ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Δημιουργία εφαρμογής δωρεών

Διπλωματική εργασία

Δημήτριος Χαραλάμπης

Επιβλέπων: Τσανάκας Παναγιώτης
Καθηγητής, ΕΜΠ

Αθήνα, Σεπτέμβριος 2025

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Δημιουργία εφαρμογής δωρεών

Διπλωματική εργασία

Δημήτριος Χαραλάμπης

Επιβλέπων: Τσανάκας Παναγιώτης
Καθηγητής, ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 19η Σεπτεμβρίου 2025.

.....	.....	.....
Π. Τσανάκας	Α. Σταφυλοπάτης	Σ. Ξύδης
Καθηγητής, ΕΜΠ	Ομότιμος Καθηγητής, ΕΜΠ	Επίκουρος Καθηγητής, ΕΜΠ

Αθήνα, Σεπτέμβριος 2025

……
Δημήτριος Χαραλάμπης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

ΔΗΛΩΣΗ ΜΗ ΛΟΓΟΚΛΟΠΗΣ ΚΑΙ ΑΝΑΛΗΨΗΣ ΠΡΟΣΩΠΙΚΗΣ ΕΥΘΥΝΗΣ

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ενυπογράφως ότι είμαι αποκλειστικός συγγραφέας της παρούσας Πτυχιακής Εργασίας, για την ολοκλήρωση της οποίας κάθε βοήθεια είναι πλήρως αναγνωρισμένη και αναφέρεται λεπτομερώς στην εργασία αυτή. Έχω αναφέρει πλήρως και με σαφείς αναφορές, όλες τις πηγές χρήσης δεδομένων, απόψεων, θέσεων και προτάσεων, ιδεών και λεκτικών αναφορών, είτε κατά κυριολεξία είτε βάσει επιστημονικής παράφρασης. Αναλαμβάνω την προσωπική και ατομική ευθύνη ότι σε περίπτωση αποτυχίας στην υλοποίηση των ανωτέρω δηλωθέντων στοιχείων, είμαι υπόλογος έναντι λογοκλοπής, γεγονός που σημαίνει αποτυχία στην Πτυχιακή μου Εργασία και κατά συνέπεια αποτυχία απόκτησης του Τίτλου Σπουδών, πέραν των λοιπών συνεπειών του νόμου περί πνευματικών δικαιωμάτων. Δηλώνω, συνεπώς, ότι αυτή η Πτυχιακή Εργασία προετοιμάστηκε και ολοκληρώθηκε από εμένα προσωπικά και αποκλειστικά και ότι, αναλαμβάνω πλήρως όλες τις συνέπειες του νόμου στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής άλλης πνευματικής ιδιοκτησίας.

στην οικογένεια μου

Περίληψη

Τα τελευταία χρόνια παρατηρείται στη χώρα μας μια προσπάθεια ψηφιακού μετασχηματισμού. Από τις μικρότερες εταιρείες μέχρι τα πιο σημαντικά κομμάτια του κρατικού μηχανισμού, όλα πλέον είναι διαθέσιμα σαν υπηρεσίες στο διαδίκτυο μέσω απλών δικτυακών ή/και mobile εφαρμογών. Τέτοια εγχειρήματα ωστόσο, δεδομένης της απήχησης που έχουν, αποτελούν αφορμή για μελέτη τόσο στην εμπειρία χρήσης, όσο και στη συστημική τους λειτουργία, καθώς τεχνικά πρέπει να χαρακτηρίζονται υψηλή προσφορά, υψηλή ταχύτητα απόκρισης και αξιοπιστία.

Επίσης, ένα πράγμα που παρατηρείται είναι η έμφυτη ανάγκη του πλήθους να παρέχει βοήθεια σε κάθε είδος κλίμακας, όταν το κάλεσμα είναι καθολικό και ο λόγος αντάξιος. Τρανά παραδείγματα αποτελούν εφαρμογές crowdfunding παγκοσμίου φήμης όπως το kick-starter, στις σελίδες του καθενός μπορεί να γίνει με την κατάλληλη διαφήμιση για να λάβει χρηματοδότηση.

Ο σκοπός λοιπόν της παρούσας διπλωματικής είναι η μελέτη και εν τέλει η δημιουργία μιας πλατφόρμας, κατά τις οποίες οι χρήστες (απλοί πολίτες και οργανισμοί) θα μπορούν να αιτηθούν και να συνεισφέρουν αγαθά και υπηρεσίες σε είδος. Θα παρακάμπτεται δηλαδή το στάδιο της απευθείας χρηματοδότησης, το οποίο κατά μια άποψη προσθέτει ένα επίπεδο αδιαφάνειας καθώς δεν είναι σίγουρο αν η χρηματοδότηση καταλήγει να εφεξυπηρετεί ποτέ τον αναγγελμένο σκοπό.

Λέξεις κλειδιά

web app, full-stack, microservices, data, Kafka, Go, Elasticsearch, Redis

Abstract

In recent years, our country has been witnessing an effort at digital transformation. From the smallest companies to the most important parts of the state apparatus, everything is now available as services on the internet through simple web and/or mobile applications. However, such projects, given their appeal, are a reason for study in both the user experience and their systemic operation, as technically they must be characterized by high offer, high response speed and reliability.

Also, one thing that is observed is the innate need of the crowd to provide help on any kind of scale, when the call is universal and the reason is worthy. Great examples are world-famous crowdfunding applications such as kick-starter, on everyone's pages it can be done with the appropriate advertising to receive funding.

The purpose of this thesis is therefore to study and ultimately create a platform where users (ordinary citizens and organizations) will be able to request and contribute goods and services in kind. This will bypass the stage of direct funding, which in one sense adds a level of opacity as it is not certain whether the funding will ever serve its stated purpose.

Λέξεις κλειδιά

web app, full-stack, microservices, data, Kafka, Go, Elasticsearch, Redis

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου Παναγιώτη Τσανάκα για την εμπιστοσύνη που μου έδειξε στην ανάθεση αυτής της διπλωματικής. Επίσης, θα ήθελα να ευχαριστήσω τον επιστημονικό συνεργάτη κ. Βρεττό Μουλό για την αδιάκοπη παροχή βοήθειας που μου παρείχε καθόλη τη διάρκεια της εκπόνησης αυτής της διπλωματικής, στον οποίο εύχομαι να βρει λίγο χρόνο να κοιμηθεί. Ακόμα, θα ήθελα να ευχαριστήσω την οικογένεια μου, την κοπέλα μου, τις φίλες και τους φίλους μου, οι οποίοι με υποστήριξαν και με υποστηρίζουν καθημερινά. Τέλος εύχομαι κάθε καλό στους καθηγητές και τους συμφοιτητές μου και τους αποχαιρετώ λέγοντας εις το επανιδείν.

Δημήτριος Χαραλάμπης

Αθήνα, Σεπτέμβριος 2025

Περιεχόμενα

Λίστα εικόνων

Λίστα γραφημάτων

Λίστα στιγμιοτύπων

Λίστα με κώδικες

Κεφάλαιο 1

Εισαγωγή

Οι δικτυακές υπηρεσίες έχουν εξελιχθεί σε αναπόσπαστο κομμάτι της καθημερινότητάς μας. Αποτελούν πλέον μέρος της παγκόσμιας προσπάθειας μπροστά σε αυτό που ονομάζεται εξέλιξη, όσον αφορά την παροχή αυτών, υπό την έννοια ότι πλέον σε άμεσο χρόνο, μπορεί κανείς αξιόπιστα να κάνει τη δουλειά του, ακόμα και από το κινητό του. Ψηφιακές λύσεις σε διάφορους τομείς βλέπουμε ακόμα και στη χώρα μας έντονα τα τελευταία χρόνια, είτε αυτές είναι δημοσίου συμφέροντος, είτε όχι. Καλούνται να σηκώσουν το «αφόρητο βάρος» της ατελείωτης γραφειοκρατίας, χωρίς να κάνουν εκπτώσεις στην ασφάλεια, στη φερεγγυότητα και στη συνέπεια των συναλλαγών που συμβαίνουν.

Τέτοιες υπηρεσίες (και κατά συνέπεια εφαρμογές) καλούνται συχνά να παράσχουν λύσεις σε προβλήματα μαζικής κλίμακας. Ανεξάρτητα από το πρακτικό κομμάτι, τα μεγάλα προβλήματα και οι μεγάλες ανάγκες απαιτούν συλλογική αντιμετώπιση. Ψηφιακές λύσεις, λοιπόν, έχουν εμφανιστεί και στο κομμάτι της έμπρακτης παροχής βοήθειας υπό τη μορφή άντλησης κεφαλαίου (crowdfunding - πχ. kickstarter), αλλά και παροχής υπηρεσιών επί πληρωμή (fiverr). Και οι δύο αυτές κυρίαρχες πλατφόρμες, ωστόσο, έχουν κερδοσκοπικό χαρακτήρα. Δεν υπάρχει, δηλαδή, κάποιο ολοκληρωμένο σύστημα το οποίο να παρέχει επιλογές για δωρεές αγαθών ή/και υπηρεσιών.

Στην υλοποίηση μιας τέτοιας λύσης, είναι σαφές ότι θα πρέπει κανείς να λάβει υπόψη πολλές παραμέτρους, οι οποίες έχουν κυρίως να κάνουν με τη φύση και τις κοινωνικές διαστάσεις τις οποίες έχει μια δωρεά. Οι πολίτες/οργανισμοί που θα συμμετείχαν σε κάτι τέτοιο, θα ήθελαν ιδανικά να υπάρχει διαφάνεια κατά τη δωρεά. Επίσης, πέρα από την καθαυτή δωρεά, διαφάνεια θα πρέπει να υπάρχει μεταξύ όλων των μερών που λαμβάνουν μέρος σε αυτή, δηλαδή των αιτούντων και των δωρητών. Ειδικά σε περιπτώσεις ανάγκης κατά τις οποίες η συναλλαγή είναι μεγάλου ενδιαφέροντος (πχ. δημόσια έκκληση για βοήθεια από κρατικούς ή δημοτικούς φορείς), η διαφάνεια τέτοιων συναλλαγών είναι δικαίωμα των πολιτών και μπορεί ακόμα να παρέχει επιπλέον κίνητρα συμμετοχής στη δράση. Ακόμα, είναι σημαντική όχι μόνο επειδή αυξάνει την αποτελεσματικότητα στην ενημέρωση, αλλά και επειδή μπορεί να βοηθήσει στο να διασφαλιστεί ότι τα οφέλη της ανάπτυξης και της παροχής βοήθειας καταμερίζονται και δεν αξιοποιούνται μόνο από κάποια ελίτ [1].

Κάτι ακόμα που καθιστά απαραίτητη μια τέτοια λύση είναι το γεγονός ότι για τους μικρούς οργανισμούς είναι πολύ δύσκολο να προσεγγίσουν το ενδιαφέρον και αυτό συμβαίνει για διάφορους λόγους. Αρχικά, οι πιο γνωστοί οργανισμοί έχουν καθιερωθεί έχοντας ήδη ιστορικό αποδεδειγμένων ενεργειών και δράσεων. Αυτό αυξάνει την αξιοπιστία τους απέναντι σε κάποιον ο οποίος προτίθεται να βοηθήσει. Έχουν δηλαδή brand name. Επίσης, οι μεγάλοι οργανισμοί, λόγω της διαχείρισης μεγαλύτερου κεφαλαίου, διαθέτουν περισσότερους πόρους και δυναμικό σε διαφήμιση και δημόσιες σχέσεις, τα οποία με τη σειρά τους θα φέρουν εκ νέου κεφάλαιο. Υπάρχει δηλαδή αυτή η ατέρμονη επανάληψη όσον αφορά την ορατότητα ενός οργανισμού. Τέλος, οι μεγάλοι οργανισμοί είναι πιο πιθανό να λάβουν δωρεές από πολίτες ή άλλα ιδρύματα μεγάλης κοινωνικής επιρροής. Τέτοιες συναλλαγές είναι απαραίτητες καθώς

αποτελούν ένα είδος κοινωνικής σφραγίδας και επικύρωσης, προσελκύοντας με αυτό τον τρόπο νέους εν δυνάμει δωρητές.

Αντικείμενο της διπλωματικής

Η παρακάτω διπλωματική αποτελεί μια προσέγγιση στη δημιουργία εφαρμογής, όπου χρήστες και οργανισμοί θα μπορούν να αιτούνται και να προσφέρουν αγαθά και υπηρεσίες. Γενικά, η δημιουργία μιας εφαρμογής ευρείας κλίμακας, η οποία διατίθεται προς διανομή, αποτελεί ένα πολύπλοκο εγχείρημα με πολλά παρακλάδια όσον αφορά τη σχεδίαση του συστήματος, την οπτική σχεδίαση της εφαρμογής, τις «επιχειρησιακές» λειτουργίες κλπ. Κατά την ανάπτυξή της, προέκυψαν προκλήσεις σε διάφορα στάδια. Η επιλογή των τεχνολογιών έγινε λαμβάνοντας υπόψη διάφορα κριτήρια, εκ των οποίων τα σημαντικότερα ήταν τα εξής:

- **Ακεραιότητα δεδομένων (Data integrity)**
  Με τον όρο ακεραιότητα δεδομένων, αναφερόμαστε στις έννοιες και τις διαδικασίες σχετικές με τη διατήρηση δεδομένων, τη διασφάλιση της ακρίβειας (data accuracy) και της συνέπειάς (data consistency) τους, καθόλη τη διάρκεια ζωής τους κατά τη χρήση τους σε ένα υπολογιστικό σύστημα. Στο σύνολό του, ο όρος περιλαμβάνει και τη φυσική ακεραιότητα των δεδομένων (physical integrity), η οποία αφορά την αποθήκευση και ανάκτηση των δεδομένων σε υλικό επίπεδο, κάτι που υπερβαίνει τα όρια της διπλωματικής αυτής.

- **Ευκολία στη χρήση (Usability)**
  Ένας από τους στόχους μιας τέτοιας εφαρμογής είναι η υψηλή υιοθέτηση από μεγάλο μέρος του κοινωνικού συνόλου. Αυτό καθιστά επιτακτική την ανάγκη για μια εύχρηστη εφαρμογή ακόμα και από άτομα μη εξοικειωμένα πλήρως με τις σύγχρονες τεχνολογίες. Άλλωστε, σε έρευνες έχει παρατηρηθεί χάσμα γενεών όσον αφορά τη φιλανθρωπική προσφορά και συγκεκριμένα στο Ηνωμένο Βασίλειο, περισσότερες από τις μισές δωρεές προέρχονται από άτομα μεγαλύτερα των 60 ετών [2]. Μπορεί λοιπόν εύκολα να εξαχθεί το συμπέρασμα ότι μια εύχρηστη εφαρμογή θα προσελκύσει περισσότερο κόσμο.

- **Συντήρηση και εξέλιξη (Maintenance)**
  Σε μεγάλα έργα λογισμικού είναι απαραίτητο να λάβει υπόψη κανείς τη λεγόμενη Developer Experience - DX ή αλλιώς «Εμπειρία Προγραμματιστή΄». Αυτό επιτυγχάνεται χρησιμοποιώντας τα κατάλληλα εργαλεία κατά την ανάπτυξη, αλλά και με προσεκτική επιλογή των τεχνολογιών που αξιοποιούνται. Μια τεχνολογία η οποία έχει χαμηλή υποστήριξη από την προγραμματιστική κοινότητα είναι πιο επιρρεπής σε σφάλματα από κάποια άλλη η οποία χαίρει εκτίμησης και ανανεώνεται συνεχώς. Αυτό το δεύτερο σκέλος είναι άμεσα εφαρμόσιμο σε αυτή τη διπλωματική, δεδομένου ότι η ανάπτυξη μέχρι τώρα έχει γίνει από ένα άτομο. Επομένως, οι τεχνολογίες που επιλέχθηκαν αποτελούν μεγάλα έργα ανοικτού κώδικα, τα οποία ήταν ενεργά κατά την περίοδο της συγγραφής. Αυτό πέρα από την ευκολότερη αποσφαλματοποίηση (δεδομένης της μεγάλης ενεργής κοινότητας), εγγυάται και μεγαλύτερη διάρκεια ζωής του έργου.

Κεφάλαιο 2

Θεωρητικό υπόβαθρο

Στο κεφάλαιο αυτό παρουσιάζονται αναλυτικά οι βασικές τεχνολογικές ενότητες, οι οποίες απαντώνται εν γένει σε οποιαδήποτε δικτυακή εφαρμογή, το παρασκήνιο (Back-End), το προσκήνιο (Front-End) και η αποθήκευση δεδομένων (Data Storage).

Αποθηκευτικός χώρος και βάσεις δεδομένων

Φύση των δεδομένων

Σήμερα, με τη ραγδαία αύξηση εφαρμογών τεχνητής νοημοσύνης, καταλαβαίνει κανείς εύκολα ότι τα δεδομένα και ο τρόπος αποθήκευσής τους, είτε υλικά είτε εννοιολογικά, αποτελούν σημαντικό κομμάτι της επιστήμης μας. Οντότητες οι οποίες σχετίζονται άμεσα με το επιχειρησιακό κομμάτι μιας εφαρμογής, χωρικά δεδομένα ή δεδομένα χρονοσειρών είναι μόνο μερικοί από τους τύπους δεδομένων οι οποίοι αξιοποιούνται κατά την ανάπτυξη μιας εφαρμογής.

Πρακτικά, πλέον για διαφορετικούς τύπους δεδομένων έχουν αναπτυχθεί διαφορετικές τεχνολογίες και βάσεις δεδομένων και αυτό βγάζει απόλυτο νόημα αν σκεφτούμε ότι κάποιος θα διαβάζει δεδομένα μιας χρονοσειράς κυρίως με γραμμικό τρόπο και ίσως την φιλτράρει, χρονικά προφανώς, διαλέγοντας εγγραφές πριν ή μετά από κάποια χρονική στιγμή. Σαφώς, κάθε εγγραφή της χρονοσειράς θα έχει κάποιο αναγνωριστικό (ID), εννοιολογικά αυτό μπορεί να αποτελείται και από τη χρονοσφραγίδα την ίδια (timestamp). Σε δεδομένα οντοτήτων, από την άλλη, υπάρχουν διαφορετικές προσεγγίσεις ανάλογα με την επιθυμία ύπαρξης σχεσιακών δομών ή όχι. Έτσι η κοινότητα έχει καταφύγει σε 2 κύριες κατηγορίες βάσεων δεδομένων:

Σχεσιακές βάσεις δεδομένων (Relational Databases)

Στις σχεσιακές βάσεις δεδομένων, ο προγραμματιστής μπορεί να ορίσει οντότητες με χαρακτηριστικά και σχέσεις μεταξύ των οντοτήτων. Οι σχέσεις μπορεί να έχουν και αυτές τα δικά τους χαρακτηριστικά. Κάθε οντότητα ορίζεται από έναν πίνακα όπου τα χαρακτηριστικά της αποτελούν τις στήλες του πίνακα, ενώ κάθε γραμμή είναι μία εγγραφή. Κάθε εγγραφή σε έναν τέτοιο πίνακα έχει ένα μοναδικό αναγνωριστικό, το οποίο στις οντότητες προκύπτει από μία γεννήτρια τυχαίων συμβολοσειρών ή είναι αριθμός ο οποίος αυξάνεται σειριακά από τη μηχανή της βάσης δεδομένων. Οι εγγραφές στους πίνακες των σχέσεων συνήθως περιέχουν ως χαρακτηριστικά αναγνωριστικά οντοτήτων, των οποίων ο συνδυασμός αποτελεί το αναγνωριστικό της εκάστοτε εγγραφής, αν αυτό δεν ορίζεται με τον παραπάνω τρόπο που αναλύθηκε για τις οντότητες. Οι σχεσιακές βάσεις δεδομένων καθιερώθηκαν στην αγορά καθώς μοντελοποιούν σύνθετες σχέσεις μεταξύ οντοτήτων και έτσι μπορούσαν να σταθούν

άξια απέναντι στις αυστηρές επιχειρησιακές απαιτήσεις των εταιρειών που τις υιοθέτησαν εξαρχής.

Μη σχεσιακές βάσεις δεδομένων (Non Relational Databases, NoSQL)

Οι μη σχεσιακές βάσεις αποκλίνουν από το άνωθι μοντέλο των σχεσιακών ως προς τη δομή των δεδομένων.Η ονοματοδοσία ιστορικά προήλθε από το γεγονός ότι αυτές οι βάσεις δε χρησιμοποιούσαν SQL (**S**tructured **Q**uery **L**anguage), δηλαδή ένα είδος γλώσσας το οποίο είχε πρωτοεμφανιστεί για να συνάδει με βάσεις του σχεσιακού μοντέλου, εξού και το No SQL. Στην πορεία ωστόσο, δεδομένου ότι τα ερωτήματα SQL είχαν καθιερωθεί και διευκολύνουν την αλληλεπίδραση με τις βάσεις, πολλές NoSQL βάσεις άρχισαν να υλοποιούν μια SQL διεπαφή με σκοπό την διευκόλυνση των προγραμματιστών. Πλέον το NoSQL έχει αποκτήσει την ερμηνεία “**N**ot **o**nly **SQL**”.

Οι βασικές υποκατηγορίες των NoSQL βάσεων δεδομένων είναι οι εξής:

- **Βάσεις δεδομένων κλειδιού - τιμής (Key-value storage databases)**
  Αποτελούν την πιο απλή έκδοση των μη σχεσιακών βάσεων. Κάθε εγγραφή διαθέτει ένα μοναδικό αναγνωριστικό. Δεδομένου αυτού και ότι συνήθως υποστηρίζουν τις βασικές εντολές (εγγραφή, πρόσβαση, διαγραφή), προσφέρουν μεγάλες ταχύτητες και συνήθως υλοποιούν λειτουργίες όπως η κατάτμηση (sharding) βάσει κλειδιού, γεγονός που τις κάνει ικανές να κατανεμηθούν σε πολλαπλούς κόμβους ενός δικτύου υπολογιστών. Η κατάτμηση είναι επί της ουσίας η κατανομή των δεδομένων σε μικρότερα σύνολο βάσει ενός κοινού χαρακτηριστικού τους.

- **Βάσεις εγγράφων (document databases)**
  Η πιο γνωστή κατηγορία μη σχεσιακών βάσεων είναι αυτή των βάσεων εγγραφών. Τα δεδομένα μπορεί να έχουν αυστηρή δομή, αλλά όχι απαραίτητα αυτή των πινάκων που αναλύθηκε παραπάνω. Η έννοια του πίνακα, αλλά ταυτίζεται μόνο λεξικά σε σύγκριση με τις σχεσιακές βάσεις. Πρακτικά οι πίνακες είναι μια συλλογή εγγράφων και χρησιμεύουν μόνο σε μια αδρή κατηγοριοποίηση των εγγράφων. Τα δεδομένα, λοιπόν, που υπόκεινται σε μια συλλογή, δύνανται να διαφέρουν δομικά και να σχετίζονται μεταξύ τους μόνο υπό εννοιολογικές περιστάσεις και άρα η διαχείρισή τους να έγκειται αποκλειστικά στον προγραμματιστή.
  Η χρήση τους αυξήθηκε με την άνοδο της ανάπτυξης των διαδικτυακών εφαρμογών, κυρίως λόγω δύο προτύπων ανταλλαγής δεδομένων, του JSON και του XML.

- **Βάσεις δεδομένων γράφων (Graph databases)**
  Στις σχεσιακές βάσεις δεδομένων έχουμε οντότητες που συσχετίζονται μεταξύ τους και αυτό πράγματι θα μπορούσε να αναπαρασταθεί με έναν γράφο, έχοντας τις οντότητες ως κόμβους και τις σχέσεις ως ακμές. Η ανάγκη όμως για βάσεις δεδομένων γράφων προέρχεται από την ανάγκη για υποστήριξη πολλών πολύπλοκων σχέσεων μεταξύ των οντοτήτων. Οι σχεσιακές βάσεις δεδομένων επενεργούν σε ένα πλήθος δεδομένων με ορισμένη δομή στις οντότητες και στις σχέσεις μεταξύ αυτών. Όταν υπάρχουν σχέσεις με ετερόκλητα χαρακτηριστικά, θα ήταν αφελές να δημιουργείται ένας πίνακας με αυτά για κάθε τέτοια περίπτωση που να αντιπροσωπεύει μικρό πλήθος σχέσεων. Οι βάσεις δεδομένων γράφων λύνουν αυτό το πρόβλημα

αποθηκεύοντας τις οντότητες σαν κόμβους και τις πολύπλοκες σχέσεις σαν ακμές μεταξύ των κόμβων.

Βάσεις δεδομένων και αποθήκευση

Πέρα από τις κατηγορίες αναπαράστασης των δεδομένων, οι οποίες αποτελούν από μόνες τους έναν τρόπο διάκρισης των βάσεων, άλλος ένας τρόπος διάκρισης είναι η τοποθεσία των δεδομένων κατά τη λειτουργία ενός συστήματος διαχείρισης βάσης δεδομένων. Ένα πρόγραμμα κατά την εκτέλεσή του αξιοποιεί τη μνήμη RAM για αποθήκευση δεδομένων προσωρινά, ενώ λέμε ότι αξιοποιεί το δίσκο όταν εγγράφει δεδομένα σε μνήμη μόνιμης κατάστασης. Η διαφορά των δύο, η τοποθεσία των δεδομένων κατά την πρόσβασή τους, δημιουργεί δύο διαφορετικές συνθήκες. Για τη μνήμη RAM υπάρχει η ανάγκη να είναι γρήγορη δεδομένου ότι αξιοποιείται από τον επεξεργαστή πολλαπλές φορές κατά την εκτέλεση ενός προγράμματος. Η μέση ταχύτητα μιας DDR5 μνήμης RAM (η οποίο τείνει να γίνει το standard για εμπορικούς και μη υπολογιστές) είναι 4-10 φορές μεγαλύτερη από έναν κοινό δίσκο στερεάς κατάστασης (**S**olid **S**tate **D**rive). Η διαφορά του throughput είναι μια τάξη μεγέθους σε GB/s. Ο δίσκος από την άλλη δεν απαιτεί το ίδιο γρήγορες ταχύτητες, αλλά η αξιοπιστία του βασίζεται στη μακρόχρονη αποθήκευση των δεδομένων.

Από τις διαφορές αυτές έχει γεννηθεί μια νεά κατηγορία βάσεων δεδομένων όσον αφορά το χρόνο ζωής των δεδομένων στη μνήμη. Αυτές είναι οι **in-memory βάσεις δεδομένων (I**n-**M**emory **D**ata**B**ases**),** των οποίων τα δεδομένα ζουν αυστηρά στη μνήμη RAM κατά την εκτέλεση και έτσι επιτυγχάνονται μεγαλύτερες ταχύτητες . Οι συμβατικές βάσεις δεδομένων μπορεί να υλοποιούν διάφορους μηχανισμούς κατά τους οποίους τα δεδομένα τους δεν εγγράφονται απευθείας στο δίσκο μετά την εκτέλεση μιας εντολής, ωστόσο αυτό δε θα έπρεπε να τις συγκαταλέγει σε in-memory βάσεις, καθώς γίνεται καθαρά για λόγους απόδοσης [3].

Παρακάτω βλέπουμε μια από τις πρώτες προσπάθειες που έγιναν για ανάλυση απόδοσης μεταξύ μιας In-Memory βάσης δεδομένων και μιας συμβατικής που αξιοποιεί το δίσκο [4].

	IMDB (eXtremeDB)	Disk Bound (db.linux)
Read	1	16.25
Write	2.6	3118.25

Table 1: Σύγκριση in-memory και disk db's

DB Integration (Raw SQL, ORM’s)

Η αλληλεπίδραση και η συνδιαλλαγή των δεδομένων μεταξύ εφαρμογής και βάσης δεδομένων (αποτελεί κρίσιμο ζήτημα στη σύγχρονη ανάπτυξη λογισμικού, καθώς καθορίζει τον τρόπο με τον οποίο οι εφαρμογές αλληλεπιδρούν με το αποθηκευμένο πληροφοριακό υλικό. Οι διαφορετικές προσεγγίσεις που χρησιμοποιούνται έχουν άμεσο αντίκτυπο στην αποδοτικότητα, την επεκτασιμότητα, αλλά και στην ικανότητα συντήρησης του κώδικα.

Η πιο άμεση μορφή αλληλεπίδρασης με μια βάση είναι μέσω Raw SQL, δηλαδή με την απευθείας χρήση εντολών της βάσης με τη μορφή κειμένου. Η προσέγγιση αυτή παρέχει

απόλυτο έλεγχο και βέλτιστη απόδοση, αφού ο χρήστης αλληλεπιδρά άμεσα με τη μηχανή της βάσης, αλλά ταυτόχρονα μπορεί να αυξήσει την πολυπλοκότητα του κώδικα και να τον κάνει πιο δύσκολο στη συντήρηση. Επιπλέον, η ενσωμάτωση ωμής SQL συχνά ενέχει τον κίνδυνο λαθών ασφάλειας, όπως το SQL injection, αν δεν χρησιμοποιούνται κατάλληλες τεχνικές προστασίας (π.χ. prepared statements).

Μια πιο σύγχρονη εναλλακτική είναι τα ORM's, ή Object-Relational Mappers (ORMs), τα οποία προσφέρουν μια πιο αφαιρετική προσέγγιση. Μέσω των ORM, τα δεδομένα της βάσης αναπαρίστανται ως αντικείμενα στην εκάστοτε γλώσσα προγραμματισμού, γεγονός που απλοποιεί την ανάπτυξη και καθιστά τον κώδικα πιο ευανάγνωστο και επαναχρησιμοποιήσιμο. Παρότι τα ORM μπορούν να μειώσουν τον χρόνο ανάπτυξης, ενδέχεται να εισάγουν μειονεκτήματα σε επίπεδο απόδοσης, καθώς η γενικευμένη παραγωγή SQL ερωτημάτων δεν είναι πάντα βελτιστοποιημένη. Εδώ προκύπτει συχνά το δίλημμα μεταξύ παραγωγικότητας και απόδοσης.

Τα ORM's εισάγουν ένα «φόρο» στην απόδοση μιας εφαρμογής καθώς απαιτείται συνεχώς η μετατροπή από raw data εξερχόμενα της βάσης σε αντικείμενο που υπόκειται στο ORM. Επίσης συνήθως εισάγουν προγραμματιστική πολυπλοκότητα που εμποδίζει την ευελιξία που παρέχει η απευθείας χρήση εντολών της γλώσσας μιας βάσης. Για αυτούς τους λόγου, δεν έγινε χρήση τους.

Τέλος, η επιλογή στρατηγικής ενσωμάτωσης εξαρτάται σε μεγάλο βαθμό από τις απαιτήσεις της εφαρμογής. Έργα με αυξημένες απαιτήσεις απόδοσης συχνά συνδυάζουν Raw SQL για τα κρίσιμα ερωτήματα και ORM για τη γενική λειτουργικότητα. Από την άλλη πλευρά, συστήματα με έμφαση στην ταχεία ανάπτυξη και ευελιξία βασίζονται σε ORM, δίνοντας έμφαση στη συντηρησιμότητα και στην ταχύτητα ανάπτυξης.

Αναζήτηση σε βάση δεδομένων

Στις βάσεις δεδομένων καλείται κανείς πολύ συχνά να αποθηκεύσει ανθρώπινο κείμενο και ακόμα πιο συχνά επιχειρεί να το αναζητήσει. Αν δε λάβει υπόψη προβλήματα που μπορεί να προκύψουν από την κωδικοποίηση διαφόρων χαρακτήρων από διαφορετικές γλώσσες και τυχόν σύμβολα, τότε παρουσιάζεται απλά το πρόβλημα εύρεσης του όρου αναζήτησης σε μια θάλασσα από χαρακτήρες. Για αυτό το πρόβλημα έχουν αναπτυχθεί διάφορες τεχνικές με τελικό σκοπό τη γρήγορη και αποδοτική αναζήτηση.

Σχηματισμός λεκτικών μονάδων (Tokenization)

Κατά την επεξεργασία κειμένου, πρέπει να υπάρξει μια μέθοδος η οποία θα μετατρέπει το κείμενο, το οποίο είναι για τον υπολογιστή ένα άτακτο σύνολο από χαρακτήρες, σε μια μορφή από την οποία θα μπορεί να εξάγει μοτίβα. Απαιτείται δηλαδή μια συστηματική μέθοδος η οποία θα είναι συνεπής και τα αποτελέσματά της θα είναι χρήσιμα. Οι πιο συνήθεις μέθοδοι είναι οι εξής:

Κατά χαρακτήρα
Το κείμενο χωρίζεται ανά χαρακτήρα. Αυτό μπορεί να φανεί πιο χρήσιμο σε φωνητικά μοντέλα, δεδομένου ότι οι φθόγγοι είναι περιορισμένοι.

Κατά λέξη
Αρκετά χρήσιμο και σύνηθες καθώς οι λέξεις αποτελούν το θεμέλιο λίθο του λόγου. Αξιοποιείται και η μέθοδος stemming (εκ του stem ή ρίζα), κατά την οποία εξάγεται ένα μέρος της λέξης, δεδομένου ότι σε όλα τα κλιτά μέρη του λόγου επηρεάζονται οι καταλήξεις. Ακόμα, κάποιες λέξεις αποτελούνται από άλλες και επηρεάζονται το (γραμματικό) χρόνο και τη (γραμματική) φωνή. Έτσι, τυχόν προθέματα και επιθέματα έχουν τη δικιά τους βαρύτητα.

Οι μονάδες που θα προκύψουν αποτελούν το κύριο υλικό για τη δεικτοδότηση σε ένα κείμενο και συνήθως παραλείπονται συνδετικοί σύνδεσμοι, σημεία στίξης, επιφωνήματα και άλλα.

Ευθεία δεικτοδότηση (Forward Indexing)

Έστω ότι έχουμε ένα βιβλίο ή μια διπλωματική εργασία στην οποία αναφέρονται διάφοροι όροι. Ο πίνακας περιεχομένων περιέχει πληροφορίες σχετικά με την πληροφορία ανά σελίδα. Θεωρητικά, δηλαδή, αν κάποιο πρόγραμμα καλούνταν να φτιάξει τον πίνακα με είσοδο μόνο το κείμενο, θα το ανέλυες ώστε να εξάγει τις λεκτικές μονάδες και θα έφτιαχνε ανά σελίδα έναν πίνακα με το ποιες εμφανίζονται ή θα μπορούσε να τις συνοψίσει ώστε να συμπυκνώσει το νόημα της σελίδας σε λίγες λέξεις. Έτσι, αν κάποιος έψαχνε έναν όρο, θα ανέτρεχε σε κάθε σελίδα σειριακά και θα συγκέντρωνε τα αποτελέσματα.

Ανεστραμμένη δεικτοδότηση (Inverse Indexing)

Με την ανεστραμμένη δεικτοδότηση, στο παραπάνω παράδειγμα αντί να έχουμε σα δείκτη τη σελίδα, έχουμε τον ίδιο τον όρο και έτσι τα αποτελέσματα στον ανεστραμμένο πίνακα όπου οι όροι μας οδηγούν σε σελίδες. Σε περιπτώση, λοιπόν, που ένας χρήστης αναζητήσει κάποιον όρο αντί να εξεταστεί κάθε σελίδα ξεχωριστά, υπάρχει αντιστοίχιση απευθείας. Τέλος, αν χρειαστεί να συνδυαστούν όροι για την αναζήτηση, γίνεται απλά μια τομή των αποτελεσμάτων ανά όρο.
Η διαδικασία αυτή ενέχει το κόστος της δεικτοδότησης κατά την προσθήκη νέου κειμένου.

Κατανεμημένες βάσεις δεδομένων

Οι κατανεμημένες βάσεις δεδομένων (distributed databases) αποτελούν μια αρχιτεκτονική προσέγγιση που αποσκοπεί στη διάχυση των δεδομένων σε πολλαπλούς κόμβους ή διακομιστές. Η επιλογή αυτή απορρέει από την ανάγκη για κλιμακωσιμότητα, υψηλή διαθεσιμότητα και ανθεκτικότητα σε σφάλματα. Σε ένα τέτοιο περιβάλλον, τα δεδομένα δεν αποθηκεύονται σε έναν κεντρικό αποθηκευτικό χώρο, αλλά κατανέμονται με τρόπο που επιτρέπει την παράλληλη επεξεργασία και την ελαχιστοποίηση της καθυστέρησης στην πρόσβαση. Αυτό έχει ιδιαίτερη σημασία σε εφαρμογές που απαιτούν χαμηλό χρόνο απόκρισης και μπορούν να εξυπηρετούν ταυτόχρονα έναν μεγάλο αριθμό χρηστών.

Ωστόσο, η κατανεμημένη φύση μιας βάσης δεδομένων εισάγει και προκλήσεις. Ένα από τα πιο συνηθισμένα ζητήματα αφορά τα πλεονάζοντα δεδομένα (redundant data). Η πλεοναστικότητα μπορεί να εμφανιστεί είτε σκόπιμα, μέσω της αναπαραγωγής (replication), είτε ακούσια, ως αποτέλεσμα ασυνέπειας στη διανομή των εγγραφών. Η αναπαραγωγή δεδομένων σε πολλούς κόμβους είναι μια συνειδητή στρατηγική, η οποία ενισχύει τη

διαθεσιμότητα: ακόμη κι αν ένας κόμβος αποτύχει, τα δεδομένα παραμένουν προσβάσιμα από κάποιον άλλο. Παράλληλα, η ύπαρξη πολλαπλών αντιγράφων επιτρέπει την εξισορρόπηση φορτίου και τη βελτίωση της συνολικής απόδοσης του συστήματος.

Από την άλλη πλευρά, τα πλεονάζοντα δεδομένα συνεπάγονται και κόστος συντήρησης. Κάθε φορά που μια εγγραφή τροποποιείται, το σύστημα πρέπει να διασφαλίσει ότι οι αλλαγές θα συγχρονιστούν σωστά σε όλα τα αντίγραφα. Το πρόβλημα αυτό αναφέρεται συχνά ως πρόκληση της συνέπειας (consistency challenge). Ανάλογα με το μοντέλο που υιοθετείται (π.χ. strong consistency, eventual consistency), η συμπεριφορά του συστήματος μπορεί να διαφοροποιείται σημαντικά.

Τέλος, είναι αξιοσημείωτο ότι σε ορισμένες περιπτώσεις, η πλεοναστικότητα δεν αποτελεί απλώς πρόβλημα, αλλά πλεονέκτημα. Η αποθήκευση δεδομένων σε πολλαπλές τοποθεσίες μπορεί να βελτιώσει την ασφάλεια, να μειώσει τον κίνδυνο απώλειας, και να επιτρέψει πιο γρήγορη ανάκτηση πληροφοριών από κόμβους που βρίσκονται γεωγραφικά εγγύτερα στον τελικό χρήστη. Συνεπώς, η διαχείριση πλεοναζόντων δεδομένων στις κατανεμημένες βάσεις είναι θέμα σχεδιαστικής στρατηγικής και εξαρτάται από τις εκάστοτε ανάγκες και προτεραιότητες της εφαρμογής.

Προσκήνιο Εφαρμογών (Front-End)

Με την ακραία εξάπλωση του διαδικτύου, ήταν αναγκαίο να αναπτυχθούν όλες οι πτυχές του. Είδαμε παραπάνω πως προέκυψαν νέες ανάγκες για την αποθήκευση των δεδομένων. Οι ανάγκες οδηγούν σε νέες τεχνολογίες. Έτσι συνέβη, λοιπόν, και στο προσκήνιο των εφαρμογών και των υπηρεσιών, δηλαδή στο κομμάτι αυτό των εφαρμογών με το οποίο αλληλεπιδρούν οι χρήστες.

Η υιοθέτηση μιας εφαρμογής έγκειται εκ πρώτης όψεως στον οπτικό σχεδιασμό της, ο οποίος πρέπει να χαρακτηρίζεται από λειτουργικότητα και σαφήνεια. Οι εφαρμογές αρχικά ήταν διαθέσιμες μέσω ενός τερματικού. Αυτό μπορεί να φαίνεται η πιο απλή επιλογή, αλλά στην πραγματικότητα επωφελούνται μόνο οι έμπειροι χρήστες. Τα τερματικά παρέχουν τη δυνατότητα για εμφάνιση ASCII χαρακτήρων, κάτι που περιόριζε την εγγενή ανάγκη των ανθρώπων για οπτικοποίηση, καθώς ήταν αδύνατο να αποτυπωθούν γραφικά στοιχεία. Αυτό, και εφεύρεση του ποντικιού υπολογιστή, το οποίο καθιστούσε την αλληλεπίδραση ακόμα πιο άμεση, γέννησαν τις γραφικές διεπαφές χρηστών (**G**raphical **U**ser **I**nterfaces).

Στον ιστό, οι γραφικές διεπαφές έκαναν την εμφάνισή τους με την HTML, οι οποίο είναι μια σημασιολογική γλώσσα. Πρακτικά μόνο υποδεικνύει το που θα φαίνεται τι στην οθόνη. Σε συνδυασμό με τα διαδοχικά φύλλα ύφους (**C**ascading **S**tyle **S**heets), αποτελούν το πρότυπο για ιστοσελίδες για πάνω από 25 χρόνια. Οι φυλλομετρητές (browsers) αναλύουν τα αρχεία HTML και κατασκευάζουν ένα δέντρο, του οποίοι οι κόμβοι είναι τα γραφικά στοιχεία που αναφέρονται στο HTML αρχείο. Έτσι, μπορεί να ορίσει κανείς στοιχεία στην οθόνη καθώς και να εμφωλεύσει στοιχεία μέσα σε άλλα στοιχεία. Η δομή αυτή ήταν αρχικά αρκετή για τις ανάγκες της εποχής καθώς με μια σχετικά απλή, αλλά στατική λύση, μπορούσε κανείς να λαμβάνει αρχεία HTML με διαφορετικό περιεχόμενο και ουσιαστικά να πλοηγηθεί σε διάφορες ιστοσελίδες.

Virtual Dom και SPA’s

Αυτό απέχει πάρα πολύ και καθόλου από τα σημερινά δεδομένα. Οι ιστοσελίδες σήμερα παρέχουν τη δυνατότητα της αλληλεπίδρασης, δεν είναι στατικές, αλλά δυναμικές. Αυτό προέκυψε, όταν οι φυλλομετρητές άρχισαν να υιοθετούν την γλώσσα Javascript, η οποία παρείχε δυνατότητες επεξεργασίας του δέντρου κατά την αλληλεπίδραση του χρήστη με τη σελίδα. Η διαφορά αυτής της λειτουργίας με τα σημερινά πρότυπα είναι ότι πλέον το δέντρο, το οποίο αποκαλείται Document Object Model (DOM), δεν ανανεώνεται εξ’ ολοκλήρου αν προκύψει κάποια αλλαγή, παρά μόνο ο συγκεκριμένος κόμβος στον οποίο έγινε η αλλαγή.

Τη λειτουργία αυτή την παρέχει μια τεχνολογία που ονομάζεται εικονικό DOM (Virtual DOM).

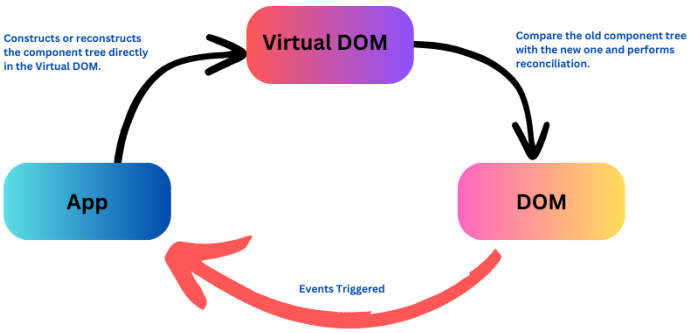


Figure 1: VDOM lifecycle

Το Virtual DOM είναι ουσιαστικά μια εσωτερική, ελαφριά αναπαράσταση του πραγματικού DOM, που διατηρείται στη μνήμη του προγράμματος περιήγησης. Μέσα από αυτό, το σύστημα μπορεί να συγκρίνει την τρέχουσα κατάσταση της σελίδας με την προηγούμενη (μέσω μιας διαδικασίας που ονομάζεται *diffing*) και να υπολογίζει τις ελάχιστες απαραίτητες αλλαγές που πρέπει να εφαρμοστούν στον πραγματικό DOM. Αυτό μειώνει δραματικά την ανάγκη για εκτεταμένα re-renders, τα οποία θα κατανάλωναν σημαντικούς πόρους και θα επιβράδυναν την εμπειρία χρήσης.

Το Virtual DOM αποτελεί κομβικό στοιχείο στις πιο σύγχρονες βιβλιοθήκες και frameworks, όπως το React και το Vue.js, και είναι ένας από τους βασικούς λόγους που κατέστησαν δυνατή την ανάπτυξη ιδιαίτερα πολύπλοκων, διαδραστικών εφαρμογών στο web. Η ύπαρξή του δεν μειώνει απλά το κόστος των λειτουργιών ενημέρωσης, αλλά επίσης επιτρέπει την υιοθέτηση πιο δηλωτικών μοντέλων ανάπτυξης, όπου ο προγραμματιστής περιγράφει απλά την επιθυμητή κατάσταση της διεπαφής, αφήνοντας το Virtual DOM να αναλάβει τις λεπτομέρειες της ενημέρωσης.

Παρόλο που η χρήση του VDOM εισάγει μια πρόσθετη πολυπλοκότητα στο επίπεδο της υλοποίησης, η αποδοτικότητα και η ευκολία ανάπτυξης που προσφέρει καθιστούν τη λύση αυτή καίρια. Πρόκειται για ένα από τα σημαντικότερα βήματα της κοινότητας του λογισμικού στην κατεύθυνση της βελτιστοποίησης της απόδοσης των web εφαρμογών, ανοίγοντας τον δρόμο για εμπειρίες χρήσης που πλησιάζουν αυτές των εγγενών εφαρμογών.

Στα ίδια πλαίσια, άρχισαν να υιοθετούνται κατά κόρον οι εφαρμογές μιας σελίδας (**S**ingle **P**age **A**pplications). Πρακτικά, αξιοποιώντας την κατά συνθήκη ανανέωση του Virtual DOM, οι προγραμματιστές αντιλήφθηκαν ότι δε χρειάζεται να ανταλλάσσονται ολόκληρες, σελίδες HTML μεταξύ ενός χρήστη και ενός εξυπηρετητή, αρκεί να υπάρχει ένας μεθοδευμένος μηχανισμός, ο οποίος θα ανανεώνει κατάλληλα το Virtual DOM, βάσει της αλληλεπίδρασης. Αυτό έχει ως αποτέλεσμα να γλιτώνονται κύκλοι ανανέωσης μιας ιστοσελίδας, άρα και χρόνος,

οπότε ο χρήστης να έχει συνολική μια ομαλότερη και γρηγορότερη εμπειρία πλοήγησης. Ο πολύπλοκος μηχανισμός αυτός ωστόσο δεν έρχεται χωρίς κόστος. Η αρχική φόρτωση μιας ιστοσελίδας ενδέχεται να παίρνει περισσότερο χρόνο διότι περιέχει εκτελέσιμο κώδικα, ο οποίος επιτρέπει τις κατά συνθήκες αλλαγές. Ωστόσο, όλο αυτό το εγχείρημα ήρθε παράλληλα με τη ριζική βελτίωση των παγκόσμιων υποδομών δικτύου (5G, οπτικές ίνες), η οποία κατέστησε δυνατή μια τέτοια αλλαγή εφόσον υιοθετήθηκαν σε επίπεδο νοικοκυριού.

Server Side Rendering ή Client Side Rendering

Υιοθετώντας αυτές τις αλλαγές, προέκυψαν δύο ακόμα ορολογίες, οι οποίες απαντώνται στο καθημερινό λεξιλόγιο ενός προγραμματιστή παρασκηνίου. Το «χτίσιμο» της ιστοσελίδας (rendering) είναι ο πολύπλοκος μηχανισμός που κατασκευάζει το δέντρο με τα στοιχεία μιας ιστοσελίδας. Αναλύει τον κώδικα HTML, προσθέτει οποιεσδήποτε λειτουργίες είναι δηλωμένες μέσω του κώδικα Javascript, προσθέτει οποιαδήποτε δεδομένα ζητούνται και παρασκευάζει την τελική μορφή HTML, η οποία θα είναι διαθέσιμη προς προβολή στον χρήστη. Τι γίνεται ωστόσο σε περιπτώσεις που αυτή η διαδικασία είναι υπολογιστικά επίπονη και χρονοβόρα; Εδώ υπάρχουν δύο προσεγγίσεις:

Κατασκευή της σελίδας στο επίπεδο του εξυπηρετητή (**S**erver **S**ide **R**endering)

Η διαδικασία κατασκευής του περιεχομένου, το οποίο ενδέχεται να είναι εμπλουτισμένο με τα δεδομένα από κάποια βάση και προσαρμοσμένο στην αλληλεπίδραση, γίνεται όλη σε επίπεδο εξυπηρετητή, δηλαδή στον υπολογιστή ο οποίος παρέχει την ιστοσελίδα στο χρήστη και όχι στον υπολογιστή του χρήστη. Με αυτή τη μέθοδο επωφελούνται οι χρήστες, καθώς οι υπολογιστικές απαιτήσεις καλύβονται προτού καν εμφανιστούν οι ιστοσελίδες σε αυτούς. Αυτό περιλαμβάνει επίσης και οποιεσδήποτε απαιτήσεις υπάρχουν όσον αφορά την αισθητική της σελίδας. Είναι συχνό φαινόμενο σε ιστοσελίδες να φορτώνεται πρώτα ο σκελετός της σελίδας και εφόσον το αρχείο css δεν έχει φορτωθεί, το αποτέλεσμα στην οθόνη δεν είναι το αναμενόμενο, με την ιστοσελίδα ωστόσο να παραμένει λειτουργική στον πυρήνα της. Συμβαίνει σε περιστάσεις με κακή σύνδεση στο διαδίκτυο. Αξιοποιώντας το SSR, η εμπειρία είναι σίγουρα πιο γρήγορη και ο τρόπος αυτός καθίσταται αναγκαίος για χρήστες με παλαιότερες συσκευές.

Κατασκευή της σελίδας στο επίπεδο του πελάτη (**C**lient **S**ide **R**endering)

Σε αυτή την περίπτωση, ο φυλλομετρητής του πελάτη φορτώνει ένα μέρος της ιστοσελίδας το οποίο είναι αρκετό για το πρώτο χτίσιμο. Στην πορεία, και όσο ο χρήστης αλληλεπιδρά με τη σελίδα, τρέχει κώδικα Javascript για να παράξει τις νέες σελίδες που προκύπτουν. Στην προσέγγιση αυτή συνίσταται καλή σύνδεση στο διαδίκτυο για μια ομαλή εμπειρία. Αυτό συμβαίνει καθώς για να χτιστεί δυναμικά το περιεχόμενο βάσει της αλληλεπίδρασης, απαιτείται ένα γενναίο μέγεθος κώδικα το οποίο θα σταλεί στον πελάτη κατά την πρώτη επίσκεψη στην ιστοσελίδα. Άρα ο αρχικός χρόνος φόρτωσης αυτής επηρεάζεται. Ακόμα, δεδομένου ότι όλη η διαδικασία γίνεται πλέον στον υπολογιστή του πελάτη, απαιτούνται υπολογιστικοί πόροι καθόλη τη διάρκεια παραμονής στην ιστοσελίδα.

Η διαδικασία λόγω αυτής της διαφοράς υπόκειται στην ορολογία του *Edge computing,* δηλαδή οι υπολογισμοί λαμβάνουν μέρος στην «άκρη» ενός δικτύου, όπου η άκρη σε αυτή την περίπτωση είναι ο εκάστοτε υπολογιστής ενός χρήστη. Πρακτικά, οι ανάγκες για υπολογιστικούς πόρους στους εξυπηρετητές μειώνονται εφόσον ένα μέρος του φόρτου εργασίας έχει μεταφερθεί από τον εξυπηρετητή στον πελάτη.

Η μέθοδος αυτή εφαρμόζεται συχνά σε ιστοσελίδες οι οποίες έχουν διαρκώς μεταβλητό περιεχόμενο και υψηλό δείκτη αλληλεπίδρασης, δηλαδή εφαρμογές σα συνομιλίες ή τα κοινωνικά δίκτυα, όπου οι αλληλεπιδράσεις προκαλούν κινούμενες εικόνες (animations). Επίσης, παρέχει στις ιστοσελίδες τη δυνατότητα να ανταλλάσσουν μικρό όγκο δεδομένων, εφόσον η επεξεργασία για την αλλαγή του περιεχομένου γίνεται στον πελάτη. Μειώνεται, άρα, ο χρόνος απόκρισης από τη στιγμή που ένας χρήστης θα αλληλεπιδράσει μέχρι αυτή που θα φανερωθεί στην οθόνη το αποτέλεσμα της ενέργειας του, αφού δεν είναι απαραίτητη η ανανέωσης ολόκληρης της σελίδας.

SSR , CSR και Μηχανές Αναζήτησης

Η θεμελιώδης λειτουργία μιας μηχανής αναζήτησης είναι η εξής: ένας μηχανισμός ανίχνευσης ιστού (crawler) φορτώνει στο παρασκήνιο ιστοσελίδες, τις αναλύει και εφαρμόζει αλγόριθμους πάνω στα μεταδεδομένα της ανάλυσης. Αν κάποια ιστοσελίδα, λοιπόν, απαιτεί επιπλέον βήματα από τον πελάτη για την κατασκευή της (οι crawlers είναι πρακτικά χρήστες-πελάτες), τότε ενδέχεται η δεικτοδότηση στη μηχανή αναζήτησης να είναι ελλιπής ή και εσφαλμένη.

Με το SSR, δεδομένου ότι το περιεχόμενο είναι έτοιμο εξαρχής, άρα σχετικά στατικότερο -στην όψη του πελάτη- από την αντίθετη περίπτωση, οι μηχανισμοί ανίχνευσης μπορούν αποδοτικότερα να επεξεργαστούν το περιεχόμενο της σελίδας ώστε να εμφανίζεται συχνότερα σε αναζητήσεις. Αυτό συμβαίνει γιατί πολλοί από αυτούς δεν τρέχουν, για λόγους ευελιξίας, ταχύτητας και ασφάλειας, τον υποκείμενο απαραίτητο κώδικα για την κατασκευής της ιστοσελίδας. Οπότε, το SSR έχει νόημα περισσότερο σε περιπτώσεις όπου το περιεχόμενο είναι ίδιο για όλους τους χρήστες, όπως είναι οι εμπορικές ιστοσελίδες με επιθυμητή αυξημένη κίνηση. Ακόμα, δεν είναι απίθανο κατά το CSR να προκύψει μια ίδια ιστοσελίδα για δύο διαφορετικά urls, καθώς οι crawlers δε γνωρίζουν εξαρχής το περιεχόμενο, διπλασιάζοντας έτσι τη δουλειά τους. Αντίθετα, το CSR ενδείκνυται για ιστοσελίδες των οποίων το περιεχόμενο είναι εξατομικευμένο. Αυτές μπορεί να είναι ιστοσελίδες όπως συγκεντρωτικά ταμπλό με μετρήσεις ή πίνακες ελέγχου διαχειριστών, αλλά και αρχικές οθόνες συνεχούς ροής όπως οι αρχικές σελίδες όλων των κοινωνικών δικτύων.

Web ή Native Apps

Ένα ακόμα πεδίο ανάπτυξης, όσον αφορά τις διεπαφές αλληλεπίδρασης μεταξύ του χρήστη και του υπολογιστή, είναι η προσαρμογή του περιεχομένου σε οθόνες διαφόρων διαστάσεων, από την άποψη της κλιμακοποίησής του, των οποιωνδήποτε λειτουργικών, σχεδιαστικών και συστημικών αλλαγών, οι οποίες είναι οι αναγκαίες ώστε η εμπειρία να είναι αναλλοίωτη, όταν ένας χρήστης αλλάζει συσκευές. Την τελευταία δεκαετία οι φορητές συσκευές κατέχουν το μεγαλύτερο μερίδιο της αγοράς [5]. Επομένως, ήταν φυσικό η στόχευση των προγραμματιστών front-end να προσανατολιστεί προς τα εκεί. Ενώ αρχικά οι επιμέρους πάροχοι λογισμικών φορητών συσκευών παρείχαν εργαλεία τα οποία μπορούσαν να φανούν

χρήσιμα μόνο για το μερίδιο της καθεμιάς εταιρείας-λογισμικού , στην πορεία βγήκαν στην επιφάνεια εργαλεία τα οποία παρέχουν οικουμενικές λύσεις ανεξαρτήτου πλατφόρμας (cross-platform). Αυτό μείωσε σημαντικά την πολυπλοκότητα και το πλήθος γραμμών κώδικα καθώς πλέον με τα εργαλεία αυτά, οι προγραμματιστές είναι ικανοί να παράγουν δικτυακές εφαρμογές αλλά και native.

Εγγενής (Native) Εφαρμογές

Οι native εφαρμογές διαφέρουν στον πυρήνα τους από τις δικτυακές κατά τον ακόλουθο τρόπο. Οι δικτυακές εφαρμογές απαιτούν κάποια μηχανή Javascript, δηλαδή έναν φυλλομετρητή, ο οποίος θα είναι υπεύθυνος να αναλύσει τις σελίδες HTML και τα αρχεία Javascript και να τα μετατρέψει σε κώδικα μηχανής για το εκάστοτε μηχάνημα στο οποίο τρέχει. Δηλαδή, το αρχικό πρόγραμμα τρέχει στο φυλλομετρητή, ο οποίος με τη σειρά του είναι πρόγραμμα και τρέχει στον υπολογιστή, άρα υπάρχει ένα περαιτέρω στάδιο κατά την πραγματική εκτέλεση του κώδικα. Οι εγγενής εφαρμογές, εντούτοις, τρέχουν απευθείας στον υπολογιστή και δεν απαιτούν κάποιο φυλλομετρητή.

Καθολικές λύσεις και cross-platform εφαρμογές

Τα πιο γνωστά τέτοια πλαίσια (frameworks) είναι:

React (cross-platform εκδοχή ReactNative).

Η React JSX είναι μια καινοτομία που προήλθε από το Facebook Inc. (πλέον Meta Platforms Inc.) και αποτέλεσε την απαρχή της εξέλιξης για την ανάπτυξη front-end εφαρμογών. Πλέον αποτελεί λογισμικό ανοικτού κώδικα ωστόσο από την ομάδα του Facebook προκύπτουν ανά καιρούς διάφορες βελτιώσεις και πρόσθετες λειτουργίες, τις οποίες χρησιμοποιεί η ομάδα πρώτα εσωτερικά. Αποτελεί μια επέκταση της γλώσσας Javascript και πρακτικά ο κώδικας μοιάζει με αυτόν της Javascript αναμεμειγμένος με HTML στοιχεία. Παρείχε στοιχεία τα οποία αύξησαν κατακόρυφα την επαναχρησιμοποίηση του κώδικα και πρόσθεσαν προγραμματιστική λογική στην απλή σημασιολογική HTML. Ήταν αυτή που εισήγαγε την έννοια του Virtual DOM που αναλύθηκε παραπάνω. Το React Native αποτελεί την προσέγγιση του έργου για native εφαρμογές.

Flutter

Το Flutter είναι μια βιβλιοθήκη της γλώσσας Dart, το οποίο αναπτύχθηκε εσωτερικά στην Google μέχρι το «άνοιγμα» του στην κοινότητα το 2017. Αποτελεί μια cross platform λύση από την άποψη ότι με τον ίδιο κώδικα και διαφορετικά build stages μπορεί κανείς να εξάγει εκτελέσιμα για Android, IOS, Desktop εφαρμογές, αλλά και για δικτυακή εφαρμογή. Σε σχέση με το React Native, το οποίο αξιοποιεί την Javascript σα γέφυρα ώστε να παραχθεί κώδικας για την εκάστοτε συσκευή που τρέχει την εφαρμογή, το Flutter παράγει απευθείας τον native κώδικα και έτσι οι χρόνοι μεταγλώττισης είναι γρηγορότεροι.

Παρασκήνιο εφαρμογών (Back-End)

Οι εφαρμογές, με την εξέλιξη των επικοινωνιών, απέκτησαν υπερτοπικό χαρακτήρα. Άρχισαν δηλαδή να είναι μέρος ενός δικτύου με αφετηρία τον υπολογιστή που τρέχει η εφαρμογή. Το μεγαλύτερο μέρος της δουλειάς σταδιακά άρχισε να συμβαίνει σε μεγάλες υπολογιστικές δομές, δωμάτια εξυπηρετητών κατά τη φυσική έννοια, αλλά πιο αφαιρετικά σε αυτό που πλέον ονομάζεται παγκόσμιος ιστός.

Αυτό πλέον έχει αποδειχθεί από τις κρισιμότερες αλλαγές που επέφερε η άνοδος του διαδικτύου από πολλές πλευρές. Η έρευνα, η εκμάθηση, και η προσβασιμότητας της πληροφορίας αναπτύχθηκαν όσο ποτέ άλλοτε. Τεχνικά, καθίστησε διαθέσιμες τις όποιες υπηρεσίες προσαρμόστηκαν σε αυτό το μοντέλο, διαθέσιμες τόσο από άποψη χρόνου λειτουργίας (uptime), αλλά προφανώς και από άποψη τοπικότητας. Ακόμα, προσέθεσε ένα ακόμα επίπεδο αφαίρεσης καθώς οι υπηρεσίες πλέον αποσυνδέθηκαν από το τοπικό υλικό και λογισμικό στο οποίο έτρεχαν και πλέον αρκεί ένας browser για μεγάλο μέρος υπηρεσιών. Τέλος, από την πλευρά του παρόχου, συγκεντρώνοντας σε συγκεκριμένα σημεία ενός δικτύου το τελικό εκτελέσιμο, καταστήθηκε πιο εύκολη η μελέτη ως προς τη διαθεσιμότητα σε περίπτωση αυξημένης κίνησης, πιο προσιτή η διαδικασία αναβάθμισης του συστήματος και άρα και πιο διαχειρίσιμα τα κόστη για συντήρηση, τόσο υλικά όσο και λογισμικά.

Η έννοια του παρασκηνίου εφαρμογών περιλαμβάνει τις απαιτήσεις αυτής, σε κομμάτι λογισμικού. Δεδομένου όμως, ότι οι εφαρμογές σήμερα, είναι χτισμένες με υψηλές προδιαγραφές όπως το να είναι σχεδόν πάντα διαθέσιμες, κατανεμημένες και κλιμακούμενες από υλική άποψη, αυτό είχε σαφές αντίκτυπο και στην ανάπτυξη λογισμικού ώστε να πληρούνται αυτές οι απαιτήσεις.

Παρακάτω θα αναλυθούν κάποιες τεχνολογίες και ορολογίες, τις οποίες είναι βέβαιο ότι θα συναντήσει κανείς αναπτύσσοντας μια διαδικτυακή εφαρμογή σήμερα.

Πρωτόκολλα επικοινωνίας και κατεύθυνση πληροφορίας

Οι παρακάτω τεχνολογίες είναι σχετικές με το ροή της πληροφορίας και με το ρόλο που έχουν τα δύο άκρα του διαύλου. Παρουσιάζονται επίσης και μερικές ενδεδειγμένες χρήσεις.

Request – Response (Αίτημα – Απάντηση)

Η πιο απλή προσέγγιση όσον αφορά την επικοινωνία μεταξύ δύο υπολογιστών. Ο ένας αιτείται κάτι και ο άλλος απαντάει, είτε θετικά είτε αρνητικά, εφόσον έχει εγκαθιδρυθεί η σύνδεση μεταξύ τους. Αν όχι, τότε το πρόβλημα υπερβαίνει τα όρια αυτής της παραγράφου και συνήθως είναι δικτυακό στον πυρήνα του. Δηλαδή είναι δυνατόν να είναι λάθος η ζητούμενη διεύθυνση ή να μη λειτουργεί ο ένας υπολογιστής.

Ο τρόπος αυτός είναι ο πιο απλός και ο πιο συνήθης και απαντάται σε όλες τις διαδικτυακές εφαρμογές. Η σύνδεση δεν είναι σε πραγματικό χρόνο και συνήθως δεν είναι αμφίδρομη από την άποψη ο πελάτης συνήθως προκαλεί την επικοινωνία. Τέλος, η διάρκεια ζωής της

σύνδεσης είναι ίδια με αυτή της ερωτοαπάντησης, δηλαδή κλείνει με το πέρας της λήψης της απάντησης από τον υπολογιστή με το αίτημα.

Server Sent Events

Τα Server Sent Events είναι μια τεχνολογία μονόδρομης επικοινωνίας. Συνήθως, αξιοποιούνται από εξυπηρετητές προς πελάτες και όχι το ανάποδο. Η σύνδεση παραμένει και εδώ ενεργή δίχως να υπάρχει δοσοληψία μηνυμάτων.
Μπορεί να χρησιμοποιηθεί για ανανεώσεις ζωντανής ροής, όπως για παράδειγμα είναι η ζωντανή τοποθεσία σε εφαρμογές με χάρτες και καθοδήγηση. Είναι μια τεχνολογία είναι περίπου 20 ετών, υπάρχουν περιορισμοί όπως το ότι κάθε φυλλομετρητής μπορεί να διατηρήσει μέχρι και 6 τέτοιες συνδέσεις ανοικτές ανά url. Είναι διαθέσιμο μέσα από το `EventSource` API της Javascript, κατά το οποίο ένας πελάτης εγγράφεται σε έναν εξυπηρετητή μέσω ενός url.

WebSockets

Τα websockets είναι εξέλιξη των Server Sent Events, όντας μια τεχνολογία αμφίδρομης επικοινωνίας. Η σύνδεση παραμένει ενεργή χωρίς απαραίτητα να υπάρχει δοσοληψία μηνυμάτων και για να κλείσει πρέπει είτε να δοθεί εντολή από κάποια άκρο είτε να υπάρξει σφάλμα δικτύου σε φυσικό επίπεδο.
Τα websockets χρησιμοποιούν πλαίσια δεδομένων πινγκ - πονγκ για να ανιχνεύσουν σφάλματα στην επικοινωνία. Έτσι, το ένα άκρο θα στείλει ένα στοιχειώδες μήνυμα και αν δε λάβει μια στοιχειώδη απάντηση μετά από έναν προκαθορισμένο αριθμό προσπαθειών, τότε η σύνδεση διακόπτεται. Η διακοπή μπορεί να γίνει αντιληπτή και από το λειτουργικό σύστημα των ενεργών συνδέσεων αν ο υπολογιστής χάσει για παράδειγμα τη σύνδεση στο διαδίκτυο. Αυτό, ωστόσο, μπορεί να πάρει μερικά λεπτά και καθιστά την πινγκ - πονγκ προσέγγιση αποδοτικότερη Ενδεδειγμένες εφαρμογές είναι οι συνομιλίες, τα παιχνίδια με πολλούς παίκτες και οι ειδοποιήσεις εφαρμογών.

Προσωρινή αποθήκευση δεδομένων (Caching)

Κατά τη χρήση μιας εφαρμογής είναι σχεδόν βέβαιο ότι κάποιο κομμάτι των δεδομένων θα ζητηθεί παραπάνω από μια φορά. Ακόμα και στη πιο απλή περίπτωση που ένας χρήστης κλείσει καταλάθος μια καρτέλα σε στατική ιστοσελίδα, όταν την ξανανοίξει θα πραγματοποιηθεί μια επερώτηση (query) το οποίο θα επιφέρει το ίδιος αποτέλεσμα. Αυτό μπορεί να αποφευχθεί πολύ απλά, αν ένα από τα δύο μέρη αποθηκεύσει προσωρινά το αποτέλεσμα της επερώτησης και ανανεώσει το αποτέλεσμα μόνο όταν είναι βέβαιο ότι ο ζητούμενος πόρος έχει τροποποιηθεί. Έτσι, την επόμενη φορά που θα ζητηθεί, αντί να χρειαστεί να ανατρέξει στη βάση ή στο δίσκο, εξυπηρετεί κατευθείαν το αίτημα από την προσωρινή αποθήκευση, αν τηρείται βέβαια το κριτήριο που θέσαμε.
Αυτό το παράδειγμα είναι το πιο αφελές, ωστόσο παρουσιάζει με απλό τρόπο πως μπορεί κανείς να εξοικονομήσει υπολογισμούς και δικτυακή κίνηση, δυνητικά μόνο με ένα bit. Το επόμενο ερώτημα, βέβαια, που θα συναντήσει κάποιος αξιοποιώντας αυτή την τεχνική είναι το εξής: τι κι αν δεν ζητηθεί ποτέ ξανά αυτός ο πόρος; Ποτέ για ένα σύστημα με δεκάδες χιλιάδες ερωτήματα προς διαχείριση μπορεί να σημαίνει χρόνος της τάξης του δεκαλέπτου. Ακόμα, τι γίνεται στην περίπτωση που ο πόρος τροποποιηθεί; Μήπως είναι προτιμότερο να ανανεωθεί πρώτα αυτή η προσωρινή μνήμη ώστε να είναι άμεσα διαθέσιμο το αποτέλεσμα; Τέτοια

ζητήματα λύνονται με τη σχεδίαση και υιοθέτηση διάφορων στρατηγικών προσωρινής αποθήκευσης, οι οποίες περιλαμβάνουν απαντήσεις για το αν θα ενημερωθεί πρώτα η προσωρινή μνήμη ή όχι, για το πόσο καιρό θα μένουν διαθέσιμοι πόροι για τους οποίους δεν υπάρχει ζήτηση κλπ.

REST, GraphQL και RPC's

Οι αναφερθείσες τεχνολογίες αφορούν ένα αφαιρετικό, σχεδιαστικό κομμάτι λογισμικού και προέκυψαν και αυτές ως απαντήσεις στις ανάγκες βελτίωσης της απόδοσης, με τρόπο είτε ευθύ είτε εν μέρει πιο έμμεσο.

Οι τεχνολογίες αυτές, κάθε μία ξεχωριστά, αποτελούν διαφορετικές προσεγγίσεις στη δομή και στον τρόπο με τα οποία ένας υπολογιστής αιτείται δεδομένα από έναν άλλον. Επίσης, πραγματεύονται μεθόδους με τους οποίους κωδικοποιούνται αυτά τα δεδομένα πριν την αποστολή.

Representational State Transfer (REST)

Με το REST, ο εξυπηρετητής αντιστοιχεί τους πόρους σε αναγνωριστικές συμβολοσειρές (url endpoints). Αξιοποιεί πολλές μεθόδους του πρωτοκόλλου HTTP για να σηματοδοτήσει διάφορες ενέργειες στα δεδομένα που πραγματεύεται κάθε ερώτημα HTTP, ενώ ενδεχόμενες αλλαγές στον τρόπο με τον οποίο ο εξυπηρετητής απαντάει, πρέπει να σηματοδοτούνται με αλλαγή στην έκδοση της διεπαφής προς τους πελάτες. Είθισται ακόμα κάθε πόρος να έχει μοναδικό αναγνωριστικό. Σημαντική λεπτομέρεια και έναυσμα προς την ανάπτυξη των τεχνολογιών που έπονται είναι το γεγονός ότι κάθε απάντηση έχει προκαθορισμένη δομή. Αυτό είναι ύψιστης σημασίας, καθώς δημιουργεί το πρόβλημα του overfetching, δηλαδή της λήψης δεδομένων τα οποία ενδέχεται να υπερβαίνουν το σκοπό του ερωτήματος. Το πρόβλημα αυτό έρχεται να λύσει το graphql. Η δομή αυτή είναι καθορισμένη στον εξυπηρετητή και οι πελάτες καλούνται να δουλέψουν με το σχήμα που είναι διαθέσιμο.

GraphQL

Το GraphQL είναι μια τεχνολογία, η οποία αναπτύχθηκε στο Facebook από το 2012, ενώ η εταιρία το έκανε διαθέσιμο στο κοινό υπό άδειες λογισμικού ανοικτού κώδικα το 2015. Πρακτικά, πρόκειται για ένα λεξικολογικό αναλυτή, ο οποίος αντιστοιχεί τα αιτήματα σε συγκεκριμένες συναρτήσεις ανά πόρο. Αυτό που είναι σημαντικό, είναι ότι ο πελάτης μπορεί να απαιτήσει μόνο συγκεκριμένα πεδία από κάθε πόρο, μειώνοντας έτσι το μέγεθος της απάντησης, κάτι που αξιοποιήθηκε από τους δημιουργούς για τις mobile εφαρμογές τους. Ωστόσο, δεδομένου ότι το GraphQL χρησιμοποιεί ένα url για όλα τα ερωτήματα, είναι δύσχρηστο όσον αφορά τεχνικές caching. Αυτό μπορεί να επιτευχθεί βέβαια, αλλά απαιτεί επιπλέον δουλειά για τον προγραμματιστή.

Remote Procedure Calls

Πρόκειται για παλιά τεχνολογία η οποία ήρθε στην επιφανεια για δεύτερη φορά στα μέσα της προηγούμενης δεκαετίας με την άνοδο των αρχιτεκτονικών συστημάτων με μικρο-υπηρεσίες. Έχει εγγενή διαχείριση για συνεχή ροή μηνυμάτων (streaming) και αποδοτικότερο μέγεθος κατά την αποστολή, καθώς το φορτίο σειριοποιείται για τη μετάδοση, σε αντίθεση με τις δύο

προηγούμενες περιπτώσεις όπου χρησιμοποιείται κωδικοποίηση κειμένου κατά κόρον (json). Ένα ακόμα πλεονέκτημα είναι ότι δύναται να επικοινωνούν δύο ή και παραπάνω ροές μηνυμάτων σε μία μόνο σύνδεση καθώς υποστηρίζει πολυπλεξία στην επικοινωνία. Είναι ιδανικό για ανομοιογενείς αρχιτεκτονικές καθώς θέτει ένα συμβόλαιο επικοινωνίας, το οποίο αγνοεί γλώσσες προγραμματισμού και έτσι είναι άμεσα καταναλώσιμο από οποιαδήποτε υπηρεσία.

Η κύρια διαφορά του, όμως, με την εγκαθιδρυμένη εναλλακτική (REST) είναι ότι η διεπαφή που εκτίθεται στο χρήστη, είναι πλήρως συνυφασμένη με τη λειτουργία που εκτελεί κάθε δράση και αποσυνδέεται από πρακτικά ζητήματα, όπως είναι η δομή ενός URL και η μέθοδος HTTP που πρέπει να χρησιμοποιηθεί. Δηλαδή δύναται ο προγραμματιστής να ορίσει στη διεπαφή ακριβώς την πράξει που θέλει να εκτελεστεί (πχ αντί για POST /app/v1/user/create + post data, υπάρχει το rpc /createUser). Αυτό το κάνει ιδανικό για την ενδοεπικοινωνία ενός συστήματος, στην οποία η απόκρυψη των μηχανισμών που επιτυγχάνεται με ένα URL δεν είναι καίριας σημασίας. Τέλος, μειώνει τις γραμμές κώδικα που καλείται να γράψει ένας προγραμματιστής γιατί παρέχει εργαλεία για αυτόματη δημιουργία κώδικα βάσει των προδιαγραφών που αυτός ορίζει.

Πιστοποίηση (Authentication) και Εξουσιοδότηση (Authorization)

Οι δύο αυτές έννοιες έχουν να κάνουν με την ασφάλεια των εφαρμογών και συνήθως συγχέονται εσφαλμένα. Η μεν πιστοποίηση έχει να κάνει με το ποιόν του χρήστη όταν αλληλεπιδρά με μια εφαρμογή και επιτυγχάνεται με την παροχή από το χρήστη κάποιου είδους πιστοποιητικού όπως είναι όνομα χρήστη και κωδικός. Πλέον, αξιοποιούνται κατά κόρον και τα βιομετρικά χαρακτηριστικά ενός ατόμου για την πιστοποίηση και συχνά επίσης γίνεται συνδυασμός μεθόδων. Οι παρακάτω δύο μέθοδοι που αναλύονται εξηγούν πρακτικά τι συμβαίνει κατά την επιτυχή πιστοποίηση ενός χρήστη.

Συνεδρίες (Sessions)

Στην πιστοποίηση με συνεδρίες (Session-Based Authentication), κατά την είσοδο του χρήστη στην εφαρμογή, δημιουργείται σε κάποια βάση δεδομένων μια συνεδρία και μεταδίδεται στο χρήστη το αναγνωριστικό της, το οποίο μεταφέρεται από τον πελάτη στον εξυπηρετητή με κάθε ερώτημα. Έτσι σε κάθε ερώτημα αρκεί η εφαρμογή να βεβαιωθεί ότι η συνεδρία είναι έγκυρη. Ένα πλεονέκτημα που έχει αυτή η μέθοδος είναι η ασφάλεια σε επιθέσεις ενδιάμεσου, καθώς ακόμα και αν κάποιος καταφέρει να υποκλέψει το αναγνωριστικό, δεν έχει πρόσβαση στην εφαρμογή εφόσον η πιστοποίηση γίνεται στον εξυπηρετητή. Αυτό επίσης επιτρέπει στην εφαρμογή να ελέγχει ακριβώς το πότε τελειώνει μια συνεδρία και έχει τη δυνατότητα να την διακόψει αυτοβούλως.

JWT Tokens

Η πιστοποίηση με τεκμήριο γίνεται ως εξής: κατά την είσοδο στην εφαρμογή, εκδίδεται από αυτή ένα τεκμήριο που περιλαμβάνει διάφορες κωδικοποιημένες πληροφορίες για τον πελάτη, αλλά και ένα μελλοντικό χρόνο λήξης του τεκμηρίου. Ο πελάτης περιλαμβάνει αυτό το τεκμήριο σε κάθε του ερώτημα. Η διαφορά με τις συνεδρίες είναι ότι αυτό δεν αποθηκεύεται κάπου, παρά μόνο αποκωδικοποιείται και αν αυτό συμβεί επιτυχώς τότε προχωράνε οι όποιες ενέργειες. Επομένως, πρόκειται για έναν τρόπο πιστοποίησης ο οποίος δεν απαιτεί μέσο

αποθήκευσης, άρα είναι ιδανικός για εφαρμογές μεταξύ πολλών domains. Η επιφύλαξη στη χρήση αυτής της μεθόδου είναι η περίπτωση κάποιος να παρεμβληθεί στην επικοινωνία και να αποσπάσει αυτό το τεκμήριο από το μήνυμα. Τότε, δυνητικά μπορεί να υποδυθεί τον αποστολέα και να κάνει τη δουλειά του ανενόχλητα.

Η εξουσιοδότηση δε, έχει να κάνει με το επίπεδο πρόσβασης που απαιτούν διάφοροι χρήστες στην εφαρμογή. Ελέγχει την πρόσβαση σε πόρους με βάση τους ρόλους, τα δικαιώματα και τις πολιτικές. Για παράδειγμα, η πιστοποιημένη σύνδεση σε μια τραπεζική εφαρμογή, δεν σημαίνει ότι ένας χρήστης μπορεί να μεταφέρει μεγάλα χρηματικά ποσά χωρίς τα κατάλληλα προνόμια (εξουσιοδότηση). Αντίστοιχα, ένας χρήστης ο οποίος έχει πιστοποιηθεί, μπορεί να έχει πρόσβαση στο προφίλ του αλλά όχι σε κάποιο ταμπλό διαχείρισης μιας ομάδας ή μιας σελίδας, εκτός αν έχει τον κατάλληλο ρόλο. Οι κοινοί μηχανισμοί εξουσιοδότησης περιλαμβάνουν κατά κύριο λόγο, τον έλεγχο πρόσβασης βάσει ρόλων (**R**ole-**B**ased **A**ccess **C**ontrol) και τον έλεγχο πρόσβασης βάσει χαρακτηριστικών (**A**ttribute-**B**ased **A**ccess **C**ontrol). Σε αντίθεση με τον έλεγχο πιστοποίησης, ο οποίος πραγματοποιείται πριν από τη χορήγηση πρόσβασης, η εξουσιοδότηση είναι μια συνεχής διαδικασία που ελέγχει τα δικαιώματα κάθε φορά που ένας χρήστης προσπαθεί να εκτελέσει μια ενέργεια ή να αιτηθεί κάποιον πόρο.

Message Brokers

Οι διαμοιραστές μηνυμάτων λειτουργούν ως εξειδικευμένο ενδιάμεσο λογισμικό που διευκολύνει την επικοινωνία μεταξύ διαφορετικών υπηρεσιών σε κατανεμημένα συστήματα. Δύνανται να μοιράσουν δεδομένα αλλά και να τα δρομολογήσουν μεταξύ εφαρμογών, παρέχοντας τρόπους σύνδεσης για πληθώρα γλωσσών προγραμματισμού και πλατφόρμων. Εφαρμόζοντας ένα επίπεδο αφαίρεσης μεταξύ των παραγωγών και των καταναλωτών μηνυμάτων, αποσυνδέουν αποτελεσματικά τα συστήματα, επιτρέποντάς τους να επικοινωνούν χωρίς άμεση γνώση των λεπτομερειών υλοποίησης, της διαθεσιμότητας ή των δυνατοτήτων επεξεργασίας ο ένας του άλλου. Αυτή η αποσύνδεση ενισχύει την ανθεκτικότητα του συστήματος, καθώς οι προσωρινές διακοπές σε μια υπηρεσία δεν επηρεάζουν απαραίτητα τις άλλες, και επιτρέπει την ανεξάρτητη κλιμάκωση διαφορετικών τμημάτων του συστήματος σύμφωνα με τις συγκεκριμένες απαιτήσεις φόρτου τους.

Το Apache Kafka, πέρα από τη βασική μεταφορά μηνυμάτων, προσφέρει δυνατότητες επιμονής μηνυμάτων για διαστήματα ορισμένα από τον χρήστη, για ανθεκτικότητα σε βλάβες του συστήματος, εγγυημένους μηχανισμούς παράδοσης μέσω πολιτικών επιβεβαίωσης παράδοσης και μηχανισμούς επαναπαράδοσης σε περίπτωση αποτυχίας. Άλλοι διαμοιραστές, όπως το RabbitMQ προσφέρουν ιεράρχηση μηνυμάτων για τη διαχείριση διαφόρων επιπέδων επείγουσας ανάγκης, σύνθετες δυνατότητες δρομολόγησης με βάση το περιεχόμενο ή τα μεταδεδομένα του μηνύματος και υποστήριξη συναλλαγών (transactions) για τη διατήρηση της συνέπειας των δεδομένων. Πολλοί brokers παρέχουν επίσης φιλτράρισμα μηνυμάτων, δυνατότητες μετασχηματισμού για την προσαρμογή μηνυμάτων μεταξύ διαφορετικών απαιτήσεων συστήματος και ειδικούς μηχανισμούς αποθήκευσης για περιπτώσεις σφάλματος κατά την παράδοση.

Pub Sub

Το μοτίβο δημοσίευσης-εγγραφής (pub/sub) αντιπροσωπεύει ένα μοντέλο ανταλλαγής μηνυμάτων όπου οι αποστολείς μηνυμάτων (publishers) ταξινομούν τα μηνύματα σε θέματα ή κανάλια αντί να τα δρομολογήσουν προς συγκεκριμένους παραλήπτες. Σε αυτό το

παράδειγμα, οι παραλήπτες μηνυμάτων (subscribers) εγγράφονται στις ουρές μηνυμάτων ενός ή περισσότερων θεμάτων και λαμβάνουν αυτόματα μηνύματα που έχουν δημοσιευτεί σε αυτά τα θέματα, δημιουργώντας μια μορφή έμμεσης διευθυνσιοδότησης. Αυτό το μοτίβο έρχεται σε αντίθεση με τα συμβατικά peer2peer μοντέλα ανταλλαγής μηνυμάτων, στα οποία τα μηνύματα έχουν προκαθορισμένο παραλήπτη. Οι δυνατότητές τους τα καθιστούν ιδιαίτερα κατάλληλα για αρχιτεκτονικές που βασίζονται σε συμβάντα, εφαρμογές ροής δεδομένων σε πραγματικό χρόνο, monitoring συστημάτων και εφαρμογές με σύνθετες επιχειρησιακές λειτουργίες όπου διαφορετικά στοιχεία πρέπει να αντιδρούν ανεξάρτητα στα ίδια συμβάντα.

Ενδιάμεσο λογισμικό (Middleware)

Με τον όρο ενδιάμεσο λογισμικό αναφερόμαστε σε ένα κομμάτι λογισμικό το οποίο παρεμβάλλεται ανάμεσα σε δύο άλλο. Στις διαδικτυακές εφαρμογές, αναφέρεται συνήθως σε κώδικα ο οποίος τρέχει μεταξύ ενός αιτήματος κάποιας υπηρεσίας σε κάποια άλλη και κυρίως σε κώδικα που παρεμβάλλεται μεταξύ των αιτημάτων που κάνει ένας πελάτης μέσω διεπαφής και της υπηρεσίας η οποία υλοποιεί τελικά την εφαρμογή. Ο κώδικας αυτός μπορεί να αποτελεί κομμάτι της εφαρμογής, αλλά ενώ συνήθως ο χαρακτήρας του είναι λειτουργικός ως προς ζητήματα εκτός της επιχειρησιακής λογικής του προβλήματος που καλείται να λύσει η εφαρμογή, για αυτό αναφέρεται ως ενδιάμεσος κώδικας.

Βασικές χρήσεις ενός ενδιάμεσου κώδικα επιλύουν ζητήματα ασφαλείας αλλά και εχεμύθειας ή επιτρέπουν/διευκολύνουν την επικοινωνία μιας υπηρεσίας με μια άλλη χωρίς να επιδρούν άμεσο στο κύριο επιχειρησιακό κομμάτι της εφαρμογής. Πολλές φορές μάλιστα, όπως στο παράδειγμα αυτής της εφαρμογής, ο ενδιάμεσος κώδικας καλείται να πληροί κάποια τεχνικά κριτήρια ώστε διαφορετικά κομμάτια αυτού, να μπορούν να συντίθενται και να εμφωλεύονται μεταξύ τους χωρίς πρόβλημα.

Κλιμακούμενες εφαρμογές

Πακέτο και διάθεση λογισμικού (packaging and deployment)

Η διαδικασία του πακεταρίσματος (packaging) λογισμικού αποτελεί ένα κρίσιμο στάδιο στον κύκλο ζωής ανάπτυξης εφαρμογών. Στόχος της είναι να συγκεντρώσει όλα τα απαραίτητα συστατικά – εκτελέσιμους κώδικες, βιβλιοθήκες, αρχεία ρυθμίσεων και τεκμηρίωση – σε μια ενιαία μορφή που να μπορεί να εγκατασταθεί και να χρησιμοποιηθεί με συνέπεια και ακρίβεια σε διαφορετικά περιβάλλοντα. Η τυποποίηση του packaging διευκολύνει τη διανομή της εφαρμογής τόσο εσωτερικά στην ομάδα ανάπτυξης όσο και προς τελικούς χρήστες.

Από την άλλη πλευρά, η διάθεση (deployment) είναι η πράξη της εγκατάστασης και ενεργοποίησης του πακεταρισμένου λογισμικού σε ένα συγκεκριμένο περιβάλλον, όπως ένας server παραγωγής, μια πλατφόρμα cloud ή ο υπολογιστής ενός τελικού χρήστη. Στη σύγχρονη ανάπτυξη λογισμικού, το deployment δεν περιορίζεται πλέον σε χειροκίνητες διαδικασίες· αντίθετα, υιοθετούνται αυτοματοποιημένα pipelines (CI/CD) που επιτρέπουν τη συνεχή ενσωμάτωση και παράδοση αλλαγών. Με αυτόν τον τρόπο, εξασφαλίζεται ταχύτητα, αξιοπιστία και δυνατότητα επαναληψιμότητας στις εκδόσεις.

Οι τεχνολογίες που έχουν κυριαρχήσει σε αυτό το πεδίο είναι οι containers (π.χ. Docker), οι οποίοι επιτρέπουν την πακετοποίηση λογισμικού μαζί με το εκτελεστικό του περιβάλλον, εξασφαλίζοντας φορητότητα και συνέπεια σε πολλαπλά συστήματα. Επιπλέον, εργαλεία

όπως το Kubernetes παρέχουν δυνατότητες ορχήστρωσης και αυτόματης κλιμάκωσης, καθιστώντας τη διάθεση λογισμικού μια διαδικασία που μπορεί να ανταποκριθεί σε απαιτητικές συνθήκες παραγωγής.

Τέλος, αξίζει να σημειωθεί ότι το packaging και το deployment δεν αποτελούν μόνο τεχνικά βήματα, αλλά συνδέονται με την ποιότητα και ασφάλεια του λογισμικού. Μέσω υπογεγραμμένων πακέτων, έλεγχων ακεραιότητας και ασφαλών pipelines, μειώνεται ο κίνδυνος αλλοίωσης του λογισμικού και διασφαλίζεται η εμπιστοσύνη τόσο των χρηστών όσο και των οργανισμών που το υιοθετούν.

Κεφάλαιο 3

Στο κεφάλαιο αυτό θα ασχοληθούμε με πρακτικά ζητήματα αρχιτεκτονικής σχεδίασης της εφαρμογής, της βάσης δεδομένων, των συστατικών που την αποτελούν σε φυσικό, δικτυακό επίπεδο, αλλά και σε επίπεδου λογισμικού. Θα δούμε:

- τη δομή της βάσης δεδομένων σε θέμα οντοτήτων και σχέσεων, πως αποθηκεύεται δηλαδή η πληροφορία για κάθε εμπλεκόμενη οντότητα και το πως αυτές αλληλεπιδρούν μεταξύ τους

- διαγράμματα αλληλεπίδρασης ενός χρήστη με την εφαρμογή

- διαγράμματα της λογικής ροής της εφαρμογής, το πώς δηλαδή μεταδίδονται οι ενέργειες του χρήστη και οι πληροφορίες ώστε να έχουμε τελικά το επιθυμητό αποτέλεσμα.

- διαγράμματα της δομής της διανομής του λογισμικού

Αρχιτεκτονικές συστημάτων

Κατά την ανάπτυξη των εφαρμογών τα τελευταία χρόνια, μπορεί κανείς εύκολα να υποθέσει ότι δεν υπήρχε κάποιο αρχέτυπο το οποίο θα υποδείκνυε πως μπορεί να πάρει κανείς μια εφαρμογή και να την επεκτείνει λειτουργικά, διατηρώντας παράλληλα την αξιοπιστία της. Οι εφαρμογές είναι πλέον τόσο πολύπλοκες που απαιτούνται περαιτέρω εφαρμογές ανάλυσης, τόσο κατά την ανάπτυξη, όσο κατά τη συντήρηση, ώστε να έχουμε σαφή εικόνα για αυτές. Όπως είναι σύνηθες στην επιστήμη, η συνεχής έρευνα και τριβή με την ανάπτυξη εφαρμογών γέννησε μοντέλα σχεδίασης και ανάπτυξης υπολογιστικών συστημάτων, εκ των οποίων αναλύουμε κάποιες παρακάτω.

Μονόλιθος

Η χρονολογικά πρώτη αρχιτεκτονική ήταν ο λεγόμενος μονόλιθος. Σε αυτή την προσέγγιση, όλες οι λειτουργίες είναι συγκεντρωμένες σε μια μονάδα και αυτό επηρεάζει έμμεσα όλα τα στάδια κατά την ανάπτυξη της εφαρμογής. Κατά τη συγγραφή του κώδικα, έχουμε σαφή πλεονεκτήματα δεδομένου ότι η εφαρμογή είναι μικρή σε μέγεθος. Όλα τα κομμάτια είναι συγκεντρωμένα, κάτι που καθιστά την αποσφαλματοποίηση πιο εύκολη. Η συγκέντρωση του ενδιαφέροντος, επίσης, καθιστά αυτή την πρακτική ιδανική για εφαρμογές μικρού μεγέθους, καθώς αφενός οι προγραμματιστές δεν επιβαρύνονται, έχοντας μόνο ένα κοινό εφαλτήριο για τον κώδικά τους, αφετέρου γιατί απαιτούνται λιγότερες ενέργειες για τη διάθεση μιας τέτοιας εφαρμογής. Τέλος, με το μονόλιθο διευκολύνεται η διατερματική δοκιμή κώδικα του κώδικα (end-to-end testing), καθώς όλες οι εμπλεκόμενες λειτουργίες είναι συγκεντρωμένες.

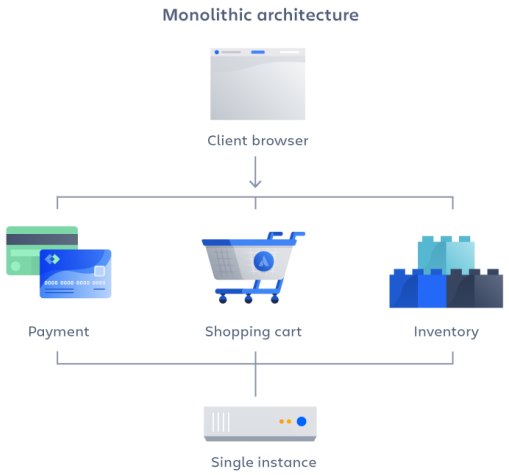


Figure 2: Monolithic Architecture [15]

Υπηρεσίες και Μικρο-υπηρεσίες

Κατά τη διάρκεια επέκτασης μιας εφαρμογής, σε μέγεθος γραμμών κώδικα αλλά και σε πλήθος χαρακτηριστικών που προσφέρει, καθίσταται ολοένα και πιο δύσκολο να συνεχίσει κανείς με αυστηρά μονολιθική τεχνική. Αυτό φαίνεται από το πιο απλό παράδειγμα της διάθεσής της. Με την παραμικρή αλλαγή θα έπρεπε να μεταβαλλόταν όλο το αναρτημένο πακέτο κώδικα και να ξεκινήσει εκ νέου η λειτουργία της εφαρμογής. Έτσι, σταδιακά ξεκίνησε η ανάπτυξη των εφαρμογών, συγκροτημένες από μικρότερες επιμέρους εφαρμογές οι οποίες επιτελούν μικρότερους σκοπούς στην αυτοτέλειά τους.

Αρχιτεκτονική προσανατολισμένη στις υπηρεσίες (Service-Oriented Architecture)

Μια αρχιτεκτονική προσανατολισμένη στις υπηρεσίες (SOA) είναι ένα πρότυπο σχεδίασης κατά το οποίο διάφορες λειτουργίες δημιουργούνται ως επαναχρησιμοποιήσιμα δομικά στοιχεία - υπηρεσίες - που παρέχουν συγκεκριμένες λειτουργίες και μπορούν να από οποιοδήποτε κόμβο ενός δικτύου αλληλοσυνδεόμενων υπηρεσιών. Κάθε υπηρεσία έχει πολύ καθορισμένο σκοπό και πεδίο δράσης και μπορεί να κληθεί από τα λοιπά στοιχεία ή υπηρεσίες εντός της αρχιτεκτονικής. Δύο βασικά πλεονεκτήματα είναι η ευκολία συντήρησης και η επεκτασιμότητα.

Στο SOA, οι συμμετέχοντες είναι εν μέρει αποσυζευγμένοι κατά την ανάπτυξη και κάθε δομική μονάδα είναι ικανή να παρέχει την είσοδο/έξοδο που ζητήθηκε. Κάποιος που καλεί μια υπηρεσία δεν χρειάζεται να γνωρίζει την έκδοση αυτής. Αντίθετα, ελέγχει ένα γενικό μητρώο και βρίσκει την πιο πρόσφατη υπηρεσία που προσφέρει τη λειτουργικότητα που απαιτείται. Συνήθως στην πράξη, οι υπηρεσίες ενώ εκτελούν διαφορετικούς σκοπούς, μοιράζονται την ίδια βάση δεδομένων και υπάρχει η έννοια του μητρώου που αναφέρθηκε παραπάνω, σαν κεντρικός κόμβος επισύνδεσης μεταξύ των υπηρεσιών.

Μικρο-υπηρεσίες (micro-services)

Οι μικρο-υπηρεσίες αποτελούν μια προσέγγιση συγγενή με τη SOA, και μάλιστα αποτελούν την εξέλιξή της. Το γεγονός ότι το μητρώο υπηρεσιών είναι μοναδικό, το καθιστά τη μονάδα ως μοναδικό σημείο αποτυχίας (single point of failure), δηλαδή τυχόν αποτυχία αυτού σηματοδοτεί την κατάρρευση όλου του συστήματος. Έτσι προέκυψε η αρχιτεκτονική μικροϋπηρεσιών, όπου οι υπηρεσίες-δομικοί λίθοι είναι πλέον πλήρως αποσυζευγμένοι. Σχεδιάζονται έτσι ώστε πιθανή αποτυχία κάποιας υπηρεσίας να μην είναι καίρια για την λειτουργία της ευρύτερης εφαρμογής. Η δομή έχει αντίκτυπο και στο προσωπικό δυναμικό, καθώς είθισται κάθε υπηρεσία να αναλαμβάνεται από διακριτή ομάδα, βελτιώνοντας έτσι τη δουλειά των προγραμματιστών και επιταχύνοντας τους χρόνους ανάπτυξης και διάθεσης των υπηρεσιών.

Διαμοιρασμός καθηκόντων

Η αποσύζευξη των υπηρεσιών είναι μια προσέγγιση ανάπτυξης η οποία αποτελεί βασική αρχή των μικρο-υπηρεσιών. Αποσκοπεί σε λογισμικό το οποίο κατά την ολοκλήρωσή του είναι καθόλα επεκτάσιμο, με υψηλή ανθεκτικότητα σε τυχόν επιπλοκές που θα προκύψουν. Βασίζεται στα παρακάτω σημεία: μικρή επίγνωση, σαφή συμβόλαια μεταξύ των υπηρεσιών και εναλλαξιμότητα.

Κάθε υπηρεσία σε ένα τέτοιο σύστημα έχει μικρή επίγνωση του περίγυρού της. Αυτό διαισθητικά μπορεί να ακούγεται εσφαλμένο αλλά σε συνδυασμό με την πλήρη αποσαφήνιση του καθεστώτος επικοινωνίας μεταξύ δύο υπηρεσιών, επιτυγχάνουμε ένα σύστημα από εναλλασσόμενα συστατικά, τα οποία δύνανται να επεκτείνονται το καθένα ανεξάρτητα από τα άλλα, είτε οριζόντια (δηλαδή ύπαρξη περισσότερων φυσικών κόμβων στο δίκτυο της εφαρμογής για τη συγκεκριμένη λειτουργία), είτε κάθετα (δηλαδή αύξηση των ευθυνών και των λειτουργιών που εκτελεί η συγκεκριμένη υπηρεσία).

Αρχιτεκτονικές καθοδηγούμενες από γεγονότα

Μια ακόμα μέθοδος που υιοθετείται συχνά στην ανάπτυξη δικτυακών εφαρμογών είναι αυτή της αρχιτεκτονικής βασισμένη στα γεγονότα. Τα γεγονότα αποτελούν σήματα που ενεργοποιούν διάφορες ενέργειες μεταξύ του συστήματος και συνήθως μπορούν να φέρουν αλλαγές στην κατάσταση του συστήματος (όπως πχ ότι κάποιος χρήστης έκανε κάποια ενέργεια) ή να φέρουν απλώς κάποια αναγνωριστικά και τυχόν κατάσταση να προσδιορίζεται από άλλα σημεία της διαδρομής που ακολουθεί το γεγονός (όπως π.χ. στη διαδρομή “αγορά” να τοποθέτησε κάποια υπηρεσία το αναγνωριστικό ενός αντικειμένου).

Οι αρχιτεκτονικές αυτές συντελούνται από δομικά στοιχεία με τουλάχιστον κάποια από τα τρία παρακάτω χαρακτηριστικά: παραγωγούς, καταναλωτές και δρομολογητές. Οι παραγωγοί κοινοποιούν τα γεγονότα στους δρομολογητές, οι οποίο τα προωθούν στους καταναλωτές. Μια υπηρεσία συνήθως είναι και παραγωγός και καταναλωτής, ενώ οι δρομολογητές συνήθως

εκτελούν αυτό το σκοπό, αφού αρχικά επεξεργαστούν την εισερχόμενη κίνηση και ενδεχομένως την φιλτράρουν.

Λειτουργία και Αρχιτεκτονική εφαρμογής

Παρακάτω θα αναλύσουμε την ιδέα της εφαρμογής από επιχειρησιακή σκοπιά και στη συνέχεια θα προσπαθήσουμε να συνδέσουμε τις ιδέες αυτές με τεχνικά διαγράμματα και ορολογίες ώστε να γίνει αντιληπτό πως μια τέτοια ιδέα υλοποιείται από την αρχή μέχρι το τέλος.

Δομή δεδομένων της εφαρμογής

Η ιδέα της εφαρμογής είναι οι εξής: Απλοί χρήστες και οργανισμοί αιτούνται και δωρίζουν αντικείμενα και υπηρεσίες κατά βούληση. Ο σκοπός είναι να μην υπάρχει καθόλου η έννοια του χρήματος. Για την ικανοποίηση των αιτημάτων, οι αιτούντες μπορούν να δημιουργήσουν εκδηλώσεις (events, είτε διαδικτυακές είτε φυσικές) και καμπάνιες προώθησης. Οι καμπάνιες μπορούν να αποτελούνται από επιμέρους εκδηλώσεις.

Πρακτικά πρέπει να ορίσουμε κάποιες οντότητες οι οποίες θα αναπαριστούν τα δεδομένα μας και θα ορίζουν τις απαιτούμενες λειτουργίες της εφαρμογής. Οι εμπλεκόμενες οντότητες είναι οι εξής: **Χρήστης (user), Οργανισμός (org), Αντικείμενο (item), Υπηρεσία (service), Εκδήλωση (event), Καμπάνια (campaign), Αίτημα (request), Δωρεά (donation)**. Αυτές είναι οι λειτουργικές οντότητες οι οποίες ορίζονται και αξιοποιούνται ώστε να προσδιορίσουμε τεχνικά την ιδέα. Στην πορεία θα προκύψουν και άλλες, οι οποίες είναι βοηθητικές και δεν έχουν νόημα για τις επιχειρησιακές λειτουργίες της εφαρμογής, αλλά εξυπηρετούν πρακτικούς σκοπούς.

Οι παραπάνω οντότητες αναπαρίστανται στο παρακάτω διάγραμμα οντοτήτων-σχέσεων.

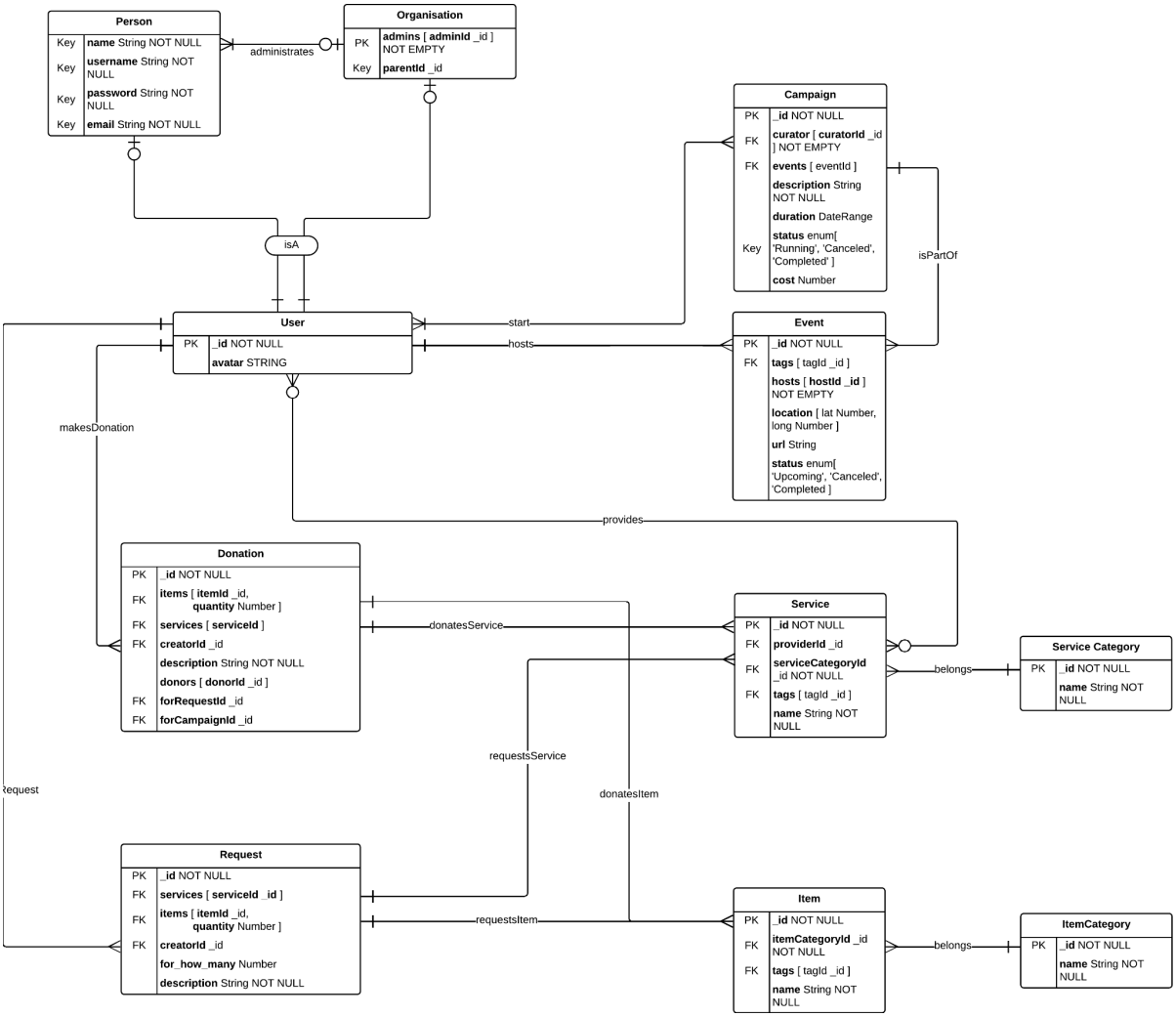


Figure 3: ER διάγραμμα σχεσιακής βάσης

Αντικείμενα και υπηρεσίες

Αντικείμενα

Τα αντικείμενα είναι το ένα από τα δύο είδη που συναλλάσσονται μεταξύ των χρηστών της εφαρμογής. Στη βάση, αποθηκεύονται είτε αυτόνομα ως αντικείμενα που κάποτε κατά τη χρήση της εφαρμογής αξιοποίησε κάποιος χρήστης, είτε ως μέρος μιας δωρεάς ή αιτήματος οπότε αποθηκεύεται μια αναφορά του αντικειμένου (αναγνωριστικό, ID) μαζί με την ορισμένη από το χρήστη ποσότητα. Ταυτόχρονα, κατά τη δημιουργία αποθηκεύεται στη βάση αναζήτησης κειμένου η απλή εκδοχή.

Υπηρεσίες

Αντίστοιχα με τα αντικείμενα, οι υπηρεσίες αποθηκεύονται στη βάση, χωρίς ποσότητα προφανώς. Επίσης, αποθηκεύονται ως μέρος δωρεάς ή αιτήματος.

Τα αντικείμενα όπως και οι υπηρεσίες ανήκουν σε κάποια κατηγορία. Στο επίπεδο της εφαρμογής, έχουν τα εξής πεδία:

- Name: όνομα
- Description: Περιγραφή
- Category: το ID της κατηγορίας που ανήκει

Παρατηρήσεις

- Όταν είναι μέρος μιας δωρεάς ή ενός αιτήματος, τα αντικείμενα έχουν ένα επιπλέον πεδίο με την ποσότητα η οποία το συνοδεύει στη συναλλαγή.
- Όταν είναι μέρος δωρεάς και μόνο, τα αντικείμενα και οι υπηρεσίες συνοδεύονται από το αναγνωριστικό της δωρεάς.

Αιτήματα

Τα αιτήματα εξυπηρετούν την κεντρική ιδέα της εφαρμογής. Μπορούν να δημιουργηθούν αυτόνομα ή να αποτελέσουν μέρος κάποιας καμπάνιας (βλ. παρακάτω). Κατά τη δημιουργία τους, ο χρήστης αναζητά αντικείμενα και υπηρεσίες για να συμπεριλάβει στο αίτημά του. Αν δε μείνει ικανοποιημένος από τις υπάρχουσες παροχές, μπορεί να δημιουργήσει νέες. Άρα η δημιουργία κάποιας παροχής αποτελεί μέρος της δημιουργίας κάποιου αιτήματος.

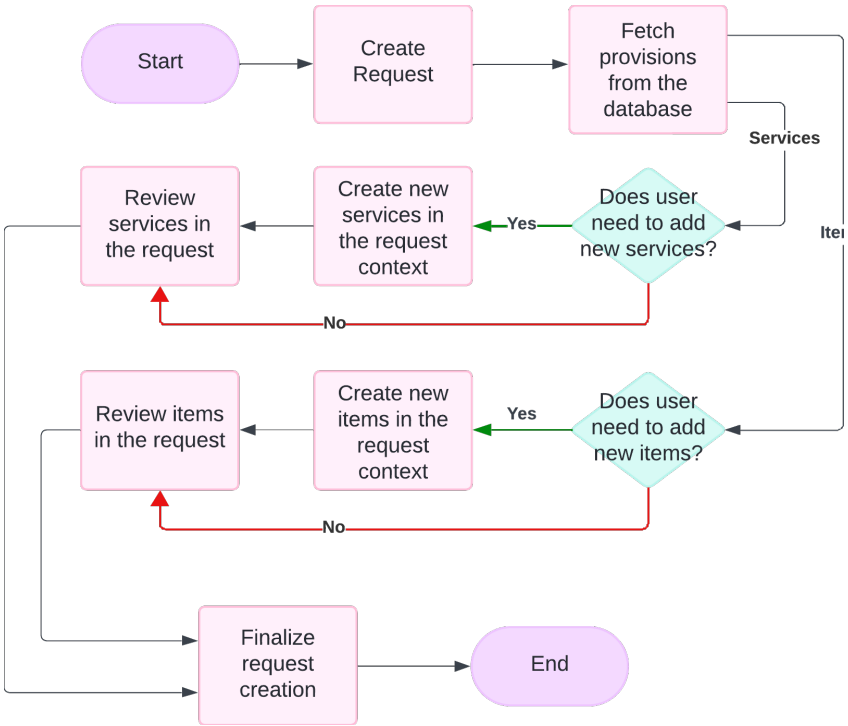


Figure 4: Διάγραμμα λογικής ροής δημιουργίας αιτήματος

Στο επίπεδο της εφαρμογής, έχουν τα εξής πεδία:

- Message: μήνυμα του αιτούντος
- Status: Η κατάσταση του αιτήματος
- UniqueAbility: Η δυνατότητα εξυπηρέτησης του αιτήματος από μοναδικό δωρητή

Η δομή αυτή εμπλουτίζεται από έναν πίνακα ζητούμενων αντικειμένων (με ποσότητα) και έναν πίνακα απαιτούμενων υπηρεσιών.

Δωρεές

Οι δωρεές αποτελούν την “απάντηση” στα αιτήματα. Οποιοσδήποτε χρήστης της εφαρμογής μπορεί να ανταποκριθεί στα αιτήματα που παρουσιάζονται και να συνεισφέρει τα αντικείμενα ή τις υπηρεσίες που ζητούνται. Τέλος, μπορεί να κάνει τη δωρεά του ανώνυμη για κάποιο χρονικό διάστημα ή και καθόλου.

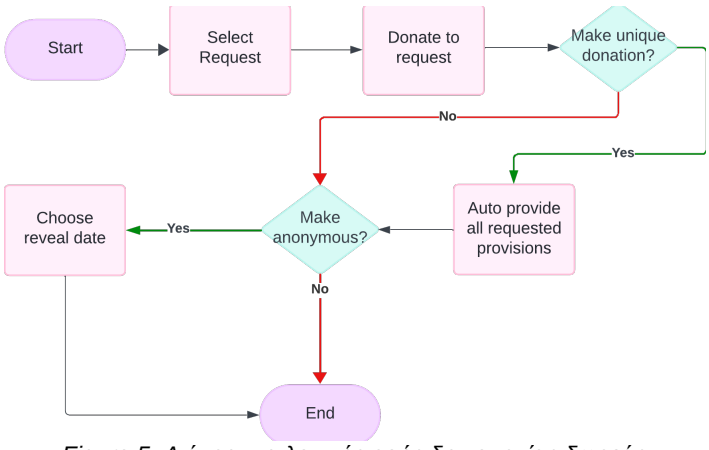


Figure 5: Διάγραμμα λογικής ροής δημιουργίας δωρεάς

Στο επίπεδο της εφαρμογής, έχουν τα εξής πεδία:

- Message: μήνυμα του αιτούντος
- Unique: αν είναι μοναδική ή όχι
- Anonymous: αν είναι ανώνυμη και
- Date_reveal: για πόσο καιρό είναι ανώνυμη
- Request_id: ποιο αίτημα εξυπηρετεί η δωρεά

Χρήστες και οργανισμοί

Ο απλός χρήστης της εφαρμογής μπορεί να εκτελέσεις τις παρακάτω λειτουργίες:

Δημιουργία αιτήματος

Ο χρήστης δημιουργεί ένα αίτημα. Μπορεί να αιτηθεί είτε αντικείμενα, προσδιορίζοντας και την ποσότητα, είτε υπηρεσίες. Αν δεν είναι ικανοποιημένος από τα υπάρχοντα αντικείμενα ή υπηρεσίες, μπορεί να δημιουργήσει μια καινούρια εγγραφή. (παραπάνω διάγραμμα)

Δημιουργία δωρεάς

Ο χρήστης ανταποκρίνεται στα αιτήματα που είναι διαθέσιμα και δωρίζει αντικείμενα είτε καθιστά τον εαυτό του διαθέσιμο για παροχή της απαιτούμενης υπηρεσίας. Ταυτόχρονα με την παροχή της υπηρεσίας, γίνεται γνωστό σε όλη την εφαρμογή ότι ο συγκεκριμένος χρήστης «σχετίζεται» με αυτά τα αντικείμενα και υπηρεσίες και του προτείνονται αυτόματα αυτόματα πιθανές δωρεές όταν προκύψει ένα σχετικό αίτημα.

Δημιουργία εκδήλωσης

Στα πλαίσια προώθησης, ο χρήστης είναι δυνατόν να δημιουργήσει κάποια εκδήλωση, δηλώνοντας είτε φυσικό τόπο, είτε διαδικτυακό παρέχοντας ένα σύνδεσμο.

Δημιουργία καμπάνιας

Σε πιο γενικό πλαίσιο, ο χρήστης δύναται να δημιουργήσει καμπάνια και να αιτηθεί αγαθά στα πλαίσια αυτής. Με την καμπάνια, πέρα από τη δημιουργία σχετικού αιτήματος, μπορεί να συσχετίσει και εκδηλώσεις.

Δημιουργία και διοίκηση οργανισμού

Τέλος, κάθε χρήστης μπορεί να δημιουργήσει ή να διευθύνει οργανισμούς. Πρακτικά, έχουν την ίδια ισχύ με τους χρήστες. Η ύπαρξή τους προσθέτει ένα επίπεδο διαχείρισης των δεδομένων της εφαρμογής.

Αναζήτηση

Όλες οι παραπάνω οντότητες είναι διαθέσιμες προς αναζήτηση βάσει των λεπτομερειών τους. Έτσι μια εκδήλωση, θα φανεί στην αναζήτηση όταν ταιριάξει ο όρος αναζήτησης με το κείμενο στην περιγραφή της, στον τίτλο ή ακόμα και στην τοποθεσία.

Παρατηρήσεις

Οι οργανισμοί μπορούν να εκτελέσουν τις ίδιες λειτουργίες με τους χρήστες πέραν από τη δημιουργία οργανισμού.

Όλες οι δημιουργίες οντοτήτων δημιουργούν γεγονότα στο σύστημα της εφαρμογής, τα οποία μέσω ενός καναλιού διάδοσης προωθούνται στο Kafka, ώστε να ενημερωθούν οι δείκτες της βάσης αναζήτησης αλλά και να διαδοθούν οποιεσδήποτε ειδοποιήσεις είναι απαραίτητο.

Στο επίπεδο της εφαρμογής, οι χρήστες και οι οργανισμοί έχουν τα εξής πεδία:

- Username: όνομα χρήστη στην εφαρμογή
- FirstName: Όνομα
- LastName: Επώνυμο
- Email: διεύθυνση ηλεκτρονικού ταχυδρομείου
- Roles: ρόλους (απλός χρήστης, διαχειριστής κλπ)
- Password: hashed τιμή του κωδικού πρόσβασης
- AvatarURL: σύνδεσμος για την εικόνα του χρήστη

ενώ οι οργανισμοί τα εξής πεδία:

- AdminIDs: τα αναγνωριστικά των διαχειριστών των οργανισμών
- Name: όνομα

- Bio: μικρή περιγραφή
- AvatarURL: σύνδεσμος για την εικόνα του οργανισμού
- Email: διεύθυνση ηλεκτρονικού ταχυδρομείου

Καμπάνιες και εκδηλώσεις

Οι καμπάνιες και οι εκδηλώσεις είναι δύο οντότητες οι οποίες συμβάλλουν σαν προωθητικές ενέργειες κάποιου χρήστη ή οργανισμού ώστε να καλύψει τις ανάγκες του σε δωρεές. Οι εκδηλώσεις μπορεί να είναι μέρος μιας καμπάνιας ή και αυτόνομες.

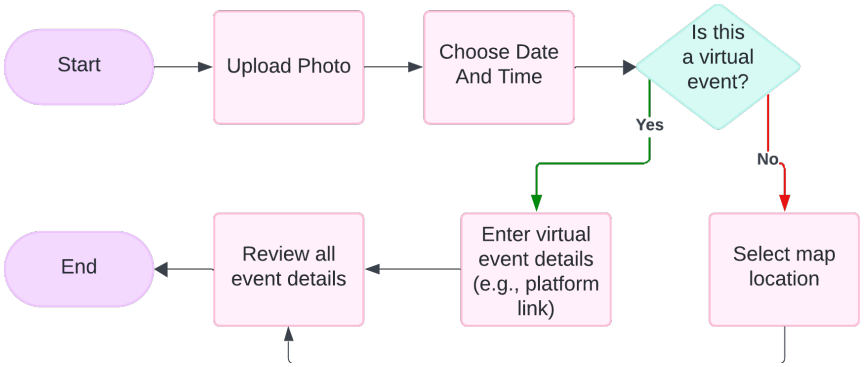


Figure 6: Διάγραμμα λογικής ροής δημιουργίας εκδήλωσης

Στο επίπεδο της εφαρμογής, οι καμπάνιες έχουν τα εξής πεδία:

- Status: η κατάσταση της καμπάνιας
- Title: τίτλος της καμπάνιας
- Description: περιγραφή
- StartDate: ημερομηνία και ώρα έναρξης
- EndDate: ημερομηνία και ώρα λήξης

  Η δομή αυτή εμπλουτίζεται από πίνακες των συνδιοργανωτών (χρηστών και λογαριασμών) και από τα ζητούμενα αγαθά, αν υπάρχουν ως μέρος αιτήματος της καμπάνιας.

ενώ οι εκδηλώσεις έχουν τα εξής πεδία:

- Title: τίτλος της εκδήλωσης
- Description: περιγραφή
- StartDate: ημερομηνία και ώρα έναρξης
- EndDate: ημερομηνία και ώρα λήξης
- CampaignID: το αναγνωριστικό της καμπάνιας στην οποία ανήκει η εκδήλωση, αν ανήκει σε κάποια
- VirtualEventLink: Αν η εκδήλωση είναι διαδικτυακή, εκεί αποθηκεύεται ο σύνδεσμος της εκδήλωσης
- Location: η τοποθεσία στην οποία θα λάβει χώρα η εκδήλωση αν είναι με φυσική παρουσία
- LocationID: το αναγνωριστικό της τοποθεσίας

Στο επίπεδο των δεδομένων, όλες οι παραπάνω οντότητες έχουν επιπρόσθετα τα παρακάτω πεδία:

- ID: αναγνωριστικό του αντικειμένου στη βάση
- CreatedAt: Χρονοσφραγίδα δημιουργίας της εγγραφής στη βάση
- MetadataHolder: γενική δομή αποθήκευσης επιπλέον πληροφορίας που μπορεί να προκύψει. Για παράδειγμα, στα αντικείμενα αποθηκεύεται το αναγνωριστικό της κατηγορίας που ανήκει. Όταν όμως “διαβάζεται” ένα αντικείμενο από τον τελικό χρήστη, αποθηκεύεται σε αυτή τη δομή το όνομα της κατηγορίας, ώστε να είναι διαθέσιμο για απεικόνιση. Συνήθως είναι πληροφορία η οποία προκύπτει από σύνδεσμους (joins) μεταξύ των πινάκων της βάσης.

Τέλος, οι χρήστες διαθέτουν avatars, τα αντικείμενα, οι υπηρεσίες, οι εκδηλώσεις και οι καμπάνιες διαθέτουν φωτογραφίες. Στη σχεσιακή βάση, δεν αποθηκεύονται τα αρχεία των φωτογραφιών, παρά μόνο μια αναφορά αυτών ως ένας σύνδεσμος στη βάση των εικόνων.

Διάγραμμα αρχιτεκτονικής και λειτουργικές λεπτομέρειες

Όπως είδαμε παραπάνω, οι σύγχρονες εφαρμογές δομούνται είπε μονολιθικά είτε σε υπηρεσίες. Το μοντέλο που εφαρμόστηκε στην παρούσα εφαρμογή είναι ένα υβρίδιο, από την άποψη ότι οι κύριες λειτουργίες της εφαρμογής συγκεντρώνονται σε μια υπηρεσία,η οποία πλαισιώνεται από άλλες βοηθητικές με μικρότερες ευθύνες, η απουσία των οποίων δεν επηρεάζει τη λειτουργικότητα της κυρίας υπηρεσίας.

Τα παραπάνω παρουσιάζονται στο διάγραμμα:

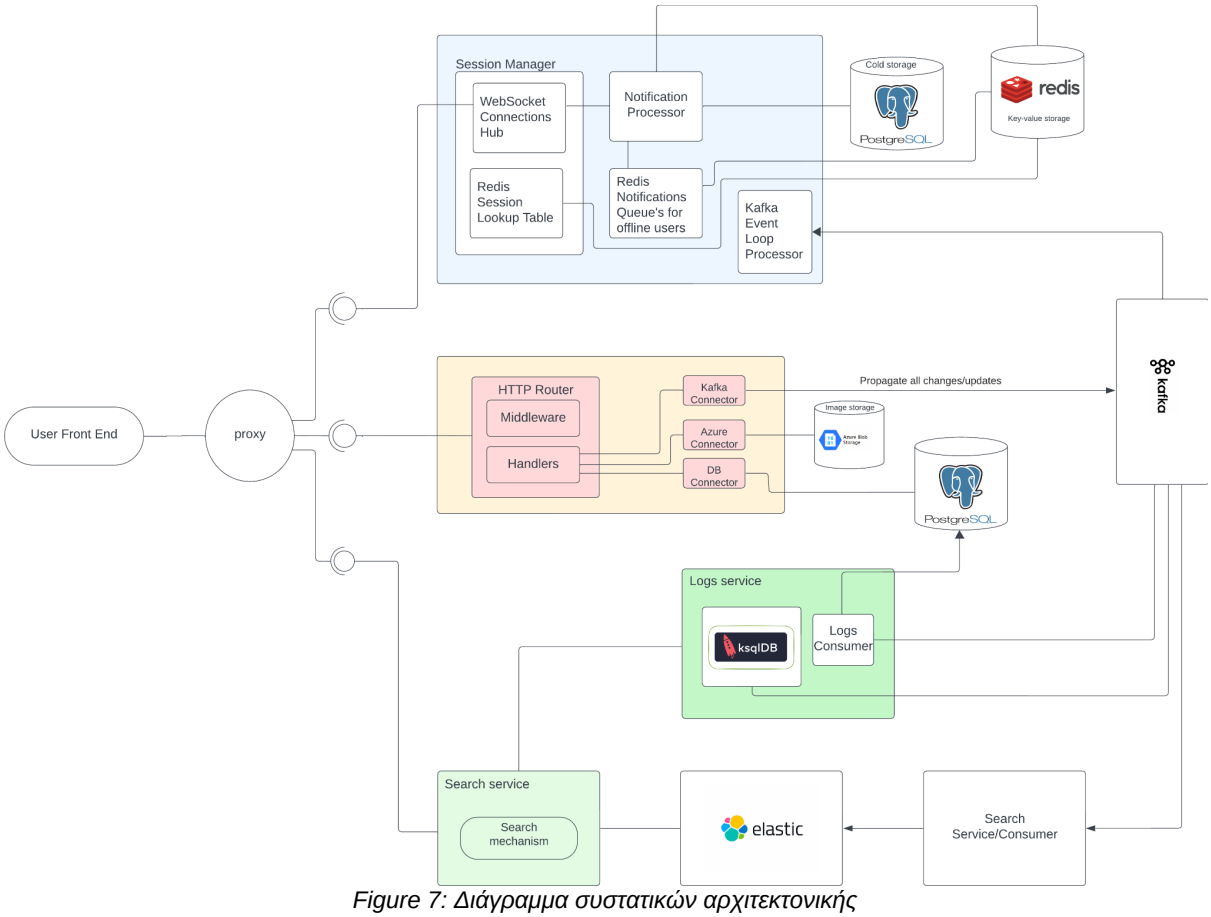


Figure 7: Διάγραμμα συστατικών αρχιτεκτονικής

Συστατικά του διαγράμματος

Κύρια υπηρεσία (κίτρινο φόντο)

Η κυρία υπηρεσία, η οποία επωμίζεται όλο το λειτουργικό βάρος, είναι υπεύθυνη για τις κύριες CRUD (**c**reate, **r**ead, **u**pdate, **d**elete) λειτουργίες όλων των οντοτήτων.

Χρησιμοποιεί μια σχεσιακή βάση δεδομένων (PostgreSQL) για την αποθήκευση των δεδομένων. Η βάση βρίσκεται σε απομακρυσμένο εξυπηρετητή σε εικονικό μηχάνημα της σχολής. Πέρα από τη βάση αυτή, είναι συνδεδεμένη με μία βάση πολυμέσων (Azure Blob Storage) για την αποθήκευση φωτογραφιών. Αφού αποθηκευτεί εκεί με προκαθορισμένο όνομα, τότε η κύρια υπηρεσία δεν εμπλέκεται περαιτέρω στη διαχείριση των αρχείων φωτογραφιών, καθώς διαχειρίζεται πλέον μόνο τις αναφορές των αρχείων ως σύνδεσμους.

Η κύρια υπηρεσία είναι επίσης συνδεδεμένη με ένα στιγμιότυπο ενός διαμοιραστή μηνυμάτων (Kafka broker), όπου γνωστοποιεί στο υπόλοιπο σύστημα τις δημιουργίες των οντοτήτων, τις μετατροπές, τις διαγραφές κλπ, πληροφορία δηλαδή που αξιοποιείται από τα υπόλοιπα κύρια συστατικά όπως θα δούμε παρακάτω.

Η κύρια υπηρεσία αποτελείται από τις παρακάτω οντότητες:

Σε επίπεδο επικοινωνίας με τον έξω κόσμο:

- **HTTP εξυπηρετητής**
  Χειρίζεται την HTTP κίνηση της υπηρεσίας. Διατηρεί δηλαδή τις συνδέσεις όσο τα παρακάτω επίπεδα συνομιλούν με τις βάσεις για τις απαραίτητες αλλαγές που εκτελούν τα HTTP αιτήματα.

- **Σύνδεσμος με σχεσιακή βάση και σύνδεσμος με βάση πολυμέσων**
  Κάθε χειριστής αιτήματος, όπως αυτοί θα αναλυθούν παρακάτω, αξιοποιεί δύο κοινές συνδέσεις, οι οποίες είναι διαθέσιμες σε όλη την υπηρεσία, μία για εκάστοτε βάση. Πρακτικά βέβαια για τη σύνδεση με τη σχεσιακή βάση, χρησιμοποιείται ένα connection pool, δηλαδή ένα πλήθος ταυτόχρονων συνδέσεων με τη βάση, όπου κάθε φορά αξιοποιείται η πρώτη διαθέσιμη [6].

Σε πρώτο εμφωλευμένο επίπεδο:

Ενδιάμεσο λογισμικό (middleware)

Στο ενδιάμεσο επίπεδο επενεργούν δύο κομμάτια λογισμικού, τα οποία είναι υπεύθυνα για την πιστοποίηση και την εξουσιοδότηση.

- **JWT Auth**
  Κατά την πιστοποίηση, εκδίδεται ένα τεκμήριο (token), όπως αναλύθηκε στο πρώτο κεφάλαιο. Σε αυτό εμπεριέχεται πληροφορία για το αν το φέρον το τεκμήριο αίτημα, προέρχεται από πιστοποιημένο χρήστη. Σε αυτό, λοιπόν, το επίπεδο συμβαίνει ο έλεγχος.

- **Εξουσιοδότηση ρόλων**
  Κάθε χρήστης της εφαρμογής έχει πρόσβαση σε συγκεκριμένες ενέργειες σε αυτή, βάση ενός συστήματος ρόλων που αποδίδονται στον εκάστοτε χρήση κατά τη δημιουργία ή με απευθείας παρέμβαση στη βάση. Σε δεύτερα στάδιο, λοιπόν, εφόσον ο χρήστης είναι πιστοποιημένος, ελέγχεται η πρόσβασή του στις συγκεκριμένες ενέργειες, ώστε να λάβει ή όχι την απαραίτητη εξουσιοδότηση.

Σε δεύτερο εμφωλευμένο επίπεδο:

Χειριστές των αιτημάτων

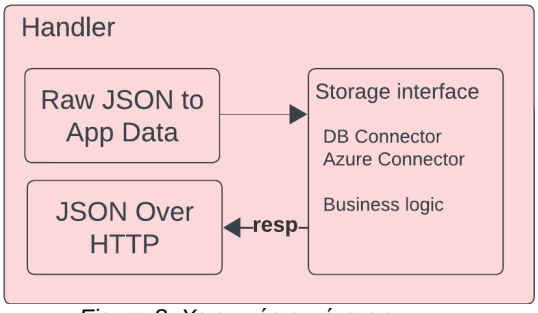


Figure 8: Χειριστής οντότητας

Είναι το κομμάτι του λογισμικού, το οποίο ευθύνεται για την αποσαφήνιση του HTTP αιτήματος και τη μετατροπή αυτής της πληροφορίας σε κάτι αξιοποιήσιμο από την εφαρμογή. Μετατρέπει τα δεδομένα που αποστέλλει το προσκήνιο σε πληροφορία παρεμφερή με τις οντότητες της εφαρμογής. Επίσης, εφαρμόζει σε αυτή διάφορους κανόνες επαλήθευσης δεδομένων (πχ. οι εκδηλώσεις πρέπει να έχουν όνομα). Πρόκειται δηλαδή για αυτό το κομμάτι κώδικα που έχει την ίδια λειτουργία με έναν αποσειριοποιητή (deserializer), εφόσον μετατρέπει τα raw δεδομένα από το HTTP αίτημα σε δεδομένα της εφαρμογής.

Αφού πλέον η πληροφορία ενός HTTP αιτήματος υπάρχει σε αξιοποιήσιμη μορφή, τότε την εκτέλεση αναλαμβάνει το Storage κομμάτι λογισμικού.
Το Storage αποτελεί ένα σύνολο ενεργειών μέσα στις οποίες εφαρμόζεται η λογική που απαιτείται για την ορθή εκτέλεση των ενεργειών. Σε επικοινωνία με τη βάση, γίνονται όλοι οι απαραίτητοι έλεγχοι ώστε να υπάρχει συνοχή των δεδομένων μεταξύ των επενεργειών σε αυτά. Για παράδειγμα, εφαρμόζονται έλεγχοι όπως το να μην υπερβούν η ποσότητα των προσφερόμενων αγαθών την ποσότητα των ζητούμενων. Στο Storage εφαρμόζεται αυτό που λέμε επιχειρησιακή λογική (business logic) της εφαρμογής. Σε πρακτικό κομμάτι κώδικα, αποτελεί επίσης ένα περιτύλιγμα γύρω από τους προκαθορισμένους οδηγούς της γλώσσας για επικοινωνία με τις δύο βάσεις και το διαμοιραστή μηνυμάτων. Οι ενέργειες είναι χωρισμένες ανά οντότητα.

Με το πέρας της ενέργειας και προτού το αποτέλεσμά της αποσταλεί πίσω στο χρήστη που την ενεργοποίησε, αυτό προωθείται στο διαμοιραστή μηνυμάτων, για περαιτέρω αξιοποίηση από τις λοιπές υπηρεσίες του συστήματος.

Περισσότερες λεπτομέρειες στο επόμενο κεφάλαιο με παράθεση του κώδικα και παραδείγματα.

Υπηρεσία αναζήτησης (πράσινο φόντο)

Η υπηρεσία αναζήτησης αποτελείται από τρία κύρια συστατικά.

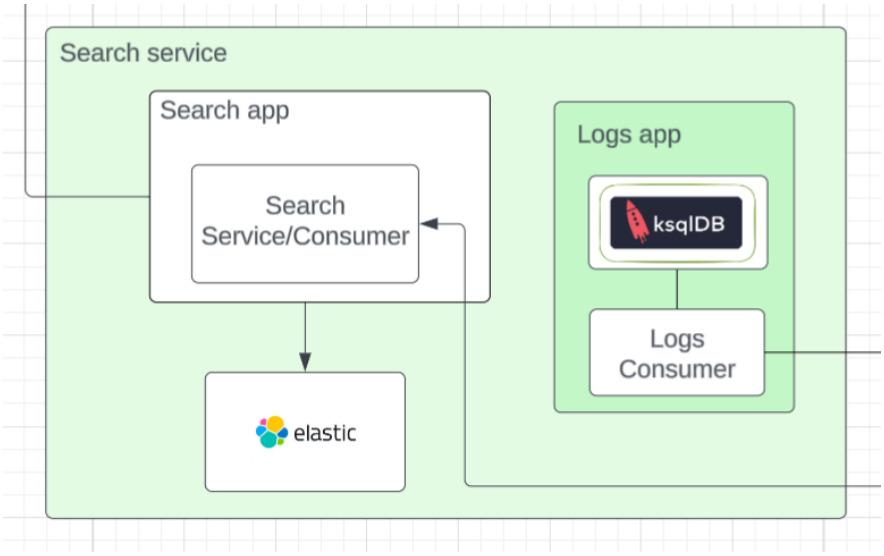


Figure 9: Συστατικά υπηρεσίας αναζήτησης

Έχουμε ένα και μοναδικό σημείο πρόσβασης για την αναζήτηση, το οποίο δέχεται ως παραμέτρους μόνο το κείμενο της αναζήτησης.

Ταυτόχρονα, ένας άλλος μηχανισμός παρακολουθεί τα κατάλληλα θέματα στο διαμοιραστή μηνυμάτων και ενημερώνει τη βάση με την εμφάνιση οποιασδήποτε αλλαγής.

Τέλος, παρέχεται από τους δημιουργούς της βάσης, πακέτο λογισμικού με το οποίο επιτρέπει τη φόρτωση σε αυτή μεγάλου όγκου δεδομένων από τη σχεσιακή βάση, μέσω διεπαφής. Επίσης, δίνεται η δυνατότητα προγραμματισμού εισαγωγής των δεδομένων ανά τακτά διαστήματα (κάθε X ώρες, κάθε Y μέρες), ωστόσο έχει προτιμηθεί η ανανέωση σε πραγματικό χρόνο μέσω μηνυμάτων.

Search logs

Στην υπηρεσία αναζήτησης προσκολλάται μια επιμέρους υπηρεσία η οποία έχει σκοπό να διοχετεύει τα αποτελέσματα των αναζητήσεων στη σχεσιακή βάση εφόσον για διάφορες χρήσεις. Αξιοποιήθηκε το ksqlDB το οποίο είναι μια σχεσιακή βάση με δυνατότητες ροών δεδομένων (streaming).

Αρχικά παράγονται δεδομένα από την υπηρεσία αναζήτησης τα οποία περιλαμβάνουν το ερώτημα αναζήτησης (search query), καθώς και ένα σύνολο από αναγνωριστικά παροχών (*provision ids*) που σχετίζονται με τα αποτελέσματα. Προκειμένου να διασφαλιστεί ότι η επεξεργασία εστιάζει στα πιο σημαντικά στοιχεία, εφαρμόζεται ένα κατώφλι συνάφειας, με βάση το relevance score του Elasticsearch. Συγκεκριμένα, μόνο εκείνα τα αναγνωριστικά που παρουσιάζουν βαθμολογία μεγαλύτερη του 0.65 σε σχέση με το μέγιστο σκορ του εκάστοτε ερωτήματος συμπεριλαμβάνονται στη συνέχεια της διαδικασίας.

Στο επόμενο στάδιο, τα δεδομένα εισέρχονται σε μια ροή (stream) που υλοποιείται στο ksqlDB. Η ροή αυτή είναι υπεύθυνη για τη διαχείριση των γεγονότων αναζήτησης σε πραγματικό χρόνο και την ομαδοποίησή τους με βάση το provision id. Η τεχνική που χρησιμοποιείται είναι η δημιουργία ενός κυλιόμενου παραθύρου διάρκειας 1 ώρας, το οποίο επιτρέπει τον υπολογισμό συγκεντρωτικών μετρήσεων. Για κάθε χρονικό παράθυρο, το ksqlDB καταγράφει το πλήθος των αναζητήσεων που σχετίζονται με κάθε αναγνωριστικό. Με

τον τρόπο αυτόν, προκύπτουν ακριβείς χρονικές τομές της ζήτησης, που προσφέρουν μια ολοκληρωμένη εικόνα της δυναμικής των αναζητήσεων.

Η επιλογή του παραθύρου έγινε λαμβάνοντας υπόψη την επίτευξη ισορροπημένου συμβιβασμού μεταξύ λεπτομέρειας και αποδοτικότητας. Από τη μία πλευρά, μικρότερα χρονικά παράθυρα θα παρείχαν μεν πιο αναλυτικές μετρήσεις, αλλά θα οδηγούσαν σε αυξημένο αριθμό των μηνυμάτων που θα έπρεπε να παραχθούν, να μεταδοθούν και να καταναλωθούν σε πραγματικό χρόνο. Από την άλλη πλευρά, μεγαλύτερα χρονικά διαστήματα θα μείωναν μεν τον φόρτο, αλλά θα απέκρυπταν σημαντικές βραχυπρόθεσμες διακυμάνσεις στη συμπεριφορά των χρηστών. Η διάρκεια της μιας ώρας επιτυγχάνει έτσι τη χρυσή τομή παρέχοντας αρκετή λεπτομέρεια για την παρακολούθηση των τάσεων σε πραγματικό χρόνο, ενώ ταυτόχρονα εξασφαλίζει ότι το σύστημα παραμένει σταθερό, αποδοτικό και επεκτάσιμο.

Τα αποτελέσματα αυτά καταλήγουν εν τέλει στη σχεσιακή βάση όπου εκεί συνδυάζονται με τα πάγια στοιχεία μιας παροχής ώστε να παρέχονται στατιστικά αναζητήσεων για κάθε μία από αυτές. Τα δεδομένα καταχωρούνται λοιπόν χρησιμοποιώντας ως πρωτεύον κλειδί τον συνδυασμό των πεδίων του αναγνωριστικού και της χρονοσφραγίδας έναρξης του εκάστοτε χρονικού παραθύρου. Η συγκεκριμένη επιλογή κλειδιού εξασφαλίζει τη μοναδικότητα των εγγραφών ανά παροχή και χρονικό παράθυρο, αποτρέποντας τη δημιουργία διπλότυπων εγγραφών και επιτρέποντας την ενημέρωση υπαρχόντων συγκεντρωτικών στατιστικών.

Υπηρεσία ειδοποιήσεων (μπλε φόντο)

Η υπηρεσία των ειδοποιήσεων αποτελείται από τον κορμό και πέντε επιμέρους συστατικά. Αξιοποιεί δύο βάσεις δεδομένων, μια σχεσιακή και μία key-value, και καταναλώνει τα γεγονότα των ειδοποιήσεων από το διαμοιραστή μηνυμάτων.

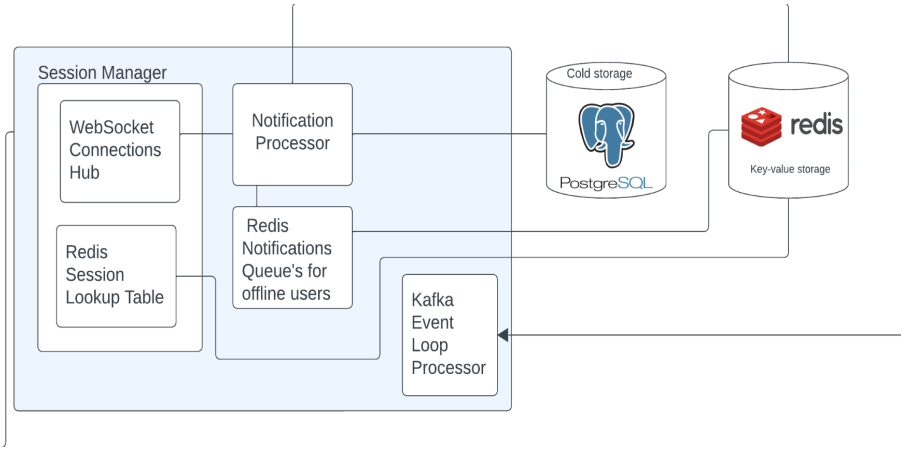


Figure 10: Συστατικά υπηρεσίας ειδοποιήσεων

Τα επιμέρους συστατικά είναι τα εξής:

WebSocket Hub

Πρόκειται για βαρυσήμαντο κομμάτι της εφαρμογής δεδομένου ότι αποτελεί το σημείο επαφής της υπηρεσίας των ειδοποιήσεων με τους χρήστες. Σε δικτυακό επίπεδο, αποθηκεύει τις πληροφορίες σύνδεσης (websockets) κάθε υπολογιστή-πελάτη-χρήστη της της εφαρμογής σε

μια in-memory δομή δεδομένων της εφαρμογής για άμεση ανταπόκριση. Ακόμα, φροντίζει να διατηρεί αυτές τις συνδέσεις ανοιχτές με μηνύματα ελέγχου όσο ο χρήστης αλληλεπιδρά και περιορίζει τις ανοιχτές συνδέσεις ανά χρήστη της εφαρμογής στις τρεις.

Session Store

Το session store είναι μια διεπαφή αποθήκευσης, η οποία αξιοποιεί την key-value βάση και δρα σε αρμονία με το WebSocket Hub. Αποθηκεύει σε ένα σύνολο, τη σύνοδο που εκκινεί η σύνδεση ενός χρήστη στην εφαρμογή για την παροχή ειδοποιήσεων και στο κλειδί με την ονομασία του χρήστη πληροφορίες για τη σύνδεση. Αποθηκεύει ακόμα πληροφορία για το χρόνο λήξης της συνόδου, την οποία αξιοποιεί το hub ώστε να μην υπάρχουν ανοιχτές συνδέσεις επ’ αόριστον. Σε κάθε αλληλεπίδραση με τον χρήστη, αυτός ο χρόνος ανανεώνεται.

Offline Queue Store

Η διεπαφή αυτή χειρίζεται ένα πλήθος λιστών για κάθε χρήστη της εφαρμογής. Εκεί αποθηκεύονται οι ειδοποιήσεις που προορίζονται για κάποιο χρήστη και είναι αδύνατο να καταλήξουν σε αυτόν δεδομένου ότι αυτός δεν είναι συνδεδεμένος. Έτσι αποθηκεύονται σε μια λίστα η οποία είναι χρονολογικά ταξινομημένη. Η εφαρμογή μπορεί να αφαιμάξει τις ειδοποιήσεις μία μία ή σε πακέτο.

Afora Store

Η διεπαφή αυτή περικλείει το μηχανισμό ο οποίος είναι υπεύθυνος για να κατευθύνει τις ειδοποιήσεις στους κατάλληλους χρήστες. Στον πυρήνα του είναι μια τομή μεταξύ δύο συνόλων. Κάθε ειδοποίηση όταν καταναλώνεται από το διαμοιραστή μηνυμάτων, αναλύεται η πληροφορία που φέρει ώστε να ενημερωθεί αυτή η βάση για το ποιους αφορά (εξού και το όνομα) η ειδοποίηση. Όταν λοιπόν επέλθει στο σύστημα μια πράξη από κάποιον χρήστη ως γεγονός, ενημερώνονται κατάλληλα οι εμπλεκόμενοι με τη βοήθεια αυτού του μηχανισμού. Αν κάποιος χρήστης δεν είναι συνδεδεμένος τη δεδομένη στιγμή, μέσω κατάλληλων μεθόδων δρομολογείται η αποθήκευση των ειδοποιήσεων ώστε να αναγνωσθούν με την επόμενη σύνδεση.

Cold Store

Όταν οι ειδοποιήσεις φτάσουν στην υπηρεσία από το διαμοιραστή μηνυμάτων, αποθηκεύονται αυτούσιες. Επίσης αποθηκεύεται σε ξεχωριστό πίνακα η σχέση των ενδιαφερόμενων χρηστών με τις αντίστοιχες ειδοποιήσεις, ώστε κάποια ειδοποίηση που αφορά πολλούς χρήστες να αποθηκεύεται μονάχα μια φορά και η κατάστασή της σχετικά με το αν έχει αναγνωστεί ή όχι να αφορά κάθε χρήστη ξεχωριστά (status: read or unread).

Λειτουργίες και ακολουθιακά διαγράμματα

Σύνδεση

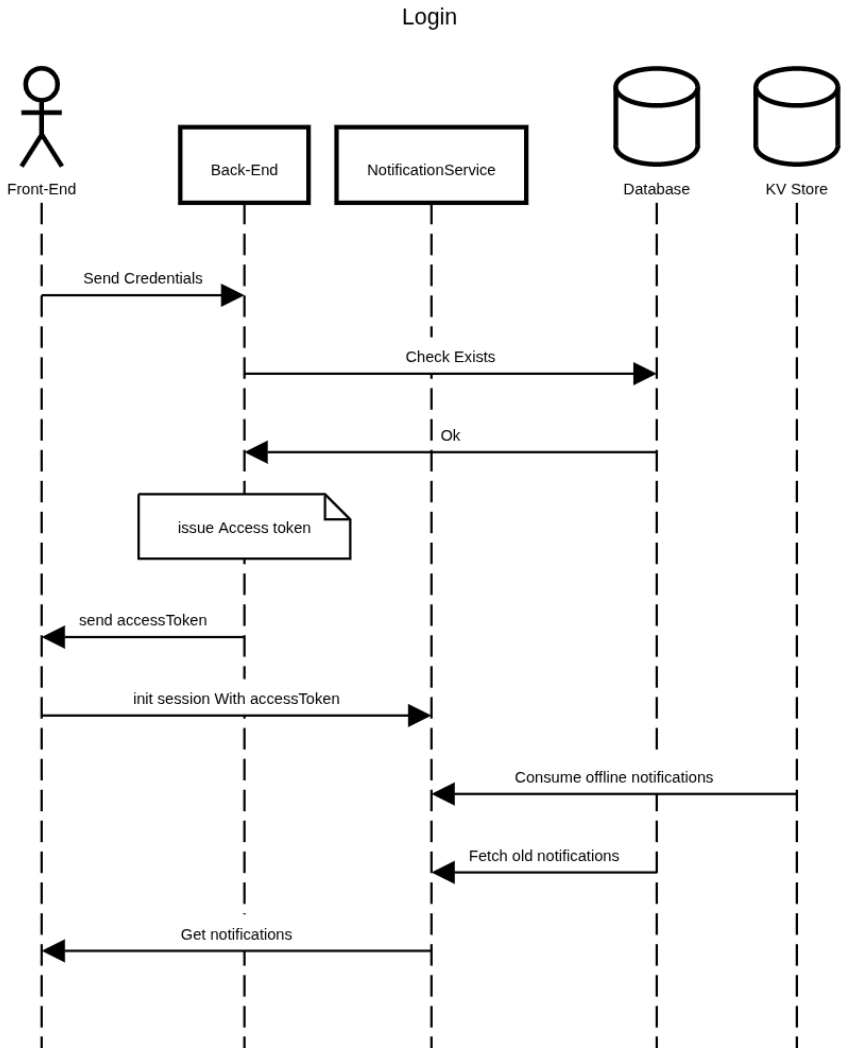


Figure 11: Ακολουθιακό διάγραμμα σύνδεσης χρήστη

Κατά τη σύνδεση ο χρήστης ενημερώνει το back-end με τα στοιχεία σύνδεσης, όπου η υπηρεσία τον πιστοποιεί ή όχι, ελέγχοντας τα δεδομένα με τη σχεσιακή βάση όπου και ζουν οι πληροφορίες χρηστών.
Εφόσον πιστοποιηθεί, δημιουργεί ένα τεκμήριο JWT (**J**son **W**eb **T**oken), το οποίο αποστέλλει σαν απάντηση ώστε αυτό να μπορεί να αξιοποιηθεί από το χρήστη προσθέτοντάς το σε κάθε HTTP αίτημα από εδώ κι ύστερα, ώστε να έχει πρόσβαση στις κατάλληλες μεθόδους και ενέργειες. Το τεκμήριο αυτό λέγεται Access Token, αφού παρέχει πρόσβαση στις υπηρεσίες και περιέχει πληροφορία όπως το αναγνωριστικό του χρήστη, τους ρόλους του και το αν είναι απλός χρήστης ή οργανισμός. Στο Front-End αποθηκεύεται στην εφαρμογή και διαρκεί στη μνήμη για διάστημα 15 λεπτών, οπότε λήγει.

Ταυτόχρονα, δημιουργείται ένα ακόμα τεκμήριο JWT το οποίο αποστέλλεται σαν cookie με την επιλογή `HTTPOnly`. Αυτή η επιλογή χρησιμοποιείται ώστε να μην μπορεί να αξιοποιηθεί το

Cookie από το φυλλομετρητή στο μέσω κώδικα JS. Η μόνη χρήση του είναι να διαβάζεται από το Back-End όταν το αρχικό τεκμήριο λήξει. Τότε αν ο χρήστης διαθέτει το απαραίτητο cookie, η σύνδεση επανεκτελείται αυτόματα χωρίς την ανάγκη να εισαχθούν τα στοιχεία σύνδεσης από την αρχή. Το τεκμήριο αυτό αποκαλείται *Refresh Token*.

Αποσύνδεση

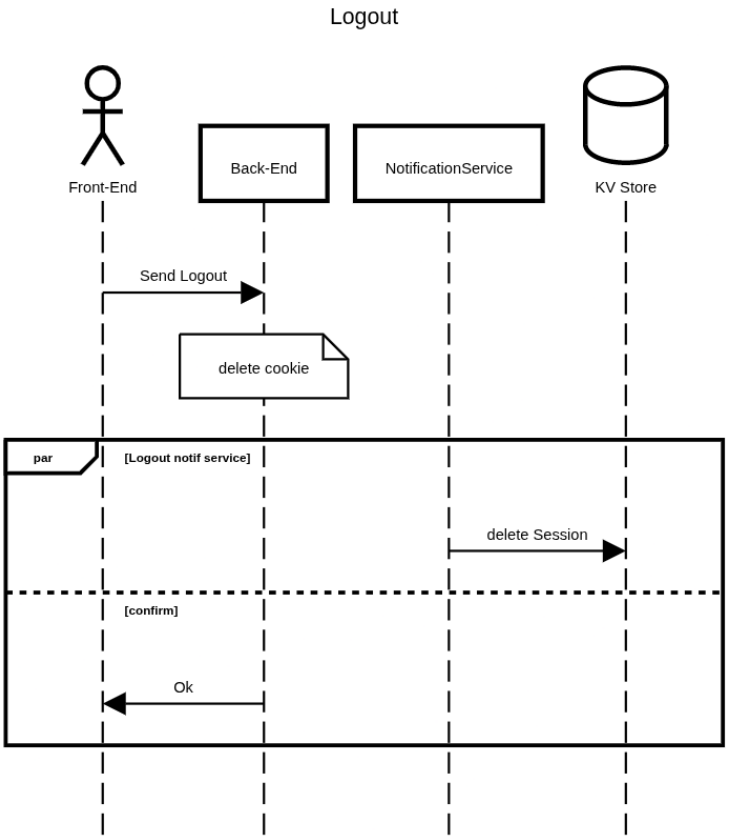


Figure 12: Ακολουθιακό διάγραμμα αποσύνδεσης χρήστη

Το γεγονός της αποσύνδεσης σηματοδοτεί τις εξής αλλαγές: αρχικά από την πλευρά του Front-End χρήστη, διαγράφεται το Access Token από τη μνήμη της εφαρμογής ώστε να μην μπορεί να αξιοποιηθεί. Επίσης, η υπηρεσία των ειδοποιήσεων διαγράφει τις πληροφορίες της συνόδου από την key-value βάση, κλείνει το κανάλι αμφίδρομης επικοινωνίας με το Front-End (δηλαδή το WebSocket σύνδεση), διαγράφοντας τελείως τη σύνδεση από το WS Hub. Στο τέλος, η κύρια υπηρεσία διαγράφει το cookie από την επικοινωνία του Front-end χρήστη, ούτως ώστε αυτός να μην μπορεί να ανανεώσει το Access Token με σκοπό την επανασύνδεση.

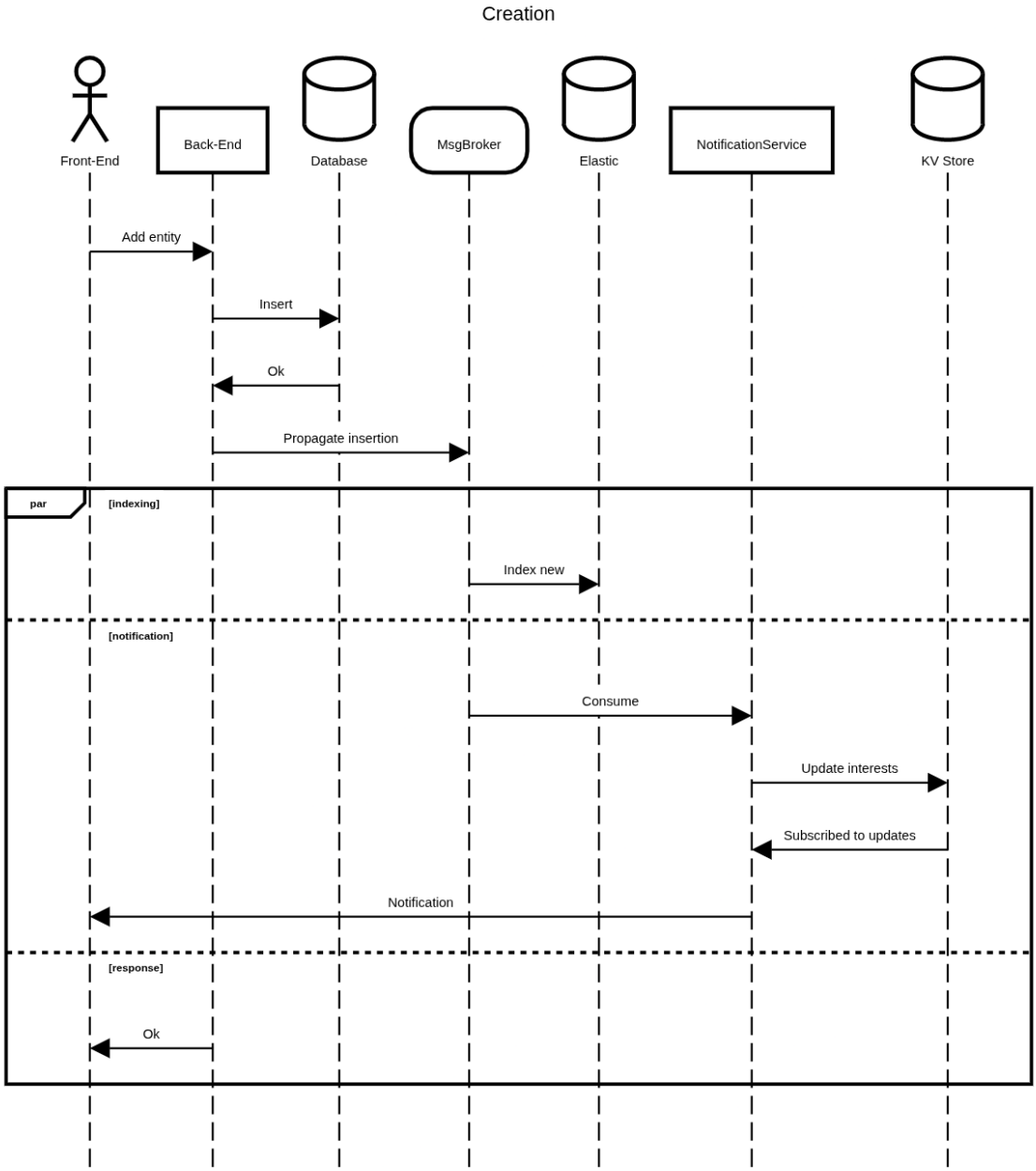


Figure 13: Ακολουθιακό διάγραμμα δημιουργίας οντότητας

Στο παραπάνω ακολουθιακό διάγραμμα αναλύεται η λειτουργία δημιουργίας οποιασδήποτε οντότητας στην εφαρμογή και πώς ενημερώνονται κατάλληλα τα συστατικά της εφαρμογής.

Αρχικά η πράξη εκκινείται από κάποιον χρήστη μέσω του Front-End. Εφόσον γίνουν οι κατάλληλοι έλεγχοι για την εγκυρότητα των νέων δεδομένων, και έλεγχοι που να διατηρούν την ακεραιότητα της εφαρμογής, τροποποιείται η βάση δεδομένων με την καινούρια πληροφορία. Προτού το αποτέλεσμα της ενέργειας φτάσει στο χρήστη, οι αλλαγές διοχετεύονται στο διαμοιραστή των μηνυμάτων. Από εκεί καταναλώνονται από την υπηρεσία των ειδοποιήσεων και την υπηρεσία της αναζήτησης, ώστε να είναι ενήμερες και να προβούν στις απαραίτητες ενέργειες.

Κεφάλαιο 4

Σε αυτό το κεφάλαιο προχωρούμε σε εις βάθος ανάλυση των τριών υπηρεσιών που παρουσιάστηκαν στο προηγούμενο κεφάλαιο, μεταβαίνοντας από τη γενική αρχιτεκτονική απεικόνιση σε μια πιο λεπτομερή μελέτη της εσωτερικής τους συμπεριφοράς. Με τη χρήση του Apache JMeter ως κύριο εργαλείο δοκιμών φόρτου, προσομοιώνουμε ρεαλιστικά μοτίβα κίνησης και αξιολογούμε την απόδοση του συστήματος μέσω διαγραμμάτων που αποτυπώνουν την απόδοση, την καθυστέρηση και τα ποσοστά σφαλμάτων.

Παράλληλα, πραγματοποιείται εμπεριστατωμένη ανάλυση του κώδικα του παρασκηνίου και του προσκηνίου. Ο συνδυασμός αυτής της εμπειρικής και ενδοσκοπικής προσέγγισης μας παρέχει μια ολοκληρωμένη εικόνα της λειτουργίας κάθε υπηρεσίας, θέτοντας τα θεμέλια για στοχευμένη βελτιστοποίηση.

Κύρια Υπηρεσία

Go

Η γλώσσα Go, αναπτύχθηκε από την Google και παρουσιάστηκε για πρώτη φορά το 2009. Δημιουργήθηκε με στόχο να προσφέρει την απλότητα άλλων “εύπεπτων” γλωσσών όπως η Python, σε συνδυασμό με την υψηλή απόδοση και τον έλεγχο μνήμης (έως ένα βαθμό) και συστήματος που προσφέρει η C. Η φιλοσοφία της Go βασίζεται στη λιτότητα και στην καθαρότητα: αποφεύγει την πολυπλοκότητα και επιβάλλει ξεκάθαρους κανόνες συγγραφής και δομής κώδικα. Είναι ένας τύπος γλώσσας που βιβλιογραφικά απαντάει στον όρο opinionated (ή δογματική), λόγω του περιορισμένου εύρους επιλογών που δίνονται στον προγραμματιστή να εκτελέσει ένα έργο.

Από τεχνικής άποψης, η Go διαθέτει ταχύτατο compiler, συγκρίσιμη απόδοση με C/C++, και ένα ενσωματωμένο σύστημα ταυτοχρονισμού με goroutines που την καθιστά εξαιρετικά αποδοτική για εφαρμογές δικτύου, εξυπηρετητές και μικροϋπηρεσίες. Η διαχείριση μνήμης γίνεται μέσω garbage collection, αλλά με έμφαση στην απόδοση και την προβλεψιμότητα Παράλληλα, η Go ενσωματώνει σημαντικά χαρακτηριστικά όπως διεπαφές (interfaces), πρότυπο σύστημα πακέτων, και στατικούς τύπους, χωρίς όμως να φορτώνει τον προγραμματιστή με υπερβολική συντακτική πολυπλοκότητα ή κληρονομικότητα.

Η Go έχει καθιερωθεί ως βασικό εργαλείο στον κόσμο του cloud computing, με εφαρμογές σε Kubernetes, Docker, Terraform και πολλές άλλες υποδομές. Η απλότητα του deployment (ένα μόνο binary), η σταθερότητα του runtime και η ευκολία debugging την καθιστούν ιδανική για παραγωγικά περιβάλλοντα υψηλών απαιτήσεων. Εταιρείες όπως η Google, Uber, Dropbox και Cloudflare βασίζονται σε αυτή για την ανάπτυξη αξιόπιστων και κλιμακούμενων υπηρεσιών.

Δομές (structs)

Στην Go, η αναπαράσταση των οντοτήτων στην εφαρμογή γίνεται με τη χρήση δομών. Οι δομές αποτελούν συλλογές μεταβλητών πρωτογενών ή μη τύπων και τα πεδία δηλώνονται με τη μορφή *Όνομα Τύπος Ετικέτες*. Οι ετικέτες αποτελούν βοηθητικά πεδία τα οποία χρησιμεύουν κυρίως κατά την αποκωδικοποίηση και κωδικοποίηση της δομής σε διάφορα εξωτερικά και μη, βοηθητικά προγράμματα.

Στην παρακάτω περίπτωση έχουμε τη δομή μιας δωρεάς, Όπως φαίνεται, το κάθε πεδίο έχει δύο ετικέτες db, json. Η τιμή της ετικέτας db καθορίζει το πώς συνδιαλέγεται το πρόγραμμα-οδηγός της γλώσσας με μια σύνδεση βάσης δεδομένων. Συγκεκριμένα, κατά την εγγραφή από το πρόγραμμα της εφαρμογής προς τη βάση, αντιστοιχίζει την τιμή και τον τύπο του πεδίου π.χ. ID με την αντίστοιχη στήλη ID σε κάποιον πίνακα της βάσης. Κατά την αντίθετη διαδρομή, δηλαδή το διάβασμα των δεδομένων από τη βάση προς την εφαρμογή, η τιμή της στήλη ID θα εναποτεθεί στο πεδίο ID ανεξάρτητα από τη σειρά που διαβάζεται από τη βάση.

Αντίστοιχα, η ετικέτα json διαθέτει τη δική της ονοματοδοσία για τη μετατροπή των πεδίων της δομής από/σε json αντικείμενα. Επιπρόσθετα, διαθέτει την περιγραφή *omitempty,* κατά την οποία πληροφορεί τα κομμάτια κώδικα υπεύθυνα για τη μετατροπή από δομή σε json, να παραλείψουν τα πεδία που δεν έχουν κάποια τιμή αν έχουν τύπο δομής, ή για αυτά με μηδενική τιμή αν έχουν κάποιο πρωτογενή τύπο.

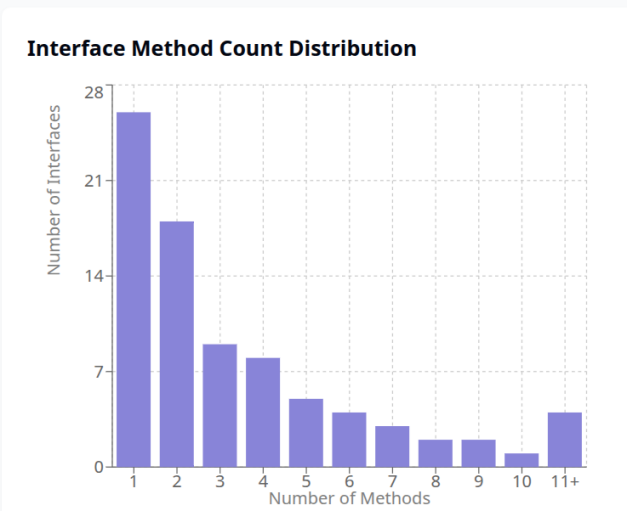
```
type Donation struct {
	ID         int64        `db:"id"           json:"id,omitempty"`
	CreatedAt  *pq.NullTime `db:"created_at"   json:"created_at,omitempty"`
	Message    string       `db:"message"      json:"message,omitempty"`
	Status     string       `db:"status"       json:"status,omitempty"`
	Unique     bool         `db:"is_unique"    json:"is_unique,omitempty"`
	Anonymous  bool         `db:"anonymous"    json:"anonymous,omitempty"`
	DateReveal *pq.NullTime `db:"date_reveal"  json:"date_reveal,omitempty"`
	RequestId  int64        `db:"request_id"   json:"request_id,omitempty"`
	HostID     int64        `db:"host_id"      json:"host_id,omitempty"`
	HostType   string       `db:"host_type"    json:"host_type,omitempty"`
	MetadataHolder
}
```

Κώδικας 1: Δωρεά

Πολυμορφισμός με interfaces

Η έννοια του πολυμορφισμού στην Go διαφέρει από την προσέγγιση των αντικειμενοστραφών γλωσσών, καθώς η γλώσσα δεν είναι αντικειμενοστραφής. Στην Go, η θεμελιώδης δομή, το λειτουργικό ανάλογο του αντικειμένου είναι οι δομές οι οποίες ορίζονται ως: type StructName struct. Μέσω της δομής, η γλώσσα μας δίνει τη δυνατότητα να ενοποιήσουμε πολλές μεταβλητές υπό ένα κοινό σκοπό, ταυτόχρονα με ένα σύνολο από λειτουργίες. Στις αντικειμενοστραφείς γλώσσες, η λειτουργικότητα ενός αντικειμένου, μπορεί να οριστεί από τις μεθόδους αυτού, αλλά και από μεθόδους οι οποίες κληρονομούνται από γονικά αντικείμενα. Στην Go, ως είθισται η κατάσταση είναι πιο απλή, καθώς μια διαμοιρασμένη λειτουργικότητα ορίζεται από τις διεπαφές ή interfaces. Τα interfaces είναι πρακτικά ένα σύνολο υπογραφών μεθόδων. Μια δομή λέμε ότι υλοποιεί το interface, αν υλοποιεί κάθε μία από τις μεθόδους του, συμφωνώντας κατά γράμμα τόσο στις παραμέτρους και τους τύπους τους, αλλά και στο σύνολο των τιμών (και τους τύπους τους) που επιστρέφουν. Η γλώσσα εσωτερικά τα χρησιμοποιεί κατά κόρον, εφόσον συμπυκνώνουν και οριοθετούν μια λειτουργία, χωρίς να είναι δεσμευτικά ως προς την υλοποίηση.

Η συνήθης πρακτική, η οποία ακολουθείται και στην *κανονική βιβλιοθήκη*, είναι οι διεπαφές να έχουν μικρό αριθμό μεθόδων. Αυτό είναι σημαντικό, καθώς όσο αυξάνει ο αριθμός των απαιτούμενων μεθόδων, τόσο αυξάνει και η πολυπλοκότητα που απαιτείται ώστε κάποια δομή να ικανοποιεί τη διεπαφή. Επίσης, δεδομένου ότι οι διεπαφές μπορούν να συντεθούν, είναι προτιμότερο να διαθέτουν λίγες μεθόδους και άρα πιο σαφή λειτουργία.



Graph 1: Πλήθος μεθόδων ανά interface στην κανονική βιβλιοθήκη

Στην εφαρμογή, χρησιμοποιούνται σε σημεία όπου η υλοποίηση διαφέρει ανά οντότητα. Για παράδειγμα, κατά τη δημιουργία οποιασδήποτε οντότητας, οι φωτογραφίες αποθηκεύονται σε ξεχωριστή βάση (αυτή για τα πολυμέσα). Επίσης για κάθε οντότητα, είναι αναγκαία η εξαγωγή των φωτογραφιών της, ώστε να αναρτηθούν. Για αυτό το λόγο έχει οριστεί η παρακάτω διεπαφή:

```
type PhotoUploadable interface {
     GetPhotos() []Photo
}
```

Κώδικας 2: Διεπαφή με φωτογραφίες

η οποία μπορεί στις απλές οντότητες να υλοποιείται με τον παρακάτω απλό τρόπο,

```
func (e Event) GetPhotos() []Photo {
     return e.Photos
}
```

Κώδικας 3: Απλή εξαγωγή φωτογραφιών

ωστόσο σε πιο σύνθετες δομές, όπως είναι μια συλλογή αντικειμένων η υλοποίηση διαφέρει, δεδομένου ότι συλλέγονται όλες οι φωτογραφίες των αντικειμένων υπό μια κοινή συλλογή.

```
func (iwq ItemsWithQuantity) GetPhotos() []Photo {
      ps := make([]Photo, 0)
      for _, item := range iwq {
            for _, p := range item.GetPhotos() {
                  ps = append(ps, p)
            }
      }
      return ps
}
```

Κώδικας 4: Σύνθετη εξαγωγή φωτογραφιών

Διεπαφές για δοσοληψίες στη σχεσιακή βάση

Ένα ακόμη παράδειγμα που αναδεικνύει τη χρησιμότητα και την ευκολία στη χρήση των διεπαφών είναι το παρακάτω.

Κατά τη δημιουργία νέων εγγραφών στη βάση ήταν αναγκαίο σε κάποιες περιπτώσεις να χρησιμοποιηθούν δοσοληψίες. Η δοσοληψία είναι ένα σύνολο ενεργειών το οποίο εκτελείται απομονωμένα από το σύστημα διαχείρισης της βάσης, εξασφαλίζοντας τη συνέπεια σε περιπτώσεις σφάλματος κατά την εκτέλεσή τους. Όταν ολοκληρωθεί η δοσοληψία, τότε ο εντολέας επικυρώνει το τέλος της και το αποτέλεσμά της είναι πλέον φανερό στα δεδομένα.

Σαφώς, μια πράξη η οποία αποτελεί μέρος μιας δοσοληψίας δύναται να υφίσταται και εκτός αυτής. Δεδομένου επίσης ότι ο μηχανισμός απαιτεί επιπρόσθετη πολυπλοκότητα στην υλοποίησή του, θα ήταν επιθυμητό να υπήρχε ένας τρόπος που να διευκολύνει την εκτέλεση μιας ενέργειας τόσο υπό τα συμφραζόμενα μιας δοσοληψίας, όσο και εκτός αυτής.

Βάσει των παραπάνω ορίστηκε η διεπαφή Executor:

```
type Executor interface {
      Exec( query string, args ...any) (sql.Result, error)
      QueryRowxContext(ctx context.Context, query string,
args ...any) *sqlx.Row
      QueryxContext(ctx context.Context, query string, args ...any)
(*sqlx.Rows, error)
      SelectContext(ctx context.Context, dest any, query string,
args ...any) error
      BindNamed(query string, arg any) (string, []any, error)
}
```

Κώδικας 5: Διεπαφή εκτελεστή

Η διεπαφή δανείζεται για τις μεθόδους την ονοματοδοσία από μεθόδους της κανονικής βιβλιοθήκης της γλώσσας για σύνδεση με σχεσιακή βάση. Αυτό έγινε εσκεμμένα ώστε οι δύο παρακάτω τύποι (απλής εκτέλεσης και εκτέλεσης με δοσοληψία), να ικανοποιούν τη διεπαφή Executor χωρίς προσθήκες, και άρα να εναλλάσσονται ως εκτελεστές, κατά τη διενέργεια μιας πράξης στη βάση.

```
type DB struct {...} // απλή εκτέλεση
type Tx struct {...} // εκτέλεση με δοσοληψία
```

Κώδικας 6: Εκτελεστές εντολών σχεσιακής βάσης

Οι συναρτήσεις λοιπόν που συνδιαλέγονται με τη βάση αρχικά λάμβαναν σαν όρισμα το ίδιο το αντικείμενο της βάσης ως παρακάτω:

```
func (s *Storage) CreateDonation(..., db *sql.DB) error { ... }
```

ενώ πλέον λαμβάνουν σαν όρισμα μια διεπαφή τύπου Executor:

```
func (s *Storage) CreateDonation(..., exec Executor) error { ... }
```

Συνεπώς με τη βοήθεια της παραπάνω διεπαφής, ορίζεται η συνάρτηση `WithTx,` η οποία περιτυλίγει τη λογική με δοσοληπτική λογική.

```
func (s *Storage) WithTx(ctx context.Context, fn func(Executor) error)
error {
      tx, err := s.SQL.DB.BeginTxx(ctx, nil) // εκκίνηση δοσοληψίας
      if err != nil {
            return err
      }
       …
      err = fn(tx) // εκτέλεση συνάρτησης υπό δοσοληπτικά συμφραζόμενα
      if err != nil {
            tx.Rollback()
            return err
      }
      return tx.Commit() // επικύρωση δοσοληψίας
}
```

Κώδικας 7: Εκτέλεση με δοσοληψία

Ταυτόχρονη εκτέλεση

Η Go παρέχει εγγενώς τη δυνατότητα για απλή και εύκολη ταυτόχρονη εκτέλεση μέσω των τύπων των Goroutines, των καναλιών και των WaitGroups.

Οι **goroutines** είναι ένα βασικό χαρακτηριστικό της γλώσσας προγραμματισμού Go που επιτρέπει την ταυτόχρονη εκτέλεση συναρτήσεων. Μια goroutine δημιουργείται με το keyword *go* πριν από μια συνάρτηση, και εκτελείται ανεξάρτητα από την κύρια ροή του προγράμματος. Οι goroutines δεν είναι τα συμβατικά νήματα (threads), αλλά προσδιορίζονται καλύτερα από τον όρο *Green Thread*, δεδομένου ότι τα διαχειρίζεται η μονάδα χρονοπρογραμματισμού της γλώσσας ελαφριές (Go runtime scheduler), και μπορούν να κλιμακώνονται σε χιλιάδες χωρίς το βάρος ενός παραδοσιακού νήματος.

Τα **κανάλια (channels)** είναι ο τρόπος με τον οποίο επικοινωνούν οι goroutines μεταξύ τους, χωρίς να απαιτείται κοινή μνήμη ή χρήση mutexes. Η προσέγγιση της γλώσσας στην επικοινωνία μεταξύ ταυτόχρονων διεργασιών/νημάτων αντιπαρατίθεται εννοιολογικά στην παραδοσιακή προσέγγιση γλωσσών όπως οι C και C++. Η διαφορά συμπυκνώνεται στη φράση *“Don’t communicate by sharing memory, share memory by communicating”* [7]*.* Κοινώς, η γλώσσα αποφεύγει να εκθέτει δεδομένα (μνήμη) ταυτόχρονα σε δύο πελάτες, όταν αυτοί επιθυμούν τη μεταξύ τους επικοινωνία. Η επικοινωνία όμως επιτυγχάνεται με το διαμοιρασμό της κυριότητας μιας μεταβλητής, όταν αυτή επικοινωνείται μέσα από τα κανάλια. Τελικώς, τα κανάλια επιτρέπουν την αποστολή και λήψη τιμών μεταξύ goroutines με ασφάλεια. Υπάρχουν κανάλια οριοθετημένα από άποψη χωρητικότητας και μη οριοθετημένα (buffered και unbuffered), και η αποστολή ή λήψη μπλοκάρει μέχρι η άλλη πλευρά να είναι έτοιμη, γεγονός που διευκολύνει το συγχρονισμό.

Τέλος, τα **WaitGroups** παρέχουν έναν τρόπο συγχρονισμού πολλαπλών goroutines. Με τη χρήση της δομής *sync.WaitGroup*, μπορούμε να περιμένουμε να ολοκληρωθούν όλες οι goroutines που έχουν ξεκινήσει, πριν συνεχίσουμε την εκτέλεση του κυρίως προγράμματος, κάτι αντίστοιχο με τον όρο *join* που συναντάται στα συμβατικά νήματα. Ο προγραμματιστής καθορίζει πόσες goroutines περιμένει (μέσω της *waitgroup.Add*), και κάθε goroutine καλεί *watigroup.Done* όταν τελειώσει. Η κύρια ρουτίνα καλεί *waitgroup.Wait* για να μπλοκάρει μέχρι όλες οι goroutines ολοκληρώσουν το έργο τους. Αυτός ο μηχανισμός είναι κρίσιμος για σωστή διαχείριση ταυτόχρονης εκτέλεσης χωρίς race conditions.

Στην εφαρμογή γίνεται χρήση των παραπάνω σε διάφορα σημεία εκ των οποίων θα αναλύσουμε δύο. Το ένα παρουσιάζεται εδώ ως μέρος της κύριας υπηρεσίας και το άλλο στο υποκεφάλαιο της υπηρεσίας ειδοποιήσεων:

Κοινοποίηση μηνυμάτων στο διαμοιραστή μηνυμάτων

Όπως αναφέρθηκε, η δημιουργία μιας νέας οντότητας, είναι αναγκαίο να γνωστοποιηθεί από την κύρια υπηρεσία στις υπόλοιπες δύο. Η υπηρεσία αναζήτησης θέλει αυτή την πληροφορία, ώστε να δεικτοδοτήσει τη νέα οντότητα ώστε να είναι έτοιμη προς αναζήτηση. Η υπηρεσία ειδοποιήσεων θέλει αυτή την πληροφορία, ώστε να καταχωρήσει τον δημιουργό τους ως έναν ενδιαφερόμενο για τυχόν ενέργειες που σχετίζονται με αυτή και να τον ενημερώσει όταν έρθει η ώρα. Πρακτικά, λοιπόν χρειάζεται ένα κομμάτι κώδικα το οποία θα λαμβάνει τη δημιουργημένη οντότητα και θα την προωθεί στο διαμοιραστή μηνυμάτων.
Αυτό συμβαίνει ως εξής. Κατά την εκκίνηση της κύριας υπηρεσίας, δημιουργείται μια goroutine, η οποία περιμένει εξελίξεις στο κανάλι *KafkaMessages.* Κατά τη δημιουργία της οντότητας, οι μέθοδοι που είναι υπεύθυνες για αυτήν κάθε φορά, γνωστοποιούν στο κανάλι αυτό το γεγονός. Η διαδικασία φαίνεται στο παρακάτω σχήμα.

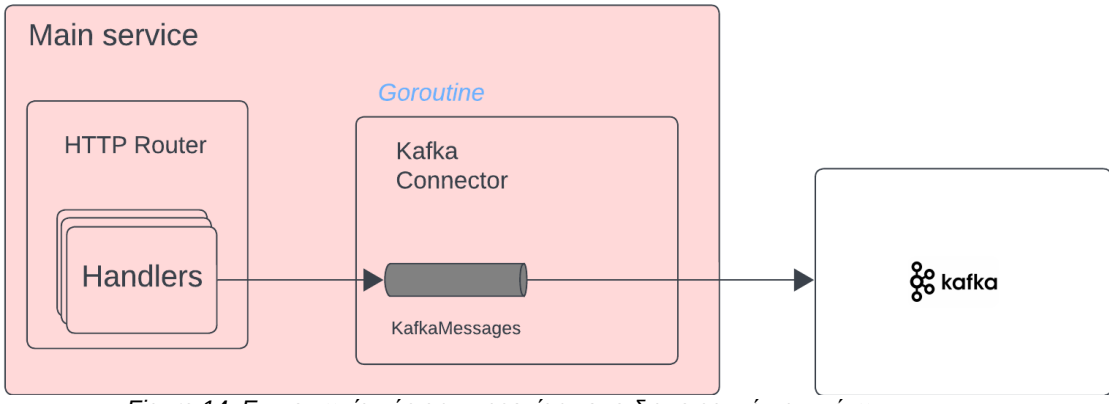


Figure 14: Επικοινωνία κύριας υπηρεσίας με το διαμοιραστή μηνυμάτων

Περιφερειακή επικοινωνία

Στο σκελετό της κύριας υπηρεσίας πραγματοποιούνται συνδέσεις με τα περιφερειακά συστήματα επικοινωνίας και δεδομένων.

Redis Cache
Η σύνδεση με τη βάση γίνεται μέσω της βιβλιοθήκης για το Valkey. Το Valkey είναι μια δωρεάν open source εναλλακτική, η οποία δημιουργήθηκε από την κοινότητα όταν το Redis άλλαξε την άδεια παροχής του ως λογισμικό. Πρακτικά, η επιλογή έγινε γιατί αυτή η βιβλιοθήκη υλοποιεί εγγενώς τη λειτουργία.

Το *Cache-Aside* είναι ένα διαδεδομένο μοτίβο διαχείρισης cache. Χρησιμοποιήθηκε για να βελτιώσει την απόδοση των αναγνώσεων από τη σχεσιακή βάση δεδομένων. Αξιοποιώντας το μηχανισμό αυτό, η ίδια η εφαρμογή είναι υπεύθυνη για τον συγχρονισμό μεταξύ cache και της σχεσιακής βάσης. Λειτουργεί ως εξής:

Όταν αιτείτα ένας πόρος:

1. Ελέγχεται πρώτα η cache, η οποία χρησιμοποιεί το full URL του πόρου ως κλειδί
2. Αν το δεδομένο υπάρχει στην cache (cache hit), το επιστρέφει αμέσως.
3. Αν λείπει (cache miss), τότε:
    - Το ανακτά από τη σχεσιακή βάση
    - Το προσθέτει στην cache για μελλοντική χρήση.
    - Και το επιστρέφει στον χρήστη.

Αυτό χρησιμοποιείται κυρίως σε πόρους οι οποίοι διαβάζονται πολύ συχνά και δεν τροποποιούνται πολύ συχνά. Τέτοιου είδους δεδομένα είναι οι εκδηλώσεις, οι καμπάνιες και τα αιτήματα.

Οι υπόλοιπες τρεις συνδέσεις, συγκεκριμένα για *PostgresQL DB, Kafka Message Broker, Azure Blob Storage* υλοποιούνται με τις ενδεδειγμένες βιβλιοθήκες.

API και Router

Ο HTTP Router αξιοποιεί την κανονική βιβλιοθήκη της γλώσσας. Το API είναι μια βοηθητική δομή η οποία περικλείει δομές για όλες τις λειτουργικότητες της εφαρμογής. Ο Router δέχεται το API σαν όρισμα.

```
type API struct {
      Storage       *Storage
      Logger        *zerolog.Logger
      kafkaProducer *kafka.Producer
      KafkaMessages chan KafkaMessage
}

type Storage struct {
      SQL *SQL
      AZ  *AZ
}

type SQL struct {
      Dialect goqu.DialectWrapper
      DB      *sqlx.DB
}

type AZ struct {
      Client *azblob.Client
}
```

Κώδικας 8: Κύριες λειτουργικές δομές

Οι μέθοδοι - χειριστές των HTTP αιτημάτων εφαρμόζουν στο API ως receivers (υποδοχείς στη γλώσσα Go). Μέσω αυτών γίνονται διαθέσιμες οι υπο-δομές στις συναρτήσεις και δεδομένου ότι ως υποδοχέας δηλώνεται ένας δείκτης σε δομή API και όχι σκέτη η δομή, τότε σε κάθε συνάρτηση γίνονται διαθέσιμες οι ίδιες δομές μέσω αυτής της αναφοράς. Σε περίπτωση που η υποδοχή γινόταν χωρίς δείκτη, θα είχαμε αντίγραφο της δομής. Δηλαδή:

```
// * marks pointer receiver
func (a *API) AddDonation(w http.ResponseWriter, r *http.Request)
error{...}
// Value receiver
func (a API) AddDonation(w http.ResponseWriter, r *http.Request)
error{...}
```

Κώδικας 9: Receiver

Αντίστοιχα, οι μέθοδοι επικοινωνίας με τη βάση εφαρμόζονται στη δομή Storage. Αυτή η δομή περιέχει τις ύπο-δομές SQL και AZ. Η AZ είναι ένα περιτύλιγμα γύρω από τη δομή της Azure βιβλιοθήκης για τη σύνδεση με τη βάση πολυμέσων. Αυτό έγινε για να εφαρμοστεί πάνω της

μια μέθοδος για ανέβασμα των φωτογραφιών, δεδομένου ότι η γλώσσα δεν επιτρέπει method receivers σε δομές εξωτερικών πακέτων.

Η δομή SQL, ωστόσο, περιέχει δύο ύπο-δομές:

DB
Αποτελεί τον τύπο για επικοινωνία με τη σχεσιακή βάση.

Dialect
Κατά την ανάπτυξη της εφαρμογής, κρίθηκε αναγκαίο σε κάποιες περιπτώσεις τα ερωτήματα στη σχεσιακή βάση να δημιουργούνται δυναμικά βάσει των τιμών κάποιων μεταβλητών. Αυτό γίνεται αντιληπτό στα ερωτήματα γενικής καταγραφής (λίστες) κάποιας οντότητας, όπου δίνεται η δυνατότητα ο χρήστης να επιλέξει φίλτρα. Εκεί παρουσιάστηκαν δύο επιλογές. Η μία ήταν να γίνει δυναμικό χτίσιμο του ερώτησης, όντας αυτή μια απλή συμβολοσειρά. Σε αυτή την επιλογή ενώ φαίνεται απλούστερη από την εναλλακτική, ενέχει ο κίνδυνος της δυσκατανόησης του κώδικα. Η δεύτερη επιλογή ήταν η χρήση κάποιου δυναμικού χτίστη ερωτημάτων (query builder).

Η δομή *Dialect* αποτελεί τύπο ενός πακέτου χτισίματος δυναμικών ερωτήσεων για τη σχεσιακή βάση. Οι πίνακες, οι στήλες και οι υπόλοιποι όροι ενός ερωτήματος αναπαρίστανται με μεταβλητές, παρέχοντας έτσι τη δυνατότητα στον προγραμματιστή να χτίσει την ερώτηση αξιοποιώντας τις δομές ελέγχου της γλώσσας. Στην πράξη φαίνεται η αξία της.

```
// Παράδειγμα χρήσης strings.Builder
values := []string{"a", "b", "c"}
var sb strings.Builder
sb.WriteString(`SELECT * FROM "test" WHERE ("d" IN (`)
for i, val := range values {
      sb.WriteString(fmt.Sprintf("'%s'", val))
      if i < len(values)-1 {
            sb.WriteString(", ")
      }
}
sb.WriteString("))")
```

Κώδικας 10: String builder μέθοδος χτισίματος επερωτήσεως σχεσιακής βάσης

ενώ:

```
// Παράδειγμα χρήσης goqu Query Builder
query, _, _ := goqu.
    From("test").
    Where(goqu.Ex{"d": []string{"a", "b", "c"}}).ToSQL()
```

Κώδικας 11: Μέθοδος χτισίματος επερωτήσεως σχεσιακής βάσης με χρήση goqu

Δηλαδή αξιοποιούνται πλήρως οι τύποι της γλώσσας για το σχηματισμό ερωτημάτων.

HTTP POST Αιτήματα

Στα αιτήματα POST ο χρήστης αποστέλλει δεδομένα στο σώμα του αιτήματος (request body), εσκεμμένα, δοθέντος ότι είναι ευαίσθητα και αποκλείονται από το url του πόρου. Τα δεδομένα αυτά είναι υπό τη μορφή form-data [8]. Για την ορθή επεξεργασία και αξιοποίηση τους έχουν οριστεί για κάθε οντότητα ξεχωριστά οι παρακάτω μέθοδοι του αναλυτή και του επικυρωτή.

Αναλυτής αιτήματος (parser)

Διεπαφή Parsable

```
type Parsable interface {
	ParseFormData(any) error
}
```

Κώδικας 12: Διεπαφή που επιδέχεται ανάλυση

Ο αναλυτής λαμβάνει την υποχρέωση να μετατρέψει τα εισερχόμενα δεδομένα σε μορφή αξιοποιήσιμη από την εφαρμογή, κυρίως σε κάποια δομή οντότητας. Για παράδειγμα για τις εκδηλώσεις:

```
func (e *Event) ParseFormData(f *multipart.Form) error {
	errors := make(ParseValidateErrors)

	for name, v := range f.Value {
		switch name {
		    …
		case "event.start_date": // date parse
			val, err := strconv.ParseInt(v[0], 10, 64)
			if err != nil {
				errors[name] = fmt.Errorf("invalid %q %q",
name,
                                                  v[0])
			}
			e.StartDate = &pq.NullTime{Time: time.Unix(val,
0),
                                                Valid: true}
			case "event.campaign_id":
			if v[0] == "null" || v[0] == "undefined" {
				e.CampaignID = nil
			} else {
				val, err := strconv.ParseInt(v[0], 10, 64)
				if err != nil {
					errors[name] = fmt.Errorf("invalid %q
%q",
                                                        name,
v[0])
				}
				e.CampaignID = &val
			}
```

```
            case "event.location": // location json data parse
                  loc := &Location{}
                  if err := json.Unmarshal([]byte(v[0]), loc);
                       err != nil {
                        errors[name] = err
                          continue
                  }
                  e.Location = *loc
            default:
                  errors[name] = fmt.Errorf("unknown field %q",
name)
            }
      }

      photos, err := FileHeaderToPhoto(f.File["photos"])
      if err != nil {
            errors["photos"] = err
      } else {
            e.Photos = photos
      }

      if len(errors) > 0 {
            return errors
      }

      return ni
}
```

Όπως φαίνεται πέρα από τα πεδία κειμένου επεξεργάζεται και τα αρχεία, τις φωτογραφίες εν προκειμένω, σε ξεχωριστή συνάρτηση.

Επικυρωτής αιτήματος (validator)

Διεπαφή Validatable

```
type Validatable interface {
      Validate() error
}
```

Κώδικας 13: Διεπαφή που επιδέχεται επικύρωση

Ο επικυρωτής είναι η μέθοδος η οποία αποφαίνεται για το αν οι τιμές που εστάλησαν από το χρήστη ακολουθούν διάφορους λογικούς κανόνες, όπως για παράδειγμα τα αντικείμενα να μην έχουν κενό όνομα ή η ποσότητά τους να μην είναι μικρότερη του μηδενός. Η διαδικασία αυτή συμβαίνει ακριβώς μετά από την ανάλυση του αιτήματος

```
func (d *DonationWithProvisions) Validate() error {
              ...
      if len(d.Items) == 0 && len(d.Services) == 0 {
            errors["provisions"] = fmt.Errorf("must contain some
items or services")
      }
      if d.Anonymous {
            if !d.DateReveal.Valid {
                  errors["dateReveal"] = fmt.Errorf("invalid date")
            } else {
                  if d.DateReveal.Time.
                        Before(time.Now().Add(time.Minute)) {
                         errors["dateReveal"] = fmt.Errorf("too
early")
                  }
            }
      }
              ...
      for i, item := range d.Items {
            err := item.Validate()
            if err != nil {
                  errors[fmt.Sprintf("item-%d", i)] = err
            }
      }
      ...
}
```

Κώδικας 14: Επικύρωση

Δοκιμές υπό φορτίο

Για τις παρακάτω δοκιμές χρησιμοποιήθηκε το εργαλείο Apache JMeter. Με το JMeter μπορεί κανείς να προσομοιάσει υψηλό HTTP φόρτο κατευθυνόμενο προς κάποια εφαρμογή ώστε να δει πως ανταποκρίνονται τα συστατικά της στις διακυμάνσεις των απαιτήσεων.

Αρχική παρατήρηση
Η προκαθορισμένη βιβλιοθήκη για τη σύνδεση με σχεσιακή βάση που χρησιμοποιήθηκε διαθέτει την εξής ρύθμιση.

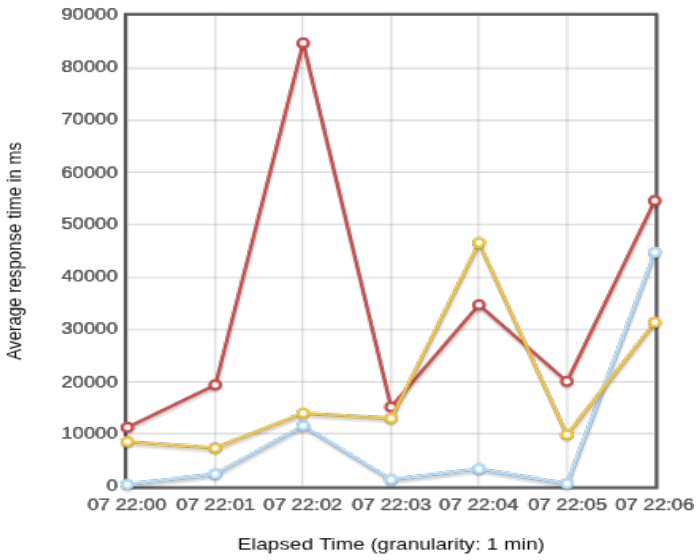
```
// SetMaxOpenConns sets the maximum number of open connections to
the database.
//
// If MaxIdleConns is greater than 0 and the new MaxOpenConns is
less than
// MaxIdleConns, then MaxIdleConns will be reduced to match the
new
// MaxOpenConns limit.
//
// If n <= 0, then there is no limit on the number of open
connections.
// The default is 0 (unlimited).
func (db *DB) SetMaxOpenConns(n int) { ... }
```

Κώδικας 15: Θέση μέγιστου αριθμού συνδέσεων

Αν δεν ορίσει ο προγραμματιστής τον αριθμό ενεργών συνδέσεων, τότε αυτός είναι απεριόριστος και επαφίεται σε περιορισμούς χαμηλότερου επιπέδου, όπως ο αριθμός ενεργών συνδέσεων που επιτρέπει η υπηρεσία της βάσης, ο αριθμός των file descriptors που είναι διαθέσιμοι στην εφαρμογή και η επιτρεπόμενη διαθέσιμη μνήμη.

Χωρίς την αλλαγή αυτή της ρύθμισης, η επικοινωνία με τη βάση κατέρρεε σε κάποιες από τις αποπειρώμενες συνδέσεις και παρουσιάζονταν *EOF* σφάλματα, τα οποία υποδείκνυαν το πρόβλημα. Αυτό γινόταν δεδομένου ότι η βιβλιοθήκη είχε αυτήν την άπληστη και επιθετική στρατηγική για νέες συνδέσεις και η βάση διαθέτει προκαθορισμένο ανώτατο όριο συνδέσεων 100.

Όπως φαίνεται στο παρακάτω διάγραμμα, δεδομένου ότι εξαντλούνταν η δεξαμενή διαθέσιμων συνδέσεων στη βάση, κάποια αιτήματα προς αυτή μπλόκαραν και ο χρόνος απόκρισης για το HTTP αίτημα ανερχόταν μέχρι και τα 88s, πράγμα εξαιρετικά ασύμβατο με τις προδιαγραφές των μηχανημάτων στα οποία έτρεχαν οι δύο υπηρεσίες.



Graph 2: Χρόνος απόκρισης χωρίς βελτιστοποίηση αριθμού συνδέσεων

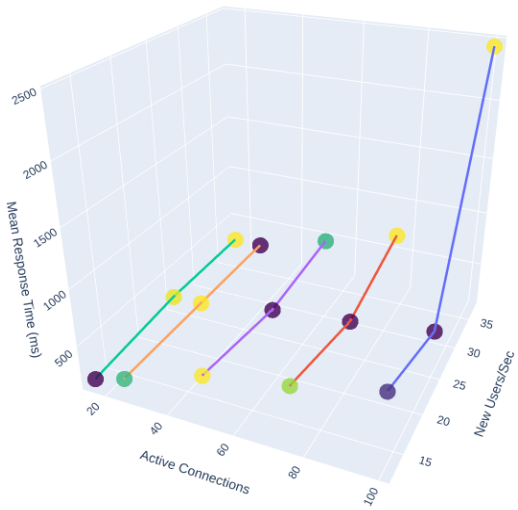
Η λύση σε αυτό το πρόβλημα ήταν να τεθεί ο αριθμός συνδέσεων σε συγκεκριμένο νούμερο. Έγιναν διάφορες δοκιμές με αρχική αυτή των 100 ανοιχτών συνδέσεων, όπου είναι και ο μέγιστος αριθμός που ορίζει η βάση.

```
psql (17.5 (Debian 17.5-1.pgdg120+1))
Type "help" for help.

thesis=# show max_connections
thesis-# ;
 max_connections
-----------------
 100
(1 row)
```

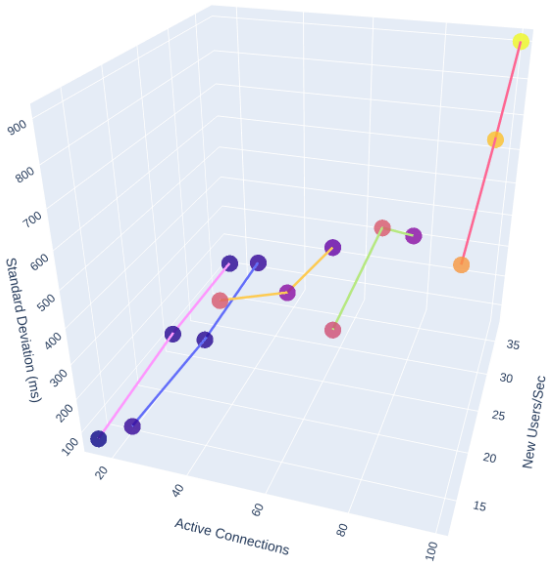
Figure 15: Μέγιστος αριθμός διαθέσιμος συνδέσεων σχεσιακής βάσης

Με το σκεπτικό ότι πολλές ανοιχτές συνδέσεις δημιουργούν επιπρόσθετη κίνηση στη βάση και στους μηχανισμούς ανάκτησης των δεδομένων, δοκιμάστηκαν τιμές χαμηλότερες από το 100. Παρακάτω φαίνονται κάποια πειραματικά αποτελέσματα. Τα δύο http requests με κόκκινο και κίτρινο ανακτούν μια λίστα με τις δωρεές και τα αιτήματα αντίστοιχα, στην τάξη των kB σε σύγκριση με τα γαλάζια σημεία που είναι οι δωρεές για το αίτημα με αναγνωριστικό id 54. Ο χρόνος αναμένεται να είναι μεγαλύτερος στις δύο πρώτες λίστες, δεδομένου ότι συμβαίνουν διάφορα joins κατά το επερώτημα στη σχεσιακή βάση, μια πράξη αρκετά χρονοβόρα. Έχουμε τα παρακάτω αποτελέσματα:



Graph 3: Διάγραμμα συσχέτισης ενεργών συνδέσεων της σχεσιακής βάσης - νέων χρηστών ανά δευτερόλεπτο - χρόνου απόκρισης χωρίς CACHE(1)

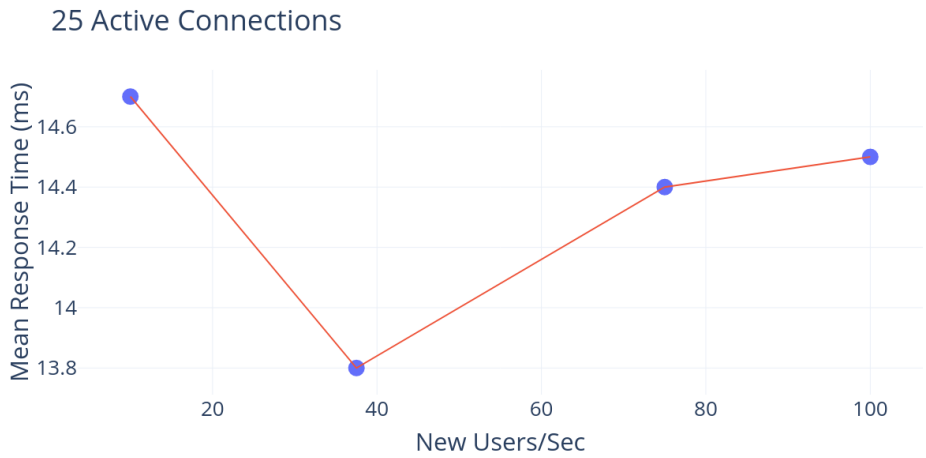
Αυξάνοντας σταδιακά τους χρήστες που “γεννιούνται” ανά δευτερόλεπτο από το JMeter από 12.5, σε 25 και τέλος σε 37.5, παρατηρήσαμε ότι οι 15 και 25 συνδέσεις είναι ικανοποιητικές τιμές που συνδυάζουν χαμηλή μέση τιμή απόκρισης (άνω διάγραμμα) και επίσης σταθερή τυπική απόκλιση (κάτω διάγραμμα)



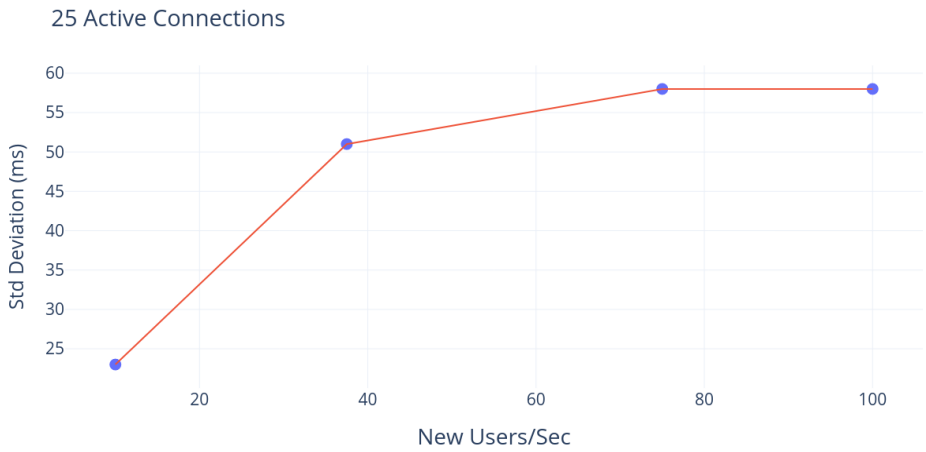
Graph 4: Διάγραμμα συσχέτισης ενεργών συνδέσεων της σχεσιακής βάσης - νέων χρηστών ανά δευτερόλεπτο - χρόνου απόκρισης με CACHE (2)

Στα επόμενα πειράματα ενεργοποιήθηκε ένας write through cache μηχανισμός για τα GET ερωτήματα. Η είσοδος στην cache συμβαίνει όταν δεν υπάρχει σε αυτή το αιτούμενο κλειδί. Κατά το cache miss δηλαδή, η τιμή εξάγεται από τη βάση, εξυπηρετείται με αυτή το αίτημα και ενημερώνεται η cache. Επίσης, με συγκεκριμένο τρόπο στα POST, UPDATE αιτήματα διαγράφεται η cache τιμή του κλειδιού εφόσον τα δεδομένα ενημερώνονται από αυτές τις μεθόδους.

Παρατηρούμε ότι παρόλο που οι νέοι χρήστες ανά δευτερόλεπτο υπερδιπλασιάζονταν, η χρήση της Redis Cache επέφερε σημαντικά αποτελέσματα και σε χαμηλότερη τάξη μεγέθους.

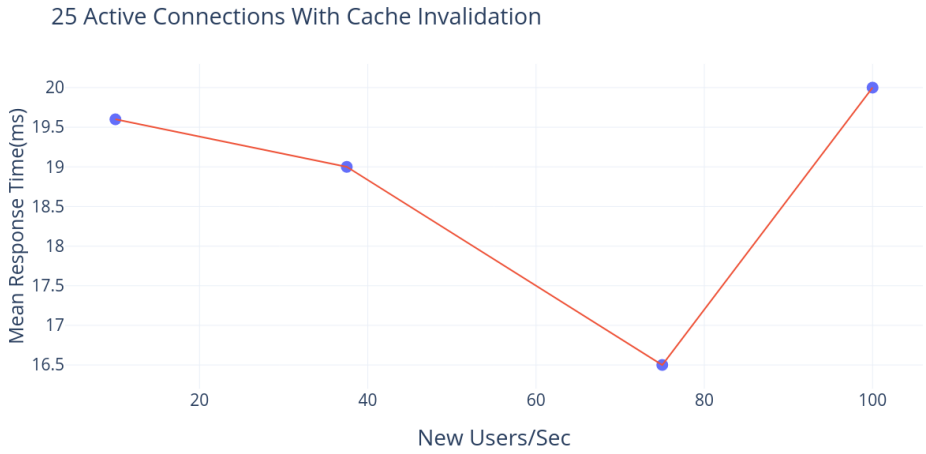


Graph 5: Μέση τιμή χρόνου απόκρισης ανά νέους χρήστες ανά δευτερόλεπτο για 25 ενεργές συνδέσεις στη σχεσιακή βάση

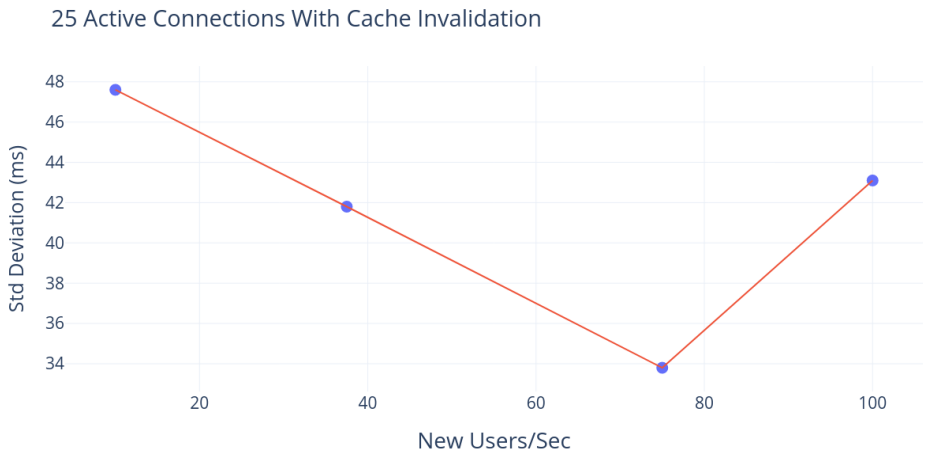


Graph 6: Κανονική απόκλιση χρόνου απόκρισης ανά νέους χρήστες ανά δευτερόλεπτο για 25 ενεργές συνδέσεις στη σχεσιακή βάση

Τέλος υλοποιήθηκε προγραμματιστικό σενάριο κατά το οποίο σε σχετικά τυχαία χρονική στιγμή αποστέλλονταν κύμα αιτημάτων στον εξυπηρετητή ώστε να εκτελεστεί η διαδικασία ακύρωσης των τιμών της cache, τόσο για οντότητα με συγκεκριμένο αναγνωριστικό αλλά ταυτόχρονα και για τη λίστα.

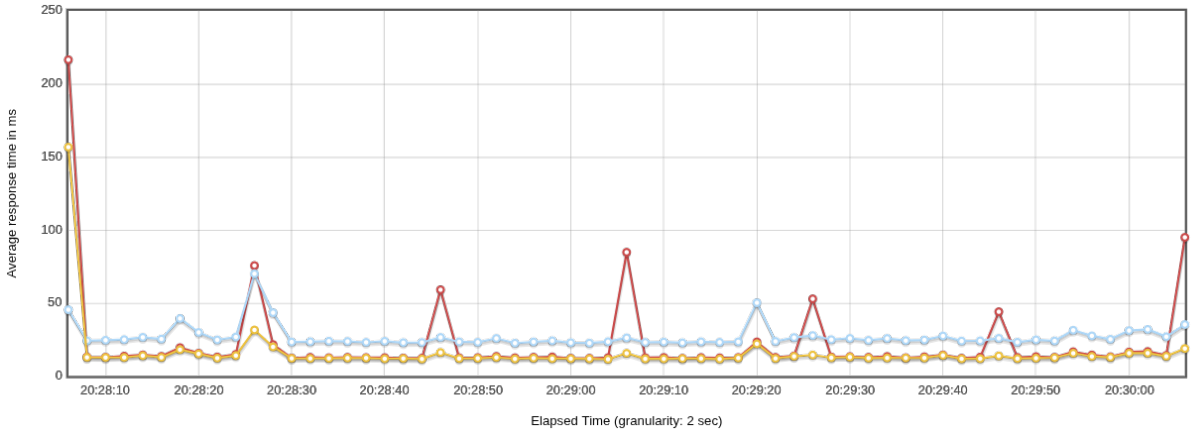


Graph 7: Μέση τιμή χρόνου απόκρισης ανά νέους χρήστες ανά δευτερόλεπτο για 25 ενεργές συνδέσεις στη σχεσιακή βάση με ακύρωση CACHE



Graph 8: Κανονική απόκλιση χρόνου απόκρισης ανά νέους χρήστες ανά δευτερόλεπτο για 25 ενεργές συνδέσεις στη σχεσιακή βάση με ακύρωση CACHE

Στο παρακάτω διάγραμμα φαίνονται οι στιγμές που αποστέλλεται το κύμα:



Graph 9: Χρόνος απόκρισης ανά 10 δευτερόλεπτα για 100 νέους χρήστες ανά δευτερόλεπτο για 25 ενεργές συνδέσεις στη σχεσιακή βάση με ακύρωση CACHE

Η χρήση της cache έριξε κατά πολύ τις τιμές απόκρισης (~270ms → ~20ms). Επίσης απορρόφησε ταχέα το “θόρυβο” που προσέθεσαν τα αιτήματα ακύρωσης (invalidation).

Υπηρεσία αναζήτησης

Σύγκριση PSQL FTS και Elastic

Η PostgresQL που χρησιμοποιήθηκε ως σχεσιακή βάση διαθέτει τη λειτουργία Full Text Search (FTS), η οποία χρησιμοποιεί ανάστροφους δείκτες (Κεφ. 2) χρησιμοποιώντας τους εγγενείς τύπους tsvector (διάνυσμα κειμένου) και tsquery για δεικτοδότηση και αναζήτηση κειμένου στις εγγραφές. Έχει σχεδιαστεί για βασικές έως μέτρια πολύπλοκες απαιτήσεις αναζήτησης απευθείας μέσα στη βάση δεδομένων. Από την άλλη η Elasticsearch, είναι μια ξεχωριστή, κατανεμημένη μηχανή αναζήτησης πλήρους κειμένου που βασίζεται στο Apache Lucene. Παρέχει προηγμένες δυνατότητες αναζήτησης και είναι βελτιστοποιημένη για

λειτουργίες αναζήτησης μεγάλης κλίμακας, βαθύτερη ανάλυση των εγγραφών και εξεζητημένη πολυκριτήρια αναζήτηση.

Αρχιτεκτονικά οι δύο μηχανές διαφέρουν δεδομένου ότι το FTS λειτουργεί μέσα στη μηχανή βάσης δεδομένων της PostgresqQL. Τα πάντα είναι στενά ενσωματωμένα και χρησιμοποιούν σύνταξη SQL. Αντίθετα η Elasticsearch εκτελείται ως αυτόνομο σύμπλεγμα κόμβων, ξεχωριστό από την κύρια βάση δεδομένων της εφαρμογής. Η Elasticsearch επιλέχθηκε και για αυτό το λόγο προκειμένου να εισάγει πλεονασμό στα δεδομένα και να απολέσει το φόρτο εργασίας της αναζήτησης από την κύρια υπηρεσία. Επίσης δεδομένου ότι το FTS της PostgreSQL είναι συγκεντρωτικό και στενά συνδεδεμένο, ενώ το Elasticsearch είναι αποκεντρωμένο, διευκολύνει την οριζόντια επεκτασιμότητα της αναζήτησης.

Ο σχηματισμός λεκτικών μονάδων είναι το πρώτο και πιο κρίσιμο βήμα στην αναζήτηση πλήρους κειμένου, καθώς καθορίζει πώς διαχωρίζεται το κείμενο σε επιμέρους λέξεις (tokens). Στην PostgreSQL, το FTS χρησιμοποιεί την textsearch υποδομή με προκαθορισμένους αναλυτές και λεξικά. Για τα ελληνικά, υποστηρίζεται ο Snowball stemmer, ο οποίος μπορεί να αποδώσει στοιχειώδη στελεχοποίηση, αλλά η ανάλυση μπορεί να είναι περιορισμένη, ειδικά για σύνθετες λέξεις ή περιπτώσεις με λανθασμένη στίξη. Αντίθετα, η Elasticsearch προσφέρει ένα πολύ πιο ευέλικτο και επεκτάσιμο pipeline που μπορεί να περιλαμβάνει: tokenizers, character filters, token filters. Επιπλέον, επιτρέπει το δυναμικό customization αυτών των σταδίων μέσω ρυθμίσεων από ευέλικτο JSON API που παρέχει . Αυτό δίνει στο χρήστη πλήρη έλεγχο στην επεξεργασία γλώσσας και καθιστά την αναζήτηση πιο ακριβή και προσαρμόσιμη για σύνθετα ελληνικά κείμενα.

Ένα σημαντικό πλεονέκτημα της Elasticsearch είναι το RESTful JSON API της. Τα ερωτήματα υποβάλλονται και τα αποτελέσματα επιστρέφονται μέσω JSON, κάτι που διευκολύνει σημαντικά τη χρήση της από frontend εφαρμογές ή microservices χωρίς την ανάγκη για σύνθετες βιβλιοθήκες και μεσολαβητές επεξεργασίας. Το JSON API διευκολύνει τη αναζήτηση με φίλτρα παρέχοντας διαισθητικά εύχρηστους τρόπους σύνθεσης πολύπλοκων ερωτημάτων, τη σύνθετη σύνδεση όρων, την προτεραιοποίηση πεδίων με απονομή βαρών κατά βούληση και *την πολυγλωσσική υποστήριξη*, όλα μέσα από παραμετροποιήσιμες δομές. Ο τρόπος αυτός ενδείκνυται για εφαρμογές όπου ο χρήστης διαμορφώνει δυναμικά κριτήρια αναζήτησης.

Ακόμα, η PostgreSQL δεν χρησιμοποιεί τον BM25 αλγόριθμο για κατάταξη των αποτελεσμάτων. Αντίθετα, υλοποιεί έναν μηχανισμό κατάταξης βασισμένο σε τη συχνότητα εμφάνισης λέξεων, τη σπανιότητα τους στο συνολικό λεκτικό σώμα και το μήκος του εγγράφου. Αυτή η προσέγγιση είναι σχετικά απλή και προσφέρει λιγότερο διακριτική ευαισθησία σε σχέση με τον BM25. Η Elasticsearch, από την άλλη, χρησιμοποιεί εξ ορισμού τον BM25. Ο BM25 λαμβάνει υπόψη την σχετική συχνότητα λέξης, την σπανιότητα στο σύνολο εγγράφων (IDF), καθώς και την κανονικοποίηση μήκους κειμένου. Αυτό τον καθιστά ιδιαίτερα αποτελεσματικό στη σειρά κατάταξης αποτελεσμάτων αναζήτησης με βάση το πόσο "σχετικά" είναι τα κείμενα προς το ερώτημα του χρήστη.

Τέλος, η PostgreSQL, ως σχεσιακή βάση, βασίζεται στη γλώσσα SQL για την αναζήτηση. Αυτό σημαίνει ότι το FTS ενσωματώνεται πλήρως σε SQL ερωτήματα και επιτρέπει τη συγχώνευση αποτελεσμάτων με άλλες δομές ή φίλτρα του σχήματος της βάσης. Ενώ αυτό προσφέρει συνέπεια και ισχύ, μπορεί να γίνει περίπλοκο και λιγότερο φιλικό για τους μη εξοικειωμένους χρήστες, ειδικά όταν εμπλέκονται tsvector, tsquery, to_tsvector, plainto_tsquery και phraseto_tsquery. Επίσης, η χρήση κανονικών εκφράσεων και GIN ευρετηρίων απαιτεί γνώση λεπτομερειών της υλοποίησης. Σε αντίθεση με το JSON της Elasticsearch, η SQL δεν παρέχει διαισθητική δομή για αναζητήσεις με πολλά κριτήρια και εμφωλευμένα πεδία, κάτι που καθιστά τις σύνθετες αναζητήσεις πιο δύσκολες στη διαχείριση και την αποσφαλμάτωση.

Μπορεί λοιπόν να είναι απαραίτητο τα δεδομένα να συγχρονίζονται από την PostgreSQL, συχνά μέσω τρίτων και προσαρμοσμένων αγωγών, και αυτό να εισάγει καθυστέρηση και διπλασιασμό, αλλά επιτρέπει την απομάκρυνση βαρέων υπολογισμών αναζήτησης από την κύρια υπηρεσία και την κύρια βάση δεδομένων.

Σύγκριση αποτελεσμάτων αναζήτησης

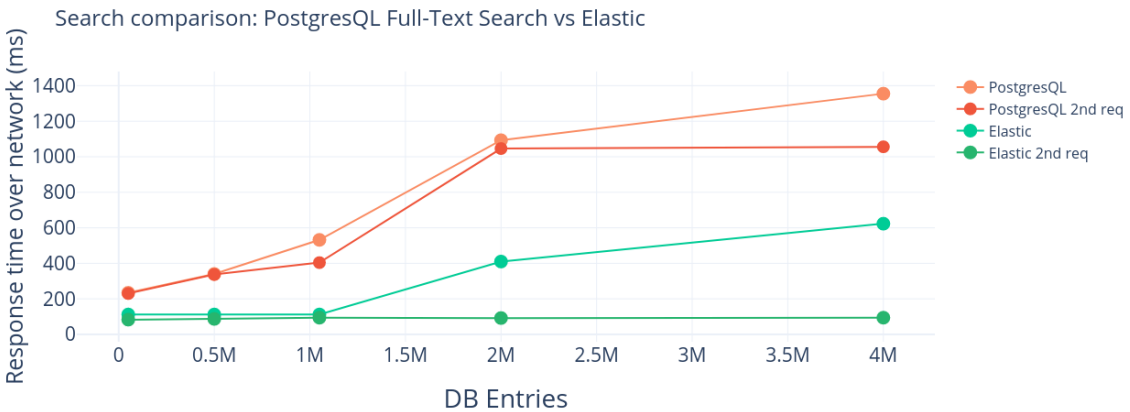
Για την αξιολόγηση και σύγκριση ταχύτητας των δύο μηχανών αναζήτησης κειμένου, πραγματοποιήθηκε το παρακάτω πείραμα.

Αρχικά, δημιουργήθηκαν βαθμηδόν dummy δεδομένα με τη χρήση του πακέτου python-fake [9], ξεκινώντας από τα 50k και φτάνοντας μέχρι τα 4m. Έπειτα έγινε εισαγωγή τους στις δύο βάσεις και τέλος χρονομετρήθηκε ο χρόνος εκτέλεσης ενός απλού επερωτήματος και στις δύο βάσεις με τη χρήση του command *time* [10].

Το επερώτημα ήταν μια απλή καταμέτρηση εγγραφών οι οποίες περιείχαν απλές λέξεις και όχι φράσεις, δεδομένου ότι η βιβλιοθήκη που δημιούργησε τα δεδομένα παράγει μεν κείμενο που αξιοποιήθηκε σαν περιγραφή του αντικειμένου, το κείμενο δε αυτό είναι ασυνάρτητο. Επιλέχθηκε η καταμέτρηση και όχι η ανάκτηση δεδομένων γιατί αφενός η ανάκτηση θα καθυστερούσε δεδομένου ότι θα έπρεπε να μετακινηθούν χιλιάδες εγγραφές, αφετέρου η default επιλογή της elastic είναι η ανάκτηση max 10k εγγραφών σε κάθε ερώτημα.

Από παρακάτω διάγραμμα μπορούμε να εξάγουμε τα εξής συμπεράσματα. Αφενός, η elastic έχει καλύτερο και αποδοτικότερο scaling factor. Με την αύξηση των εγγραφών, ο χρόνος απόκρισης παρέμεινε στην ίδια τάξη μεγέθους, ενώ για εγγραφές μικρότερες σε πλήθος από το εκατομμύριο, μεταβλήθηκε ±3 ms ο χρόνος απόκρισης.

Αφετέρου η elastic διαθέτει καλύτερο caching μηχανισμό εσωτερικά για τα ερωτήματα. φαίνεται στο διάγραμμα πως όταν γίνονταν δύο ίδια ερωτήματα διαδοχικά, ο χρόνος απόκρισης παρέμενε σταθερά κάτω από τα 100ms. Αυτό είναι απόρροια του γεγονότος ότι η elastic αξιοποιεί σε μεγάλο βαθμό τις cache λειτουργίες του ίδιου του λειτουργικού συστήματος [11].



Graph 10: Σύγκριση χρόνου απόκρισης μεταξύ Elasticserch και Postgres FTS για αυξανόμενο πλήθος εγγραφών

Λεπτομέρειες στην υλοποίηση με Elasticsearch

Η λειτουργία της αναζήτησης διαφέρει από οντότητα σε οντότητα. Οι οντότητες έχουν ομαδοποιηθεί στους παρακάτω δείκτες με εννοιολογικά κριτήρια. Η εφαρμογή διαχειρίζεται:

- Χρήστες και οργανισμούς, τα οποία ενοποιούνται σε ένα κοινό πλαίσιο αναζήτησης υπό τον όρο **profile,**
- Καμπάνιες και εκδηλώσεις, δηλαδή οντότητες που έχουν να κάνουν με χρονοπρογραμματισμό και σχεδίαση, το **planning,**
- Δωρεές και αιτήματα, δηλαδή οντότητες οι οποίες έχουν να κάνουν με την κύρια διάδραση μεταξύ των χρηστών και υπόκεινται στο δείκτη **actions,**
- Αντικείμενα και υπηρεσίες, τα οποία είναι τα μέσα ανταλλαγής μεταξύ των χρηστών και ανήκουν στο δείκτη **provisions**.

Ο παραπάνω διαχωρισμός έγινε με γνώμονα έννοιες και αρχές από το **Domain-Driven Design (DDD)**, όπως διατυπώνονται στο [12] .

Με αυτή την προσέγγιση, επιτρέπεται οντότητες από την επιχειρησιακή λογική να συγκατοικούν κάτω από την ίδια ομπρέλα όταν και μόνο όταν ανήκουν στο ίδιο σημασιολογικό πλαίσιο — δηλαδή, σε αυτό που η θεωρία ονομάζει *bounded context*.

Ο διαχωρισμός των δεδομένων σε δείκτες όπως profiles, planning, actions και provisions δεν αποτελεί απλώς τεχνική επιλογή, αλλά είναι προέκταση της επιχειρησιακής γνώσης μέσα στο τεχνικό σύστημα. Κάθε δείκτης αντιστοιχεί σε μια περιοχή της εφαρμογής με διαφορετικές απαιτήσεις και διακριτές σημασιολογίες. Ο διαχωρισμός αυτός παρέχει στα δεδομένα αυτό το οριοθετημένο σημασιολογικό πλαίσιο, το οποίο αξιοποιείται ώστε να παρέχει την κατάλληλη ερμηνεία κάθε φορά. Για παράδειγμα, διάφορες οντότητες έχουν ένα πεδίο status. Για τις καμπάνιες αυτό τίθεται είτε κατά τη δημιουργία της καμπάνιας, είτε στην πορεία όταν προστίθενται αντικείμενα και υπηρεσίες ως αιτήματα στην καμπάνια, είτε όταν προστίθενται εκδηλώσεις υπό την αιγίδα της. Στα αιτήματα προφανώς το status έχει άλλη ερμηνεία και ενδέχεται να έχει διαφορετικές διαθέσιμες τιμές, οι οποίες σχετίζονται με το αν είναι ενεργό, αν έχει δεχτεί κάποια δωρεά κ.ο.κ.. Κοινώς, με το διαχωρισμό αποτρέπεται το semantic drifting (εννοιολογική απόκλιση), κάτι που σε ένα ενιαίο δείκτη, θα φανέρωνε τον κίνδυνο σημασιολογικής σύγχυσης θεωρητικά και κακής δεικτοδότησης άρα αργής αναζήτησης πρακτικά.

Ακόμα, ο διαχωρισμός ωφελεί επίσης την ταχύτητα και την ακρίβεια της αναζήτησης με πολλούς τρόπους. Είτε αυτός είναι από το γεγονός ότι αναζητούνται λιγότερες εγγραφές, είτε υπό την έννοια ότι απαιτείται μικρότερο φιλτράρισμα των αποτελεσμάτων, είτε ακόμα από τη σκοπιά του caching. Η Elasticsearch βάση, με τον τρόπο που λειτουργεί σε επίπεδο θραύσματος (Shard) διατηρεί μια μνήμη cache ανά request που συμβαίνει. Στην περίπτωση αυτή της εφαρμογής που αφενός, στο πρώιμο επίπεδο διαθέτει μοναδικά shards ανά δείκτη και αφετέρου έχει σημεία όπου γίνεται κατ΄ εξακολούθηση η αναζήτηση όμοιων πραγμάτων, λειτουργεί βελτιωτικά [13].

Τέλος, με τον τρόπο αυτό επιτυγχάνεται και καλύτερος διαχωρισμός αρμοδιοτήτων μεταξύ ομάδων ανάπτυξης, καθώς κάθε bounded context (δείκτης) μπορεί να εξελιχθεί αυτόνομα, με δική του δομή, σχήμα και κύκλο ζωής, χωρίς να επηρεάζει ή να εξαρτάται από τα υπόλοιπα. Αυτό συντελεί όχι μόνο στην τεχνική βιωσιμότητα της εφαρμογής, αλλά και στην ενίσχυση της

σαφήνειας στην ερμηνεία και της συμβατότητας ανάμεσα στον κώδικα και την επιχειρησιακή λογική που εξυπηρετεί.

Υπηρεσία Ειδοποιήσεων

Διαχείριση νέας σύνδεσης και εισερχόμενων μηνυμάτων

Η υπηρεσία ειδοποιήσεων εκτελεί δύο διακριτές διεργασίες ταυτόχρονα. Αφενός, καταναλώνει μηνύματα από το διαμοιραστή μηνυμάτων και ενημερώνει τη βάση ενδιαφερόμενων βάσει αυτών. Αφετέρου, διαχειρίζεται εισερχόμενες συνδέσεις web-socket, από τις οποίες για κάθε μια γεννά μια goroutine, η οποία αναλαμβάνει να εξυπηρετήσει τα αιτήματά της. Κάθε μια τέτοια goroutine, επικοινωνεί με τη γονική μέσω καναλιών ενημερώνοντας την για το lifetime της.

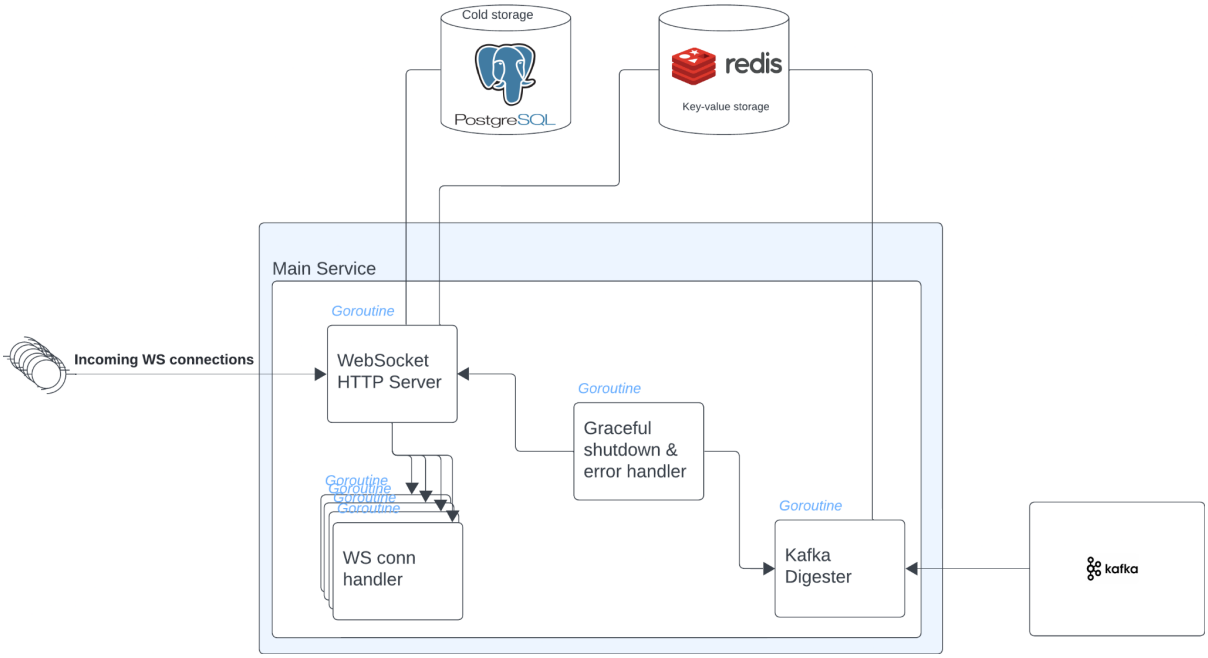


Figure 16: Συστατικά υπηρεσίας ειδοποιήσεων

Διαδρομή εισερχόμενης ειδοποίησης

Η εισερχόμενη ειδοποίηση είναι ένα μήνυμα το οποίο διαβάζεται από τη δομή της εφαρμογής που ονομάζεται καταναλωτής. Πρόκειται για κώδικα που συνεχώς δειγματοληπτεί τη σύνδεση με το διαμοιραστή μηνυμάτων και σε περίπτωση εισερχόμενου μηνύματος, γεννά μια goroutine για να το επεξεργαστεί όπως φαίνεται στο παρακάτω διάγραμμα.

Η επεξεργασία περιλαμβάνει αρχικά την αποσειριοποίηση των δεδομένων από τη μορφή του μηνύματος, και μετατροπή σε μορφή αξιοποιήσιμη από την εφαρμογή. Έπειτα, εξάγει την πληροφορία για το ποιους αφορά η ειδοποίηση και ενημερώνει τις αντίστοιχες εγγραφές της βάσης με τους ενδιαφερόμενους. Η πληροφορία για το ποιος πρέπει να ενημερωθεί ποικίλει από οντότητα σε οντότητα. Σύμφωνα με τον παρακάτω πίνακα.

Οντότητα		Ποιος λαμβάνει ειδοποίηση
Καταχώρηση Δωρεάς	Σε απλό αίτημα	Λοιποί δωρητές
	Σε αίτημα καμπάνιας	Δημιουργός καμπάνιας και λοιποί δωρητές και παρευρισκόμενοι στις εκδηλώσεις της καμπάνιας
Προσέλευση (ή Μη Προσέλευση) σε εκδήλωση		Δημιουργός εκδήλωσης
Προσθήκη εκδήλωσης		Παρευρισκόμενοι στις εκδηλώσεις της καμπάνιας

Τέλος, συμβαίνει η δρομολόγηση του μηνύματος υπό συνθήκη στους ενδιαφερόμενους χρήστες.

```
func (s *Service) ProcessKafkaIncoming(ctx context.Context) {
 ...
     for run {
         select {
         case <-ctx.Done():
             run = false
             default:
                 event := s.consumer.Poll(-1)
                 switch msg := event.(type) {
                 case *kafka.Message:
                 ...
                     go func() {
                        errorCheck(s.storeNotification(ctx, n))
                        errorCheck(s.updateInterests(ctx, *n))
                        errorCheck(s.routeNotification(ctx, *n))
                 ...
             }()
            ...
          }
      }
 ...
}
```

Κώδικας 16: Επεξεργασία εισερχόμενου μηνύματος στο Kafka (ειδοποιήσεις)

Στο επόμενο διάγραμμα φαίνεται η λογική με την οποία δρομολογούνται οι ειδοποιήσεις. Η λογική διαφέρει

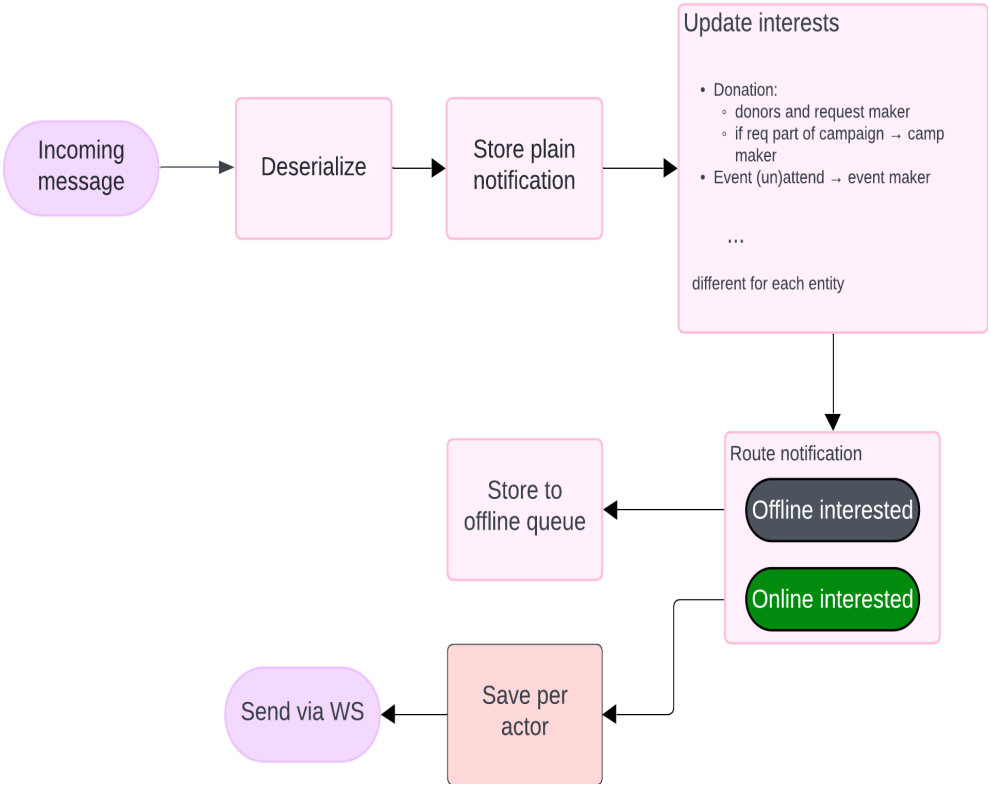


Figure 17: Διάγραμμα λογικής ροής υπηρεσίας ειδοποιήσεων

Στην παρακάτω εικόνα φαίνονται οι διεργασίες που συμβαίνει όταν κάποιος χρήστης συνδεθεί και πρέπει να καταναλώσει ειδοποιήσεις που τον αφορούν, οι οποίες δημιουργήθηκαν ενώ αυτός ήταν εκτός σύνδεσης.

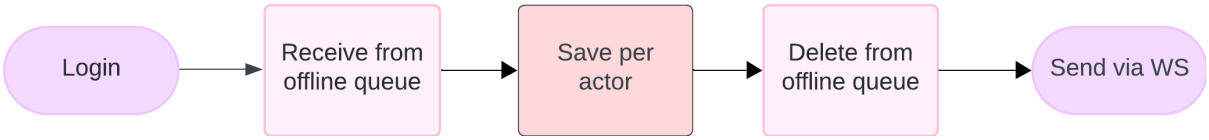


Figure 18: Διάγραμμα λογικής ροής κατανάλωσης offline ειδοποίησης

Οι ειδοποιήσεις αποθηκεύονται αυτούσιες σε πίνακα της βάσης και σε ξεχωριστό αποθηκεύεται η σχέση της κάθε ειδοποίησης με τους ενδιαφερόμενους. Αυτή η σχέση χαρακτηρίζεται και από ένα πεδίο κατάστασης, το οποίο σηματοδοτεί αν έχει αναγνωσθεί η ειδοποίηση από τον ενδιαφερόμενο ή όχι.

Προσκήνιο

Για το παρασκήνιο αξιοποιήθηκε το react-js framework, ένα σύγχρονο πλαίσιο ανάπτυξης διεπαφών αλληλεπίδρασης. Η React θεμελίωσε στην αγορά το Virtual Dom, το και αποτελεί μία από τις σταθερές παγκοσμίως για ανάπτυξη παρασκηνίου εφαρμογών.
Το περιβάλλον αλληλεπίδρασης μιας εφαρμογής σχηματίζει την πρώτη σε ένα χρήστη και οφείλει να είναι συνεπές και εύχρηστο. Σήμερα παρέχεται από την κοινότητα, πληθώρα επιλογών από βιβλιοθήκες οι οποίες ακολουθούν συστηματικά μοτίβα για την παροχή δομικών λίθων για τη σχεδίαση και την παρουσίαση μιας εφαρμογής. Στη συγκεκριμένη

εφαρμογή αξιοποιήθηκε η **mui/material** βιβλιοθήκη. Αυτή αποτελεί ένα έργο ανοικτού κώδικα για το react-js η οποία παρέχει components, έτοιμα δηλαδή κομμάτια κώδικα-συστατικά, τα οποία ικανοποιούν τα πρότυπα του Material Design.

Material UI

Το Material Design είναι ένα σύστημα σχεδιασμού που αναπτύχθηκε από την Google, με σκοπό να προσφέρει μια συνεπή και ενοποιημένη εμπειρία σε όλες τις πλατφόρμες και συσκευές. Εμπνευσμένο από τον φυσικό κόσμο και τις υφές του, το Material Design χρησιμοποιεί διάφορα εφέ, στο φωτισμό, στις σκιές και κινούμενα σχέδια για να αποδώσει ιεραρχία και διαδραστικότητα. Η θεμελιώδης μεταφορά του είναι – ένας ψηφιακός χώρος που συμπεριφέρεται όπως το χαρτί και το μελάνι, προσφέροντας απτικές επιφάνειες και ευέλικτες διατάξεις για καθοδήγηση του χρήστη.

Στον πυρήνα του, το Material Design δίνει έμφαση στη σαφήνεια, τη νοηματική δομή και τη δράση του χρήστη. Συστατικά στοιχεία όπως κουμπιά, κάρτες και μενού πλοήγησης σχεδιάζονται ώστε να είναι τόσο λειτουργικά όσο και οπτικά συνεπή, διασφαλίζοντας την προσβασιμότητα και τη χρηστικότητα σε διάφορα μεγέθη οθόνης. Το σύστημα ενθαρρύνει τη δυναμική χρήση χρωμάτων, τυπογραφίας και εικονογραφίας, βασισμένα σε λεπτομερείς σχεδιαστικές αρχές. Μέσω αυτής της προσέγγισης, το Material Design επιδιώκει να δημιουργεί εμπειρίες οικείες αλλά και βαθιά ελκυστικές.

React Components

Τα react components (συστατικά) είναι οι δομικοί λίθοι σε κάθε εφαρμογή react. Είναι κομμάτια κώδικα διεπαφής, παραμετροποιήσιμα και επαναχρησιμοποιήσιμα, τα οποία διαθέτουν αυτόνομη δομή, λογική και συμπεριφορά. Στον πυρήνα τους, δεν είναι τίποτα άλλο από συναρτήσεις της γλώσσας JavaScript οι οποίες επιστρέφουν μικρά ή και μεγάλα κομμάτια της διεπαφής που φαίνεται στο χρήστη. Τα components μπορούν να εμφωλευτούν το ένα στο άλλο, διαγράφοντας μια δενδρική ιεραρχία. Component μπορεί να είναι από κάτι μικρό όσο ένα εικονίδιο ή ένα κουμπί, αλλά ακόμη και κάτι πολύ σύνθετο, όσο μια ολόκληρη σελίδα της εφαρμογής. Στις αρχικές εκδόσεις της react, τα components ορίζονταν κυρίως ως κλάσεις με διακριτές μεθόδους για την αρχικοποίηση, τις αλλαγές στην εσωτερική κατάσταση του συστατικού, μεθόδους θέτες (setters) για τις εσωτερικές μεταβλητές και φυσικά τη συνάρτηση για το οπτικό αποτέλεσμα του συστατικού. Στη σύγχρονη εκδοχή έχει επικρατήσει η χρήση των hooks («άγκιστρα») για την εκτέλεση αυτών των λειτουργιών, αλλά και πολλών ακόμη.

Τα δύο hooks το οποία εμπλέκονται άμεσα με τον κύκλο ζωής και το οπτικό αποτέλεσμα ενός component είναι τα useState και useEffect. Το useState παρέχει στον προγραμματιστή μια μεταβλητή και μια συνάρτηση για να τροποποιηθεί αυτή η μεταβλητή. Κάθε φορά που τροποποιείται, το component ανανεώνεται οπτικά λαμβάνοντας υπόψη την αλλαγή. Αυτό συμβαίνει επίσης όταν κάποιο από τα props του component ανανεωθούν.

Τα props (properties δηλ. ιδιότητες) είναι τα ορίσματα τα οποία λαμβάνει το component δεδομένου ότι πλέον τα components ορίζονται ως συναρτήσεις. Τα props είναι read-only, άρα το component μπορεί είτε να τα διαβάσει μόνο είτε να τα περάσει σαν props σε κάποιο component παιδί. Αυτή η μονόδρομη τακτική της react, διευκολύνει την αποσφαλμάτωση, αφού καθιστά πιο προβλέψιμη τη συμπεριφορά των components.

Η ανανέωση του component γίνεται επίσης μέσω της useEffect. Αυτή, στη συναρτησιακή προσέγγιση, είναι μια συνάρτηση που ορίζεται εντός του component και η οποία τρέχει κατά σε κάθε ανανέωση του component. Αυτό μπορεί να αποβεί μοιραίο όταν η ίδια η useEffect προκαλεί μια ανανέωση, άρα δημιουργείται μια ατέρμονη επανάληψη, η οποία πλέον εντοπίζεται από τη μηχανή της react και ενημερώνεται ο προγραμματιστής. Η useEffect, πέρα από την Effect συνάρτηση που εκτελείται όπως αναλύθηκε, λαμβάνει σαν όρισμα και έναν πίνακα μεταβλητών, η μεταβολή των οποίων πυροδοτεί μια ανανέωση. Αυτό το μοτίβο είναι εμφανές και σε άλλα hooks και είναι μία από τις βασικές λειτουργίες της react. Αν ο εν λόγω πίνακας είναι κενός, τότε η συνάρτηση εκτελείται μια φορά κατά τον κύκλο ζωής του component, μόνο κατά την δημιουργία του.

React Hooks

Τα hooks είναι ειδικές συναρτήσεις της react που επιτρέπουν στον προγραμματιστή να αλληλεπιδράσει με την κατάσταση-μνήμη και τη συμπεριφορά καθόλη τη διάρκεια ζωής, ενός component. Ακόμη, μπορεί να πυροδοτήσει παρενέργειες (side effects), δεδομένης μιας εισερχόμενης αλλαγής (π.χ. λήψη δεδομένων από κάποιο API), να αλληλεπιδράσει με το DOM και να απομνημονεύσει πράγματα ώστε αυτά να μην επαναδημιουργηθούν μεταξύ των ανανεώσεων του component σε κάποια αλλαγή κατάστασής του. Επίσης, συμμετέχουν σημαντικά στην αναγνωσιμότητα και στην οργάνωση του κώδικα ενισχύοντας την εμπειρία του προγραμματιστή.

Context

Το context, δηλαδή τα συμφραζόμενα, είναι η προσπάθεια της γλώσσας για ένα σημείο καθολικής μνήμης στην εφαρμογή. Η δομή της γλώσσας με τα συστατικά είναι δενδρική. Δηλαδή, κάποια components έχουν παιδιά, στα οποία προφανώς μπορούν να περάσουν δεδομένα υπό τη μορφή των props. Τι γίνεται ωστόσο αν κάποια μεταβλητή, ορισμένη στη ρίζα ενός δέντρου components, είναι απαραίτητη σε κάποιο component αρκετά επίπεδα κάτω;

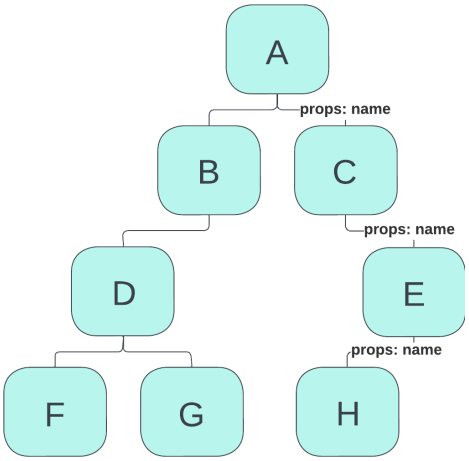


Figure 19: React component tree με props

Η αφελής λύση είναι η μεταβλητή να δίδεται σαν prop σε κάθε παιδί έως ότου να φτάσουμε στο επίπεδο. Αυτό δυσκολεύει την αναγνωσιμότητα του κώδικα και μπορεί να μπερδέψει τα πράγματα. Η λύση σε αυτό είναι το context, το οποίο πρακτικά αποτελεί μια δομή μνήμης/κατάστασης, η οποία εδρεύει εκτός της δενδρικής δομής των συστατικών και είναι προσβάσιμη από καθένα από αυτά, όπως φαίνεται στην παρακάτω εικόνα.

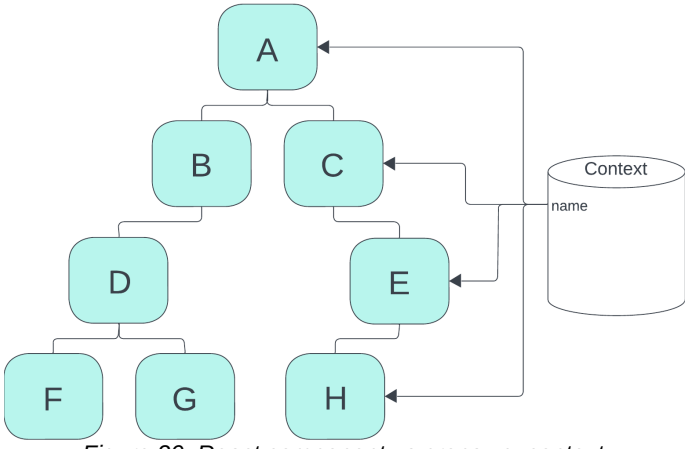


Figure 20: React component με props και context

Η προσέγγιση αυτή βοηθάει σε περιπτώσεις όπου κάποιες τιμές πρέπει να είναι καθολικά διαθέσιμες. Το πιο τρανό παράδειγμα αποτελεί η χρήση του JWT token, το οποίο πρέπει να επισυνάπτεται σε κάθε αίτημα στο API.

Redux Toolkit (RTK) και RTK Query

Καταλαβαίνει κανείς ότι η χρησιμότητα του context, μπορεί να εφαρμοστεί σε οτιδήποτε. Από το θέμα, τη γλώσσα της εφαρμογής, αλλά ακόμα πιο πολύ στα εισερχόμενα δεδομένα από κάποιο API. Δεδομένης της σημαντικότητας, αλλά και της σύνδεσης αυτής της λειτουργίας με τη λειτουργία ανάκτησης δεδομένων από κάποια υπηρεσία, υπάρχουν πλέον πολλές βιβλιοθήκες οι οποίες συγκεντρώνουν αυτές τις λειτουργίες υπό τη μορφή βιβλιοθήκης. Στην εφαρμογή έγινε χρήση της βιβλιοθήκης Redux Toolkit Query [14].

Το RTK Query αποτελεί μια σύγχρονη βιβλιοθήκη λύση για την ανάκτηση και τη διαχείριση των εισερχόμενων δεδομένων. Η χρήση του Redux στοχεύει στη μείωση της πολυπλοκότητας και της επαναληψιμότητας κώδικα που αφορά την ανάκτηση των δεδομένων, την προσωρινή τους αποθήκευση, τη σελιδοποίηση και τη διαρκή ανανέωση (polling). Επίσης, επιτρέπει την αυτόματη δημιουργία hooks για την πραγματοποίηση αιτημάτων προς APIs και

Ένα από τα βασικά χαρακτηριστικά του RTK Query είναι η προσέγγιση "data-first", όπου η κατάσταση της εφαρμογής καθορίζεται κυρίως από τα δεδομένα που ανακτώνται από απομακρυσμένες πηγές. Αντί να ανανεώνονται τα δεδομένα χειροκίνητα στο context που παρέχει και να απαιτείται ρητός συγχρονισμός, το RTK Query διαχειρίζεται αυτόματα την προσωρινή αποθήκευση και την ανανέωση των δεδομένων. Αυτό μειώνει σημαντικά την πιθανότητα εμφάνισης σφαλμάτων λόγω ασυνεπειών στην κατάσταση της εφαρμογής, ενώ παράλληλα καθιστά τον κώδικα του παρασκηνίου πιο καθαρό και εστιασμένο στην επιχειρησιακή λογική.

Συγκεκριμένα, παρέχει αυτόματους μηχανισμούς για την ανάκτηση και την προσωρινή αποθήκευση. Το RTK Query χρησιμοποιεί ετικέτες για τη διαχείριση της cache της εφαρμογής. Συγκεκριμένα, εξάγονται δεδομένα προγραμματιστικά ή/και χειροκίνητα από τα τα αποτελέσματα ενός API Call, όπως φαίνεται παρακάτω, τα οποία χρησιμοποιούνται ως ετικέτες στην key-value εσωτερική δομή για την cache. Αναλυτικά:

```
    getDonations: build.query({
      query: ({ page = 1, pageSize = 5 } = {}) => ({
        url: "/donation/",
        params: { page, pageSize },
      }),
      providesTags: (result, error, arg) =>
        result ? [
          { type: "Donation", id: "Donation-PARTIAL-LIST" },
          ...result.data.map(({ id }) => ({ type: "Donation",
id }))
         ] : []
    })
```

Κώδικας 17: RTK query: build query

Ο κώδικας εξασφαλίζει την ανάκτηση δεδομένων από ένα συγκεκριμένα endpoint, αυτό για τις δωρεές. Στο πεδίο *providesTags*, φαίνεται ότι ενημερώνεται η cache με ετικέτες οι οποίες έχουν σαν τύπο το κλειδί “Donation” και σαν αναγνωριστικό, το id του κάθε νεόφερτου donation και το αναγνωριστικό “Donation-PARTIAL-LIST”, το οποίο αποτελεί το αναγνωριστικό της λίστας δωρεών που ανακτώνται για την αρχική οθόνη.

Αντίστοιχα με τους μηχανισμούς ανάκτησης, υπάρχει μηχανισμός ακύρωσης των υπάρχοντων εγγραφών με σκοπό την εκ νέου ανάκτηση.

```
  addDonation: build.mutation({
      query: ({ requestId, donation }) => ({
        url: `/donation/request/${requestId}`,
        method: "POST",
        body: donation,
      }),
      invalidatesTags: (result, error, { requestId }) => [
          result ? { type: "DPR", id: "DPR-PARTIAL-LIST" },
          { type: "Donation", id: "Donation-PARTIAL-LIST" },
          { type: "Request", id: requestId }
         ] : []
    }),
```

Κώδικας 18: RTK query: build mutation

Στην περίπτωση παραπάνω, έχουμε τη δήλωση για την πράξη προσθήκης νέας δωρεάς. Από τη δήλωση στην πρώτη γραμμή, φαίνεται ότι αυτή η πράξη αποτελεί *mutation*, δηλαδή θα μεταλλάξει την τρέχουσα κατάσταση. Για αυτό μας δίνεται η δυνατότητα μέσω του πεδίου *invalidateTags* να ορίσουμε ποιες ετικέτες των προσωρινά αποθηκευμένων δεδομένων θα

τροποποιηθούν. Έτσι, κατά την επιτυχή προσθήκη νέας δωρεάς ενημερώνονται τρεις ετικέτες. Αυτή του αιτήματος στο οποίο αναφέρεται η δωρεά, γιατί έχει τροποποιηθεί η κατάσταση των αντικειμένων/υπηρεσιών που το αφορούν, της λίστας των δωρεών ανά αίτημα (**D**onation**P**er**R**equest) και των δωρεών γενικά.

Το RTK παρέχει ακόμα λεπτομερή έλεγχο όσον αφορά την εκτέλεση και την επανεκτέλεση μιας πράξης ανάκτησης δεδομένων μέσω των hooks που παρέχει. Τα εν λόγω hooks είναι άμεσα συνυφασμένα με δηλώσεις endpoints όπως τις παραπάνω που είδαμε, τόσο για queries (απλή ανάκτηση και εναπόθεση στην cache), όσο και για mutations (τροποποίηση ήδη υπάρχουσας εγγραφής).

```
const { data, isLoading, isSuccess, isError, error } =
    useGetDonationsQuery(
        { page, pageSize: rowsPerPage },
        {
          skip: condition,
          refetchOnMountOrArgChange: true,
          refetchOnFocus: true,
        },
      );
```

Κώδικας 19: Χρήση RTK query

Το παραπάνω απόσπασμα δηλώνεται στο σώμα ενός component. Η χρήση του useGetXQuery στο RTK Query παρέχει διαφορετικές δυνατότητες ανάκτησης δεδομένων, ανάλογα με τις ανάγκες της εφαρμογής. Το useGetXQuery είναι ένα hook που εκτελεί αυτόματα ένα αίτημα κατά την αρχικοποίηση του component, επιστρέφοντας μια σειρά από τιμές όπως data, error, isLoading, isFetching και refetch. Αυτές οι τιμές επιτρέπουν στον προγραμματιστή να παρακολουθεί την πορεία του αιτήματος και να διαχειρίζεται δυναμικά την εμφάνιση ή την επαναφόρτωση των δεδομένων. Παράλληλα, το RTK παρέχει και το use**Lazy**GetXQuery, στις τιμές επιστροφής του οποίου, περιλαμβάνεται επιπλέον μια συνάρτηση ενεργοποίησης του αιτήματος (trigger), δίνοντας έτσι τη δυνατότητα καθυστερημένης εκτέλεσης του query, μόνο όταν αυτό καταστεί αναγκαίο (π.χ. μετά από ενέργεια χρήστη). Τέλος, και τα δύο hooks μπορούν να δεχτούν παραμέτρους όπως pollingInterval, refetchOnMountOrArgChange, και skip, παρέχοντας ευελιξία στον έλεγχο της συχνότητας εκτέλεσης, της συμπεριφοράς ανανέωσης, και της προαιρετικής παράκαμψης του αιτήματος.

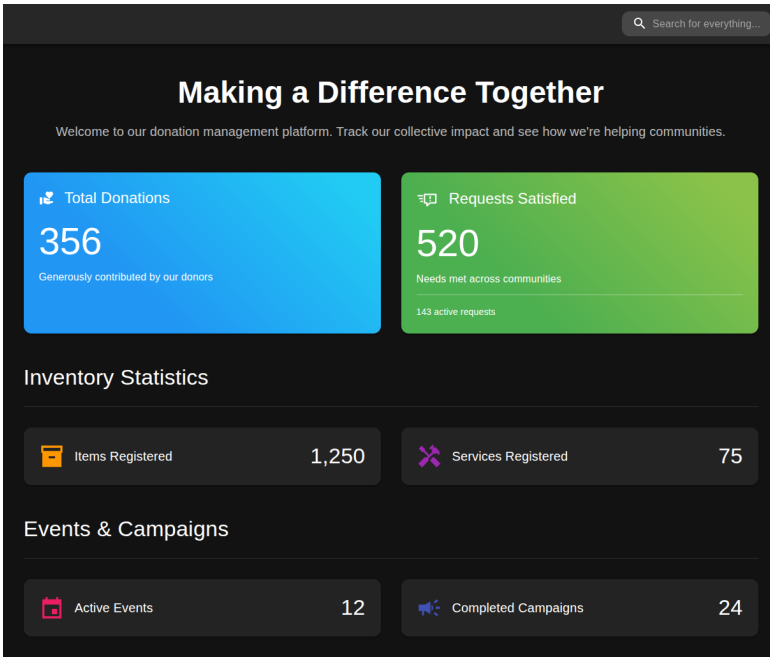
Τα δύο αυτά hooks δημιουργούνται αυτόματα από το RTK Query με τη δημιουργία ενός πεδίου query ή mutation όπως αυτά που είδαμε παραπάνω.

Κεφάλαιο 5

Στο παρόν κεφάλαιο θα δούμε στιγμιότυπα από την εφαρμογή και όλες τις σελίδες στις οποίες μπορεί να πλοηγηθεί ο χρήστης κατά τη χρήση της.

Αρχική σελίδα

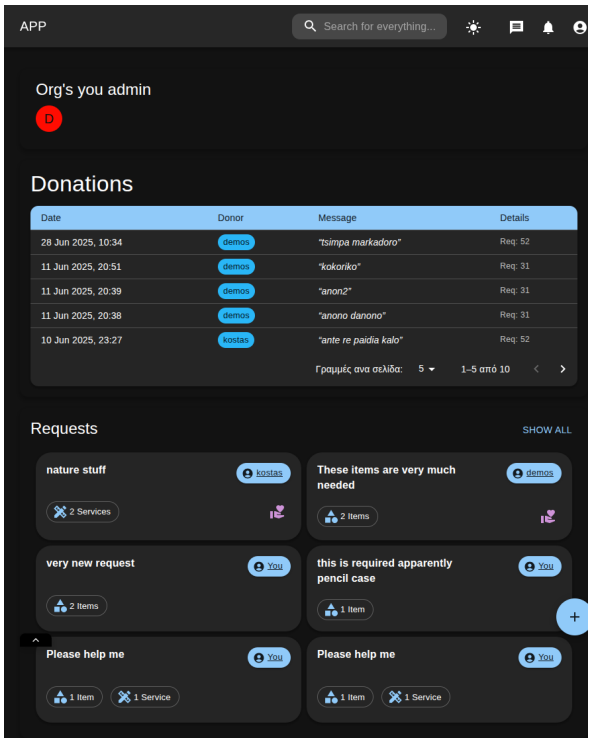
Η αρχική σελίδα περιλαμβάνει πληροφορίες σχετικά με γενικά στατιστικά της εφαρμογής, όπως τις συνολικές δωρεές, τα συνολικά αιτήματα που έχουν εξυπηρετηθεί



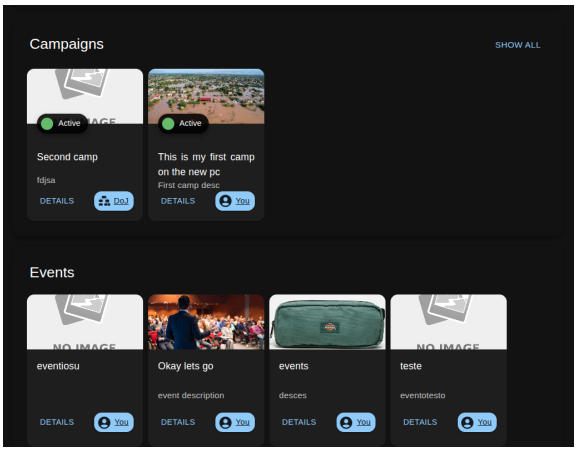
Screenshot 1: Αρχική οθόνη χωρίς σύνδεση

Αρχική σελίδα (σύνδεση)

Στο παρακάτω στιγμιότυπο φαίνεται η αρχική σελίδα αφού συνδεθεί ο χρήστης. Χωρίζεται σε 6 διακριτές κατηγορίες. Την μπάρα πλοήγησης, ένα πεδίο με τους διαθέσιμους οργανισμούς τους οποίους είναι διαχειριστής ο χρήστης, ένα πεδίο με τις πιο πρόσφατες δωρεές, ένα με το πιο πρόσφατα αιτήματα, ένα με εκδηλώσεις και ένα με καμπάνιες.



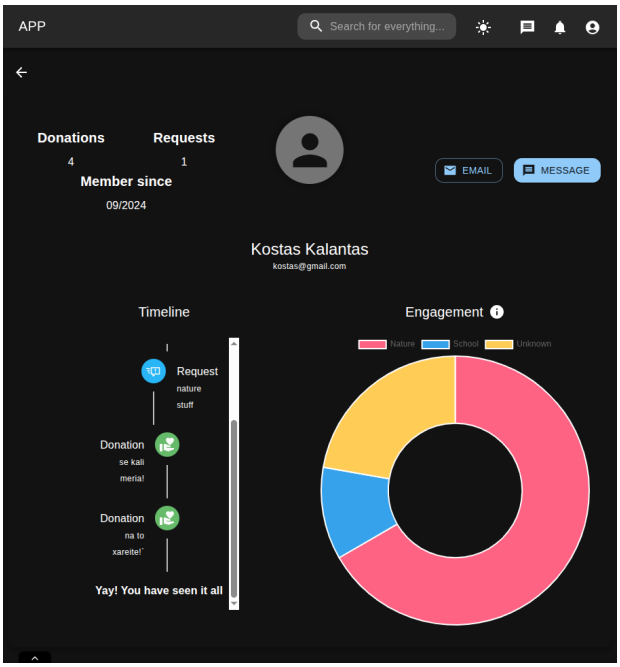
Screenshot 2: Αρχική οθόνη με σύνδεση (1)



Screenshot 3: Αρχική οθόνη με σύνδεση (2)

Σελίδα προφίλ χρήστη

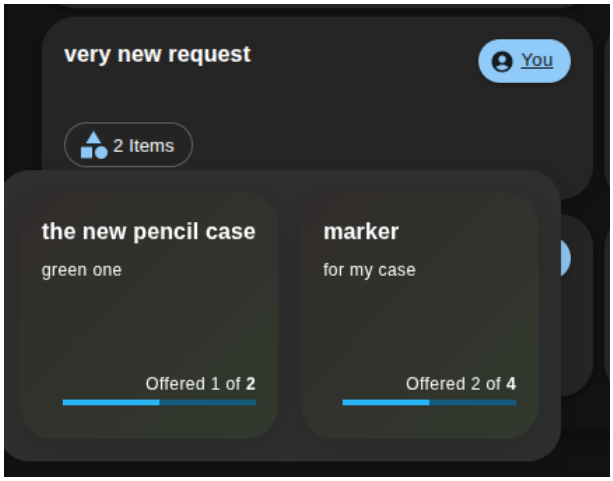
Στην παρούσα σελίδα φαίνεται το ιστορικό ενεργειών κάθε χρήστη, όσον αφορά αιτήματα και δωρεές, αλλά και μια απεικόνιση των κατηγοριών με τη μεγαλύτερη αλληλεπίδραση.



Screenshot 4: Σελίδα προφίλ χρήστη

Καρτέλα αιτήματος

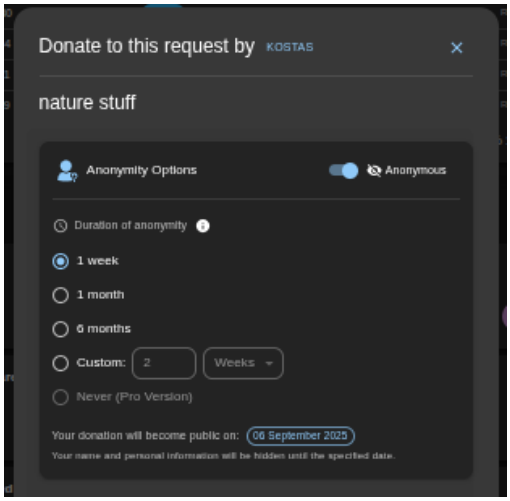
Στην καρτέλα αιτήματος εμφανίζονται πληροφορίες σχετικά με το αίτημα και τις παροχές τις οποίες αιτείται. Με αιώρηση πάνω από τις πληροφορίες των παροχών εμφανίζεται περαιτέρω πληροφορία για αυτές, όπως η απαιτούμενη ποσότητα, φωτογραφίες, περιγραφή κλπ.



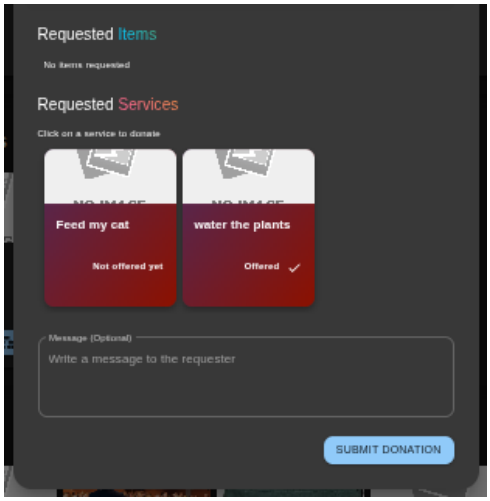
Screenshot 5: Hover Αιτούμενες παροχές

Φόρμα δημιουργίας δωρεάς

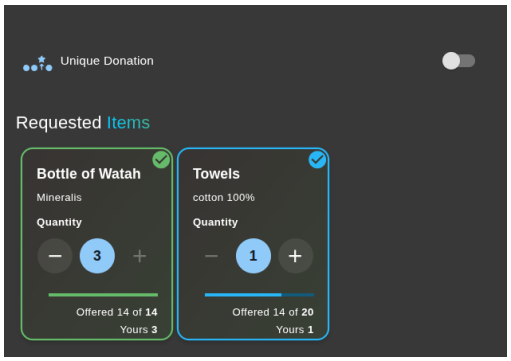
Στη φόρμα δημιουργίας δωρεάς ο χρήστης έχει τη δυνατότητα επιλογής διατήρησης της ανωνυμίας του δωρητή, επιλογής μοναδικής προσφοράς αν το επιτρέπει ο αιτών και επιλογής ποσότητας των αντικειμένων. Τέλος δίνεται η δυνατότητα προσθήκης μηνύματος.



Screenshot 6: Φόρμα καταχώρησης δωρεάς (1)



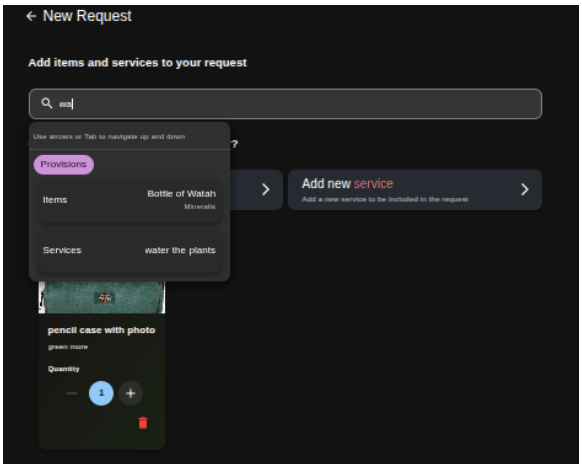
Screenshot 7: Φόρμα καταχώρησης δωρεάς (2)



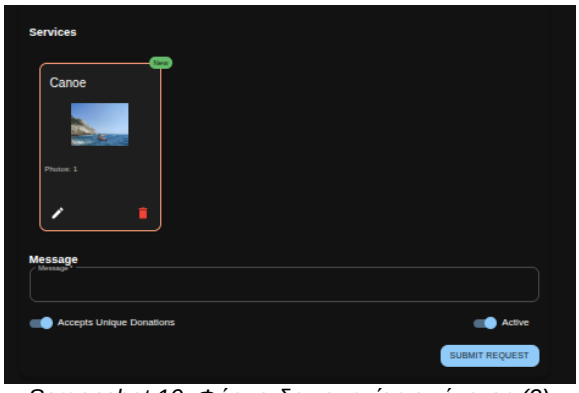
Screenshot 8: Επιλογέας αιτούμενων παροχών

Φόρμα δημιουργίας αιτήματος

Στη φόρμα ο χρήστης μπορεί να προσθέσει νέα αντικείμενα και δωρεές, αν δεν τον ικανοποιούν τα ήδη υπάρχοντα τα οποία είναι διαθέσιμα μέσω της αναζήτησης. Δύναται επίσης να προσθέσει κάποιο μήνυμα.

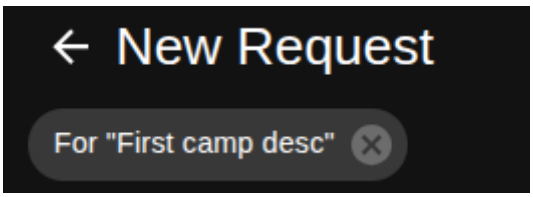


Screenshot 9: Φόρμα δημιουργίας αιτήματος (1)



Screenshot 10: Φόρμα δημιουργίας αιτήματος (2)

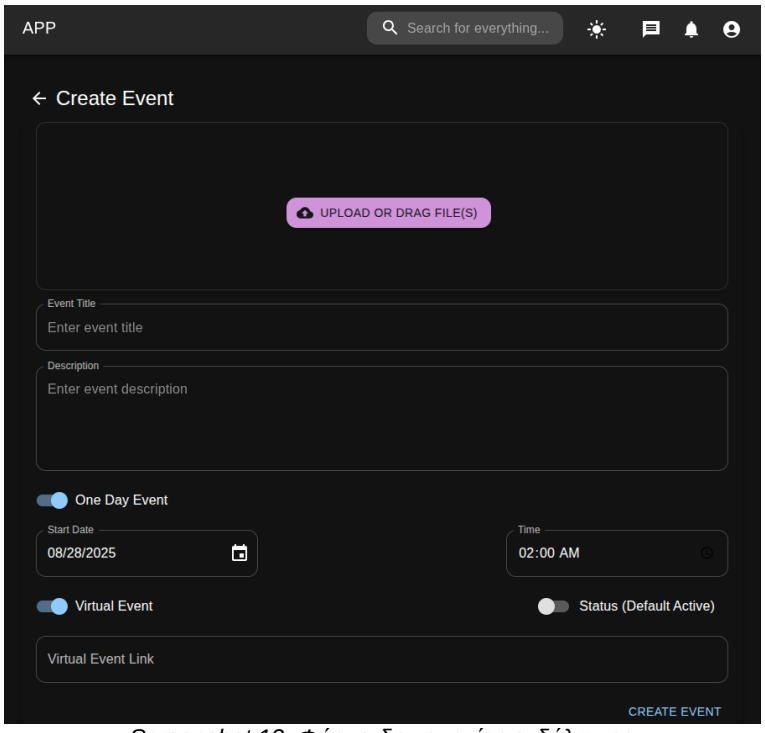
Στην ίδια φόρμα καταλήγει κάποιος χρήστης όταν θέλει να προσθέσει αιτούμενα αντικείμενα ή υπηρεσίες στα πλαίσια μιας καμπάνιας.



Screenshot 11: Αίτημα υπό καμπάνια

Φόρμα δημιουργίας εκδήλωσης

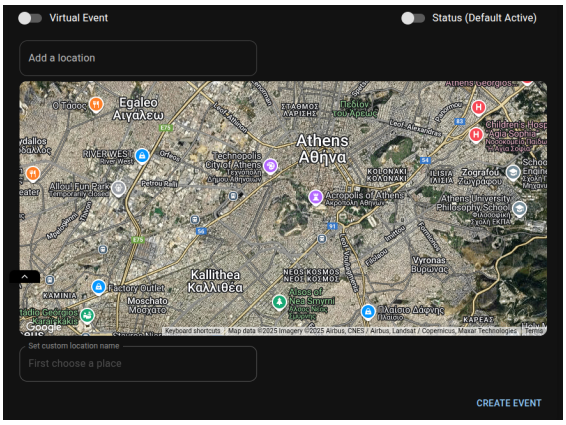
Στην παρακάτω φόρμα ο χρήστης εκτός από φωτογραφίες, ημερομηνίες, τίτλο και περιγραφή μπορεί να επιλέξει ανάμεσα στο αν η εκδήλωση είναι διαδικτυακή ώστε να παρέχει κάποιο σύνδεσμο ή δια ζώσης ώστε να παρέχει ακριβής τοποθεσία μέσω της χρήσης χάρτη.



Screenshot 12: Φόρμα δημιουργίας εκδήλωσης

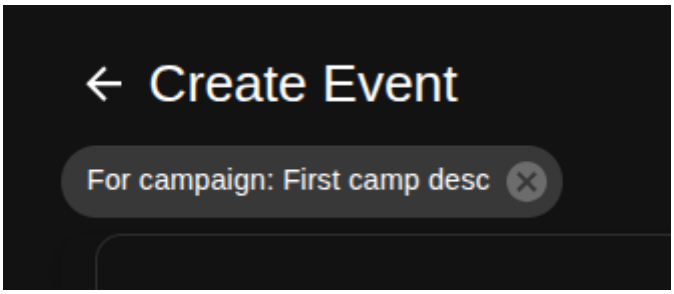
Με χάρτη

Ο χρήστης μπορεί να επιλέξει όνομα της επιλογής του για την επιλεγμένη τοποθεσία.



Screenshot 13: Εκδήλωση με φυσική παρουσία

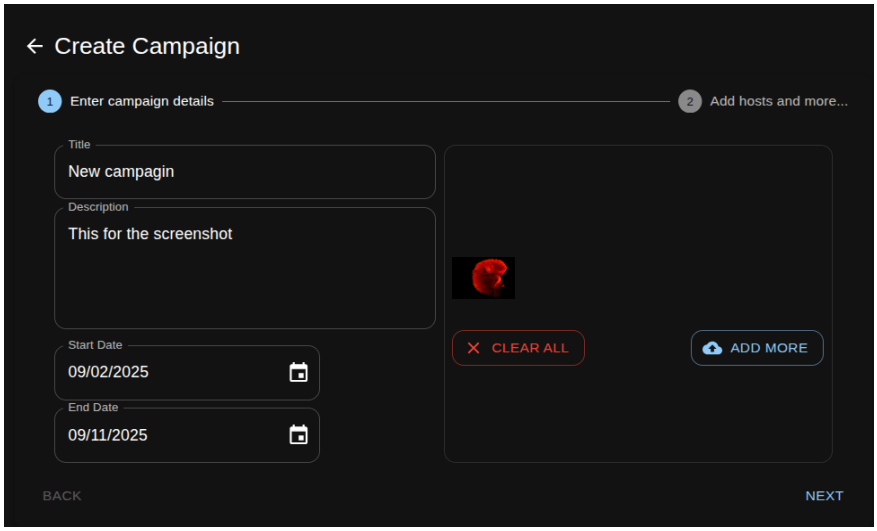
Κατά αντιστοιχία με τα αιτήματα, δεδομένου ότι οι εκδηλώσεις μπορούν να λαμβάνουν μέρος υπό κάποια καμπάνια, έχουμε το αντίστοιχο χαρακτηριστικό όταν αυτό συμβαίνει



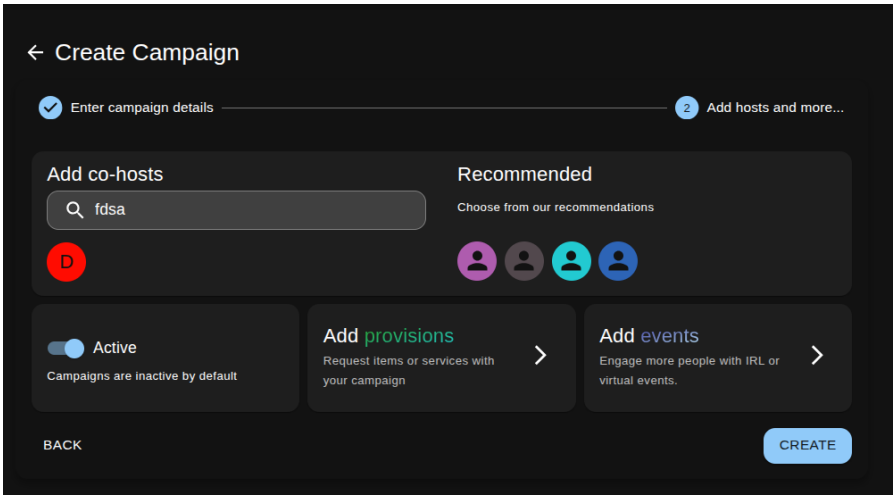
Screenshot 14: Εκδήλωση υπό καμπάνια

Φόρμα δημιουργίας καμπάνιας

Κατά τη δημιουργία καμπάνιας, ο χρήστης καλείται να επιλέξει όνομα, περιγραφή και ημερομηνία έναρξης και λήξης. Δίνεται η δυνατότητα να προστεθούν φωτογραφίες, όπως και να γίνει επιλογή συνδιοργανωτών. Τέλος ο χρήστης μπορεί να μεταβεί στις παραπάνω φόρμες ώστε να προσθέσει εκδηλώσεις και ζητούμενες παροχές στην καμπάνια του.



Screenshot 15: Φόρμα δημιουργίας καμπάνιας (1)

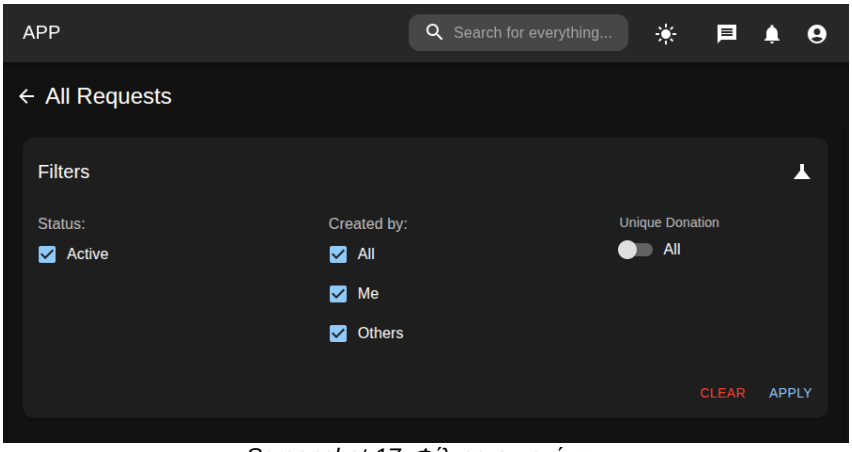


Screenshot 16: Φόρμα δημιουργίας καμπάνιας (2)

Λίστα οντότητας (αιτήματος)

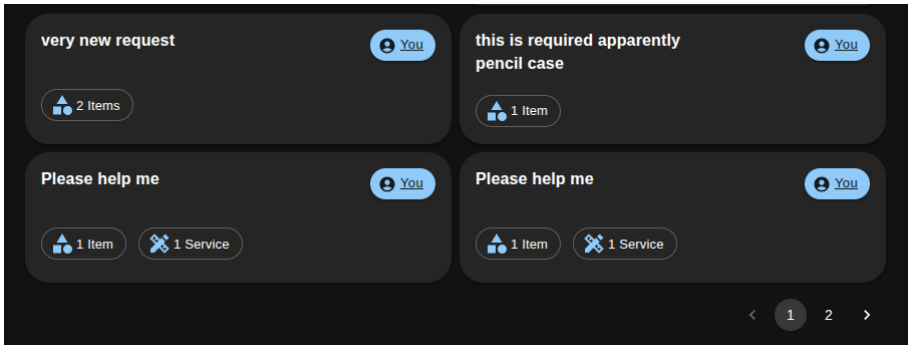
Φίλτρα

Φίλτρα παρέχονται στις δύο σελίδες-λίστες για τα αιτήματα και τις καμπάνιες.



Screenshot 17: Φίλτρα οντοτήτων

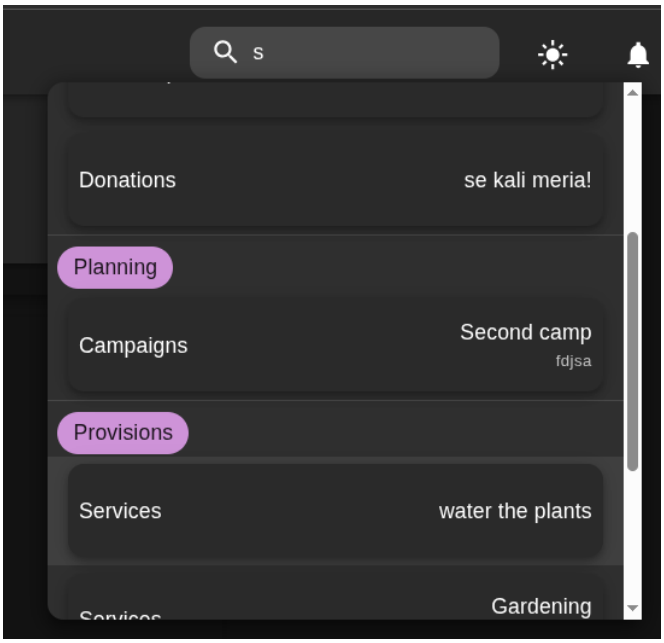
Με σελιδοποίηση



Screenshot 18: Λίστα οντοτήτων με σελιδοποίηση

Υπηρεσία αναζήτησης

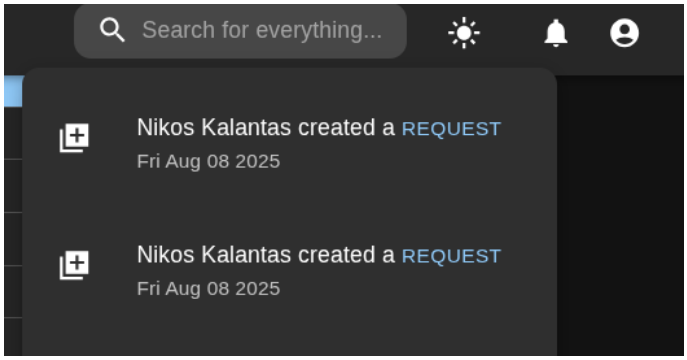
Η υπηρεσία αναζήτησης παρουσιάζεται με ένα custom react component. Κατηγοριοποιεί τα αποτελέσματα ανά elastic index. Με χρήση κατάλληλου πεδίου στον κώδικα μπορεί κανείς να περιορίσει το εύρος αναζήτησης ανά index. Έτσι, στην αρχική σελίδα για παράδειγμα, η αναζήτηση γίνεται σε όλες τις εγγραφές ανεξαρτήτου index, ενώ μέσα στην εφαρμογή κατά τη δημιουργία αιτήματος αναζητούνται μόνο αντικείμενα και υπηρεσίας ενώ κατά τη δημιουργία καμπάνιας αναζητούνται μόνο προφίλ όταν είναι να γίνει η επιλογή συνδιοργανωτών.



Screenshot 19: Κουτί αποτελεσμάτων αναζήτησης

Ειδοποιήσεις

Λίστα ειδοποιήσεων η οποία παρέχει πληροφορίες σχετικά με τον εκτελεστή της ενέργειας η οποία πυροδότησε την ειδοποίηση και για την ενέργεια την ίδια.



Screenshot 20: Κουτί αποτελεσμάτων ειδοποιήσεων

Κεφάλαιο 6 (future work)

Η παρούσα web εφαρμογή έχει ήδη υλοποιηθεί και προσφέρει τις βασικές λειτουργικότητες που απαιτούνται για την εξυπηρέτηση των χρηστών της. Ωστόσο, όπως συμβαίνει σε κάθε σύγχρονο πληροφοριακό σύστημα, η πρώτη έκδοση αποτελεί μονάχα την αρχή μιας συνεχούς διαδικασίας βελτίωσης. Στο πλαίσιο αυτό, είναι αναγκαίο να εξεταστούν μελλοντικές κατευθυντήριες γραμμές ανάπτυξης, οι οποίες θα βελτιώσουν την εφαρμογή εν τω συνόλω.

Οι τρεις κύριοι άξονες που αναλύονται σε αυτό το κεφάλαιο είναι:

1. Η προσωποποιημένη προσαρμογή των λειτουργιών προς τον τελικό χρήστη.
2. Η δημιουργία προφίλ μνήμης ανά υπηρεσία για την καλύτερη παρακολούθηση και βελτιστοποίηση των υπηρεσιών.
3. Η στρατηγική ενίσχυσης της επεκτασιμότητας και της διαχείρισης πόρων ώστε η εφαρμογή να ανταποκριθεί σε αυξημένη ζήτηση στο μέλλον.
4. Εκτεταμένη δοκιμή (extensive testing).

Προσωποποιημένη προσαρμογή

Η τρέχουσα υλοποίηση της εφαρμογής παρέχει κοινή εμπειρία χρήσης για όλους τους χρήστες. Αν και αυτό διασφαλίζει την ομοιογένεια, δεν εκμεταλλεύεται τις δυνατότητες που προσφέρουν τα εξατομικευμένα δεδομένα. Στο μέλλον, η εισαγωγή μηχανισμών προσωποποιημένης προσαρμογής θα επιτρέψει:

- Καθοδήγηση με βάση το ιστορικό: Η εφαρμογή θα μπορεί να προτείνει στον χρήστη λειτουργίες ή περιεχόμενο με βάση προηγούμενες ενέργειες.
- Διαμόρφωση προφίλ χρηστών: Δημιουργία και αποθήκευση προτιμήσεων, ώστε κάθε χρήστης να έχει προσαρμοσμένο περιβάλλον.
- Εφαρμογή συστημάτων συστάσεων: Χρήση αλγορίθμων μηχανικής μάθησης για την πρόβλεψη ενδιαφερόντων και την παροχή σχετικού περιεχομένου.

Η μελλοντική αυτή κατεύθυνση δεν στοχεύει μόνο στη βελτίωση της εμπειρίας χρήστη, αλλά και στην αύξηση της αλληλεπίδρασης με το σύστημα, κάτι που ενδέχεται να έχει θετική επίδραση στη διατήρηση και ανάπτυξη της βάσης χρηστών.

Προκλήσεις

- Χρειάζεται ιδιαίτερη προσοχή σε ζητήματα ιδιωτικότητας και συμμόρφωσης με το GDPR.
- Υπάρχει ο κίνδυνος overfitting, η οποία μπορεί να περιορίσει την ποικιλία του περιεχομένου.

Services Memory Profiling ως εργαλείο βελτιστοποίησης

Η τρέχουσα έκδοση της εφαρμογής λειτουργεί αποτελεσματικά, ωστόσο δεν υπάρχει ακόμα οργανωμένος μηχανισμός για την παρακολούθηση της χρήσης μνήμης ανά υπηρεσία. Καθώς η εφαρμογή μεγαλώνει και προστίθενται νέες λειτουργίες, η πολυπλοκότητα αυξάνεται, και μαζί της αυξάνονται και οι απαιτήσεις σε πόρους.

Στο μέλλον, η προσέγγιση με προφίλ μνήμης θα επιτρέψει:

- Τον εντοπισμό πιθανών διαρροών μνήμης (memory leaks) σε πραγματικό χρόνο.
- Την καταγραφή των μοτίβων κατανάλωσης πόρων ώστε να είναι εφικτή όποια βελτιστοποίηση σε επίπεδο κώδικα
- Τη βελτίωση της σταθερότητας του συστήματος, αποτρέποντας ανεπιθύμητες διακοπές λόγω υπερφόρτωσης.

Εργαλεία που μπορούν να αξιοποιηθούν

- **pprof** για εφαρμογές σε Go, το οποίο παρέχει αναλυτικά προφίλ μνήμης και CPU.
- Διάφορα εργαλεία για δημιουργία dashboards με σκοπό την απεικόνιση μετρικών του συστήματος
- Container-level monitoring μέσω container manager (πχ k8s) για απομόνωση και παρακολούθηση κάθε υπηρεσίας ξεχωριστά.

Επεκτασιμότητα και διαχείριση πόρων ως στρατηγική ανάπτυξης

Μέχρι στιγμής, η εφαρμογή ανταποκρίνεται με επιτυχία στον τρέχοντα όγκο αιτημάτων και χρηστών. Ωστόσο, τυχούσα αναμενόμενη αύξηση της χρήσης στο μέλλον καθιστά απαραίτητη τη μελέτη της επεκτασιμότητας και της διαχείρισης πόρων.

Προτεινόμενες μελλοντικές κατευθύνσεις

- Υλοποίηση οριζόντιας επεκτασιμότητας με κατανομή φορτίου (load balancing) σε πολλαπλούς κόμβους.
- Auto-scaling πολιτικές σε περιβάλλον αυτόματης διαχείρισης ώστε να προστίθενται αυτόματα νέοι πόροι όταν αυξάνεται η ζήτηση.
- Βελτιστοποίηση βάσης δεδομένων κυρίως με indexing στη σχεσιακή βάση.
- Ενσωμάτωση παρακολούθησης σε πραγματικό χρόνο (real-time monitoring) για CPU, RAM και network bandwidth, με στόχο την πρόληψη προβλημάτων έναντι της

αντιμετώπισης αυτών εκ των υστέρων.

Εκτεταμένη δοκιμή ως θεμέλιο αξιοπιστίας

Πέρα από τις τρεις προηγούμενες κατευθύνσεις, η συστηματική και εκτεταμένη δοκιμή αποτελεί καθοριστικό πυλώνα για την ποιοτική εξέλιξη της εφαρμογής. Μέχρι στιγμής, οι έλεγχοι έχουν εστιάσει σε λειτουργικές δοκιμές (functional testing), αλλά η μελλοντική στρατηγική πρέπει να είναι πολύ πιο ολοκληρωμένη και πολυδιάστατη.

Δοκιμές σε επίπεδο μονάδας (Unit Testing)

Η κάλυψη με unit tests θα διασφαλίσει ότι κάθε επιμέρους συνιστώσα της εφαρμογής (API endpoints, services, βιβλιοθήκες) λειτουργεί μεμονωμένα όπως αναμένεται.

Δοκιμές ολοκλήρωσης (Integration Testing)

Η εφαρμογή αποτελείται από πολλαπλές υπηρεσίες που συνεργάζονται. Τέτοιες δοκιμές θα ελέγχουν την ομαλή αλληλεπίδραση μεταξύ των υπηρεσιών (π.χ., επικοινωνία microservices, ανταλλαγή δεδομένων με τη βάση PostgreSQL, indexing στο Elasticsearch).

Δοκιμές ασφαλείας (Security Testing)

Μελλοντικά, θα πρέπει να υλοποιηθούν penetration tests και vulnerability scans ώστε να ελαχιστοποιηθούν οι κίνδυνοι παραβίασης. Αυτό είναι κρίσιμο ειδικά σε συνδυασμό με την προσωποποιημένη προσαρμογή, η οποία προϋποθέτει συλλογή και αποθήκευση προσωπικών δεδομένων.

Σχέση μεταξύ των αξόνων μελλοντικής εργασίας

Η εκτεταμένη δοκιμή δεν αποτελεί ξεχωριστό πυλώνα, αλλά λειτουργεί υποστηρικτικά προς όλες τις κατευθύνσεις:

- Για την προσωποποιημένη προσαρμογή, οι δοκιμές χρηστικότητας και ασφαλείας εξασφαλίζουν ότι η νέα λειτουργικότητα είναι αξιόπιστη και συμμορφώνεται με κανονισμούς.
- Για το memory profiling, οι δοκιμές απόδοσης βοηθούν στον εντοπισμό και επιβεβαίωση βελτιώσεων.
- Για την επεκτασιμότητα, οι stress tests και load tests δείχνουν αν το auto-scaling λειτουργεί σωστά.

Με αυτόν τον τρόπο, το testing γίνεται το ενοποιητικό στοιχείο που συνδέει όλες τις μελλοντικές εργασίες.

Συνολική Σύνθεση και Στρατηγική Μελλοντικής Εξέλιξης

Οι τρεις αυτοί άξονες δεν λειτουργούν απομονωμένα αλλά αλληλοσυμπληρώνονται. Η προσωποποιημένη προσαρμογή αναβαθμίζει την εμπειρία του χρήστη. Το services memory profiling παρέχει τη διαγνωστική υποδομή για τη διαχείριση της πολυπλοκότητας που αυτή συνεπάγεται. Και η επεκτασιμότητα με τη σωστή διαχείριση πόρων προσφέρει τη δυνατότητα μακροχρόνιας βιωσιμότητας της εφαρμογής.

Η μελλοντική εργασία, συνεπώς, θα κινηθεί σε δύο επίπεδα:

1. Βραχυπρόθεσμα, με την εισαγωγή μηχανισμών παρακολούθησης και προφίλ μνήμης και ελέγχου.
2. Μακροπρόθεσμα, με την επένδυση σε εξατομίκευση και στρατηγική επεκτασιμότητας, ώστε η εφαρμογή να μπορεί να εξυπηρετήσει πολλαπλάσιους χρήστες με τον ίδιο βαθμό απόδοσης.

Βιβλιογραφία

[1] Kaufmann, D., & Bellver, A. (2005). Transparenting transparency: Initial empirics and policy applications. *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.808664

[2] Smith, S. (2012). *Mind the gap: The growing generational divide in charitable giving* (p. 16). Charities Aid Foundation.

[3] PostgreSQL Global Development Group. (n.d.). Non-durable settings. In *PostgreSQL documentation*. https://www.postgresql.org/docs/current/non-durability.html

[4] Kabakus, A. T., & Kara, R. (2017). A performance evaluation of in-memory databases. *Journal of King Saud University - Computer and Information Sciences*, *29*(4), 520–525. https://doi.org/10.1016/j.jksuci.2016.06.007

[5] Statista. (n.d.). *Global mobile traffic 2025*. Retrieved from https://www.statista.com/statistics/277125/share-of-website-traffic-coming-from-mobile-devices/

[6] Go Team. (n.d.). Database/sql package. In *Go packages*. Retrieved from https://pkg.go.dev/database/sql#DB

[7] Go Team. (n.d.). Share memory by communicating. *The Go Programming Language Blog*. Retrieved from https://go.dev/blog/codelab-share

[8] Mozilla. (2024, July 24). FormData — Web APIs. *MDN Web Docs*. https://developer.mozilla.org/en-US/docs/Web/API/FormData

[9] Faker Community. (n.d.). *Welcome to Faker's documentation!* (Version 37.6.0). Retrieved from https://faker.readthedocs.io/en/master/

[10] *time(1) — Linux manual page*. (n.d.). Retrieved from https://man7.org/linux/man-pages/man1/time.1.html

[11] Elastic. (2021, March 4). Elasticsearch caching deep dive: Boosting query speed one cache at a time. *Elastic Blog*. https://www.elastic.co/blog/elasticsearch-caching-deep-dive-boosting-query-speed-one-cache-at-a-time

[12] Evans, E. (2003). *Domain-driven design: Tackling complexity in the heart of software*. Addison-Wesley.

[13] Elastic. (n.d.). The shard request cache. *Elasticsearch documentation*. Retrieved from https://www.elastic.co/docs/reference/elasticsearch/rest-apis/shard-request-cache

[14] Redux Team. (n.d.). *Redux Toolkit*. Retrieved from https://redux-toolkit.js.org/

[15] Atlassian. (n.d.). *Microservices vs. monolithic architecture*. Retrieved from https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith