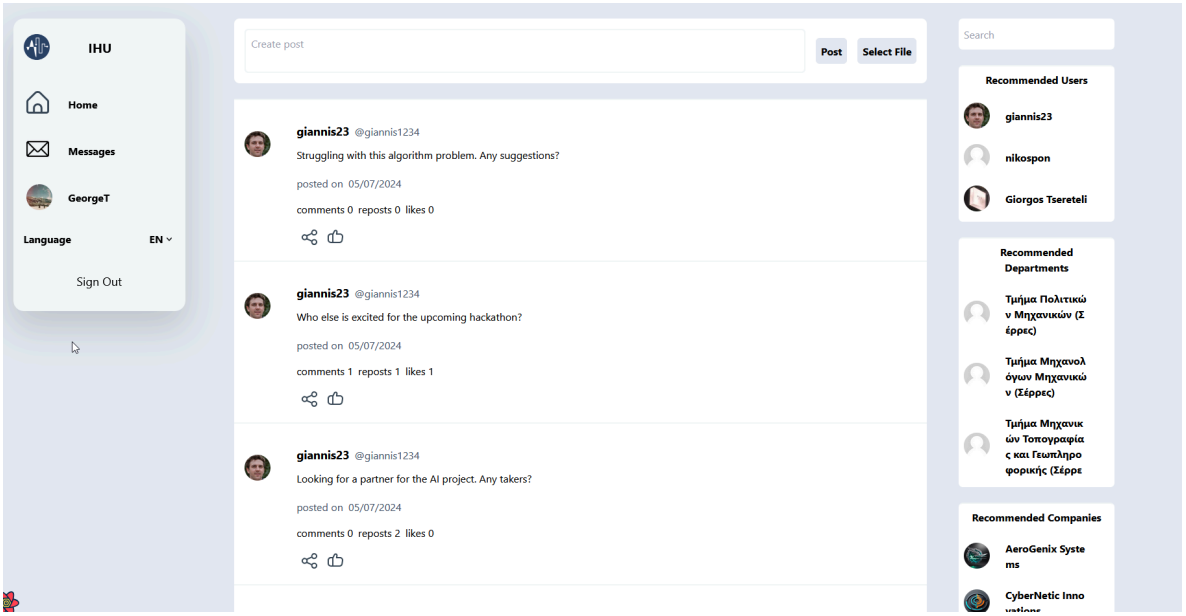
ΔΙΕΘΝΕΣ ΠΑΝΕΠΙΣΤΗΜΙΟ ΤΗΣ ΕΛΛΑΔΟΣ

ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΗΛΕΚΤΡΟΝΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

«Δημιουργία εφαρμογής κοινωνικής δικτύωσης για τις υπηρεσίες του τμήματος»



Φοιτητής

Γκιόργκι Τσέρετελι 164840

Επιβλέπων

Δρ. Κυριάκος Τσιακμάκης

Σεπτέμβριος 2024

Δημιουργία εφαρμογής κοινωνικής δικτύωσης για τις υπηρεσίες του τμήματος

Κωδικός: 23198

Φοιτητής: Τσέρετελι Γκίοργκι

Εισηγητής: Δρ. Κυριάκος Τσιακμάκης

Ημερομηνία ανάληψης Π.Ε. 30-03-2023

Ημερομηνία περάτωσης Π.Ε. 10-09-2024

Βεβαιώνω ότι είμαι ο συγγραφέας αυτής της εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης, έχω καταγράψει τις όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών, εικόνων και κειμένου, είτε αυτές αναφέρονται ακριβώς είτε παραφρασμένες. Επιπλέον, βεβαιώνω ότι αυτή η εργασία προετοιμάστηκε από εμένα προσωπικά, ειδικά ως πτυχιακής εργασία, στο Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του ΔΙ.ΠΑ.Ε.

Η έγκριση της πτυχιακής εργασίας από το Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του Διεθνούς Πανεπιστημίου της Ελλάδος, δεν υποδηλώνει απαραιτήτως και αποδοχή των απόψεων του συγγραφέα, εκ μέρους του Τμήματος.

Πρόλογος

Ο λόγος που επέλεξα αυτό το project ήταν επειδή πρόκειται για μια web εφαρμογή και με ενδιαφέρουν οι τεχνολογίες web, η κατασκευή αυτής της εφαρμογής μου επιτρέπει να μάθω ένα ευρύ φάσμα δεξιοτήτων, front-end γλώσσες και frameworks, back-end frameworks και δημιουργία ενός REST API και πώς να αλληλεπιδρώ με μια βάση δεδομένων. Επιπλέον, αυτή είναι μια εφαρμογή που θα μπορούσε να χρησιμοποιηθεί όντως για το πανεπιστήμιο και να αναπτυχθεί και να βελτιωθεί από άλλους φοιτητές στις δικές τους εργασίες.

Περίληψη

Το αντικείμενο της παρούσας πτυχιακής εργασίας είναι η κατασκευή μιας διαδικτυακής εφαρμογής, συγκεκριμένα μιας πλατφόρμας micro-blogging για το πανεπιστήμιο, ώστε οι φοιτητές να μπορούν να επικοινωνούν μεταξύ τους και να ενημερώνονται για τα νέα του πανεπιστημίου.

Η εφαρμογή είναι μια fullstack εφαρμογή κατασκευασμένη με JavaScript τόσο για το front-end όσο και για το back-end. Με τη βοήθεια διαφορετικών frameworks και με Firebase για την αυθεντικοποίηση και την αποθήκευση αρχείων και PostgreSQL για την αποθήκευση δεδομένων.

«Creation of a social networking application for the services of the department»

Abstract

The subject of this thesis is building a web application, specifically a micro-blogging platform for the university so that students can be able to communicate with each other and keep up with university news.

The application is a fullstack application built with JavaScript for both the front-end and the back-end. With the help of different frameworks and with Firebase for authentication and file storage and PostgreSQL for data storage.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τη μητέρα μου που με στήριξε όλα αυτά τα χρόνια.

Περιεχόμενα

Κεφάλαιο 1ο: Τεχνολογίες

Η εφαρμογή αποτελείται από δύο μέρη, την Web App: (Frontend) που είναι φτιαγμένο με το React, ένα framework της JavaScript, το Next.js, ένα framework που είναι χτισμένο πάνω στο react και τον server (Backend) που είναι φτιαγμένο με το Node.js ένα runtime environment για την κατασκευή web servers, το Express.js ένα framework του Node.js, το Prisma έναν object relational mapper (ORM) για την επικοινωνία με την PostgreSQL, μια σχεσιακή βάση δεδομένων . Η εφαρμογή χρησιμοποιεί επίσης το Firebase, ένα Backend as a Service (BaaS) που αναπτύχθηκε από την Google, για την αυθεντικοποίηση των χρηστών και την αποθήκευση αρχείων.

Στο παρακάτω κεφάλαιο θα κάνουμε μια επισκόπηση των τεχνολογιών που χρησιμοποιήθηκαν για την κατασκευή αυτού του πρότζεκτ.

1.1 React

Το React είναι ένα framework javascript για την κατασκευή διεπαφών χρήστη. Δημιουργήθηκε και συντηρείται από το facebook και διαθέτει ένα μεγάλο οικοσύστημα βιβλιοθηκών ανοιχτού κώδικα. Κατά την κατασκευή διεπαφών χρήστη με javascript, ο χειρισμός του html DOM γίνεται απευθείας. Το React χρησιμοποιεί components, τα οποία είναι συναρτήσεις που ενθυλακώνουν τόσο τον κώδικα html όσο και τη λογική ενός τμήματος του UI. Το React χρησιμοποιείται για τη δημιουργία εφαρμογών single-page (SPAs) όπου το περιεχόμενο ενημερώνεται δυναμικά χωρίς επαναφόρτωση ολόκληρης της σελίδας. Τα components του react σε μια εφαρμογή κατασκευάζονται ιεραρχικά, με κάθε component να είναι υπεύθυνο για την απεικόνιση (render) ενός τμήματος του ui.

1.1.1 Virtual dom

Το εικονικό DOM του React είναι μια ελαφριά αναπαράσταση του δέντρου DOM. Παρακολουθώντας τις διαφορές μεταξύ του εικονικού και του πραγματικού DOM, το React μπορεί να βελτιστοποιήσει τις ανανεώσεις (rerenders) του UI. [1]

1.1.2 JSX

Το JSX είναι μια επέκταση σύνταξης για τη JavaScript που επιτρέπει στους προγραμματιστές να γράφουν κώδικα HTML μέσα στη JavaScript. Το JSX χρησιμοποιείται μέσα σε components. [2]

1.1.3 Components

Τα components είναι μέρη του UI (γραμμένα ως απλές συναρτήσεις javascript), για παράδειγμα ένα κουμπί. Τα components μπορούν να δέχονται παραμέτρους (props) και γράφονται με JSX και μπορούν να είναι στατικά στοιχεία του UI ή μπορούν να είναι διαδραστικά με τη χρήση state και react

hooks. Τα components μπορούν να χρησιμοποιηθούν μέσα σε άλλα components, π.χ. το component button μπορεί να χρησιμοποιηθεί μέσα στο component Home page της ιστοσελίδας. Τα components διευκολύνουν την επαναχρησιμοποίηση, τη συντήρηση και την επέκταση μεγάλων βάσεων κώδικα.

1.1.4 State

Το State είναι η μνήμη των component, χρησιμοποιείται για να παρακολουθεί τις διάφορες τιμές στο component κατά τη διάρκεια των rerenders. Για παράδειγμα, ένα component που εμφανίζει μια φόρμα πρέπει να παρακολουθεί όλες τις τιμές των inputs, αυτό επιτυγχάνεται με τη δημιουργία μιας μεταβλητής state με το useState hook. Ο κώδικας μέσα στο component ενημερώνει την τιμή της μεταβλητής όταν αλλάζει η τιμή του input. [3]

1.1.5 Hooks

Τα hooks είναι συναρτήσεις που επιτρέπουν στα functional components να χρησιμοποιούν το state και άλλα features του React χωρίς να γράψετε ένα class component.

useState: επιτρέπει στα functional components να διαχειρίζονται το state.

Η useEffect μας επιτρέπει να αλληλεπιδρούμε και να συγχρονιζόμαστε με ένα εξωτερικό σύστημα, για παράδειγμα να αλληλεπιδρούμε με το API του φυλλομετρητή ή να παίρνουμε κάποια δεδομένα, εκτελείται μετά το πρώτο rendering του component και στη συνέχεια εκτελείται μετά από κάθε επανα-rendering εξ ορισμού. Η useEffect αποτελείται από δύο μέρη, τη συνάρτηση που εκτελεί τον κώδικά μας και τη λίστα εξαρτήσεων (dependency array). Αν θέλουμε να εκτελέσουμε τον κώδικα μόνο μία φορά, πρέπει να παρέχουμε έναν άδειο πίνακα εξαρτήσεων, αυτό θα εκτελέσει τη συνάρτηση όταν το component rendering γίνει για πρώτη φορά. Αν το useEffect μας εξαρτάται από κάποια τιμή/τιμές και θέλουμε να εκτελείται κάθε φορά που αλλάζουν αυτές οι τιμές, μπορούμε να προσθέσουμε αυτές τις τιμές στον πίνακα εξαρτήσεων. Το useEffect θα εκτελεστεί εάν κάποια από τις τιμές στον πίνακα εξαρτήσεων αλλάξει μετά από ένα render. [4]

useContext: το useContext επιτρέπει στην εφαρμογή μας να έχει μια γενική μεταβλητή state. Αυτό είναι χρήσιμο σε περιπτώσεις όπου πρέπει να κάνουμε τη μεταβλητή state μας χρησιμοποιήσιμη σε πολλά μέρη, για παράδειγμα στις πληροφορίες ενός συνδεδεμένου χρήστη. Αντί να περνάμε τη μεταβλητή από component σε component με χρήση props, μπορούμε να την αποθηκεύσουμε σε ένα context, καθιστώντας την διαθέσιμη για κάθε component. [5]

useReducer: είναι ένα hook του React για τη διαχείριση σύνθετων state, είναι μια εναλλακτική λύση αντι το useState που είναι πιο κατάλληλο για απλά state. Components με πολλές ενημερώσεις state είναι δύσκολο να διαχειριστούν, για παράδειγμα μια σύνθετη φόρμα. Για αυτές τις περιπτώσεις, μπορούμε να μεταφέρουμε όλη τη λογική ενημέρωσης του state έξω από το component μας σε μία μόνο συνάρτηση, αυτή η συνάρτηση κάνει διαφορετικά πράγματα ανάλογα με τον «τύπο ενέργειας» που λαμβάνει. Αυτοί οι τύποι ενέργειας ορίζονται από τον προγραμματιστή, για παράδειγμα, ο τύπος ενέργειας «login» θα αποθήκευε τα δεδομένα των χρηστών σε ένα γενικό context για να χρησιμοποιηθούν από την εφαρμογή, ένας τύπος ενέργειας «logout» θα έκανε το state null. [6]

1.2 Node.js

Το Node.js είναι ένα περιβάλλον εκτέλεσης (runtime environment) της JavaScript ανοικτού κώδικα και cross-platform που εκτελείται με τη V8 JavaScript engine ανοικτού κώδικα. [7]

Αρχικά, η javascript χρησιμοποιούνταν για τη σύνταξη front-end κώδικα σε web browsers και οι προγραμματιστές έπρεπε να μάθουν μια διαφορετική γλώσσα για να γράψουν backend κώδικα για τον server τους, το node επιτρέπει την εκτέλεση της JavaScript έξω από ένα web browser, επομένως ένας προγραμματιστής που γνωρίζει javascript μπορεί πλέον να γράψει τόσο front-end όσο και backend κώδικα.

Με το node.js μπορούμε να δημιουργήσουμε web servers και εργαλεία γραμμής εντολών. Το Node.js έρχεται με τη δική του τυποποιημένη βιβλιοθήκη συναρτήσεων για την αλληλεπίδραση με το σύστημα αρχείων, τη δικτύωση, τη ροή δεδομένων και πολλά άλλα.

1.3 NPM

Το Npm είναι ο διαχειριστής πακέτων του Node. Χρησιμοποιείται για τη δημοσίευση και τη λήψη βιβλιοθηκών άλλων προγραμματιστών για το οικοσύστημα javascript. [8] Όλες οι εξαρτήσεις ενός πρότζεκτ εγκαθίστανται στα αρχεία node_modules.

Υπάρχουν εναλλακτικοί διαχειριστές πακέτων, όπως το Yarn και το pnpm.

1.4 Express.js

Το Express.js είναι ένα framework για το Node.js. Το Express παρέχει πολλές δυνατότητες που το Node δεν παρέχει από μόνο του. [9].

1.4.1 Routing:

Τα routes/endpoints είναι διαφορετικά σημεία URL με μια συγκεκριμένη μέθοδο αίτησης HTTP (GET,POST,PUT,DELETE και άλλα) όταν ταυτίζονται με το route, εκτελείται μια συνάρτηση που εκτελεί την αντίστοιχη εργασία ανάλογα με τη μέθοδο HTTP. [10]

Για παράδειγμα, ένα αίτημα GET επιστρέφει δεδομένα από τη βάση δεδομένων και ένα αίτημα POST προσθέτει δεδομένα στη βάση δεδομένων.
Οι συναρτήσεις δρομολόγησης λαμβάνουν εξ ορισμού 2 παραμέτρους το αίτημα και το αντικείμενο απόκρισης.

Το αντικείμενο request έχει πληροφορίες σχετικές με το αίτημα που λαμβάνει ο διακομιστής, όπως το σώμα του αιτήματος, τις παραμέτρους του ερωτήματος και άλλα. [11]
Το αντικείμενο απόκρισης αντιπροσωπεύει την απάντηση HTTP που θα στείλουμε στον client. [12]

1.4.2 Middleware:

Το middleware είναι συναρτήσεις που εκτελούνται πριν από τη συνάρτηση του endpoint που καλέσαμε. Μπορεί να έχει πρόσβαση και να τροποποιεί τα αντικείμενα αίτησης και απόκρισης της συνάρτησης δρομολόγησης (router). Έχει επίσης πρόσβαση στη συνάρτηση next η οποία επιτρέπει στο middleware να εκτελέσει το επόμενο middleware ή τη συνάρτηση δρομολόγησης αν δεν υπάρχει άλλο middleware. [13]

Κάθε endpoint μπορεί να έχει ένα ή περισσότερα middleware για εξουσιοδότηση, επικύρωση εισερχόμενων δεδομένων και άλλα.

Το middleware μπορεί να σταματήσει την αίτηση από το να φτάσει στη συνάρτηση δρομολόγησης, για παράδειγμα, αν έχουμε ένα middleware για τον έλεγχο ταυτότητας του χρήστη και ο χρήστης είναι μη έγκυρος, μπορούμε να "πετάξουμε" (throw) ένα σφάλμα μέσα στο middleware, πράγμα που σημαίνει ότι ο client θα λάβει ένα σφάλμα και η συνάρτηση δρομολόγησης δεν θα εκτελεστεί ποτέ.

1.4.3 Serving static files:

Μπορούμε να στείλουμε στατικά αρχεία όπως εικόνες, αρχεία pdf και άλλα. [14]

1.5 Next.js

Το Next.js είναι ένα full-stack framework React, χτισμένο πάνω στο Nodejs. Το Next.js χρησιμοποιεί το React για την κατασκευή των components του. Επιτρέπει στους προγραμματιστές να δημιουργούν server-side rendered (SSR) και statically generated sites (SSG). [15]

Μερικά από τα χαρακτηριστικά του Next.js είναι
Λειτουργίες δρομολόγησης, rendering, data fetching functions , αυτόματες βελτιστοποιήσεις

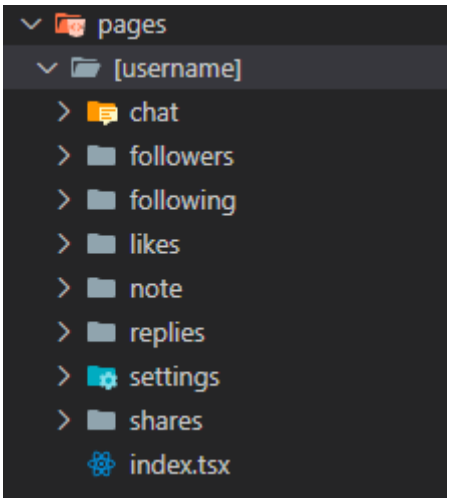
1.5.1 Routing

Σε αντίθεση με το React, το Next.js παρέχει έναν ενσωματωμένο δρομολογητή. Στο react θα πρέπει να εγκαταστήσετε μια εξωτερική βιβλιοθήκη, που ονομάζεται react-router. Το Next.js παρέχει έναν δρομολογητή βασισμένο σε αρχεία, όπου κάθε αρχείο/φάκελος αντιπροσωπεύει μια σελίδα στον ιστότοπό μας. [16]
Στη δομή των φακέλων του πρότζεκτ που δημιουργείται από το Next.js έχουμε ένα φάκελο pages όπου βρίσκονται όλες οι διαδρομές για τις σελίδες μας. Μπορούμε να δημιουργήσουμε διαδρομές για τις σελίδες μας με δύο διαφορετικούς τρόπους, μπορούμε είτε να δημιουργήσουμε αρχεία .js, jsx, .ts ή .tsx στο φάκελο pages με το όνομα της σελίδας στην οποία θέλουμε να πλοηγηθεί.
Για παράδειγμα: pages/chat.tsx το οποίο θα μας πλοηγήσει στη σελίδα /chat

Ή μπορούμε να δημιουργήσουμε φακέλους με το όνομα της σελίδας μας και να βάλουμε ένα αρχείο με το όνομα index
Για παράδειγμα: pages/chat/index.tsx και αυτό θα μας πλοηγούσε στη σελίδα /chat

Σε αυτό το πρότζεκτ χρησιμοποιούμε μια προσέγγιση με βάση το φάκελο/index.tsx

Μπορούμε να δημιουργήσουμε δυναμικές διαδρομές, για παράδειγμα μπορεί να θέλουμε ένα επαναχρησιμοποιήσιμο component που να εμφανίζει το προφίλ του χρήστη στην εφαρμογή μας, μπορούμε να το κάνουμε αυτό τυλίγοντας το όνομα του φακέλου μας σε αγκύλες []



Σχήμα 1.1: φάκελος με τις σελίδες του χρήστη

Στην εικόνα του παραδείγματος, έχουμε ένα φάκελο που παίρνει το όνομα χρήστη στη σελίδα των χρηστών δυναμικά, και το στοιχείο που κάνει render το UI για κάθε χρήστη είναι μέσα στο αρχείο index.tsx, στο πρόγραμμα περιήγησης η διαδρομή θα είναι απλά «https://example.com/user1»

Μπορούμε να δημιουργήσουμε εμφωλευμένες διαδρομές με την εμφωλευμένη δομή των φακέλων μας

Για παράδειγμα: pages/[όνομα χρήστη]/settings/index.tsx μας πλοηγεί στη σελίδα ρυθμίσεων του χρήστη που περάσαμε δυναμικά. Η διεύθυνση URL θα είναι «https://example.com/user1/settings»

1.5.2 Layouts

Στο Next.js μπορούμε να δημιουργήσουμε διατάξεις (layouts) για την εφαρμογή μας, ένα συγκεκριμένο Layout για ολόκληρη την εφαρμογή ή/και πολλαπλά Layouts για διαφορετικές σελίδες. [16]

Shared Layout

Για να δημιουργήσουμε μια ενιαία κοινή διάταξη για ολόκληρη την εφαρμογή δημιουργούμε ένα component που περιέχει τη διάταξη που θέλουμε, και στη συνέχεια τοποθετούμε το child prop μέσα στο τμήμα της HTML όπου θέλουμε να εμφανίζεται μέσα στη διάταξή μας. Το child prop είναι το κύριο component που εμφανίζει τη σελίδα μας μέσα στον δρομολογητή των σελίδων.

```
import Navbar from './navbar'
import Footer from './footer'

export default function Layout({ children }) {
  return (
        <>
        <Navbar />
        <main>{children}</main>
        <Footer />
        </>
  )
}
```

Παράδειγμα: Έχουμε ένα component Layout που έχει τη γενική διάταξη της εφαρμογής μας, παίρνει τα παιδιά prop και τα εμφανίζει μέσα στο main div. [16]

Στη συνέχεια, καλούμε το Layout component μέσα στο _app.tsx το οποίο χρησιμοποιείται για την αρχικοποίηση των σελίδων. [16]

```
import Layout from '../components/layout'

export default function MyApp({ Component, pageProps }) {
  return (
        <Layout>
        <Component {...pageProps} />
        </Layout>
  )
}
```

Παράδειγμα: Το component Layout είναι τυλιγμένο γύρω από ολόκληρη την εφαρμογή μας (<Component {...pageProps}>)

Per Page Layout

Για να δημιουργήσουμε μια διάταξη ανά σελίδα προσθέτουμε μια ιδιότητα getLayout στη σελίδα.

```
import Layout from '../components/layout'
import NestedLayout from '../components/nested-layout'

export default function Page() {
  return (

  )
}

Page.getLayout = function getLayout(page) {
  return (
        <Layout>
        <NestedLayout>{page}</NestedLayout>
        </Layout>
  )
}
```

Παράδειγμα: Εχουμε το component Page, κάτω από αυτό προσθέτουμε την ιδιότητα (property) getLayout στο component Page όπου ορίζουμε ότι το component Page μας (παράμετρος page μέσα στο getLayout) θα πρέπει να περάσει ως παιδί στο component Per Page Layout μας.

Στη συνέχεια, στο _app.tsx καλούμε τη συνάρτηση getLayout. [16]

```
export default function MyApp({ Component, pageProps }) {
  // Use the layout defined at the page level, if available
  const getLayout = Component.getLayout ?? ((page) => page)

  return getLayout(<Component {...pageProps} />)
}
```

1.5.3 Linking and Navigating

Το Next.js παρέχει ένα ενσωματωμένο component συνδέσμου για πλοήγηση μεταξύ σελίδων, δεν επαναφορτωνει τον ιστότοπο σε αντίθεση με την ετικέτα <a> και εκτελεί πρόσθετες βελτιστοποιήσεις στο παρασκήνιο. [17]

1.5.4 Redirecting

Μπορούμε να χρησιμοποιήσουμε τη μέθοδο .push του useRouter για να ανακατευθύνουμε τον χρήστη σε μια άλλη σελίδα όταν κάνει κλικ σε ένα κουμπί. [18]

```
import { useRouter } from 'next/router'

export default function Page() {
  const router = useRouter()

  return (
       <button type="button" onClick={() => router.push('/dashboard')}>
       Dashboard
       </button>
  )
}
```

Παράδειγμα: Χρησιμοποιούμε τη μέθοδο .push για να ανακατευθύνουμε τον χρήστη στη σελίδα /dashboard όταν κάνει κλικ στο κουμπί. [18]

1.5.5 RENDERING

Το Next.js μας παρέχει μεθόδους για να παίρνουμε δεδομένα κατά τη δημιουργία εφαρμογών Server-side Rendered (SSR) και Static Site Generator (SSG) εκτός από το rendering από την πλευρά του client, αλλά πριν από αυτό ας δούμε τι είναι το SSR και το SSG.

SSR

Το Server-side rendering είναι μια μέθοδος rendering του UI στην εφαρμογή μας. Στην SSR, ο διακομιστής είναι υπεύθυνος για τη δημιουργία της HTML που θα εμφανίσει ο ιστότοπος και την αποστολή της στον client. [23]

Το κύριο πλεονέκτημα αυτής της μεθόδου είναι ότι το πρόγραμμα περιήγησης δεν εκτελεί πολύ κώδικα JavaScript για να εμφανίσει τον ιστότοπο, καθιστώντας την περιήγηση του χρήστη ταχύτερη.

SSG

Η Static Site Generation είναι μια τεχνική που χρησιμοποιείται για ιστότοπους όπου τα δεδομένα είναι στατικά και δεν αλλάζουν. Σε αυτή την τεχνική η εφαρμογή γίνεται render κατά τη διάρκεια της διαδικασίας κατασκευής (building process) , με τη λήψη των απαραίτητων δεδομένων από το backend και τη δημιουργία στατικών αρχείων HTML για κάθε σελίδα. [24]

CSR

Στο client side rendering, δημιουργούμε το περιεχόμενο HTML της σελίδας στον client, και οι σελίδες γίνονται rendered με τα δεδομένα σε κάθε αίτημα fetch που κάνουμε. Το μειονέκτημα αυτής της τεχνικής είναι ότι πρέπει να περιμένουμε το φυλλομετρητή να κατεβάσει την αρχική html και το javascript.

1.5.6 DATA FETCHING

Για την SSG μπορούμε να χρησιμοποιήσουμε τη συνάρτηση getStaticProps, όπου μπορούμε να καλέσουμε το API μας για να λάβουμε τα δεδομένα που χρειαζόμαστε για να εμφανιστούν. Το Next.js προ-προβάλλει (pre-render) τη σελίδα κατά τη στιγμή της δημιουργίας. [20]

Για SSR μπορούμε να χρησιμοποιήσουμε τη συνάρτηση getServerSideProps. Αυτή η συνάρτηση εκτελείται σε κάθε αίτηση και τα δεδομένα που επιστρέφονται από αυτή τη συνάρτηση εμφανίζονται στο component Page. Αυτή η συνάρτηση εκτελείται στον διακομιστή. [19]

Σε αυτή την εφαρμογή δεν χρησιμοποιούμε SSR ή SSG οπότε χρησιμοποιούμε τη βασική μέθοδο λήψης δεδομένων από την πλευρά του client. Για το σκοπό αυτό χρησιμοποιούμε τη βιβλιοθήκη React Query η οποία κάνει την λήψη δεδομένων πολύ καλύτερη, φέρνει δεδομένα κατά τη φόρτωση του component αυτόματα, χωρίς να χρειάζεται να χρησιμοποιήσουμε το useEffect hook, κάνει caching και πολλά άλλα.

1.5.7 Optimising

Το Next.js κάνει κάποιες αυτόματες βελτιστοποιήσεις, μερικά παραδείγματα:
Εικόνες: αλλάζει το μέγεθος των εικόνων για διαφορετικές συσκευές, κάνει lazy loading τις εικόνες και πολλά άλλα. [21]
Εάν η εφαρμογή χρησιμοποιεί assets στο φάκελο /public μπορούμε απλά να τα χρησιμοποιήσουμε στον κωδικα μας κάνοντας αναφορά στο όνομα της διαδρομής τους

```
<NavbarButtons
  icon={"/home.png"}
  text={"Home"}
  alt={"Home Icon"}
></NavbarButtons>;
```

Για παράδειγμα, στον παραπάνω κώδικα, η διαδρομή του «/home.svg» είναι public/home.svg, μπορούμε απλά να αναφερθούμε στο μέρος μετά το public για να το χρησιμοποιήσουμε οπουδήποτε χωρίς να εισάγουμε την εικόνα πάνω από τον κώδικα

1.5.7 API Routes

Όπως αναφέρθηκε, το Next.js είναι ένα fullstack framework που είναι χτισμένο πάνω στο node.js, εξαιτίας αυτού, το Next.js μας επιτρέπει επίσης να δημιουργήσουμε ένα API απευθείας στο Next.js project μας, χρησιμοποιώντας το φάκελο api μέσα στο φάκελο pages. [22] Αυτά τα endpoints του API λειτουργούν ως serverless functions. Οι serverless functions είναι endpoints που λειτουργούν κατά παραγγελία, σε αντίθεση με τα API που τρέχουν σε διακομιστές οι οποίοι είναι ενεργά όλη την ώρα, τα serverless endpoints είναι εκτός σύνδεσης και «τρέχουν» όταν τους γίνεται ένα αίτημα. Ένα μειονέκτημα των serverless functions είναι η λεγόμενη ψυχρή εκκίνηση (cold start), επειδή ο διακομιστής είναι αδρανής πριν από την υποβολή του αιτήματος, υπάρχει μια καθυστέρηση μερικών δευτερολέπτων πριν ο διακομιστής ξεκινήσει και αρχίσει να χειρίζεται το αίτημα.

1.5.8 TypeScript Support

Nextjs. Παρέχει ενσωματωμένη υποστήριξη TypeScript.

1.6 TypeScript

Η TypeScript είναι μια γλώσσα προγραμματισμού που αναπτύσσεται και συντηρείται από τη Microsoft και αποτελεί υπερσύνολο (superset) της JavaScript, δηλαδή JavaScript με πρόσθετα χαρακτηριστικά. [25]

Η TypeScript παρέχει ασφάλεια τύπων (type safety) στη JavaScript, η οποία βοηθάει στη διόρθωση σφαλμάτων κατά τη διάρκεια της ανάπτυξης.

Για παράδειγμα, θα μπορούσαμε να έχουμε αρχικοποιήσει μια μεταβλητή που ονομάζεται «age» με την τιμή 20, εμείς , ως ο προγραμματιστής που έγραψε αυτόν τον κώδικα, γνωρίζουμε ότι αυτή η μεταβλητή πρέπει να είναι τύπου αριθμού, αλλά τι γίνεται αν ένας άλλος προγραμματιστής έρθει λίγα χρόνια αργότερα και προσθέσει μια τιμή συμβολοσειράς σε αυτή τη μεταβλητή, ή τι γίνεται αν χρησιμοποιήσουμε αυτή τη μεταβλητή σε μια συνάρτηση και της περάσουμε με κάποιο τρόπο έναν άλλο τύπο μεταβλητής; Αυτοί οι τύποι σφαλμάτων θα ήταν δύσκολο να εντοπιστούν χωρίς το TypeScript να ελέγχει συνεχώς ότι οι τύποι που περνάμε στις μεταβλητές μας είναι σωστοί.

Το TypeScript παρέχει:

- βασικούς τύπους όπως αριθμός, συμβολοσειρά, boolean
- Custom types
- Utility Types για την τροποποίηση ή επέκταση των υφιστάμενων τύπων
- Τύπους για παραμέτρους συναρτήσεων και τύπους επιστροφής
- Generics

Και πολλά άλλα χαρακτηριστικά.

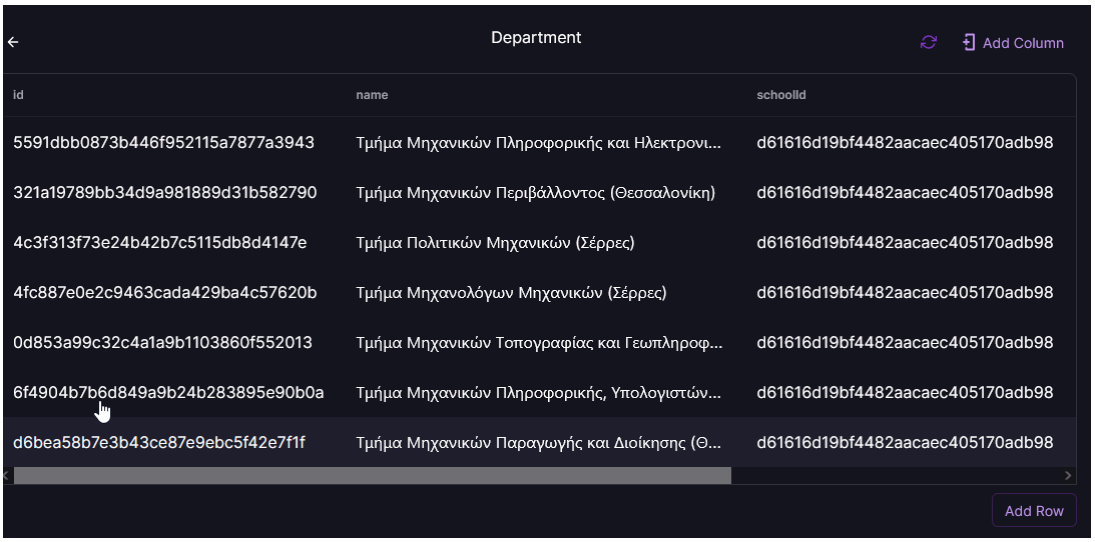
Ένα πρόσθετο χαρακτηριστικό που είναι πολύ χρήσιμο είναι ότι έχοντας καλά τυποποιημένες συναρτήσεις και αντικείμενα μας δίνει autocomplete.

Το TypeScript έχει σχεδιαστεί για να είναι εύκολο να προσαρμοστεί σταδιακά, μπορούμε να το κάνουμε αυτό μετονομάζοντας τα αρχεία μας .js σε .ts επέκταση αρχείου (ή .tsx για τα στοιχεία React / Next.js) και προσθέτοντας σιγά σιγά τύπους. Από προεπιλογή οι περισσότερες μεταβλητές, συναρτήσεις κ.λπ. είναι τύπου any , δηλαδή δεν έχουν συγκεκριμένο τύπο.

1.7 PostgresSQL

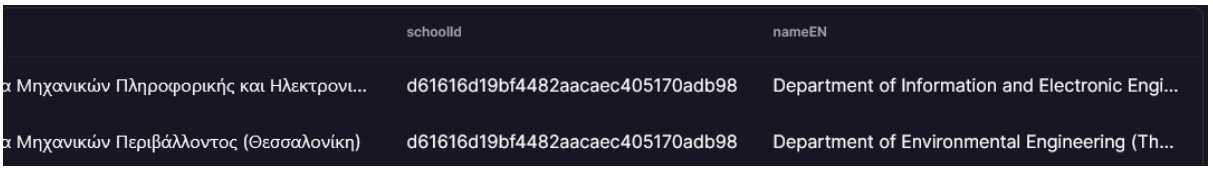
Η PostgresSQL είναι μια σχεσιακή βάση δεδομένων που χρησιμοποιεί τη γλώσσα SQL. [26]

Η σχεσιακή βάση δεδομένων είναι μια βάση δεδομένων όπου τα δεδομένα σχετίζονται μεταξύ τους. Στις σχεσιακές βάσεις δεδομένων έχουμε πίνακες, οι οποίοι αποθηκεύουν δεδομένα που σχετίζονται με μια από τις οντότητες της εφαρμογής μας, για παράδειγμα έναν πίνακα Users. Οι πίνακες έχουν ιδιότητες διαφόρων τύπων όπως varchar,int κ.λπ. Σε αυτούς τους πίνακες κάθε γραμμή είναι μια εγγραφή δεδομένων και κάθε στήλη είναι μια ιδιότητα αυτής της εγγραφής δεδομένων.



Σχήμα 1.2: Πίνακας Department με κάθε γραμμή να είναι μια εγγραφή και οι στήλες οι ιδιότητες

Η σχέση δημιουργείται από μια γραμμή ενός πίνακα που παραπέμπει σε ένα id μιας γραμμής ενός άλλου πίνακα.



Σχήμα 1.3: Κάθε εγγραφή του Department συνδέεται με μια εγγραφή στον πίνακα School μέσω της στήλης schoolId.

Το postgres έχει πολλές λειτουργίες, όπως ταυτόχρονη χρήση (Concurrency), υψηλές επιδόσεις, views, αποθηκευμένες διαδικασίες (stored procedures), triggers, ευρετηρίαση (Indexing) και πολλά άλλα.

Στην εφαρμογή μας χρησιμοποιούμε το Postgres λόγω της σχεσιακής φύσης των δεδομένων μας.

1.8 Prisma ORM

Το Prisma είναι ένας type safe Object Relational Mapper (ORM) που χρησιμοποιούμε για την αλληλεπίδραση με τη βάση δεδομένων PostgreSQL. [27]

Ένας ORM είναι ένας τρόπος αλληλεπίδρασης με τη βάση δεδομένων μας με αντικειμενοστραφή τρόπο. Ο ORM μας επιτρέπει να ορίσουμε το σχήμα της βάσης δεδομένων μας, να κάνουμε ερωτήματα στη βάση δεδομένων μας με έναν απλοποιημένο αντικειμενοστραφή τρόπο. Το Prisma μας επιτρέπει επίσης να κάνουμε εύκολα migrations.

1.8.1 Schema

Στο Prisma ορίζουμε τους πίνακες SQL ως αντικείμενα TypeScript.

```
datasource db {
  provider = "postgresql"
  url  = env("DATABASE_URL")
}

generator client {
  provider = "prisma-client-js"
}

model User {
  id           Int     @id @default(autoincrement())
  createdAt DateTime @default(now())
  email        String   @unique
  name         String?
  role         Role    @default(USER)
  posts        Post[]
}

model Post {
  id           Int     @id @default(autoincrement())
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
  published Boolean  @default(false)
  title        String   @db.VarChar(255)
  author       User?  @relation(fields: [authorId], references: [id])
  authorId  Int?
}
```

Σε αυτον τον κώδικα, το σχήμα prisma θα δημιουργήσει δύο πίνακες SQL, τους Post και User..

1.8.2 Querying

Αντί να γράφουμε ερωτήματα SQL, μπορούμε να καλούμε διάφορες μεθόδους στα αντικείμενά μας.

```
SELECT * FROM users id = 123456
```

```
const user = await prisma.user.findUnique({
  where: {
       id: '60d5922d00581b8f0062e3a8',
  },
})
```

Στο παραπάνω παράδειγμα, επιλέγουμε έναν χρήστη με βάση το ID τόσο στην SQL όσο και στο Prisma. Στο παράδειγμα της Prisma, η findUnique είναι μια ενσωματωμένη μέθοδος σε κάθε ένα από τα αντικείμενα/πίνακες μας. Μέσα στη μέθοδο έχουμε τη λέξη-κλειδί «where» η οποία μας επιτρέπει να επιλέξουμε τον χρήστη με βάση ένα πεδίο. Το Prisma έχει πολλές μεθόδους και πολλές διαφορετικές λέξεις-κλειδιά που μπορούμε να χρησιμοποιήσουμε μέσα στις μεθόδους μας, για φιλτράρισμα, ταξινόμηση, σελιδοποίηση, επιλογή συγκεκριμένων πεδίων και πολλά άλλα.

1.8.3 Migrations

Χρησιμοποιούμε migrations όταν αλλάζει το σχήμα της βάσης δεδομένων μας ή όταν μετακινούμαστε από μια βάση δεδομένων σε μια άλλη. Σε κάθε migration το Prisma δημιουργεί ένα αρχείο .sql που μας επιτρέπει να παρακολουθούμε το ιστορικό των αλλαγών του σχήματος της βάσης δεδομένων μας. Το Prisma διευκολύνει την κατασκευή πρωτοτύπων και τη δοκιμή διαφορετικών σχημάτων βάσεων δεδομένων. Μπορούμε να χρησιμοποιήσουμε την εντολή «migrate dev» για να χρησιμοποιήσουμε τα migrations κατά τη διάρκεια της ανάπτυξης και την εντολή «migrate deploy» για να μεταφέρουμε τα migrations στην production database μας.

1.9 Firebase

Η Firebase είναι μια πλατφόρμα για την ανάπτυξη εφαρμογών που συντηρείται από την Google. [28]

Παρέχει πολλές δυνατότητες για τη δημιουργία εφαρμογών:

- Firestore - μια βάση δεδομένων βασισμένη σε έγγραφα
- Cloud Functions - serverless functions
- Hosting για τις front-end εφαρμογές
- Αποθήκευση αρχείων
- Αυθεντικοποίηση

Και πολλά άλλα.

Στην εφαρμογή μας χρησιμοποιούμε το Firebase για την αυθεντικοποίηση και την αποθήκευση αρχείων.

1.9.1 Authentication

Η Firebase μας δίνει διαφορετικούς τρόπους για την εγγραφή χρηστών [29]

- Ηλεκτρονικό ταχυδρομείο και κωδικός πρόσβασης: Οι χρήστες μπορούν να συνδεθούν με το ηλεκτρονικό τους ταχυδρομείο και έναν κωδικό πρόσβασης, η firebase διαχειρίζεται τη διατήρηση του κωδικού πρόσβασης ως μυστικο, μας δίνει επίσης έναν τρόπο επαναφοράς των κωδικών πρόσβασης.
- Εγγραφή μέσω διαφορετικών παροχών (providers): Μπορούμε να χρησιμοποιήσουμε διαφορετικούς παρόχους όπως το Facebook, το Google, το Github και άλλα για να συνδεθούμε στην εφαρμογή.
- Αριθμός τηλεφώνου: Καταχωρεί τους χρήστες με τον αριθμό τηλεφώνου τους, το sms χρησιμοποιείται για την επαλήθευση του αριθμού.

1.9.2 Storage

Το Firebase storage είναι μια υπηρεσία αποθήκευσης για την αποθήκευση διαφόρων τύπων αρχείων. Μπορούμε να ανεβάζουμε και να κατεβάζουμε αρχεία, να τα κάνουμε δημόσια ή ιδιωτικά και να ταξινομούμε τα αρχεία μας σε διαφορετικούς φακέλους.

1.10 Socket.io

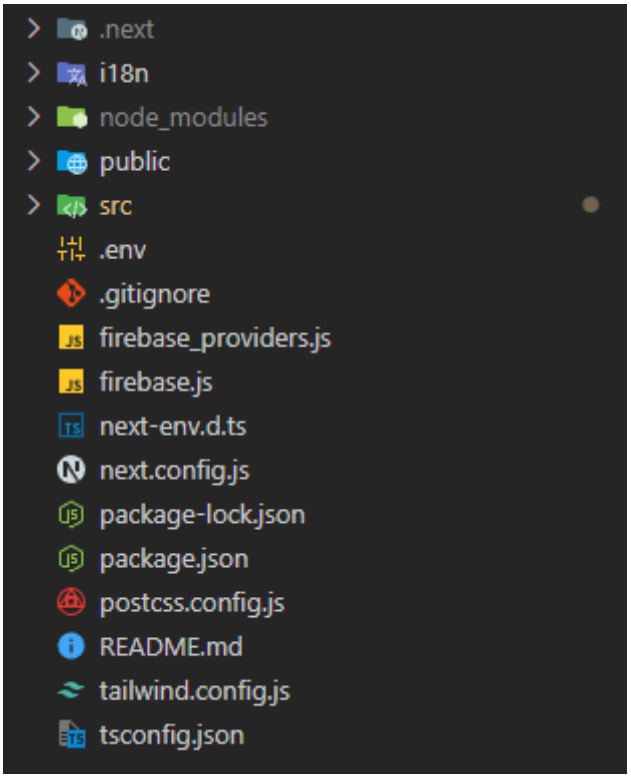
Το WebSocket είναι ένα πρωτόκολλο που επιτρέπει την αμφίδρομη μόνιμη επικοινωνία μεταξύ ενός client και ενός διακομιστή. [30] Χρησιμοποιούμε τη βιβλιοθήκη Socket.io για να υλοποιήσουμε το WebSocket στην εφαρμογή μας για την επικοινωνία μεταξύ των χρηστών. [31]

Κεφάλαιο 2ο: Επισκόπηση κώδικα

Στην παρακάτω ενότητα θα εξηγήσουμε πώς είναι δομημένοι οι φάκελοι του πρότζεκτ, πώς σχεδιάζονται τα διάφορα features και οι λειτουργίες τόσο στο frontend όσο και στο backend του πρότζεκτ.

2.1 Frontend Κώδικας

2.1.1 Δομή φακέλου



Σχήμα 2.1: Δομή φακέλου

Στο πρότζεκτ έχουμε τους ακόλουθους φακέλους:

- **.next**: αυτός περιέχει τα αποτελέσματα του τελικού έργου αφού το χτίσουμε (build).
- **I18n**: περιέχει τις μεταφράσεις για το κείμενο της εφαρμογής στα ελληνικά και στα αγγλικά
- **Node_modules**: περιέχει όλες τις εξαρτήσεις (dependencies) μας
- **Public**: περιέχει όλα τα assets μας όπως εικονίδια και εικόνες
- **Src**: αυτός ο φάκελος περιέχει όλο τον κώδικα της εφαρμογής μας

Εκτός από αυτούς τους φακέλους, έχουμε κάποια αρχεία ρυθμίσεων, μερικά από αυτά δημιουργούνται από το Next.js και δεν χρειάζεται ποτέ να τα τροποποιήσουμε, άλλα όμως χρειάζεται,

για παράδειγμα το αρχείο next.config.js, το οποίο θα εξερευνήσουμε περισσότερο σε επόμενες ενότητες.

Έχουμε επίσης το αρχείο package.json το οποίο είναι κοινό σε όλες τις διαδικτυακές εφαρμογές, αυτά περιέχουν πληροφορίες που σχετίζονται με το πρότζεκτ μας, όπως ποιες εξαρτήσεις χρησιμοποιούμε, διάφορες εντολές για την τοπική εκτέλεση της εφαρμογής, την κατασκευή της εφαρμογής (build) κ.λπ.

```
{
  "name": "kaiwa",
  "version": "0.1.0",
  "private": true,
  "scripts": {
       "dev": "next dev",
       "build": "next build",
       "start": "next start",
       "lint": "next lint"
  },
  "dependencies": {
       "@next-auth/prisma-adapter": "^1.0.5",
       "@prisma/client": "^4.11.0",
       "@types/react": "18.0.28",
       "@types/react-dom": "18.0.11",
       "firebase": "^10.1.0",
       "next": "13.2.3",
       "next-auth": "^4.20.1",
       "react": "18.2.0",
       "react-dom": "18.2.0",
       "react-query": "^3.39.3",
       "socket.io-client": "^4.7.3"
  },
  "devDependencies": {
       "@types/node": "^18.14.6",
       "autoprefixer": "^10.4.13",
       "postcss": "^8.4.21",
       "prisma": "^4.11.0",
       "tailwindcss": "^3.2.7",
       "ts-node": "^10.9.1",
       "typescript": "^4.9.5"
  }
}
```

Το αρχείο “package.json“ ορίζει τα metadata, τις εξαρτήσεις (dependencies) και τα scripts του project. Καθορίζει το όνομα του project, την έκδοση και τις εξαρτήσεις που απαιτούνται τόσο για το production (“dependencies“) όσο και για το development (“devDependencies“). Η ενότητα “scripts“ περιλαμβάνει εντολές για την εκτέλεση της εφαρμογής στο development (“dev“), την κατασκευή της για το production (“build“) και την εκκίνηση του production server (“start“)

Firebase.js: αυτό είναι το αρχείο όπου συνδέουμε την εφαρμογή μας με το πρότζεκτ της firebase που έχουμε δημιουργήσει στον ιστότοπο της firebase.

```
import { initializeApp } from "firebase/app";
import { getAuth } from "firebase/auth";

const firebaseConfig = {
  apiKey: "SECRET",
  authDomain: "SECRET",
  projectId: "SECRET",
  storageBucket: "SECRET",
  messagingSenderId: "SECRET",
  appId: "SECRET",
};

export const app = initializeApp(firebaseConfig);
export const auth = getAuth(app);
```

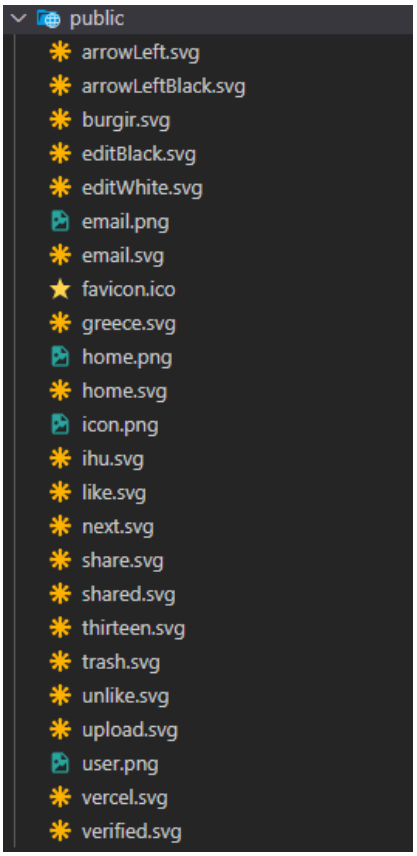
Σε αυτόν τον κώδικα έχουμε ένα αντικείμενο firebaseConfig που μπορούμε να βρούμε στην κονσόλα της firebase. Περνάμε το αντικείμενο ως παράμετρο στην initializeApp για να αρχικοποιήσουμε την firebase στο project μας. Χρησιμοποιούμε το getAuth για να αρχικοποιήσουμε την αυθεντικοποίηση στο project μας.

Firebase_provider.js: εδώ αρχικοποιούμε τον πάροχο που θα χρησιμοποιήσουμε για τον έλεγχο ταυτότητας, στην προκειμένη περίπτωση για την είσοδο στο Google.

```
import { GoogleAuthProvider } from 'firebase/auth';

const provider = new GoogleAuthProvider();

export default provider;
```

2.1.2 Φάκελος Public

Το public αρχείο περιέχει όλα τα στατικά μας στοιχεία, τα εικονίδια svg και κάποιες προεπιλεγμένες εικόνες που θα εμφανίζονται στην εφαρμογή μας.



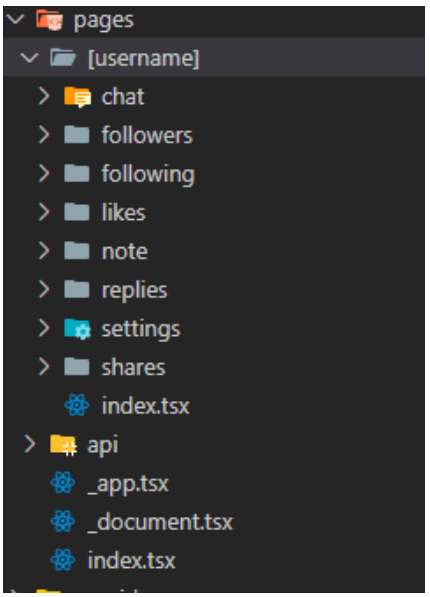
Σχήμα 2.2: Φάκελος Public

2.1.3 Φάκελος Src

Ο φάκελος Src περιέχει πολλούς φακέλους, οι οποίοι οργανώνουν τα components, τους τύπους, τα hooks και άλλα κομμάτια του κώδικα.

2.1.4 Φάκελος Pages

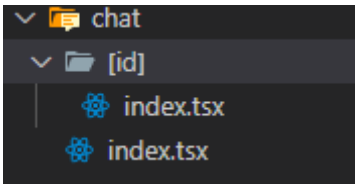
Ο κύριος φάκελος μέσα στο φάκελο src είναι ο φάκελος pages, εδώ έχουμε τις επιμέρους σελίδες των εφαρμογών και τα προεπιλεγμένα αρχεία Next.js: index.tsx, _app.tsx, _document.tsx που καθορίζουν πώς θα είναι δομημένη η εφαρμογή μας.



Σχήμα 2.3: Φάκελος Pages

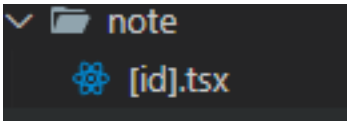
Μέσα στο φάκελο pages έχουμε το φάκελο [username], ο οποίος εμφανίζει δεδομένα δυναμικά ανάλογα με το προφίλ του χρήστη στο οποίο βρισκόμαστε χρησιμοποιώντας το αρχείο index.tsx μέσα στο φάκελο [username]. Οι άλλοι φάκελοι μας πλοηγούν σε διαφορετικές σελίδες του χρήστη που επισκεπτόμαστε, για παράδειγμα ο φάκελος chat θα μας εμφανίσει το UI για τη διαδρομή «https://www.examaple.com/[username]/chat».

Κάθε ένας από τους φακέλους έχει ένα αρχείο index.tsx μέσα στο οποίο είναι υπεύθυνο για την εμφάνιση του UI, εκτός από τους φακέλους chat και note.



Σχήμα 2.4: Φάκελος Chat

Στο φάκελο συνομιλίας έχουμε εμφωλευμένη δρομολόγηση για διαφορετικές συνομιλίες. Το index.tsx του φακέλου chat θα εμφανίζει το γενικό UI της σελίδας συνομιλιών, ενώ το index.tsx μέσα στο φάκελο [id] θα εμφανίζει τη συνομιλία που έχει επιλέξει ο χρήστης.

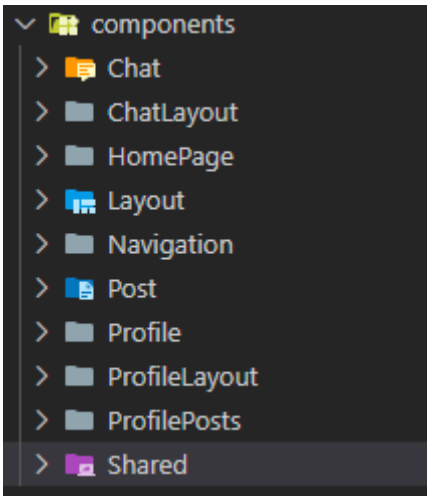


Σχήμα 2.5: Φάκελος Note

Ομοίως, στο φάκελο note, το [id].tsx θα εμφανίσει το συγκεκριμένο note που έχουμε ανοίξει. Εναλλακτικά, θα μπορούσαμε να σχεδιάσουμε αυτόν τον φάκελο ως [id] και να έχουμε ένα αρχείο index.tsx μέσα σε αυτόν.

2.1.5 Φάκελος Components

Εδώ κρατάμε όλα τα components που χρησιμοποιούμε μέσα στις σελίδες μας και μέσα σε άλλα components.



Σχήμα 2.6: Φάκελος Components

- Chat: περιέχει το component για την προβολή μηνυμάτων συνομιλίας
- ChatLayout: περιέχει το component για τη διάταξη ανά σελίδα της σελίδας συνομιλίας
- HomePage: περιέχει components που χρησιμοποιούνται στην αρχική σελίδα
- Layout: περιέχει το component για τη διάταξη ολόκληρης της εφαρμογής
- Navigation: περιέχει components για την side navigation bar
- Post: περιέχει components για την εμφάνιση των post και των post reply στη σελίδα χρηστών, στην αρχική σελίδα και σε μεμονωμένα post
- Profile: περιέχει components για την εμφάνιση των πληροφοριών των χρηστών στο προφίλ τους
- ProfileLayout: περιέχει τη διάταξη ανά σελίδα για τις διαδρομές υπό το χρήστη
- ProfilePosts: περιέχει components για την εμφάνιση των αναρτήσεων που έχει δημιουργήσει ένας χρήστης, τις οποίες έχει κάνει like, τις οποίες έχει κοινοποιήσει, στις οποίες έχει

απαντήσει. Αυτά τα components χρησιμοποιούνται μέσα στα components του φακέλου Post και εμφανίζονται στη σελίδα του χρήστη.

- Shared: περιέχει components που χρησιμοποιούνται σε πολλαπλές σελίδες και components

2.1.6 Layouts

Component διάταξης: Περιέχει το section<main> για την εμφάνιση της διάταξης σε μια μεγάλη οθόνη.

```
<main className="md:flex flex-row py-8 px-4">
  <section className="hidden sm:basis-4/6  md:basis-96 md:block">
       <Nav></Nav>
  </section>
  <section className="block md:hidden mb-4">
       <div className="w-4/5 m-auto flex justify-between">
       <Image src={"/ihu.svg"} alt={"ihu"} width={40} height={40}></Image>
       <Image
       src={"/burgir.svg"}
       alt={"burgir"}
       width={40}
       height={40}
       className=" cursor-pointer"
       onClick={toggleMenu}
       ></Image>
       </div>
  </section>
  <section className="basis-full">
       {user && (user?.hasRegistered ? <>{children}</> : <ProfileDetails />)}
  </section>
</main>;
```

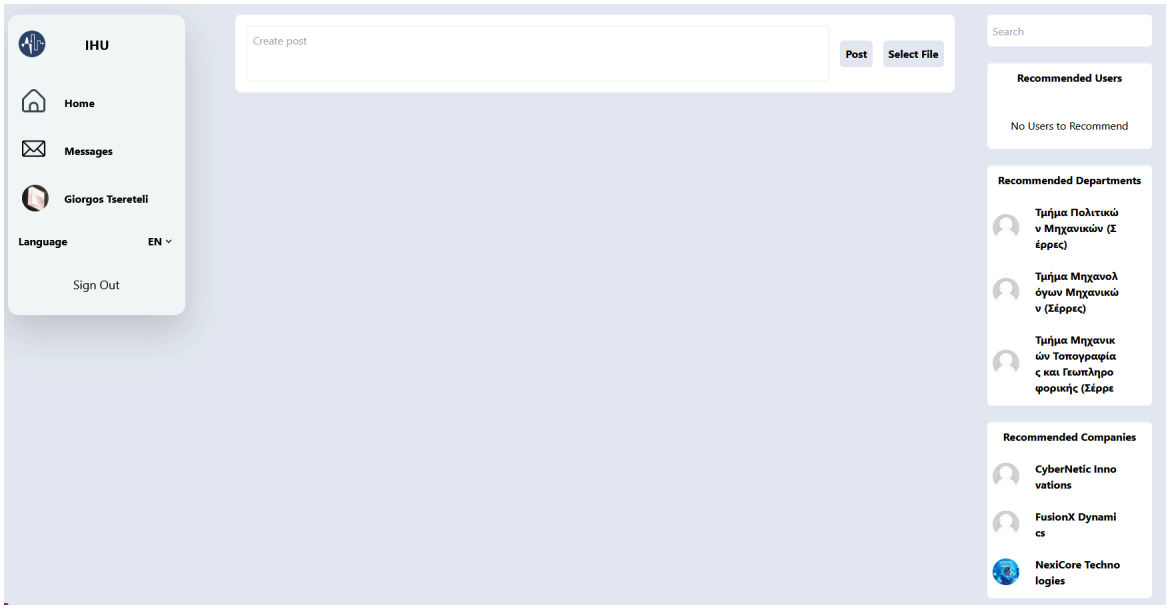
Εάν η οθόνη ξεπεράσει ένα συγκεκριμένο πλάτος, το component <Nav> εμφανίζεται στην αριστερή πλευρά, στη δεξιά πλευρά είτε τα children props που του περνάμε εάν ο χρήστης έχει ολοκληρώσει την εγγραφή του ή το component για να συμπληρώσει τα στοιχεία του προφίλ του.

Πάνω από αυτό έχουμε ένα εναλλακτικό div για την εμφάνιση του side nav με διαφορετικό τρόπο αν χρησιμοποιούμε tablet ή κινητό.

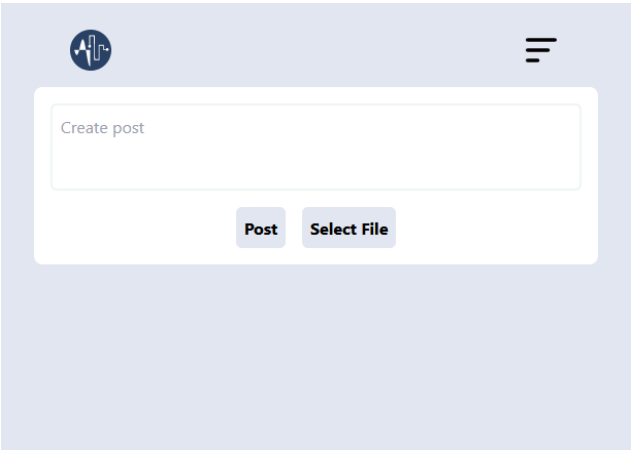
```
<div
  style={{
        position: "absolute",
        top: 0,
        left: 0,
        zIndex: 5,
        backgroundColor: "white",
        width: "100%",
        height: "100%",
        display: menuOpen ? "flex" : "none",
        flexDirection: "column",
        justifyContent: "center",
        alignItems: "center",
  }}
>
  <div>
        {user ? (
        <>
        {" "}
        <Link href="/" onClick={toggleMenu}>
        {" "}
        <NavbarButtons
               icon={"/home.png"}
               text={"Home"}
               alt={"Home Icon"}
        ></NavbarButtons>
        </Link>
        <Link href={`/${user.id}/chat`} onClick={toggleMenu}>
        <NavbarButtons
               icon={"/../public/email.png"}
               text={"Messages"}
               alt={"Message Icon"}
        ></NavbarButtons>
        </Link>
        <Link href={`/${user.id}`} onClick={toggleMenu}>
        <NavbarButtons
               icon={user.profileImage ?? ""}
               text={`${user.displayName}`}
               alt={"Profile Icon"}
        ></NavbarButtons>
        </Link>
        <Logout />
        <div className="flex justify-center font-bold cursor-pointer mt-4">
```

```
            <p onClick={toggleMenu}>X</p>
            </div>
            </>
            ) : (
            <Login />
            )}
        </div>
    </div>;
```

Div για την εμφάνιση της πλοήγησης σε μικρή οθόνη, μέσα στο div εισάγουμε το component NavbarButtons το οποίο εμφανίζει τα διάφορα κουμπιά για την πλοήγηση στην αρχική σελίδα ή στο προφίλ του χρήστη κ.λπ., καθώς και τα components Logout και Login ώστε ο χρήστης να μπορεί είτε να συνδεθεί στην εφαρμογή είτε να αποσυνδεθεί.



Σχήμα 2.7: Παράδειγμα διάταξης σε μεγάλη οθόνη



Σχήμα 2.8: Παράδειγμα διάταξης σε μικρή οθόνη

Σχήμα 2.8: Navbar σε μικρή οθόνη όταν εμφανίζεται όταν πατάτε το hamburger menu

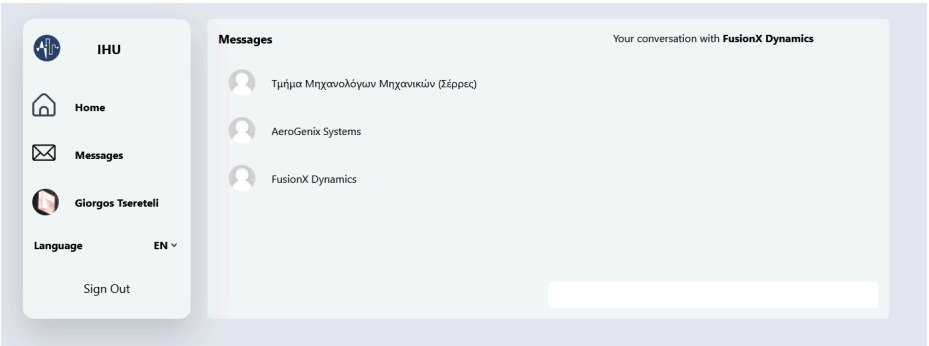
ChatLayout: Το component ChatLayout.tsx μέσα στο φάκελο ChatLayout εμφανίζει τη διάταξη για τη σελίδα των συνομιλιών μας. Είναι υπεύθυνο για:

- Λήψη και δημιουργία των συνομιλιών μεταξύ διαφορετικών χρηστών
- Σύνδεση στο socket με τα σωστά διαπιστευτήρια για να είναι σε θέση να ταυτοποιηθούν

παράδειγμα κώδικα: σύνδεση με το socket αν υπάρχουν συνομιλίες.

```
if (data && data.data) {
  socket.auth = { userId: user.id };
  socket.connect();
}
```

- Εμφάνιση του component Chat



Σχήμα 2.9: Παράδειγμα διάταξης σελίδας συνομιλίας

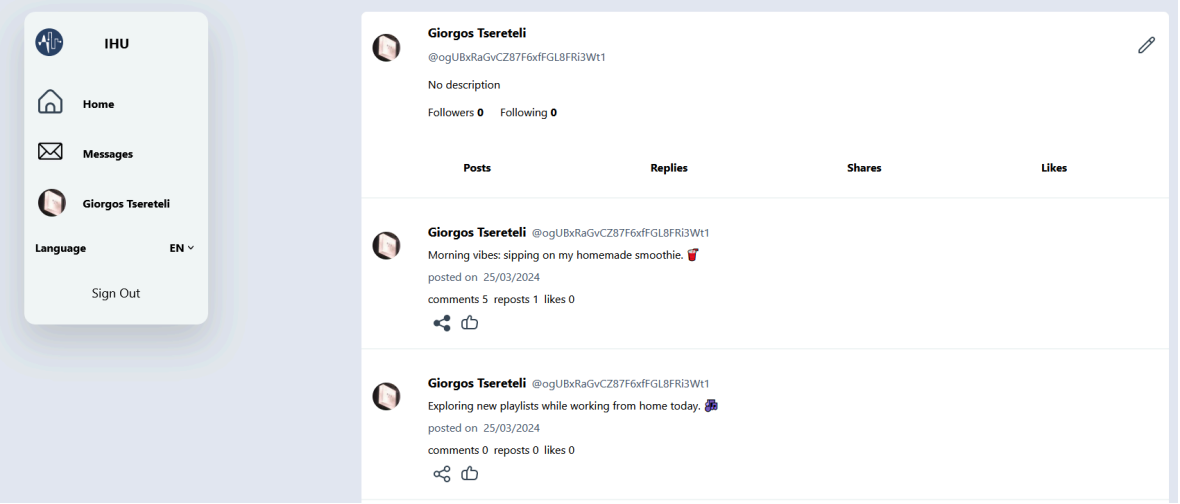
ProfileLayout: ProfileLayout.tsx μέσα στο φάκελο ProfileLayout εμφανίζει τη διάταξη στη σελίδα των χρηστών. Είναι υπεύθυνο για:

- την λήψη των δεδομένων του χρήστη
- Εμφάνιση των πληροφοριών των χρηστών
- την εμφάνιση των components που περνάμε σε αυτό

```
export default function ProfileLayout({ children }: any) {
  const router = useRouter();
  const { locale } = router;
  const t = locale === "en" ? en : gr;
  const { username } = router.query; //from pages [username]
  const { firebaseUser } = useAuthContext();

  const { data, isLoading, isError } = useQuery({
       queryKey: ["userProf", username],
       queryFn: async () => fetchUser(firebaseUser, username as string),
  });

  if (isLoading) {
       return <>{t.loading}</>;
  }

  if (isError) {
       return <>{t.somethingWentWrong}</>;
  }

  if (data) {
       return (
       <div className="sm:w-11/12 md:w-3/4 m-auto bg-white h-full rounded-lg">
       <ProfileCard data={data as UserPrisma} />
       <div className="flex justify-between md:justify-around p-8">
       <Link className="font-bold" href={`/${username}`}>
              {t.userPosts}
       </Link>
       <Link className="font-bold" href={`/${username}/replies`}>
              {t.userReplies}
       </Link>
       <Link className="font-bold" href={`/${username}/shares`}>
              {t.userShares}
       </Link>
       <Link className="font-bold" href={`/${username}/likes`}>
              {t.userLikes}
```

```
            </Link>
            </div>
            {children}
            </div>
            );
        } else {
            return <p>{t.userNotFound}</p>;
        }
    }
```

Αυτό το component παίρνει τα δεδομένα των χρηστών με τη συνάρτηση fetchUser, η μεταβλητή firebaseUser χρησιμοποιείται για να πάρει το token firebase του τρέχοντος χρήστη, το username είναι το προφίλ που επισκεπτόμαστε, δείχνουμε διαφορετικό ui ενώ περιμένουμε τα δεδομένα και αν προκύψει κάποιο σφάλμα, όταν έχουμε τα δεδομένα των χρηστών τα εμφανίζουμε χρησιμοποιώντας το component ProfileCard, και κάτω από αυτό εμφανίζουμε τα children props που περνάνε σε αυτό το component Π.χ. το component για τις δημοσιεύσεις των χρηστών, τις απαντήσεις, το share, τα likes



Σχήμα 2.10: Παράδειγμα διάταξης προφίλ

Για να χρησιμοποιήσουμε το γενικό component Layout, απλά το εισάγουμε στο component _app.tsx, για τα components ChatLayout και ProfileLayout, τα εισάγουμε στα components που σχετίζονται με τα Layouts. Εισάγουμε το component ProfileLayout στο φάκελο pages μέσα στους φακέλους Followers, Following, Likes, Replies, Shares και το index.tsx

Εχουμε το component Profile το οποίο εμφανίζει τις δημοσιεύσεις που έχει κάνει ο χρήστης χρησιμοποιώντας το component <Posts>, αυτό το component περνάει ως παιδί στο ProfileLayout

```
const Profile: NextPageWithLayout = () => {
  const router = useRouter();
  const { username } = router.query;
  const { firebaseUser } = useAuthContext();

  return (
        <div id="infinite">
        <Posts
        queryKey={"post"}
        queryVal={username as string}
        queryFunction={fetchPosts}
        queryFunctionProps={[username, firebaseUser]}
        ></Posts>
        </div>
  );
};

Profile.getLayout = function getLayout(page: ReactElement) {
  return <ProfileLayout>{page}</ProfileLayout>;
};
```

Μέσα στο _app.tsx απλά εισάγουμε το component γενικής διάταξης και το Next.js αναλαμβάνει την εμφάνιση της ανεξάρτητης διάταξης κάθε σελίδας

```
export type NextPageWithLayout<P = {}, IP = P> = NextPage<P, IP> & {
  getLayout?: (page: ReactElement) => ReactNode;
};

type AppPropsWithLayout = AppProps & {
  Component: NextPageWithLayout;
};

export default function App({
  Component,
  pageProps: { session, ...pageProps },
  ...appProps
}: AppPropsWithLayout) {
  const [queryClient] = useState(() => new QueryClient());

  const getLayout = Component.getLayout ?? ((page) => page);

  return (
```

```
            <AuthContextProvider>
            <QueryClientProvider client={queryClient}>
            <Hydrate state={pageProps.dehydratedState}>
            <Layout>
                    <ReactQueryDevtools initialIsOpen={false} />
                    {getLayout(<Component {...pageProps} />)}
            </Layout>
            </Hydrate>
            </QueryClientProvider>
            </AuthContextProvider>
      );
}
```

Εδώ έχουμε το component App όπου έχουμε εισάγει το γενικό layout component μας, έχουμε επίσης εισάγει το AuthContextProvider το οποίο εξηγούμε στην ενότητα 2.1.12 και το QueryClientProvider το οποίο μας επιτρέπει να χρησιμοποιήσουμε τη βιβλιοθήκη react query. Τυλίγουμε (wrap) όλες αυτές τις μεθόδους γύρω από την εφαρμογή μας έτσι ώστε αυτές οι βιβλιοθήκες και τα συστατικά να μπορούν να χρησιμοποιηθούν σε όλη την εφαρμογή μας.

2.1.7 Posts Component

Το component Posts είναι το κύριο component που είναι υπεύθυνο για την εμφάνιση μιας λίστας αναρτήσεων, χρησιμοποιούμε αυτό το component στην αρχική σελίδα για να εμφανίσουμε τις αναρτήσεις των χρηστών που ακολουθείτε, στο προφίλ των χρηστών το χρησιμοποιούμε για να εμφανίσουμε τις αναρτήσεις που έχει κάνει ένας χρήστης, έχει μοιραστεί , έχει κάνει like και έχει σχολιάσει.

Το component Posts δέχεται μια λίστα παραμέτρων, που ορίζονται σε αυτή τη διεπαφή typescript.

```
export interface InfinityQuery {
        queryKey: string;
        queryVal: string | string[];
        queryFunction: Function;
        queryFunctionProps: any[];
}
```

Τα props που παίρνει χρησιμοποιούνται για τη δυναμική κλήση διαφορετικών συναρτήσεων λήψης δεδομένων με το React Query, μια βιβλιοθήκη λήψης δεδομένων όπως αναφέρθηκε στην ενότητα Τεχνολογίες. Δεδομένου ότι οι αναρτήσεις έχουν την ίδια μορφή στις σελίδες όπου χρησιμοποιούμε

το component, πρέπει απλώς να αλλάξουμε τη συνάρτηση για να φέρνουμε δεδομένα από διαφορετικά endpoints.

- QueryKey: επιτρέπει στο React query να αναγνωρίζει διαφορετικά queries
- QueryVal: περνάμε ένα QueryVal μαζί με το QueryKey για να κάνουμε την ταυτοποίηση των δεδομένων πιο μοναδική.
- QueryFunction: η συνάρτηση λήψης δεδομένων που περνάμε στο React Query
- QueryFunctionProps: τα props που λαμβάνει η συνάρτηση του query.

```
const Profile: NextPageWithLayout = () => {
  const router = useRouter();
  const { username } = router.query;
  const { firebaseUser } = useAuthContext();

  return (
       <div id="infinite">
       <Posts
       queryKey={"post"}
       queryVal={username as string}
       queryFunction={fetchPosts}
       queryFunctionProps={[username, firebaseUser]}
       ></Posts>
       </div>
  );
};
```

Στον κώδικα του παραδείγματος, διαβάζουμε τις δημοσιεύσεις των χρηστών στο προφίλ τους. Περνάμε το κλειδί δημοσίευσης και το όνομα χρήστη επειδή θέλουμε τις δημοσιεύσεις του τρέχοντος χρήστη, περνάμε τη συνάρτηση fetchPosts που δημιουργήσαμε για να καλέσουμε το endpoint των δημοσιεύσεων του χρήστη και περνάμε τις παραμέτρους για να λάβουμε τα δεδομένα του σωστού χρήστη και να επαληθεύσουμε τον χρήστη με το firebase token του.

```
const {
  data,
  isError,
  isLoading,
  fetchNextPage,
  hasNextPage,
```

```
  isFetchingNextPage,
}: UseInfiniteQueryResult<Page> = useInfiniteQuery(
  [queryKey, queryVal],
  ({ pageParam = 0 }) => queryFunction(...queryFunctionProps, pageParam),
  {
       getNextPageParam: (data: { obj: any; nextPage: number | null }) => {
       return data.nextPage;
       },
  }
);
```

Χρησιμοποιούμε τη συνάρτηση useInfiniteQuery για να κάνουμε σελιδοποίηση των αποτελεσμάτων. Το getNextPageParam ελέγχει αν το πεδίο data.nextPage έχει τιμή. Εάν υπάρχει άλλη σελίδα μπορούμε να καλέσουμε τη συνάρτηση fetchNextPage() για να λάβουμε περισσότερα δεδομένα.

Παραδειγμα: loadmore συνάρτηση που καλεί την fetchNextPage() αν υπάρχει επόμενη σελίδα και δεν λαμβάνουμε δεδομένα αυτή τη στιγμή

```
const loadMore = async () => {
  if (!isFetchingPreviousPage && hasPreviousPage) await fetchPreviousPage();
};
```

Παραδειγμα: Εμφανίζει το element load more και το κάνει να μπορεί να γίνει κλικ αν υπάρχουν περισσότερες σελίδες και δεν λαμβάνουμε δεδομένα αυτή τη στιγμή.

```
{
  !isFetchingPreviousPage && hasPreviousPage && (
       <div className="pb-2">
       {" "}
       <p
       onClick={loadMore}
       className="flex justify-center m-4 hover:cursor-pointer"
       >
       {t.loadMore}
       </p>
       </div>
  );
}
```

Χρησιμοποιούμε το ίδιο μοτίβο και στις άλλες σελίδες που αναφέραμε, ενώ περνάμε τις κατάλληλες παραμέτρους.

Μέσα στο component Posts απεικονίζουμε διαφορετικά components ανάλογα με τη διαδρομή και τα δεδομένα που έχουμε λάβει.

- Στη σελίδα των απαντήσεων, εμφανίζουμε την απάντηση του χρήστη μαζί με την ανάρτηση ή το σχόλιο στο οποίο απάντησε ο χρήστης (<PostPost> και <ReplyPost> components αντίστοιχα μαζί με προσαρμοσμένη html πάνω από αυτά για την εμφάνιση των πληροφοριών του χρήστη)
- Στη σελίδα αναρτήσεων εμφανίζουμε απλώς τις αναρτήσεις των χρηστών (component <ContentPost>).
- Στις σελίδες κοινοποίησης και like δείχνουμε τις δημοσιεύσεις ή τις απαντήσεις που κοινοποίησε ή έκανε like ο χρήστης (components <PostPost> και <ReplyPost>)

2.1.8 ListOfUsers component

Στο προφίλ των χρηστών μπορούμε να δούμε ποιον ακολουθεί ο χρήστης και ποιος τον ακολουθεί, γι' αυτό χρησιμοποιούμε το κοινόχρηστο component <ListOfUsers> το οποίο όπως και το component <Posts> δέχεται παραμέτρους για τα δεδομένα που θα πάρει και θα εμφανίσει. Καλούμε αυτό το component μέσα στους φακέλους pages/following και pages/followers

παράδειγμα λήψης ακολούθων χρηστών χρησιμοποιώντας το component ListOfUsers

```
{
  !isFetchingPreviousPage && hasPreviousPage && (
       <div className="pb-2">
       {" "}
       <p
       onClick={loadMore}
       className="flex justify-center m-4 hover:cursor-pointer"
       >
       {t.loadMore}
       </p>
       </div>
  );
}
```

2.1.9 Home Component

Το component home είναι το αρχείο index.tsx κάτω από το φάκελο pages, αυτό εμφανίζει την αρχική σελίδα και περιλαμβάνει τα εξής:

- <CreatePost> που μας επιτρέπει να δημιουργούμε αναρτήσεις
- Το component<Posts> που αναφέραμε πιο πάνω
- <Search> component που μας επιτρέπει την αναζήτηση χρηστών
- <Recommended> component το οποίο ανάλογα με τις παραμέτρους εμφανίζει προτεινόμενους χρήστες, προτεινόμενα πανεπιστημιακά τμήματα και προτεινόμενες εταιρείες

```
export default function Home() {
  const { firebaseUser } = useAuthContext();
  const router = useRouter();
  const { username } = router.query;

  const fetchPosts = async (firebaseUser: User, page: number) => {
       if (!firebaseUser) return null;

       const token = await firebaseUser.getIdToken();
       const res = await fetch(
       `${process.env.NEXT_PUBLIC_API_KEY}/user/feed?page=${page}`,
       {
       headers: {
       Authorization: `Bearer ${token}`,
       },
       }
       );
       const response = await res.json();
       let data = response.data;
       const nextPage = response.nextPage;

       return { data, nextPage };
  };

  return (
       <>
       <div className="flex">
       <main className="flex flex-col flex-1">
       <CreatePost />

       <div className="m-auto mt-6 rounded-lg w-11/12">
              <Posts
              queryKey={"feed"}
              queryVal={username as string}
              queryFunction={fetchPosts}
              queryFunctionProps={[firebaseUser]}
```

```
                />
            </div>
            </main>
            <div className="hidden lg:flex w-3/12">
            <div>
                {" "}
                <Search />
                <Recommended
                recommendationQueryKey="recommendedUsers"
                recommendationType="users"
                type="Users"
                />
                <Recommended
                recommendationQueryKey="recommendedDepartments"
                recommendationType="departments"
                type="Departments"
                />
                <Recommended
                recommendationQueryKey="recommendedCompanies"
                recommendationType="companies"
                type="Companies"
                />
            </div>
            </div>
            </div>
            </>
        );
    }
```

Μέσα στο component Posts περνάμε τη συνάρτηση fetchPosts η οποία επιστρέφει αναρτήσεις από τους χρήστες που ακολουθούμε.

2.1.10 Recommended Component

Χρησιμοποιούμε αυτό το component για να εμφανίζουμε τους προτεινόμενους χρήστες, τις εταιρείες και τα πανεπιστημιακά τμήματα. Παίρνει ως props τον τύπο της σύστασης που θέλουμε να λάβουμε και καλεί το σχετικό endpoint και αλλάζει το κείμενο του UI δυναμικά με βάση τα props.

2.1.11 Search Component

Χρησιμοποιούμε αυτό το component για την αναζήτηση χρηστών, καλούμε το endpoint αναζήτησης κάθε φορά που ένας χρήστης πληκτρολογεί στο πεδίο εισαγωγής.

2.1.12 Context

Έχουμε ένα AuthContext για να κρατάμε τον χρήστη μας συνδεδεμένο και να έχουμε τα στοιχεία του αποθηκευμένα σε ένα γενικό μέρος για να μπορούμε να έχουμε εύκολη πρόσβαση από άλλα components. Χρησιμοποιούμε τη μέθοδο onAuthStateChanged του firebase authentication που παρακολουθεί την κατάσταση των χρηστών, όταν η κατάσταση αλλάζει σε logged in, παίρνουμε τον συνδεδεμένο χρήστη και αποστέλλουμε ένα action χρησιμοποιώντας το useReducer hook για να αποθηκεύσουμε τον χρήστη στην κατάσταση του AuthContext μας, όταν αποσυνδεθεί καλούμε το dispatch action για την αποσύνδεση και τον αφαιρούμε από την κατάσταση.

2.1.13 Hooks

Έχουμε διαφορετικά hooks για διαφορετικές κλήσεις του endpoint

- useDeletePost: διαγράφει μια δημοσίευση/απάντηση
- useFollowUser: ακολουθεί ή διαγράφει την ακολούθηση ενός άλλου χρήστη
- useLikePost: κάνει like ή unlike μια δημοσίευση
- useLikeReply: κάνει like ή unlike μια απάντηση
- useSharePost: κοινοποιεί ή καταργεί την κοινοποίηση της δημοσίευσης
- useShareReply: κοινοποιεί ή καταργεί την κοινοποίηση μιας απάντησης

2.1.14 Login and Logout Components

Το login και το loging out γίνεται στο frontend με τη χρήση της μεθόδου firebase signInWithPopup για το login και της μεθόδου signOut για το logging out.

Όταν κάνουμε κλικ στο κουμπί sign in, εμφανίζεται ένα popup του Google auth provider, το οποίο μας ζητά να συνδεθούμε με έναν λογαριασμό google. Όταν συνδεόμαστε, η firebase δημιουργεί έναν χρήστη στο Authentication της, μετά από αυτό στέλνουμε το GoogleId του χρήστη και άλλα διαπιστευτήρια στο backend για να δημιουργήσουμε τον χρήστη αν δεν υπάρχει , ή απλά να συνδεθούμε αν υπάρχει, στη συνέχεια έχουμε τα στοιχεία του χρήστη στο AuthContext μας για να χρησιμοποιήσουμε τα στοιχεία και το token του στην υπόλοιπη εφαρμογή και να τον κρατήσουμε συνδεδεμένο στην εφαρμογή μας.

Όταν κάνουμε κλικ στο sign out η μέθοδος firebase μας αποσυνδέει και αφαιρούμε τα στοιχεία του χρήστη από το AuthContext μας.

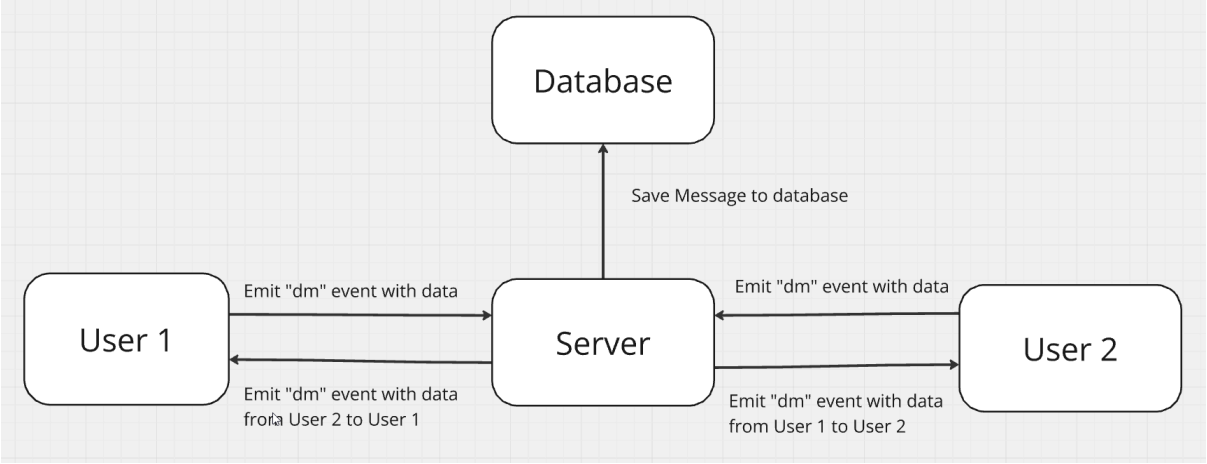
2.1.15 Socket

Αναφέραμε παραπάνω ότι στο component Chat συνδεόμαστε με το socket, πριν το κάνουμε αυτό αρχικοποιούμε το socket στο φάκελο providers, το αρχικοποιούμε περνώντας τη διεύθυνση URL του backend μας στη μέθοδο **io** της client βιβλιοθήκης socket.io, θέτουμε επίσης την επιλογή **autoConnect** σε **false** έτσι ώστε η εφαρμογή μας να μην συνδέεται στο socket από την αρχή, αλλά μόνο όταν το θέλουμε.

Εισάγουμε τον socket client στο Chat component όπου τον έχουμε να εκπέμπει και να ακούει συγκεκριμένα socket events χρησιμοποιώντας τη μέθοδο **on** για την ακρόαση events και τη μέθοδο **emit** για την εκπομπή events.

- 'Dm' event, εκπέμπουμε ένα **dm event** όταν στέλνουμε ένα μήνυμα, μαζί με πρόσθετες πληροφορίες σχετικά με το ποιος στέλνει το μήνυμα σε ποιον και το **id** της συνομιλίας στην οποία βρισκόμαστε, όταν λαμβάνουμε ένα **dm event** ενημερώνουμε το state messages που έχουμε στο component για να δείξουμε τα νέα μηνύματα.

Τα αρχικά μηνύματα λαμβάνονται σε ένα useInfiniteQuery όταν φορτώνουμε για πρώτη φορά το component.



Σχημα 2.11: Διαγραμμα ροης Chat

Σε αυτό το σχήμα έχουμε ένα διάγραμμα ροής για το πώς λειτουργεί το chat:
Ο χρήστης 1 εκπέμπει ένα event dm με τα απαραίτητα δεδομένα, ο διακομιστής επικυρώνει τα δεδομένα και εκπέμπει ένα event dm στον χρήστη 2, αποθηκεύοντάς το επίσης στη βάση δεδομένων. Το ίδιο συμβαίνει και όταν ο χρήστης 2 στέλνει ένα μήνυμα.

2.1.16 Internationalisation

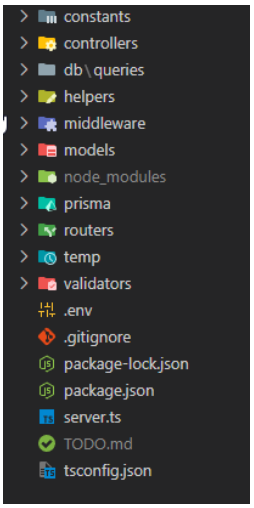
Έχουμε διεθνοποίηση για τα ελληνικά και τα αγγλικά. Για να το πετύχουμε αυτό πρέπει να ρυθμίσουμε το αρχείο next.config.js με τις γλώσσες που θέλουμε να είναι διαθέσιμες στην εφαρμογή μας. Χρησιμοποιούμε το sub-path routing στην εφαρμογή μας, αυτό θα έχει την αγγλική γλώσσα στην

προεπιλογή και στη διαδρομή και για την Ελλάδα η ιστοσελίδα θα ανακατευθύνει στο www.example.com/gr/examplepath κλπ για κάθε σελίδα.

Έχουμε έναν selector γλώσσας στη navbar, όταν ο χρήστης επιλέγει μια γλώσσα, η ιδιότητα locale του αντικειμένου router αλλάζει είτε σε «en» για τα αγγλικά είτε σε «gr» για τα ελληνικά. Έχουμε επίσης 2 αρχεία json με properties που έχουν κείμενο σε διαφορετικές γλώσσες, ένα στα αγγλικά και ένα στα ελληνικά. Χρησιμοποιούμε το router.locale για να επιλέξουμε αν θα χρησιμοποιήσουμε το αγγλικό json για μετάφραση ή το ελληνικό, στη συνέχεια χρησιμοποιούμε μια μεταβλητή μετάφρασης «t» και την αρχικοποιούμε με το αντίστοιχο json. Στη συνέχεια, όπου έχουμε κείμενο καλούμε το property του json που χρειαζόμαστε στο συγκεκριμένο σημείο του κώδικα html. για παράδειγμα <h1>{{t.title}}.</h1>

2.2 Backend

2.2.1 Δομή Φακέλου Backend



Σχήμα 2.12: Δομή φακέλου Backend

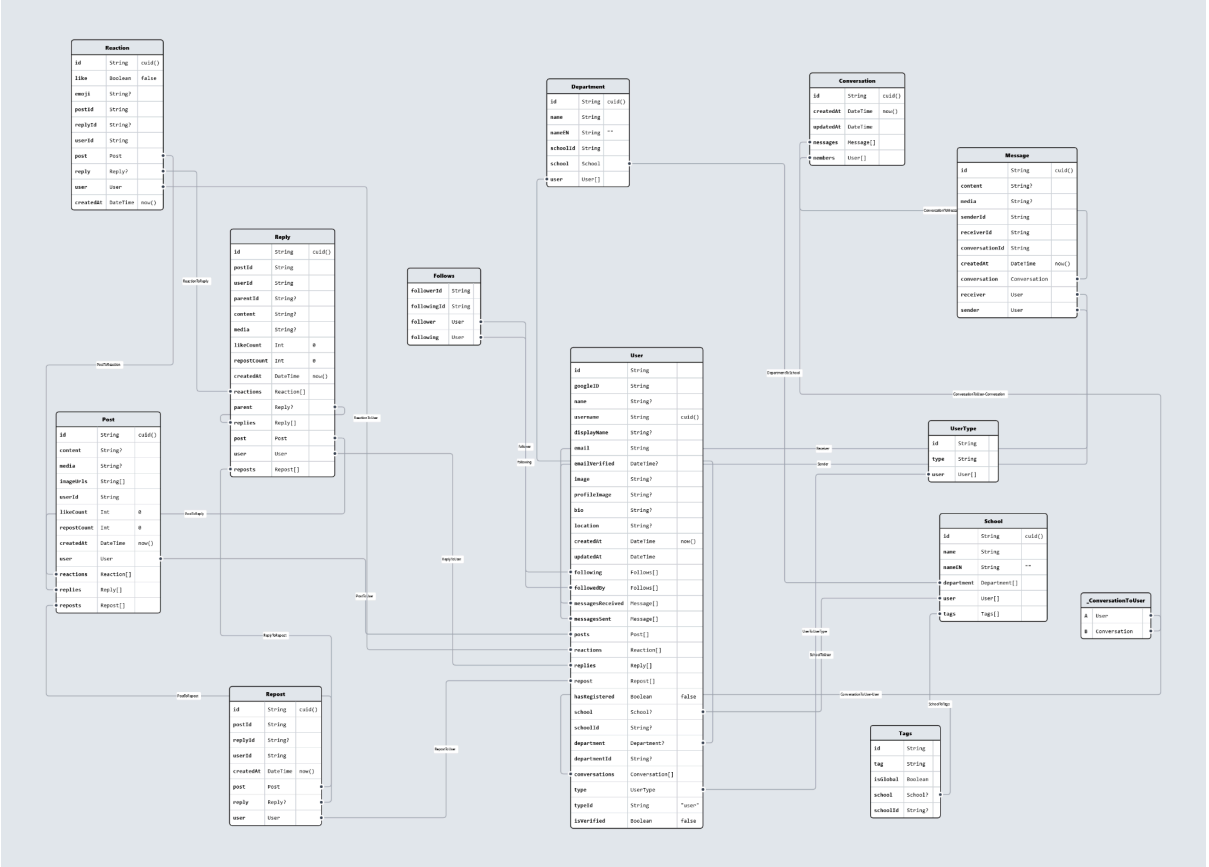
Στο backend έχουμε τους φακέλους:

- constants: περιέχει τα αρχεία ρυθμίσεων της firebase
- Controllers: περιέχει τους controllers για όλα τα endpoints
- Db: περιέχει τα περισσότερα από τα ερωτήματα της βάσης δεδομένων
- Helpers: περιέχει βοηθητικές συναρτήσεις για το ανέβασμα εικόνων και άλλα.
- Middleware: περιέχει middleware για την αυθεντικοποίηση του χρήστη
- Μοντέλα: περιέχει τύπους typescript
- Prisma: περιέχει το σχήμα της βάσης δεδομένων prisma και τα αρχεία migration
- Routers: περιέχει τις διαδρομές για τα endpoints
- Validators: περιέχει validators για ορισμένα από τα post requests, που επιβεβαιώνουν τα δεδομένα που στέλνει ο χρήστης στο post request
- Temp: οι εικόνες που μεταφορτώνονται αποθηκεύονται εδώ προσωρινά όσο μεταφορτώνονται στο firebase storage και στη συνέχεια διαγράφονται μόλις ολοκληρωθεί η μεταφόρτωση.

2.2.2 Server Initialisation

Αρχικοποιούμε τον διακομιστή μέσα στο αρχείο server.ts. Αυτό είναι το κύριο αρχείο το οποίο έχει τα εξής: αρχικοποίηση διακομιστή, κεντρικό middleware χειρισμού σφαλμάτων, δρομολόγηση στον κεντρικό δρομολογητή, server side web socket client για τη συνομιλία

2.2.3 Prisma Schema



Σχήμα 2.13: Prisma Schema

Το σχήμα μας περιέχει τους ακόλουθους πίνακες:

- User: πίνακας που σχετίζεται με τον χρήστη, περιέχει πολλαπλά πεδία όπως id, googleID, name, profileImage κ.λπ., σχετίζεται με πολλούς πίνακες ως ένας προς πολλούς και πολλοί προς πολλούς.
- UserType: καθορίζει τον τύπο ενός χρήστη, μπορεί να είναι είτε «χρήστης» είτε «εταιρεία» είτε «πανεπιστήμιο».
- Tags: ετικέτες μιας ανάρτησης π.χ. μαθηματικά, προγραμματισμός κ.λπ. Προς το παρόν δεν χρησιμοποιείται, αλλά θα μπορούσε να προστεθεί ως πρόσθετη λειτουργία.
- Follows: αυτός ο πίνακας περιέχει τις ακόλουθες σχέσεις μεταξύ των χρηστών, υποδεικνύοντας ποιος ακολουθεί ποιον
- Post: περιέχει τις πληροφορίες που σχετίζονται με τις δημοσιεύσεις. Σχετίζεται με πολλαπλούς πίνακες, ώστε να παρακολουθεί τις απαντήσεις, τα likes, τις αναδημοσιεύσεις και ποιος έκανε την ανάρτηση

- Repost: περιέχει λεπτομέρειες που σχετίζονται με την αναδημοσίευση μιας δημοσίευσης ή μιας απάντησης
- Reply: περιέχει λεπτομέρειες που σχετίζονται με μια απάντηση σε ανάρτηση ή άλλη απάντηση
- Reaction: περιέχει λεπτομέρειες σχετικά με το ποιος έκανε like σε ποια ανάρτηση
- Conversation: περιέχει λεπτομέρειες σχετικά με τις συνομιλίες
- Message: περιέχει τα μηνύματα που σχετίζονται με συνομιλίες
- School: περιέχει τις διάφορες σχολές του πανεπιστημίου
- Department: περιέχει τα διάφορα τμήματα των σχολών

2.2.4 Middleware

Έχουμε ένα middleware που ελέγχει αν ο χρήστης είναι εξουσιοδοτημένος να αλληλεπιδράσει με το API. Σε κάθε αίτηση ο client στέλνει ένα JSON Web Token (JWT) στην επικεφαλίδα της αίτησης χρησιμοποιώντας τη μέθοδο του firebase client για να πάρει το δημιουργημένο ID token του χρήστη για την τρέχουσα περίοδο σύνδεσης. Το middleware ελέγχει αν υπάρχει η επικεφαλίδα εξουσιοδότησης, αν υπάρχει τότε χρησιμοποιούμε τη μέθοδο verifyIdToken της firebase που ελέγχει αν το JWT που λάβαμε είναι έγκυρο, αν είναι έγκυρο ανακτούμε το uid του χρήστη και το περνάμε στο controller για να χρησιμοποιηθεί αν χρειάζεται στο αίτημα. Εάν δεν είναι έγκυρο, απαγορεύουμε στον χρήστη να συνεχίσει την αίτηση.

2.2.5 Error handler

Έχουμε μια συνάρτηση χειρισμού σφαλμάτων που χρησιμοποιούμε για να κεντρικοποιήσουμε τον χειρισμό σφαλμάτων μέσα στο αρχείο server.ts.
Περνάμε τους controllers μας ως παράμετρο σε αυτή τη συνάρτηση, αυτή η συνάρτηση στη συνέχεια επιστρέφει μια άλλη συνάρτηση η οποία εκτελεί τον controller και αν είναι επιτυχής τελειώνει κανονικά, αν ένα σφάλμα δημιουργείται μέσα στον controller τότε καλούμε την επόμενη συνάρτηση η οποία περνάει το σφάλμα στο middleware χειρισμού σφαλμάτων που έχουμε ορίσει στο server.ts μας.
Ένας άλλος λόγος για τον οποίο χρησιμοποιούμε τη συνάρτηση χειρισμού σφαλμάτων είναι για να αποφύγουμε να τυλίγουμε τον κώδικα κάθε ελεγκτή σε ένα μπλοκ try/catch, αντ' αυτού απλά «πετάμε» ένα σφάλμα με τον κατάλληλο http status code και μήνυμα και ο error handler κάνει τα υπόλοιπα. Εξακολουθούμε να χρησιμοποιούμε ένα μπλοκ try/catch σε ορισμένα μέρη του κώδικα όπου θέλουμε να στείλουμε ένα προσαρμοσμένο μήνυμα σφάλματος, για παράδειγμα όταν ανεβάζουμε ένα αρχείο το τυλίγουμε σε ένα μπλοκ try/catch για να στείλουμε ένα απλό μήνυμα ότι η φόρτωση του αρχείου απέτυχε.

2.2.6 Validators

Έχουμε validators στον κωδικα μας για να επικυρώνουμε τα εισερχόμενα δεδομένα σε αιτήσεις POST, για αυτό χρησιμοποιούμε τη βιβλιοθήκη Zod. Απλά ορίζουμε ένα σχήμα για τα εισερχόμενα δεδομένα που περιμένουμε να λάβουμε, για παράδειγμα, ένα αντικείμενο με όνομα ιδιότητας τύπου string, στη συνέχεια στον ελεγκτή μας όπου χρησιμοποιούμε αυτόν τον επικυρωτή εξάγουμε τα δεδομένα από το σώμα της αίτησης και τα περνάμε στη μέθοδο ανάλυσης του Zod για να εκτελέσει μια επικύρωση στο σχήμα μας. Αν δεν είναι επιτυχής, πετάμε ένα σφάλμα με ένα προσαρμοσμένο μήνυμα και μια λίστα από προβλήματα, τα προβλήματα εξηγούν στον client ποια πεδία λείπουν ή έχουν ασύμβατους τύπους. Εάν η επικύρωση είναι επιτυχής, χρησιμοποιούμε τα επικυρωμένα δεδομένα στον controller μας.

2.2.6 Router

Στο API μας, έχουμε τις ακόλουθες γενικές διαδρομές:

- /api/chat
- /api/user
- /api/note
- /api/reply
- /api/schools

Κάθε endpoint έχει υπο-endpoints. Για παράδειγμα, μια αίτηση GET στο /api/user/:id

θα επέστρεφε τον χρήστη με το συγκεκριμένο ID.

Για τη διαχείριση αυτών των paths, έχουμε ένα ξεχωριστό αρχείο δρομολόγησης για κάθε ένα από τα πέντε κύρια paths, όπως chat.ts, user.ts, κ.λπ. Μέσα σε κάθε αρχείο δρομολόγησης, αντιστοιχίζουμε τους controllers στα αντίστοιχα paths για κάθε υπο-endpoint.

Τα endpoints μας έχουν πρόθεμα /api/, οπότε η πλήρης διαδρομή θα είναι 'www.example.com/api/user/'.

Για να το πετύχουμε αυτό, εισάγουμε και τα πέντε αρχεία δρομολόγησης στο middleware δρομολόγησης, το οποίο αντιστοιχίζει κάθε αρχείο δρομολόγησης στην αντίστοιχη διαδρομή του (π.χ. το /chat αντιστοιχίζεται στο αρχείο chat.ts). Στη συνέχεια, εισάγουμε το γενικό αρχείο δρομολόγησης στο αρχείο server.ts και το αντιστοιχίζουμε στη διαδρομή /api.

Με αυτόν τον τρόπο, δημιουργούμε μια προσαρμοσμένη διαδρομή /api/* για κάθε endpoint. Χωρίς αυτή τη ρύθμιση, τα τελικά μας σημεία θα έμοιαζαν απλά με 'www.example.com/user/:id'.

Το πλεονέκτημα αυτής της προσέγγισης είναι ότι μας επιτρέπει να δίνουμε version στα endpoints μας. Για παράδειγμα, θα μπορούσαμε να ονομάσουμε τα τρέχοντα endpoints /api/v1. Αργότερα, θα μπορούσαμε να δημιουργήσουμε ένα άλλο middleware δρομολόγησης και να εισάγουμε τις διαδρομές για τη version 2 του API μας, επιτρέποντάς μας να αλλάζουμε και να διαμορφώνουμε σταδιακά το API μας ανάλογα με τις ανάγκες.

2.2.7 Controllers

Οι controllers είναι συναρτήσεις που είναι υπεύθυνες για την εκπλήρωση των αιτημάτων σε κάθε endpoint. Κάνουν πράγματα όπως

- σύνδεση με τη βάση δεδομένων
- ανάγνωση και επεξεργασία δεδομένων πριν από την αποστολή τους στον client
- Ενημέρωση δεδομένων, π.χ. ενημέρωση του ονόματος προφίλ ενός χρήστη.
- διαγραφή δεδομένων από τη βάση δεδομένων κ.λπ.
- Χειρισμός σφαλμάτων
- Επικύρωση των εισερχόμενων δεδομένων χρησιμοποιώντας τα σχήματα επικύρωσης που έχουμε ορίσει

Στην εφαρμογή μας έχουμε έναν διαφορετικό controller για κάθε endpoint, όλοι αυτοί οι controllers συνδέονται με τη βάση δεδομένων postgresql και μέσω του prisma client μας εκτελούν την απαραίτητη λειτουργία της βάσης δεδομένων. Είναι οργανωμένοι στο φάκελο controllers με υποφακέλους για controllers για κάθε σημαντική διαδρομή για παράδειγμα:

Η διαδρομή /api/users εισάγει τους controllers από το φάκελο controllers/users.

Κάθε controller είναι ένα ξεχωριστό αρχείο μέσα στον αντίστοιχο υποφάκελο της διαδρομής.

2.2.8 Database Queries

Εκτελούμε τις λειτουργίες της βάσης δεδομένων χρησιμοποιώντας τον prisma client, ο client αρχικοποιείται μέσα στο φάκελο prisma που περιέχει επίσης το σχήμα της βάσης δεδομένων και τα migrations. Οι κλήσεις της βάσης δεδομένων οργανώνονται στο φάκελο db/queries με υποφακέλους για τα ερωτήματα που εκτελούνται σε συγκεκριμένους πίνακες της βάσης δεδομένων, για παράδειγμα ο υποφάκελος db/queries/user περιέχει ερωτήματα που εκτελούνται στον πίνακα user

Αυτά τα ερωτήματα είναι συναρτήσεις που λαμβάνουν παραμέτρους και χρησιμοποιούν τον prisma client για να εκτελέσουν λειτουργίες της βάσης δεδομένων στους αντίστοιχους πίνακες.

Εκτελούμε ερωτήματα όπως:

- Εύρεση μιας μοναδικής καταχώρησης χρησιμοποιώντας τη μέθοδο prisma.TABLENAME.findUnique
- Εύρεση πολλαπλών καταχωρίσεων χρησιμοποιώντας τη μέθοδο prisma.TABLENAME.findMany
- Εκτέλεση συναλλαγών για ενημέρωση,διαγραφή ή για ενημέρωση/δημιουργία εγγραφών ταυτόχρονα με τη μέθοδο prisma.$transaction και περνώντας ως παράμετρο έναν πίνακα με τα ερωτήματα prisma που πρέπει να εκτελέσουμε.
- Δημιουργία καταχωρήσεων με τη μέθοδο prisma.TABLENAME.create
- Ενημέρωση καταχωρήσεων με τη μέθοδο prisma.TABLENAME.update
- Διαγραφή καταχώρησης με χρήση της μεθόδου prisma.TABLENAME.delete

2.2.9 Socket.io

Το Socket Client αρχικοποιείται στο αρχείο server.ts με τη μέθοδο Server που κάνουμε import από το socket.io περνώντας τον express server μας ως παράμετρο.
Χρησιμοποιούμε 3 μεθόδους του socket.io για τη συνομιλία,

- τη μέθοδο **on** η οποία παρακολουθεί ένα event που έχουμε ορίσει και μια συνάρτηση με τον κώδικα που θα εκτελεστεί όταν το event που έχουμε ορίσει μας σταλεί από τον front-end client.
- Η μέθοδος **emit** η οποία εκπέμπει ένα event για να το λάβει ο client με όποια δεδομένα θέλουμε να στείλουμε.
- Η συνάρτηση **use** η οποία λαμβάνει ένα middleware για το socket μας.

Στο middleware μας ελέγχουμε:

- αν ο χρήστης που έχει συνδεθεί στο socket μας υπάρχει στη βάση δεδομένων μας
- Εάν ο χρήστης υπάρχει και είναι ήδη συνδεδεμένος στο socket, αφαιρούμε την προηγούμενη καταχώρηση για να αποφύγουμε τις διπλές συνδέσεις
- Τέλος, αποθηκεύουμε τον χρήστη στη λίστα των συνδεδεμένων χρηστών του socket μας με το ID της βάσης δεδομένων του ως αναγνωριστικό του socket του

Ακούμε 3 events στη μέθοδο on:

- 'Connection' event το οποίο αρχικοποιεί τον socket server και μέσα σε αυτό εκτελούμε όλες τις συναρτήσεις μας και άλλες μεθόδους socket
- event 'Dm', ο client στέλνει ένα event dm όταν στέλνεται ένα μήνυμα, μέσα σε αυτή τη συνάρτηση αποθηκεύουμε το μήνυμα στη βάση δεδομένων και το εκπέμπουμε στον χρήστη που λαμβάνει το μήνυμα αφού έχουμε επικυρώσει ότι έχει ήδη δημιουργηθεί μια συνομιλία μεταξύ των δύο χρηστών
- event 'Disconnect' το οποίο αφαιρεί τον χρήστη από τον socket server αφού έχει αποχωρήσει

Έχουμε ένα event στη μέθοδο emit:

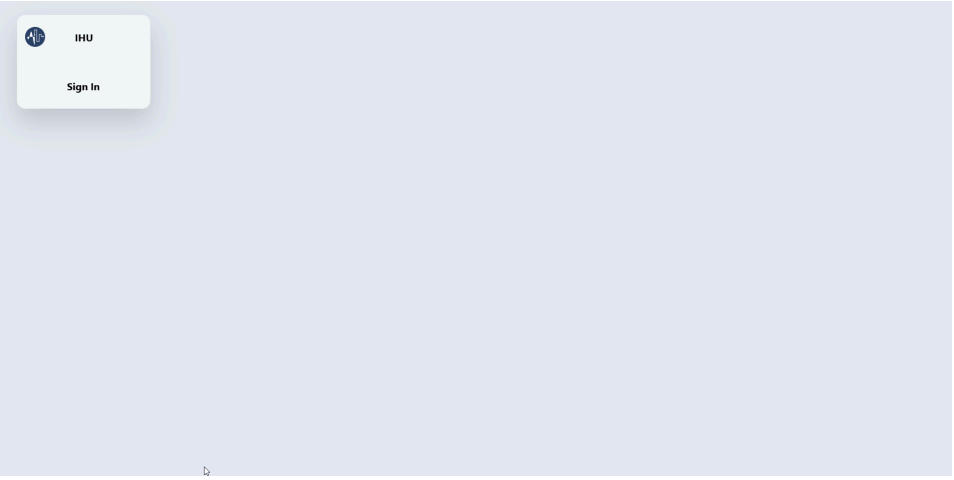
- «Users» event το οποίο στέλνει τη λίστα των συνδεδεμένων χρηστών στο socket στον client.

Κεφάλαιο 3: Οδηγίες χρήσης

Σε αυτό το κεφάλαιο θα δείξουμε πώς χρησιμοποιείται η εφαρμογή

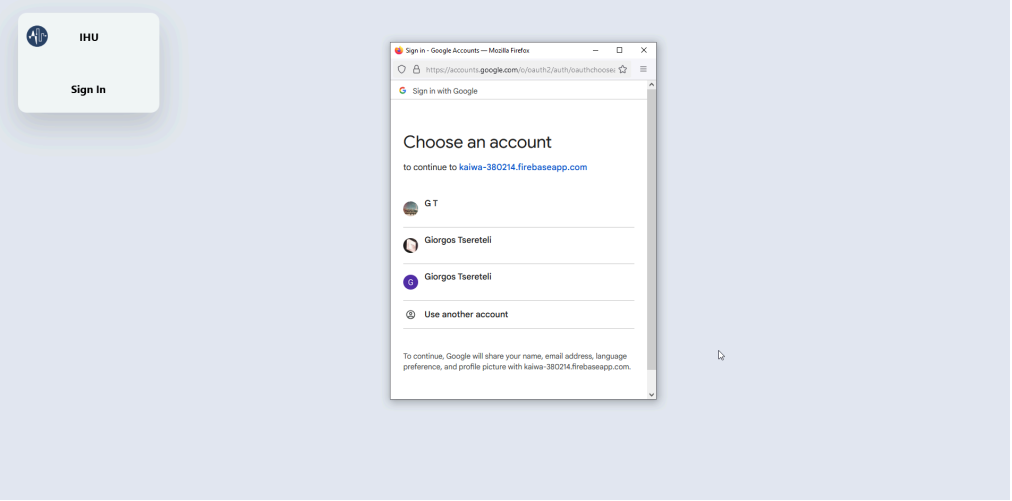
3.1 Σύνδεση

Στην αρχική σελίδα έχουμε μια κενή σελίδα εκτός από μια μπάρα πλοήγησης όπου μπορείτε να κάνετε κλικ στο κουμπί σύνδεσης για να συνδεθείτε αν δεν είστε ήδη



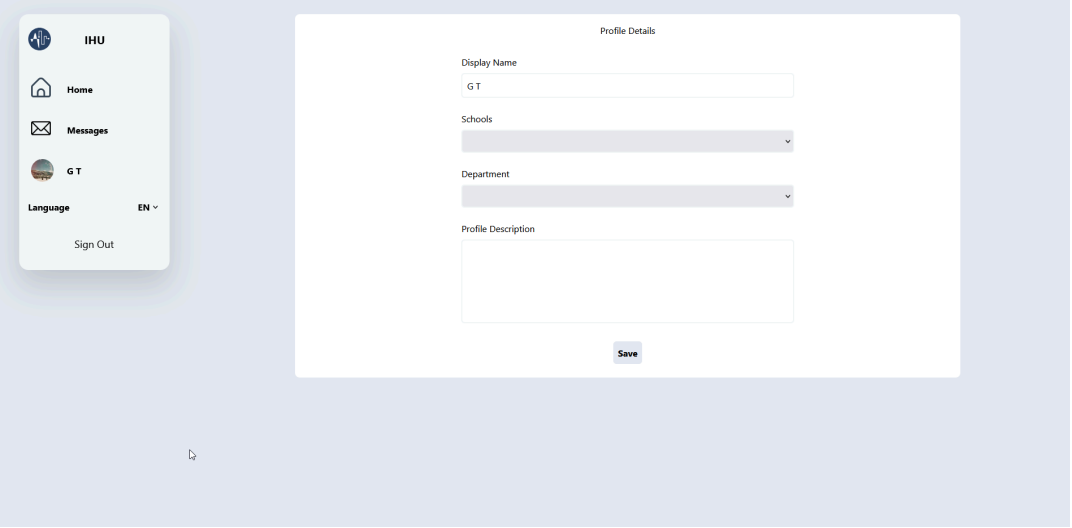
Σχήμα 3.1: Αρχική σελίδα

Όταν κάνετε κλικ στο κουμπί, θα εμφανιστεί ένα popup σύνδεσης του παρόχου google, εδώ μπορείτε να επιλέξετε το λογαριασμό σας στο google για να συνδεθείτε στην εφαρμογή με τον οποίο θα συνδεθείτε.



Σχήμα 3.2: Popup σύνδεσης

Αφού επιλέξετε έναν λογαριασμό θα μεταφερθείτε στην αρχική σελίδα, η εφαρμογή ελέγχει αν έχετε ολοκληρώσει τη διαδικασία εγγραφής,αν συνδέεστε για πρώτη φορά η διαδικασία εγγραφής δεν έχει ολοκληρωθεί οπότε εμφανίζουμε μια οθόνη όπου μπορείτε να συμπληρώσετε τα στοιχεία σας όπως το όνομά σας, μια σύντομη περιγραφή για τον εαυτό σας και τη δυνατότητα να επιλέξετε τη σχολή και το τμήμα σας

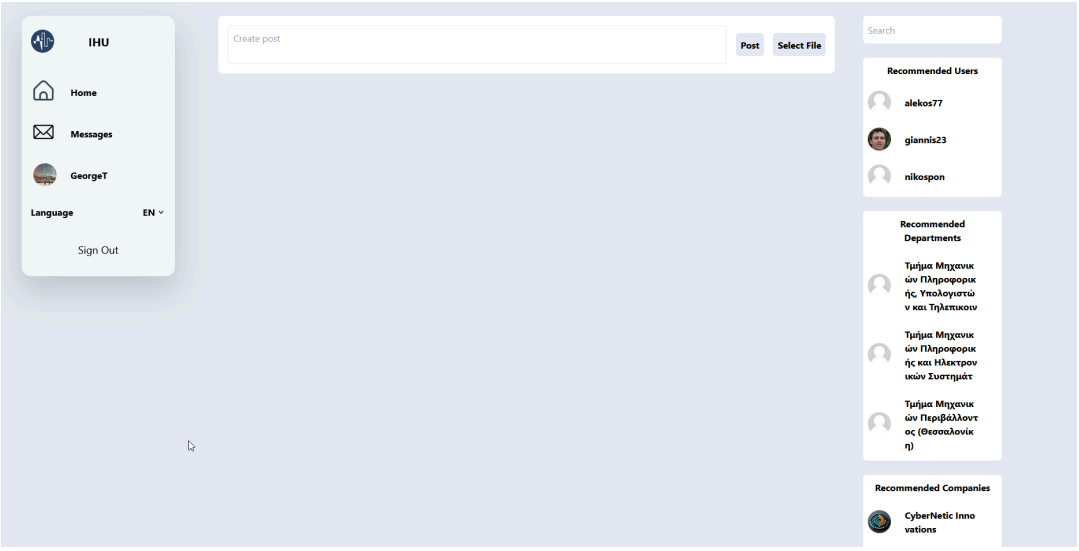


Σχήμα 3.3: Profile Details

Αφού συμπληρώσετε τα στοιχεία σας και αποθηκεύσετε θα μεταφερθείτε στην αρχική σελίδα

3.2 Αρχική σελίδα

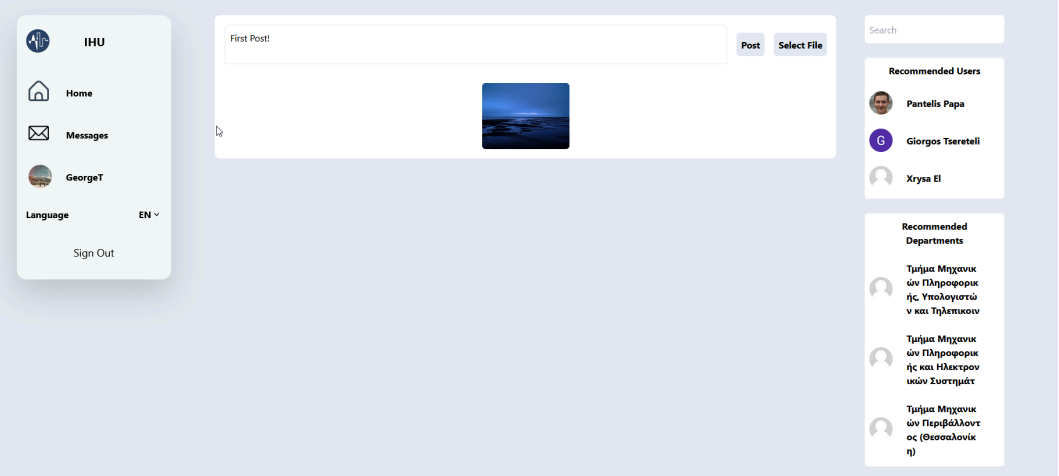
Η αρχική σελίδα έχει 3 τμήματα



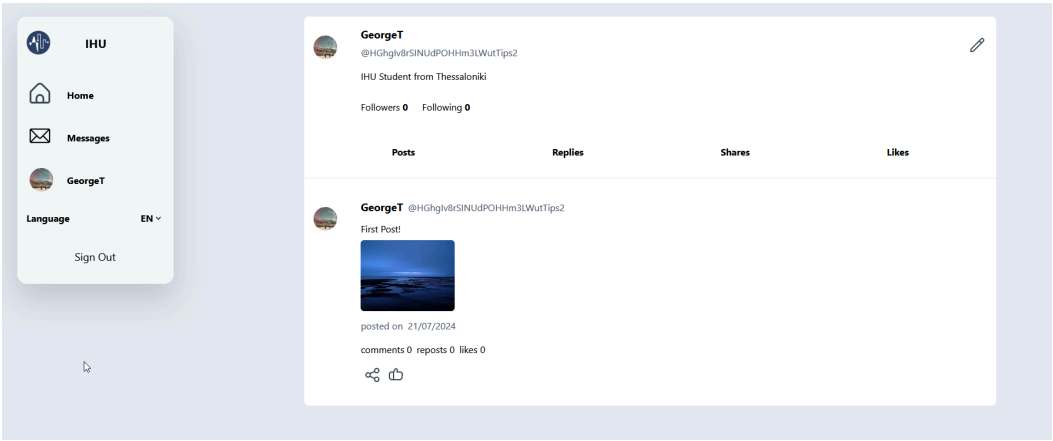
Σχήμα 3.4: Αρχική σελίδα

Αριστερά το navbar με κουμπιά για να μεταβείτε στην αρχική σελίδα, στα μηνύματά σας, στο προφίλ σας, να επιλέξετε γλώσσα και να αποσυνδεθείτε.

Στη μέση έχουμε ένα τμήμα για τη δημιουργία μιας δημοσίευσης,
Εδώ μπορείτε να δημιουργήσετε μια ανάρτηση, μετά τη δημιουργία της μπορείτε να την δείτε στο προφίλ σας.

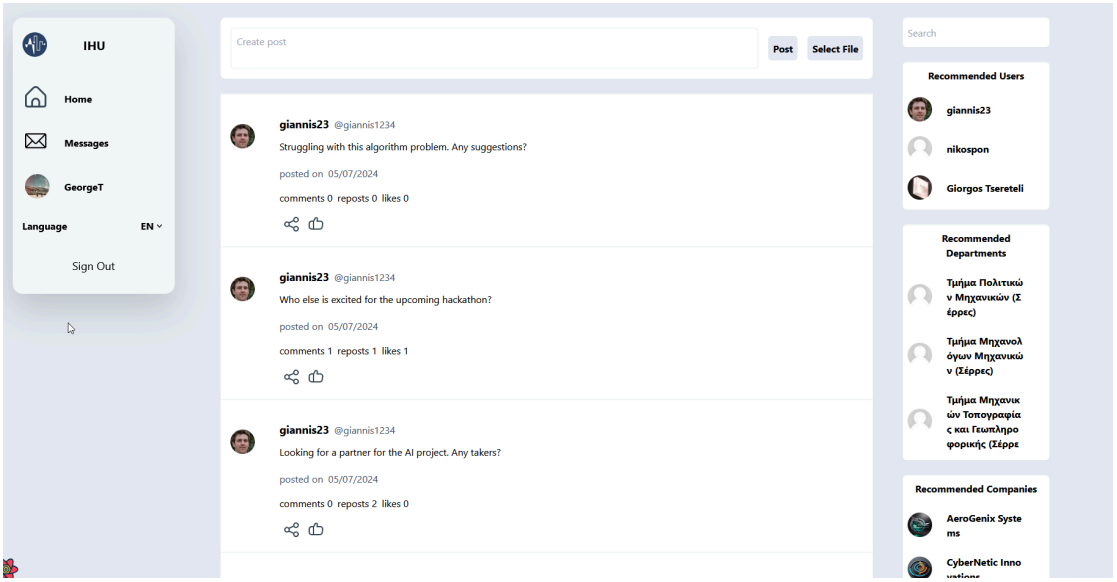


Σχήμα 3.5: Δημιουργία Ανάρτησης



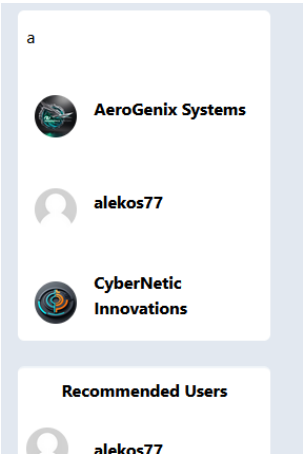
Σχήμα 3.6: Ανάρτηση στο προφίλ του χρηστη

Κάτω από αυτό θα εμφανιστεί μια λίστα με τις δημοσιεύσεις όταν ακολουθείτε κάποιον.



Σχήμα 3.7: Δημοσιεύσεις από άτομα που ακολουθείτε

Στα δεξιά έχουμε μια μπάρα αναζήτησης που αναζητά χρήστες με βάση το όνομα τους , και 3 τμήματα συστάσεων, ένα για να σας προτείνουμε χρήστες από το τμήμα σας, ένα για να σας προτείνουμε τα επίσημα προφίλ των τμημάτων του πανεπιστημίου, ένα για να σας προτείνουμε εταιρείες που συνεργάζονται με το πανεπιστήμιο



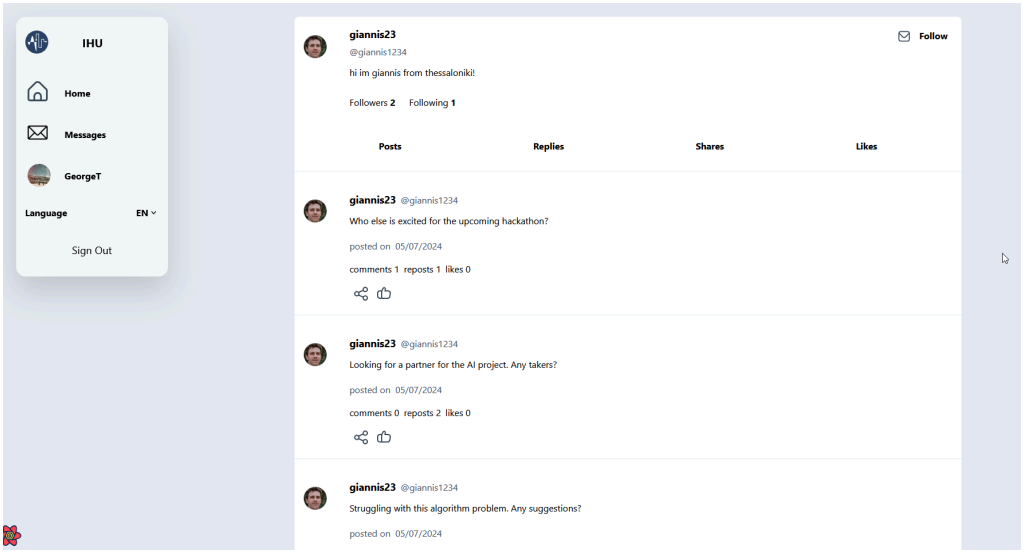
Σχήμα 3.8: Αναζήτηση χρηστών

3.3 Προφίλ χρηστών

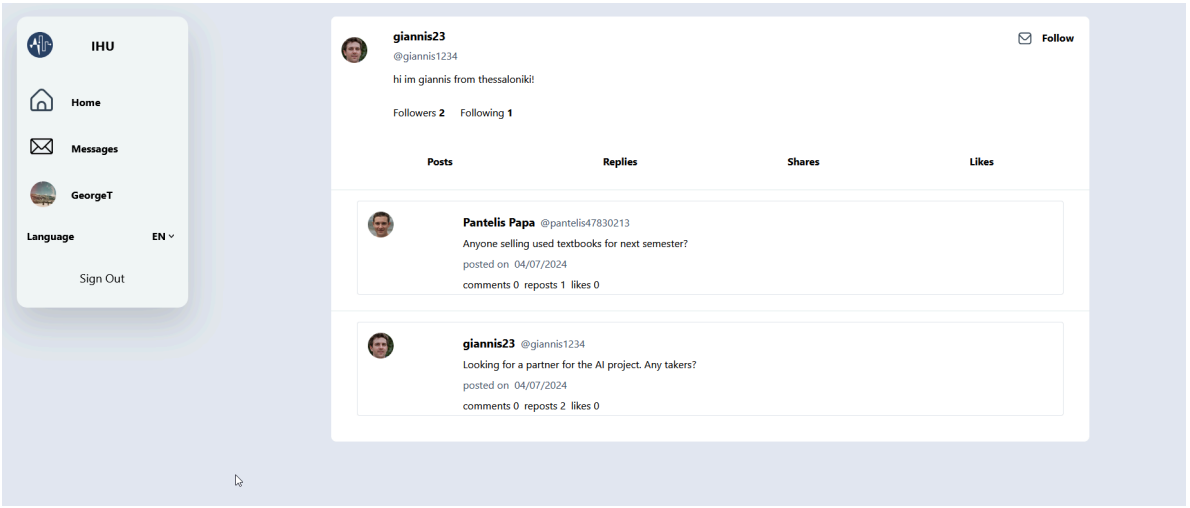
Κάνοντας κλικ σε έναν από τους χρήστες μπορείτε να πλοηγηθείτε στο προφίλ του, εδώ μπορείτε να δείτε τις πληροφορίες των χρηστών, τα άτομα που ακολουθούν, τα άτομα που τους ακολουθούν και 4 πεδία που περιέχουν τις δημοσιεύσεις των χρηστών, τις δημοσιεύσεις στις οποίες έχουν απαντήσει, τις δημοσιεύσεις που έχουν μοιραστεί και τους έχουν κάνει like. Επιπλέον, υπάρχουν δύο κουμπιά, ένα για να ακολουθήσετε/ακυρώσετε την ακολουθία του χρήστη και ένα για να στείλετε μήνυμα στον χρήστη.

Εάν βρίσκεστε στο προφίλ σας, θα υπάρχει μόνο ένα κουμπί για την επεξεργασία των στοιχείων του προφίλ σας, το οποίο σας μεταφέρει σε μια άλλη σελίδα για να επεξεργαστείτε τα στοιχεία του προφίλ σας.

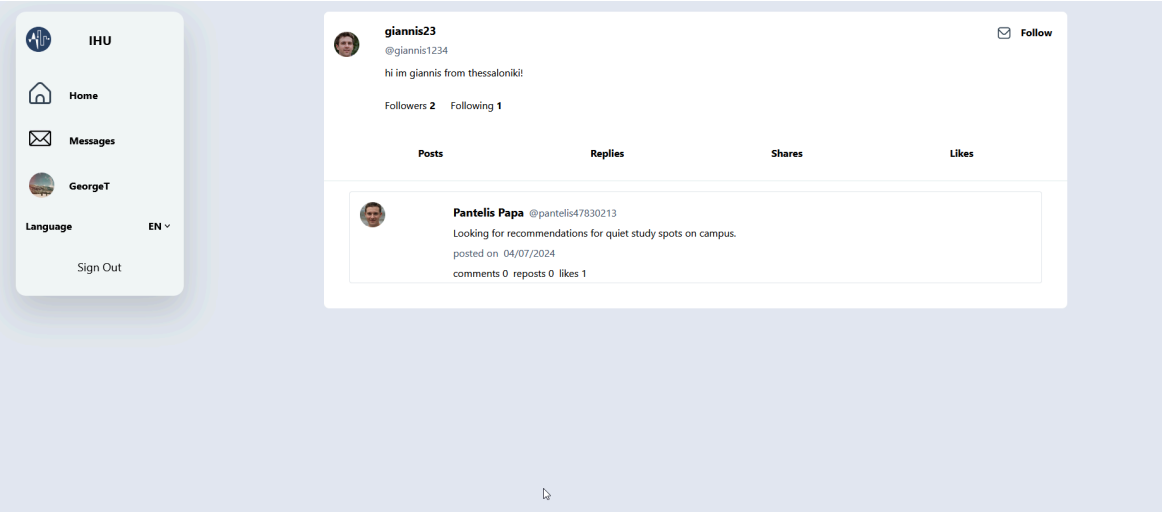
Τα προφίλ των τμημάτων και των εταιρειών έχουν ένα εικονίδιο δίπλα στο όνομά τους που επιβεβαιώνει ότι είναι επίσημοι λογαριασμοί



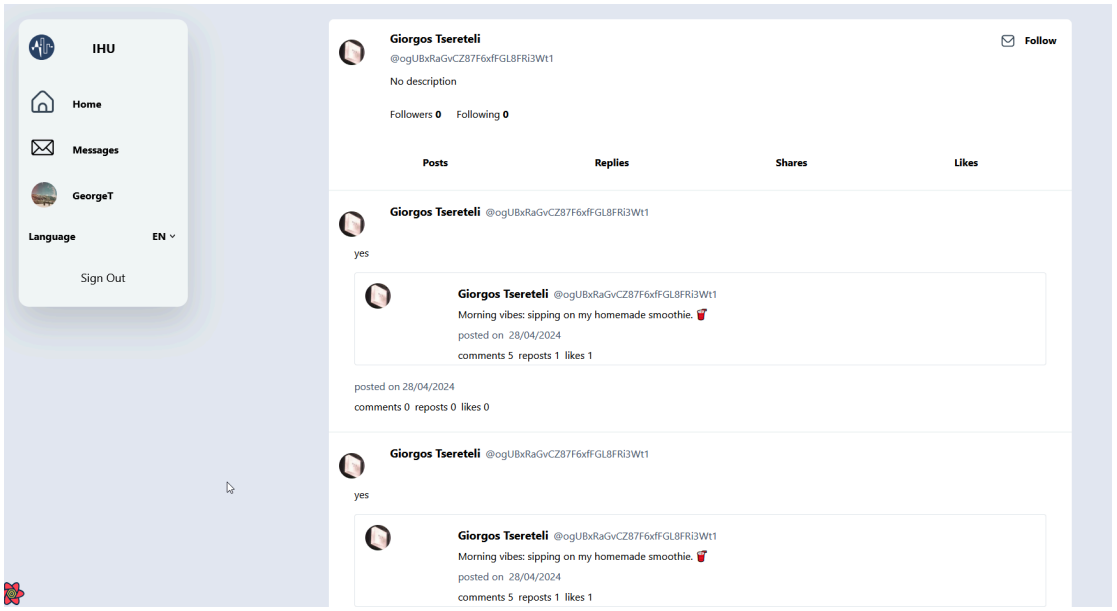
Σχήμα 3.9: Προφιλ χρήστη και οι δημοσιεύσεις του



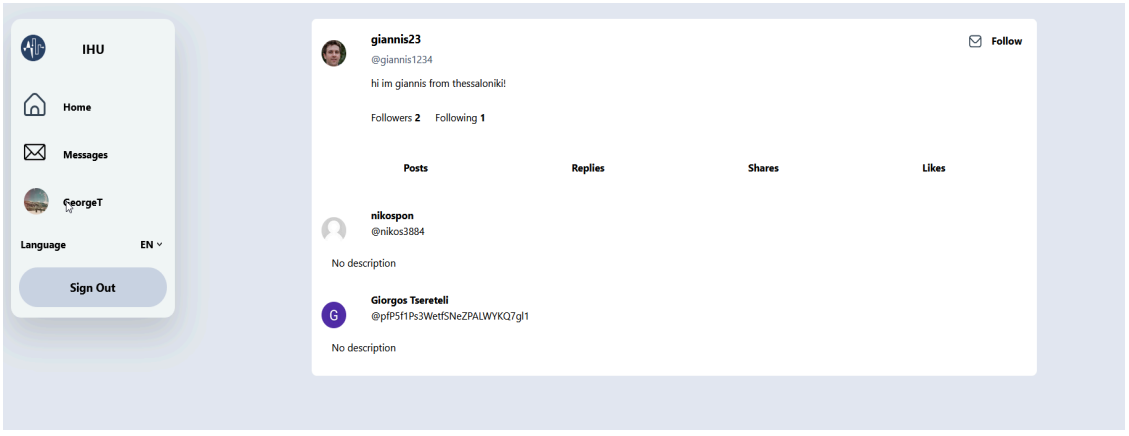
Σχήμα 3.10: Δημοσιεύσεις που έχουν μοιραστεί



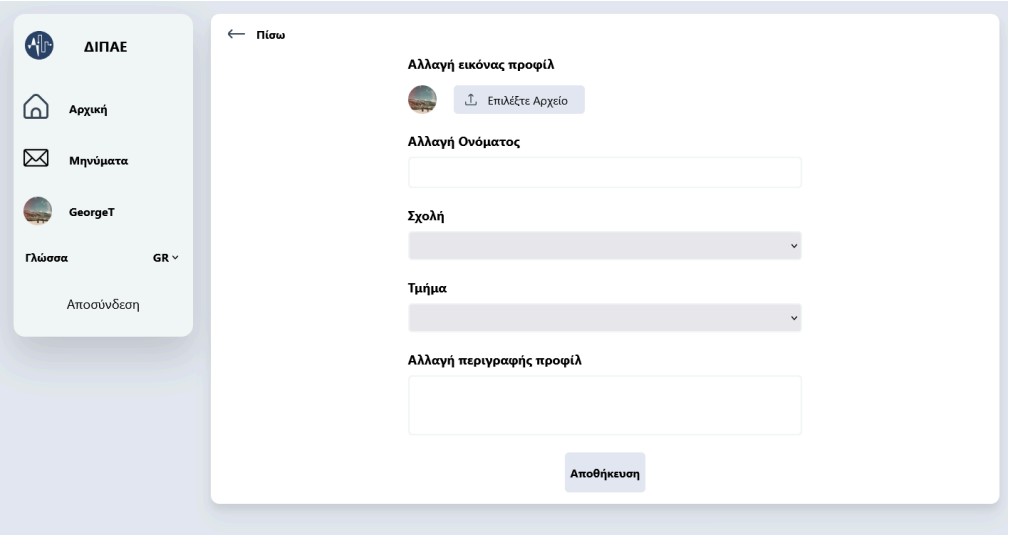
Σχήμα 3.11: Δημοσιεύσεις που έχουν κάνει like



Σχήμα 3.12: Δημοσιεύσεις στις οποίες έχουν απαντήσει



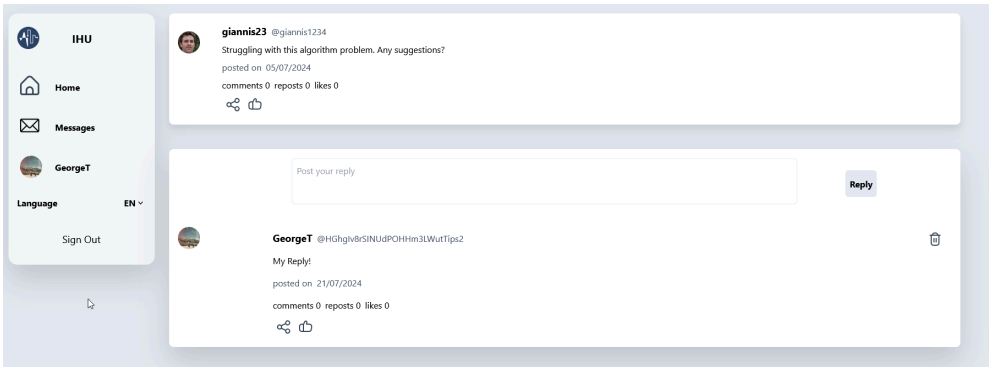
Σχήμα 3.13: Ακόλουθοι του χρήστη, το τμήμα "ακολουθεί" έχει τον ίδιο σχεδιασμό



Σχήμα 3.14: Επεξεργασία των στοιχείων του προφίλ

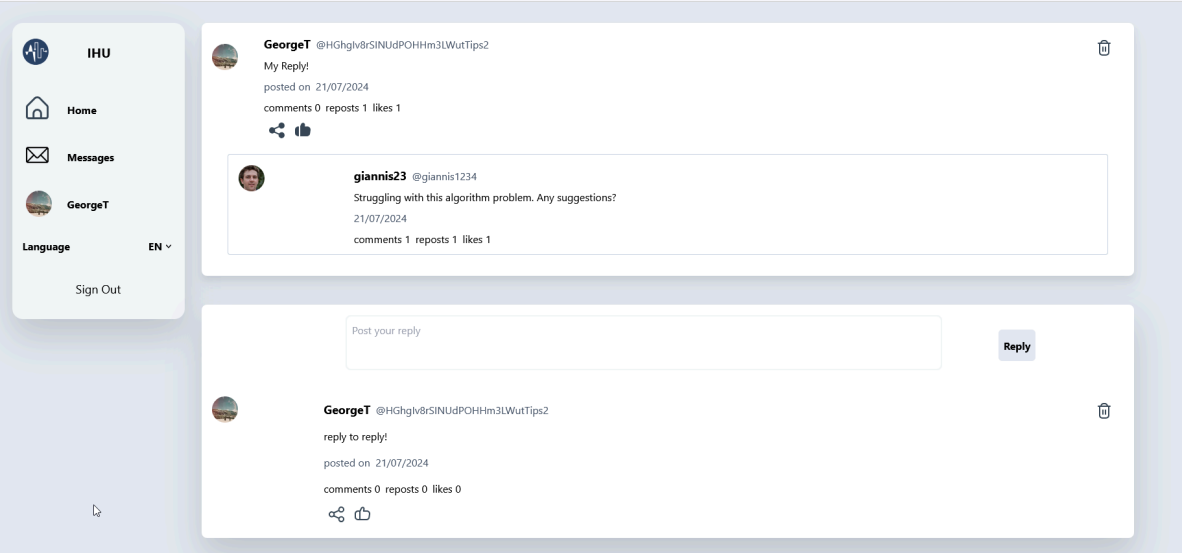
3.4 Αναρτήσεις

Κάνοντας κλικ σε μια ανάρτηση μπορούμε να δούμε το περιεχόμενό της και πότε δημιουργήθηκε, μπορούμε να της κάνουμε like, να τη μοιραστούμε και να απαντήσουμε σε αυτήν.



Σχήμα 3.14: Ανάρτηση

Μπορούμε να κάνουμε κλικ σε μια απάντηση η οποία μας μεταφέρει στη σελίδα της απάντησης όπου μπορούμε να δούμε το περιεχόμενο της απάντησης και την ανάρτηση/απάντηση στην οποία απάντησε ο χρήστης. Μπορούμε να της κάνουμε like, να τη μοιραστούμε και να απαντήσουμε σε αυτή την απάντηση. Μπορούμε και πάλι να κάνουμε κλικ στη νέα απάντηση για να δούμε το περιεχόμενό της κ.λπ. επ' άπειρον.

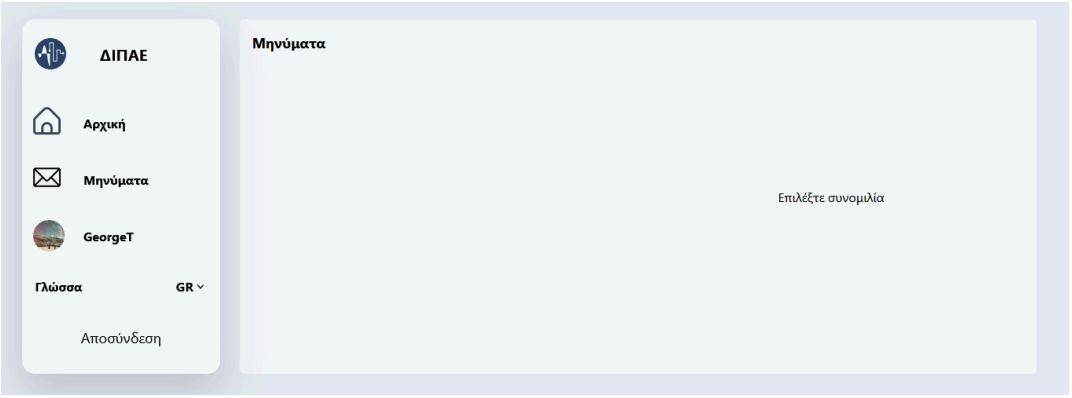


Σχήμα 3.15: Απάντηση

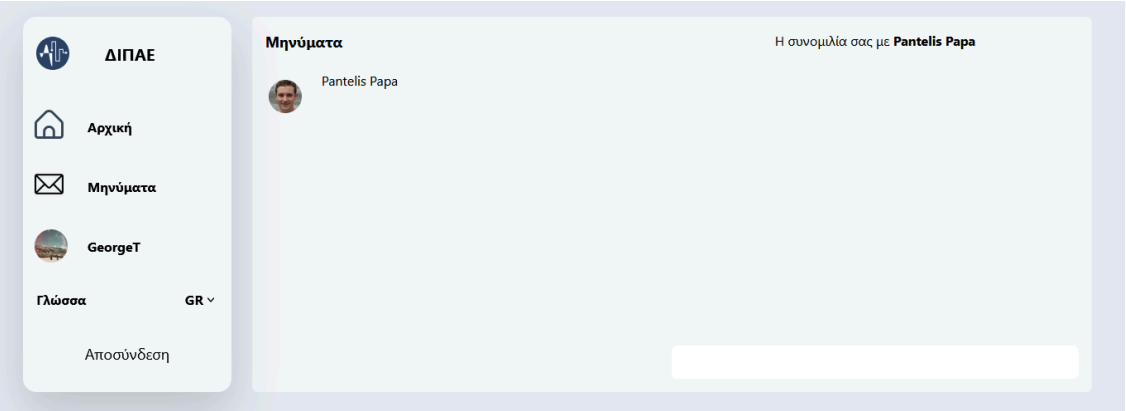
Μπορούμε να διαγράψουμε τις αναρτήσεις/απαντήσεις που έχουμε δημιουργήσει

3.5 Μηνύματα

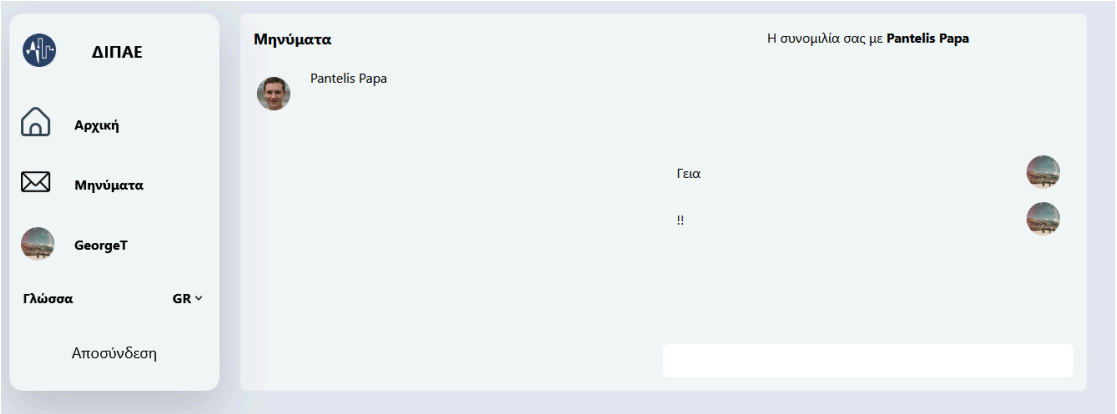
Κάνοντας κλικ στο κουμπί μήνυμα στη μπάρα πλοήγησης μεταφερόμαστε στη σελίδα συνομιλίας, η οποία αρχικά είναι κενή. Εάν μεταβούμε στο προφίλ ενός χρήστη και κάνουμε κλικ στο κουμπί με το εικονίδιο του mail, μεταφερόμαστε στη σελίδα συνομιλίας και δημιουργούμε μια νέα συνομιλία με αυτόν τον χρήστη.



Σχήμα 3.16: σελίδα συνομιλίας χωρίς συνομιλίες



Σχήμα 3.17: σελίδα συνομιλίας με μια συνομιλία



Σχήμα 3.18: σελίδα συνομιλίας με μια συνομιλία

Αυτή η συνομιλία χρησιμοποιεί το socket.io και είναι σε πραγματικό χρόνο. Τα μηνύματα αποθηκεύονται στη βάση δεδομένων, οπότε αν ο χρήστης δεν είναι συνδεδεμένος, θα τα δει όταν συνδεθεί.

Κεφάλαιο 4: Συμπεράσματα και προτάσεις για βελτιώσεις

Ο στόχος αυτής της εργασίας ήταν να δημιουργηθεί μια πλατφόρμα που θα μπορούσε να είναι χρήσιμη στους φοιτητές.

Η πλατφόρμα δίνει τη δυνατότητα στους φοιτητές να αλληλεπιδρούν μεταξύ τους, να ζητούν βοήθεια για τα μαθήματα και να βρίσκουν άτομα για να συνεργαστούν σε διάφορα projects, να ενημερώνονται για τα νέα των τμημάτων/πανεπιστημίου μέσω των λογαριασμών των τμημάτων, επιπλέον η πλατφόρμα δίνει τη δυνατότητα στους χρήστες να βρίσκουν ενδιαφέρουσες εταιρείες στο κλάδο μας, να ενημερώνονται για ευκαιρίες πρακτικής άσκησης και θέσεις εργασίας μέσω των προφίλ τους.

Θα μπορούσαμε να προσθέσουμε επιπλέον βελτιώσεις στην εφαρμογή, όπως:

- Διαχειριστικό περιβάλλον:

  Στην εφαρμογή μας έχουμε λογαριασμούς για τα πανεπιστημιακά τμήματα και τις εταιρείες που συνεργάζονται με το πανεπιστήμιό μας, αλλά προς το παρόν είναι "hard coded" μέσω της βάσης δεδομένων. Θα μπορούσαμε να έχουμε ένα διαχειριστικό περιβάλλον για το πανεπιστήμιο και τα τμήματα όπου θα μπορούσαν να κάνουν πράγματα όπως, να δημιουργούν διαπιστευτήρια σύνδεσης για τις εταιρείες και τους χρήστες, να έχουν μια λίστα με τις αναρτήσεις που ανεβαίνουν ώστε να μπορούν να τις παρακολουθούν και να διαγράφουν τυχόν ακατάλληλο περιεχόμενο. Το διαχειριστικό "superadmin" θα μπορούσε να δημιουργεί λογαριασμούς για τα τμήματα ή τις σχολές και κάθε τμήμα/σχολή θα μπορούσε να έχει το δικό του διαχειριστικό περιβάλλον ώστε να μπορεί να κάνει πράγματα που αφορούν μόνο το τμήμα/σχολή τους.

  Επιπλέον, θα μπορούσαμε επίσης να έχουμε επαληθευμένους λογαριασμούς για τους καθηγητές
- Πρόσθετες μέθοδοι σύνδεσης:

  Προς το παρόν είμαστε σε θέση να συνδεθούμε μόνο μέσω ενός λογαριασμού google, θα μπορούσαμε να προσθέσουμε άλλους τρόπους σύνδεσης, για παράδειγμα έναν τρόπο σύνδεσης με τα διαπιστευτήρια του πανεπιστημίου μας ή ο διαχειριστής θα μπορούσε να δημιουργήσει διαπιστευτήρια για κάθε χρήστη ή θα μπορούσαμε να έχουμε μια απλή σύνδεση με email/password, αλλά μόνο με το email του πανεπιστημίου μας.
- Σελίδα θέσης εργασίας:

  Θα μπορούσαμε να έχουμε μια σελίδα όπου οι λογαριασμοί των εταιρειών θα μπορούν να προσθέτουν θέσεις εργασίας

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] https://legacy.reactjs.org/docs/faq-internals.html

[2] https://react.dev/learn/writing-markup-with-jsx

[3] https://react.dev/learn/state-a-components-memory

[4] https://react.dev/learn/synchronizing-with-effects

[5]https://react.dev/learn/passing-data-deeply-with-context

[6]https://react.dev/learn/extracting-state-logic-into-a-reducer

[7] https://nodejs.org/en

[8] https://www.npmjs.com/about

[9] https://expressjs.com/

[10] https://expressjs.com/en/starter/basic-routing.html

[11] https://expressjs.com/en/4x/api.html#req

[12] https://expressjs.com/en/4x/api.html#res

[13] https://expressjs.com/en/guide/writing-middleware.html

[14] https://expressjs.com/en/starter/static-files.html

[15] https://nextjs.org/docs

[16] https://nextjs.org/docs/pages/building-your-application/routing/pages-and-layouts

[17] https://nextjs.org/docs/pages/building-your-application/routing/linking-and-navigating

[18] https://nextjs.org/docs/pages/building-your-application/routing/redirecting

[19] https://nextjs.org/docs/pages/building-your-application/rendering/server-side-rendering

[20] https://nextjs.org/docs/pages/building-your-application/rendering/static-site-generation

[21] https://nextjs.org/docs/pages/building-your-application/rendering/server-side-rendering

[22]https://nextjs.org/docs/pages/building-your-application/optimizing/images

[23] https://nextjs.org/docs/pages/building-your-application/routing/api-routes

[24] https://vuejs.org/guide/scaling-up/ssr.html

[25] https://www.gatsbyjs.com/docs/glossary/static-site-generator/

[26] https://nextjs.org/learn-pages-router/basics/api-routes

[27] https://www.typescriptlang.org/

[28] https://www.postgresql.org/

[29] https://www.prisma.io/docs/orm

[30] https://firebase.google.com/

[31] https://firebase.google.com/docs/auth

[32] https://websockets.spec.whatwg.org/

[33] https://socket.io/

ΠΑΡΑΡΤΉΜΑΤΑ

Παραρτήματα απο βασικά κομμάτια του backend κώδικα.

user-exists.ts

```
import { Request, Response } from 'express';
import { prisma } from '../../prisma/client';

export const exists = async (req: Request, res: Response) => {
    const uid = res.locals.uid;
    const { googleID, email, displayName, photoURL } = req.body;

    const user = await prisma.user.findUnique({
        where: {
            id: uid,
        },
    });

    if (user) return res.status(200).json(user);

    const newUser = await prisma.user.create({
        data: {
            id: googleID,
            email: email,
            googleID: googleID,
            displayName: displayName,
            username: displayName,
            profileImage: photoURL,
        },
    });

    res.status(200).json(newUser);
};
```

finalize-registration.ts

```
import { Request, Response } from 'express';
import { getDepartment } from '../../db/queries/school/get-department';
import { getSchoolQuery } from '../../db/queries/school/get-school';
```

```
import { findUser } from '../../db/queries/user/get-user';
import { prisma } from '../../prisma/client';
import { finalizeRegistrationSchema } from
'../../validators/finalize-registration';

export const finalizeRegistration = async (req: Request, res: Response) => {
    const uid = res.locals.uid;

    const userExists = await findUser(uid);
    if (!userExists) throw { status: 404, message: 'user does not exist' };

    const result = finalizeRegistrationSchema.safeParse(req.body);
    if (!result.success) throw { status: 400, message: result.error.issues };

    const data = result.data;

    const departmentExists = await getDepartment(data.departmentId);
      if (!departmentExists) throw { status: 400, message: "department does not
exist" };

    const schoolExists = await getSchoolQuery(data.schoolId);
    if (!schoolExists) throw { status: 400, message: 'School does not exist' };

    const user = await prisma.user.update({
        where: {
            id: uid
        },
        data: { ...data, hasRegistered: true }

    });

    res.status(200).json({ user });
};
```

search-users.ts

```
import { Request, Response } from "express";
import { prisma } from "../../prisma/client";

export const searchUsers = async (req: Request, res: Response) => {
    console.log(req.body);
```

```
    const { username } = req.body;

    const users = await prisma.user.findMany({
        where: {
            displayName: {
                contains: username,
                mode: 'insensitive'
            },
            id: { not: res.locals.uid }

        },
        take: 3
    });

    res.status(200).json({ users });
};
```

recommended-users.ts

```
import { Request, Response } from 'express';
import { prisma } from '../../prisma/client';
import { Department, User } from '@prisma/client';
import { recommendedUsers } from '../../helpers/recommendations/recommended-user';
import { recommendedDepartments } from '../../helpers/recommendations/recommended-departments';
import { recommendedCompanies } from '../../helpers/recommendations/recommended-companies';

export const recommended = async (req: Request, res: Response) => {
    const uid = res.locals.uid;
    const { id } = req.params;

    const user = await prisma.user.findUnique({
        where: {
            id: uid
        }, select: {
            departmentId: true,
            schoolId: true,
```

```
        }
    });

    let recommended: User[] | Department[] = [];
    if (id === "users") {
          recommended = await recommendedUsers(uid, user?.departmentId as string,
user?.schoolId as string);
    }

    if (id === "departments") {
        recommended = await recommendedDepartments(user?.schoolId as string);
    }
    if (id === "companies") {
        recommended = await recommendedCompanies();
    }

    res.status(200).json({ recommended });
};
```

user-feed.ts

```
import { Request, Response } from 'express';
import { LIMIT } from '../../constants/firebase';
import { prisma } from '../../prisma/client';

export const feed = async (req: Request, res: Response) => {
    const uid = res.locals.uid;
    const { page } = req.query;

    const PAGE = parseInt(page as string);
    if (PAGE < 0 || isNaN(PAGE)) throw { status: 400, message: 'invalid param' };

    const followingUsers = await prisma.follows.findMany({
        where: {
            followerId: uid
        }, select: {
            followingId: true
        }
    });

    const postsCount = await prisma.post.count({
        where: {
```

```
        userId: {
            in: followingUsers.map(f => f.followingId)
        }
    }
});

let posts = await prisma.post.findMany({
    where: {
        userId: {
            in: followingUsers.map(f => f.followingId)
        }
    },
    include: {
        _count: {
            select: {
                replies: true,
                reactions: true,
                reposts: true,
            },
        },
        user: true,
    }, orderBy: {
        createdAt: 'desc'
    },
    skip: PAGE * LIMIT,
    take: LIMIT
});

posts = await Promise.all(posts.map(async post => {
    const likedPost = await prisma.reaction.findFirst({
        where: {
            userId: uid,
            postId: post.id,
        },
    });

    if (likedPost) {
        (post as any).hasLiked = true;
    } else {
        (post as any).hasLiked = false;
    }

    const sharedPost = await prisma.repost.findFirst({
```

```
                where: {
                    userId: uid,
                    postId: post.id
                }
            });

            if (sharedPost) {
                (post as any).hasShared = true;
            } else {
                (post as any).hasShared = false;
            }

            return post;
        }));

        const hasMorePages = postsCount > (PAGE + 1) * LIMIT;
        const nextPage = hasMorePages ? PAGE + 1 : null;
        res.status(200).json({ data: posts, nextPage });
    };
```

get-messages.ts

```
    import { Request, Response } from "express";
    import { prisma } from "../../prisma/client";
    import { LIMIT } from "../../constants/firebase";

    export const getMessages = async (req: Request, res: Response) => {
        const { id } = req.params;
        const { page } = req.query;

        const PAGE = parseInt(page as string);
        if (PAGE < 0) throw { status: 400, message: 'invalid param' };
        const SKIP_COUNT = LIMIT * PAGE;


        let messages = await prisma.message.findMany({
            where: {
                conversationId: id
            },
            orderBy: {
                createdAt: 'desc'
            }, take: LIMIT, skip: SKIP_COUNT
```

```
    });

    const totalMessages = await prisma.message.count({
        where: {
            conversationId: id
        },
    });

    messages = messages.reverse();

    const hasMorePages = totalMessages > (SKIP_COUNT + LIMIT);
    const nextPage = hasMorePages ? PAGE + 1 : null;

    res.status(200).json({ data: messages, nextPage });
};
```

create-note.ts

```
import { Request, Response } from 'express';
import { MAX_POST_IMAGES } from '../../constants/firebase';
import { filterImages } from '../../helpers/images/filter-images';
import { getImageBuffer } from '../../helpers/images/get-image-buffer';
import { imageUploader } from '../../helpers/images/image-uploader';
import { unlinkImages } from '../../helpers/images/unlink-images';
import { prisma } from '../../prisma/client';

export const createNote = async (req: Request, res: Response) => {
    const note = req.body;

     if (typeof note.content !== 'string') throw { status: 400, message: 'Invalid
data.' };

    const uid = res.locals.uid;
    let imgUrls: string[] = [];

    const images: Express.Multer.File[] = req.files as Express.Multer.File[];

    if (images && images.length) {
         if (images.length > MAX_POST_IMAGES) throw { status: 400, message: 'Max
images: 10' };
             const storagePath = `user/${uid}/note/${uid}_${note.content ?? new
Date().valueOf()}/images/${uid}_${new Date().valueOf()}`;
```

```
        const imagesToUpload = filterImages(images);
        const imageBuffers = getImageBuffer(imagesToUpload) as Buffer[];
        imgUrls = await imageUploader(imageBuffers, storagePath);
        unlinkImages(imagesToUpload);
    };
    const post = await prisma.post.create({
        data: { content: note.content, userId: uid, imageUrls: imgUrls },
    });

    res.status(200).json(post);
};
```