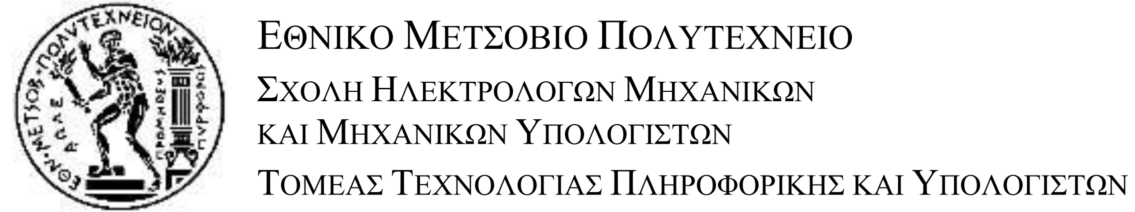


ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

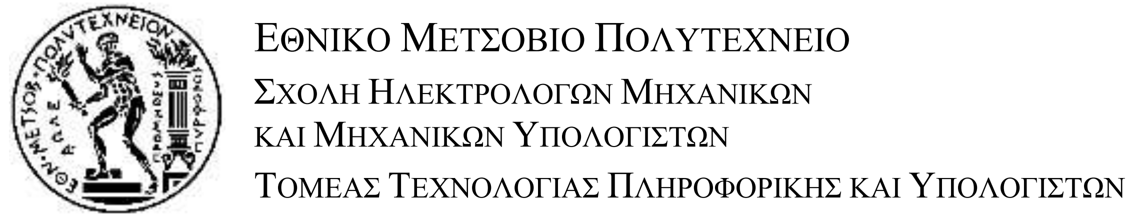
Αρχιτεκτονικές λογισμικού: Ανάπτυξη δικτυακής εφαρμογής γραφικής απεικόνισης χρονοσειρών με χωρική αναφορά με χρήση πρότυπων υπηρεσιών web

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Δημήτριος, Κ. Καραγιάννης

Επιβλέπων : Βασίλειος Βεσκούκης
Αναπληρωτής Καθηγητής

Αθήνα, Σεπτέμβριος 2021



Αρχιτεκτονικές λογισμικού: Ανάπτυξη δικτυακής εφαρμογής γραφικής απεικόνισης χρονοσειρών με χωρική αναφορά με χρήση πρότυπων υπηρεσιών web

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Δημήτριος, Κ. Καραγιάννης

Επιβλέπων : Βασίλειος Βεσκούκης
Αναπληρωτής Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 15^{η} Ιουλίου 2021.

………..............................	………..............................	………..............................
Βασίλειος Βεσκούκης	Νικόλαος Παπασπύρου	Παναγιώτης Τσανάκας
Αναπληρωτής Καθηγητής	Καθηγητής	Καθηγητής

Αθήνα, Σεπτέμβριος 2021

....
Δημήτριος, Κ. Καραγιάννης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Δημήτριος Καραγιάννης, 2021
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Σήμερα είμαστε ήδη στην εποχή των εφαρμογών, όπου ο κάθε χρήστης έχει πρόσβαση σε απεριόριστα δεδομένα και υπηρεσίες οι οποίες υπάρχουν στο διαδίκτυο. Υπάρχει μια πληθώρα τεχνολογιών και μεθοδολογιών που επιτρέπουν σε έναν μηχανικό να μελετήσει, να σχεδιάσει και εν τέλει να υλοποιήσει το σύστημα που θέλει να φτιάξει. Πολύ νωρίς στον σχεδιασμό θα χρειαστεί να επιλέξει μια αρχιτεκτονική στην οποία θα ακολουθήσει. Έχει πολλές επιλογές όπως το MVC, τα microservices, τα SOA, APIs. Όλες έχουν τα πλεονεκτήματα και τα μειονεκτήματα τους. Έπειτα θα πρέπει να επιλέξει τις τεχνολογίες που θα χρησιμοποιήσει, γλώσσες προγραμματισμού, βιβλιοθήκες, προγράμματα, επιλογές που αν γίνουν σωστά θα τον διευκολύνουν, ενώ αν δεν γίνουν τότε θα χάσει πολύ χρόνο με πράγματα σε δευτερεύοντα πράγματα.

Προσπαθώντας να κατασκευαστεί μια πλατφόρμα για την αναπαράσταση χρονοσειρών ενεργειακών δεδομένων, τα οποία συνδέονται με μια γεωαναφορά (σταθμός παραγωγής, αιολικό πάρκο, κτλ.) και ακολουθώντας την ενδεικνυόμενη μεθοδολογία σχεδιασμού, δημιούργησα μια πλατφόρμα στην οποία ο χρήστης μπορεί να αναζητήσει δεδομένα που τον ενδιαφέρουν μέσω χάρτη και κλασικής αναζήτησης μέσα από πεδία σε μία φόρμα, η οποία φιλοξενείτε σε ένα GUI σχεδιασμένο για την μέγιστη παραγωγικότητα. Όλη η αρχιτεκτονική καταγράφθηκε με διαγράμματα μέσω του Visual Paradigm και δόθηκε έμφαση στην πλήρη περιγραφή όσων έγιναν. Χρησιμοποιήθηκε ένα Geoserver για την παρουσίαση των γεωδεδομένων τα οποία παρουσιάζονται μέσω της βιβλιοθήκης Mapbox. Τα metadata αποθηκεύονται σε μία Postgres και σερβίρονται μέσα από ένα API γραμμένο σε Javascript με ExpressJS. Για την κατασκευή του GUI χρησιμοποιήθηκε η Svelte για το λογικό κομμάτι και η TailwindCSS για το styling. Προφανώς και γίνεται ενδελεχή ανάλυση αυτόν των επιλογών καθώς και παρουσίαση τους.

Λέξεις Κλειδιά

MVC, Microservices, SOA, APIs, Web, GeoServer, WFS, Svelte, TailwindCSS

Abstract

Nowadays we live at the age of apps. Everyone has access to an infinite amount of data and services to interact. If an engineer wants to implement a service like that or utilize a big set of data to draw conclusion, he is presented with a huge number of options. These tools come in many forms. Some are used to help you design the app (or service), create it, or even observer its performance later. The first thing he would have to decide is the overall architecture to use, an option that will affect every decision he will make later both negatively and positively. Some real popular options are MVC, Microservice, SOA or web API. Next he will have to decide the language, the libraries, the software he is gone use to implement his design. On this stage he should make choices that will help him finish the project quickly without compromising on quality.

I tried to create a platform for managing spatially-referenced observations using standardized web services and choosing the best architecture for the job. The user is able to search on the page both visually and through a form to find the data he wants to export, focusing on maximizing the usability of the platform. The entire architecture was documented using diagrams in Visual Paradigm, with focus on describing all the aspect of the project the best way possible. I used a Geoserver to serve the geo-references we store that are displayed with the Mapbox library. Both metadata and data are store in a Postgres database and are provided to the user through an API we also designed with ExpressJS in Javacript. The GUI is made using Svelte and Taildwind, two libraries that focus no making the development experience better without compromising on performance. All these option I took will be analyzed thoroughly.

Key Words

MVC, Microservices, SOA, APIs, Web, GeoServer, WFS, Svelte, TailwindCSS

Πίνακας περιεχομένων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ....1
ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ....3
Περίληψη....5
Λέξεις Κλειδιά....5
Abstract....6
Key Words....6
Chapter 1: Γενικά για αρχιτεκτονικές λογισμικού....10
1.1 Πρότυπα περιγραφής αρχιτεκτονικών....11
1.2 Χαρακτηριστικές αρχιτεκτονικές εφαρμογών web....12
1.2.1 MVC και παράγωγα του....13
1.2.2 Microservices....16
1.2.3 Microservice – Παράδειγμα προς μελέτη....20
1.2.4 SOA....22
1.2.5 SOA – Παράδειγμα προς μελέτη....25
1.2.6 API – Abstract Programming Interface....25
Chapter 2: Προδιαγραφές μιας δικτυακής εφαρμογής για τη γραφική απεικόνιση χρονοσειρών με χωρική αναφορά....32
2.1 Περιπτώσεις χρήσης....32
2.2 Γενικά....32
2.3 Business process: ροές δεδομένων....42
Chapter 3: WFS web service standard για διανυσματικά χωρικά δεδομένα....45
Chapter 4: Σχεδίαση και υλοποίηση της εφαρμογής....48
4.1 Γενική αρχιτεκτονική....48
4.1.1 Βάση δεδομένων και γεω-αναφορές....48
4.1.2 API Service....50
4.1.3 Web app – GUI....53
4.1.4 Web app – Δεδομένα....58
4.2 Διαγράμματα UML (APIs, Sequence, …)....62
4.3 Εργαλεία και τεχνολογίες που χρησιμοποιήθηκαν....67
4.3.1 Typescript....67
4.3.2 Svelte και Rollup....68
4.3.3 Sapper -- SvelteKit....70
4.3.4 Tailwind CSS....71

4.3.5 Mapbox 74
4.3.6 Geoserver 76
4.4 Διάταξη της εφαρμογής στο okeanos 80
Chapter 5: Σχολιασμός της αρχιτεκτονικής και της εμπειρίας 80
Παράρτημα: Δόμηση κώδικα και git Repo 81
Βιβλιογραφία 83

Chapter 1: Γενικά για αρχιτεκτονικές λογισμικού

Η αρχιτεκτονική λογισμικού είναι η δομή (ή οι δομές) του συστήματος, οι σχέσεις μεταξύ των δομικών στοιχείων της δομής (ή των δομών), καθώς και οι ιδιότητες αυτών των στοιχείων. Εάν η σχεδίαση είναι το σύνολο των αποφάσεων που αφορούν τον τρόπο επίλυσης του προβλήματος, τότε η αρχιτεκτονική σχεδίαση είναι το σύνολο των αποφάσεων σχεδίασης που είναι δύσκολο να αλλάξουν.

Επίσης η αρχιτεκτονική ενός πληροφοριακού συστήματος εμπεριέχει αποφάσεις σχετικά με τη λειτουργικότητα (functionality), τη χρηστικότητα (usability), την ανθεκτικότητα (resilience), τις επιδόσεις (performance), την επαναχρησιμοποίηση (reuse), τον εύληπτο χαρακτήρα (comprehensibility), τους οικονομικούς και τους τεχνολογικούς περιορισμούς (economic and technology constraints), αλλά και την αισθητική του συστήματος (aesthetics). [1]

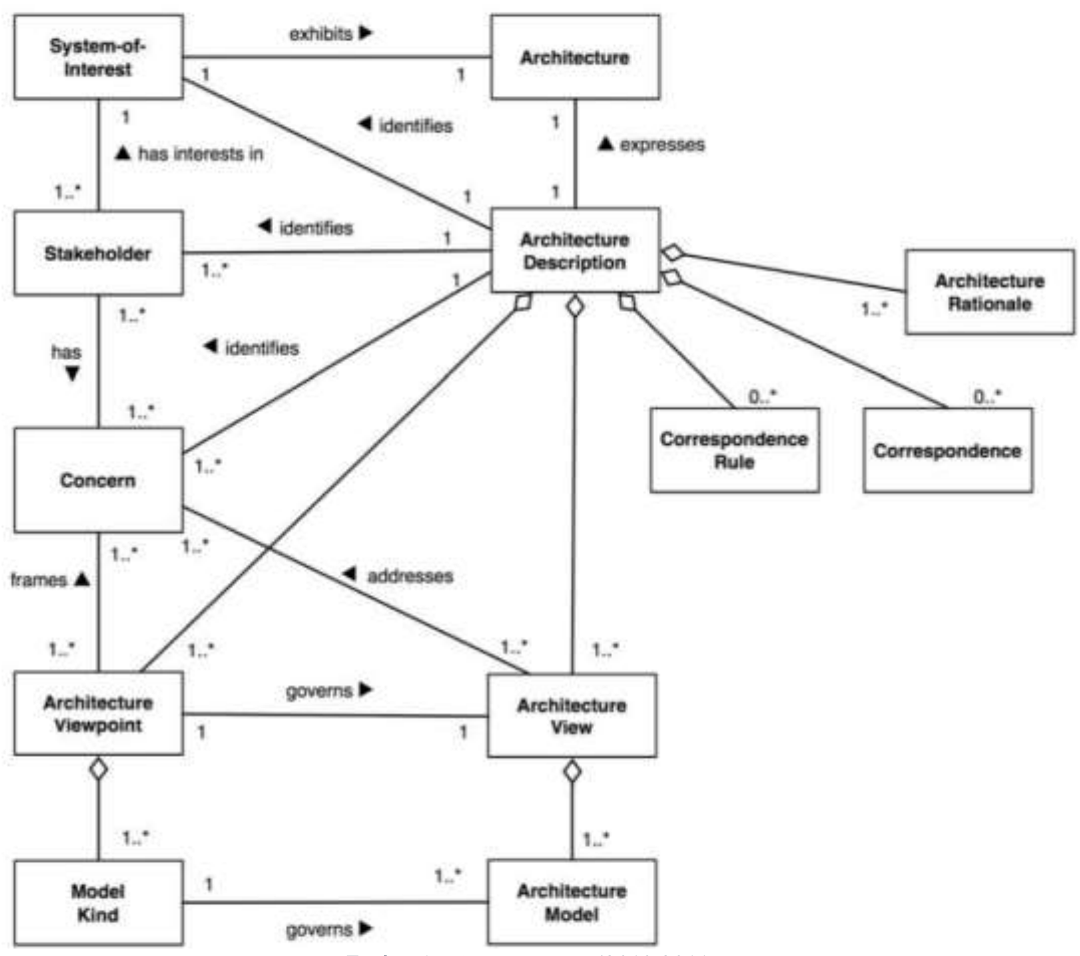
Όταν ένας πρόγραμμα φτιάχνετε από πολλούς προγραμματιστές και πρέπει για κάποιο τρόπο, το σύνολο του κώδικα που θα γράψουν να είναι ομοιογενές και να πετυχαίνει το στόχο για τον οποίο ξεκίνησε η κατασκευή του, πρέπει να μπουν όρια και περιορισμοί στο πως θα δομηθεί ο κώδικας. Αυτά εξασφαλίζουν το ότι το σύστημα θα γίνει εύκολα κατανοητό από τρίτους και πιο εύκολα συντηρήσιμο καθώς η αρχιτεκτονική θα αποτελέσει την βάση πάνω στην οποία θα ξεκινήσει να χτίζεται το σύστημα. Σε project με πολλές γραμμές και πολλά αρχεία πρέπει να υπάρχει ένας τρόπος, που να μπορεί κάποιος ο οποίος θέλει να δει πως είναι υλοποιημένο ένα feature,να το βρει εύκολα χωρίς να έχει πλήρης γνώση του τι βρίσκεται που. Το ίδιο πρόβλημα εμφανίζεται και κατά την συντήρηση καθώς πολύ συχνά το άτομο το οποίο είναι υπεύθυνο να διορθώσει ένα πρόβλημα δεν είναι αυτό που έγραψε την αρχική υλοποίηση και πρέπει κάπως να βρει που βρίσκεται. Όλες οι αρχιτεκτονικές έχουν έναν τρόπο να λύσουν ή έστω να περιορίσουν αυτά τα προβλήματα.

Είναι πολύ σημαντικό όταν στήνεται ένα σύστημα και είναι γνωστό από πριν ότι θα αποτελέσει ένα πολύ μεγάλο λογισμικό οικοδόμημα, να μπουν οι σωστές βάσεις για τις ανάγκες του. Αυτό μπορεί να σημαίνει ότι θα πρέπει να μπουν βαθιά θεμέλια, που να τηρούν πολλές προϋποθέσεις γιατί ο σκοπός είναι ή να χτιστεί ένας ουρανοξύστης ή να μπουν μικρότερα θεμέλια γιατί δεν θα υπάρχουν τόσες ανάγκες από τους χρήστες. Και στις δύο περιπτώσεις πρέπει να επιλεγεί η κατάλληλη αρχιτεκτονική διότι υπάρχουν πολλές αρχιτεκτονικές που κάνουν για την ίδια δουλειά αλλά είναι σχεδιασμένες για άλλου βεληνεκούς συστήματα. Είναι δουλειά του σχεδιαστή να κάνει την σωστή επιλογή.

Για παράδειγμα το να στηθεί ένα web API μπορεί να γίνει με σχεδόν άπειρες διαφορετικές τεχνολογίες, με σχεδόν όλες τις γλώσσες προγραμματισμού. Αν δεν υπάρχουν τεράστιες λειτουργικές απαιτήσεις, μπορούν να χρησιμοποιηθούν όλες οι τεχνολογίες αλλά αν υπάρχουν μεγάλες απαιτήσεις στον αριθμό των παράλληλων χρηστών που πρέπει να υποστηρίζει το σύστημα, πολλές από αυτές δεν θα μπορούν να ανταπεξέλθουν. Θα πρέπει δηλαδή ο σχεδιαστής να ξέρει τουλάχιστον χοντρικά πόσο απαιτητική θα πρέπει να είναι η τεχνολογία που φτιάχνει, γιατί αλλιώς θα πάρει αποφάσεις που αργότερα θα φανούν ως υποδεέστερες των απαιτήσεων.

1.1 Πρότυπα περιγραφής αρχιτεκτονικών

Σήμερα υπάρχουν πολλά πρότυπα περιγραφής αρχιτεκτονικών λογισμικού τα οποία χρησιμοποιούνται από ομάδες/εταιρίες για την δημιουργία και την οργάνωση του κώδικα που υλοποιεί τις λειτουργικές απαιτήσεις ενός συστήματος. Όλα αυτά τα πρότυπα ανεξάρτητα με το ποια είναι η εσωτερική τους δόμηση, πρέπει να περιγράψουν κάποιες βασικές έννοιες, οντότητες και διαδικασίες. [1]



Εικόνα 1: ISO/IEC/IEEE 42010:2011(E)

Στο σχήμα περιγράφεται τι πρέπει μία αρχιτεκτονική να ορίζει και με ποιον τρόπο, ώστε να είναι πλήρης και συμβατή με το πρότυπο. Συγκεκριμένα πρέπει να ορίζεται το *πλαίσιο της αρχιτεκτονικής*, η *περιγραφή* της αρχιτεκτονικής, διάφορες μερικές περιγραφές και η οπτική τους γωνία, οι ανησυχίες, το περιβάλλον, τα εμπλεκόμενα άτομα/οργανισμοί (stakeholders) και τα διάφορα μοντέλα περιγραφής της αρχιτεκτονικής. Για την καλύτερη κατανόηση αυτών των όρων ακολουθεί μια περιγραφή τους μαζί με κάποια παραδείγματα.

Το *πλαίσιο της αρχιτεκτονικής* αποτελείται από ένα σύνολο από συμβάσεις αρχές και πρακτικές που θα χρησιμοποιηθούν για την δημιουργία της περιγραφής. Αυτές αναπτύσσονται στα πλαίσια του οργανισμού/ομάδας και έχουν ως σκοπό,· οι περιγραφές μιας αρχιτεκτονικής που θα φτιαχτούν από μία ομάδα να "*μοιάζουν*" με αυτές που θα φτιάξει μία άλλη.

Για παράδειγμα μία εταιρία μπορεί να αποφασίσει ότι θα χρησιμοποιούν το ίδιο σύστημα ονοματολογίας μεταβλητών (camel case, snake case, pascal case). Πρόκειται για μία σύμβαση που γίνεται με σκοπό οι προγραμματιστές να μπορούν να διαβάζουν το κώδικα που έχουν γράψει άλλοι εύκολα και άρα να είναι πιο παραγωγικοί. Πολύ συχνά αυτές οι

απαιτήσεις υπάρχουν γιατί ο οργανισμός ανάπτυξης λογισμικού που στήνει την αρχιτεκτονική θέλει πολλές ομάδες να δουλεύουν παράλληλα στο σύστημα και αν δεν ορίσει αυτούς τους κανόνες τότε θα καταλήξει η κάθε ομάδα να γράφει κώδικα που άλλες δεν θα μπορούν να διαβάσουν ή να χρησιμοποιήσουν. Ένα άλλο παράδειγμα θα ήταν η χρήση των ίδιων εργαλείων όταν αυτό είναι εφικτό (compilers, formatters, editors, κτλ.)

Η περιγραφή της αρχιτεκτονικής αποτελείται από σχέδια, διαγράμματα και ότι άλλο χρειάζεται για την κατασκευή του συστήματος. Πρόκειται για μια πλήρη αναπαράσταση που περιέχει ότι είναι απαραίτητο για να υλοποιηθεί το σύστημα και να δικαιολογηθούν όλες οι σχεδιαστικές επιλογές. Αναγνωρίζονται όλοι οι ενδιαφερόμενοι καθώς και οι ανησυχίες τους. Επίσης καταγράφεται με ποιο σκεπτικό έγιναν οι διάφορες επιλογές, ποιες άλλες αξιολογήθηκαν (και ας μην επιλέχθηκαν), ποια είναι η θεμιτή χρήση του συστήματος, η ελαστικότητα του, τα όρια του, καθώς και διάφορες αναλύσεις του από τις διαφορετικές οπτικές γωνίες των ενδιαφερόμενων.

Παράδειγμα μιας περιγραφής αρχιτεκτονικής θα ήταν ένα σύνολο από διαγράμματα σε ένα λογισμικό σχεδιασμού όπως το Visual Paradigm που χρησιμοποιήθηκε στην παρούσα εργασία. Mockups, Wireframes, Logos και άλλες περιγραφές οπτικοακουστικού υλικού θα μπορούσαν να σχεδιαστούν σε οποιασδήποτε περιβάλλον σχεδίασης όπως το Figma, Adobe Illustrator, Photoshop, κτλ. Και αυτά αποτελούν κομμάτι της περιγραφής και μάλιστα μπορεί να είναι κομμάτι των απαιτήσεων που δόθηκαν προς υλοποίηση και δεν υπήρχε ουσιαστικός έλεγχος στην μορφή τους.

Τέλος είναι πολύ σημαντικό να γίνει αντιληπτό ποια είναι τα άτομα/ομάδες που ως εμπλεκόμενα μέρη έχουν κάποιο ενδιαφέρον πάνω στο σύστημα (stakeholders). Αυτά τα άτομα θα υποδείξουν λειτουργικές απαιτήσεις καθώς και ανησυχίες τους (concerns) που θα πρέπει να συνυπολογιστούν κατά τον σχεδιασμό. Πολλές φορές οι ανησυχίες τους δεν είναι εύκολο να αναλυθούν σε λειτουργικές ή σχεδιάστηκες απαιτήσεις γιατί μπορεί να είναι αυθαίρετες στην φύση τους. Αυτό που πρέπει να γίνει είναι να φτιαχτούν διάφορες μερικές περιγραφές σχεδιασμένες για να τους δείξουν πως το σύστημα είμαι συμβατό με τις ανάγκες τους.

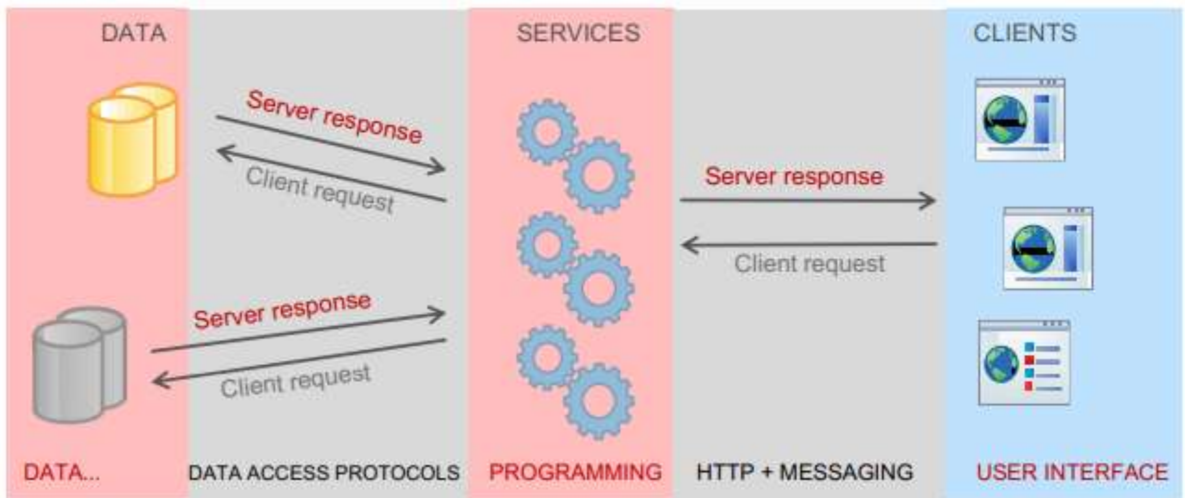
Γεια παράδειγμα ένας οικονομικός αναλυτής δεν ενδιαφέρεται για όλες τις λεπτομέρειες ενός συστήματος. Ενδιαφέρεται για το υλικοτεχνικό κόστος του, δηλαδή τα πάγια κόστη σε υπολογιστική και αποθηκευτική ισχύ καθώς και το κόστος ανάπτυξης του συστήματος για να μπορεί να υπολογίσει κατά πόσο αυτό το σύστημα είναι εντός των χρημάτων που ο πελάτης είναι πρόθυμος να δώσει. Η ανάλυση του μπορεί να μας οδηγήσει σε επανασχεδιασμό του συστήματος.

Είναι σημαντικό να αναφερθεί ότι ο σχεδιασμός μιας αρχιτεκτονικής δεν είναι αναγκαστικά ευθύγραμμη διαδικασία. Σε οποιοδήποτε στάδιο ζωής του συστήματος μπορεί να υπάρξουν αλλαγές και βελτιώσεις ακόμα και μετά την παράδοση και εγκατάσταση του. Για αυτό πρέπει να γίνει όσο το δυνατόν πιο πλήρη περιγραφή, που να περιέχει τι έχει φτιαχτεί ώστε όταν αργότερα πρέπει να γίνουν αλλαγές (όχι υποχρεωτικά από τον αρχικό σχεδιαστή) να μπορεί κάποιος από τα σχέδια να καταλάβει τι επιλογές έχουν γίνει και γιατί, τι συμβιβασμοί έχουν γίνει, τα γνωστά λάθη που υπάρχουν και ποιους επηρεάζουν.

1.2 Χαρακτηριστικές αρχιτεκτονικές εφαρμογών web

Με τα χρόνια οι διαδικτυακές εφαρμογές εξελίχθηκαν από απλές στατικές ιστοσελίδες κειμένου με ελάχιστη οπτική επεξεργασία σε πολύπλοκα συστήματα με GUI που ενημερώνονται δυναμικά. Για να γίνει αυτό έπρεπε να αλλάξει ο τρόπος με τον οποίο δημιουργούσε κάποιος ιστοσελίδες και να μεταβεί σε πιο ολοκληρωμένες λύσεις όπως μία πλήρης αρχιτεκτονική νοοτροπία. Όλες οι αρχιτεκτονικές σήμερα στην πλειοψηφία τους

ακολουθούν την three-tier Architecture η οποία διαχωρίζει την εφαρμογή σε τρία λογικά και φυσικά τμήματα το καθένα με διαφορετικές αρμοδιότητες. [2]



Εικόνα 2: 3-tier Architecture

Το Presentation tier είναι το τμήμα που είναι υπεύθυνο για την παρουσίαση της εφαρμογής στον χρήστη. Χρησιμοποιεί τις κατάλληλες τεχνολογίες για να τρέξει στον client για το οποίο σχεδιάστηκε (web, mobile, desktop).

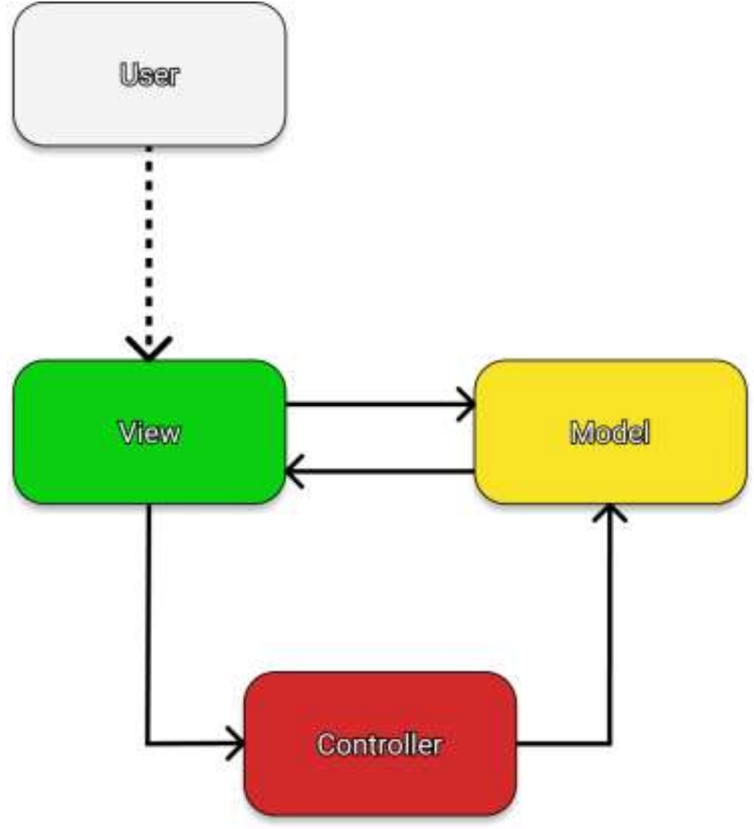
Το Service tier το οποίο λέγεται και αλλιώς logic ή application tier είναι ο κορμός της αρχιτεκτονικής. Σε αυτό μαζεύονται οι πληροφορίες από το Presentation tier και επεξεργάζονται σύμφωνα με το business logic της εφαρμογής. Πολύ συχνά χρησιμοποιεί πληροφορίες από το Data tier για να κάνει αυτήν την επεξεργασία.

Το Data tier, το οποίο πολύ συχνά λέγεται και βάση δεδομένων, είναι το μέρος όπου οι πληροφορίες που μάζεψε το Service tier αποθηκεύονται. Για την αποθήκευση μπορούν να χρησιμοποιηθούν Relational database management systems (RDBMS) όπως οι PostgreSQL, MySQL, MariaDB, μπορούν να χρησιμοποιηθούν NoSQL Database servers όπως Cassandra, CouchDB, MongoDB ή όποιος άλλος αποθηκευτικός τρόπος κρίνεται κατάλληλος για τα δεδομένα που χειρίζεται το σύστημα.

Αυτός ο τρόπος δόμησης κώδικα έχει πολλά πλεονεκτήματα και στην απόδοση του και στο ότι επιτρέπει την παράλληλη δημιουργία του συστήματος. Στις διαδικτυακές εφαρμογές η πιο διαδεδομένη τεχνοτροπία που είναι συμβατή με αυτή την αρχιτεκτονική λέγεται MVC

1.2.1 MVC και παράγωγα του

Το MVC είναι ο πιο διαδεδομένος τρόπος δόμησης ενός λογισμικού. Ο κώδικας διαχωρίζεται σε τρία μέρη, το μοντέλο (Model), την Όψη (View), και τον ελεγκτή (Model). Αυτός ο διαχωρισμός επιτρέπει την παράλληλη εργασία σε 2 ή και 3 κομμάτια του λογισμικού χωρίς να χρειάζεται ιδιαίτερος συντονισμός μεταξύ τον ατόμων που γράφουν τα διάφορα μέρη. Επίσης επειδή το κάθε κομμάτι του κώδικα είναι αυτοτελές και δεν επηρεάζει τα άλλα, μπορεί να ξαναχρησιμοποιηθεί ευκολότερα. Ο διαχωρισμός των ευθυνών για κάθε αυτοτελές τμήμα του κώδικα είναι πολύ σημαντικός.



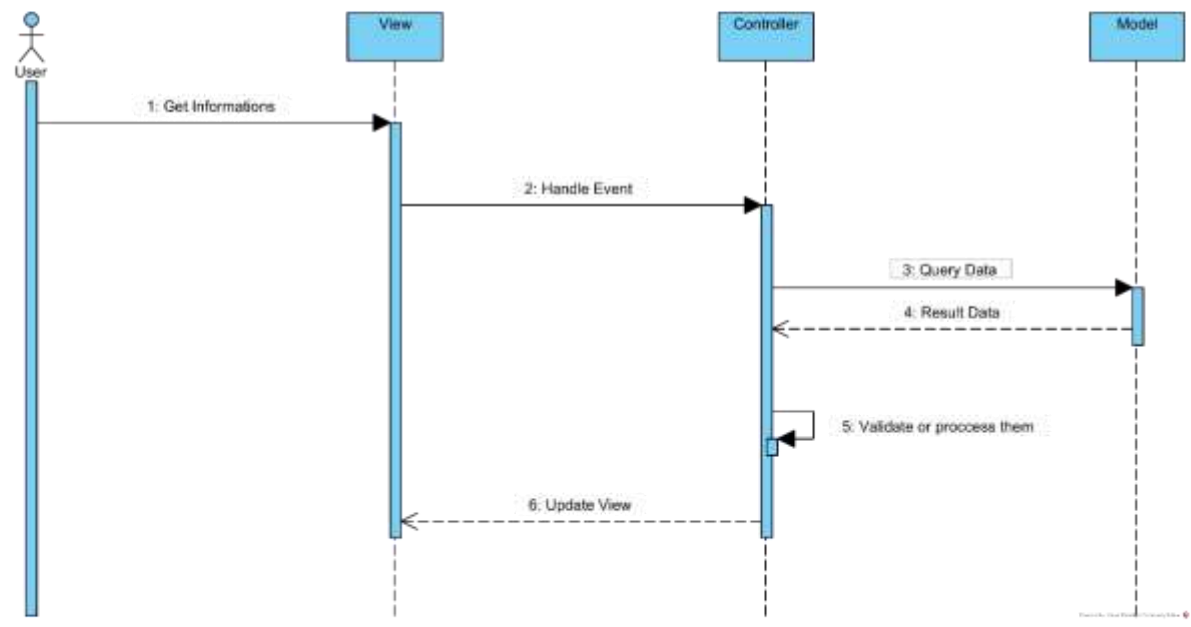
Εικόνα 3: MVC Block Diagram

Η επικοινωνία των επιμέρους τμημάτων περιγράφεται πλήρως από το παραπάνω διάγραμμα. Η γενικότερη ιδέα είναι ότι ο χρήστης αλληλοεπιδρά με το View το οποίο ενημερώνει και ενημερώνεται από το Model μέσω μηνυμάτων τα οποία χειρίζεται ο Controller.

Το Model είναι το μέρος εκείνο του κώδικα που είναι υπεύθυνο για την διαχείριση των δεδομένων του συστήματος. Όλες οι αλλαγές στα δεδομένα πρέπει να περνάνε από αυτό. Όταν κάποιος θέλει πρόσβαση σε αυτά επίσης πρέπει να τα ζητήσει από αυτό. Για παράδειγμα σε ένα ολοκληρωμένο σύστημα υπάρχει μία βάση δεδομένων (RDBMS) ή ένα άλλο ανάλογο μέσο αποθήκευσης πληροφορίας. Όταν ένας χρήστης θέλει πρόσβαση στα δεδομένα που είναι αποθηκευμένα εκεί, δεν πηγαίνει να τα πάρει κατευθείαν, αλλά τα ζητάει από το μοντέλο που ξέρει πια είναι η εσωτερική δόμησή τους και πως να τα ζητήσει. Το μοντέλο έχει επίσης την δυνατότητα να αρνηθεί να δώσει δεδομένα στις περιπτώσεις όπου η πρόσβαση σε αυτά απαγορεύεται. Μπορεί να κάνει κάποια τροποποίηση στην μορφή τους ή να υλοποιεί κάποια διαδικασία μοναδική για το σύστημα. Όλα αυτά πρέπει να υλοποιηθούν με κώδικα, ο κώδικας αυτός είναι το Model.

Το View χρησιμοποιεί τα δεδομένα που της έδωσε το Model και διαμορφώνει το τι βλέπει ένας χρήστης. Επίσης είναι υπεύθυνο να μετατρέπει τις ενέργειες των χρηστών (πάτημα κουμπιών, συμπλήρωση κειμένου, κτλ.) σε μηνύματα τα οποία δίνει στον Controller για να τροποποιήσει το Model μέσο αυτού αν το επιθυμεί. Στις εφαρμογές διαδικτύου που τρέχουν σε browsers όπως πχ ο Chrome, το View είναι ένα html έγγραφο που βλέπει ο χρήστης. Είναι σημαντικό να ξέρουμε ότι δεν είναι υποχρεωτικό η όψη να είναι πάντα html. Σε κινητά το view μπορεί να είναι ένα xml αρχείο όπως στο android, ένα xaml αρχείο όπως στα windows ή να ορίζεται το GUI δηλωτικό τρόπο όπως στην Swift για το IOS. Επίσης υπάρχουν περιπτώσεις όπου τα δεδομένα δεν χρησιμοποιούνται για να δει κανείς ένα αποτέλεσμα αλλά περνάνε απλά σε μία κατάλληλη μορφή κειμένου ώστε να γίνει αργότερα η

επεξεργασία τους. Τέλος υπάρχουν περιπτώσεις όπου και ο κώδικας μπορεί να είναι κομμάτι του View. Ένας εμπειρικός κανόνας για το τι καθιστά κομμάτι του View είναι: *Ότι αποτελεί απάντηση από ένα server και τρέχει σε σελιδοδείκτη/κινητό/υπολογιστή*. Στα δυναμικά συστήματα και ο κώδικας που είναι κομμάτι του View μπορεί επίσης να δομηθεί με MVC.



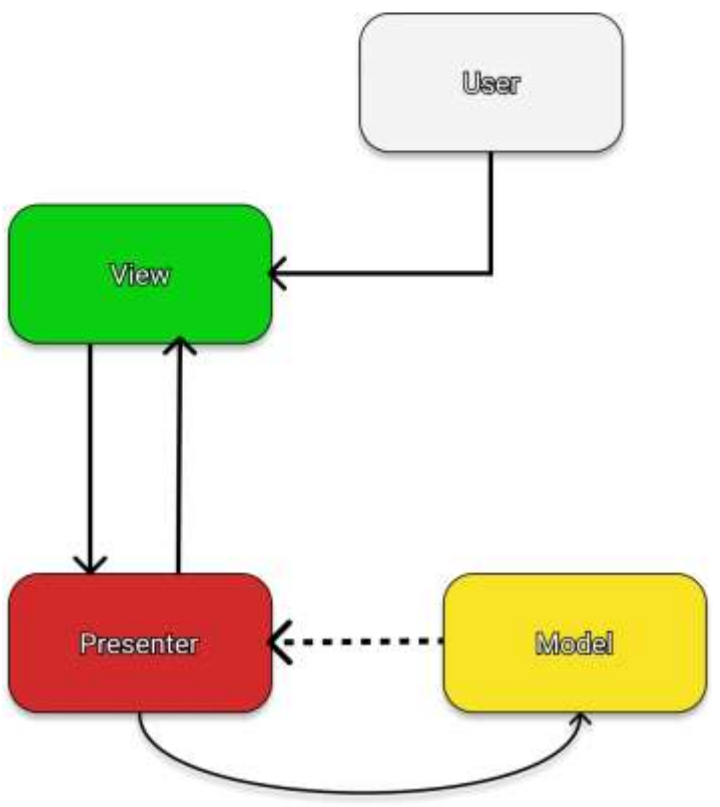
Εικόνα 4: Generic MVC

Ο Controller είναι το τρίτο κομμάτι του MVC και είναι υπεύθυνο για την ενορχήστρωση του συστήματος. Είναι υπεύθυνος για τον χειρισμό των μηνυμάτων που του περνάει το View, τα οποία θα περάσει στο Model. Άλλες αρμοδιότητες του, ανάλογα με τις τεχνολογίες που χρησιμοποιούνται μπορούν να θεωρηθούν η δρομολόγηση (routing), η καταγραφή των αλληλεπιδράσεων (logging) καθώς και η διασύνδεση με άλλες εφαρμογές.

Σε αυτό το σημείο πρέπει να αναφερθεί ότι ο διαχωρισμός που έγινε παραπάνω δεν είναι ο μόνος. Στην πράξη κάθε ομάδα ορίζει το τι είναι κομμάτι του controller, του view, και του model. Όσο πιο περίπλοκο είναι το σύστημα τόσο πιο χαλαρές είναι οι γραμμές μεταξύ model και controller. Σήμερα με τη χρήση τεχνολογιών όπως η React και η Angular, η όψη έχει περισσότερες αρμοδιότητες και δεν είναι πλέον απλά ένα έγγραφο που πρέπει να παρουσιαστεί. Τέλος υπάρχουν υβριδικές προσεγγίσεις όπως το server side rendering (SSR) όπου υπάρχει κώδικας που ανάλογα την περίπτωση θα τρέξει στον controller ή στο view. Όλα αυτά θεωρούνται MVC αλλά όπως γίνεται εύκολα αντιληπτό δεν τηρούν τον διαχωρισμό των αρμοδιοτήτων.

Το MVC ήταν από της πρώτες αρχιτεκτονικές οργάνωσης κώδικα που καθιερώθηκαν για την κατασκευή ιστοσελίδων και υιοθετήθηκε από σχεδόν όλες τις ομάδες. Με τα χρόνια και την εισαγωγή των κινητών στην αγορά δημιουργήθηκαν και άλλες τεχνικές, παράγωγα του MVC, που χρησιμοποιούνται για την κατασκευή εφαρμογών.

Το MVP είναι από αυτά τα παράγωγα. Η διαφορά είναι ότι αντί για Controller υπάρχει ο Presenter. Η αρμοδιότητα του είναι να δουλεύει σαν ενδιάμεσος μεταξύ του View και του Model που παραμένουν τα ίδια. Δέχεται δεδομένα από το Model και τα επιστρέφει επεξεργασμένα στο view. Η διαφορά του από το MVC είναι ο Presenter είναι αυτός που αντιδράει στα μηνύματα από το View, έχει δηλαδή κάποιες από τις αρμοδιότητες του Model στο MVC. Χρησιμοποιείται κυρίως για την δημιουργία GUI σε κινητά αλλά και στο web.



Εικόνα 5: MVP

Υπάρχουν πολλές ακόμα μεθοδολογίες που προήλθαν από το MVC και χρησιμοποιούνται μόνο για μία πλατφόρμα. Ουσιαστικά πρόκειται για αυτό που αναφέρθηκε νωρίτερα, ότι δηλαδή η κάθε πλατφόρμα και κάθε το framework πείρε την αρχική ιδέα του MVC και την τροποποίησε για να ταιριάζει στη δικιά τους μεθοδολογία. Από όλες αυτές τις παρεκκλίνουσες τεχνολογίες μόνο το MVP και το ποιο καινούργιο MVVM (το οποίο δεν θα αναλυθεί καθόλου) βρήκαν ευρεία αποδοχή, και αυτή κυρίως στις πλατφόρμες των κινητών. Παρά τις διαφορές τους από το MVC αποτελούν και οι δυο 3-tier architectures.

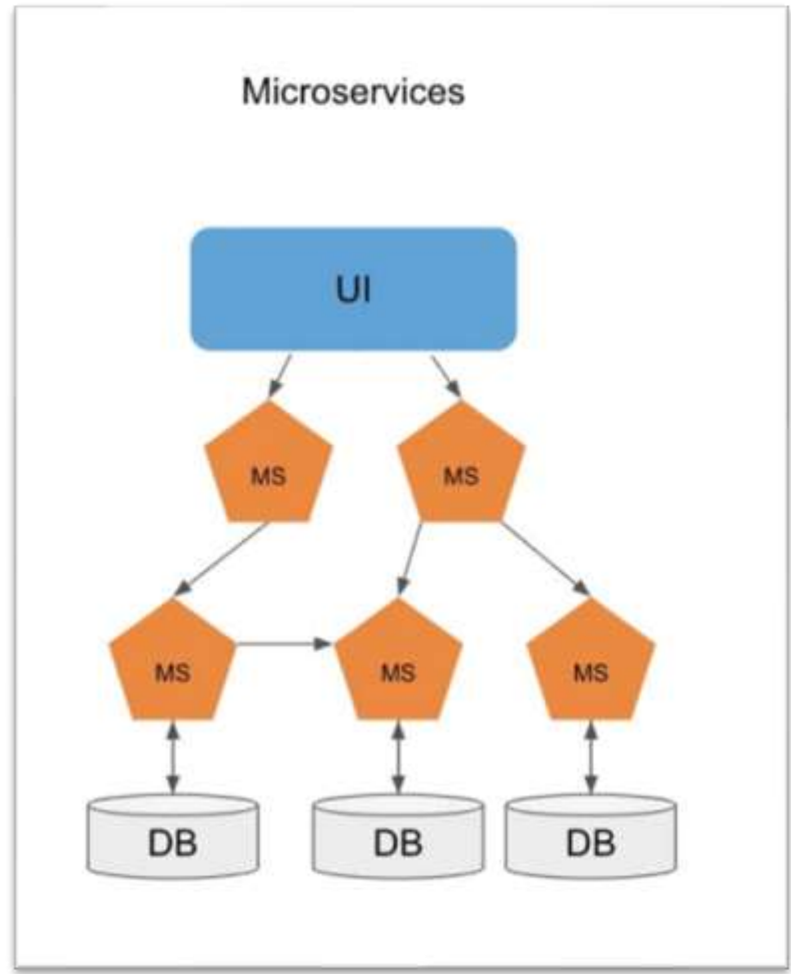
1.2.2 Microservices

Το MVC είναι μια μονολιθική αρχιτεκτονική στην φύση του. Αυτό σημαίνει ότι όλος ο κώδικας είναι σχεδιασμένος να τρέχει μαζί, είναι γραμμένος στην ίδια γλώσσα και αν γίνουν αλλαγές σε ένα κομμάτι του πρέπει όλη η εφαρμογή να εγκατασταθεί πάλι. Αυτό σε μικρά συστήματα είναι ευκολότερο γιατί οι λειτουργικές απαιτήσεις δεν υπερβαίνουν τα ανώτερα όρια της υπολογιστικής ισχύς που μπορεί να έχει ένα σύστημα. Σήμερα είναι αρκετά συνηθισμένο τα συστήματα που στήνονται να πρέπει να χειρίζονται εκατομμύρια χρηστών και δισεκατομμύρια συναλλαγών μεταξύ αυτών και του συστήματος.

Ένας από τους τρόπους να υπερβούμε αυτά τα όρια και τις δυσκολίες που εμφανίζονται κατά την λειτουργία και συντήρηση μονολιθικών συστημάτων είναι η διάσπαση αυτής της αρχιτεκτονικής. Έτσι λοιπόν ορίζονται τα microservices ως ένα σύνολο από πολλές μικρές αυτόνομες υπηρεσίες οι οποίες αλληλοεπιδρούν μεταξύ τους ή με τον χρήστη μέσω πρωτοκόλλων επικοινωνίας που είναι αγνωστικιστικά ως προς τις τεχνολογίες που χρησιμοποιούνται.

Το πρώτο που πρέπει να γίνει ώστε το σύστημα που στήνετε να χρησιμοποιεί την αρχιτεκτονική τον microservices είναι να σπάσει το ολικό πλάνο σε ένα σύνολο από χαρακτηριστικά (features) τα οποία θα μπορούν να αποτελέσουν μία αυτοτελή υπηρεσία το καθένα. Δεν είναι υποχρεωτικό να είναι τόσα τα microservice όσα και τα features τους

συστήματος αλλά μπορούν να χρησιμοποιηθούν ως μία πρώτη διάσπαση και αργότερα να ξανά-ομαδοποιηθούν με βάση την ομοιότητα τους και τον αριθμό των ξεχωριστών υπηρεσιών που κρίνεται απαραίτητο να υπάρχουν. Σε δεύτερη φάση και εφόσον αυτό θεωρηθεί απαραίτητο μπορεί ένα microservice να σπάσει σε 2 ή παραπάνω ανάλογα με τις ανάγκες που θα προκύψουν. [3]



Εικόνα 6:General micro services architecture

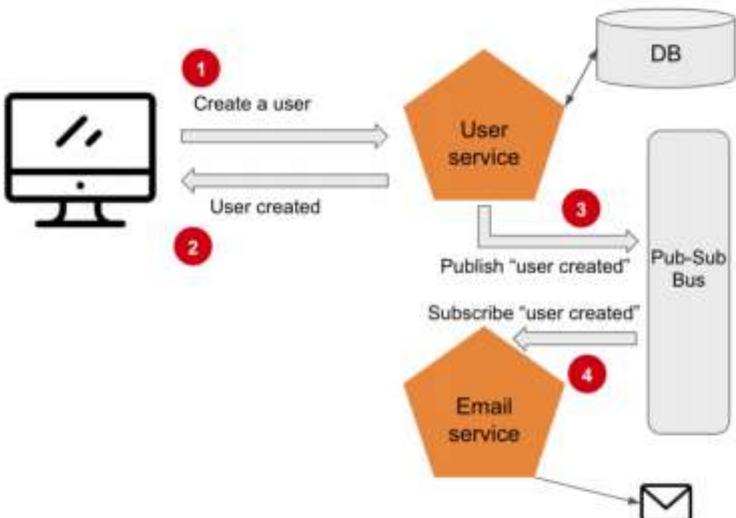
Αν το σύστημα μας έχει υποσυστήματα τα οποία είναι πολύ βαριά σε απαιτήσεις υλικού και δικτύου τότε μπορεί κάποιος με την χρήση microservices να χρησιμοποιήσει 2 ή και παραπάνω instances του service αυτού ώστε να ισορροπήσει της απαιτήσεις μεταξύ τους χωρίς να πρέπει να κλωνοποιήσει ολόκληρο το σύστημα. Έτσι λύνεται το πρόβλημα της διαχείρισης απαιτήσεων που υπερβαίνουν τα φυσικά όρια ενός μηχανήματος με το ελάχιστο δυνατό κόστος.

Υπάρχουν πολλά πλεονεκτήματα στο να σχεδιαστεί ένα σύστημα με αυτή την αρχιτεκτονική. Αρχικά το κάθε service μπορεί να φτιαχτεί με οποιοδήποτε σετ τεχνολογιών είναι το κατάλληλο για αυτό. Για παράδειγμα ένα service που είναι υπεύθυνο για την διαχείριση χρηστών είναι τελείως διαφορετικό από ένα άλλο που είναι υπεύθυνο για την δημιουργία στατιστικών αναλύσεων. Έτσι η ομάδα που θα στήσει το service διαχείρισης χρηστών μπορεί να χρησιμοποιήσει NodeJS (express) και αυτός που κάνει στατιστική ανάλυση μπορεί να χρησιμοποιήσει python. Έτσι οι 2 ομάδες μπορούν να δουλεύουν με τα κατάλληλα εργαλεία για την δουλειά που κάνουν. Αυτό δεν θα ήταν δυνατό σε μία

μονολιθική κατασκευή καθώς θα έπρεπε όλοι να χρησιμοποιούν την ίδια γλώσσα προγραμματισμού.

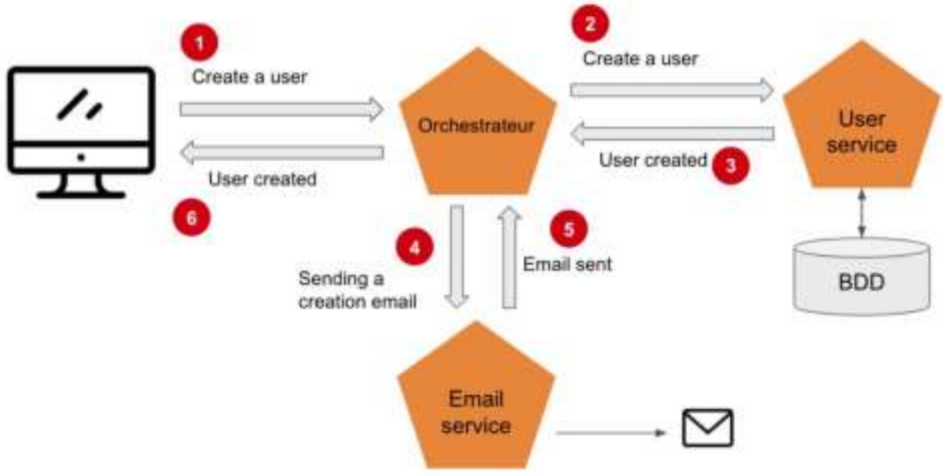
Μία ανάγκη που προκύπτει από τον μεγάλο αριθμό των διαφορετικών διεργασιών είναι αυτή για συγχρονισμό και επικοινωνία. Αυτό μπορεί να επιτευχθεί με την αποστολή μηνυμάτων μεταξύ διαφόρων services. Η ερώτηση είναι πως αυτά τα μηνύματα θα φτάνουν από την μία διαδικασία στην άλλη. Μπορεί μια εφαρμογή να μιλάει κατευθείαν με άλλη αλλά τι γίνεται αν ένα μήνυμα χαθεί ή η διεργασία δεν είναι εκεί για να το παραλάβει. Τι θα γίνει αν ένα μήνυμα πρέπει δεν φτάνει, πρέπει να ξανασταλεί, πρέπει να υποθέσει το microservice που το έστειλε όταν παραλήφθηκε, μπορεί κάπως να το ελέγξει; Όλες αυτές οι ερωτήσεις ή ανάλογες του τίθενται κάθε φορά που στήνεται ένα distributed σύστημα. Στον τομέα των microservices προτάθηκαν δύο μεθοδολογίες για να εξασφαλιστεί η ορθότητα του συστήματος, choreography και orchestration. Και οι δύο προσεγγίζουν το πρόβλημα διαφορετικά και έχουν πλεονεκτήματα και μειονεκτήματα. [4]

Όταν υλοποιούνται τα microservices με choreography κάθε μία από αυτές δουλεύει ως εξής. Πρώτα εκτελεί μία εσωτερική της διεργασία (που είναι και ο σκοπός της) και μετά δημοσιεύει ένα μήνυμα, το οποίο άλλα microservices μπορούν παρακολουθούν. Κάθε microservice μπορεί να ακούει μηνύματα από όσα άλλα microservices θέλει και να ενημερώνεται για αλλαγές που το ενδιαφέρουν. Το πρόβλημα αυτής της υλοποίησης είναι ότι πρέπει να γίνει πάρα πολύ καλός σχεδιασμός γιατί κάθε microservice πρέπει κάνει το ίδιο subscribe στα μηνύματα των άλλων, κάτι που θέλει προσοχή για λάθη στην υλοποίηση.



Εικόνα 7: Microservices με choreography

Ο εναλλακτικός τρόπος υλοποίησης είναι το Orchestration. Αντί να πρέπει κάθε microservice να πάει να κάνει subscribe στα μηνύματα όλων των άλλων, υπάρχει ο orchestrator που κάνει αυτός subscribe σε όλα και ξεκινάει αυτός το επόμενο microservice που πρέπει να ξεκινήσει. Με αυτήν την τεχνική απλοποιείται πολύ η υλοποίηση της ενδοεπικοινωνίας μεταξύ των microservices και με τον έξω κόσμο καθώς αντί να υπάρχει ένας αυθαίρετος τρόπος που πρέπει να συνδεθεί το κάθε microservice με όλα τα άλλα, τώρα απλά συνδέεται με τον orchestrator και αυτό υλοποιεί την σύνδεση του με τα υπόλοιπα.

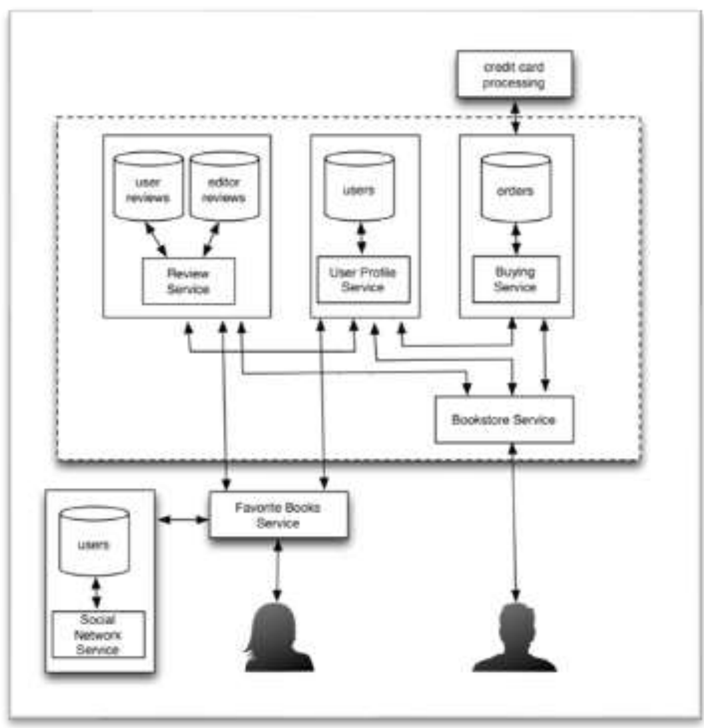


Εικόνα 8: Microservices με orchestration

Όσον αφορά την απόδοση και την ταχύτητα του συστήματος έχουν γίνει μελέτες για το ποια είναι ανώτερη από τις δύο. Σύμφωνα με έρευνα που έγινε πάνω στην συγκριτική απόδοση των δύο μεθόδων συμπεράθηκε ότι η τεχνική του choreography είναι πολύ ποιο γρήγορη από το orchestration κάτι το οποίο θα μπορούσε να αποφανθεί στο ότι σε μεγάλο αριθμό microservices ο orchestrator καταπονείται από τον μεγάλο όγκο των μηνυμάτων που πρέπει να επεξεργαστεί. [5]

Στο ίδιο άρθρο γίνεται ιδιαίτερη αναφορά στην δυσκολία υλοποίηση του choreography όσο μεγαλώνει ο αριθμός των microservices και των μηνυμάτων που μεταβιβάζονται. Μία ομάδα από developers που δουλεύει πάνω σε ένα microservices μπορεί να μην έχει καμία ιδέα για το πως μια άλλη, πάνω στην οποία πρέπει να συνδεθεί, έχει υλοποιηθεί γιατί την έφτιαξε κάποια άλλη ομάδα και άρα αυτές οι ομάδες πρέπει να συνεργαστούν για να επιτευχθεί αυτή η επικοινωνία. Το πρόβλημα περιπλέκει παραπάνω όταν μία τρίτη ομάδα έρθει σε δεύτερο χρόνο να κάνει αλλαγές ή διορθώσεις και δεν ξέρει πως υλοποιήθηκε αυτή η επικοινωνία και πόσα microservices θα πρέπει να αλλάξει για να κάνει την δουλειά της.

1.2.3 Microservice – Παράδειγμα προς μελέτη



Εικόνα 9: Bookstore as a Microservice

Παίρνοντας το σχήμα σαν παράδειγμα παρατηρούμε τα εξής. Κάθε service μπορεί να επικοινωνεί με άλλα services ή με τον εξωτερικό κόσμο. Επίσης φαίνεται το πως δύο χρήστες που χρησιμοποιούν δύο διαφορετικά services αλληλοεπιδρούν με το σύστημα. Αν παρατηρήσουμε την εικόνα βλέπουμε ότι ο χρήστης αριστερά δεν χρησιμοποιεί το υποσύστημα "*Buying Service*" γιατί δεν το χρειάζεται.

Για τη σωστή δόμηση ενός ολοκληρωμένου συστήματος με microservices πρέπει να θέσουμε κάποιες λειτουργικές απαιτήσεις από το εκάστοτε service. Τα ελάχιστα που πρέπει να έχει ένα service είναι αντοχή, διαφάνεια, αυτοματισμός, ορθότητα και σταθερότητα.

Αν υπάρξει κάποιο πρόβλημα σε ένα service, αυτό πρέπει να μπορεί να απομονωθεί χωρίς να καταρρεύσει πλήρως το σύστημα. Έτσι τα λάθη απομονώνονται και οι αλλαγές μπορούν να γίνονται χωρίς να χρειάζονται επανεκκινήσεις ολόκληρου του συστήματος. Φυσικά παρόλο που τα προβλήματα περιορίζονται αυτό προσθέτει πολύ περισσότερα σημεία που ο κώδικας μπορεί να πέσει (points of failure) γιατί όλα τα services πρέπει να είναι ανθεκτικά ως προς την πτώση όλων των άλλων services που επικοινωνούν.

Πρέπει επίσης να έχει στηθεί ένας μηχανισμός ώστε στην πιθανή πτώση ενός service αυτό να γίνεται γνωστό. Έτσι η κατάσταση των services πρέπει να είναι ορατή από έξω, ώστε να μπορεί κάποιος σε περίπτωση προβλήματος να παρέμβει και να το φτιάξει. Αν και αυτό είναι ιδιαίτερα δύσκολο να γίνει, πλέον όλοι οι πάροχοι cloud έχουν ένα μηχανισμό για να το υλοποιούν. Επίσης υπάρχουν έτοιμες τεχνολογίες που το κάνουν όπως το Openstack.

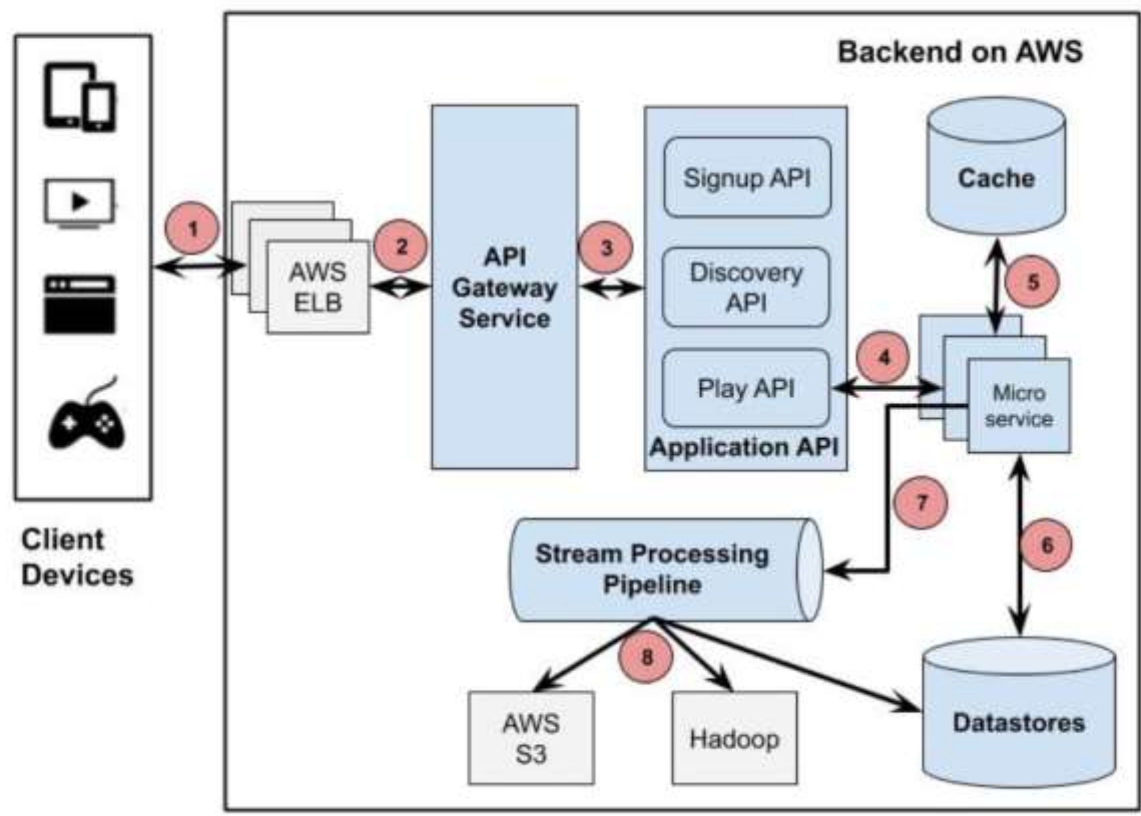
Ο αυτοματισμός των διαφόρων διαδικασιών την κατασκευής και συντήρησης του συστήματος είναι κάτι που δεν είναι απαραίτητο αλλά τις περισσότερες φορές υπάρχει για προφανοί λόγους. Μεγάλο μέρος της συντήρησης και της ενσωμάτωσης των αλλαγών είναι αυτοματοποιήσιμο και μπορεί να γλιτώσει στον κατασκευαστή του πολύ χρόνο. Έστω για παράδειγμα το σύστημα αποτελούμενο από 10 microservices και 1 από αυτά πέφτει. Η διαδικασία του να ξανασηκωθεί είναι η ίδια όσες φορές και να συμβεί αυτό, οπότε είναι αυτοματοποιήσιμη. Όλοι οι cloud providers δίνουν εργαλεία για να υλοποιήσουμε ακριβώς

αυτό. Αν υπάρχει πρόβλημα με την καινούργια έκδοση ενός λογισμικού υπάρχουν μηχανισμοί που επιτρέπουν να γυρίσει το λογισμικό σε προηγούμενη έκδοση αυτόματα.

Αυτοματισμοί μπορούν να χρησιμοποιηθούν και κατά την δημιουργία του λογισμικού, ώστε όταν η ομάδα που γράφει τον κώδικα έχει έτοιμες τις αλλαγές που θέλει να υλοποιήσει να μπορούν με ένα μηχανισμό να προωθούν της αλλαγές στο production. Τέτοιοι μηχανισμοί έχουν πάρει το όνομα CI/CD από τις αγγλικές λέξεις Continuous Integration/ Continuous Development και υπάρχουν γιατί γίνεται να σηκωθεί ένα micro service αυτόματα που θα κάνει μία δουλειά και μετά θα ξανά κατέβει.

Τέλος πρέπει να δομηθούν τα microservices με ένα τρόπο που εξασφαλίζουν την ορθότητα και σταθερότητα του συστήματος. Ένας καλός τρόπος να γίνει αυτό, είναι να προσανατολιστεί η κατασκευή γύρω από business concepts, έτσι η δομή θα είναι πιο ορθή και συνεκτική με λιγότερα λάθη. Φυσικά αυτό δεν είναι υποχρεωτικό.

Από όλα τα παραπάνω είναι προφανές ότι τα microservices είναι αρκετά πιο περίπλοκη από τις κλασικές αρχιτεκτονικές όπως το MVC που αναλύθηκε νωρίτερα. Είναι λοιπόν θεμιτό να αναρωτηθεί κάποιος κατά πόσο αξίζει ο αυξημένος όγκος δουλειάς και η πολυπλοκότητα για να στηθεί ένα τόσο περίπλοκο σύστημα. Η απάντηση είναι ότι "εξαρτάται". Αν οι λειτουργικές απαιτήσεις μπορούν να καλυφθούν με μια ποιο εύκολη αρχιτεκτονική από άποψη σχεδιασμού τότε δεν χρειάζεται. Επίσης υπάρχουν περιπτώσεις που το σύστημα δεν μπορεί να σπάσει σε καλά ορισμένα υποσυστήματα ή οι απαιτήσεις συγχρονισμού είναι τέτοιες που όλες οι ενέργειες των χρηστών θα χτυπούσαν απαγορευτικά μεγάλο αριθμό microservices. Φυσικά υπάρχουν και περιπτώσεις όπως του Netflix το οποίο αναγκάστηκε να μεταβεί σε microservices καθώς ο όγκος των δεδομένων που έπρεπε να σερβίρει στους χρήστες του ήταν αδύνατον να επιτευχθεί με κάτι άλλο. [6]



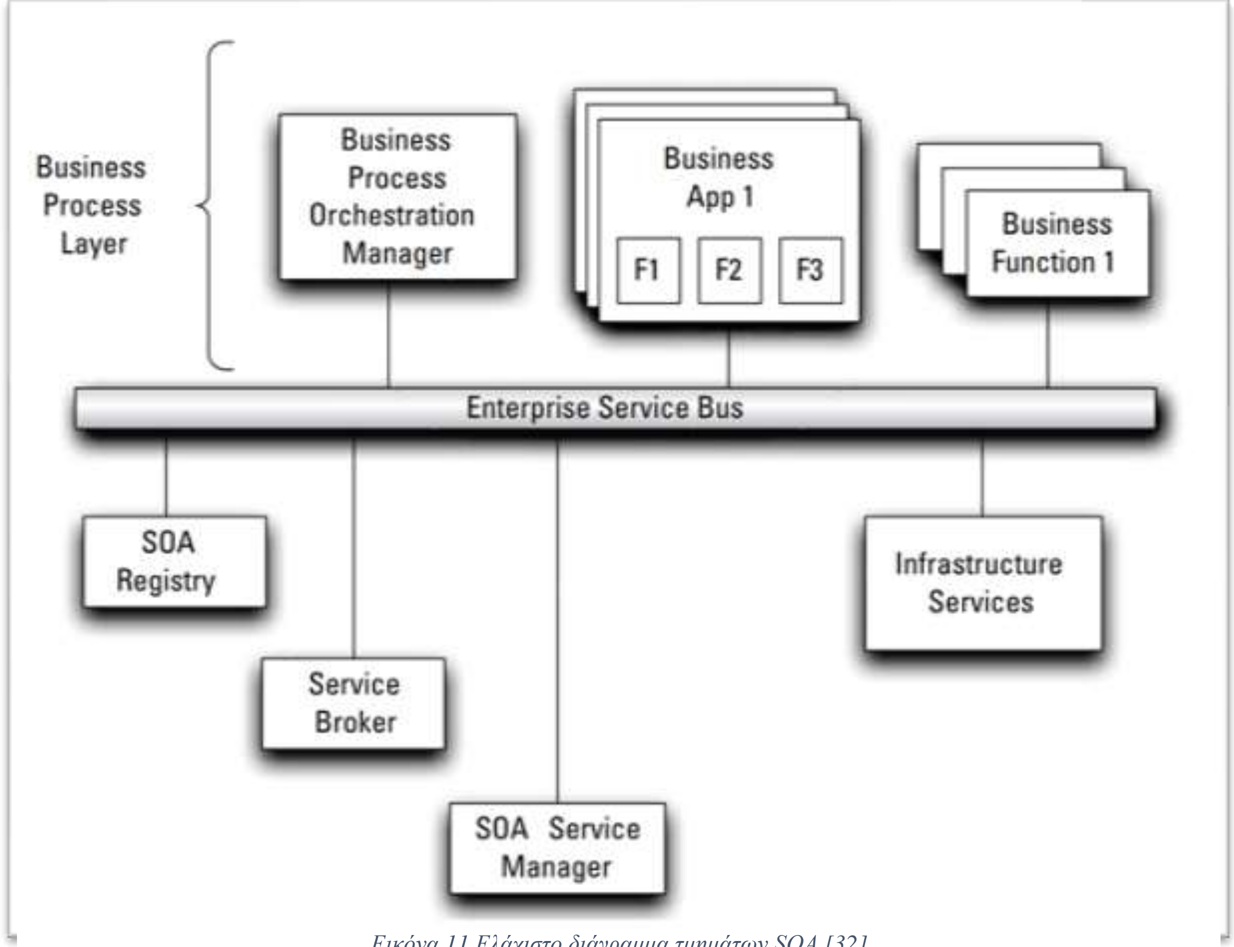
Εικόνα 10: Netflix AWS Backend

1.2.4 SOA

Το ακρώνυμο SOA βγάνει από τις λέξεις Service Oriented Architecture και τέτοιου τύπου συστήματα χρησιμοποιούνται κατά κόρον σε μεγάλες επιχειρήσεις/οργανισμούς. Ο λόγος είναι ότι όταν ένα σύστημα έχει σχεδιαστεί με αυτή την αρχιτεκτονική είναι εύκολο να εξασφαλιστεί η ορθότητα των διαδικασιών. Αυτό σημαίνει ότι αν υπάρχει κάποιος κανόνας που πρέπει να ακολουθηθεί (νομικό πλαίσιο, εταιρικές διαδικασίες), αυτό είναι εύκολο να οριστεί και να θεμελιωθεί ορθά.

Κατά την ανάλυση των microservices αναφέρθηκε ότι υπάρχουν περιπτώσεις όπου πρέπει να χρησιμοποιηθούν άλλες λύσεις γιατί κάποιες διαδικασίες απαιτούν μεγάλο όγκο δια συνδεσιμότητας των services. Σε αυτές τις περιπτώσεις ενδείκνυται να χρησιμοποιηθεί το SOA για να υλοποιηθούν αυτές οι διαδικασίες. Το κλασικό παράδειγμα είναι μία τράπεζα όπου είναι απαγορευτικό να γίνουν λάθη ή να χαθούν μηνύματα. Επίσης είναι προφανές ότι μία συναλλαγή χρησιμοποιεί μεγάλο αριθμό συστημάτων ώστε να εξασφαλισθεί η ορθότητα της.

Η αρχιτεκτονική των SOA έχει πιο αυστηρές απαιτήσεις για τη δόμηση της, με την έννοια ότι υπάρχουν κάποιες ελάχιστες δομές που πρέπει να υπάρχουν. Επίσης η τοπολογία του συστήματος είναι πιο αυστηρή γιατί όλες οι δομές επικοινωνούν μέσω του Enterprise Service Bus. Αυτή είναι και η ουσιαστική διαφορά με τα microservices. Αντί να υπάρχουν πολλά κανάλια επικοινωνίας, υπάρχει ένα στο οποίο πάνω μιλάνε όλοι και ακούει όποιος θέλει.



Εικόνα 11 Ελάχιστο διάγραμμα τμημάτων SOA [32]

Στο παραπάνω διάγραμμα φαίνονται τα ελάχιστα κομμάτια του SOA που απαιτούνται για την ορθή του λειτουργία. Φαίνεται ότι όλα τους είναι συνδεμένα πάνω στο ESB και ότι πέρα από τις εφαρμογές και τις διαδικασίες που συνδέουμε πάνω του, υπάρχουν κάποια στοιχεία που αποτελούν υποδομές του συστήματος.

Το κομμάτι του συστήματος που στο διάγραμμα ονομάζεται Business Process Layer αποτελείται από τις διάφορες εφαρμογές που είναι συνδεδεμένες στο σύστημα και τις διεργασίες (Business Functions) που υλοποιούν τις λειτουργικές απαιτήσεις. Πρόκειται για την ουσία της επαγγελματικής δραστηριότητας και περιέχει τα εργαλεία που χρησιμοποιούν οι χρήστες του συστήματος. Η ειδοποιός διαφορά μεταξύ αυτών τον διεργασιών και των υπόλοιπων είναι ότι περιέχουν λειτουργικότητα μοναδική για τον οργανισμό που στήνει το σύστημα ενώ οι υπόλοιπες είναι απαραίτητες για ορθή λειτουργεία μια SOA αρχιτεκτονικής και ένα implementation τους υπάρχει σε κάθε τέτοιο σύστημα.

Η αλληλεπίδραση μεταξύ αυτών των λειτουργιών γίνεται μέσω του Business Process Orchestration Manager ο ποιος συνδέει ανθρώπους με ανθρώπους, ανθρώπους με διαδικασίες, διαδικασίες με διαδικασίες.

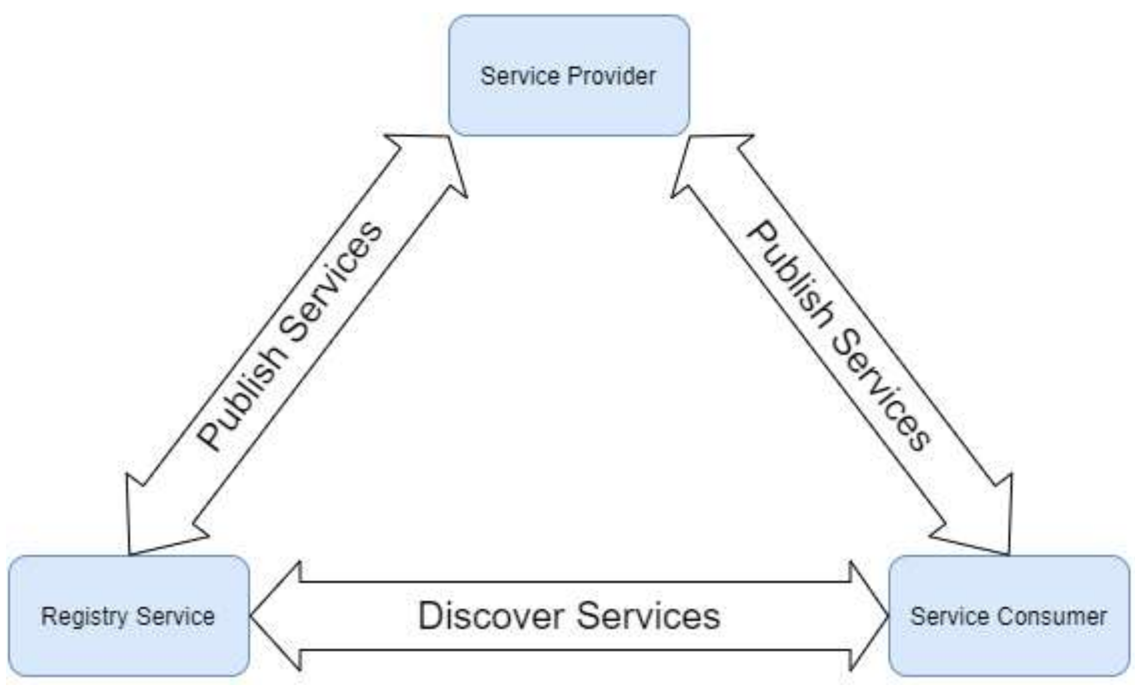
Όταν μια υπηρεσία θέλει να συνδεθεί με μία άλλη τότε χρησιμοποιεί τον Service Broker. Η διαφορά μίας διαδικασίας (process) από μία υπηρεσία (service) είναι ότι η υπηρεσία μπορεί να είναι ένα πλήρες υποσύστημα ενώ μία διαδικασία είναι συνήθως κάτι μεμονωμένο. Ένα σύνολο από διαδικασίες αποτελεί μία υπηρεσία.

Υπάρχει και ένα σύνολο από διεργασίες οι οποίες πρέπει να υπάρχουν παρότι δεν συνδέονται άμεσα με την επιχείρηση γιατί χρειάζονται για την εξασφάλιση της ορθής λειτουργίας του συστήματος. Στο κάτω μέρος του διαγράμματος φαίνεται το SOA Registry, Service Broker, SOA Service Manager και Infrastructure Services.

Ο ρόλος του SOA Service Manager είναι να βεβαιωθεί ότι οι τεχνολογίες λειτουργούν με συνεπή και προβλέψιμο τρόπο. Ο στόχος είναι να δημιουργηθεί ένα περιβάλλον όπου όλες οι διεργασίες συνεργάζονται για να βελτιώσουν την ροή της επαγγελματικής διαδικασίας. Ουσιαστικά πρόκειται για μία διαχειριστική μονάδα που προσπαθεί να μειώσει τις περιπτώσεις καθυστερήσεων λόγο κακού συντονισμού των διαδικασιών.

Σε αυτό το σημείο καλό είναι να αναφερθεί που και πως όλοι αυτοί οι μηχανισμοί είναι αποθηκευμένοι και πως κάποια καινούργια υπηρεσία αποκτά πρόσβαση στο σύστημα. Αυτό γίνεται μέσα από τα SOA Registry και Repository.

Το Registry έχει και δίνει σε όποιον ζητήσει τις πληροφορίες για κάθε service που είναι εγγεγραμμένο πάνω στο service bus, της λειτουργίες του και το πως συνδέονται με άλλες υπηρεσίες. Γενικά εκεί ζουν όλοι οι κανόνες και οι περιγραφές που σχετίζονται με το κάθε στοιχείο, για όλα τα στοιχεία που είναι ενεργά στο σύστημα. Ο Service Broker σαν μηχανισμός επωφελείται ιδιαίτερα από την ύπαρξη του Registry καθώς χρησιμοποιεί αυτές τις πληροφορίες για να υλοποιήσει τις λειτουργίες του. Οι προγραμματιστές και οι επιχειρηματικοί αναλυτές επωφελούνται επίσης καθώς μπορούν να χρησιμοποιήσουν της πληροφορίες για να επιλέξουν τα κομμάτια του συστήματος που θέλουν και να τα χρησιμοποιήσουν για να στήσουν άλλα εργαλεία.



Εικόνα 12: Αρμοδιότητες Registry Service

Το Repository περιέχει όλο τον κώδικα για όλα τα κομμάτια του συστήματος που αποτελούν τις υπηρεσίες που προσφέρονται. Η ουσιώδης διαφορά είναι ότι δεν περιέχει αναγκαστικά πληροφορίες για το πως συνδέονται οι διεργασίες, αλλά τον κώδικα που τις υλοποιεί.

Τέλος πρέπει να γίνει κατανοητό πότε η χρήση της αρχιτεκτονικής των SOA είναι καλή επιλογή και πια είναι τα πλεονεκτήματα και μειονεκτήματα της. Το κύριο μειονέκτημα της είναι ότι απαιτεί πολύ καλή οργάνωση των απαιτήσεων και των αναγκών του συστήματος πριν καν ξεκινήσει η κατασκευή του. Το τελικό σύστημα θα είναι και αρκετά πιο αυστηρό σε απαιτήσεις, από όποια εφαρμογή θέλει να συνδεθεί πάνω στο ESB. Ο λόγος είναι ότι για να εξασφαλισθεί η ορθότητα πρέπει να υπάρχουν απαιτήσεις προς όλους όσο μικρό ή μεγάλο είναι το σύστημα.

Το κέρδος είναι όπως έχει γίνει ήδη προφανές ότι εξασφαλίζονται με σχετικά προφανή τρόπο οι επιχειρησιακές απαιτήσεις με τρόπο που είναι εύκολα κατανοητός ακόμα και από άτομα χωρίς τεχνικές γνώσεις. Είναι ιδανική αρχιτεκτονική για συστήματα όπου είναι απαραίτητο να μην χαθεί ούτε ένα μήνυμα, ούτε μία εντολή. Αυτή είναι και η ποιοτική διαφορά του από τα microservices.

Ένα ακόμα καλό, αν και λιγότερο σημαντικό για οργανισμούς που θέλουν τα τρέχουν σε δικές τους υποδομές είναι ότι όλα τα cloud services σήμερα υλοποιούν μία μορφή ESB και το δίνουν προς χρήση στους πελάτες τους για να στήσουν τις δικές τους υποδομές. Αυτό μειώνει το κόστος της αρχικής επένδυσης αλλά έρχεται με αυξημένο λειτουργικό κόστος αργότερα.

1.2.5 SOA – Παράδειγμα προς μελέτη

Παίρνοντας μια τράπεζα, σαν παράδειγμα, στο Business process Layer, ζουν η ιστοσελίδα, το σύστημα συναλλαγών, το σύστημα αποστολείς ειδοποιήσεων, τα συστήματα που χρησιμοποιούν οι υπάλληλοι στις τράπεζες για να κάνουν την δουλειά τους, το λογισμικό των ATMs και πολλά ακόμη.

Ο Business orchestration manager στη συγκεκριμένη περίπτωση θα δούλευε ως εξής. Αν ένας άνθρωπος βγάλει λεφτά από ένα ATM (άνθρωπος με διαδικασίες) πρέπει και η εφαρμογή στο κινητό του να δείξει αυτή την συναλλαγή (διαδικασίες με διαδικασίες). Αν ο άνθρωπος αποφασίσει να πάει στο ταμείο για να βγάλει λεφτά και αλληλοεπιδρά με τον υπάλληλο, ο τρόπος που θα αλληλοεπιδράσουν για να είναι έγκυρη η συναλλαγή είναι πλήρως ορισμένη (άνθρωπος με άνθρωπο).

Θα μπορούσε το σύστημα να χρησιμοποιήσει το ESB του AWS. Σε αυτό, το Repository θα περιείχε τον κώδικα που υλοποιεί τις υπηρεσίες που προσφέρονται ή που είναι για εσωτερική χρήση και θα μπορούν να είναι αποθηκευμένες σε ένα version control system όπως το *git*.

1.2.6 API – Abstract Programming Interface

Στην εποχή του διαδικτύου υπάρχουν άπειρα συστήματα που αλληλοεπιδρούν με άπειρα άλλα. Τα APIs επιτρέπουν την επικοινωνία μεταξύ εφαρμογών. Το κύριο πλεονέκτημα τους είναι ότι επιτρέπουν να συνδεθεί μία εφαρμογή με μία άλλη χωρίς να παίζει ρόλο η εσωτερική τους δόμηση. Η ιδέα είναι ότι εφόσον η πρώτη προσφέρει τις υπηρεσίες της μέσω ενός API, όλες οι άλλες μπορούν να καταναλώνουν αυτό το API χωρίς να χρειάζεται να ξέρουν τίποτα άλλο για την εφαρμογή. Αν ακόμα υποθέσουμε ότι υπάρχουν πολλές εφαρμογές που χρησιμοποιούν παρόμοια δομημένα API τότε η υλοποίηση μιας εφαρμογής που τα χρησιμοποιεί όλα είναι μακράν απλούστερη.

Τα API μπορούν να χωριστούν σε ομάδες με βάση 2 κατηγοριοποιήσεις. Μπορούν να χωρισθούν με τη βάση μορφή της σειριοποίησης των δεδομένων που προσφέρουν (JSON, XML, Protobuff, κτλ.) ή με βάση την δόμηση του API του ίδιου (REST, GraphQL, SOAP, RPC). Δεν υπάρχει κάποιος κανόνας που υποχρεώνει ένα API να ανήκει μόνο σε 1 κατηγορία από τις παραπάνω, αντίθετα τα καλύτερα APIs υλοποιούν πολλές από τις παραπάνω τεχνολογίες. Να σημειωθεί ότι όλες οι τεχνολογίες που θα εξεταστούν θα είναι τεχνολογίες που χρησιμοποιούνται στο διαδίκτυο αλλά ο όρος API χρησιμοποιείται και για προγράμματα που επικοινωνούν μέσω άλλων πρωτοκόλλων ή που συνυπάρχουν στο ίδιο σύστημα και θέλουν να επικοινωνήσουν. [7]

```xml
<bookstore>
 <book category="COOKING">
  <title lang="en">Everyday Italian</title>
  <author>Giada De Laurentiis</author>
  <year>2005</year>
  <price>30.00</price>
 </book>
 <book category="CHILDREN">
  <title lang="en">Harry Potter</title>
  <author>J K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
 </book>
 <book category="WEB">
  <title lang="en">Learning XML</title>
  <author>Erik T. Ray</author>
  <year>2003</year>
  <price>39.95</price>
 </book>
</bookstore>
```

Εικόνα 13:Bookstore Data in XML

Ως προς την σειριοποίηση η πιο παλιά τεχνολογία είναι η Extensible Markup Language (XML). Πρώτο παρουσιάστηκε το 1996 για να ανανεωθεί τελευταία φορά το 2006. Σκοπός της ήταν να είναι μία εύκολη δομή κατάλληλη για το internet που θα μπορούσε να παρουσιάσει κάθε μορφή δομημένων δεδομένων.

Σήμερα χρησιμοποιείται ακόμα και πολλές παρόμοιες δομές αρχείου έχουν ευρεία χρήση όπως ή html που χρησιμοποιείται για την παρουσίαση ιστοσελίδων, η xaml που χρησιμοποιείται κυρίως για να παρουσιάσει UI σε άλλες πλατφόρμες όπως της Microsoft. Φυσικά η ίδια η xml είναι ικανή να παρουσιάσει όλες τις μορφές δεδομένων. [8]

Η πιο συχνή μορφή σειριοποίησης δεδομένων είναι σε JavaScript Object Notation (JSON). Δεν υπάρχει ξεκάθαρη ημερομηνία δημιουργίας της τεχνολογίας γιατί ξεκίνησε ως υπό-τμήμα ενός plugin που έτρεχε σε Netscape για να δώσει παραπάνω δυνατότητες στο τότε ακόμα αδύναμο browser. Ακολούθησαν πολλά μέχρι το 2002 που δημιουργήθηκε το JSON.org και το 2006 που το Yahoo δημιούργησε το πρώτο γνωστό JSON API. [9]

```json
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

Εικόνα 14: JSON Contact Information

Η σύνταξη αποτελεί υποσύνολο της JavaScript αλλά λειτουργικό ισοδύναμο της xml καθώς όλες οι μορφές δεδομένων μπορούν επίσης να αναπαρασταθούν με JSON. Το πλεονέκτημα του είναι ότι η σύνταξη του είναι πολύ πιο πυκνή, δεν χρειάζονται tags για άνοιγμα και κλείσιμο και μπορεί να αναπαραστήσει λίστες. Ένα επιπλέον καλό είναι ότι η αναπαράσταση ενός json στην μνήμη είναι η ίδια με αυτή της JavaScript από όπου και προήλθε. Έτσι είναι πολύ πιο εύκολο να διαβαστούν και να γραφτούν δεδομένα σε αυτή που είναι και η βασική γλώσσα των σελιδοδεικτών. [10]

Τα XML και JSON αρχεία έχουν ένα κοινό στοιχείο. Αναπαριστούν και τα δύο την πληροφορία τους με κείμενο. Αυτό αν και βολικό δεν είναι υποχρεωτικό. Συνηθίζεται στο διαδίκτυο να γίνονται τα πάντα κείμενο γιατί είναι εύκολο να βρεθούν λάθη και είναι πιο γρήγορη η ανάπτυξη χωρίς μεγάλο κόστος στην ταχύτητα των προγραμμάτων. Σήμερα όμως υπάρχουν πολλές εφαρμογές που τρέχουν σε browsers και είναι ιδιαίτερα απαιτητικές σε κατανάλωση δικτύου (download). Αυτό δημιούργησε την ανάγκη για πρωτόκολλα επικοινωνίας με καλύτερες επιδόσεις από αυτές που ήδη υπήρχαν.

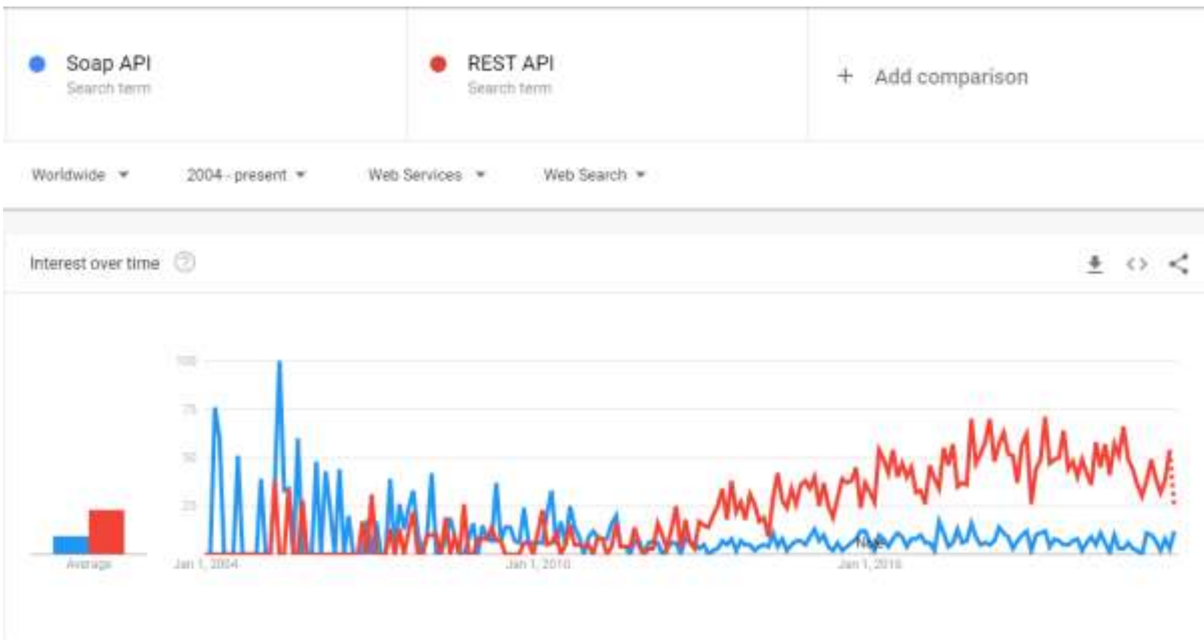
Το καλύτερο πρωτόκολλο που υπάρχει αυτή την στιγμή είναι το Protobuff. Πρόκειται για μια υβριδική τεχνολογία όπου η περιγραφή των δεδομένων γίνεται με κείμενο αλλά η σειριοποίηση γίνεται σε δυαδική μορφή. Έτσι ο δημιουργός του API δίνει στους καταναλωτές του ένα αρχείο ".proto" που περιέχει την περιγραφή και ορίζει πλήρως πως θα γίνει η επικοινωνία. Αυτή η τεχνική αν και πιο δύσκολη στην υλοποίηση από ότι οι κλασικές, είναι ο καλύτερος τρόπος να στείλεις δεδομένα τα οποία θα είναι μέγιστα συμπυκνωμένα αλλά και εύκολα χρησιμοποιήσιμα. [11]

```
syntax = "proto2";

package scommon;

message Message {
required string From = 1;
required string To = 2;
required string Body = 3;
required string time = 4;
}
```

Εικόνα 15: Protobuff Message Description

Έχοντας αναλύσει πλήρως το σε τι μορφή θα λάβει ή θα δώσει κάποιος δεδομένα πρέπει να γίνει αναφορά και στη μορφή του API του ίδιου. Οι διάφορες τεχνολογίες έχουν διαφορές οι οποίες όμως δεν κάνουν αναγκαστικά την μία καλύτερη από την άλλη αλλά δίνει στη μία πλεονεκτήματα που οι άλλες δεν έχουν.

Αν και τυπικά χρησιμοποιείται ακόμα και υπάρχουν εφαρμογές που ακόμα αναπτύσσονται με αυτή την αρχιτεκτονική, θα γίνει μια αναφορά στο **SOAP** που θα θυμίζει ιστορική, καθώς πλέον δεν χρησιμοποιείται για την κατασκευή νέων API για λόγους που επίσης θα αναλυθούν. Το Simple Object Access Protocol χρησιμοποιεί κυρίως XML για να μεταφέρει δεδομένα μέσω του HTTP πρωτοκόλλου, αν και θεωρητικά θα μπορούσε να τρέξει σε οποιοδήποτε text transfer protocol. Ο λόγος που δεν χρησιμοποιείται πλέον είναι το ότι απαιτεί, η κάθε υπηρεσία να ορίζει όλες τις διαθέσιμες διαδικασίες, τις παραμέτρους, τους τύπους και ότι άλλο θεωρείται απαραίτητο για να περιγράψει κανείς ένα σύστημα. Αν και αυτό μπορεί να θεωρηθεί καλό και επέτρεπε την αυτόματη δημιουργία κώδικα που χειρίζεται το API θεωρήθηκε αρκετά δύσκαμπτο για το Internet που κινήθηκε προς τεχνολογίες που δίνουν στο προγραμματιστή μεγαλύτερη ευελιξία.

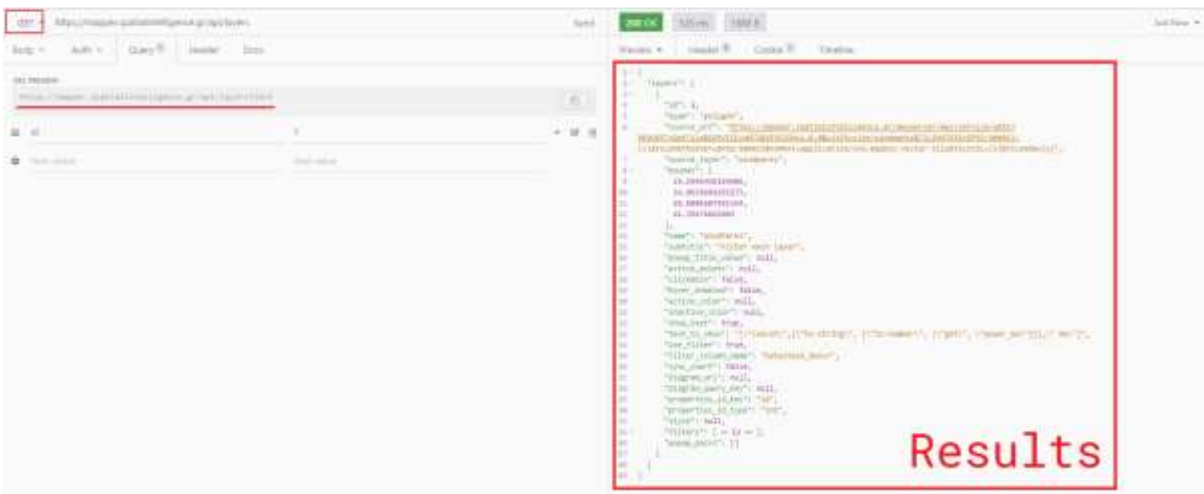


Εικόνα 16: Soap vs Rest on Google Trends

Όπως φαίνεται στο σχήμα τα REpresentational State Transfer (REST) APIs είναι η διαδεδομένη τεχνολογία σήμερα σύμφωνα και με τα στατιστικά που δίνει η Google. Αυτό μπορεί να ερμηνευτεί ως το ότι σήμερα όταν κάποιος ψάχνει με ποια τεχνική να δομήσει τον κώδικα του προτιμάει REST.

Τα δεδομένα κωδικοποιούνται σε JSON, HTML ή σε XML και στέλνονται μέσα από HTTP πρωτόκολλο ως κείμενο, συνηθίζεται σε JSON. Πρόκειται για ένα ιδιαίτερα ελαφρύ πρωτόκολλο. Ορίζει ένα σύνολο από ενέργειες (GET, POST, UPDATE, DELETE, PATCH) που μπορεί κάποιος να στείλει σε μία διεύθυνση μαζί με τα σχετικά δεδομένα, αν υπάρχουν. Λόγο αυτής της απλότητας η κοινότητα έχει ορίσει διαφορετικές καλές πρακτικές για να βοηθήσει στην ορθή δόμηση καινούργιων APIs (RESTless, RESTful). Ένα ακόμα καλό του REST είναι ότι δεν κρατάει καταστάσεις (Stateless). Κάθε Request αξιολογείται ανεξάρτητα από τις εντολές που ήρθαν πριν από αυτό και δεν επηρεάζει αυτά που ακολουθούν.

Αν πρέπει να αναφέρουμε ένα μειονέκτημα του REST είναι αποτέλεσμα της ευελιξίας του. Λόγω της απλότητας του υπάρχουν απεριόριστοι τρόποι κάποιος να υλοποιήσει το δικό του API και έτσι είναι εύκολο να είναι ασυνεπή και να υπάρχουν λάθη που σε άλλες τεχνολογίες δεν θα υπήρχαν. Συνηθίζεται λοιπόν πριν χρησιμοποιηθεί έναν τέτοιο API να πρέπει να διαβαστούν οι οδηγίες που έγραψε ο δημιουργός το, αν το έκανε, αλλιώς είναι πρακτικά άχρηστο και μπορούν να βρεθούν λεπτομέρειες μόνο μέσα από πειραματισμό. [7]



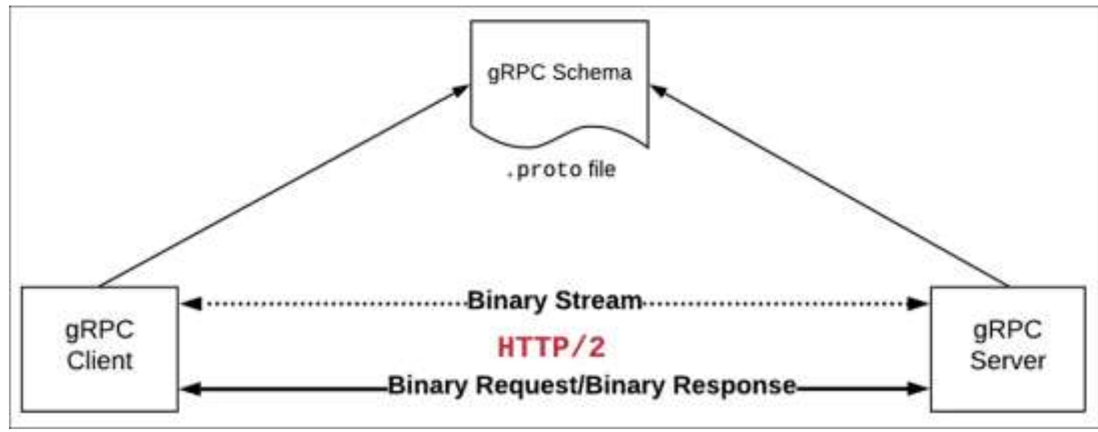
Εικόνα 17:Rest API Example

Είναι προφανές λοιπόν ότι όταν δημιουργείται ένα REST API πρέπει ο δημιουργός του να γράψει και το κατάλληλο documentation ώστε οι χρήστες του να μπορούν να το χρησιμοποιήσουν. Υπάρχουν διάφοροι τρόποι να γίνει αυτό, οι περισσότεροι προτιμούν ένα website όπου έχουν μία λίστα με το τι μπορεί να κάνει το API και παραδείγματα για το πως να το χρησιμοποιήσει κάποιος. Το πρόβλημα είναι ότι υπάρχει ανομοιομορφία στο πως στήνετε αυτό το site και επειδή δεν υπάρχει η πληροφορία σε απλή μορφή κειμένου έχει χαθεί η δυνατότητα των SOAP να δημιουργούν κώδικα κατευθείαν από το documentation.

Για αυτό δημιουργήθηκε το OpenAPI 3.0. Δημιουργήθηκε το 2010 από τον Tony Tam υπό το όνομα Swagger και αργότερα μετανομάστηκε σε OpenAPI Specification (OAS) το 2016, όντας πλέον σε ξεχωριστή οντότητα με το ίδιο όνομα υπό την χορηγία του Linux Foundation. Το OpenAPI Specification ορίζει ένα standard, αγνωστικό ως προς την γλώσσα, που λειτουργεί ως ένα interface για τα RESTful APIs. [12] Επιτρέπει σε ανθρώπους αλλά και σε προγράμματα να βρουν και να καταλάβουν τις δυνατότητες μιας υπηρεσίας χωρίς να έχουν πρόσβαση στον πηγαίο κώδικα, το documentation, ή με άλλο τρόπο. Όταν έχει οριστεί σωστά, ένας καταναλωτής του API μπορεί να καταλάβει και να αλληλοεπιδράσει με μία υπηρεσία με των ελάχιστο δυνατό κόπο. Επίσης επιτρέπει σε προγράμματα να

δημιουργήσουν κώδικα που να αλληλοεπιδρά με το API αυτόματα. Κάποιος μπορεί να ορίσει πλήρως όλες τις λειτουργικές δυνατότητες τους συστήματος, από τις παραμέτρους των κλήσεων μέχρι και παραδείγματα χρήσης και να γράψει την περιγραφή σε ένα αρχείο κειμένου το οποίο αργότερα θα δώσει στους χρήστες του.

Υπάρχουν περιπτώσεις όπου το REST δεν καλύπτει όλες τις ανάγκες μιας εφαρμογής γιατί χρειάζεται μια αμφίδρομη επικοινωνία. Το REST δεν είναι σχεδιασμένο να κάνει κάτι τέτοιο. Το κενό που άφησε το SOAP ήρθαν να το καλύψουν οι διάφορες Remote Procedure Call (RPC) τεχνολογίες όπως το gRPC που φτιάχτηκε από την Google (πλέον είναι open source) και το SignalR της Microsft. Αυτές οι τεχνολογίες επιτρέπουν έναν υπολογιστή να εκτελέσει μία διαδικασία που τρέχει σε κάποια άλλη διεύθυνση στο δίκτυο και ουσιαστικά επιτρέπει τη μη διακοπτόμενη επικοινωνία μεταξύ υπολογιστών. Αυτή είναι και η ουσιώδης διαφορά με το REST. Συνήθως οι RPC τεχνολογίες χρησιμοποιούν δυαδικά πρωτόκολλα επικοινωνίας αλλά δεν υπάρχει κανόνας που το υποχρεώνει. Το gRPC χρησιμοποιεί το HTTP 2.0 που έχει πλέον την δυνατότητα να στέλνει binary δεδομένα και τα κωδικοποιεί με Protobuff. [13]



Εικόνα 18: gRPC over HTTP2

Μία καινούργια τεχνολογία που έχει εμφανιστεί τα τελευταία χρόνια και έχει αποκτήσει πολλούς χρήστες πολύ γρήγορα είναι το GraphQL. Σχεδιάστηκε από το Facebook για εσωτερική χρήση το 2012 και δημοσιεύτηκε το 2015. Από το 2018 αποτελεί αυτόνομη οντότητα κάτω από το Linux Foundation. Σχεδιάστηκε να κωδικοποιεί τα δεδομένα του με JSON και προσπαθεί να λύσει πολλά από τα προβλήματα των REST API. [14] Αντί απλές λέξεις για να περιγράφουν το τι θέλεις από ένα URI έχεις μία καινούργια γλώσσα σχεδιασμένη για να ζητάς από το server ακριβώς αυτό που θες.

Explorer

GraphiQL Prettify History Explorer

payloads
roadster
rocket
rockets
limit: 10
offset:
active
boosters
company
cost_per_launch
country
description
diameter
engines
first_flight
first_stage
height
id
landing_legs
mass
name
payload_weights
second_stage
stages
success_rate_pct
type
wikipedia

```
{
  rockets(limit: 10) {
    id
    name
    stages
  }
}
```


```
{
  "data": {
    "rockets": [
      {
        "id": "falcon1",
        "name": "Falcon 1",
        "stages": 2
      },
      {
        "id": "falcon9",
        "name": "Falcon 9",
        "stages": 2
      },
      {
        "id": "falconheavy",
        "name": "Falcon Heavy",
        "stages": 2
      },
      {
        "id": "starship",
        "name": "Starship",
        "stages": 2
      }
    ]
  }
}
```


Εικόνα 19: SpaceX GraphQL API [15]

Μπορείς με μία εντολή σε ένα GraphQL server να πάρεις όλα τα δεδομένα που θες και μόνο αυτά, αντί να πρέπει να κάνεις πολλά request στο REST API για να πάρεις τα δεδομένα που θες και μερικά που δεν θες. Έχει ενσωματωμένο έλεγχο εγκυρότητας δομών και τύπων και για αυτό είναι ιδιαίτερα εύχρηστη από την μεριά του καταναλωτή του API. Το κύριο πρόβλημα της είναι το caching καθώς η ευελιξία από την μεριά των χρηστών, τους κάνει να υλοποιούν ελαφρός διαφορετικές εντολές ο καθένας και άρα τα δεδομένα που ζητούν δεν μπορούν να σωθούν για να επαναχρησιμοποιηθούν. Επίσης είναι αρκετά ποιο περίπλοκο στην υλοποίηση από ότι το ισοδύναμο REST API και άρα δεν συμφέρει για μικρότερα συστήματα.

Chapter 2: Προδιαγραφές μιας δικτυακής εφαρμογής για τη γραφική απεικόνιση χρονοσειρών με χωρική αναφορά

Πριν μιλήσουμε για την περιγραφή του συστήματος που δημιουργήθηκε καλό είναι να αποσαφηνιστούν κάποιοι όροι στον τίτλο.

- Ο όρος "χωρική αναφορά" θα μπορούσε να σημαίνει σημείο, πολύγωνο ή άλλη πληροφορία που αναπαρίσταται σε χάρτη και τοποθετεί τα δεδομένα στον παγκόσμιο χάρτη.
- Οι χρονοσειρές αποτελούνται από μετρήσεις φυσικών μεγεθών που αναπαρίστανται σε χρονικό διάγραμμα.

2.1 Περιπτώσεις χρήσης

Το σύστημα που στήθηκε είναι αγνωστικό ως προς τον τύπο των δεδομένων. Αυτό το κάνει ιδιαίτερα ευέλικτο και ικανό να υποστηρίξει πολλές επαγγελματικές ομάδες που εργάζονται σε τελείως διαφορετικά γνωστικά πεδία. Ενεργειακά, ιατρικά ή μετεωρολογικά δεδομένα και γενικά ότι μπορεί να αναπαρασταθεί σε διάγραμμα το οποίο όμως έχει και γεωγραφική τοποθέτηση είναι ιδανική πληροφορία για να αναπαρασταθεί στο σύστημα.

Παράδειγμα 1: Κάποιος ο οποίος θέλει να πάρει μετεωρολογικά δεδομένα μίας πόλης από 1, 2 ή και παραπάνω σταθμούς και να τα αναλύσει, ή να εξάγει ένα υποσύνολο, μπορεί να το κάνει. Μπορεί να εστιάσει σε μια χρονική περίοδο που τον ενδιαφέρει, ακόμα και να ρυθμίσει το σύστημα να του δίνει κινητούς μέσους όρους ή αθροίσματα εβδομάδας.

Παράδειγμα 2: Τα ενεργειακά δεδομένα είναι επίσης πληροφορία που μπορεί να συνδέετε με μία γεωγραφική αναφορά όπως ένας σταθμός, ένα αιολικό πάρκο μία γεωγραφική περιφέρεια ή μια χώρα. Κάποιος αναλυτής θα μπορούσε να φορτώσει τα Ευρωπαϊκά ενεργειακά δεδομένα κατανάλωσης και να προσπαθήσει να βγάλει συμπεράσματα ή να πάρει αποφάσεις.

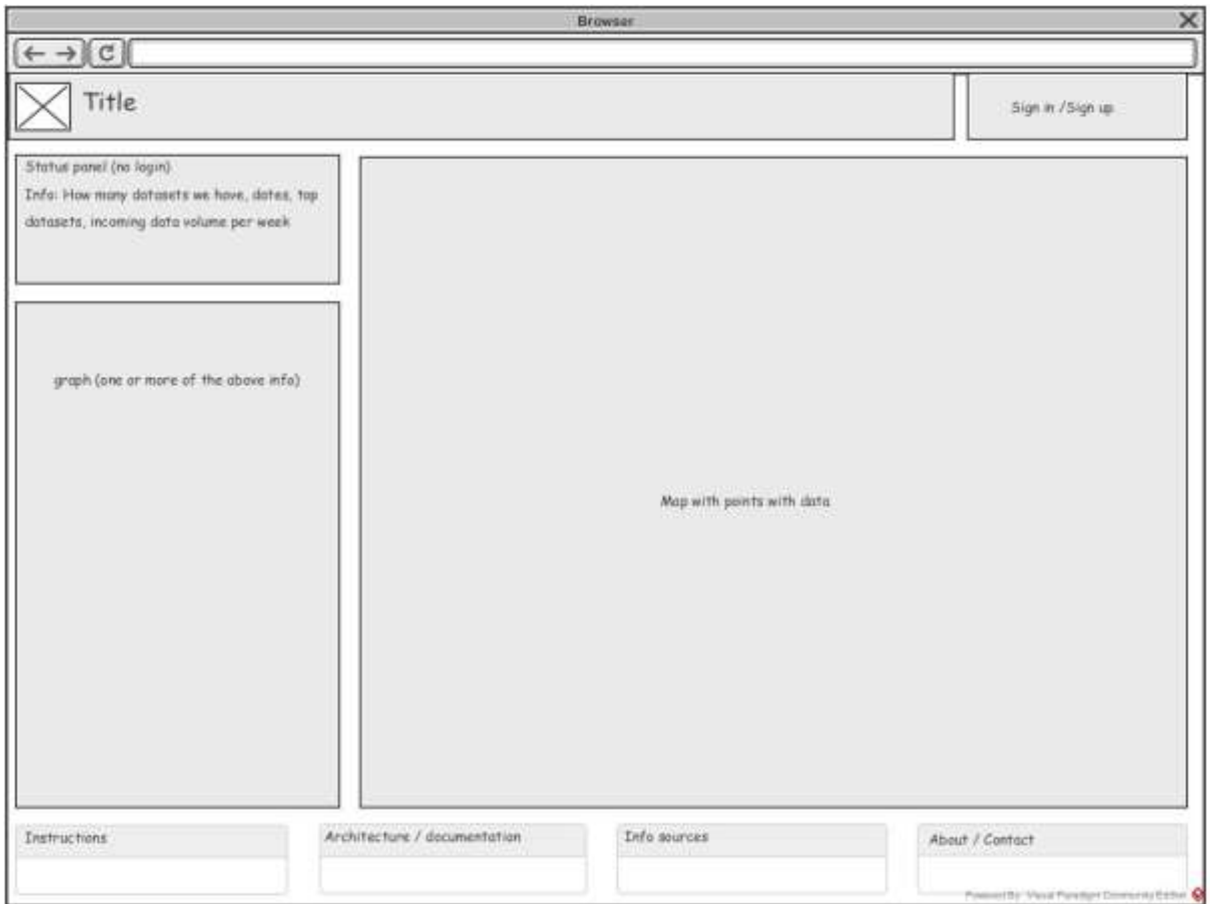
Παράδειγμα 3: Το σύστημα θα μπορούσε να χρησιμοποιηθεί για την ανάλυση των μολύνσεων μιας πανδημίας σε μία χώρα ή για την σύγκριση της πορείας της με μία άλλη. Αφού φορτωθεί στο σύστημα με ένα χάρτη γεωγραφικών περιοχών και τα δεδομένα που θέλουμε να αναλύσουμε, τα οποία θα μπορούσαν να είναι χρονοσειρές για πόσους έχουν νοσήσει, για πόσους έχουν πεθάνει, ή για τους εμβολιασμένους, κτλ. Έτσι θα μπορούσε κάποιος να απομονώνει και να μελετάει τα δεδομένα μια περιοχής που των ενδιαφέρει ξεχωριστά από μια άλλη και να βλέπει την πορεία της πανδημίας ανά νομό, πόλη, χώρα ή όπως επιλέξει.

2.2 Γενικά

Ξεκινάμε από την παραδοχή ότι πρόκειται για δικτυακή εφαρμογή (web app) αλλά πρέπει να οριστεί ποιες συσκευές υποστηρίζονται καθώς όλες οι συσκευές που έχουν Internet, έχουν πρόσβαση στο site. Η ιστοσελίδα θα έχει σαν κύριο στόχο το να υποστηρίζει desktop οθόνες αλλά θα γίνει προσπάθεια να υποστηρίξει όλα τα μεγέθη οθονών ακόμα και τα πολύ μικρά. Αυτό θα καταστήσει το site λειτουργικό και σε tablet ή κινητά. Το πρώτο πράγμα που χρειάζεται για την κατασκευή του είναι ένα σχέδιο του πως θέλουμε το site να φαίνεται. Το πρώτα σχέδια που γίνονται σε κάθε εφαρμογή λέγονται *Wireframes*.

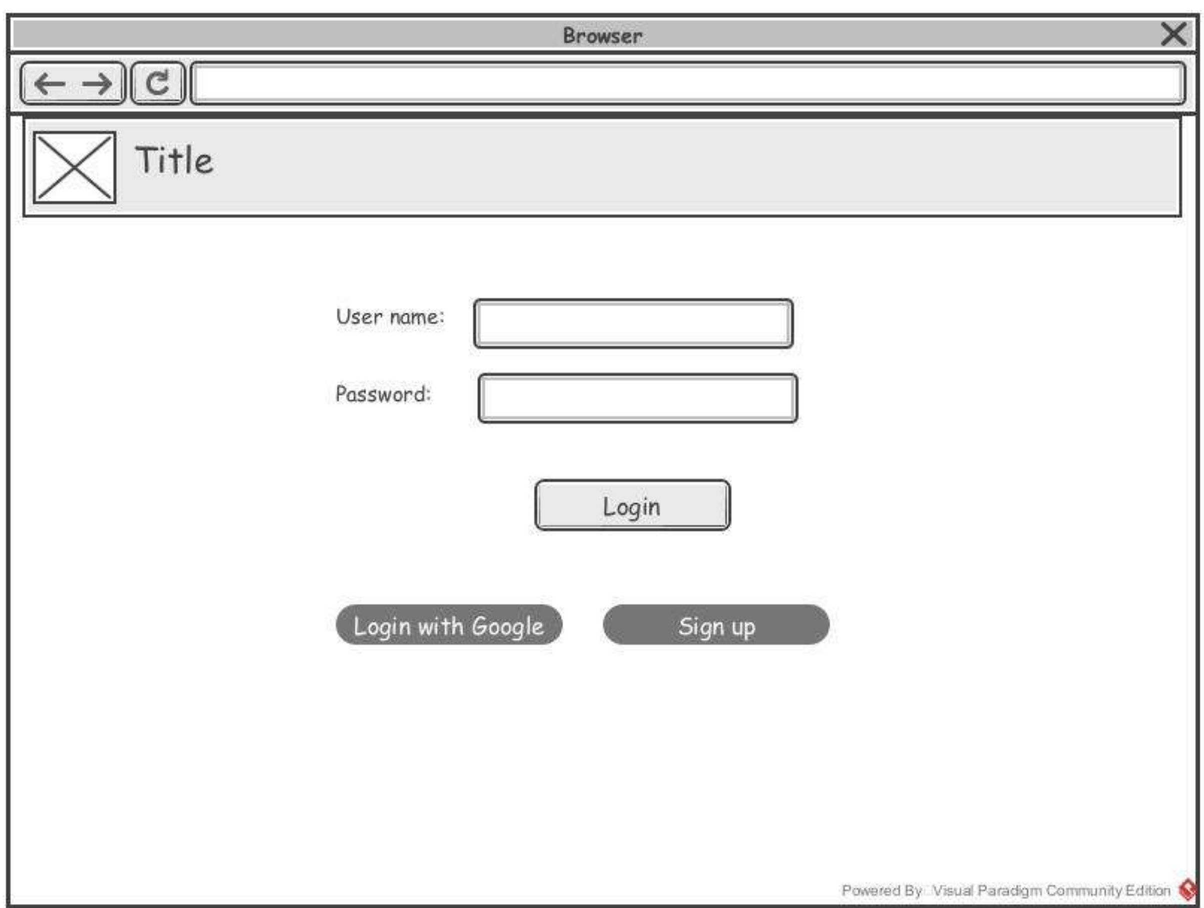
Το σχέδιο είναι στο Landing Page να έχει ένα χάρτη και μερικά στατιστικά για το σύστημα. Ουσιαστικά θα δείχνει ένα set από δημόσια δεδομένα για να κάνει το σύστημα οπτικά πλήρες όταν ένας καινούργιος χρήστης έρχεται στην πλατφόρμα για να δει τις ικανότητες της. Αριστερά βάζουμε δύο κουτιά στα οποία θα μπουν λειτουργικές

πληροφορίες για το σύστημα. Για να μπορέσει κάποιος να αποκτήσει πρόσβαση πρέπει να κάνει εγγραφή ή να συνδεθεί με το κουμπί πάνω δεξιά. Αυτό θα τον πάει στη Login Page.



Εικόνα 20: Landing Page

Η Σελίδα Login δεν έχει ιδιαίτερες λειτουργικές απαιτήσεις. Αφού ο χρήστης συμπληρώσει τα στοιχεία του επιστρέφει στην κεντρική σελίδα. Το κουμπί Login with Google αφαιρέθηκε σε δεύτερο χρόνο από το σχέδιο. Το Κουμπί Sign up θα στέλνει τον χρήστη στη Sign Up Page ώστε αν δεν έχει κωδικό να μπορεί να φτιάξει ένα.



Εικόνα 21: Sign In Page

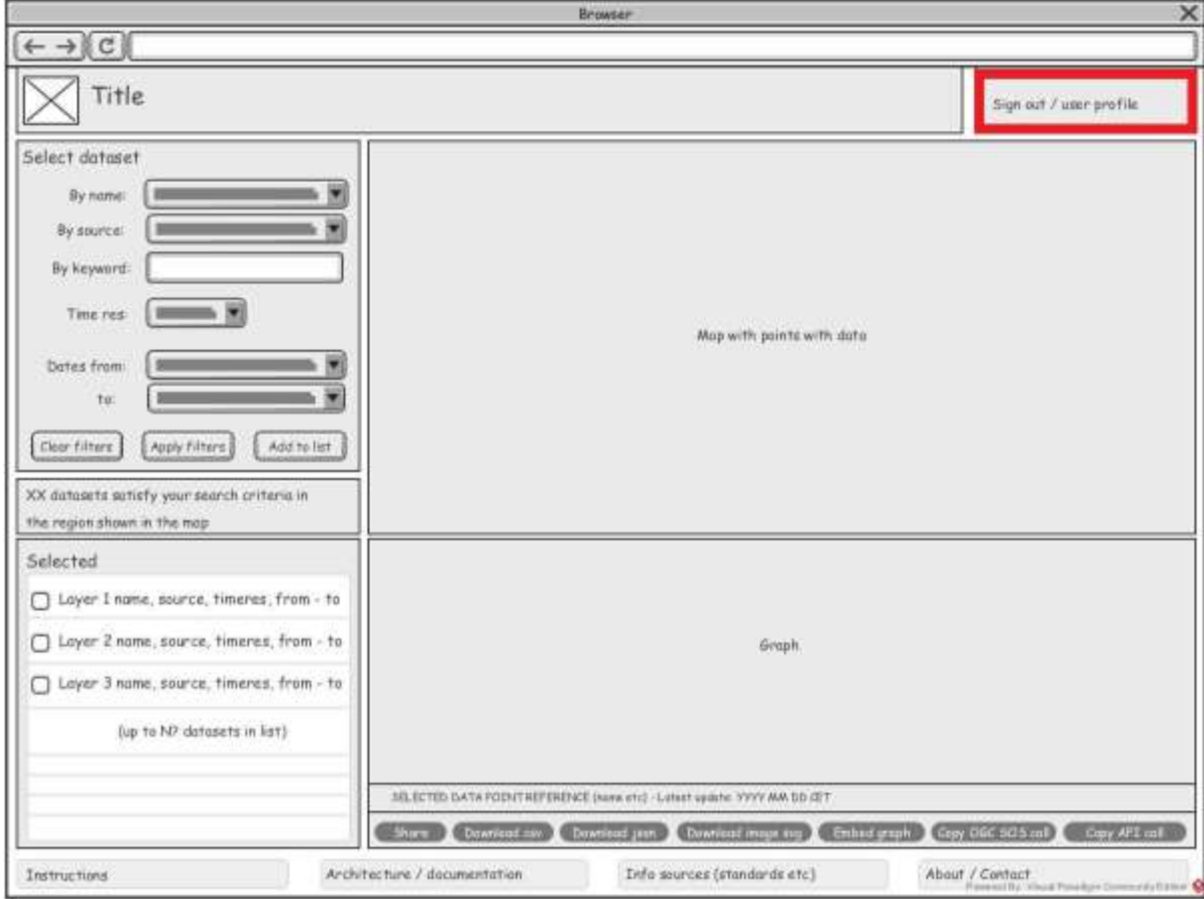
Αφού ο χρήστης συμπληρώσει την φόρμα τον προσθέτουμε στο σύστημα και τον γυρνάμε πίσω στην κεντρική σελίδα. Ζητήθηκε να τηρηθεί το ελάχιστο δυνατό σε σύγχρονες πρακτικές ασφαλείας:

1. Ελάχιστο μήκος κωδικού
2. Να περιέχει μικρά και κεφαλαία γράμματα
3. Να περιέχει τουλάχιστον 1 σύμβολο

Επίσης πρέπει οι κωδικοί να μην κρατιούνται στην βάση σε απλή μορφή κειμένου, το πως είναι κομμάτι της υλοποίησης. Οι τεχνολογίες που χρησιμοποιήθηκαν για να γίνε αυτό υπάρχουν έτοιμες σε βιβλιοθήκες που έχουν δοκιμαστεί εκτενώς ως προς την ασφάλεια. Παρόλο που απαιτείται εγγραφή στο σύστημα οποιοδήποτε μπορεί να έρθει και να φτιάξει έναν κωδικό.

Εικόνα 22: Sign up Page

Αφού λοιπόν ο χρήστης έχει συνδεθεί ή εγγραφεί, επιστρέφει στην Landing page όπου έχει όμως αλλάξει. Φυσικά αν έχει συνδεθεί στο παρελθόν παραμένει συνδεμένος μέχρι να πατήσει το κουμπί "Sign out" πάνω δεξιά. Ως το προς το πως θα φαίνεται αυτό το κουμπί δόθηκε δημιουργική ελευθερία και αποφασίστηκε να εμφανίζει μία λίστα με τις επιλογές με κουμπιά κάτω από αυτό ώστε να μπορούν αργότερα να προστεθούν και άλλες λειτουργικές απαιτήσεις όπως για παράδειγμα λειτουργίες διαχείρισης. Τα GUI πρέπει να αποτελείται από 4 βασικά κομμάτια. Το καθένα από τα κουτιά που αποτελούν το interface για έχει μέσα του ένα Component.

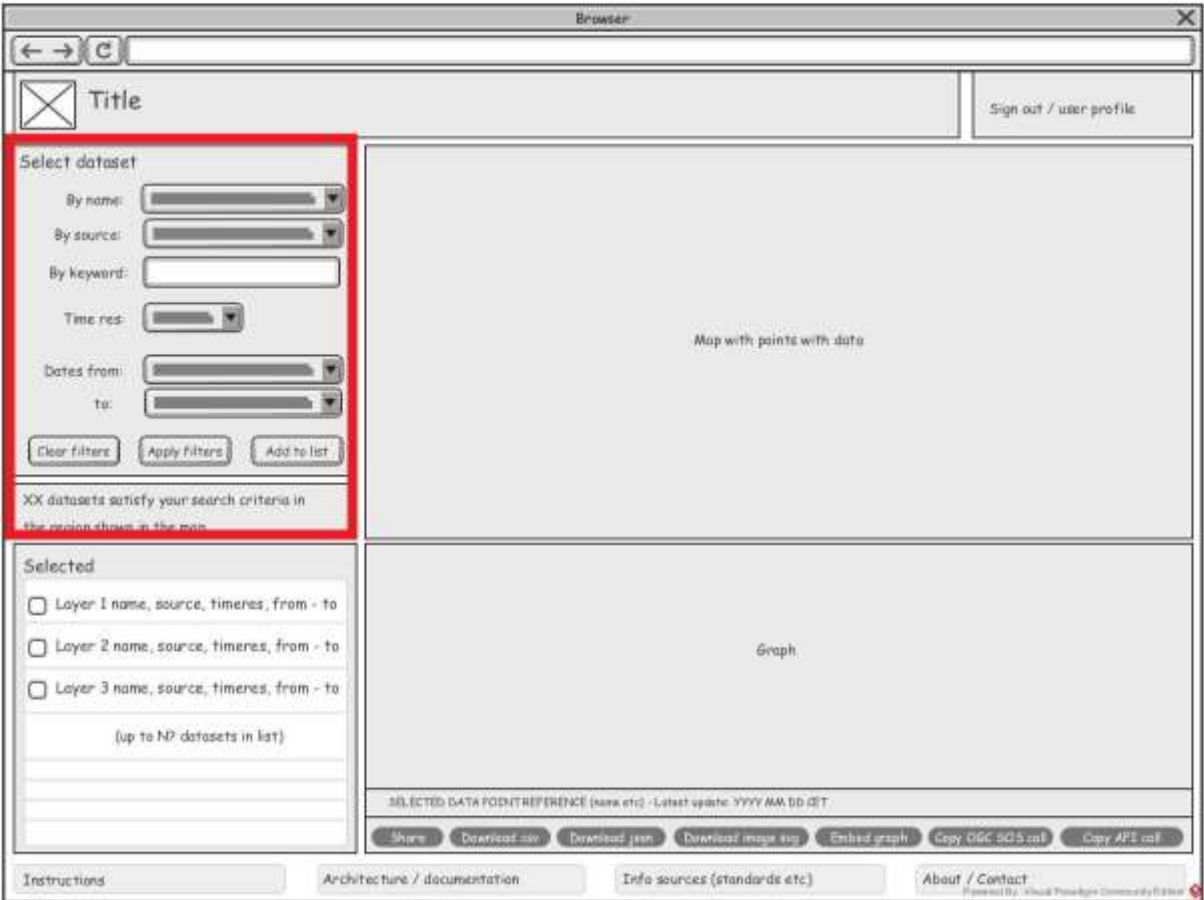


Εικόνα 23: Landing page After logging

Πάνω αριστερά βρίσκεται το Search Component. Στη πλατφόρμα θα υπάρχει η δυνατότητα να αναζητήσει κανείς κάποια set δεδομένων και αυτό θα γίνεται από εκεί. Τα πεδία της φόρμας αντιστοιχούν σε διαφορετικές πληροφορίες που μπορεί να έχουν τα δεδομένα:

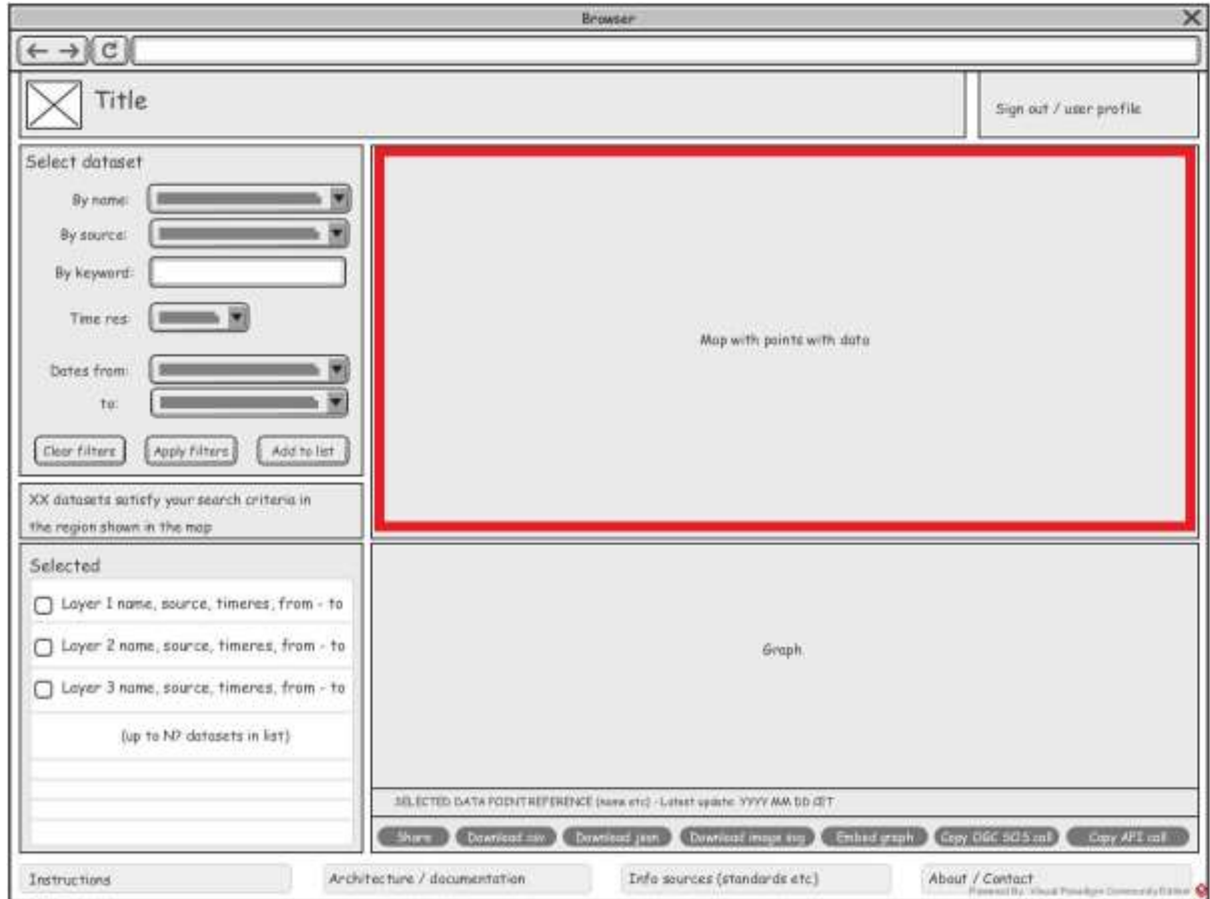
- Το πρώτο πεδίο είναι το όνομα του set και είναι πληροφορία που δίνει ο πάροχος των δεδομένων. Το πεδίο υποστηρίζει "Fuzzy Search", δηλαδή με τα πρώτα γράμματα που πληκτρολογεί ο χρήστης έρχονται προτάσεις για το όνομα που ψάχνει.
- Το δεύτερο είναι μία λίστα που του διαθέσιμους παρόχους δεδομένων. Έτσι μπορεί να διαλέξει ο χρήστης δεδομένα από διαφορετικές χρονοσειρές που μπορούν όμως να αναπαρασταθούν στο ίδιο διάγραμμα.
- Στο τρίτο πεδίο μαζεύονται λέξεις κλειδιά τα οποία μπορεί να έχουν οι χρονοσειρές ανεξάρτητα από τον πάροχο ή το όνομα της χρονοσειράς. Για παράδειγμα μπορεί να υπάρχουν δεδομένα αιολικής ενέργειας από πολλούς παρόχους. Όλα αυτά τα δεδομένα όμως θα μπορούν να έχουν τη λέξη κλειδί για έτσι μπορείς να τις πάρεις όλες με μία αναζήτηση.
- Ακολουθούν 3 πεδία τα οποία αποτελούν ιδιότητες των χρονοσειρών. Πρώτο είναι η χρονική διάσταση των δεδομένων, δηλαδή αν τα δεδομένα είναι ανά ώρα, μέρα, βδομάδα, μήνα, κτλ.
- Μετά από αυτό βρίσκονται τα πεδία "από", "έως" ημερομηνία. Αυτά τα πεδία λειτουργούν σαν περιορισμοί και όταν κάποιος τα ζητήσει δουλεύει σαν σύσταση. Αν μία χρονοσειρά έχει δεδομένά μετά την "από" ημερομηνία ή/και πριν την

“έως” ημερομηνία τότε η αναζήτηση θε την γυρίσει. Αυτά τα πεδία θα επηρεάσουν τα δεδομένα που θα γυρίσει το σύστημα. Αν μία χρονοσειρά είναι αποτέλεσμα της αναζήτησης τότε αυτά τα πεδία θα πουν στο server να μην γυρίσει όλα τα δεδομένα αλλά αυτά που βρίσκονται εντός των ορίων.



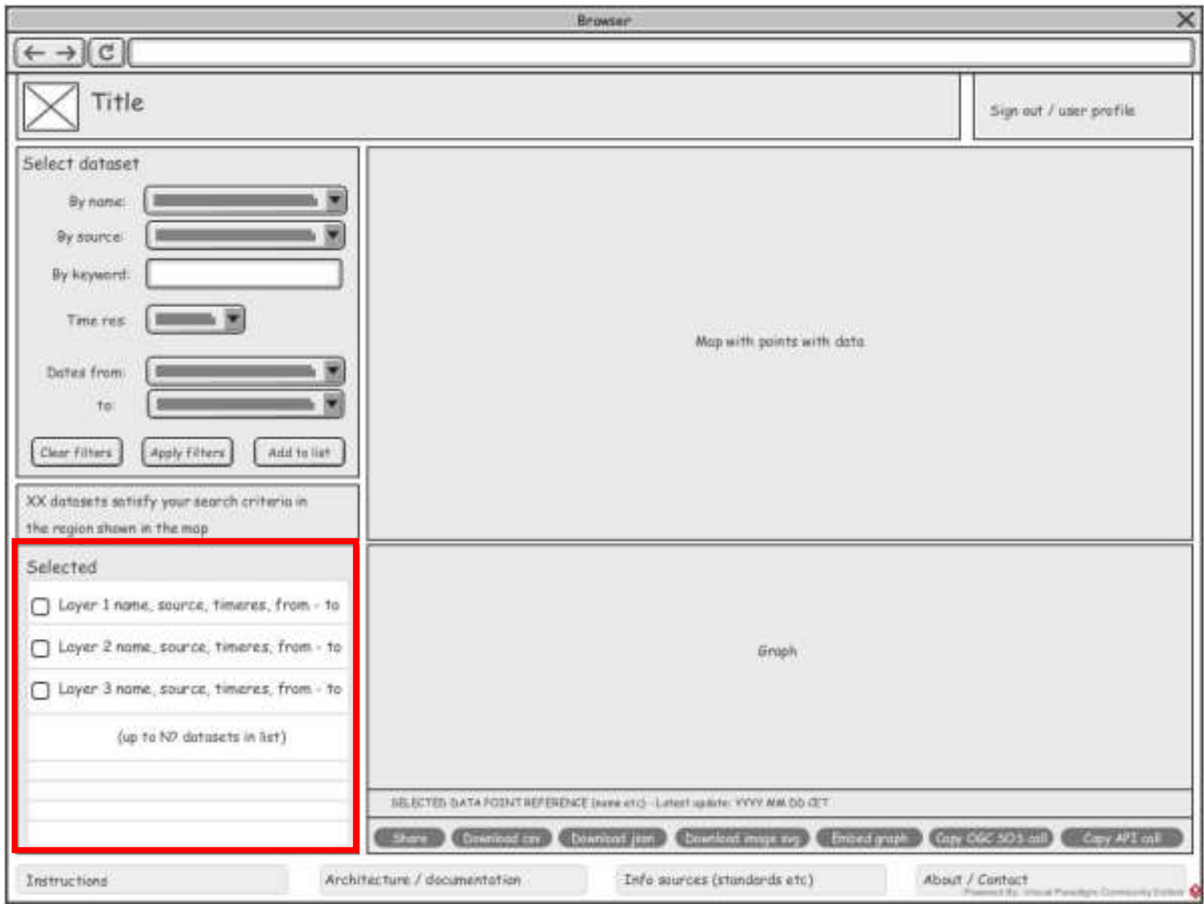
Εικόνα 24: Search Component Highlighted

Πάνω δεξιά βρίσκεται το Map Component όπου θα υπάρχει ένας χάρτης που θα δείχνει τις χωρικές-αναφορές που σχετίζονται με τα αποτελέσματα τις αναζήτησης. Όταν κάποιος πατάει το κουμπί Apply Filter ο χάρτης ανανεώνεται αυτόματα για να αναπαραστήσει τις γεωμετρίες που συνδέονται με το αποτέλεσμα της αναζήτησης. Όταν ο χρήστης θα κάνει click σε ένα σημείο του χάρτη θα εμφανίζεται ένα παράθυρο που θα έχει τις πληροφορίες του σημείο ώστε να είναι σίγουρος ότι διάλεξε το σημείο που τον ενδιαφέρει.



Εικόνα 25: Map Component Highlighted

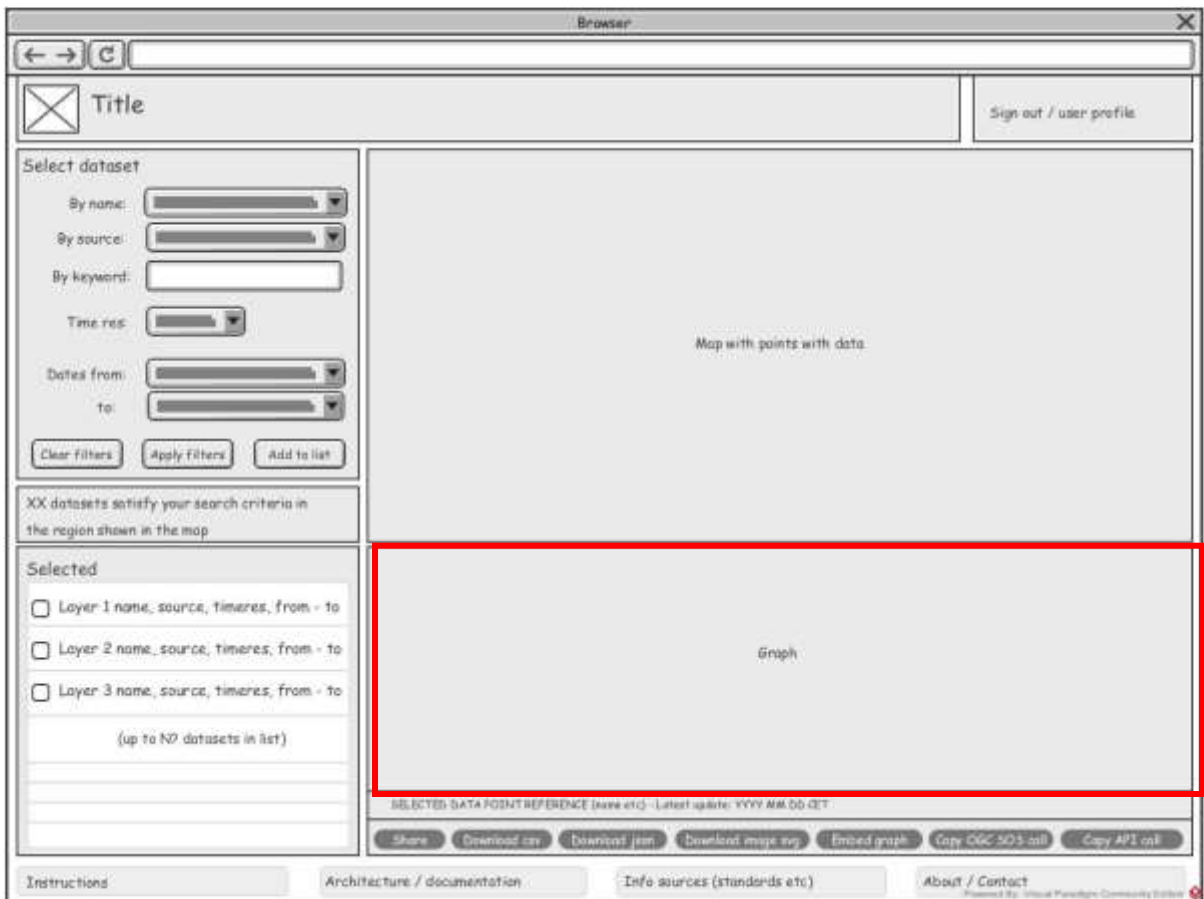
Κάτω αριστερά υπάρχει το Select Component. Σε αυτό ο χρήστης θα διαλέγει τις χρονοσειρές τις οποίες έχει διαθέσιμες να δείξει στο διάγραμμα. Θα υπάρχει πάνω πάνω μία λίστα με τους διαθέσιμους παρόχους και από εκεί θα διαλέγει ο χρήστης τις πληροφορίες ποιου παρόχου θέλει να φορτώσει στο διάγραμμα. Μετά θα εμφανίζεται από κάτω μια λίστα με τις επιλεγμένες χρονοσειρές. Αν ένα set δεδομένων έχει φορτωθεί κατά την αναζήτηση αλλά δεν έχει επιλεγεί στον χάρτη περιοχή όπου να υπάρχουν δεδομένα για αυτό τότε θα εμφανίζεται σαν μία άδεια λίστα. Αν έχει επιλεγεί έστω και έναν σημείο τότε ο χρονοσειρά που αντιστοιχεί σε αυτό θα είναι διαθέσιμη για να την επιλέξει ο χρήστης.



Εικόνα 26: Select Component Highlighted

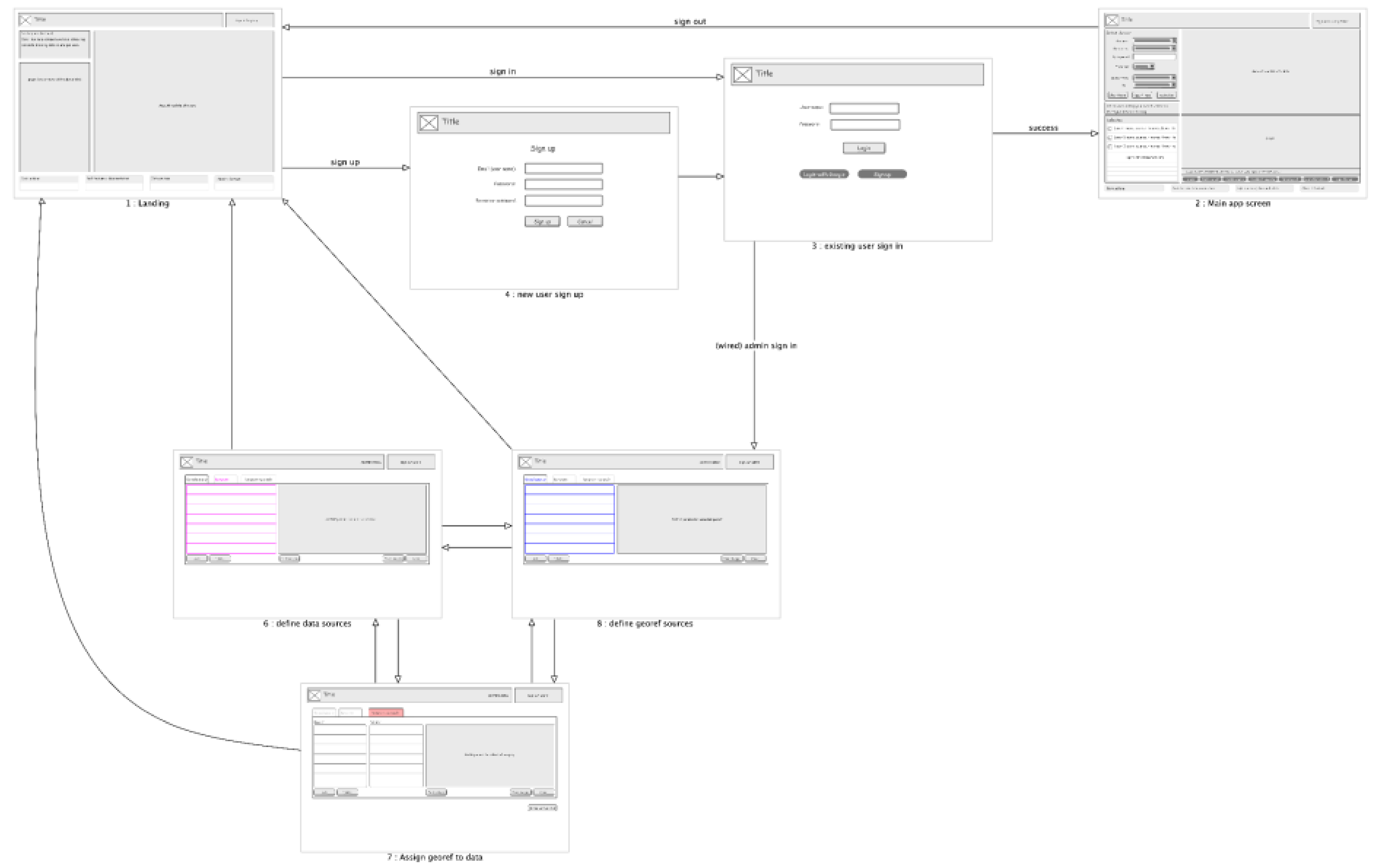
Κάτω δεξιά βρίσκεται το Graph Component στο οποίο θα γίνει η αναπαράσταση των χρονοσειρών. Αυτό το component παίρνει ως είσοδο της επιλογές του χρήστη στο χάρτη και στο Select Component για να βρει ποιες χρονοσειρές θα ζητήσει από το server. Αν αλλάξουν οι επιλογές του χρήστη, τότε το διάγραμμα θα ανανεωθεί αυτόματα με τις τελευταίες επιλογές του.

Κάτω από το διάγραμμα υπάρχουν κάποια κουμπιά. Το κάθε κουμπί δίνει στον χρήστη των αποτέλεσμα των επιλογών του στο διάγραμμα σε διαφορετική μορφή. Αν για παράδειγμα επιλεγούν 2 χρονοσειρές σε ένα χρονικό παράθυρο αυτά τα κουμπιά σου δίνουν τη δυνατότητα να πάρει τα δεδομένα σε μορφή CSV, JSON, σαν εικόνα ακόμα και ως iframe που θα χρησιμοποιηθεί σε άλλο site. Στην περίπτωση του iframe υπάρχει έναν υποσύστημα της ιστοσελίδας που παίρνει ένα URL με το ποια δεδομένα θέλει ο χρήστης και τα σερβίρει σε διάγραμμα χωρίς τίποτα άλλο. Αυτό είναι το URL που δίνεται όταν κάποιος πατάει το κουμπί “embedded graph”.



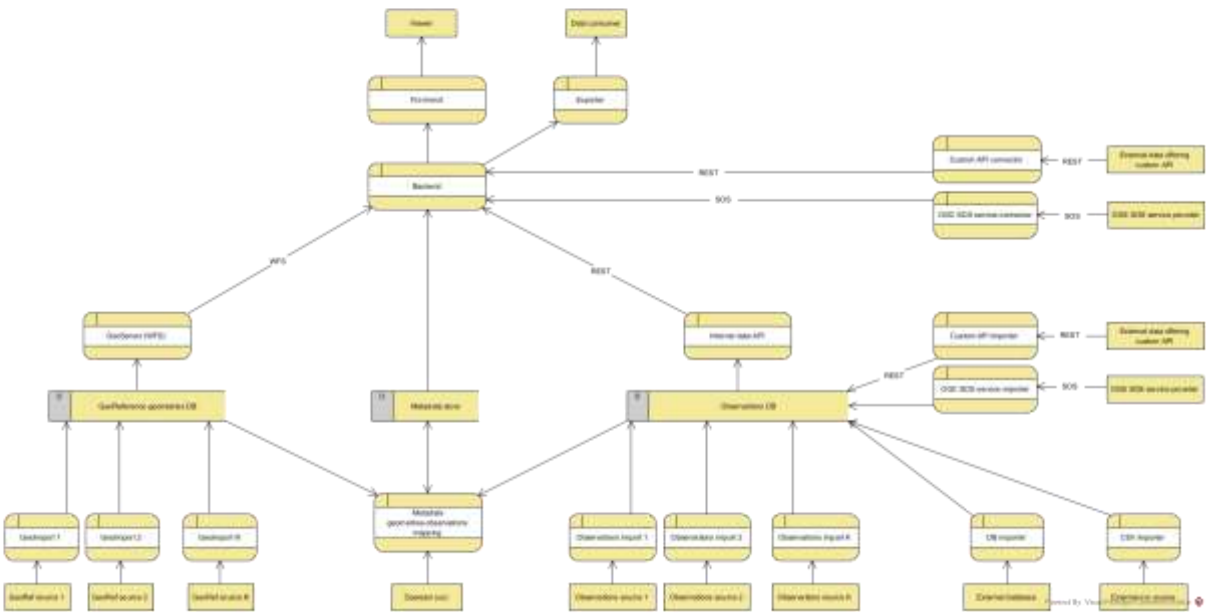
Εικόνα 27: Graph Component Highlighted

Για λόγους πληρότητας ακολουθεί και το διάγραμμα με όλες τις σελίδες και τις διασυνδέσεις τους. Όπως φαίνεται υπάρχουν και διαγράμματα για το πως θα φαινόταν μελλοντικά μερικές σελίδες που συνδέονται με την διαχείριση των δεδομένων από ένα Admin Panel. Αυτές οι σελίδες δεν έχουν υλοποιηθεί


sign out
sign in
sign up
success
Title
Sign up
Sign up
Cancel
Login
(wired) admin sign in
1 : Landing
2 : Main app screen
3 : existing user sign in
4 : new user sign up
6 : define data sources
8 : define georef sources
7 : Assign georef to data

2.3 Business process: ροές δεδομένων

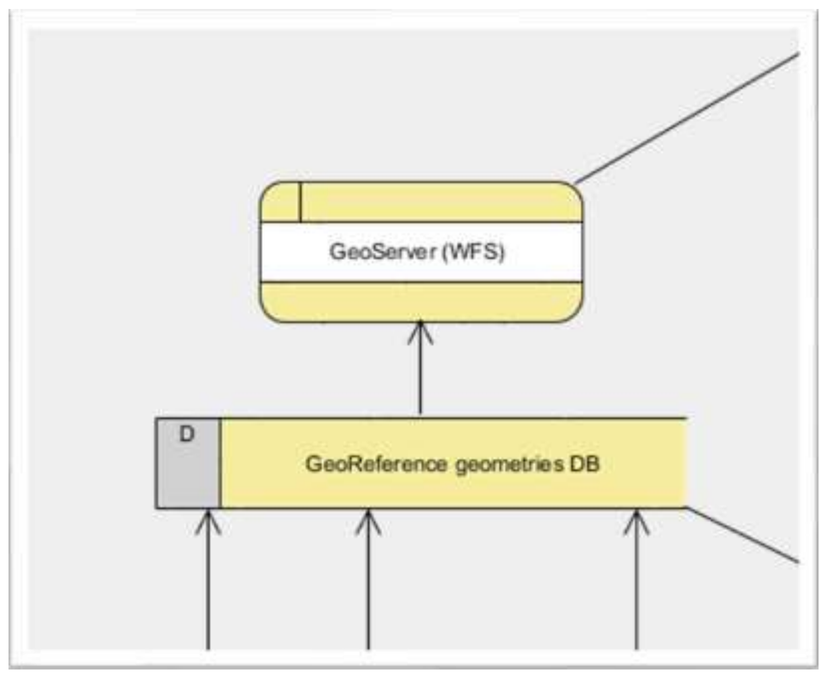
Η γενική περίπτωση απαιτεί το σύστημα να μπορεί να δεχθεί δεδομένα με διάφορα πρωτόκολλα και από διάφορες πηγές. Αυτά θα μπορούσαν είτε να αποθηκευτούν σε εσωτερική βάση δεδομένων είτε σε εξωτερική ή ακόμα και με άλλους μηχανισμούς. Το σύστημα φτιάχτηκε με τρόπο που αυτό θα είναι εύκολο να γίνει σε ένα δεύτερο κύκλο εργασιών αλλά υλοποιήθηκε η δυνατότητα να επικοινωνεί με REST API πρώτα ως ελάχιστη λειτουργική ικανότητα.



Εικόνα 28: Big Picture of the System

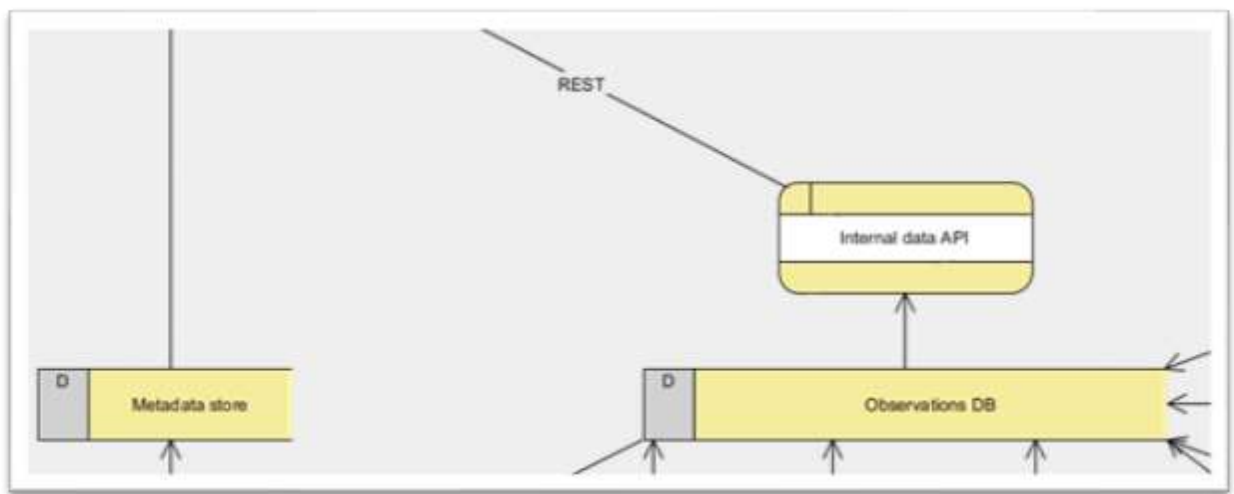
Στο σχήμα φαίνονται τα διάφορα δομικά στοιχεία τα οποία θα μπορούσαν να είναι κομμάτι του συστήματος ή να μπορούν να αλληλοεπιδρούν με αυτό. Σε γενικές γραμμές μπορούν να χωριστούν σε 2 ομάδες, εσωτερικά συστήματα και εξωτερικά. Τα εξωτερικά αποτελούνται κυρίως από συστήματα εισαγωγής δεδομένων (importers) που θα χρησιμοποιούνται για να βάλουν δεδομένα κάθε λογής στο σύστημα. Αυτή η εργασία δεν θα εστίασε σε αυτούς και θα χρησιμοποίησε είτε έτοιμα προγράμματα είτε μικρά προγράμματα γραμμένα σε *python για την εισαγωγή των δεδομένων.*

Εστιάζοντας λοιπόν στα εσωτερικά συστήματα παρατηρούμε ότι μπορούν να σπάσουν σε 4 κομμάτια, το Map Layer, το Data Layer, το Backend Layer και το Frontend Layer.



Εικόνα 29: Map Layer

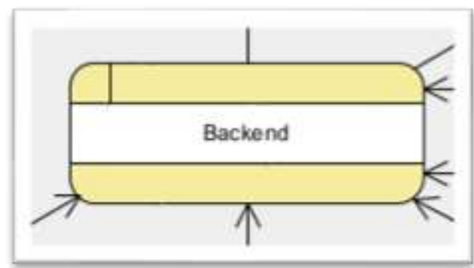
Το Map Layer είναι το API αυτό που θα χρησιμοποιηθεί για τη δημοσίευση δεδομένων χαρτών. Τα δεδομένα τα ίδια θα ζουν σε μια βάση κατάλληλη για την αποθήκευση τέτοιων δεδομένων. Το λογισμικό Geoserver που είναι και το σύστημα που προσφέρει τα δεδομένα μέσω του πρότυπου πρωτοκόλλου υπηρεσιών web WFS, στο οποίο θα αναφερθούμε παρακάτω) είναι για μια μονολιθική εφαρμογή που έχει όλες τις υποδομές που μπορεί κάποιος να χρειαστεί για να προσφέρει χάρτες. Φυσικά οι ικανότητες είναι πολύ περισσότερες από αυτές που θα χρησιμοποιήθηκαν γιατί μπορεί να χρησιμοποιήσει όλα ή σχεδόν όλα τα πρότυπα γεω-δεδομένων.



Εικόνα 30: Data Layer

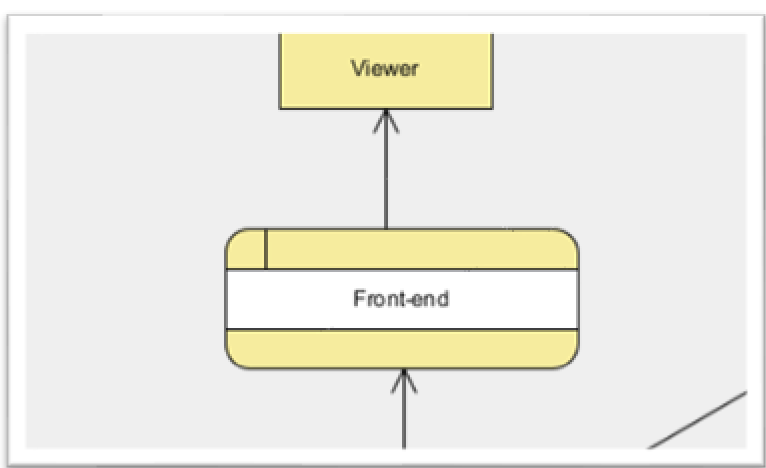
Το Data Layer έχει την αρμοδιότητα να διαχειρίζεται δεδομένα χρονοσειρών και meta-δεδομένα που θα συνδέουν τα διάφορα συστήματα μαζί. Το Internal Data API μαζί με το Observation DB είναι υπεύθυνα για τη διαχείριση των δεδομένα που αποθηκεύονται εσωτερικά στο σύστημα. Επειδή το σύστημα πρέπει να είναι αυθαίρετο ως προς τον τρόπο αποθήκευσης των δεδομένων, το εσωτερικό DATA API συνδέεται με τη βάση και προσφέρει τα δεδομένα σε μια μορφή κατάλληλη προς κατανάλωση. Το metadata store θα συνδέεται

κατευθείαν με το Backend Layer και θα προσφέρει τις πληροφορίες για το πως τα δεδομένα χαρτών συνδέονται με τις χρονοσειρές και το πως θα διαρθρωθεί το Fronted Layer. Τα meta-δεδομένα θα είναι τα δεδομένα που επιστρέφονται από το Search Component για να φανούν στον χάρτη και να γεμίσουν τα διαγράμματα με πληροφορίες.



Εικόνα 31: Backend Layer

Το Backend Layer είναι υπεύθυνο για την διαχείριση και την διασύνδεση όλων των προηγουμένων. Όλες οι λειτουργίες που αναφέρθηκαν νωρίτερα όπως η αναζήτηση, η δυνατότητα εγγραφής και γενικά η διαχείριση του GUI γίνεται από το Backend. Είναι το μόνο που συνδέετε αυστηρά με το Fronted Layer, στην τελική υλοποίηση μάλιστα προσφέρει και τα αρχεία στα οποία υλοποιείται το GUI. Ενώ δίνεται η δυνατότητα τα δεδομένα να αποθηκεύονται αλλού, δίνεται η δυνατότητα ο καθένας να χρησιμοποιήσει το δικό του Geoserver και τα δικά του meta-δεδομένα, το Backend είναι η ραχοκοκαλιά του συστήματος που τα συνδέει όλα μέσα από πίνακες που έχουν τις πληροφορίες των παρόχων και το τι προσφέρουν.



Εικόνα 32: Fronted Layer

Το τελευταίο κομμάτι του συστήματος και ίσως το ποιο σημαντικό είναι το Fronted Layer. Σε αυτό υλοποιείται όλη η αλληλεπίδραση με τον χρήστη και οι διάφορες δυνατότητες της εφαρμογής. Τα διαγράμματα που παρουσιάστηκαν στην προηγούμενη ενότητα θα υλοποιηθούν εδώ. Από άποψη λειτουργικότητας είναι το μεγαλύτερο κομμάτι του συστήματος καθώς απαιτείται να συνδέσει πολλές τεχνολογίες μαζί και να το κάνει με έναν τρόπο που είναι ευπαρουσίαστος και λειτουργικός.

Chapter 3: WFS web service standard για διανυσματικά χωρικά δεδομένα

Το Web Feature Service είναι ένα API το οποίο εξειδικεύεται στη παροχή, δημιουργία και αλλαγή των γεωγραφικών δεδομένων. Όταν πρώτο εμφανίστηκε άλλαξε το πως χρησιμοποιούνταν οι χάρτες καθώς μέχρι τότε η συνήθης πρακτική ήταν να στέλνονται τα δεδομένα με πρωτόκολλα ανταλλαγής αρχείων (FTP). Η πιο πρόσφατη έκδοση του 2.0.2 ορίστηκε από το Open Geospatial Consortium (OGC) το 2014 και αποτελεί μέχρι και σήμερα ένα από τους καλύτερους αν όχι τον καλύτερο τρόπο να διαχειριστεί κάποιος δεδομένα χρηστών.

Το διεθνές Standard ορίζει διαδικασίες εύρεσης, αναζήτησης, κλειδώματος και διαδικασίες συναλλαγής πληροφοριών. Επίσης ορίζει τρόπους να αποθηκευτούν και να παραμετροποιηθούν αναζητήσεις (queries). Συγκεκριμένα το standard ορίζει έντεκα δυνατότητες που πρέπει να έχει έναν σύστημα για να είναι συμβατό. [16]

Εικόνα 33: Το πρότυπο WFS

Το πρότυπο WFS περιλαμβάνει μια σειρά από λειτουργίες όπως: η λήψη πληροφοριών για τα δεδομένα και η επεξεργασία των δεδομένων (π.χ. ενημέρωση, προσθήκη, διαγραφή κ.ά.). Με το αίτημα DescribeFeatureType, ο χρήστης ζητάει από τον εξυπηρετητή να περιγράψει τα χαρακτηριστικά κάθε γεω-αναφοράς που είναι διαθέσιμη. Με το αίτημα GetFeature, ζητάει από τον εξυπηρετητή να ανακτήσει το σύνολο των χαρτογραφικών δεδομένων από ένα θεματικό επίπεδο ή ένα υποσύνολό τους όπως ορίζεται μέσω ενός ερωτήματος το οποίο αφορά τις ιδιότητες του. Ο εξυπηρετητής ανακτά τα χαρτογραφικά δεδομένα από τη βάση δεδομένων, τα κωδικοποιεί σε μια κατάλληλη μορφή (GML ή άλλη) και τα επιστρέφει. Βάσει του πρότυπου WFS-T (Web Feature Service Transactional) με το αίτημα (Transaction), ο πελάτης ζητά από τον εξυπηρετητή να επέμβει στα δεδομένα με τις διαδικασίες της προσθήκης (Inset), της διόρθωσης (Update) και της διαγραφής (Delete). Υπάρχουν επίσης τρις εντολές αναζήτησης που επιτρέπουν το πάρει κάποιος τιμές ή γεω-αναφορές από τις υποδομές αποθήκευσης που υπάρχουν:

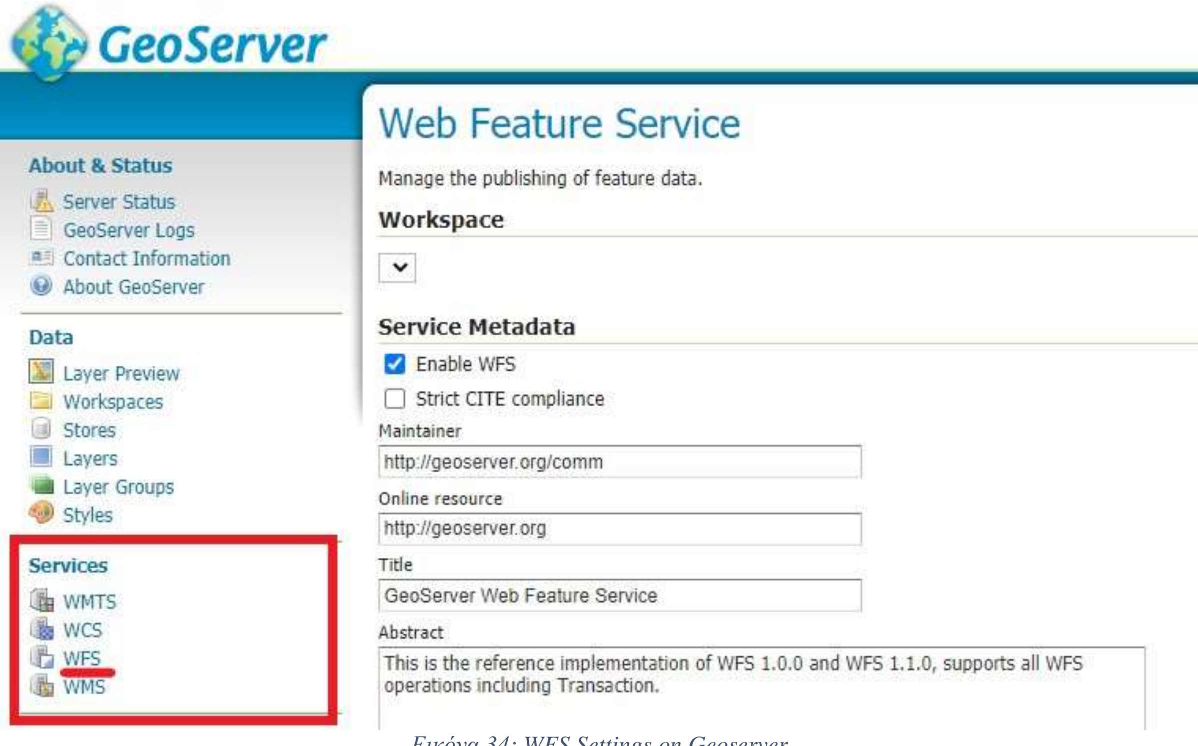
- GetPropertyValue (query operation)
- GetFeature (query operation)
- GetFeatureWithLock (query & locking operation))

Αυτές είναι οι βασικές λειτουργίες που θα χρειαστεί κάποια εφαρμογή για να συνδεθεί σε ένα σύστημα μέσω του πρωτοκόλλου και να πάρει δεδομένα. Το standard ορίζει

πέρα από αυτές και άλλες υπηρεσίες που χρησιμοποιούνται για να μπουν δεδομένα στο σύστημα που υλοποιούν τα transaction που αναφέρθηκαν παραπάνω:

- LockFeature (locking operation)
- Transaction (transaction operation)
- CreateStoredQuery (stored query operation)
- DropStoredQuery (stored query operation)
- ListStoredQueries (stored query operation)
- DescribeStoredQueries (stored query operation)

Ο χάρτης που θα υπάρχει στο web app που στήνουμε θα πρέπει να σερβίρουμε ένα χάρτη στο Map Component. Αυτό θα γίνει με το WFS πρωτόκολλο και μια υπηρεσία που λέγεται Geoserver, το οποίο σου επιτρέπει να δώσεις τους χάρτες σε χρήστες με διάφορα πρωτόκολλα, ένα από τα οποία είναι και το WFS. [17]



Εικόνα 34: WFS Settings on Geoserver

Τα υπόλοιπα πρωτόκολλα που παρέχει είναι και αυτά εναλλακτικές του WFS και μπορεί το Geoserver να δώσει τα ίδια δεδομένα και με τα τέσσερα πρωτόκολλα. Υπάρχουν επίσης πολλές ρυθμίσεις που εξασφαλίζουν την ασφάλεια του συστήματος και μπορούν να απενεργοποιηθούν ή να ενεργοποιηθούν σχεδόν όλες οι δυνατότητες του. Υποστηρίζονται διάφορα πρωτόκολλα κωδικοποίησης των δεδομένων και της επικοινωνίας

Features

Maximum number of features

1000000

Maximum number of features for preview (Values <= 0 use the maximum number of features)

50

- [ ] Return bounding box with every feature
- [ ] Ignore maximum number of features when calculating hits
- [ ] Activate complex to simple features conversion

Extra SRS codes for WFS capabilities generation

Service Level

- ( ) Basic
- ( ) Transactional
- (•) Complete

GML 2

SRS Style

OGC HTTP URL

- [x] Override GML Attributes

GML 3

SRS Style

OGC Experimental URN

- [ ] Override GML Attributes

GML 3.2

SRS Style

OGC URN

- [ ] Override GML Attributes
- [ ] Override MIME Type

Conformance

- [ ] Encode canonical WFS schema location

Encode response with

- (•) One "featureMembers" element
- ( ) Multiple "featureMember" elements

SHAPE-ZIP output format

- [ ] Use ESRI WKT format for SHAPE-ZIP generated .prj files

Stored Queries

- [x] Allow Global Stored Queries

Εικόνα 35: Geoserver WFS Feature Configurations

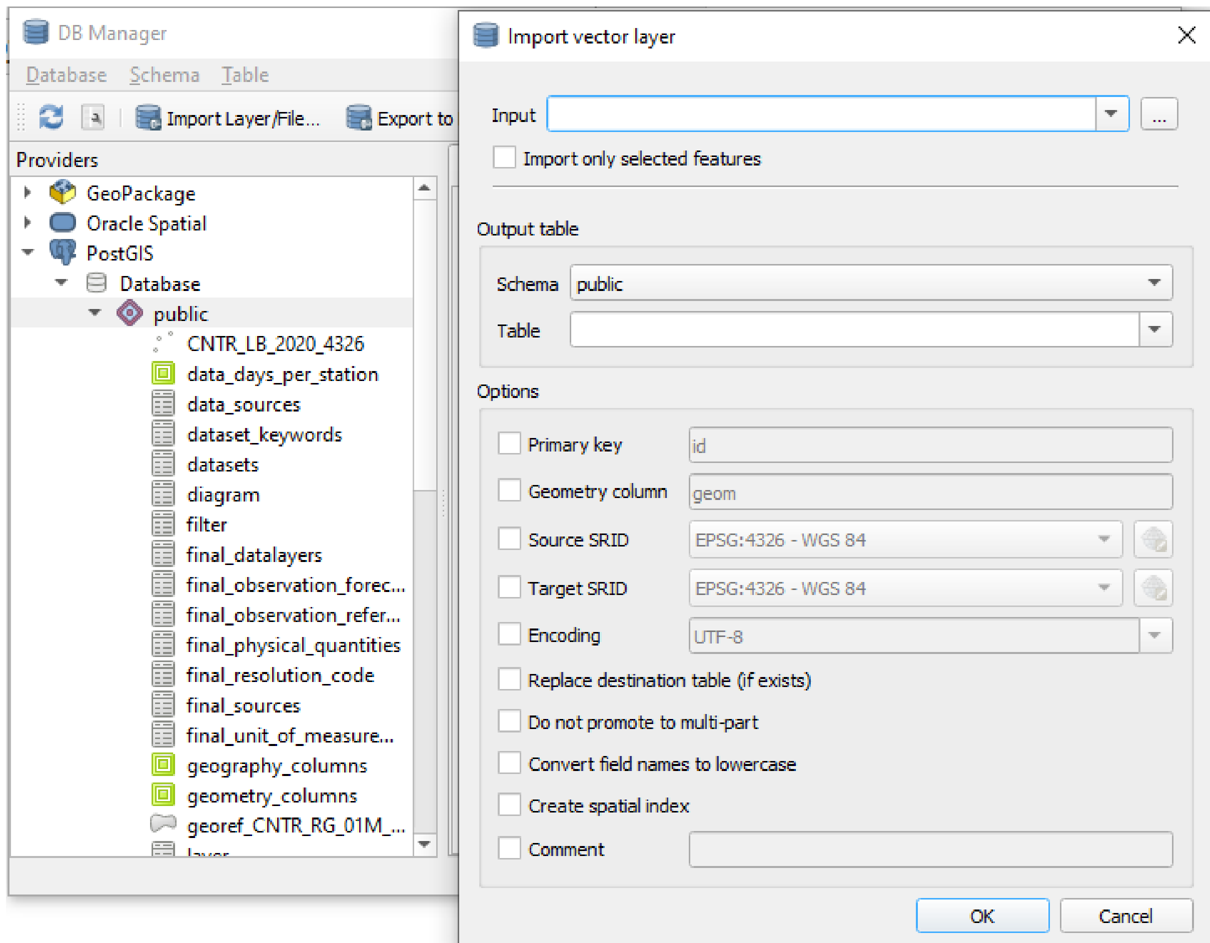
Chapter 4: Σχεδίαση και υλοποίηση της εφαρμογής

4.1 Γενική αρχιτεκτονική

Η αρχιτεκτονική που σχεδιάστηκε χρησιμοποιεί τις τεχνολογίες που κρίθηκαν ότι είναι κατάλληλες για το σύστημα που πρέπει να κατασκευαστεί. Έχουν χρησιμοποιηθεί διαφορετικές τεχνολογίες για τα δεδομένα, για το API Service και για το GUI όπως προστάζει η 3-tier architecture αλλά έχουν χρησιμοποιηθεί και υβριδικές τεχνολογίες που έχουν ως σκοπό την βελτίωση της απόδοσης της εφαρμογής.

4.1.1 Βάση δεδομένων και γεω-αναφορές

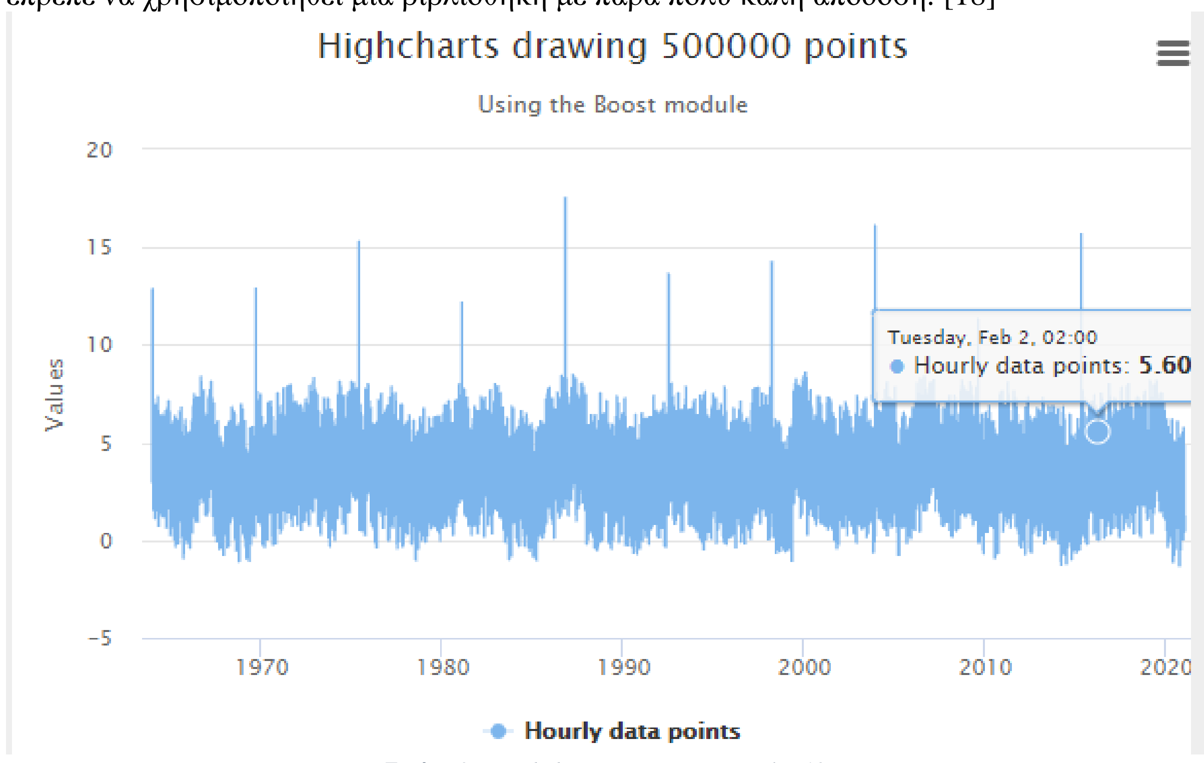
Το πιο βασικό στοιχείο είναι η βάση δεδομένων. Εκεί θα πρέπει να αποθηκεύονται τα meta-δεδομένα και τα δεδομένα που θα προσφέρονται από εμάς. Επιλέχτηκε η PostgresQL γιατί έχει την δυνατότητα μέσω της επέκτασης PostGIS να αποθηκεύσει και τα τους χάρτες που προσφέρονται και τα δεδομένα μας. Έτσι απλοποιείται η εγκατάσταση και δίνονται τα δεδομένα στον χρήστη με τον βέλτιστο τρόπο. Υπήρχε και η δυνατότητα οι χάρτες να αποθηκευτούν με μορφή αρχείου μέσα στο Geoserver. Κάτι τέτοιο όμως θα περιόριζε το μέγεθος της εγκατάστασης του και θα μας ανάγκαζε να πρέπει να συνδεθούμε στο server με FTP και να τα ανεβάζουμε. Αντίθετα τώρα όταν υπάρχει ένας καινούργιος χάρτης να προστεθεί, αυτό γίνεται μέσα από αυτόματα εργαλεία όπως αυτό που δίνει το extension PostGIS ή άλλα καλύτερα. Επιπλέον έχει χρησιμοποιεί το extension της postgres pg_trig που επιτρέπει την καλύτερη αναζήτηση της βασης.



Εικόνα 36: Qgis Database Vector Layer Importer

Από την ανάπτυξη που έγινε νωρίτερα γίνεται προφανές ότι χρησιμοποιήθηκε το λογισμικό Geoserver για να δώσει τα διανυσματικά δεδομένα χωρικής αναφοράς των χρονοσειρών μέσω του WFS πρωτοκόλλου. Το Geoserver υποστηρίζει πολλά επιπλέον πρωτόκολλα επικοινωνίας όπως θα ήταν το WMS. Αυτό είναι επίσης κατάλληλο για αναπαράσταση χαρτών αλλά αντί να δίνει τα δεδομένα όπως είναι αποθηκευμένα στην βάση, ζωγραφίζει τις εικόνες και τις δίνει πίσω, δηλαδή χειρίζεται χωρικά δεδομένα σε μορφή raster (εικόνων). Επειδή στο GUI χρησιμοποιήθηκε η βιβλιοθήκη MapboxGL για την αναπαράσταση των δεδομένων πάνω σε χάρτη προτιμήθηκε το WFS γιατί δίνει την δυνατότητα να ρυθμιστεί η εμφάνιση του χάρτη από την βιβλιοθήκη και όχι από το Geoserver. Αυτή η πρακτική είναι καλύτερη καθώς το σύστημα έχει αποθηκευμένα τα δεδομένα για το πως θα πρέπει να φαίνεται ένας χάρτης σε δική του αποθηκευτική δομή και αν χρησιμοποιούσαμε WMS θα έπρεπε κάπως αυτά να συγχρονίζονται με το Geoserver, κάτι που θα περιέπλεκε τις διαδικασίες συντήρησης και θα μας υποχρέωνε να συνδεόμαστε στο Geoserver για να κάνουμε αλλαγές στην εμφάνιση των χαρτών.

Για την αναπαράσταση των χρονοσειρών θα γίνει με τη χρήση της βιβλιοθήκης Highcharts. Από όλες τις εναλλακτικές αποδείχτηκε η καλύτερη για τα APIs και για την ταχύτητα της. Επειδή στο σύστημα μας μπορεί ο χρήστης να έχει απεριόριστο αριθμό δεδομένων φορτωμένα, κάτι που εύκολα μπορεί να καταντήσει μη λειτουργικό έναν browser έπρεπε να χρησιμοποιηθεί μία βιβλιοθήκη με πάρα πολύ καλή απόδοση. [18]



Εικόνα 37: Highcharts Big Data Example [19]

Τα δεδομένα πρέπει να φτάσουν με έναν τρόπο στον χρήστη. Τα γεω-δεδομένα φτάνουν στον χρήστη κατευθείαν από το Geoserver. Τα υπόλοιπα φτάνουν με την χρήση ενός REST API.

4.1.2 API Service

Χρησιμοποιούνται web τεχνολογίες και στο Backend γιατί απλοποιούν πολύ την ανάπτυξη και δίνουν κάποιες δυνατότης που επιτρέπονται μόνο όταν υπάρχει η ίδια γλώσσα προγραμματισμού στο server και στον client. Το REST Framework που χρησιμοποιήθηκε είναι το ExpressJS. Πρόκειται για ένα μινιμαλιστικό framework γραμμένο με javascript και χρησιμοποιείται κατά κόρων για API μικρού ή και μεγαλύτερου βεληνεκούς. Πολύ συχνά συνδέονται πάνω του και διάφορες βιβλιοθήκες που επεκτείνουν τις ικανότητες του. [20]

```
import logger from 'morgan';
import cookieParser from 'cookie-parser';
import bodyParser from 'body-parser';
import cors from 'cors';
import session from 'express-session';
import sessionFileStore from 'session-file-store';
```

Εικόνα 38: API main libraries

Η βιβλιοθήκη “morgan” είναι ένας μεσάζοντας (middleware) που κάνει log όλες τις εντολές που έρχονται στο server. Ρυθμίστηκε να στέλνει το αποτέλεσμα στο terminal. Οι βιβλιοθήκες “cokieParser” και “bodyParser” είναι επίσης middleware και διαβάζουν το Request και δίνουν πίσω τα δεδομένα που πήραν σε κατάλληλη μορφή προς χρήση (το cookie και το σώμα αντίστοιχα). Οι βιβλιοθήκη “session” και “sessionFile” δουλεύουν μαζί για να δώσουν ένα μηχανισμό ώστε να μπορεί το server να χρησιμοποιήσει sessions. Υπάρχει μια ακόμα βιβλιοθήκη που λέγεται Sapper αλλά θα αναλυθεί ξεχωριστά καθώς η δουλειά της είναι πολύ πιο περίπλοκη.

```
const app = express();

app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
// Session Handling
const FileStore = new sessionFileStore(session);
app.use(
    session({ ···
    })
);

// Info: Setup Libraries
app.use(
    logger('dev', {})
);
app.use(cors());

app.use(cookieParser());
```

Εικόνα 39: Express Library Setup

Το επόμενο κομμάτι του συστήματος είναι τα Routes που θα σερβίρουν τα δεδομένα. Για την καλύτερη οργάνωση του κώδικα χωρίστηκε το κάθε υποσύστημα στο δικό του sub-

router και συνδέθηκαν πάνω στο κεντρικό (η εντολή app.use() συνδέει κάτι πάνω στο router). Υπάρχουν 5 υποσυστήματα, το πρώτο λέγεται *layer* και είναι υπεύθυνο για τα metadata των χαρτών. Το δεύτερο λέγεται diagrams και υπεύθυνο και τα metadata που χρειάζονται για το διάγραμμα και το data υποσύστημα δίνει στον χρήστη τα δεδομένα που έχουμε αποθηκεύσει στην βάση. Τα συστήματα auth και search υλοποιούν τις συνώνυμες λειτουργίες για το GUI.

```
// Info: Routes
const base = API_PATH ?? '/api';
app.use(`${base}/layers`, layerRouter);
app.use(`${base}/diagrams`, diagramRoute);
app.use(`${base}/data`, dataRoute);
app.use(`${base}/auth`, auth);
app.use(`${base}/search`, search);
```

Εικόνα 40: Subsystem Routers

Επόμενο κομμάτι του συστήματος είναι η χρήση της βιβλιοθήκης Sapper. Η βιβλιοθήκη αυτή συνδέεται άμεσα με την βιβλιοθήκη που χρησιμοποιήθηκε για το GUI και λέγεται Svelte. Η δουλειά της είναι να σερβίρει τα στατικά αρχεία και όταν κανένα από τα άλλα routes μέχρι τώρα δεν χειρίστηκε το Request να το χειρίζεται αυτό.

```
// Info: Sapper
app.use(
    compression({ threshold: 0 }),
    sirv('static', { dev }),
    sapper.middleware({
        session: (req, res) => {
            return {
                token: req.session.token,
            };
        },
    })
);
```

Εικόνα 41: Sapper Configuration

Αποτελεί μια πλήρη framework για web apps με την Svelte για τον ορισμό των GUI. Υπάρχουν τρία μαγικά αρχεία, το “template.html” που αποτελεί την βάσει όλων των σελίδων, το “_layout.svelte” που αποτελεί το πρώτο Svelte Component (παρακάτω θα εξηγηθεί η έννοια αυτή) που τρέχει και είναι κοινό για όλες τις σελίδες και το “_error.svelte” που αποτελεί το Component που τρέχει όταν υπάρχει κάποιο πρόβλημα (404 page).

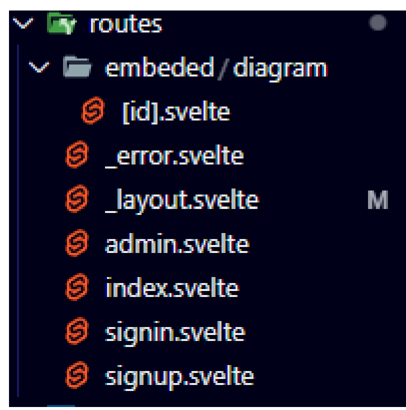
```html
<!doctype html>
<html lang="en">
<head>…
</head>
<body >
  <!-- The application will be rendered inside this element,
       because `src/client.js` references it -->
  <div id="sapper">%sapper.html%</div>
</body>
</html>
```

Εικόνα 42: template.html

```svelte
<svelte:head>
    <title>{status}</title>
</svelte:head>

<h1>{status}</h1>

<p>{error.message}</p>

{#if dev && error.stack}
    <pre>{error.stack}</pre>
{/if}
```


Εικόνα 43: _error.svelte

Έχει το δικό του router και έτσι όλα τα αρχεία που ανήκουν στον φάκελο “routes” και δεν ξεκινάνε από underscore αποτελούν και ένα route της εφαρμογής. Οι υπό-φάκελοι αποτελούν sub routes και υποστηρίζει επίσης query parameters όταν ένα route έχει το όνομα του σε brackets. Τέλος χρησιμοποιώντας το life cycle σύστημα της Svelte υλοποιεί server side rendering. Τέλος παρέχει έτοιμο service worker για request caching και δίνει την δυνατότητα να εγκατασταθεί η εφαρμογή στα κινητά που επισκέπτονται το site. Πρόκειται για ένα πλήρες application framework που σου επιτρέπει να κάνεις ότι μπορεί μια web εφαρμογή.



Εικόνα 44:GUI routes

4.1.3 Web app – GUI

Για την ανάπτυξη του client χρησιμοποιήθηκε το Svelte. Αυτή η τεχνολογία μας επιτρέπει να γράφουμε ιστοσελίδες με τρόπο που θυμίζει την κατασκευή εφαρμογών με την χρήση όμως standard web technologies. Μπορείς να γράφεις κομμάτια του GUI σε αυτόνομα components και αυτά να τα συνδέεις μαζί για να πάρεις το τελικό αποτέλεσμα. Από μόνη της η Svelte δημιουργεί ένα single page application και την ικανότητα να έχουμε πολλές σελίδες μας την δίνει το Sapper. Καλό είναι να πω ότι και τις δύο βιβλιοθήκες τις φτιάχνει η ίδια ομάδα και έχουν σχεδιαστεί για να δουλεύουν σαν πακέτο. Μαζί αυτές οι δύο αποτελούν έναν υβριδικό σύστημα όπου μέρος του GUI τρέχει στο client και μέρος στο server.

Όταν κάποιος θέλει να φτιάξει μία σελίδα με Svelte το πρώτο πράγμα που κάνει είναι να στήσει ένα καινούργιο npm project. Ευτυχώς οι δημιουργοί της δίνουν ένα έτοιμο template το οποίο μπορεί να πάρει κάποιος και να ξεκινήσει να δουλεύει. Περιέχει την Svelte ένα απλό file server development και τον bundler που θα συνδέσει όλα τα κομμάτια. Όταν κάποιος θέλει να στήσει το project του και με Sapper στο server η διαδικασία είναι η ίδια αλλά αλλάζει το template. Στο σύστημα μας έχουμε χρησιμοποιήσει το template του Sapper λίγα πράγματα αλλάζουν.

```
npx degit sveltejs/template my-svelte-project
# or download and extract this .zip file
cd my-svelte-project
# to use TypeScript run:
# node scripts/setupTypeScript.js

npm install
npm run dev
```

Εικόνα 45: Svelte install instructions from their website

Αγνοώντας προς το παρόν τον bundler ας εστιάσουμε στη Svelte. Το project περιέχει ένα "main.js" file που αποτελεί το entry point της εφαρμογής. Το μόνο που κάνει είναι να ξεκινάει το App Component που ζει στο αρχείο App.svelte, να το συνδέει με το *document.body* και είναι το βασικό component το συστήματος. Στο αντίστοιχο Sapper Project θα λεγόταν "client.js" αλλά θα έκανε την ίδια δουλειά.

```
import App from './App.svelte';

const app = new App({
  target: document.body,
  props: {
    name: 'world'
  }
});

export default app;
```

Εικόνα 46:Template Svelte Project

Κάθε αρχείο svelte αποτελείτε από τρία βασικά μέρη. Το script, το style, και το UI. Τα δύο πρώτα ζουν σε special tags με το ίδιο όνομα ενώ, ότι είναι έξω από αυτά τα δύο tags αποτελεί το UI. Μέσα στο script γράφουμε κανονική javascript η οποία τρέχει με το που το Component συνδεθεί πάνω στο DOM.

Μετά από αυτό προστίθεται το UI σαν υπο-δέντρο του αφού πρώτα έχουν υπολογιστή μέσα από interpolation τα δυναμικά του κομμάτια, για παράδειγμα μια μεταβλητή που ορίστηκε στο script μπορεί να αναπαρασταθεί στο UI αν την βάλουμε μέσα σε brackets. Τέλος το style section αποτελείται από css classes που ορίζονται τοπικά και επηρεάζουν μόνο το συγκεκριμένο component.

```
<script>
  export let name;
</script>

<main>
  <h1>Hello {name}!</h1>
  <p>Visit the <a href="https://svelte.dev/tutorial">Svelte tutorial
</main>

<style>
  main {
    text-align: center;
    padding: 1em;
    max-width: 240px;
    margin: 0 auto;
  }

  h1 {
    color: #ff3e00;
    text-transform: uppercase;
    font-size: 4em;
    font-weight: 100;
  }
</style>
```

Εικόνα 47: Default Svelte App

Αυτά είναι τα "βασικά" που κάνει σχεδόν κάθε template system, όμως η Svelte είναι μία πλήρης αρχιτεκτονική. Για αυτό και δίνει πολλούς παραπάνω μηχανισμούς για να κάνει δυναμικό το UI. Ο πρώτος από αυτούς είναι ένας τρόπος να φτιάχνουμε conditional UI. Ας πάρουμε το αρχείο "_layout.svelte" ως παράδειγμα του δικού μας συστήματος. Πρώτο πράγμα που φαίνεται είναι ότι στο script tag έχουν γίνει import διάφορα αρχεία που κάποια από τα οποία είναι Components και άλλα είναι κομμάτια της βιβλιοθήκης. Το Sapper μας δίνει ένα σύνολο από πληροφορίες μέσα από τον μηχανισμό των stores όπως το path στο οποίο βρισκόμαστε. Με αυτό θέλω να υλοποιήσω την εξής λογική: Στα routes "**/embed/***" δεν θέλω να κάνει τίποτα γιατί από εκεί θέλω να σερβίρω τα iframes και άρα δεν χρειάζονται global layout. Σε όλα τα άλλα routes θέλω να βάλεις το NAV Component πάνω πάνω και από κάτω ότι χρειάζεται η σελίδα. Το πρώτο σκέλος αυτού που θέλουμε να κάνουμε υλοποιείται με το if..else statement. Η ιδέα είναι απλή, αν το condition μετά το "**#if**" είναι αληθής τότε γίνεται render εκείνο το κομμάτι UI αλλιώς γίνεται το κομμάτι "**:else**". Η svelte υποστηρίζει και "**:else if**" blocks με την προφανή λειτουργία.

Φαίνεται λοιπόν ότι στο else block πέρα από το Modal και το SvelteToast Components που δεν παράγουν UI (ένα component μπορεί να μην έχει ένα από 3 μέρη του) έχουμε πάνω πάνω το Nav Component και μετά ένα slot για να μπει το Component που αντιστοιχεί το route.

```
<script lang="ts">
import Modal from '../components/modal/Modal.svelte';
import { modalState } from '../components/modal/store';

import { SvelteToast } from '@zerodevx/svelte-toast';

import Nav from '../components/Nav.svelte';
import { stores } from '@sapper/app';
import { onMount } from 'svelte';
const { page } = stores();
</script>

{#if $page.path.startsWith('/embeded')}
    <slot />
{:else}
    <Modal show="{$modalState}" />
    <div class="scroll h-screen w-screen px-2 ">
        <Nav />

        <main
            class=" flex flex-col space-y-2 lg:space-y-0 items-center lg:grid  lg:grid-cols-12
            <slot />
        </main>
    </div>
    <SvelteToast options="{{ reversed: true, intro: { y: 0 }, dismissable: false }}" />
{/if}
```

Εικόνα 48: _layout.svelte

Αναφέρθηκαν δύο μηχανισμοί της Svelte, τα stores και τα slots. Τα store είναι ένας μηχανισμός για αποθήκευση πληροφορίας υπό το πρότυπο των subscribers. Το store έχει την πληροφορία και όποιος την θέλει κάνει subscribe για να ακούει για αλλαγές. Αν θέλω να την πάρω μία φορά τότε είτε χρησιμοποιώ την συνάρτηση get() που δίνει η Svelte ή χρησιμοποιώ το "$" πριν από την μεταβλητή μόνο όμως στο κομμάτι του UI.

Τα slot είναι σαν μία άδεια θέση στην οποία μπορεί να μπει όποιο άλλο Component θέλει. Όπως στην html μπορώ να βάλω μέσα σε ένα div κάτι, έτσι στην Svelte δηλώνουμε που θα μπει αυτό που είναι μέσα στα tags. Στο συγκεκριμένο Component Τα slot γεμίζουν από τα Component που αντιστοιχούν στο route που είμαστε. Για παράδειγμα στην αρχική στο slot μπαίνει το index.svelte. Με αυτό τον μηχανισμό κάνει το routing to Sapper. Φυσικά κανείς δεν μας εμποδίζει από το να χρησιμοποιηθεί και με άλλα Components.

```
<Box title="Map" class="h-full lg:col-span-8 min-h-full">
    <PrivateMap />
    <div slot="actions" class="flex">
        <ScreenShot />
        <FullscreenTool />
    </div>
</Box>
```

Εικόνα 49: Box Component

Ένας ακόμα μηχανισμός για να φτιάξουμε UI με Svelte είναι τα each blocks. Στην εφαρμογή έχω μια λίστα από πληροφορίες για διαγράμματα που θέλω να δείξω στον χρήστη. Σε αυτή την λίστα θα μπορούν να επιλέξουν κάποια από αυτά για να φορτωθούν στο

συνολικό διάγραμμα. Εγώ έχω μια μεταβλητή diagrams (στο script block ακριβώς από πάνω) που την φορτώνω με την λίστα από τις πληροφορίες και θέλω να τις δείξω στο Select Component, σε οποιαδήποτε αλλαγή της επιλογής του χρήστη αυτόματα ενημερώνεται το διάγραμμα. Η σύνταξη είναι η εξής ξεκινάω με την λέξη “**#each**” ακολουθεί η μεταβλητή του script block που θέλω να διαβώ και μετά η λέξη “**as**” και το όνομα που θέλουμε να αναθέσουμε στο κάθε element. Εσωτερικά έχουμε πρόσβαση σε αυτή την μεταβλητή και την χρησιμοποιούμε για να φτιάξουμε το UI που αντιστοιχεί σε αυτό που θέλουμε

```
<ul class="list-disc ml-8">
    {#each diagram.extra.series as series}
        <div class="flex items-center">
            <input
                class="h-4 w-4 mr-2"
                bind:group="{selected}"
                type="checkbox"
                on:change="{onSeriesSelectionChange}"
                value="{JSON.stringify({
                    diagram_id: diagram.id,
                    series_id: series.id,
                })}" />
            <p>{series.name}</p>
        </div>
    {/each}
</ul>
```

Εικόνα 50: Each block Example

Στο προηγούμενο παράδειγμα φαίνονται δύο directive πάνω στο input που δεν είναι κανονική html. Με το “bind:” directive μπορούμε να δώσουμε σε μια μεταβλητή την τιμή που έχει ένα input ανά πάσα ώρα και στιγμή, εδώ χρησιμοποιώ το bind group γιατί έχω πολλά inputs και θέλω όλες τις τιμές σε ένα group ή αλλιώς μια λίστα. Αν είχα ένα textfield θα μπορούσα με το directive “bind:value” να δώσω σε μια μεταβλητή στο script block την τιμή που γράφει ο χρήστης ανά πάσα ώρα και στιγμή και η Svelte το εξασφαλίζει αυτό με εσωτερικούς της μηχανισμούς.

Με το “**on:**” directive ορίζω event listeners και του περνάω μια συνάρτηση για να καλέσει όταν γίνει το event. Η Svelte αναλαμβάνει να συνδέσει τον Event Listener όταν συνδεθεί το component με το DOM και να τον αποσυνδέσει αν βγει από αυτό. Η συνιστάμενη μεθοδολογία είναι να ορίζουμε στο script block μεταβλητής και να κάνουμε ανάθεση lambda functions.

```
const onSeriesSelectionChange = async () => {
    await tick();
    let objs = selected.map(x => JSON.parse(x));
    DiagramToSeriesMap.set(objs);
};
```

Εικόνα 51

Φυσικά η Svelte έχει και πολλά πράγματα ακόμα για τα οποία δεν έκανα εκτενή αναφορά είτε γιατί δεν τα χρησιμοποίησα κατά την ανάπτυξη είτε γιατί είναι πράγματα που είναι εκτός των τωρινών μου γνώσεων. Έχει πολλούς μηχανισμούς για animations και

δυναμική CSS, έχει τα props που είναι παράμετροι για που μπορείς να περάσεις στο κάθε Component, μηχανισμούς για ασύγχρονους υπολογισμούς και πολλά άλλα. Ότι και να χρηστεί ένας developer για την ανάπτυξη της εφαρμογής υπάρχει, και φυσικά επείδη βρισκόμαστε σε javascript περιβάλλον μπορούν να χρησιμοποιήσουμε την μεγαλύτερη πηγή βιβλιοθηκών που υπάρχει το npm.

Κάτι τελευταίο που πρέπει να αναφέρω για να εξηγήσω ένα μηχανισμό της Sapper είναι το life cycle ενός Svelte app. Κάθε Component στην Svelte βρίσκεται σε ένα από τα τέσσερα στάδια: beforeUpdate, onMount, afterUpdate, onDestroy.

Εικόνα 52: Svelte Component Life cycle

Το στάδιο beforeUpdate είναι το πρώτο πράγμα που γίνεται με το που δημιουργείται ένα Component. Το Mount είναι το στάδιο αφού το UI έχει συνδεθεί με το DOM και σου εξασφαλίζει ότι το UI έχει γίνει render στο DOM. Το AfterUpdate τρέχει αφού μετά από κάποια update.Την πρώτη φορά που συνδέεται κάποιο Component τρέχουν και οι τρεις στη σειρά, αλλά από εκεί και πέρα στις αλλαγές τρέχει μόνο οι beforeUpdate και η afterUpdate. Τέλος υπάρχει και το onDestroy όπου είναι το στάδιο ακριβώς πριν αποσυνδεθεί ένα Component από το DOM. Η Svelte μας δίνει μηχανισμούς για κάνουμε hook σε αυτά τα event μέσα από συναρτήσεις με το ίδιο όνομα με το event. Καλό είναι να πω πως ο κώδικας στο script τρέχει πριν από όλα αυτά τα event και αν θέλουμε συνδεθούμε σε αυτά, από εκεί το κάνουμε.

Αυτό τον μηχανισμό χρησιμοποιεί η Sapper για να κάνει το SSR. Ουσιαστικά όταν ζητάς από την Sapper ένα route, θα πάει και θα τρέξει τον κώδικα μέσα σε κάθε Component script block και θα κάνει generate όσων τον δυνατόν μεγαλύτερο μέρος του site. Θα πάρει αυτό το αποτέλεσμα που περιέχει html, CSS και javascript και το επιστρέφει στον χρήστη. Ότι ζει μέσα σε ένα event αποθηκεύετε και γυρνάει σαν κώδικας αντί να εκτελεστεί και άρα η εφαρμογή τρέχει το ίδιο είτε με Sapper είτε χωρίς, αλλά με Sapper τρέχει γρηγορότερα.

4.1.4 Web app – Δεδομένα

Με την χρήση των μηχανισμών που δίνει η Svelte μπορούμε να στήσουμε το GUI όπως θέλουμε, το ερώτημα είναι πως χειριζόμαστε τα δεδομένα που παίρνουμε από τα server. Που θα αποθηκευτούν και πως πληροφορίες που φτιάχνει ένα Component θα φτάσουν σε ένα άλλο που μπορεί να μην συνδέονται καθόλου στο GUI. Το framework της Sapper δεν μας δίνει κάποιο δικό του μηχανισμό αλλά μπορούμε να χρησιμοποιήσουμε οποιαδήποτε βιβλιοθήκη για state management που τρέχει σε ένα javascript περιβάλλον θέλουμε. Υπάρχουν πολλές βιβλιοθήκες που θεωρούνται καλές σε αυτά αλλά στην πορεία συνειδητοποίησα ότι η Svelte έχει ένα μηχανισμό που είναι σχεδιασμένος για αυτό, τα stores.

Τα stores λειτουργούν σαν single source of truth από όπου θα ενημερώνονται όλα τα Component για αλλαγές στα δεδομένα και θα υπολογίζονται όλα τα δεδομένα που παράγονται από την αναζήτηση του χρήστη. Φυσικά μπορώ να έχω πολλά stores και μάλιστα μπορώ έχω stores που δημιουργούνται κατευθείαν από συνδυασμό των άλλων. Ας εξηγήσω πως με λιγότερες από 100 γραμμές javascript υλοποιείται το data flow όλης της εφαρμογής.

```
1   import { derived, writable, Readable, Writable } from 'svelte/store';
2   import type { Diagram } from '../api/models/Diagram';
3   import type { DatasetExt, Source } from '../api/models/Source';
4
5   // Info: This stores the names of The sources available to the system and there ids
6   export const sourceNames: Writable<Partial<Source>[]> = writable([]);
7   export const FromDate: Writable<string> = writable(null);
8   export const ToDate: Writable<string> = writable(null);
9
10  // Info: A list of Ids that are features selected on the map
11  export const SelectedIds: Writable<string[]> = writable([]);
12
13  // Info: This will be the in memory representaion of a dataset
14  // Info: so everything else will look here for changes
15  export const PrivateDataset: Writable<DatasetExt[]> = writable([]);
16
17  // Info: When you click on the clear or search again you recalculate this object
18  // Info: that is the same with PrivateDataset but with only the enabled series
19  export const SelectedDatasets: Readable<DatasetExt[]> = derived(
20      [PrivateDataset, SelectedIds],
21 >    ([datasets, seriesIds], set) => {···
36      }
37  );
38
39  // Info: List of Selected Series id and the the id of their diagram
    Dimitris Karagiannis, a month ago | 1 author (Dimitris Karagiannis)
40  export interface DiagramToSeries {
41      diagram_id: number;
42      series_id: number;
43  }
44  export const DiagramToSeriesMap: Writable<DiagramToSeries[]> = writable([]);
45
46  export const DiagramToShow: Readable<Diagram> = derived(
47      [SelectedDatasets, DiagramToSeriesMap, FromDate, ToDate],
48 >    ([datasets, diagramToSeriesMap, fromDate, toDate], set) => {···
91      }
92  );
93
94  // Info: I have to store them so on new Search I can clean them up;
95  export const LayerIds = writable([]);
96  export const SourceIds = writable([]);
97
```

Εικόνα 53: stores.ts

Πρώτον υπάρχουν δύο τύποι store, Readable, Writable με μόνη διαφορά είναι το αν μπορείς να αλλάξεις την τιμή τους με εντολή ή ενημερώνεται αυτόματα. Για την δημιουργία ενός store χρησιμοποιώ τις συναρτήσεις writable() που δημιουργεί ένα Writable και derived() που παράγει ένα Readable.

Τα πρώτα τρία stores του αρχείου “stores.ts” παράγονται από το Search Component της Svelte και έχουν μέσα τα available sources και το timeframe που έχει επιλέξει ο χρήστης, αν το έχει κάνει. Ο λόγος είναι ότι το Select Component έχει ένα drop down με τα available sources και χρειάζεται την πληροφορία που παράγει το Search. Τα “FromDate” “ToDate” stores χρησιμοποιούνται από το Graph Component για να κάνει τα σωστά Request στο Server για τα δεδομένα.

Το “SelectedIds” store έχει την πληροφορία των περιοχών που έχει επιλέξει ο χρήστης. Όταν ο χρήστης κάνει click πάνω στο χάρτη, ένα popup εμφανίζεται και του επιτρέπει να επιλέξει είτε να μην επιλέξει μια περιοχή. Αυτό η διαδικασία μεταφράζεται σε μία λίστα με επιλεγμένα location ids.

Το “PrivateDataset” περιέχει τα διαθέσιμα δεδομένα που μας γύρισε το server όταν κάναμε αναζήτηση. Όλα τα arguments από την φόρμα μαζεύονται στο Search Component και δημιουργούν ένα Request, τα δεδομένα που γυρνάει το server αποθηκεύονται εδώ.

Ακολουθεί το “SelectedDatasets” που είναι και το πρώτο derived store. Παίρνει σαν argument μία λίστα από stores στα οποία θέλει να ακούει για αλλαγές και μια συνάρτηση που θα καλέσει όταν γίνει μία αλλαγή. Αυτή η συνάρτηση έχει δύο arguments, πρώτα μια λίστα με τα δεδομένα όλων των stores και μία συνάρτηση που λέγεται set και θα την χρησιμοποιήσει για να θέσει τα δεδομένα. Η συγκεκριμένη συνάρτηση για παράδειγμα παίρνει τα διαθέσιμα δεδομένα(PrivateDataSet) και μια λίστα από επιλεγμένα σημεία στον χάρτη και αφαιρεί από την λίστα όλα τις σειρές που δεν ανήκουν σε επιλεγμένες τοποθεσίες.

```
17  // Info: When you click on the clear or search again you recalculate this object
18  // Info: that is the same with PrivateDataset but with only the enabled series
19  export const SelectedDatasets: Readable<DatasetExt[]> = derived(
20      [PrivateDataset, SelectedIds],
21      ([datasets, seriesIds], set) => {
22          let results = [];
23
24          datasets.forEach(dset => {
25              // Copy Dataset
26              let nDataset: DatasetExt = JSON.parse(JSON.stringify(dset));
27
28              // Filter series By their id
29              nDataset.diagrams.forEach(diagram => {
30                  diagram.extra.series = diagram.extra.series.filter(s => seriesIds.includes(s.id));
31              });
32
33              results.push(nDataset);
34          });
35          set(results);
36      }
37  );
```

Εικόνα 54: SelctedDatasets creation

Αυτό το store το καταναλώνει το Select Compoent για να φτιάξει το GUI. Σε παράδειγμα νωρίτερα είδαμε μια πως κάνουμε each iteration στην Svelte για να δημιουργήσουμε GUI, η λίστα που προσπεράσαμε είναι αυτή που φτιάχνετε εδώ και μάλιστα είναι διπλό iteration παρόλο που δείξαμε μόνο το εσωτερικό για το παράδειγμα.

Οι τιμές που συλλέγονταν σε αυτό το κώδικα με το directive “bind:group” αποθηκεύονται στο επόμενο store, το “DiagramToSeriesMap”. Περιέχει τα ids των χρονοσειρών που έχει επιλεγεί από την λίστα και σε ποιο διάγραμμα ανήκουν.

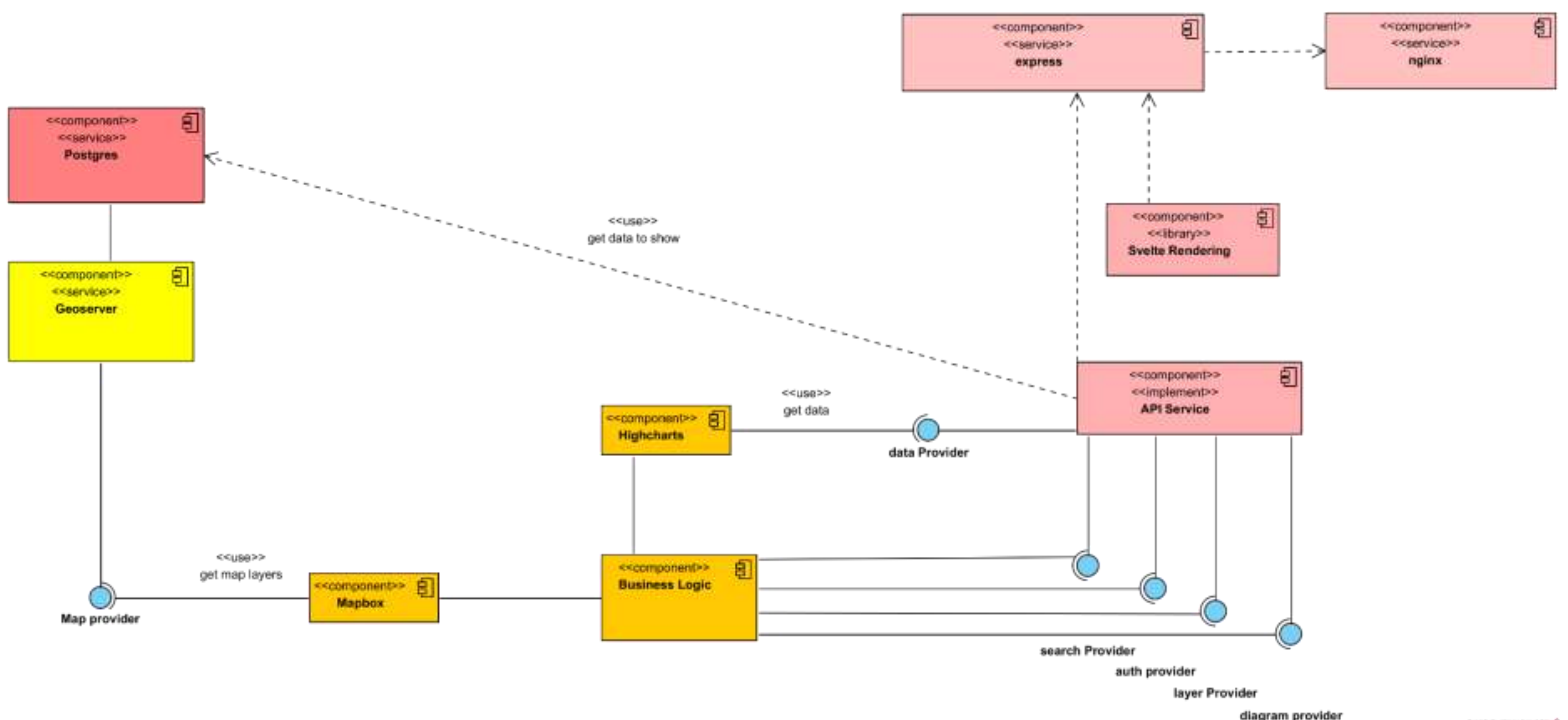
Το “DiagramToShow” είναι και αυτό derive store που παίρνει σαν argument την λίστα με τις επιλεγμένες χρονοσειρές (τα ids τους) και τα επιλεγμένα δεδομένα και τις επιλεγμένες ημερομηνίες (από εώς). Παράγει ένα “Diagram” object που έχει όλες τις πληροφορίες για να ζωγραφιστή ένα διάγραμμα για να καταναλωθεί από το αντίστοιχο Component.

Τα τελευταία store δεν έχουν δεδομένα που συνδέονται με τα άλλα είναι βοηθητικά. Όταν ανανεώνεται ο χάρτης λόγο καινούργια αναζήτησης ή επειδή άλλαξαν τα fromDate,

toDate πρέπει να έχω τα ids των δομών που έχω φτιάξει μέσα στην βιβλιοθήκη. Αυτά τα stores έχουν τα ids αυτών των δομών για να μπορώ εύκολα να καθαρίζω τον χάρτη.

Αυτή όλη ο ροή των δεδομένων στην εφαρμογή, τα διάφορα Component παράγουν δεδομένα τα οποία άλλα Component καταναλώνουν για να παράγουν τα δικά τους. Στο δικό σύστημα όλα ξεκινάνε με το Search Component το οποίο παρέχει τις αρχικές πληροφορίες. Μετά από την αλληλεπίδραση του χρήστη τα δεδομένα καταλήγουν στο διάγραμμα.

4.2 Διαγράμματα UML (APIs, Sequence, …)



Complete Architecture Diagram

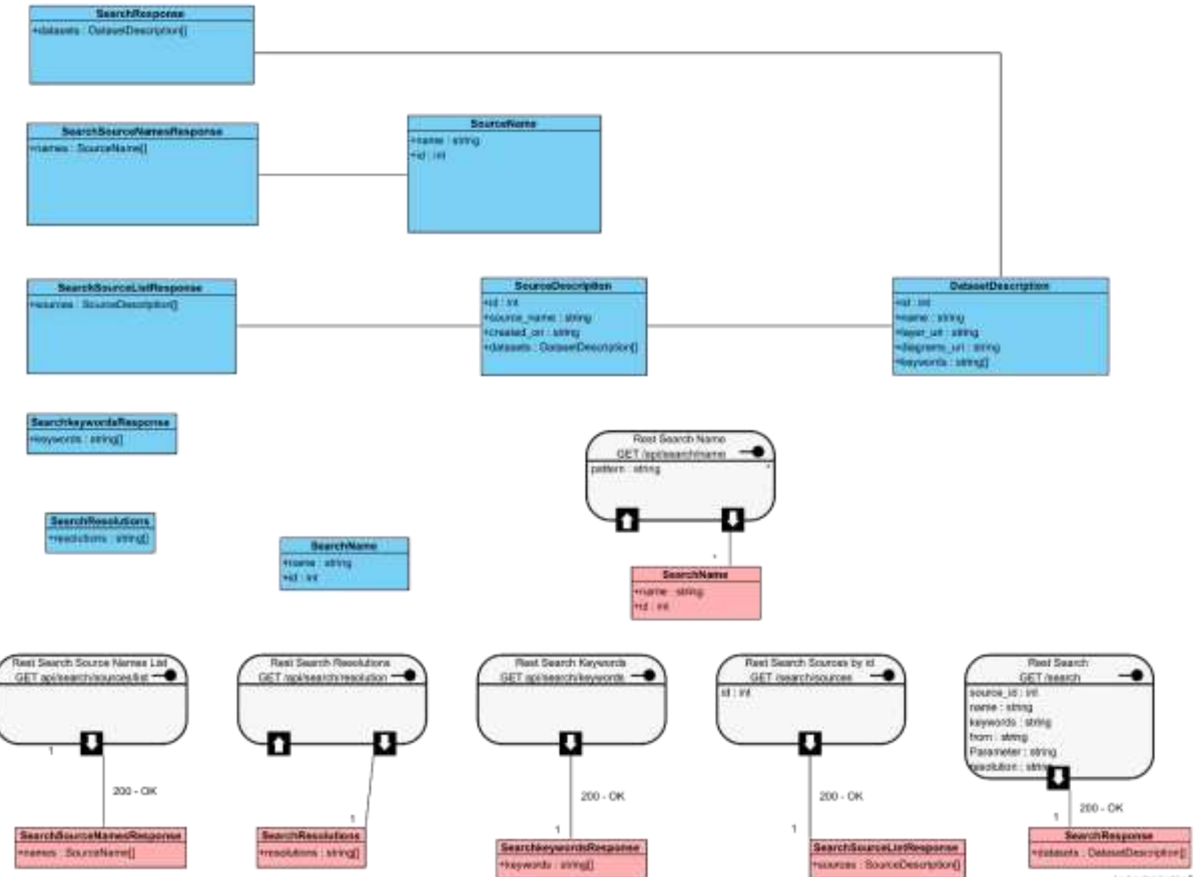
Έχει ήδη αναπτυχθεί στα προηγούμενα κεφάλαια πως υλοποιήθηκε η αρχιτεκτονική από άποψη τεχνολογιών που χρησιμοποιήθηκαν και το πως αυτές αλληλοεπιδρούν μεταξύ τους. Το προηγούμενο Component Diagram περιέχει ακριβώς αυτό αλλά δεν εστιάζει καθόλου στις λεπτομέρειες της υλοποίησης αλλά στο πώς τα διάφορα συστήματα συνδέονται μεταξύ τους. Το διάγραμμα περιγράφει πλήρως την τωρινή υποδομή αλλά το σύστημα που σχεδιάστηκε δεν είναι υποχρεωτικό να δουλεύει έτσι.

Τα χρώματα στο διάγραμμα δηλώνουν το ποια στοιχεία είναι κομμάτι του ίδιου συστήματος. Τα πορτοκαλί αποτελούν των client της εφαρμογής, Βλέπετε ότι όλο το GUI και γενικά ότι γράφτηκε σε typescript ή svelte αναπαρίστανται με το Business Logic. Έχω ξεχωρίσει το Highcharts και το Mapbox γιατί αποτελούν μεγάλες βιβλιοθήκες που χρησιμοποιήθηκαν σαν μαύρα κουτιά. Το Business Logic απλά τα χρησιμοποιεί και χρησιμοποιεί τα δεδομένα που παίρνει από το API για να τα ρυθμίσει και να τα τροφοδότηση με δεδομένα.

Το Geoserver και το Postgres είναι με διαφορετικά χρώματα από τα άλλα αντικείμενα γιατί αποτελούν αυτόνομες υπηρεσίες με τις δικές τους ρυθμίσεις και μπορούν να χρησιμοποιηθούν και από άλλες εφαρμογές.

Τα ροζ διαγράμματα αποτελούν το Backend της εφαρμογής. Το Svelte Rendering , το Express, το API όπως είπα και νωρίτερα αποτελούν βιβλιοθήκες και εργαλεία που συνδέονται μεταξύ τους για να λειτουργούν ως ένα REST server. Το nginx είναι το reverse proxy που όλη η εφαρμογή τρέχει από πίσω, μας επιτρέπει να τρέχουμε το api σε ένα περιβάλλον που θα λειτουργεί μαζί με άλλες εφαρμογές.

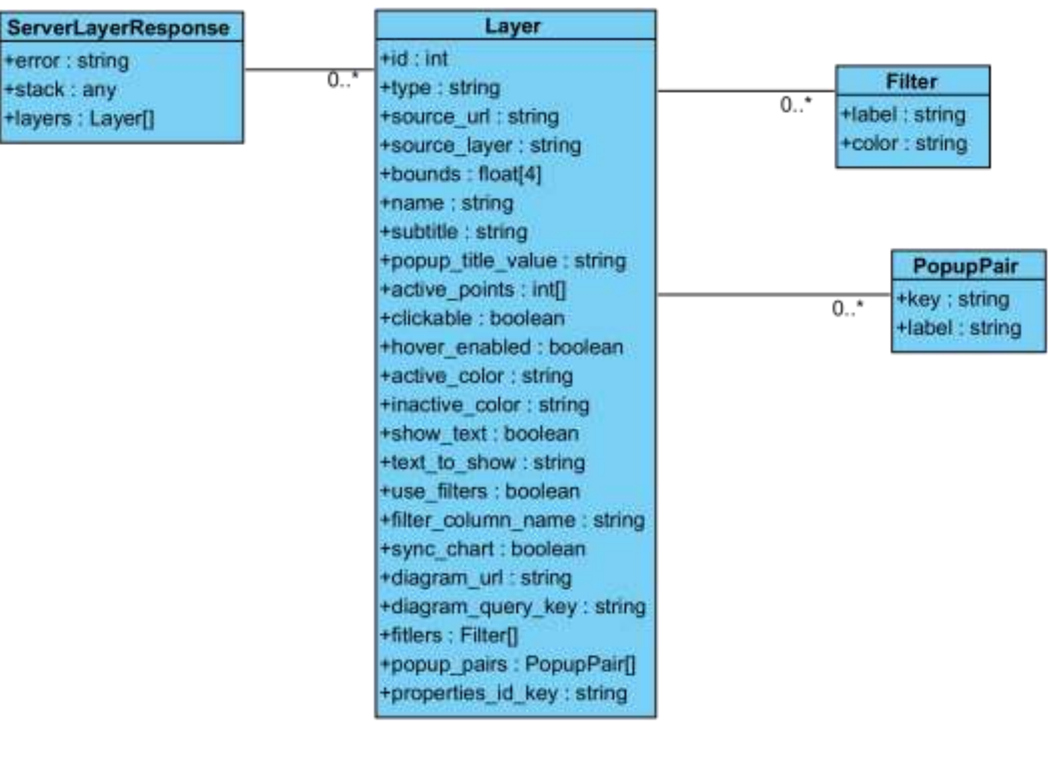
Θεωρητικά μπορούν να υπάρχουν πολλά Geoserver, πολλά diagram providers, πολλά data provider και πολλά layer providers. Εδώ επιλέχτηκε να υλοποιηθούν όλα σε ένα σύστημα καθώς αυτό επιτρέπει το να συνδεθούν τα διάφορα υποσυστήματα και να απλοποιηθεί πολύ το ολικό σύστημα. Το auth provider και το search provider πρέπει να είναι μοναδικά και αποτελεί εσωτερική υπηρεσία γιατί είναι απαραίτητα για να αναζητήσει κάποιος δεδομένα. Οι λόγοι που αυτό συμβαίνει είναι ότι η επικοινωνία των διαφόρων components γίνεται μέσω των meta-δεδομένων που μπορούν να έχουν και διευθύνσεις εκτός των δικών μας υποδομών.

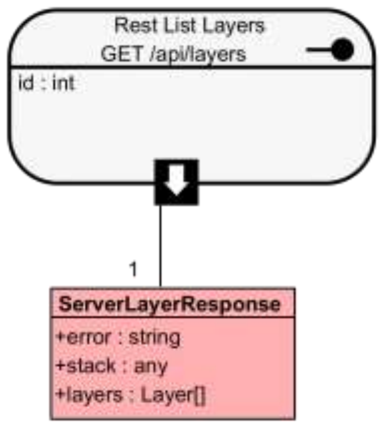


Εικόνα 55: Search Requests

Το Search Component έχει την δυνατότητα για auto-complete στο πεδίο όνομα και για λίστα των διαθέσιμων keywords. Αυτό γίνεται με την χρήση ενός extension της Postgresql που υλοποιεί "fuzzy search" και λέγεται ***pg_trgm***. Αυτό δίνει πρόσβαση στην συνάρτηση "SIMILARITY (text,text)" που δίνει έναν αριθμό από το 0 έως το 1. Το μηδέν σημαίνει καμία σχέση και το ένα είναι πλήρης συσχέτιση. Επίσης η λίστα με τις διαθέσιμες πηγές έρχεται με ανάλογη κλήση με το API. Όταν κάποιος λοιπών κάνει κλικ στο "Apply Filter" γίνεται ένα request στο server που γυρνάει μια λίστα με τα διαθέσιμα datasets.

Αυτό το αντικείμενο έχει ένα πεδίο με το URI που από εκεί θα πάρει την περιγραφή του layer στο χάρτη που θέλει να δείξει. Σε δεύτερο χρόνο θα ζητήσει να πάρει αυτές τις πληροφορίες και να τις μεταβιβάσει στο Map Component. Αφού γίνει λοιπών η αναζήτηση, η λίστα με τα Datasets αποθηκεύονται σε ένα "store" από όπου μπορούν τα διάφορα άλλα Components να ενημερωθούν για αλλαγές στην λίστα.



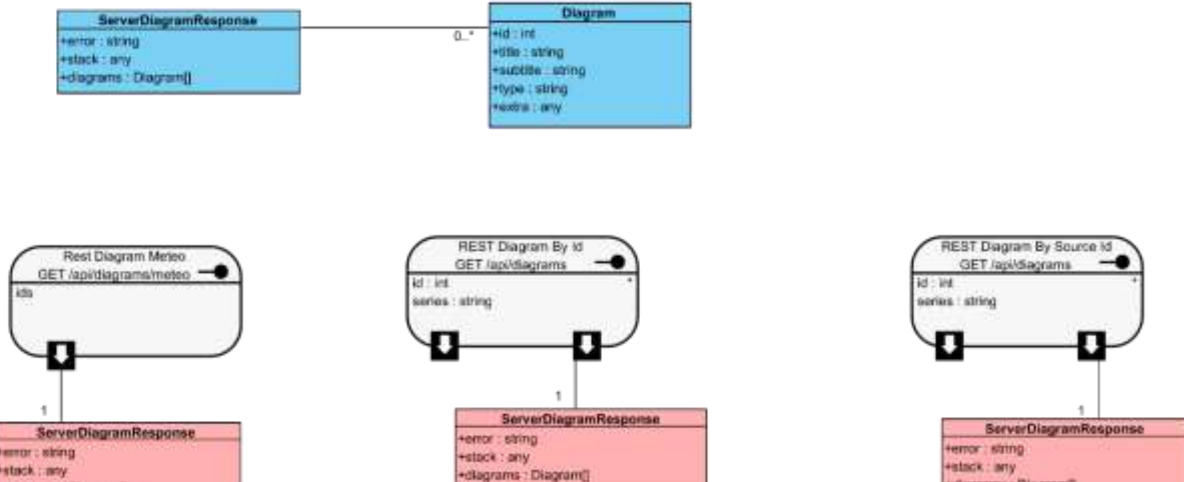


Εικόνα 56: Map Requests

Το Map Component παίρνει τα URLs των layers και πάει σε αυτά να ζητήσει την κάθε περιγραφή. Αυτές τις περιγραφές τις χρησιμοποιεί για να ρυθμίσει το Mapbox και να συνδεθεί με τον Geoserver. Τα διάφορα πεδία του περνιούνται στη βιβλιοθήκη ρυθμίζουν το πως θα φαίνονται οι γεωμετρίες με ιδιότητες όπως χρώμα, ορατότητα, border, κείμενο κτλ. Άλλα πεδία ρυθμίζουν τι θα περιέχει το pop up, αν είναι clickable ή όχι, τα bounds του layer. Εδώ φαίνεται το γεγονός ότι δεν υπάρχει μία πηγή χαρτών αλλά υποστηρίζονται όλες οι πηγές που είναι συμβατές με Mapbox.

Όταν κάποιος κάνει click στο χάρτη πετάγεται ένα παράθυρο (pop up) με πληροφορίες για τα δεδομένα που συνδέονται με το σημείο. Όταν επιλέξει κάποιος μια τοποθεσία προστίθεται το φίλτρο (*diagram_query_key)* στην λίστα των τοποθεσιών έχουν επιλεγεί για να πάρουμε διαγράμματα από αυτά. Ο χρήστης μπορεί να ακυρώσει την επιλογή του με το να κάνει στο σημείο και μετά στο κουμπί “Desellect”.

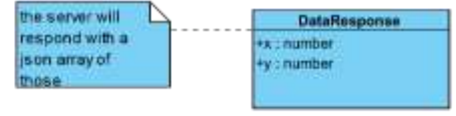
Η λίστα με τα επιλεγμένα σημεία κρατάτε σε ένα “store” και αυτής όπως η περιγραφή των layers για να μπορούν όλα τα Components να ενημερωθούν για τις αλλαγές σε αυτή. Όταν γίνονται αλλαγές σε αυτές προκαλούνται ενημερώσεις που προκαλούν αλλαγές στα δεδομένα που παίρνουν τα επόμενα components.

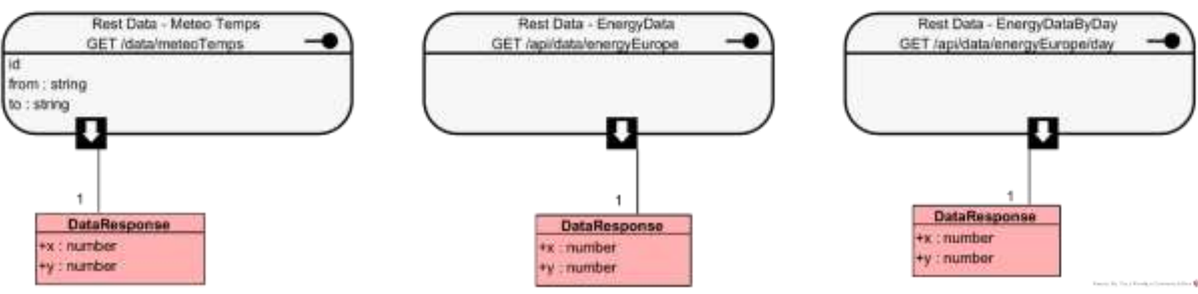


Εικόνα 57: Diagram Requests

Σε αυτό το σημείο έχουμε μία λίστα από διευθύνσεις από τις οποίες μπορούμε να πάρουμε περιγραφές διαγραμμάτων που μας ενδιαφέρουν και μαζί με τα φίλτρα που πήραμε από τον χάρτη φτιάχνονται οι διευθύνσεις από τις οποίες θα πάρουμε δεδομένα. Αυτές καλούνται και παίρνουμε απαντήσεις της παραπάνω μορφής.

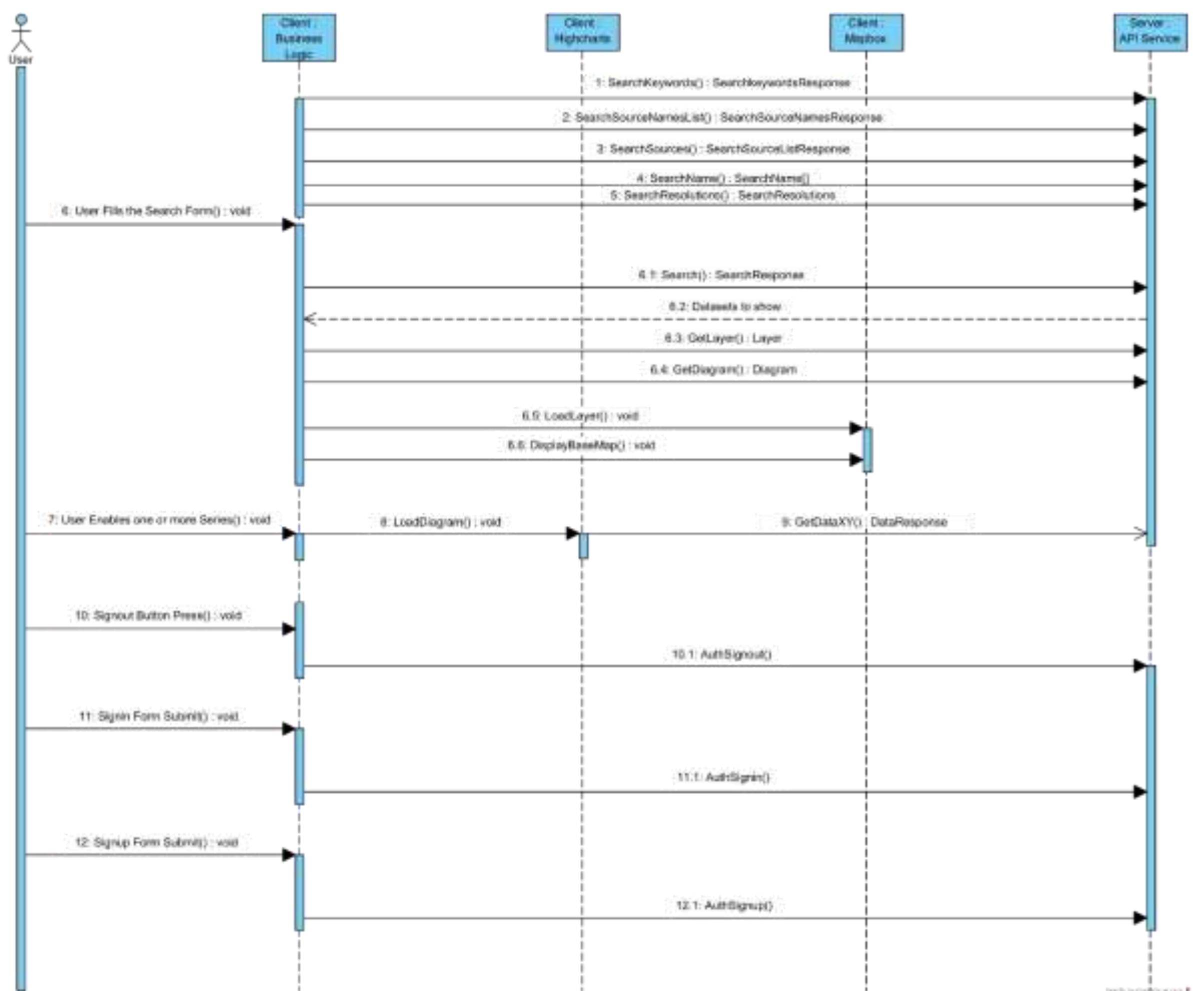
Το περίεργο στο πάνω διάγραμμα είναι το πεδίο extra. Αυτό το πεδίο είναι πολυμορφικό και ο μόνος τρόπος να καταλάβουμε τι δεδομένα έχει είναι να διαβάσουμε το πεδίο type. Για παράδειγμα έχει υλοποιηθεί το type: “rest-multi-dates” που σημαίνει ότι το πεδίο extra είναι λίστα από αντικείμενα που έχουν ένα πεδίο “url” που θα καλέσεις για να πάρεις τα δεδομένα κάθε σειράς. Τα δεδομένα που θα γυρίσεις θα είναι λίστα X,Y πεδία όπου το X είναι ημερομηνία. Ένας άλλος τύπος θα ήταν: “inline-multi-dates” όπου θα ίσχυε το ίδιο όπως τα δεδομένα θα ήταν μέσα στο πεδίο extra. Είναι προφανές ότι μπορούν μέσω αυτής της μεθοδολογίας να υλοποιηθούν όλοι οι πιθανοί διαφορετικοί μηχανισμοί για να αντλήσει το σύστημα δεδομένα από μια πηγή τους.





Έχουν υλοποιηθεί 3 διαφορετικά παραδείγματα από data providers που δίνουν διαφορετικά δεδομένα. Οι μόνες απαιτήσεις ώστε κάποιος να υλοποιήσει τον δικό του

πάροχο δεδομένων είναι να μπορεί η εφαρμογή να τον φτάσει μέσω του διαδικτύου και να παρέχει δεδομένα που έχει με τρόπο που να μπορούν να καταναλωθούν από την ιστοσελίδα με JavaScript. Μία κρυφή απαίτηση είναι ότι το πεδίο Y το διαγράμματος πρέπει να είναι αριθμός, Αυτό είναι περιορισμός του Highcharts.



The flow of Actions Diagram

Το παραπάνω διάγραμμα δείχνει την πλήρη ροή των δεδομένων και αυτό επηρεάζεται από τις ενέργειες του χρήστη. Δείχνει ποιοι είναι οι καταναλωτές των δεδομένων όπως αναλύθηκε νωρίτερα.

4.3 Εργαλεία και τεχνολογίες που χρησιμοποιήθηκαν

Χρησιμοποιηθήκαν επί των πλείστων πρότυπες τεχνολογίες. Η μόνη επιλογή η οποία δεν είναι *ακόμα πρώτη επιλογή* είναι η **Svelte.** Στη συνέχεια θα γίνει αναφορά στο γιατί επιλέχτηκε ενάντια σε πολύ δημοφιλείς συναφείς τεχνολογίες όπως η React και η Angular.

4.3.1 Typescript

Το project θα πρέπει να γραφτεί σε μία γλώσσα κατάλληλη για web development. Η πρώτη επιλογή συνήθως είναι η JavaScript που είναι και η γλώσσα που τρέχουν οι browsers. Το πρόβλημα της είναι ότι είναι πολύ εύκολο να γίνουν λάθη που δεν θα μπορούσαν να βρεθούν μέχρι ο χρήστης να πέσει πάνω τους. Τα περισσότερα από αυτά λύνονται με την

χρήση Typescript μία γλώσσας υπέρ-σύνολο της JavaScript και μεταγλωττίζεται σε αυτή. Για το Backed/ API χρησιμοποιήθηκε επίσης Typescript ώστε να μπορεί να χρησιμοποιηθεί μόνο ένα repo για όλον τον κώδικα.

Η ουσιώδης διαφορά της Typescript είναι ότι έχει τύπους. Σου επιτρέπει όταν ορίζει μια μεταβλητή να δίνεις και τον τύπο της (ή αν μπορεί θα τον βρει μόνη της), και άρα αν αργότερα την χρησιμοποιήσεις με λάθος τρόπος, όπως το να την δώσεις σε μια συνάρτηση που παίρνει άλλο τύπο, ο compiler της γλώσσας θα χτυπήσεις σφάλμα. Βέβαια επειδή κληρονομεί το type system της javascript για το runtime υπάρχει ο τύπος "any" για να γεμίζει τα κενά. Βέβαια η typescript σου αν ρυθμίσεις τον compiler όπως εσύ επιθυμείς ώστε να σου δώσει την μέγιστη ευελιξία. Μπορείς δηλαδή να επιτρέπει της χρήση του τύπου any ή και καθόλου, το οποίο μάλιστα θεωρείται και καλή πρακτική ώστε να αναγκάζεσαι κατά το development να χειρίζεσαι όλα τα πιθανά λάθη.

Πέρα από τους βασικούς τύπους που αντιστοιχούν στην πραγματική μορφή των δεδομένων, σου επιτρέπει η γλώσσα να φτιάχνεις δικούς σου τύπους και μάλιστα σου επιτρέπει να κάνεισ δυναμικούς συνδυασμούς. Για παράδειγμα μπορείς να έχεις τον τύπο ενός αντικειμένου και αργότερα επειδή του πρόσθεσες μερικά πεδία να του αλλάξεις τον τύπο στην επέκταση του αρχικού του. Μπορείς επίσης να φτιάξεις τύπους πολύ περιορισμένους αντί για άλλους γενικούς. Για παράδειγμα μπορείς να φτιάξεις έναν τύπο που θα έχει μία ή δύο περιπτώσεις. Ο compiler χρησιμοποιεί αυτή την πληροφορία για να κάνει optimizations και μετά τους πετάει για να διατηρήσει την ταχύτητα του.

```
type EmailLocaleIDs = "welcome_email" | "email_heading";
type FooterLocaleIDs = "footer_title" | "footer_sendoff";

type AllLocaleIDs = `${EmailLocaleIDs | FooterLocaleIDs}_id`;

        type AllLocaleIDs = "welcome_email_id" | "email_heading_id" | "footer_title_id" | "footer_
```

Εικόνα 58: Typescript string types

Τέλος πρέπει να αναφερθώ σε ένα ακόμη πλεονέκτημα στο να δουλεύεις με compiler. Μπορείς να δουλεύεις στην typescript και να κάνεις χρήση των τελευταίων features της γλώσσας (τελευταία ενημέρωση 2021) και να μπορείς να κάνεις target μέχρι και παλιούς browser μέσα από μια τεχνική που λέγεται polyfills. Ουσιαστικά αν πούμε στον compiler ότι θέλουμε να κάνουμε target IE7 ή άλλους παλιούς browsers θα μεταφράσει τον κώδικα σε άλλο συμβατό με την έκδοση της javascript που τρέχει το target σύστημα. Επίσης όλες οι συναρτήσεις την standard βιβλιοθήκης θα φορτωθούν σαν να είναι δικός μας κώδικας και θα τρέξει πριν από την εφαρμογή μας.

4.3.2 Svelte και Rollup

Έχει γίνει εκτενής αναφορά στο πως κάποιος χρησιμοποιεί την svelte για να στήσει το GUI του και να διαχειριστεί το τα δεδομένα και το state της εφαρμογής του. Δεν έχει αναλυθεί είναι το πως το καταφέρνει και γιατί να επιλεγεί αυτή αντί για ισοδύναμες τεχνολογίες όπως η React. [21]

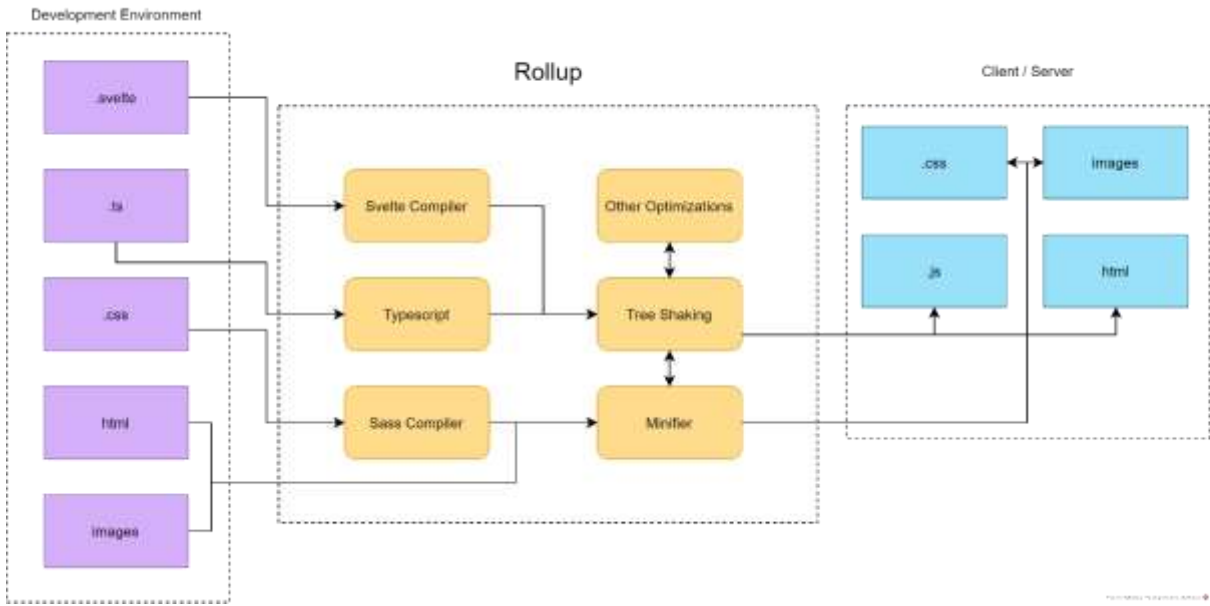
Στα project που στήνουμε με svelte υπάρχει ένα πρόγραμμα που λέγεται bundler. Σε αυτή την εργασία χρησιμοποιήθηκε το rollup καθώς ήταν το προεπιλεγμένο αλλά υποστηρίζεται και το webpack που είναι το πιο γνωστό. [22] Η βασική δουλειά ενός bundler

είναι να πάρει τον κώδικα που έχουμε γράψει και βρίσκεται σε πολλά αρχεία και να τα ενώσει σε ένα μεγάλο πακέτο. Ο λόγος που το κάνουμε αυτό είναι γιατί είναι πολύ πιο αποδοτικό να φορτώσουμε κώδικα σε ένα μεγάλο πακέτο αντί για πολλά μικρά. Έτσι μπορούμε να γράφουμε κώδικα στην μορφή που βολεύει για development και να στέλνουμε στο production ένα μεγάλο πακέτο. [23]

Πέρα από τις βασικές λειτουργίες οι bundlers κάνουν και διάφορα κόλπα για να βελτιώσουν την απόδοση της εφαρμογής. Η τεχνική του tree-shaking παίρνει όλα τα αρχεία μας και προσπαθεί να πετάξει όσα δεν χρειάζονται για να μειώσει τον μέγεθος του κώδικα μας. Άρα οι bundlers μπορούν να επεξεργαστούν και τον κώδικα που τους δίνεται, δεν είναι υποχρεωτικό να τους δίνεται javascript. Στην πράξει μπορεί να δώσεις και να πάρεις CSS, html, json και ότι άλλο θες. [24]

Μέσα από extensions μπορούν να διαβάσουν οποιαδήποτε μορφή αρχείου και να το προσαρμόσουν ώστε να τρέχει σωστά στο browser. Αυτό επιτρέπει για παράδειγμα να γράφω τον κώδικα σε typescript και μέσα από ένα extension να βάλω το rollup να καλέσει των compiler να πάρει το αποτέλεσμα και να ενσωματώσει στο πακέτο που στέλνει στον client.

Όλες οι πλατφόρμες όπως η React, η Angular και η Svelte χρησιμοποιούν αυτόν τον μηχανισμό για να φτιάξουν της εφαρμογές τους. Φτιάχνουν δηλαδή ένα extension για τον bundler που καλεί μια βιβλιοθήκη που επεξεργάζεται τον κώδικα τους και παράγει javascript css και html. Πρόκειται στην πράξη για ένα πλήρως configurable build pipeline, που μπορεί κάποιος να πάρει και να προσαρμόσει στο framework του.



Εικόνα 59: Rollup Pipeline

Η Svelte είναι ένας compiler. Κάθε svelte αρχείο μεταφράζεται σε ένα javascript αντικείμενο και όλη η λειτουργικότητα του γίνεται με μία κλάσει που παίρνει σαν παραμέτρους κάποιες συναρτήσεις που ρυθμίζουν την συμπεριφορά του. Άρα κάθε Svelte αρχείο έχει ένα προς ένα αναπαράσταση στην javascript. Αυτό είναι αρκετά ενδιαφέρον γιατί δεν είναι ο συνηθισμένος τρόπος που υλοποιείται η functionality που παρέχει η Svelte. Άρα αν κάποιος πάρεις ένα Svelte Component και το κάνει compile παίρνει μία κλάση που μπορεί να χρησιμοποιήσει κατευθείαν σε javascript χωρίς την svelte. Παίρνεις το αντικείμενο το κάνεις initialize και το χρησιμοποιείς.

Ο λόγος που η Svelte είναι διαφορετική από ισοδύναμες της τεχνολογίες είναι ότι δεν χρησιμοποιεί το virtual DOM [25]. Όλες τα άλλα web app frameworks έχουν αυτό το πρόγραμμα τους στο runtime της εφαρμογής. Πρόκειται για μία κρυφή δομή στην μνήμη που έχει ένα αντίγραφό το πραγματικού DOM, πάνω στο οποίο γίνονται όλες οι αλλαγές. Όταν το runtime αποφασίσει ότι είναι ώρα για αλλαγές συγχρονίζεται με το πραγματικό DOM δείχνει στο χρήστη τις αλλαγές. Αυτό είναι μια τεχνική για να γλιτώσει η εφαρμογή κάποια DOM events που θεωρούνται αργά και να τα κάνει στο virtual.

Όμως αυτό έρχεται με κόστος το diffing. Ο συγχρονισμός του virtual με το πραγματικό DOM δεν είναι δωρεάν αλλά κοστίζει πρέπει να βρεθούν οι διαφορές τους. Σε μικρού όγκου σελίδες αυτό το κόστος δεν είναι μεγάλο και πράγματι οι εφαρμογές γίνονται πιο αποδοτικές αλλά αν έχει ένα μεγάλο αριθμό τα nodes στο DOM αυτό αρχίζει να κοστίζει πολύ. Η Svelte είναι το πρώτο framework που κατάφερε συγκρίσιμη απόδοση και σε πολλές περιπτώσεις καλύτερη χρησιμοποιώντας μόνο το πραγματικό DOM.

Το γεγονός ότι δεν χρειάζεται το virtual DOM μειώνει πάρα πολύ το κόστος σε δεδομένα μου έχει η svelte για τον χρήστη. Το κόστος του να χρησιμοποιήσεις Svelte είναι αισθητά μικρότερο καθώς το ελάχιστο react app ξεκινάει 379KB και μερικές διορθώσεις μπορεί να φτάσει τα 90KB [26], ενώ όταν έκανα build το ελάχιστο Svelte app μου πείρε 4KB στο δίσκο.

4.3.3 Sapper -- SvelteKit

Το Sapper είναι το framework που χρησιμοποιεί την svelte σαν βάση για να παρέχει ένα πλήρες application framework μου routing, server side rendering, service worker και φυσικά αφού βασίζεται στο Express μπορεί α χρησιμοποιηθεί σαν ένα server γενικής χρήσης.

Κατά την διάρκεια της ανάπτυξης της εργασία το project έχει μετά-ονομαστεί σε SvelteKit το οποίο θα αποτελέσει την τελική έκδοση της Sapper αφού το project είναι αρκετά ώριμο για να σταθεροποιηθεί το API του. Το project δεν έχει ολοκληρωθεί αλλά όταν τελειώσει θα αποτελεί το καινούργιο όνομα του ίδιου συστήματος, αφού σύμφωνα με το documentation που έχουν δημοσιεύει μέχρι τώρα δεν έχουν γίνει σημαντικές αλλαγές στο API. Φυσικά στο web development τα πράγματα δεν μένουν σταθερά ποτέ.

Σαν αρχιτεκτονική το sapper αποτελεί ένα middleware για το express. Ένας middleware είναι μια συνάρτηση που δουλεύει ως εξής, παίρνει ένα Request που του δίνεται από το express και είτε απαντάει με ένα Response, είτε αλλάζει το Request με κάποιον τρόπο και το περνάει στο επόμενο middleware. Αυτό που κάνει η sapper είναι να παίρνει το Request, να βλέπει το route και να ελέγχει το router του. Αν έχει το ανάλογο route απαντάει με αυτό αλλιώς απαντάει με κάποιο error page. Στην ουσία το sapper έχει ενσωματωμένο ένα τμήμα του runtime της Svelte και τρέχει τα Components που αντιστοιχούν στο route, κάνοντας χρήση αυτού του runtime για να υπολογίσει ποια θα είναι η μορφή του DOM πριν ξεκινήσει να τρέχει το lifetime της Svelte μαζί με την κατάσταση της μνήμης στην javascript. Όλα αυτά τα επιστρέφει στον χρήστη κάνοντας την σελίδα να φορτώνει γρηγορότερα σε πιο αργούς υπολογιστές ή χειρότερες συνδέσεις όπως για παράδειγμα σε ένα κινητό.

Αυτή η τεχνική επιτρέπει να σύνολο από βελτιστοποιήσεις όπως data prefetching, search engine optimization, response caching. Φυσικά αυτό γίνεται με ένα κόστος στην απόδοση του server, καθώς ένα μέρος της δουλειάς που κανονικά θα έκανε ο client, τώρα την κάνει το server. Άρα σίγουρα αυξάνονται οι απαιτήσεις στο server. Δεν πρόκειται πλέον απλά για ένα web app αλλά για τεχνολογία που τρέχει μερικώς στο server και μερικώς στον client.

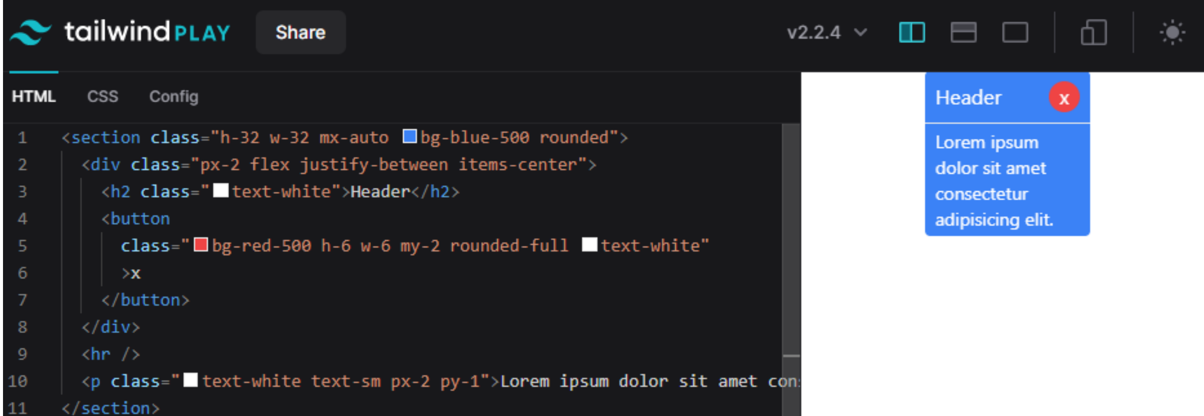
4.3.4 Tailwind CSS

Όλη η CSS της εφαρμογής γράφτηκε με το framework της Tailwind. Προτιμήθηκε από εναλλακτικές για την ευελιξία του καθώς και των ορθολογικό σχεδιασμό του, που δεν παρεμβαίνει στον τρόπο που σχεδιάστηκε το site. Από την στιγμή που ανακοινώθηκε φαίνεται υιοθετήθηκε ραγδαία από τους developer ανά τον κόσμο. [27]

Το πρώτο πράγμα που κάνει η Tailwind είναι να αφαιρεί το styling από όλα τα element της html [28]. Έτσι όλα τα head elements (h1, h2, …) φαίνονται το ίδιο, έχουν αφαιρεθεί τα margin και τα padding από όλα τα στοιχεία της σελίδα, οι λίστες δεν έχουν style και οι εικόνες γίνονται block elements για να αποφευχθούν προβλήματα με το alignment τους. Με αυτό τον τρόπο ανεξάρτητα από τον browser η σελίδα θα ξεκινάει ως ο ίδιος άδειος καμβάς προς χρήση.

Εικόνα 60: Tailwind reseted header styles

Το API της Tailwind είναι βασισμένο στο περιεχόμενο, αυτό σημαίνει ότι μπορείς να γράψεις τα κείμενα, να βάλεις τις εικόνες, τα διαγράμματα και ότι άλλο θες πάνω στην σελίδα σου και μετά να ασχοληθείς με το πως θα το κάνεις να φαίνεται όμορφο. Για να γίνε ιατό αντιληπτό ας πάρουμε το παράδειγμα της δημιουργίας μια κάρτας.



Εικόνα 61: Card Example

Το πρώτο πράγμα που φαίνεται στο κώδικα αριστερά είναι το section tag που θα αποτελέσει την κάρτα. Του έχουν δοθεί μια σειρά από κλάσεις που αριστερά προς δεξιά κάνουν το εξής. Θέτουν το ύψος και το πλάτος στα 32 units, θέτουν τα margin δεξιά και αριστερά στην τιμή “auto”, κάνουν το background μπλε και στρογγυλεύουν τις γωνίες. Το div που είναι το στην επόμενη γραμμή θέτει το padding στο οριζόντιο άξονα στα 2 units, έχει

display property flex, και τοποθετεί τα παιδιά του όπως φαίνονται στο σχήμα με τις επόμενες δύο classes. Από τα ονόματα των classes είναι εύκολο να αντιληφθεί κανείς τι περίπου κάνει το κάθε class και για τις λεπτομέρειες έχουν αναλυτικότατο documentation [29].

Με τον συνδυασμό πολλών απλών κλάσεων μπορούμε να έχουμε το ίδιο αποτέλεσμα που θα είχαμε αν φτιάχνε δύο ή τρεις περίπλοκες κλάσεις. Αυτό επιταχύνει πολύ το development γιατί τις περισσότερες φορές δεν χρειάζεται να γράψουμε CSS καθώς ότι και να θέλουμε να υλοποιήσουμε έχει ήδη γίνει στην Tailwind. Μάλιστα οι κλάσεις που έχουν απόλυτα μεγέθη όπως το “h-32” χρησιμοποιούν μεγέθη σχετικά με το font το font της σελίδας που λέγονται rem και είναι ο καθιερωμένος τρόπος να ορίζουμε μεγέθη όταν θέλουμε να φτιάξουμε ένα responsive interface.

h-12	height: 3rem;
h-14	height: 3.5rem;
h-16	height: 4rem;
h-20	height: 5rem;
h-24	height: 6rem;
h-28	height: 7rem;
h-32	height: 8rem;
h-36	height: 9rem;
h-40	height: 10rem;

Εικόνα 62: Tailwind height classes

Η tailwind μας δίνει κλάσεις για το όταν θέλουμε να θέσουμε τιμές ανάλογα με το μέγεθος της οθόνης ή με να αλλάξουμε το στυλ όταν ο χρήστης κάνει click σε ένα κουμπί ή hover ή με μάζι κάποιον άλλο από του pseudo-selectors της CSS. Όλες τις οι κλάσεις έχουν αντίστοιχες με άλλα προθέματα που κάνουν την ίδια δουλειά αλλά μόνο αν συμβεί αυτό για το οποίο το πρόθεμα ελέγχει.

```
<!-- Width of 16 by default, 32 on medium screens, and 48 on large screens -->
<img class="w-16 md:w-32 lg:w-48" src="...">

<!-- On Hover the button will have a darker shade of red -->
<button class="bg-red-500 hover:bg-red-700 ...">
  Hover me
</button>
```

Εικόνα 63: Tailwind state driven changes

Όλες αυτές τις κλάσεις προφανώς και δεν τις έφτιαξαν με το χέρι, αντίθετα έγραψαν ένα πλήρες built σύστημα βασισμένο στον PostCSS preprocessor ο οποίος δημιουργεί αυτόματα τις κλάσεις της Tailwind που χρησιμοποιούμαι. Στο δικό μας project αυτός ο preprocessor συνδέθηκε πάνω στο rollup bundler για να γίνεται αυτόματα αυτή επεξεργασία σε όλο μας το project.

Για να μπορεί το PostCSS να βρει όλα τα αρχεία στα οποία μπορεί να έχουν ορίσει Tailwind classes υπάρχει ένα config από το οποίο έχει μέσα την πληροφορία του ποια αρχεία να ελέγξει για classes. Από αυτό το config μπορούμε να επεκτείνουμε την Tailwind, καθώς μπορούμε να ορίσουμε να δικά μας χρώματα, να ενεργοποιήσουμε απενεργοποιημένες classes, να συνδεθούμε και να πειράξουμε το τα διάφορα συστήματα της. Το δικό μας σύστημα για παράδειγμα έχω προσθέσει μερικά δικά μου χρώματα και ένα καινούργιο grid layout για να υλοποιήσω εύκολα το GUI. Αυτό απλοποιεί πολύ και την περίπτωση που θέλουμε αργότερα να κάνουμε αλλαγές. Έστω ότι θέλουμε αργότερα να αλλάξουμε το "container" χρώμα, αυτή η αλλαγή αυτόματα θα ενημερώσει και όλες classes που χρησιμοποιούν αυτό το χρώμα (bg-container, border-container, text-container, κτλ).

```
module.exports = {
    purge: {
      mode:"all",
        content: ['./src/**/*.svelte', './src/**/*.html'],
    },
    darkMode: false, // or 'media' or 'class'
    theme: {
        extend: {
            colors: {
                container: '#d8e0e8',
                red: '#BD373E',
                gray: '#ececec',
                cyan: '#1c4a6b',
            },
            gridTemplateRows: {
                // Simple 8 row grid
                '8': 'repeat(8, minmax(0, 1fr))',
            },
            gridRow: {
                'span-7': 'span 7 / span 7',
            },
        },
    },
    variants: {
        extend: {
            fontSize: ['hover', 'focus'],
            cursor: ['hover', 'focus'],
            borderRadius: ['focus'],
            borderWidth: ['focus'],
        },
    },
    plugins: [],
};
```

Εικόνα 64: tailwind.config.js

Το μόνο πρόβλημα που έχει η Tailwind είναι ότι για να υλοποιηθεί κάτι παίρνει συχνά μεγάλο αριθμό κλάσεων, ενώ αν η αντίστοιχη CSS είχε γραφεί με το χέρι θα ήταν ελάχιστες κλάσεις. Υπάρχει τρόπος αυτό να αποφευχθεί που ταυτόχρονα χρησιμοποιεί. Το root file της Tailwind είναι το αρχείο "global.pcss" από όπου λέμε στην PostCSS να χρησιμοποιήσει τις κλάσεις της Tailwind, όμως επιτρέπεται εκεί να ορίσουμε και δικές μας οι οποίες θα προκύψουν ως ο συνδυασμός πολλών μικρών κλάσεων και φυσικά δικιά μας CSS.

```
@tailwind base;
@tailwind components;
@tailwind utilities;

@layer utilities {
  .container {
    @apply flex bg-container text-cyan  rounded-tl-lg rounded-br-lg max-w-full;
  }


  .button {
      @apply border bg-container text-cyan p-1 lg:p-2 rounded hover:bg-opacity-90 focus:bg-opacity-95;
  }
  .button-raised {
      @apply bg-cyan;
      @apply text-white;
  }
  .button-rounded {
    @apply rounded-full;
  }
}
```

Εικόνα 65: global.pcss

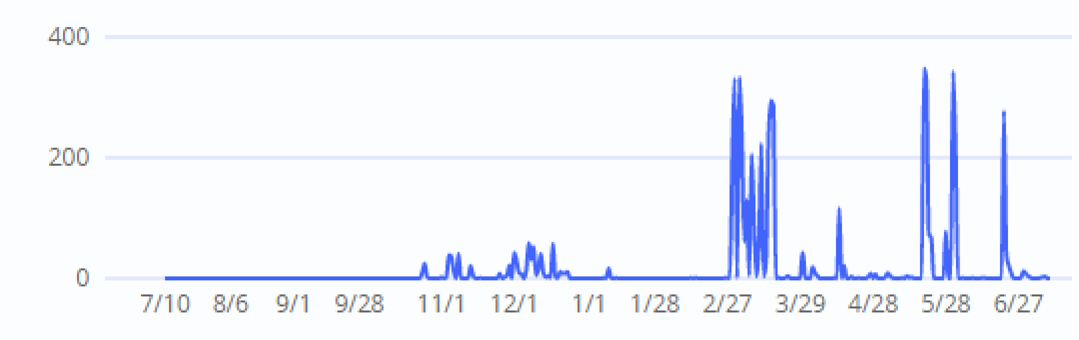
Οι πρώτες τρεις γραμμές αποτελούν τα βασικά layer της Tailwind και συνήθως δεν τα πειράζουμε, όμως μπορούμε να προσθέσουμε τις δικές μας κλάσεις στο τελευταίο layer όπου θα έχουμε και πρόσβαση σε όλη την υποδομή της Tailwind. Όπως φαίνεται στην εικόνα η κλάση container φτιάχνετε από τον συνδυασμό πέντε κλάσεων ενώ οι κλάση button από επτά. Το directive "@apply" παίρνει τις κλάσεις που ακολουθούν και ενώνει την CSS τους. Αυτός είναι ο ιδανικός τρόπος να χειριστούμε περιπτώσεις όπου έχουμε επαναλαμβανόμενους συνδυασμούς από κλάσεις ή υπάρχει ένα UI element που επαναλαμβάνετε.

4.3.5 Mapbox

Το Mapbox είναι ένα ολοκληρωμένο σύστημα αναπαράστασης γεωγραφικών δεδομένων που έχει όλες τις δυνατότητες που θα χρειαζόταν κάποιος για να υλοποιήσει μια εφαρμογή που περιέχει χάρτες. Υπάρχουν εναλλακτικές αλλά προτιμήθηκε για το πολύ καλό του API. Υποστηρίζει το να αναπαραστήσει όλες τις μορφές δεδομένων από εικόνες μέχρι δυναμικά δεδομένα να συνδεθεί με standard APIS όπως το WFS και δίνει στο προγραμματιστής τρόπους να πάρει τα δεδομένα που κουβαλάνε οι χάρτες μαζί τους.

Υποστηρίζει όλες τις μεγάλες πλατφόρμες (JS, IOS, Android, Unity) και πρόκειται μια υπηρεσία επί πληρωμή. Σου δίνει τη δυνατότητα να φτιάξεις τους χάρτες σου κατευθείαν στην πλατφόρμα τους και να αλλάξεις από τα χρώματα και τα κείμενα, μέχρι την αναπαράσταση ενός τρισδιάστατου χάρτη στη unity. Επίσης προσφέρει υπηρεσίες για αναζήτηση τοποθεσιών, turn-by-turn navigation, data visualization και asset tracking.

Κατά την ανάπτυξη της εργασία χρησιμοποιήθηκε το free tier της πλατφόρμας λόγο της μικρής χρήσης που έγινε αλλά αν μια εφαρμογή κάνει εκτενή χρήση των APIs και έχει πολλούς χρήστες τότε θα έχει χρεώσεις όπως αυτές ορίζονται στο σελίδα τους [30]. Για να γίνεται ορθολογική χρήση της πλατφόρμα όταν κάποιος θέλει να χρησιμοποιήσει τις βιβλιοθήκες τους παίρνει ένα API key που πρέπει να βάλει σε όλες τις εφαρμογές του. Έτσι ξέρουν ότι οι εντολές που γίνονται προς αυτούς είναι σωστές. Προσφέρουν επίσης στατιστικά χρήσης για το κάθε API key ξεχωριστά ώστε να μπορεί κάποιος να δει πως χρησιμοποιεί η εφαρμογή του το mapbox.



Εικόνα 66: Our usage stats

Εμείς χρησιμοποιήσαμε τη javascript βιβλιοθήκη τους που είναι κατάλληλη για web browsers. Το entry point του API τους είναι το Map αντικείμενο που παίρνει ένα id για το html element που θα συνδεθεί επάνω και μερικές ρυθμίσεις ακόμα ορίζουν το zoom που θα κεντράρει ο χάρτης, το access token και πολλά άλλα. Μετά μπορείς να ορίσεις μία πηγή χαρτών όπου και μπαίνουν Geoserver layers που ζητάει η εφαρμογή. Έτσι ένα αντικείμενο χάρτη στο οποίο πάνω έχεις συνδέσει όλα τα sources σου, από τα οποία μπορείς αργότερα να φτιάξεις layers για τον χάρτη.

```
export function createPublicMultiPolygonLayer(l: Layer, map: mapboxgl.Map) {
    const sourceId = `source-${l.id}`;
    map.addSource(sourceId, {
        type: 'vector',
        tiles: [l.source_url],
        bounds: l.bounds,
    });
    l.filters.forEach(({ label, color }, index) => {
        const lid = `layer_${l.id}_${index}`;

        map.addLayer({
            id: lid,
            source: sourceId,
            type: 'fill',
            'source-layer': l.source_layer,
            filter: l.use_filter ?
                    ['==', l.filter_column_name, label] :
                    ['boolean', '1', true],
            layout: {
                // visibility: l.use_filter ? (true ? 'visible' : 'none') : 'visible',
            },
            paint: {
                'fill-color': color,
                'fill-opacity': 0.5,
                // 'fill-opacity': 0.8,
                'fill-outline-color': '#000000',
            },
        });
    });
}
```

Εικόνα 67: Creating Mapbox Layers

Εδώ φαίνεται ότι από ένα source φαίνονται ότι μπορούμε να φτιάξουμε πολλά layers από το ίδια δεδομένα. Αυτό γίνεται γιατί μπορούμε από την ίδια πηγή να βάλουμε ένα φίλτρο κα να χρωματίσουμε τα μισά δεδομένα με το ένα χρώμα και τα υπόλοιπα με το ένα άλλο χρώμα χρησιμοποιώντας το αντίθετο φίλτρο. Άρα το Mapbox Layer είναι ένα style για το πως φαίνονται τα πράγματα μαζί με μία σύνδεση σε ένα Mapbox source για να πάρει τα δεδομένα που του αντιστοιχούν. Οι ρυθμίσεις για το πως φαίνεται ένα layer μπορούν να αλλάξουν και κατά στην εκτέλεση.

4.3.6 Geoserver

Η πιο πλήρης υλοποίηση του standard του WFS καθώς και άλλα API χαρτών, προσφέρεται από το σύστημα που λέγεται GeoServer. Πρόκειται για μια πλήρη σουίτα εργαλείων και υπηρεσιών εξειδικευμένων στην παροχή χωρικών δεδομένων με πρότυπα OGC.

Εικόνα 68: Geoserver available services

Τα services που φαίνονται στο σχήμα είναι όλα τρόποι να παρέχονται χάρτες σε εφαρμογές. Οπότε στην ουσία το GeoServer παίρνει τα δεδομένα από κάποιο αποθηκευτικό μέσω και τα δίνει με οποιαδήποτε από τις παραπάνω εφαρμογές.

Data Type	Workspace	Store Name	Type
	nurc	arcGridSample	ArcGrid
	nurc	img_sample2	WorldImage
	nurc	mosaic	ImageMosaic
	tiger	nyc	Directory of spatial files (shapefiles)
	data	postgis	PostGIS
	sf	sf	Directory of spatial files (shapefiles)
	sf	sfdem	GeoTIFF
	topp	states_shapefile	Shapefile
	topp	taz_shapes	Directory of spatial files (shapefiles)
	nurc	worldImageSample	WorldImage

Εικόνα 69: Geoserver Storage Connectors

Όπως φαίνεται διάφοροι τρόποι αποθήκευσης υποστηρίζονται και από το σύστημα. Κατά την ιστορική αναδρομή αναφέρθηκε ότι πριν το WFS, στέλνονταν τα αρχεία με FTP. Η μορφή των αρχείων που θα στέλναμε είναι τα Shapefile, GeoTIFF και άλλα. Οπότε είναι προφανές ότι αυτοί οι τρόποι αποθήκευσης είναι συμβατοί και με το GeoServer. O Τύπος αποθήκευσης PostGIS είναι επέκταση της βάσης δεδομένων Postgres και αποτελεί ένα πολύ αποδοτικό τρόπο αποθήκευσης. Ό λόγος που γίνεται ειδική αναφορά είναι το ότι σε αυτή την μορφή τα δεδομένα μπορούν να ζουν και άλλο υπολογιστικό σύστημα, βελτιώνοντας την απόδοση του συστήματος περεταίρω.

Type	Title	Name	Store	Enabled	Native SRS
	World rectangle	tiger:giant_polygon	nyc	✓	EPSG:4326
	Manhattan (NY) points of interest	tiger:poi	nyc	✓	EPSG:4326
	Manhattan (NY) landmarks	tiger:poly_landmarks	nyc	✓	EPSG:4326
	Manhattan (NY) roads	tiger:tiger_roads	nyc	✓	EPSG:4326
	A sample ArcGrid file	nurc:Arc_Sample	arcGridSample	✓	EPSG:4326
	North America sample imagery	nurc:Img_Sample	worldImageSample	✓	EPSG:4326
	Pk50095	nurc:Pk50095	img_sample2	✓	EPSG:32633

Εικόνα 70: Geoserver Layer

Τα διάφορα δεδομένα λοιπόν, το καθένα στην δικιά του μορφή, πρέπει με ένα μηχανισμό να εκτεθούν προς τους χρήστες. Ο μηχανισμός αυτός λέγεται “Layers” και στην πράξη είναι το εξής. Μαζεύοντας δεδομένα από μια πηγή φτιάχνονται κάποια meta-δεδομένα για αυτές τις πληροφορίες. Στην πρώτη στήλη του παραπάνω πίνακα βλέπω ότι οι Layers έχουν διαφορετικούς τύπους. Οι δύο κύριοι τύποι είναι πολύγονα και σημεία. Στην τελευταία στήλη φαίνεται ότι το κάθε Layer επιτρέπετε να είναι σε διαφορετικό σύστημα αναφοράς. Συγκεκριμένα το GeoServer πρέπει να έχει το κατάλληλο σύστημα αναφοράς στο οποίο είναι δοσμένα τα δεδομένα για να μπορεί να τα παρουσιάσει σωστά. Αυτό γίνεται γιατί δίνει την δυνατότητα σε αυτό που καλεί το σύστημα να επιλέξει σε ποιο σύστημα αναφοράς θέλει τα δεδομένα και το GeoServer κάνει την μετατροπή.

Μία άλλη πολύ χρήσιμη λειτουργία του GeoServer είναι η ικανότητα του να δώσει τα δεδομένα σε όποια μορφή του ζητηθεί. Έχει επίσης την δυνατότητα να του προστεθούν και καινούργιες μορφές μέσω επεκτάσεων.

Tile Image Formats

- [x] application/json;type=geojson
- [x] application/json;type=topojson
- [x] application/json;type=utfgrid
- [x] application/vnd.mapbox-vector-tile
- [ ] image/gif
- [x] image/jpeg
- [x] image/png
- [ ] image/png8
- [ ] image/vnd.jpeg-png
- [ ] image/vnd.jpeg-png8

Εικόνα 71: Geoserver available Response types

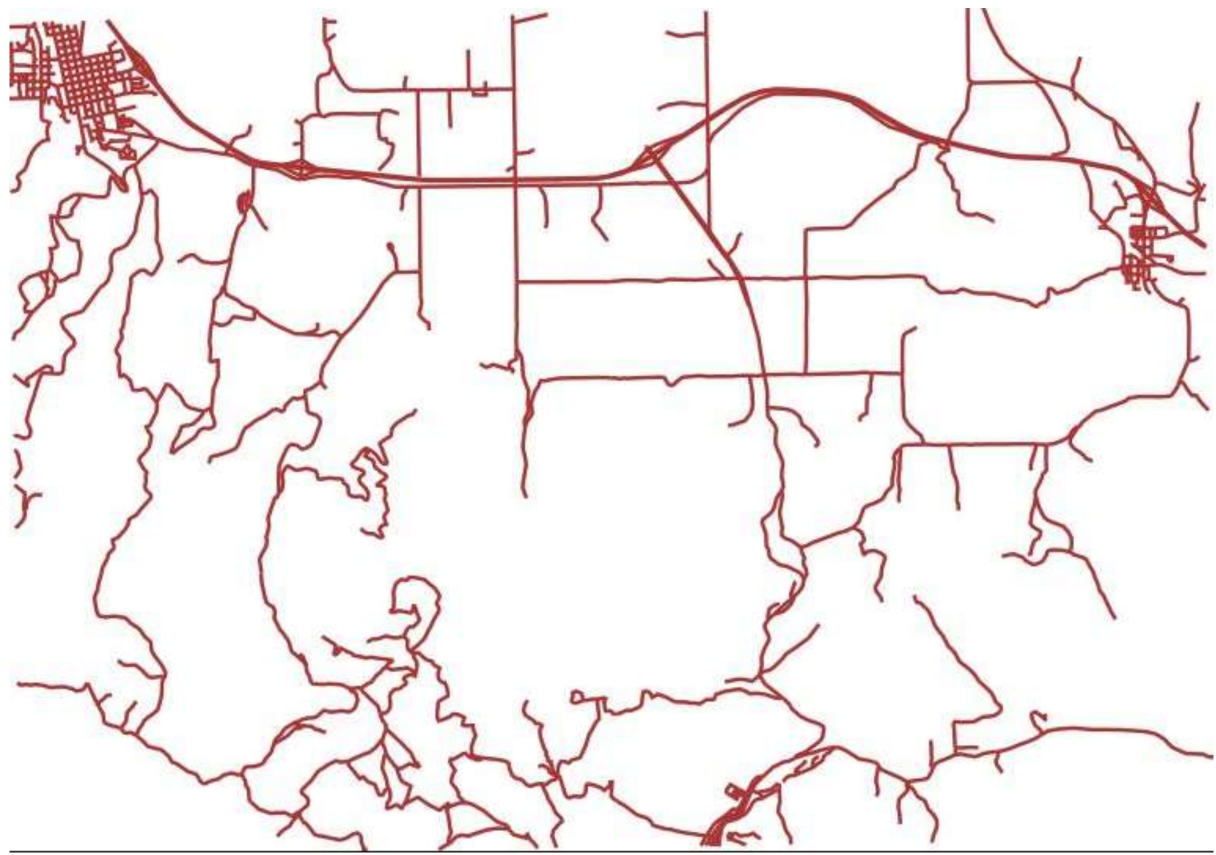
Τέλος πρέπει να γίνει μια αναφορά σε χαρακτηριστικά παραδείγματα χρήσης της τεχνολογίας. Ως χαρακτηριστικό παράδειγμα θα πάρω αυτή την Request:

http://geoserver.example/geoserver/data/wms?service=WMS&version=1.1.0&request=GetMap&layers= sf:%3Aroads&bbox=-181.800003051758%2C-90.7678298950195%2C181.800003051758%2C84.5328063964844&width=768&height=370&srs=EPSG%3A4326&styles=&format=application/openlayers

Ας σπάσουμε την εντολή στα διάφορα μέρη της. Στην αρχή υπάρχει το URL της ιστοσελίδα το οποίο έχει το path: "/geoserver/data/wms" που όπως επίσης φαίνεται θα χρησιμοποιήσεις τον μηχανισμό WMS. Μετά ακολουθούν οι παράμετροι:

Όνομα	**Τιμή**	**Περιγραφή**
Service	WMS	Πρωτόκολλο
Version	1.1.0	Έκδοση πρωτοκόλλου
Request	GetMap	Η εντολή προς το server
Layers	sf:roads	Το layer που θέλουμε
Bbox	181.800003051758, -90.7678298950195, 181.800003051758, 84.5328063964844	4 αριθμοί που λένε που είναι το 0,0 του χάρτη σε σχέση με των παγκόσμιο χάρτη
Width	7688	Το μήκος του χάρτη
Height	3708	Το ύψος του χάρτη
Srs	EPSG:4326	Το σύστημα συντεταγμένων
Format	Image/jpeg	Η μορφή που ζητάω να είναι τα δεδομένα

Όπως φαίνεται στην παραπάνω λίστα με την εντολή GetMap ζητήθηκε το layer sf:roads σε μορφή OpenLayers map με μήκος 7688, ύψος 3708, σε σύστημα συντεταγμένων EPSG:4326 και το παραπάνω bounding box. Το αποτέλεσμα είναι το παρακάτω:

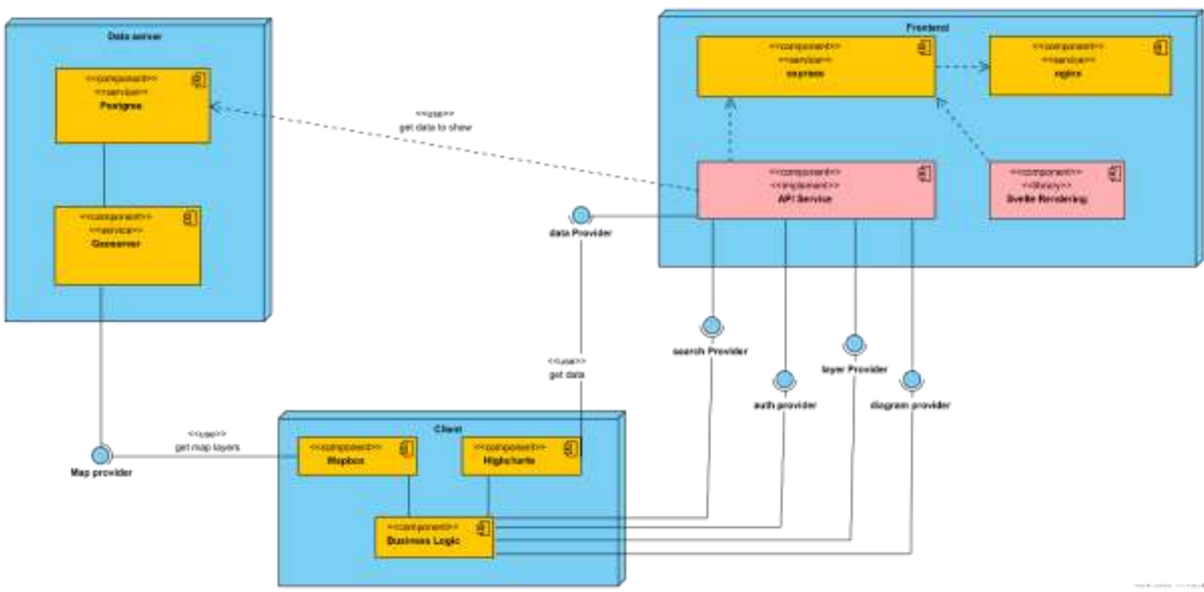


Εικόνα 72: Geoserver roads layer

Με άλλες παραμέτρους θα μπορούσα να είχα πάρει τα ίδια δεδομένα με άλλη μορφή. Βλέποντας όλα αυτά είναι προφανές ότι το GeoServer είναι ευέλικτο και δίνει τι δυνατότητα στους χρήστες να πάρουν δεδομένα με τρόπο που ταιριάζει στην εφαρμογή που στήνουν. Για παράδειγμα στο σύστημα που στήθηκε για της ανάγκες αυτής της εργασίας χρησιμοποιήθηκε η επέκταση που επιτρέπει στο GeoServer να δίνει τα δεδομένα με μορφή συμβατή με Mapbox που χρησιμοποιείται.

4.4 Διάταξη της εφαρμογής στο okeanos

Για την ανάπτυξη της εφαρμογή δόθηκαν από τον okeanos, δυο εικονικά μηχανήματα και 2 IPs για να στηθούν οι υπηρεσίες. Στην μία στήθηκε το Geoserver μαζί με την βάση δεδομένων και στο άλλο στήθηκαν το API μαζί με το file hosting για τα αρχεία που σερβίρονται στον χρήστη. Τα υπόλοιπα όπως φαίνεται παραπάνω θα τρέχουν στον σελιδοδείκτη του χρήστη. Στο Fronted server έγινε σύνδεση με το github για να υλοποιηθεί ένα ποιο εύκολο workflow. Μέσα από ειδικά script υλοποιήθηκε Continuous Integration, Continuous Development (CI/CD) ώστε όταν στο τέλος της ημέρας γινόταν συγχρονισμός του κώδικα στο github με αυτόν που έτρεχαν τα server.



Εικόνα 73: Api Representation

Chapter 5: Σχολιασμός της αρχιτεκτονικής και της εμπειρίας

Ο σχεδιασμός του συστήματος έγινε με βασικό κριτήριο τη μέγιστη ευελιξία και επεκτασιμότητα του. Θα ήταν πολύ εύκολο κάποιος να σχεδιάσει ένα καινούργιο interface για κινητά ή για table ή ακόμα μια καινούργια ιστοσελίδα η οποία συνδέεται με το API για να υλοποιήσει τις λειτουργίες του.

Αν κάποιος έχει τα δικά του δεδομένα σε μια μορφή που τον βολεύει, μπορεί να μπει στο σύστημα χωρίς να τα φορτώσει σε δική μας βάση αρκεί να μας πεις πως να τα πάρει το σύστημα ώστε να γίνει η σύνδεση. Μπορεί να έχει σε δικό του σύστημα τα meta-δεδομένα και να τα δίνει αυτός. Το μόνο που θα χρειαστεί είναι για να μπορεί να τα βρει κάποιος με αναζήτηση να εγγραφθεί σαν ένα καινούργιος πάροχος.

Επειδή δόθηκε μεγάλη ελευθερία στις τεχνολογίες που θα μπορούσαν να χρησιμοποιηθούν, έγιναν στην πορεία πολλά μικρά project με διάφορες τεχνολογίες όπως React, η Angular, η Vue για το GUI και η .NET για το backend. Όλες θα έβγαζαν το ίδιο αποτέλεσμα οπτικά και σαν λειτουργικότητα, η επιλογή των τεχνολογιών έγινε για λόγους προσωπικής προτίμησης.

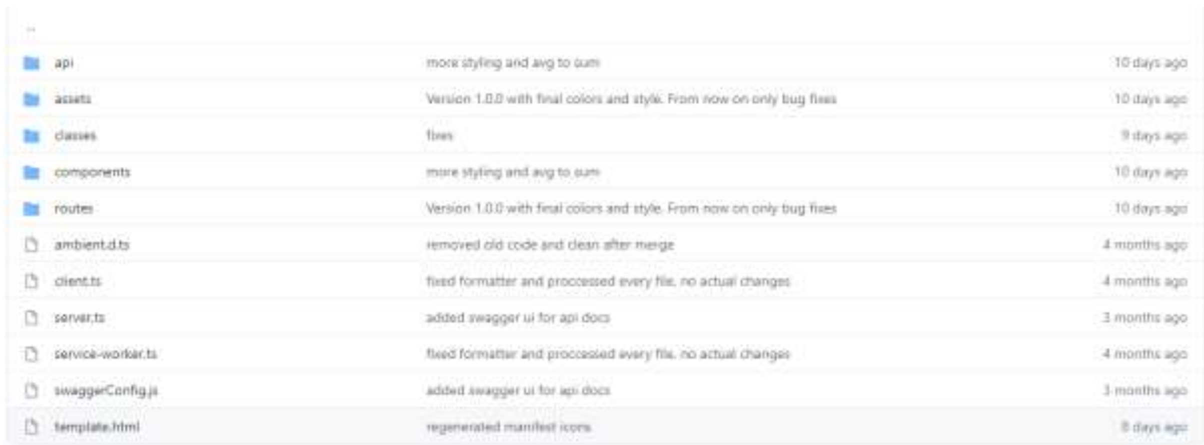
Στον okeano μου δόθηκαν δύο VM στα οποία έπρεπε να χωρέσουν όλες οι λειτουργίες. Τρέχουν και τα δύο Ubuntu 20.04, με NodeJS 12 και ήταν πλήρως αναβαθμισμένα κατά την διάρκεια της χρήσης τους.

Παράρτημα: Δόμηση κώδικα και git Repo

.github/workflows	fixing a bug with bcrypt deployment	4 months ago
.vscode	imported data and more ui fixes	last month
data_import	imported data and more ui fixes	last month
database	imported data and more ui fixes	last month
src	regenerated manifest icons	8 days ago
static	regenerated manifest icons	8 days ago
.env.sample	Proper Auth flow impl and some routing implemented	4 months ago
.eslintignore	fixed formatter and proccessed every file, no actual changes	4 months ago
.eslintrc.js	fixed formatter and proccessed every file, no actual changes	4 months ago
.gitignore	imported data and more ui fixes	last month
.prettierignore	fixed formatter and proccessed every file, no actual changes	4 months ago
.prettierrc	fixed formatter and proccessed every file, no actual changes	4 months ago
README.md	removed old code and clean after merge	4 months ago
package-lock.json	fixed diagram and buttons	last month
package.json	Version 1.0.0 with final colors and style. From now on only bug fixes	10 days ago
postcss.config.js	fixed formatter and proccessed every file, no actual changes	4 months ago
rollup.config.js	fixed diagram and buttons	last month
svelte.config.js	Selection component now interacts with the graph	4 months ago
tailwind.config.js	Version 1.0.0 with final colors and style. From now on only bug fixes	10 days ago
tsconfig.json	Selection component now interacts with the graph	4 months ago

Εικόνα 74Κώδικας στο VCS

Το project είχε πολλά μέλη οπότε είναι καλό να δομηθεί με έναν τρόπο που να είναι εύκολη η διαχείριση και η συντήρηση του project. Ό τρόπος που το έκανε είναι ότι όλος ο κώδικας ζει μέσα στο φάκελο "**src**", όλα τα αρχεία που θέλω απλά να είναι διαθέσιμα ζουν στο φάκελο "**static**", όλα τα script που δημιουργούν την βάση βρίσκονται στο φάκελο "**database**", διάφορα "**python script**" που βοήθησαν στο να μπουν δεδομένα στην βάση ζουν στο "**data_import**". Τα υπόλοιπα αρχεία δεν περιέχουν κώδικα αλλά ρυθμίσεις για τα διάφορα κομμάτια του development environment. Για παράδειγμα το ".eslintrc.js" ρυθμίζει τον linter που έρχεται μαζί με το project και λέγεται Eslint, ο φάκελος ".vscode" περιέχει τις ρυθμίσεις του IDE που χρησιμοποίησα και λέγετε "**VS Code**". Ο φάκελος που έχει περετέρω ενδιαφέρον είναι ο φάκελος που έχει τον πηγαίο κώδικα.



Εικόνα 75:src

Εδώ ζει και το API server και ο κώδικας που τρέχει στον υπολογιστή του χρήστη. Το αρχείο "server.ts" είναι το αρχείο που αποτελεί το entry point για το API, το "client.ts" είναι το ίδιο αλλά για τον χρήστη. Ενδιαφέρον έχει το αρχείο "template.html" το οποίο περιέχει βασικό σκελετό της ιστοσελίδα. Στην πράξη το sapper παίρνει αυτό το αρχείο και ανάλογα με το route προσθέτει το κατάλληλο σώμα. Τα "routes" ζούν μέσα στο φάκελο route και ανάλογα με το όνομα το Sapper τα χρησιμοποιεί. Ο φάκελος assets περιέχει αρχεία styling για να μπορεί να γίνει global styling στην εφαρμογή. Στο φάκελο classes ζουν τύποι, αντικείμενα και λογική γραμμένη σε typescript που χρησιμοποιείται από διάφορα components και διάφορα μέρη της ιστοσελίδας. Στο φάκελο components συναντάς τα κομμάτια του site που είναι κοινά για όλο το site, για παράδειγμα το footer, η Μπάρα στην κορυφή με το κουμπί για να συνδεθεί κάποιος και γενικά όλα τα κομμάτια κώδικα που μπορούσαν να επαναχρησιμοποιηθούν και ήταν γραμμένα σε Svelte.

src/api

Εικόνα 76: src/api/routes

Στο φάκελο API ζουν οι συναρτήσεις που αποτελούν την υλοποίηση των διαφόρων APIs. Όπως φαίνεται και στην δεύτερη εικόνα το κάθε API είναι ξεχωριστό αλλά επειδή τα έχω συνδέσει στο ίδιο repo μπορούν και μοιράζονται τις βοηθητικές συναρτήσεις μαζί με τα DTOs που ζουν στο Model.

Βιβλιογραφία

1] S. &. S. E. S. C. o. t. I. C. Society, «ISO/IEC/IEEE 42010:2011(E), Systems and software engineering — Architecture description,» 2011-12.

2] I. C. Education, "Three-Tier Architecture," IBM, 28 10 2020. [Online]. Available: https://www.ibm.com/cloud/learn/three-tier-architecture. [Accessed 7 7 2021].

3] P. A. Morgan Bruce, Microservices in Action, Manning Publications Co, 2019.

4] N. B. Π.Τσανάκας, *Ntua - Τεχνολογίες Υπηρεσιών Λογισμικού,* Athens, 2021.

5] C. K. Rudrabhatla, «Comparison of Event Choreography and Orchestration Techniques in Microservice Architecture,» (IJACSA) International Journal of Advanced Computer Science and Applications,, Los Angeles, USA, 2018.

6] C. D. Nguyen, «A Design Analysis of Cloud-based Microservices Architecture at Netflix,» 1 5 2020. [Ηλεκτρονικό]. Available: https://medium.com/swlh/a-design-analysis-of-cloud-based-microservices-architecture-at-netflix-98836b2da45f.

7] A. Dascalu, «SOAP vs REST vs gRPC vs GraphQL,» 5 5 2021. [Ηλεκτρονικό]. Available: https://dev.to/andreidascalu/soap-vs-rest-vs-grpc-vs-graphql-1ib6.

8] «XML,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/XML.

9] «JSON,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/JSON.

10] «Introducing JSON,» [Ηλεκτρονικό]. Available: https://www.json.org/json-en.html.

11] «Protocol Buffers,» [Ηλεκτρονικό]. Available: https://developers.google.com/protocol-buffers/docs/overview.

12] «OpenAPI Specification #History,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/OpenAPI_Specification#:~:text=9%20External%20links-,History,Reverb%20Technologies%2C%20Wordnik's%20parent%20company.&text=In%20July%202017%2C%20the%20OpenAPI%20Initiative%20released%20version%203.0..

13] «RPC,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Remote_procedure_call.

14] «GraphQL,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/GraphQL.

15] «GraphiQL Explorer,» SpaceX, [Ηλεκτρονικό]. Available: https://api.spacex.land/graphql/.

16] «Δημοσιοποίηση χαρτών και χαρτογραφικών δεδομένων,» [Ηλεκτρονικό]. Available: https://repository.kallipos.gr/bitstream/11419/2518/1/13_chapter_12.pdf.

17] «WFS reference -- GeoServer 2.19.x User Manual,» Geoserver, [Ηλεκτρονικό]. Available: https://docs.geoserver.org/master/en/user/services/wfs/reference.html.

[18] «Render millions of chart points with the Boost Module,» Highcharts, [Ηλεκτρονικό]. Available: https://www.highcharts.com/tutorials/highcharts-high-performance-boost-module/.

[19] [Ηλεκτρονικό]. Available: https://www.highcharts.com/demo/line-boost.

[20] «Express - Node.js web application framework,» [Ηλεκτρονικό]. Available: https://expressjs.com/.

[21] [Ηλεκτρονικό]. Available: https://svelte.dev/.

[22] «Rollup,» [Ηλεκτρονικό]. Available: https://rollupjs.org/guide/en/.

[23] N. Nalakath, «Module Bundlers in 5 Minutes — the What, the Why, and the Which,» [Ηλεκτρονικό]. Available: https://betterprogramming.pub/javascript-module-bundlers-2a1e9307d057.

[24] A. Gimeno, «How JavaScript bundlers work,» [Ηλεκτρονικό]. Available: https://medium.com/@gimenete/how-javascript-bundlers-work-1fc0d0caf2da.

[25] «Virtual DOM and Internals,» React, [Ηλεκτρονικό]. Available: https://reactjs.org/docs/faq-internals.html.

[26] rajaraodv, «Two Quick Ways To Reduce React App's Size In Production,» Medium, 2016. [Ηλεκτρονικό]. Available: https://rajaraodv.medium.com/two-quick-ways-to-reduce-react-apps-size-in-production-82226605771a.

[27] S. Gooding, «State of CSS 2020 Survey Results: Tailwind CSS Wins Most Adopted Technology, Utility-First CSS on the Rise,» 2020. [Ηλεκτρονικό]. Available: https://wptavern.com/state-of-css-2020-survey-results-tailwind-css-wins-most-adopted-technology-utility-first-css-on-the-rise.

[28] «Tailwind Preflight,» [Ηλεκτρονικό]. Available: https://tailwindcss.com/docs/preflight.

[29] «Tailwind Documentation,» [Ηλεκτρονικό]. Available: https://tailwindcss.com/docs.

[30] «Mapbox Pricing,» [Ηλεκτρονικό]. Available: https://www.mapbox.com/pricing/.

[31] D. P. a. S. J. Armando Fox, Engineering Software as a Service: An Agile Approach Using Cloud ComputingSecond Edition, 2.0a1, 2020.

[32] R. B. M. K. D. F. H. Judith Hurwitz, Service Oriented Architecture for Dummies 2nd IBM Limited Edition, Wiley Publishing, Inc., 2009.

[33] W3C, «XML Technology,» [Ηλεκτρονικό]. Available: https://www.w3.org/standards/xml/.

[34] R. Thurlow, « RPC: Remote Procedure Call Protocol Specification Version 2,» 5 2009. [Ηλεκτρονικό]. Available: https://datatracker.ietf.org/doc/html/rfc5531.

[35] «Docs,» [Ηλεκτρονικό]. Available: https://docs.geoserver.org/latest/en/user/.