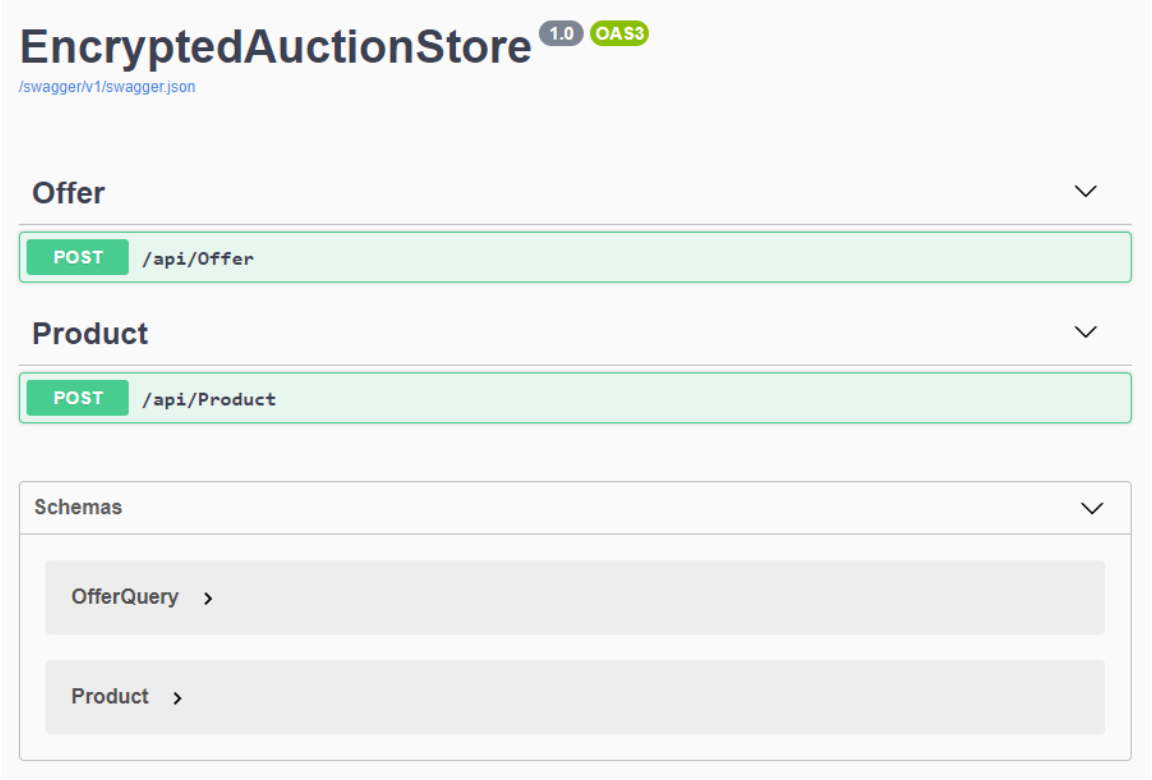
Η κλάση Product βλέπουμε ότι ικανοποιεί το Interface ILSHWeightedHashable μέσω της συνάρτησης GetWeightedStringsToHash. Η συνάρτηση αυτή αποδίδει βάρη στις παραμέτρους του προϊόντος ως εξής:

- Η μάρκα του προϊόντος συμβάλει κατά 10% στο συνολικό βάρος της αναζήτησης.
- Το μοντέλο του προϊόντος συμβάλει κατά 40% στο συνολικό βάρος της αναζήτησης.
- Ο συνδυασμός μάρκας και μοντέλου συμβάλει κατά 50% στο συνολικό βάρος της αναζήτησης

Οι τιμές αυτές επιλέχθησαν μετά από πειραματισμό ώστε να δίνουν σωστά αποτελέσματα σε συνήθεις αναζητήσεις.

### 4.5.2 Λειτουργικότητα

Η διεπαφή της εφαρμογής όπως απεικονίζεται με χρήση της βιβλιοθήκης swagger εμφανίζεται παρακάτω.




*Εικόνα 17. Απεικόνιση διεπαφής EncryptedAuctionAggregator μέσω της βιβλιοθήκης swagger.*

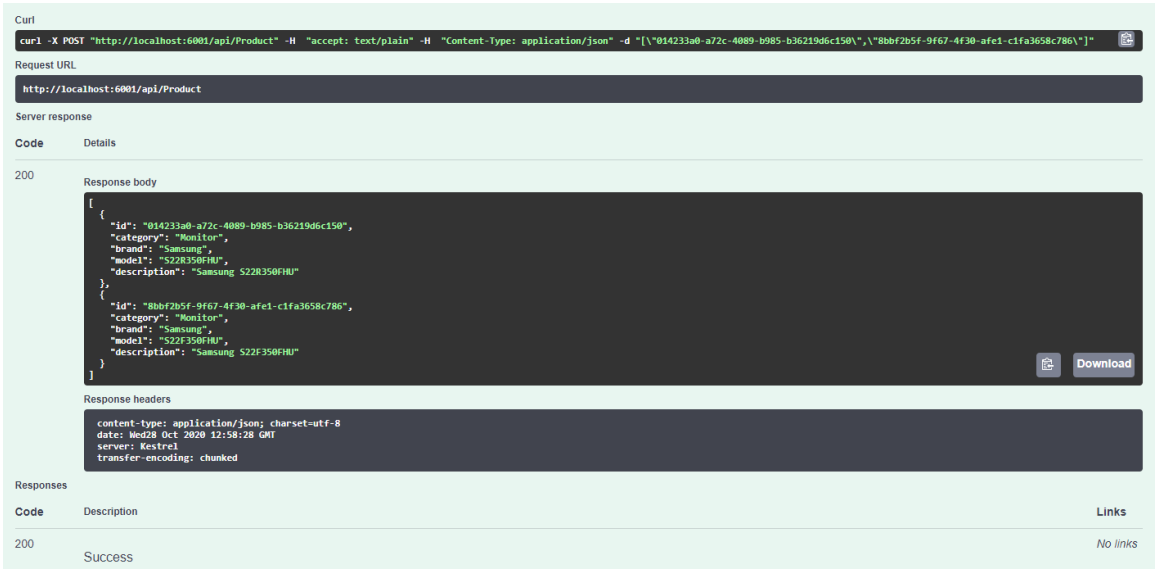
Παρατηρούμε ότι η εφαρμογή προσφέρει 2 λειτουργίες την λειτουργία λήψης προσφοράς και την λειτουργία ανάκτησης προϊόντος.

Για την ανάκτηση προϊόντων από τους πελάτες χρησιμοποιείται η κλάση ProductController, η οποία διαχειρίζεται τις HTTP κλήσεις που δρομολογούνται στην διεύθυνση /api/Product.

Στην κλήση POST η εφαρμογή δέχεται στο σώμα του HTTP μηνύματος μια λίστα από μοναδικά Id προϊόντων και επιστρέφει στο σώμα της απάντησης μια λίστα από αντικείμενα τύπου Product.

Έτσι η εφαρμογή του πελάτη μπορεί μετά την αναζήτηση στην υπηρεσία EncryptedAuctionAggregator να χρησιμοποιήσει τα Id που έλαβε για να λάβει τις πληροφορίες του κάθε προϊόντος που ταιριάζει στην αναζήτηση του χρήστη.

```
[Route("api/[controller]")]
[ApiController]
public class ProductController : ControllerBase
{
    private ILogger<ProductController> _logger;
    private DBContext _db;
    public ProductController(ILogger<ProductController> logger, DBCo
ntext dbContext)
    {
        _logger = logger;
        _db = dbContext;
    }
    [HttpPost]
    public List<Product> GetProducts([FromBody] List<Guid> guids)
    {
        return _db.Products.Where(p => guids.Contains(p.Id)).ToList(
);
    }
}
```




*Εικόνα 18. Παράδειγμα απάντησης τις κλήσης /api/Product*

Στην απάντηση της κλήσης /api/Product παρατηρούμε ότι επιστρέφεται το σύνολο της πληροφορίας των προϊόντων που δεν ήταν γνωστή στον EncryptedAuctionAggregator. Έχοντας πλέον αυτήν την πληροφορία η εφαρμογή EncryptedAuctionClient μπορεί να σχηματίσει για τον χρήστη μια λίστα με τα πιθανά αντικείμενα που του προτείνονται από το σύστημα.

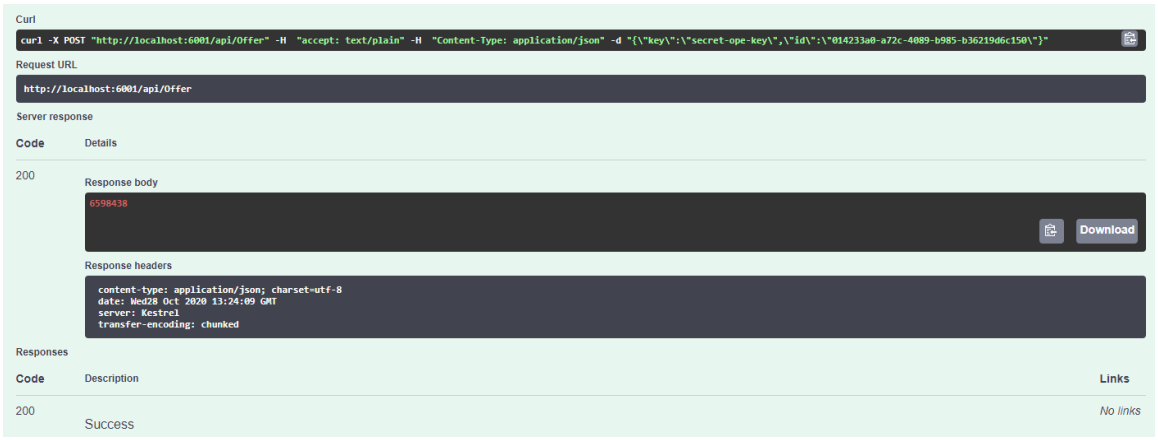
Για την λήψη προσφοράς για ένα προϊόν από τους πελάτες χρησιμοποιείται η κλάση OfferController, η οποία διαχειρίζεται τις HTTP κλήσεις που δρομολογούνται στην διεύθυνση /api/Offer.

Στην κλήση POST η εφαρμογή δέχεται στο σώμα του HTTP μηνύματος ένα αντικείμενο τύπου OfferQuery και επιστρέφει στο σώμα της απάντησης η τιμή του προϊόντος.

Η επιλογή του να λαμβάνονται σε διαφορετικό βήμα οι πληροφορίες του προϊόντος και η τιμή του προϊόντος πάρθηκε ώστε η λειτουργία της εφαρμογής να προσομοιώνει την λειτουργία μιας δημοπρασίας.

```
[Route("api/[controller]")]
[ApiController]
public class OfferController : ControllerBase
{
    private readonly ILogger<OfferController> _logger;
    private readonly DBContext _db;
    public OfferController(ILogger<OfferController> logger, DBContex
t dbContext)
    {
        _logger = logger;
        _db = dbContext;
    }
    [HttpPost]
    public long GetOffer(OfferQuery query)
    {
        var ope = new OPE(query.Key);
        var prod = _db.Products.FirstOrDefault(p => p.Id == query.Id
);
        var encryptedPrice = ope.Encrypt(Convert.ToInt32(Math.Ceilin
g(prod.Price)));
        return encryptedPrice;
    }
}
```

Για την λήψη της τιμής ενός προϊόντος χρησιμοποιείται η συνάρτηση GetOffer. Βλέπουμε ότι η συνάρτηση χρησιμοποιεί την συνάρτηση Encrypt της βιβλιοθήκης OrderPreservingEncryptionDotNet. Η συγκεκριμένη βιβλιοθήκη χρησιμοποιεί ένα κλειδί ώστε να κρυπτογραφήσει μια τιμή ακεραίου. Η ανάγκη για κρυπτογράφηση απαλείφεται καθώς δεν υπάρχει ενδιάμεσος κόμβος στην επικοινωνία μεταξύ πελάτη και καταστήματος όπως υπήρχε στον αρχικό σχεδιασμό του συστήματος, έτσι η κρυπτογράφηση αυτή θεωρείται πλέον περιττή. Ο κώδικας όμως δεν αφαιρέθηκε έτσι ώστε να μπορεί να χρησιμοποιηθεί σε μελλοντική υλοποίηση αν αυτό καταστεί αναγκαίο.




*Εικόνα 18. Παράδειγμα απάντησης τις κλήσης /api/Offer*

Στο παράδειγμα εκτέλεσης της κλήσης /api/Offer παρατηρούμε ότι ο χρήστης στέλνει μέσω του EncryptedAuctionClient το Id του προϊόντος που τον ενδιαφέρει καθώς και ένα κλειδί κρυπτογράφησης για την κρυπτογράφηση της πραγματικής τιμής και λαμβάνει την κρυπτογραφημένη τιμή του προϊόντος, την οποία μετά μπορεί να επαναφέρει χρησιμοποιώντας το κλειδί κρυπτογράφησης. Έτσι η εφαρμογή EncryptedAuctionClient μπορεί πλέον να ιεραρχήσει τις προσφορές των καταστημάτων σε φθίνουσα σειρά τιμής.

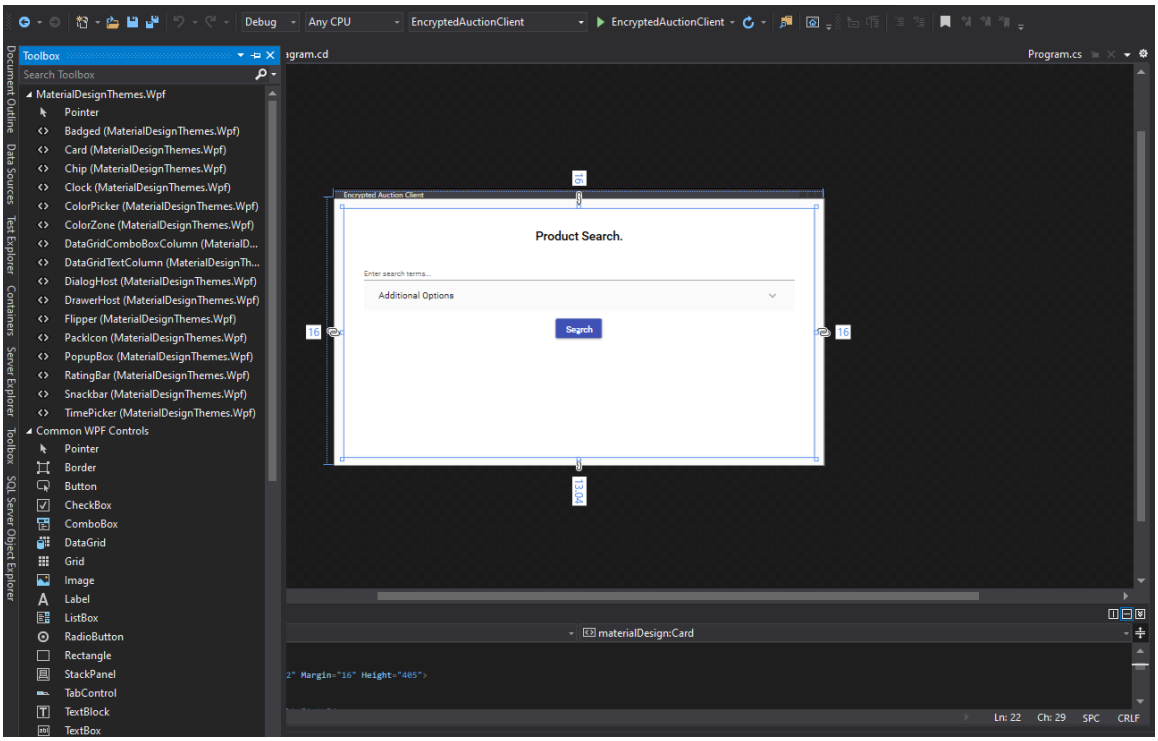
## 4.6 EncryptedAuctionClient

Η εφαρμογή του πελάτη είναι μια εφαρμογή γραφικού περιβάλλοντος που χρησιμοποιεί το πακέτο Windows Presentation Foundation για την υλοποίηση της.

### 4.6.1 Εικαστικός σχεδιασμός

Για ευκολότερη διαμόρφωση του εικαστικού σκέλους της εφαρμογής χρησιμοποιήθηκε το πρόγραμμα Visual Studio 2019. Το πρόγραμμα αυτό μας παρέχει τη δυνατότητα οπτικού σχεδιασμού εφαρμογών με την χρήση του visual designer. Πρόκειται για ένα γραφικό περιβάλλον στο οποίο μια πληθώρα γραφικών στοιχείων δίνονται στον χρήστη για να τα βάλει σε όποιο σημείο της εφαρμογής του επιλέξει. Έτσι ο χρήστης του Visual Studio μπορεί να σχεδιάσει γραφικά περιβάλλοντα σε WPF χωρίς να έχει γνώση της γλώσσας XAML. Χρησιμοποιώντας την βιβλιοθήκη MaterialDesignThemes μας παρέχονται επιπλέον γραφικά στοιχεία που κάνουν ευκολότερη την σωστή εικαστική απεικόνιση του αποτελέσματος που επιθυμούμε.

Η εφαρμογή σχεδιάστηκε με γνώμονα την απλότητα και την λειτουργικότητα, χωρίς στοιχεία τα οποία μπορεί να είναι επιθυμητά σε μια εμπορική εφαρμογή αλλά κρίνονται ως μη αναγκαία σε μια εφαρμογή που στην ουσία κάνει επίδειξη της λειτουργίας ενός συστήματος.




*Εικόνα 19. Περιβάλλον Visual WPF Designer*

### 4.6.2 Λειτουργικότητα

Η εφαρμογή παρέχει στον χρήστη τη δυνατότητα να γραψεί τους όρους αναζήτησής του καθώς και να επιλέξει τις παραμέτρους για ελάχιστη ομοιότητα και μέγιστο αριθμο αποτελεσμάτων. Όταν ο χρήστης είναι έτοιμος και πατήσει το κουμπί για αναζήτηση καλειται η συνάρτηση Button_Click η οποία καλεί εσωτερικά την συνάρτηση Search.

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
    if (!String.IsNullOrEmpty(textBox.Text))
    {
        double minSim = 0.3;
        minSim = MinimumSimilaritySlider.Value;
        int maxResults = 5;
        int.TryParse(MaxResultsTextBox.Text, out maxResults);
        var searchQuery = textBox.Text;
        try
        {
            var results = await Search(searchQuery, maxResults, minS
im);
            if (results.Count == 0){
                MessageBox.Show("No result was found for your search
 query, please try again!");
                return;
            }
```

```
            var searchResultWindow = new SearchResultWindow(searchQu
ery, results);
            searchResultWindow.Show();
        }
        catch
        {
            throw;
        }
    }
}
```

Η συνάρτηση ButtonClick διαχειρίζεται το πάτημα του κουμπιού αναζήτησης. Συγκεντρώνει από το γραφικό περιβάλλον τις επιλογές του χρήστη και καλεί την συνάρτηση Search. Εάν βρεθούν αποτελέσματα από την αναζήτηση μεταφέρει τον χρήστη στην οθόνη αποτελεσμάτων. Αν δεν βρεθούν αποτελέσματα, εμφανίζει στον χρήστη ένα μήνυμα που τον ενημερώνει και τον καλεί να προσπαθήσει ξανα.

```
private async Task<List<SearchResult>> Search(string searchString, i
nt maxResults, double minSim)
{
    var hashSeed = await apiClient.GetHashSeed();
    var minHasher = new MinHasher<Guid>(hashSeed);
    var hashedSearchString = minHasher.GetMinHashSignature(searchStr
ing);
    var hashedResults = (await apiClient.HashedSearch(new HashedSear
chQuery()
    {
        MaxResults = maxResults,
        MinimumSimilarity = minSim,
        SearchTerm = hashedSearchString
    }))?.ToList();
    var groupedResults = hashedResults.GroupBy(r => r.Store);
    var tasks = new List<Task<IEnumerable<ProductBase>>>();
    foreach(var storeResults in groupedResults)
    {
        tasks.Add(apiClient.GetProducts(storeResults.Key, storeResul
ts.Select(r => r.Product.Id).ToList()));
    }
    await Task.WhenAll(tasks);
    foreach(var task in tasks.Where(t => !t.IsFaulted))
    {
        foreach(var product in task.Result)
        {
            var idx = hashedResults.ToList().FindIndex(h => h.Produc
t.Id == product.Id);
            hashedResults[idx].Product = product;
```

```
        }
    }
    return hashedResults;
}
```

Η συνάρτηση Search εκτελεί τις ακόλουθες πράξεις:

- Καλεί την HTTP κλήση GET: /api/Search ώστε να πάρει το κλειδί του LSH.
- Δημιουργεί το LSH hash των όρων αναζήτησης με χρήση του κλειδιού που έλαβε.
- Καλεί την HTTP κλήση POST: /api/Search στέλνοντας την αναζήτηση του χρήστη στο EncryptedAuctionAggregator.
- Χωρίζει τα αποτελέσματα που έλαβε ανά κατάστημα.
- Καλεί ξεχωριστά σε κάθε κατάστημα την HTTP κλήση POST: /api/Products για να λάβει τις πληροφορίες για τα προϊόντα.
- Επιστρέφει τα προϊόντα.

Όταν η αναζήτηση ολοκληρωθεί η εφαρμογή συγκεντρώνει τα αποτελέσματα από τα καταστήματα και δημιουργεί μια λίστα με προϊόντα καθώς και τον αριθμό των καταστημάτων που παρέχουν το κάθε προϊόν. Ο χρήστης μετά μπορεί να επιλέξει ένα προϊόν που παρέχεται από πολλά καταστήματα και να λάβει για αυτό προσφορές.

```
private async Task GetOffers(ProductViewModel product)
{
    List<OfferViewModel> offersToGet = new List<OfferViewModel>();
    foreach (var result in _results)
    {
        if(
                result.Product.Brand == product.Brand
            && result.Product.Model == product.Model
            && !offersToGet.Any(o=>o.ProductId == result.Product.Id)
            )
        {
            offersToGet.Add(new OfferViewModel()
            {
                ProductId = result.Product.Id,
                StoreApiUrl = result.Store.ApiUrl,
                StoreName = result.Store.Name
            });
        }
    }
    var gotOffers = await ApiClient.GetOffers(offersToGet);
    OfferList = gotOffers.OrderBy(o => o.EncryptedPrice).ToList();
    OnPropertyChanged(nameof(OfferList));
    OfferListVisibility = Visibility.Visible;
    OnPropertyChanged(nameof(OfferListVisibility));
    OfferTerm = $"Offers for: \"{product.Brand} {product.Model}\"";
    OnPropertyChanged(nameof(OfferTerm));
}
```

Την λειτουργία της λήψης προσφοράς επιτελεί η συνάρτηση GetOffers. Η συνάρτηση αυτή για το προϊόν που επέλεξε ο χρήστης κάνει τις απαραίτητες κλήσεις HTTP Post /api/Offer στα καταστήματα που διαθέτουν το προϊόν, δέχεται τις τιμές των καταστημάτων και δημιουργεί μία λίστα φθίνουσας τιμής την οποία εμφανίζει στον χρήστη.

# 5. Εκτέλεση και χρήση συστήματος

Η εκτέλεση και λειτουργία του συστήματος περιλαμβάνει δύο σκέλη, την εκτέλεση των υπηρεσιών EncryptedAuctionAggregator και EncryptedAuctionStore και αφού οι υπηρεσίες ολοκληρώσουν την αρχικοποίησή τους, την εκτέλεση του προγράμματος «πελάτη» EncryptedAuctionClient και την αναζήτηση στα προϊόντα.

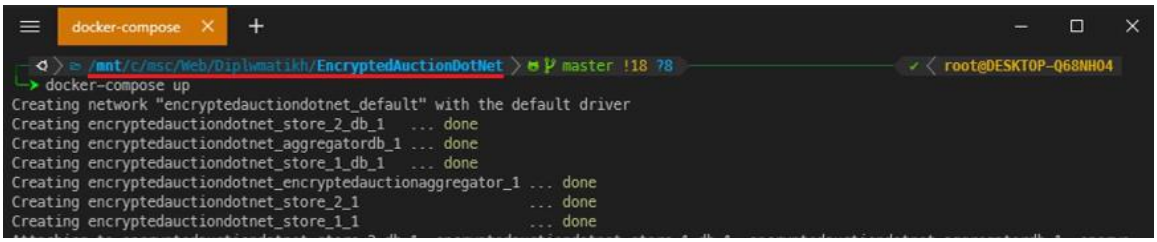
## 5.1 Εκτέλεση υπηρεσιών

Για την εγκατάσταση και εκτέλεση του συστήματος είναι προαπαιτούμενα:

- Να υπάρχει εγκατεστημένη η τελευταία έκδοση του docker και docker-compose.
- Να έχουμε τα αρχεία κώδικα της λύσης της εφαρμογής. Ο κώδικας βρίσκεται διαθέσιμος στην πλατφόρμα github με το όνομα EncryptedAuctionDotNet.

Λεπτομερείς οδηγίες για την παραμετροποίηση και εγκατάσταση του συστήματος είναι εκτός του στόχου της παρούσας εργασίας. Για τον λόγο αυτό αναγράφονται στο αρχείο README.md του κώδικα στο github.

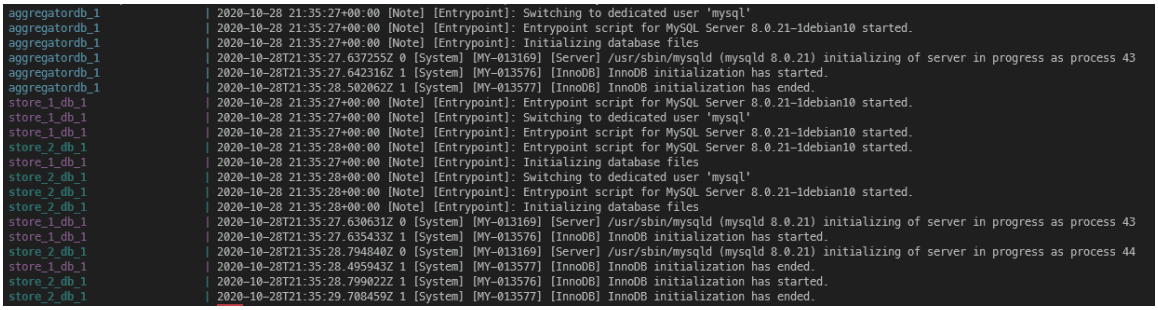
Παρουσιάζονται και αναλύονται παρακάτω κάποιες εικόνες από την εκτέλεση των υπηρεσιών στη γραμμή εντολών με χρήση της εντολής docker-compose up.




*Εικόνα 20. Καταγραφές αρχικοποίησης υπηρεσιών*

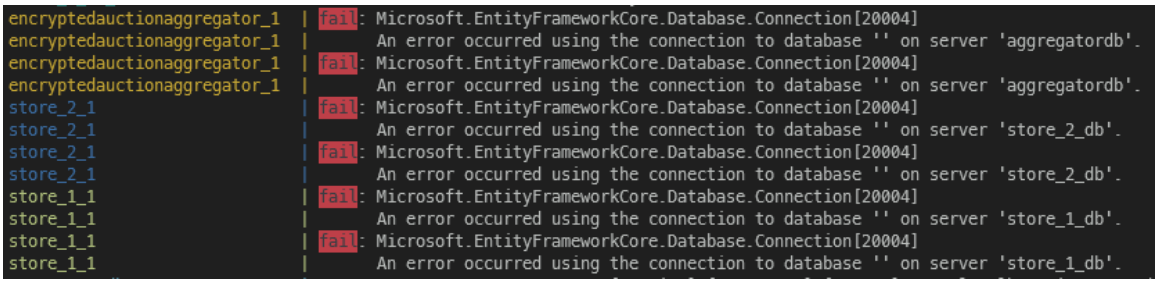
Βλέπουμε ότι η εντολή εκτελείται όταν ο χρήστης βρίσκεται στον κεντρικό φάκελο του project. Η εντολή αυτή ξεκινάει την αρχικοποίηση και εκτέλεση 6 docker container.

- Ενός EncryptedAuctionAggregator (encryptedauctiondotnet_encryptedauctionaggregator_1)
- Δύο EncryptedAuctionStore (encryptedauctiondotnet_store_1_1 & encryptedauctiondotnet_store_2_1)
- Τριών βάσεων δεδομένων MySQL (aggregatordb_1, store_1_db_1 & store_2_db_1)



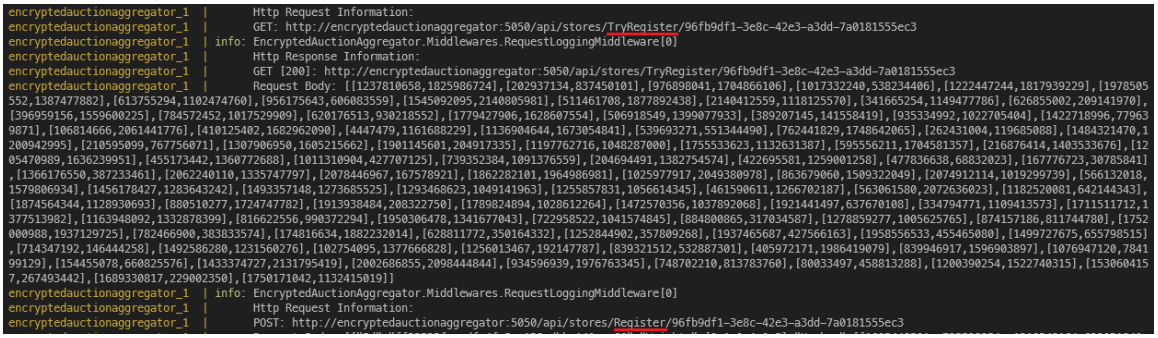

*Εικόνα 21. Καταγραφές αρχικοποίησης βάσεων δεδομένων*

Στη συνέχεια παρατηρούμε την αρχικοποίηση των 3 βάσεων δεδομένων. Σημαντικό είναι να σημειωθεί ότι με αυτόν τον τρόπο εκτέλεσης, δηλαδή μέσω του docker-compose από την γραμμή εντολών, όλες οι υπηρεσίες περνούν τα μνήματά τους στην ίδια κονσόλα, κάνοντας έτσι ευκολότερη την παρακολούθηση της κατάστασης του συστήματος.




*Εικόνα 22. Μηνύματα αποτυχίας σύνδεσης στη βάση δεδομένων*

Κατά την αρχικοποίηση των υπηρεσιών ιστού, κάθε υπηρεσία προσπαθεί να συνδεθεί στη βάση δεδομένων της. Είναι πολύ πιθανό να αποτύχει η σύνδεση καθώς η βάση δεδομένων βρίσκεται ακόμα σε διαδικασία αρχικοποίησης. Για τον λόγο αυτό κάθε υπηρεσία κάνει επανειλημμένες προσπάθειες για σύνδεση στη βάση δεδομένων ως ότου αυτό καταστεί εφικτό.



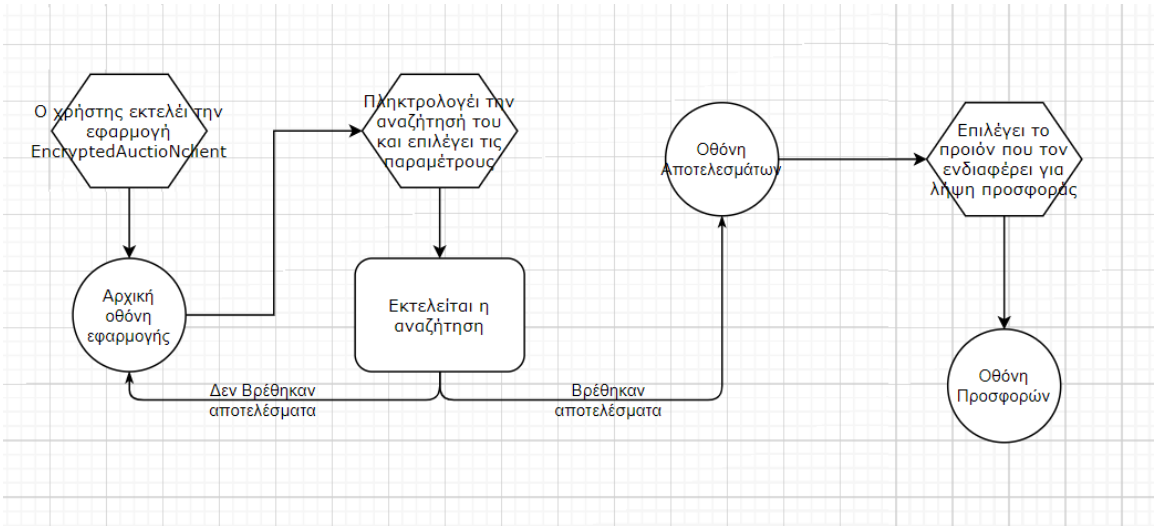

*Εικόνα 23. Κλήσεις καταχώρησης προϊόντων στη βάση του EncryptedAuctionAggregator*

Όταν οι βάσεις είναι πλέον έτοιμες οι υπηρεσίες συνδέονται σε αυτές και είναι πλέον διαθέσιμες για να προσφέρουν την λειτουργικότητά τους μέσω κλήσεων HTTP.

Η πρώτες κλήσεις γίνονται από τα καταστήματα προς το EncryptedAuctionAggregator για καταχώρηση των προϊόντων τους, εδώ βλέπουμε να εκτελείται η κλήση TryRegister που δίνει στο κατάστημα το κλειδί του LSH. Έπειτα ακολουθεί η κλήση Register που μεταφέρει τα κρυπτογραφημένα προϊόντα του καταστήματος στο EncryptedAuctionAggregator. Έπειτα από αυτή την διαδικασία οι υπηρεσίες είναι έτοιμες να δεχτούν κλήσεις από την εφαρμογή «πελάτη».

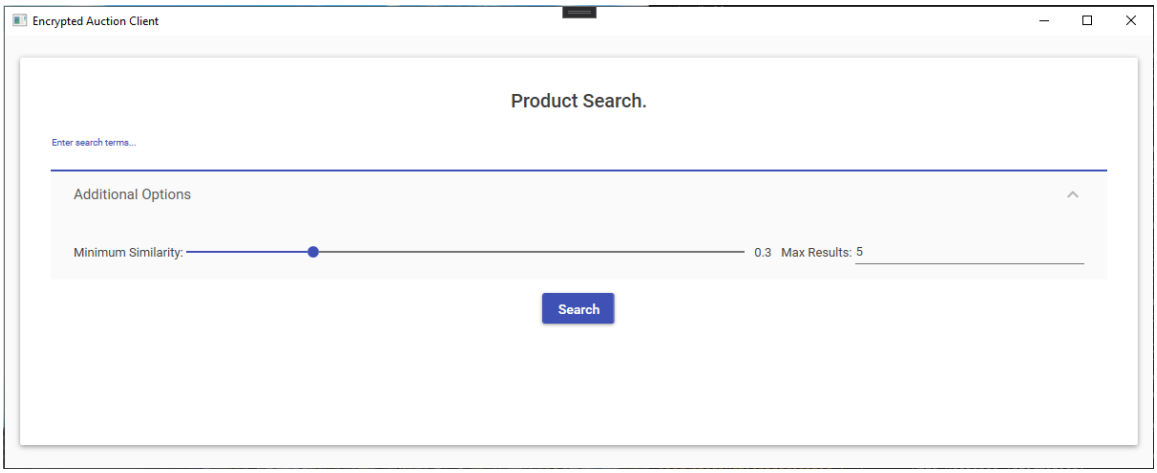
## 5.2 Εκτέλεση αναζήτησης μέσω της εφαρμογής

Ή βασική περίπτωση χρήσης όσον αφορά τους πελάτες του συστήματος είναι η αναζήτηση προϊόντων και σύγκριση τιμών μεταξύ των διάφορων καταστημάτων που διαθέτουν το επιθυμητό προϊόν.




*Εικόνα 24. Διάγραμμα χρήσης της εφαρμογής EncryptedAuctionClient*

Όπως βλέπουμε και από το διάγραμμα όταν ο χρήστης εκκινεί την εφαρμογή οδηγείται στην αρχική οθόνη της εφαρμογής.

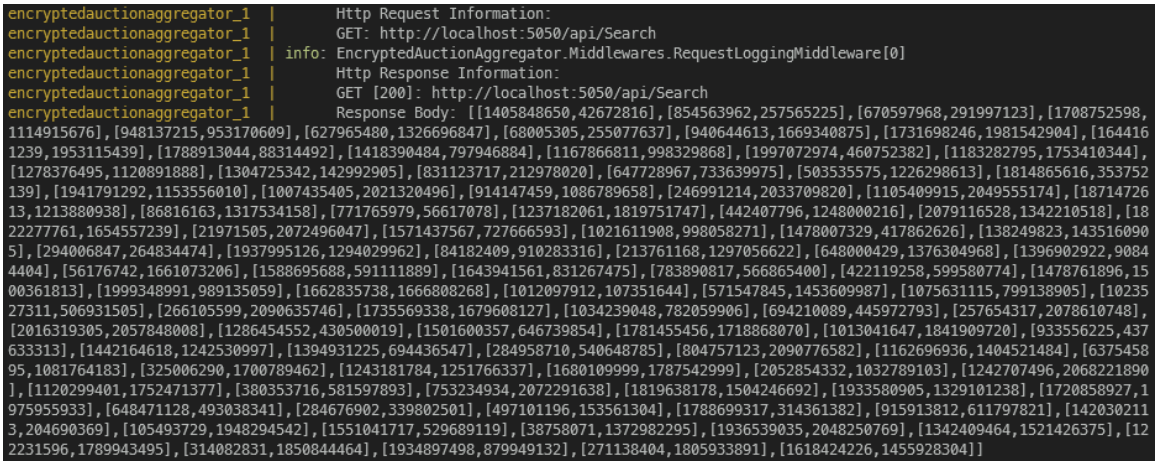



*Εικόνα 25. Αρχική οθόνη εφαρμογής EncryptedAuctionClient*

Στην οθόνη αυτή ο χρήστης πληκτρολογεί τους όρους της αναζήτησης του και μπορεί να ρυθμίσει τις παραμέτρους της αναζήτησης χρησιμοποιώντας τα πεδία “Minimum Similarity” και “Max Results”. Έπειτα ο χρήστης πατώντας το κουμπί “Search” πραγματοποιεί την αναζήτηση.

Τότε εκτελούνται οι ακόλουθες κλήσεις:

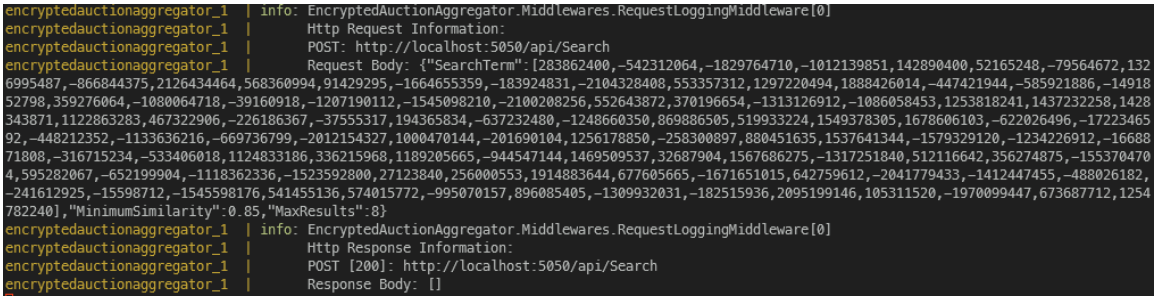
- EncryptedAuctionClient → EncryptedAuctionStore GET /api/Search



```
encryptedauctionaggregator_1  |       Http Request Information:
encryptedauctionaggregator_1  |       GET: http://localhost:5050/api/Search
encryptedauctionaggregator_1  | info: EncryptedAuctionAggregator.Middlewares.RequestLoggingMiddleware[0]
encryptedauctionaggregator_1  |       Http Response Information:
encryptedauctionaggregator_1  |       GET [200]: http://localhost:5050/api/Search
encryptedauctionaggregator_1  |       Response Body: [[1405848650,42672816],[854563962,257565225],[670597968,291997123],[1708752598,
1114915676],[948137215,953170609],[627965480,1326698647],[68005305,255077637],[940644613,1669340875],[1731698246,1981542904],[164416
1239,1953115439],[1788913044,88314492],[1418390484,797946884],[1167866811,998329868],[1997072974,460752382],[1183282795,1753410344],
[1278376495,1120891888],[1304725342,142992905],[831123717,212978020],[647728967,733639975],[503535575,1226298613],[1814865616,353752
139],[1941791292,1153556010],[1007435405,2021320496],[914147459,1086789658],[246991214,2033709820],[1105409915,2049555174],[18714726
13,1213880938],[86816163,1317534158],[771765979,56617078],[1237182061,1819751747],[442407796,1248000216],[2079116528,1342210518],[18
22277761,1654557239],[21971505,2072496047],[1571437567,727666593],[1021611908,998058271],[1478007329,417862626],[138249823,143516090
5],[294006847,264834474],[1937995126,1294029962],[84182409,910283316],[213761168,1297056622],[648000429,1376304968],[1396902922,9084
4404],[56176742,1661073206],[1588695688,591111889],[1643941561,831267475],[783890817,566865400],[422119258,599580774],[1478761896,15
00361813],[1999348991,989135059],[1662835738,1666808268],[1012097912,107351644],[571547845,1453609987],[1075631115,799138905],[10235
27311,506931505],[266105599,2090635746],[1735569338,1679608127],[1034239048,782059906],[694210089,445972793],[257654317,2078610748],
[2016319305,2057848008],[1286454552,430500019],[1501600357,646739854],[1781455456,1718868070],[1013041647,1841909720],[933556225,437
633313],[1442164618,1242530997],[1394931225,694436547],[284958710,540648785],[804757123,2090776582],[1162696936,1404521484],[6375458
95,1081764183],[325006290,1700789462],[1243181784,1251766337],[1680109999,1787542999],[2052854332,1032789103],[1242707496,2068221890
],[1120299401,1752413777],[380353716,581597893],[753234934,2072291638],[1819638178,1504246692],[1933580905,1329101238],[1720858927,1
975955933],[648471128,493038341],[284676902,339802501],[497101196,153561304],[1788699317,314361382],[915913812,611797821],[142030211
3,204690369],[105493729,1948294542],[1551041717,529689119],[38758071,1372982295],[1936539035,2048250769],[1342409464,1521426375],[12
2231596,1789943495],[314082831,1850844464],[1934897498,879949132],[271138404,1805933891],[1618424226,1455928304]]
```


*Εικόνα 26. Καταγραφή κλήσης GET: /api/Search*

- EncryptedAuctionClient → EncryptedAuctionStore POST /api/Search



```
encryptedauctionaggregator_1  | info: EncryptedAuctionAggregator.Middlewares.RequestLoggingMiddleware[0]
encryptedauctionaggregator_1  |       Http Request Information:
encryptedauctionaggregator_1  |       POST: http://localhost:5050/api/Search
encryptedauctionaggregator_1  |       Request Body: {"SearchTerm":[283862400,-542312064,-1829764710,-1012139851,142890400,52165248,-79564672,132
6995487,-866844375,2126434464,568360994,91429295,-1664655359,-183924831,-2104328408,553357312,1297220494,1888426014,-447421944,-585921886,-14918
52798,359276064,-1080064718,-39160918,-1207190112,-1545098210,-2100208256,552643872,370196654,-1313126912,-1086058453,1253818241,1437232258,1428
343871,1122863283,467322906,-226186367,-37555317,194365834,-637232480,-1248660350,869886505,519933224,1549378305,1678606103,-622026496,-17223465
92,-448212352,-1133636216,-669736799,-2012154327,1000470144,-201690104,1256178850,-258300897,880451635,1537641344,-1579329120,-1234226912,-16688
71808,-316715234,-533406018,1124833186,336215968,1189205665,-944547144,1469509537,32687904,1567686275,-1317251840,512116642,356274875,-155370470
4,595282067,-652199904,-1118362336,-1523592800,27123840,256000553,1914883644,677605665,-1671651015,642759612,-2041779433,-1412447455,-488026182,
-241612925,-15598712,-1545598176,541455136,574015772,-995070157,896085405,-1309932031,-182515936,2095199146,105311520,-1970099447,673687712,1254
782240],"MinimumSimilarity":0.85,"MaxResults":8}
encryptedauctionaggregator_1  | info: EncryptedAuctionAggregator.Middlewares.RequestLoggingMiddleware[0]
encryptedauctionaggregator_1  |       Http Response Information:
encryptedauctionaggregator_1  |       POST [200]: http://localhost:5050/api/Search
encryptedauctionaggregator_1  |       Response Body: []
```

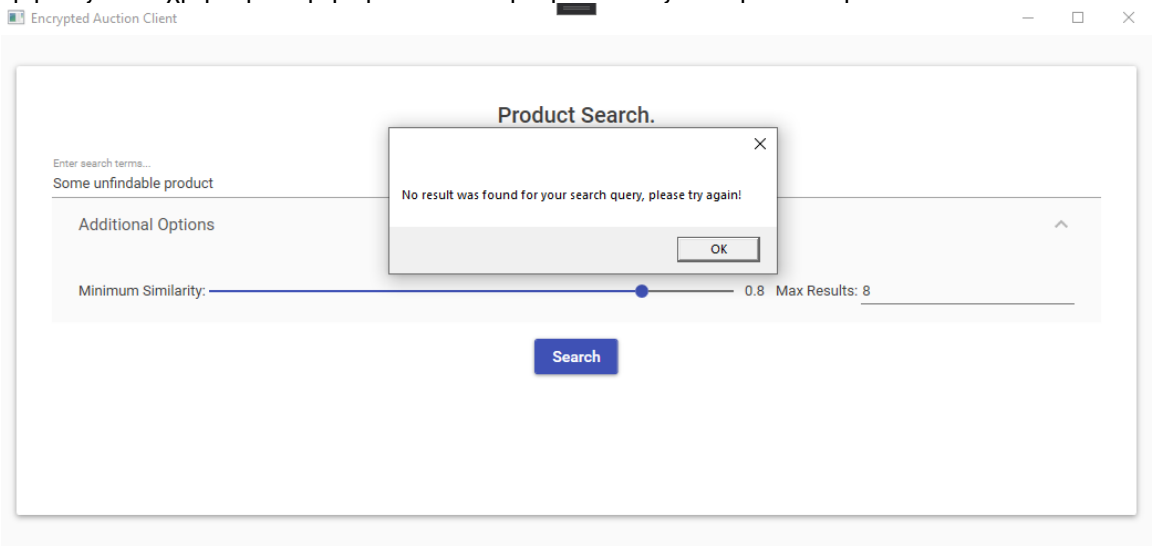

*Εικόνα 27. Καταγραφή κλήσης POST: /api/Search χωρίς αποτελέσματα*

```
encryptedauctionaggregator_1  |       Http Request Information:
encryptedauctionaggregator_1  |       POST: http://localhost:5050/api/Search
encryptedauctionaggregator_1  |       Request Body: {"SearchTerm":[283862402,-542312063,-1829764733,-1012139872,142890400,52165249,-
79564669,1326995462,-866844383,2126434467,568360993,91429295,-1664655359,-183924832,-2104328412,553357312,1297220494,1888425991,-447
421947,-585921887,-1491852798,359276070,-1080064736,-39160918,-1207190110,-1545098233,-2100208253,552643873,370196654,-1313126912,-1
086058453,1253818240,1437232257,1428343846,1122863265,467322883,-226186361,-37555317,194365830,-637232478,-1248660350,869886505,5199
33218,1549378305,1678606082,-622026493,-1722346590,-448212351,-1133636221,-669736799,-2012154327,1000470150,-201690109,1256178849,-2
58300922,880451617,1537641345,-1579329120,-1234226909,-1668871808,-316715257,-533406041,1124833186,336215968,1189205664,-944547167,1
469509537,32687905,1567686275,-1317251839,512116640,356274850,-1553704702,595282049,-652199901,-1118362334,-1523592799,27123843,2560
00553,1914883621,677605664,-1671651040,642759589,-2041779454,-1412447455,-488026205,-241612925,-15598712,-1545598173,541455136,57401
5749,-995070175,896085380,-1309932032,-182515935,2095199146,105311520,-1970099447,673687713,1254782243],"MinimumSimilarity":0.450000
00000000007,"MaxResults":8}
encryptedauctionaggregator_1  | info: EncryptedAuctionAggregator.Middlewares.RequestLoggingMiddleware[0]
encryptedauctionaggregator_1  |       Http Response Information:
encryptedauctionaggregator_1  |       POST [200]: http://localhost:5050/api/Search
encryptedauctionaggregator_1  |       Response Body: [{"product":{"id":"3fab2349-f5b9-4384-97ab-0391c7f3eb19","category":null,"brand
":null,"model":null,"description":null},"store":{"id":"c388b9a1-e6e3-4733-a070-feeca5618816","name":"FirstStore","apiUrl":"http://lo
calhost:5000"},"similarity":0.742},{"product":{"id":"23d9550f-f755-4957-a7c6-c9983be6ceb9","category":null,"brand":null,"model":null
,"description":null},"store":{"id":"96fb9df1-3e8c-42e3-a3dd-7a0181555ec3","name":"SecondStore","apiUrl":"http://localhost:5001"},"si
milarity":0.742},{"product":{"id":"d337efc7-c0cc-453f-9f99-a354de5443ce","category":null,"brand":null,"model":null,"description":nul
l},"store":{"id":"96fb9df1-3e8c-42e3-a3dd-7a0181555ec3","name":"SecondStore","apiUrl":"http://localhost:5001"},"similarity":0.616},{
"product":{"id":"959bb8e5-9cca-4dbc-8b8f-93d421fb7cf0","category":null,"brand":null,"model":null,"description":null},"store":{"id":"
c388b9a1-e6e3-4733-a070-feeca5618816","name":"FirstStore","apiUrl":"http://localhost:5000"},"similarity":0.616},{"product":{"id":"8f
c9a27f-27eb-4b81-a27d-fd2a04dde1ba","category":null,"brand":null,"model":null,"description":null},"store":{"id":"96fb9df1-3e8c-42e3-
a3dd-7a0181555ec3","name":"SecondStore","apiUrl":"http://localhost:5001"},"similarity":0.589},{"product":{"id":"a400e9b2-fa46-4f3b-a
5a3-5c7ecb3f561e","category":null,"brand":null,"model":null,"description":null},"store":{"id":"c388b9a1-e6e3-4733-a070-feeca5618816"
,"name":"FirstStore","apiUrl":"http://localhost:5000"},"similarity":0.589},{"product":{"id":"639080ec-201b-4592-a7f0-6437fb902b2f","
category":null,"brand":null,"model":null,"description":null},"store":{"id":"c388b9a1-e6e3-4733-a070-feeca5618816","name":"FirstStore
","apiUrl":"http://localhost:5000"},"similarity":0.5880000000000001},{"product":{"id":"b36fb98b-bf25-43b6-be5a-8a06fffa669a","catego
ry":null,"brand":null,"model":null,"description":null},"store":{"id":"96fb9df1-3e8c-42e3-a3dd-7a0181555ec3","name":"SecondStore","ap
iUrl":"http://localhost:5001"},"similarity":0.5880000000000001}]
```


*Εικόνα 28. Καταγραφή κλήσης POST: /api/Search με επιστροφή αποτελεσμάτων*

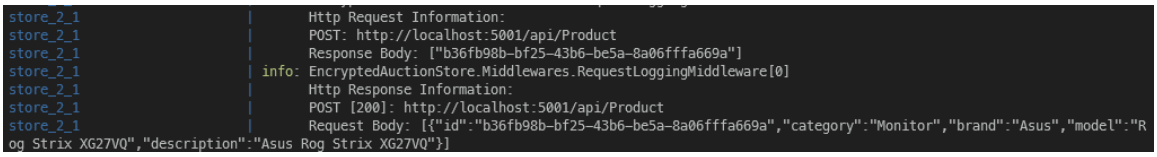
Στο σημείο αυτό υπάρχουν δύο πιθανότητες, είτε η υπηρεσία EncryptedAuctionAggregator δεν βρήκε αποτελέσματα που ταιριάζουν στην αναζήτηση είτε βρήκε και τα επέστρεψε ως απάντηση στο EncryptedAuctionClient. Στην περίπτωση που δεν βρέθηκαν αποτελέσματα η εφαρμογή

εμφανίζει στο χρήστη ένα μήνυμα και τον προτρέπει να ξανά προσπαθήσει.



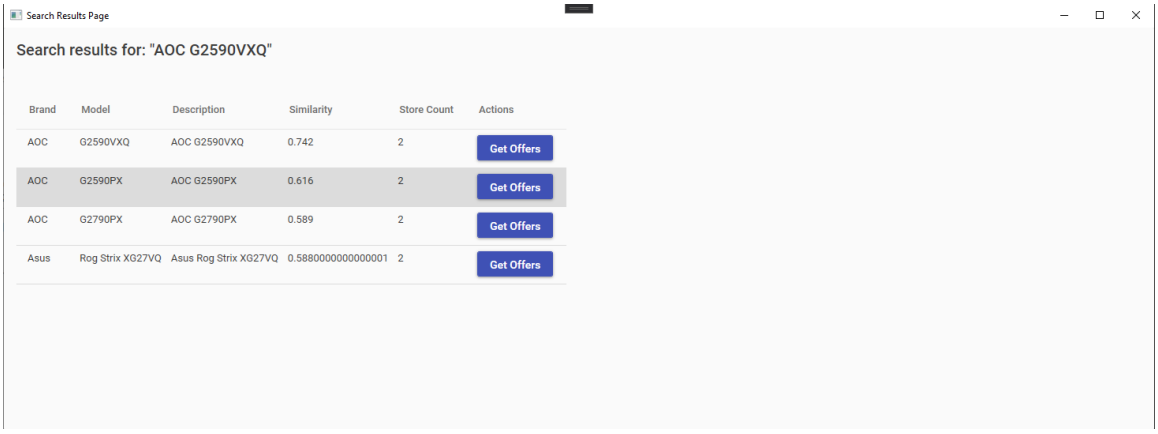

*Εικόνα 29. Μήνυμα αποτυχίας αναζήτησης.*

Σε αντίθετη περίπτωση η εφαρμογή EncryptedAuctionClient εκτελεί παράλληλα την κλήση POST /api/Product προς όλα τα EncryptedAuctionStore των οποίων προϊόντα προτάθηκαν από την αναζήτηση. Οι κλήσεις αυτές γίνονται παράλληλα και συγκεντρώνουν τα στοιχεία των προϊόντων.



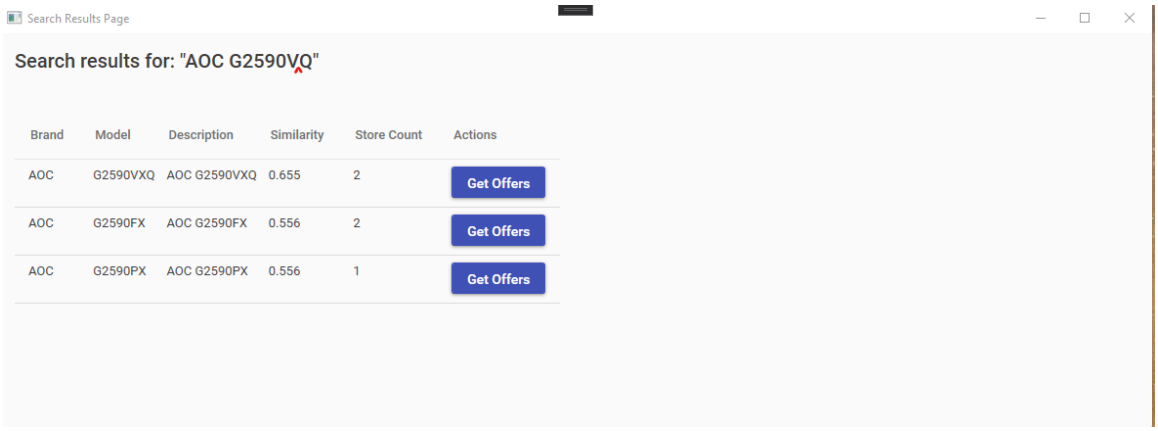

*Εικόνα 30. Καταγραφή κλήσης POST: /api/Product.*

Έπειτα ο χρήστης οδηγείται στην οθόνη αποτελεσμάτων της αναζήτησης.



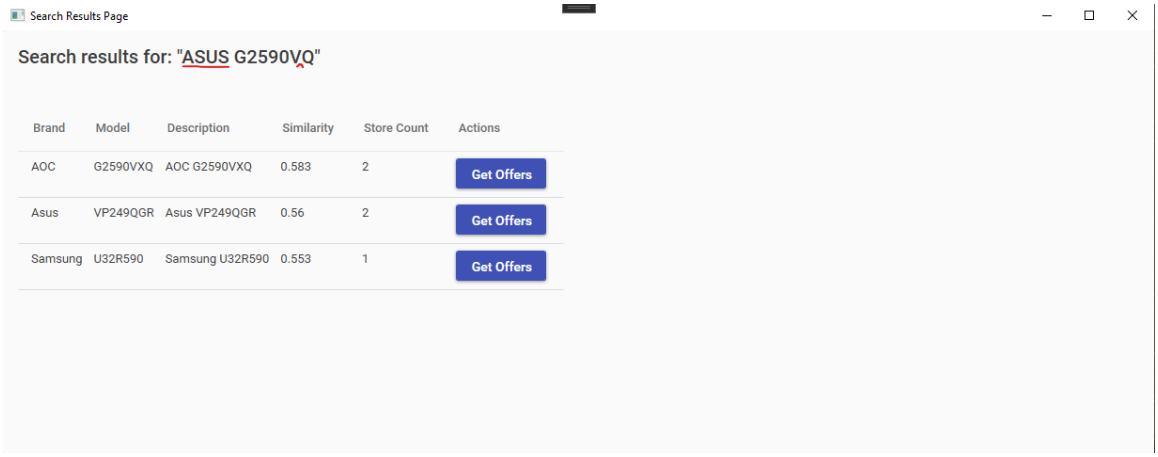

*Εικόνα 31. Οθόνη αποτελεσμάτων ακριβούς ταιριάσματος*

Στο συγκεκριμένο παράδειγμα εκτέλεσης θεωρούμε ότι ο χρήστης αναζητά την οθόνη μάρκας AOC και μοντέλου G2590VXQ. Στο παραπάνω στιγμιότυπο της εκτέλεσης της εφαρμογής φαίνεται ένα ακριβές ταίριασμα αναζήτησης αποτελέσματος (exact match). Ακολουθούν περιπτώσεις στις οποίες παρόλο που ο χρήστης κάνει λάθος στην διατύπωση των όρων αναζήτησης το σύστημα του επιστρέφει το προϊόν που έψαχνε με μεγαλύτερη ομοιότητα από ότι τα υπόλοιπα. Αυτό οφείλεται στα διαφορετικά βάρη που καθορίζονται σε επίπεδο παραμέτρων των προϊόντων και συνυπολογίζονται στον αλγόριθμο αναζήτησης LSH.



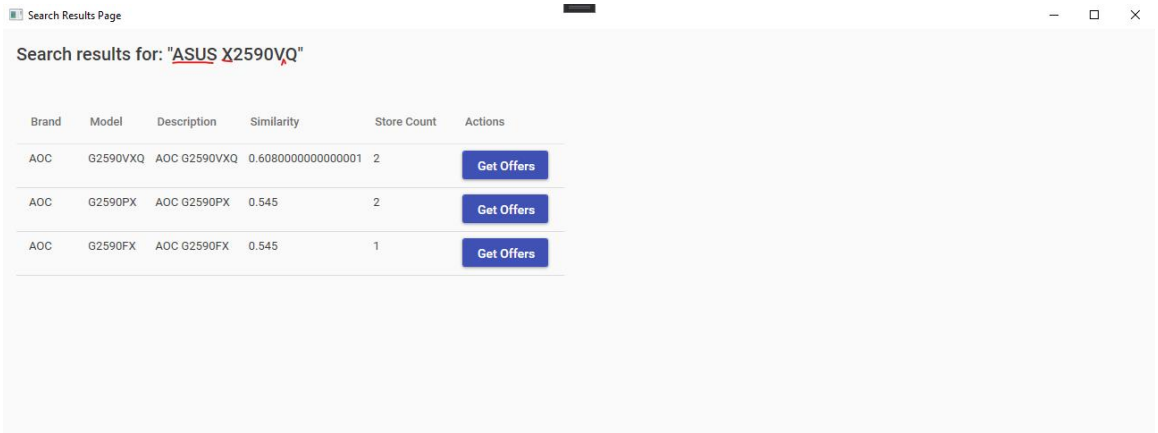

*Εικόνα 32. Οθόνη αποτελεσμάτων μερικούς ταιριάσματος (partial match).*

Στην παραπάνω εικόνα βλέπουμε την επιστροφή σωστού αποτελέσματος σε περίπτωση που ο χρήστης κάνει λάθος ενός χαρακτήρα.




*Εικόνα 33. Οθόνη αποτελεσμάτων μερικούς ταιριάσματος (partial match).*

Στην παραπάνω εικόνα βλέπουμε πως ακόμα και αν ο χρήστης κάνει λάθος στην μάρκα του προϊόντος αλλά και τυπογραφικό λάθος στο όνομα το σύστημα επιστρέφει με σχετικά μεγάλη ομοιότητα το προϊόν που ο χρήστης έψαχνε.




*Εικόνα 34. Οθόνη αποτελεσμάτων μερικούς ταιριάσματος (partial match).*

Στην παραπάνω εικόνα βλέπουμε την δυνατότητα του αλγορίθμου να αντιστέκεται στα λάθη του χρήστη, στην συγκεκριμένη περίπτωση το σύστημα μας δίνει σωστά αποτελέσματα παρόλο που ο χρήστης έκανε 3 λάθη στην αναζήτηση.

Στην οθόνη αυτή ο χρήστης μπορεί να δει τα προϊόντα που του προτείνονται από τον EncryptedAuctionAggregator από το σύνολο των καταστημάτων που συμμετέχουν στο σύστημα.

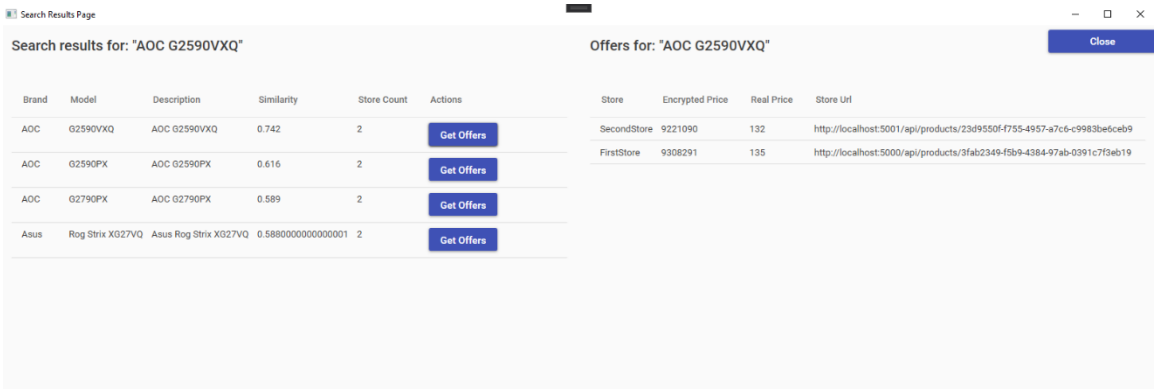
Στον πίνακα εμφανίζονται η μάρκα του προϊόντος, το μοντέλο του προϊόντος, μια σύντομη περιγραφή (που για λόγους ευκολίας είναι η σύμπτυξη της μάρκας και του μοντέλου), η ομοιότητα του προϊόντος με τους όρους αναζήτησης όπως υπολογίστηκε με την χρήση του αλγόριθμου LSH με χρήση των κατακερματισμένων μορφών του προϊόντος και των όρων αναζήτησης και ο αριθμός των καταστημάτων που διαθέτουν το προϊόν. Τα προϊόντα εμφανίζονται με σειρά ομοιότητας, από αυτό που εμφανίζει τη μέγιστη ομοιότητα με την αναζήτηση προς αυτό που εμφανίζει την ελάχιστη. Τέλος δίπλα σε κάθε προϊόν εμφανίζεται ένα κουμπί “Get Offers” με το οποίο ο χρήστης μπορεί να ζητήσει από τα καταστήματα που διαθέτουν το προϊόν να κάνουν προσφορές τιμών για το συγκεκριμένο προϊόν.

Πατώντας ο χρήστης το κουμπί “GetOffers” εκτελούνται παράλληλα κλήσεις /api/Offer προς όλα τα καταστήματα που διαθέτουν το συγκεκριμένο προϊόν.

```
store_2_1                   | info: EncryptedAuctionStore.Middlewares.RequestLoggingMiddleware[0]
store_2_1                   |       Http Request Information:
store_2_1                   |       POST: http://localhost:5001/api/Offer
store_2_1                   |       Response Body: {"Key":"OdLoGaUkMqBbWYbETDGBEOAcpKI7SjQN","Id":"23d9550f-f755-4957-a7c6-c9983be6ceb9"}
store_2_1                   | info: EncryptedAuctionStore.Middlewares.RequestLoggingMiddleware[0]
store_2_1                   |       Http Response Information:
store_2_1                   |       POST [200]: http://localhost:5001/api/Offer
store_2_1                   |       Request Body: 9221090
store_2_1                   | dbug: EncryptedAuctionStore.Middlewares.RequestLoggingMiddleware[0]
store_2_1                   |
store_2_1                   |       Request Body: 9221090
```

*Εικόνα 35. Καταγραφή κλήσης /api/Offer*

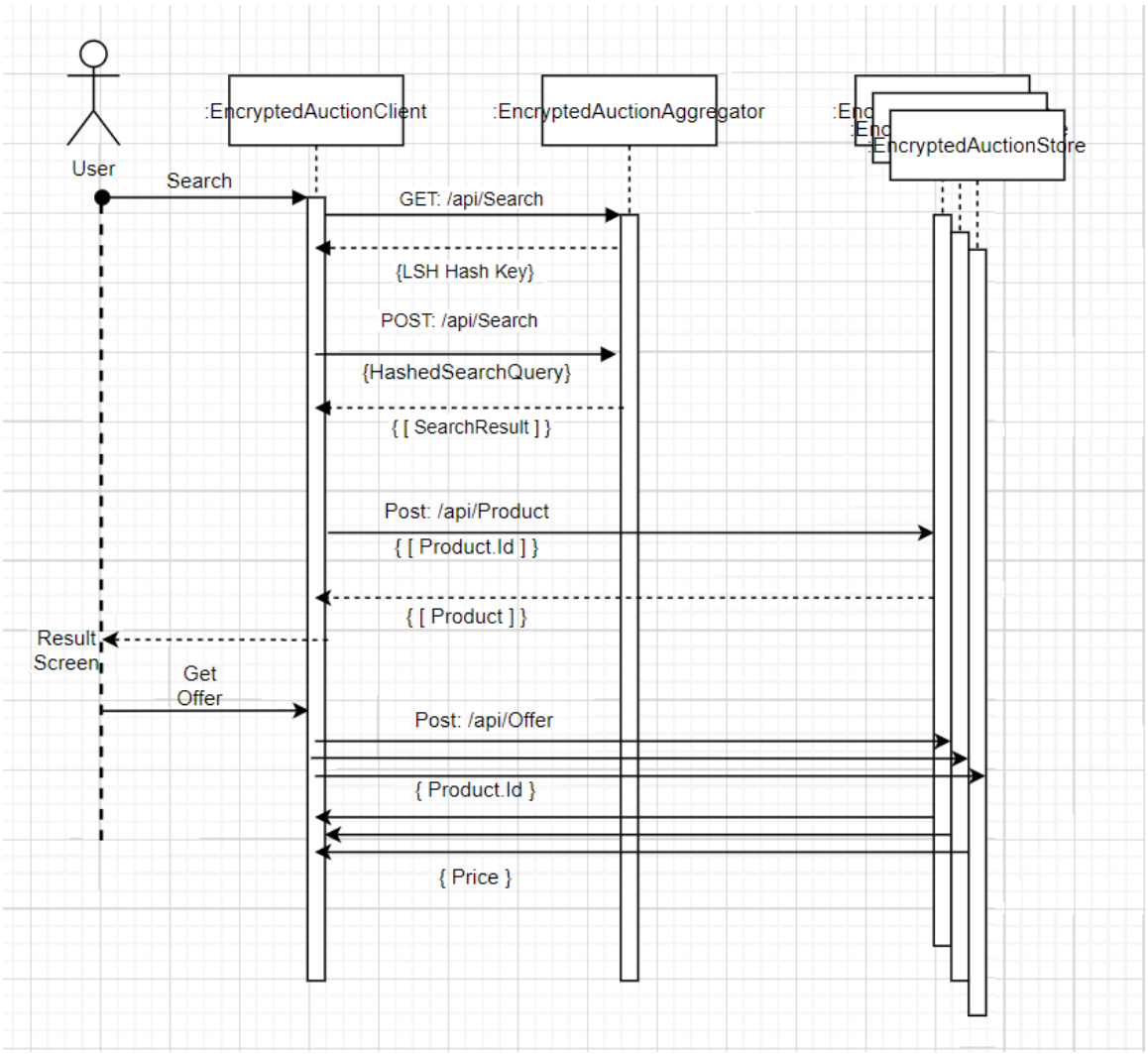
Όταν η εφαρμογή λάβει τις απαντήσεις των κλήσεων εμφανίζει στον χρήστη την οθόνη προσφορών.




*Εικόνα 36 Οθόνη προσφορών.*

Στην οθόνη αυτή εμφανίζονται με σειρά από την χαμηλότερη προς την υψηλότερη τιμή, τα καταστήματα που διαθέτουν το προϊόν. Βλέπουμε ότι στην τελευταία στήλη του πίνακα βρίσκεται ο σύνδεσμος μέσω του οποίου ο πελάτης θα μπορούσε θεωρητικά να προμηθευτεί το προϊόν.

Εδώ τελειώνει ένας συνηθισμένος κύκλος χρήσης της παρούσας υλοποίησης του συστήματος. Το σύνολο της ροής της πληροφορίας όπως περιεγράφηκε παρουσιάζεται στο ακόλουθο διάγραμμα ροής. Στο διάγραμμα εμφανίζονται οι υπηρεσίες και οι κλήσεις που ανταλλάσσουν μεταξύ τους. Με βέλος συνεχούς γραμμής εμφανίζονται οι κλήσεις ενώ κάτω από το βέλος εμφανίζεται η πληροφορία που μεταφέρουν. Με βέλος διακεκομμένης γραμμής εμφανίζονται οι απαντήσεις με το περιεχόμενο της απάντησης επίσης κάτω από το βέλος.




*Εικόνα 37. Διάγραμμα ροής πληροφορίας συστήματος EncryptedAuction κατά τη λειτουργία αναζήτησης προϊόντος*

# 6. Συμπεράσματα και πιθανές επεκτάσεις

## 6.1 Συμπεράσματα

Η παρούσα διατριβή είχε ως στόχο να διερευνήσει την αποτελεσματικότητα του αλγορίθμου LSH σε κεντρικοποιημένα συστήματα αναζήτησης πληροφορίας που αποκρύπτουν την πληροφορία από τον κεντρικό κόμβο. Σημαντικά σημεία που έκριναν την καταλληλόλητα ή μη του αλγορίθμου για συστήματα τέτοιου τύπου παρουσιάζονται παρακάτω:

**1. Απόκρυψη πραγματικής πληροφορίας από τον κεντρικό κόμβο.**

Στην συγκεκριμένη υλοποίηση ο κεντρικός κόμβος EncryptedAuctionAggregator δεν έχει σε καμία φάση της λειτουργίας του πρόσβαση σε πραγματικά, εκμεταλλεύσιμα δεδομένα. Τόσο τα δεδομένα των επιμέρους συστημάτων όσο και οι όροι της αναζήτησης φτάνουν στον κεντρικό κόμβο σε κατακερματισμένη μορφή. Επιπλέον ο αλγόριθμος LSH δεν επιτρέπει την ανακατασκευή των δεδομένων στην αρχική τους μορφή.

**2. Ορθά αποτελέσματα**

Τα αποτελέσματα του συστήματος ανταποκρίνονται στους όρους της αναζήτησης, στις δοκιμές που έγιναν. Επιπλέον χάριν του σταθμισμένου αλγόριθμου LSH η αναζήτηση μπορεί να βελτιστοποιηθεί περεταίρω χωρίς σημαντικές αλλαγές στον κώδικα ή την αρχιτεκτονική του συστήματος.

**3. Επεκτασιμότητα**

Ό αλγόριθμος του LSH εφαρμόστηκε στην παρούσα εργασία για αναζήτηση σε προϊόντα. Τόσο όμως ο αλγόριθμος όσο και η βιβλιοθήκη που τον υλοποιεί εκφράζονται με γενικευμένες δομές δεδομένων και διεπαφές (Interfaces). Έτσι η ίδια αρχιτεκτονική θα μπορούσε να χρησιμοποιηθεί για οποιοδήποτε τύπο δεδομένων με οσοδήποτε πολλά χαρακτηριστικά.

**4. Ελαχιστοποίηση του φόρτου των επιμέρους κόμβων**

Η απαιτητική επεξεργαστικά λειτουργία της αναζήτησης λαμβάνει χώρα μόνο στον κεντρικό κόμβο. Οι επιμέρους υπηρεσίες απαντούν μόνο σε κλήσεις περιορισμένης έκτασης και μόνο εφόσον ένας χρήστης ενδιαφέρεται για κάποιο από τα δεδομένα τους. Αυτό κάνει την αρχιτεκτονική επικερδή για τους επιμέρους κόμβους καθώς με μικρό κόστος μπορούν να συνδεθούν σε μία κεντρική υπηρεσία που παρέχει την αναζήτηση.

## 6.2 Μελλοντικές επεκτάσεις

Η υλοποίηση που ακολουθεί την εργασία αποτελεί μια εφαρμογή για επίδειξη των δυνατοτήτων του αλγορίθμου σέ ένα σύστημα αναζήτησης. Για να καταστεί ένα πλήρως λειτουργικό σύστημα με εμπορικές χρήσεις οι ακόλουθες επεκτάσεις θα έπρεπε να υλοποιηθούν:

- Εξουσιοδότηση και πιστοποίηση χρηστών. (Authorization & Authentication). Σύστημα εγγραφής και σύνδεσης των χρηστών στην εφαρμογή.
- Επεκτασιμότητα του κεντρικού κόμβου. Ο κεντρικός κόμβος θα έπρεπε να μπορεί να αυξάνει και να μειώνει σε υπολογιστικούς πόρους ανάλογα με τις ανάγκες του συστήματος. Την λειτουργία αυτή μπορούμε να την πετύχουμε με χρήση περισσότερων λιγότερων docker container στους οποίους η κίνηση θα δρομολογείται μέσω ενός εξισορροπιστή φορτίου (load balancer).

- Βελτίωση της χρήσης του αλγορίθμου LSH είτε με ενσωμάτωση του στη βάση δεδομένων είτε με σταδιακή ανάγνωση και επεξεργασία των προϊόντων (data stream) είτε με caching.
- Δημιουργία περιβάλλοντος διεπαφής καταστήματος ή ενσωμάτωση της λειτουργίας αγοράς προϊόντος στο api του καταστήματος.
- Διαχειριστικό περιβάλλον καταστήματος για προσθήκη, αφαίρεση και επεξεργασία προϊόντων.
- Βελτίωση και επέκταση της εφαρμογής πελάτη, δημιουργία web και mobile εφαρμογής για τον πελάτη.

## 6.3 Γνώσεις που αποκτήθηκαν

Η παρούσα εργασία συνέβαλλε στην απόκτηση γνώσεων σχετικά τόσο με την υλοποίηση αλγορίθμων όσο και με την ανάπτυξη συστημάτων διαδικτύου. Συγκεκριμένα αποκτήθηκαν γνώσεις σχετικά με τα παρακάτω αντικείμενα:

- Υλοποίηση εφαρμογών με χρήση του προγραμματιστικού περιβάλλοντος .Net Core και των υπολοίπων εργαλείων του οικοσυστήματος (ASP.Net Core, EF Core & WPF)
- Παραμετροποίηση και εκτέλεση εφαρμογών σε περιβάλλον docker container. Οργάνωση και εκτέλεση συστήματος πολλών docker container με χρήση του εργαλείου docker-compose.
- Κατανόηση του αλγορίθμου MinHash και LSH και τροποποίησή τους ώστε να εφαρμοστούν βέλτιστα στο σύστημα που υλοποιήθηκε.

# Βιβλιογραφία

Broder, A., 1997. *On the resemblance and containment of documents.* Salerno, Italy, Italy, IEEE.

Dumitru, B., Matthew, S., Glenn, T. & Vineet, B., 2010. RAPID detection of gene–gene interactions in genome-wide association studies. *Bioinformatics,* 26(22), p. 2856–2862.

Jure, L., Anand, R. & Ullman, J. D., 2010. *Mining of Massive Datasets.* s.l.:s.n.

Konstantin, B. et al., 2015. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nature Biotechnology,* Volume 33, p. 623–630.

Lianyong, Q., Xuyun, Z., Wanchun, D. & Qiang, N., 2017. A Distributed Locality-Sensitive Hashing-Based Approach for Cloud Service Recommendation From Multi-Source Data. *IEEE Journal on Selected Areas in Communications,* pp. 2616 - 2624.

Piotr, I. & Rajeev, M., 1988. *Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality,* s.l.: Stanford University Department of Computer Science.

Yushi, J. & Shumeet, B., 2008. VisualRank: Applying PageRank to. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE,* 30(11), pp. 1877-1890.