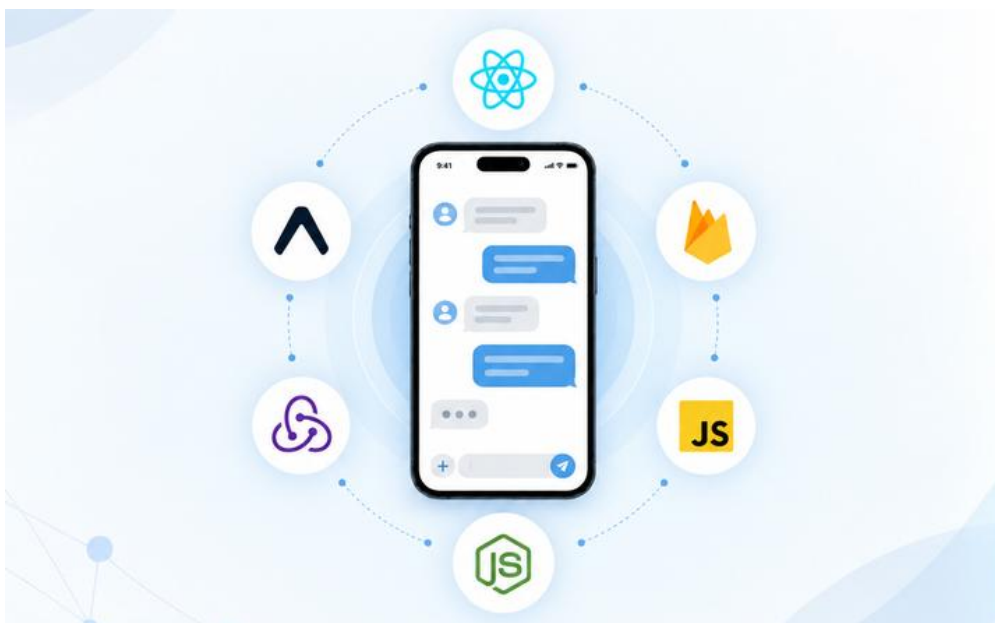


ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΗΛΕΚΤΡΟΝΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Ανάπτυξη Εφαρμογής Android/iOS Ανταλλαγής Μηνυμάτων



Του φοιτητή:
Χατζηαντωνίου Ιωάννη
Αρ. Μητρώου: 073239

Επιβλέπων
Ονοματεπώνυμο:
Κιοσκερίδης Ιορδάνης
Βαθμίδα: Καθηγητής

Ημερομηνία 6/9/2026

Τίτλος Π.Ε.

Ανάπτυξη Εφαρμογής Android/iOS Ανταλλαγής Μηνυμάτων

Κωδικός Π.Ε. 26247

Ονοματεπώνυμο φοιτητή
Χατζηαντωνίου Ιωάννης

Ονοματεπώνυμο εισηγητή
Κιοσκερίδης Ιορδάνης

Ημερομηνία ανάληψης Δ.Ε.
31-03-2026

Ημερομηνία περάτωσης Δ.Ε.
06-09-2026

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 πτυχιακή εργασία, στο Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του ΔΙ.ΠΑ.Ε.

Η παρούσα πτυχιακή εργασία αποτελεί πνευματική ιδιοκτησία του φοιτητή Χατζηαντωνίου Ιωάννη που την εκπόνησε. Στο πλαίσιο της πολιτικής ανοικτής πρόσβασης, ο συγγραφέας/δημιουργός εκχωρεί στο Διεθνές Πανεπιστήμιο της Ελλάδος άδεια χρήσης του δικαιώματος αναπαραγωγής, δανεισμού, παρουσίασης στο κοινό και ψηφιακής διάχυσης της εργασίας διεθνώς, σε ηλεκτρονική μορφή και σε οποιοδήποτε μέσο, για διδακτικούς και ερευνητικούς σκοπούς, άνευ ανταλλάγματος. Η ανοικτή πρόσβαση στο πλήρες κείμενο της εργασίας, δεν σημαίνει καθ' οιονδήποτε τρόπο παραχώρηση δικαιωμάτων διανοητικής ιδιοκτησίας του συγγραφέα/δημιουργού, ούτε επιτρέπει την αναπαραγωγή, αναδημοσίευση, αντιγραφή, πώληση, εμπορική χρήση, διανομή, έκδοση, μεταφόρτωση (downloading), ανάρτηση (uploading), μετάφραση, τροποποίηση με οποιονδήποτε τρόπο, τμηματικά ή περιληπτικά της εργασίας, χωρίς τη ρητή προηγούμενη έγγραφη συναίνεση του συγγραφέα/δημιουργού.

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

«Στην Μητέρα»

Πρόλογος

Η επιλογή του θέματος της παρούσας πτυχιακής εργασίας προέκυψε μέσα από το ενδιαφέρον μου για την ανάπτυξη εφαρμογών και τις σύγχρονες τεχνολογίες λογισμικού. Έχοντας ήδη αποκτήσει μια πρώτη εμπειρία με τεχνολογίες ανάπτυξης διαδικτυακών εφαρμογών, διαπίστωσα στη συνέχεια ότι αντίστοιχες γνώσεις μπορούν να αξιοποιηθούν και για τη δημιουργία εφαρμογών για κινητές συσκευές. Η δυνατότητα αυτή μου δημιούργησε την περιέργεια να κατανοήσω καλύτερα τον τρόπο με τον οποίο διαφορετικές τεχνολογίες και επιμέρους λειτουργίες μπορούν να συνεργαστούν για τη δημιουργία μιας ολοκληρωμένης εφαρμογής.

Η ανάπτυξη μιας εφαρμογής ανταλλαγής μηνυμάτων αποτέλεσε για τον λόγο αυτό μια ιδιαίτερα ενδιαφέρουσα επιλογή. Ήθελα να κατανοήσω στην πράξη τον τρόπο με τον οποίο συνδυάζονται η διεπαφή χρήστη, η διαχείριση της κατάστασης και η αποθήκευση δεδομένων, ώστε να υλοποιούνται λειτουργίες όπως η ανάκτηση και η εμφάνιση όλων των μηνυμάτων σε ένα παράθυρο συνομιλίας ή η ανάκτηση και διαχείριση των δεδομένων των χρηστών που συμμετέχουν στους διαφορετικούς διαλόγους.

Μέσα από την εκπόνηση της εργασίας απέκτησα ουσιαστικότερη εμπειρία στην ανάπτυξη εφαρμογών για κινητές συσκευές, εξοικειώθηκα με νέες τεχνολογίες και κατανόησα καλύτερα τον τρόπο με τον οποίο διαφορετικά τμήματα μιας εφαρμογής συνεργάζονται μεταξύ τους για την υλοποίηση ενός ολοκληρωμένου συστήματος.

Περίληψη

Σκοπός της παρούσας πτυχιακής εργασίας είναι η σχεδίαση και η ανάπτυξη μιας εφαρμογής ανταλλαγής μηνυμάτων για κινητές συσκευές, αξιοποιώντας το React Native και το Expo ως βασικές τεχνολογίες ανάπτυξης. Η εφαρμογή στοχεύει στην παροχή ενός σύγχρονου και εύχρηστου περιβάλλοντος επικοινωνίας, το οποίο επιτρέπει στους χρήστες να ανταλλάσσουν μηνύματα με άμεσο και αποτελεσματικό τρόπο.

Επιπλέον, η εργασία αποσκοπεί στη μελέτη και την εφαρμογή σύγχρονων τεχνικών ανάπτυξης εφαρμογών πολλαπλών πλατφόρμων (cross-platform), διερευνώντας τα πλεονεκτήματα που προσφέρει η χρήση μίας κοινής βάσης κώδικα για την ανάπτυξη εφαρμογών σε διαφορετικά λειτουργικά συστήματα κινητών συσκευών.

Τέλος, μέσω της υλοποίησης της εφαρμογής επιδιώκεται η πρακτική αξιοποίηση των γνώσεων που σχετίζονται με τον σχεδιασμό διεπαφών χρήστη, τη διαχείριση δεδομένων και την ανάπτυξη ολοκληρωμένων εφαρμογών κινητών συσκευών, καθώς και η αξιολόγηση της αποτελεσματικότητας των τεχνολογιών που χρησιμοποιήθηκαν.

Για την επίτευξη του σκοπού της παρούσας πτυχιακής εργασίας τέθηκαν οι ακόλουθοι στόχοι.

Η μελέτη των σύγχρονων τεχνολογιών ανάπτυξης εφαρμογών κινητών συσκευών και η διερεύνηση των δυνατοτήτων που προσφέρει το React Native σε συνδυασμό με το Expo.

Η σχεδίαση μιας εφαρμογής ανταλλαγής μηνυμάτων με σύγχρονη αρχιτεκτονική, η οποία να είναι επεκτάσιμη, εύκολη στη συντήρηση και αποδοτική ως προς τη διαχείριση των λειτουργιών της.

Η υλοποίηση μηχανισμού αυθεντικοποίησης χρηστών, ώστε να διασφαλίζεται η ασφαλής πρόσβαση στις λειτουργίες της εφαρμογής.

Η ανάπτυξη λειτουργιών δημιουργίας συνομιλιών και ανταλλαγής μηνυμάτων μεταξύ των χρηστών.

Η αξιοποίηση της πλατφόρμας Firebase για την αυθεντικοποίηση των χρηστών, την αποθήκευση και συγχρονισμό των δεδομένων της εφαρμογής, καθώς και τη διαχείριση των πληροφοριών που σχετίζονται με τις συνομιλίες και τα μηνύματα.

Η αξιολόγηση της λειτουργικότητας, της απόδοσης και της αξιοπιστίας της εφαρμογής μέσω δοκιμών, με στόχο την επαλήθευση της ορθής λειτουργίας της σε πραγματικές συνθήκες χρήσης.

Η επίτευξη των παραπάνω στόχων οδηγεί στην ανάπτυξη μιας ολοκληρωμένης εφαρμογής ανταλλαγής μηνυμάτων, η οποία αξιοποιεί σύγχρονες τεχνολογίες ανάπτυξης κινητών εφαρμογών, προσφέροντας μια αξιόπιστη και εύχρηστη λύση επικοινωνίας.

Κατά την ανάπτυξη της εφαρμογής δόθηκε ιδιαίτερη έμφαση στη σχεδίαση του περιβάλλοντος διεπαφής χρήστη (User Interface - UI), στη δημιουργία μιας ομαλής εμπειρίας χρήσης (User Experience - UX), καθώς και στην αξιοποίηση σύγχρονων τεχνολογιών ανάπτυξης κινητών εφαρμογών.

Επιπλέον, εξετάζονται οι βασικές αρχές σχεδιασμού μιας εφαρμογής ανταλλαγής μηνυμάτων, οι τεχνολογίες που χρησιμοποιήθηκαν για την υλοποίησή της και η αρχιτεκτονική που ακολουθήθηκε για την οργάνωση του λογισμικού.

Η εργασία φιλοδοξεί να παρουσιάσει τόσο το θεωρητικό υπόβαθρο που σχετίζεται με την ανάπτυξη εφαρμογών κινητών συσκευών όσο και την πρακτική διαδικασία υλοποίησης μιας ολοκληρωμένης εφαρμογής ανταλλαγής μηνυμάτων, αναδεικνύοντας τα πλεονεκτήματα που προσφέρουν οι σύγχρονες τεχνολογίες ανάπτυξης πολλαπλών platforms.

«Messaging App Development for Android/iOS Devices»

Ioannis Chatziantoniou

Abstract

The present diploma thesis concerns the design and implementation of a cross-platform mobile messaging application developed using React Native and Expo. The main objective of the study was to examine the integration of modern mobile application development technologies within a unified communication system capable of supporting user management, data synchronization, message exchange, and real-time interaction.

The implemented application incorporates a wide range of functionalities associated with contemporary messaging platforms. These include user registration and authentication, profile management, user search, the creation of private and group conversations, and the exchange of text and image messages. Additional functionality was developed to support message replies, the marking and retrieval of liked messages, group participant management, and push notifications for newly received content.

Particular emphasis was placed on the organization and management of application data. Firebase services were utilized for authentication, data persistence, real-time synchronization, and multimedia content storage, while Redux Toolkit was employed for centralized application state management. Furthermore, the application provides functionality for modifying group conversation information, adding and removing participants, leaving group conversations, and accessing user-related and conversation-related information.

The final outcome of the thesis is a fully functional mobile messaging application that integrates communication, data management, state management, multimedia support, and notification mechanisms within a single environment. The implementation demonstrates the effective combination of React Native, Expo, Firebase, and Redux Toolkit for the development of a structured and interactive real-time communication application for mobile devices.

Ευχαριστίες

Με την ολοκλήρωση της παρούσας πτυχιακής εργασίας, κλείνει για εμένα ένας ιδιαίτερα σημαντικός και μακροχρόνιος κύκλος. Η πορεία μέχρι αυτό το σημείο δεν ήταν σύντομη ούτε εύκολη, χρειάστηκαν αρκετά χρόνια, επιμονή, υπομονή και συνεχή προσπάθεια. Μετά από πολλές αλλαγές και στο ίδιο το τμήμα αλλά και στην ίδια την ζωή. Για τον λόγο αυτό, η ολοκλήρωση της εργασίας αυτής έχει για εμένα ιδιαίτερη προσωπική σημασία.

Ένα ξεχωριστό και ιδιαίτερα μεγάλο ευχαριστώ οφείλω στον Κύριο Κιοσκερίδη ο οποίος με βοήθησε, δεν αγνόησε το αίτημά μου για ανάληψη θέματος και μου έδωσε την ευκαιρία να ολοκληρώσω σε μία περίοδο που αμφέβαλλα αν θα τα καταφέρω.

Τέλος, θα ήθελα να ευχαριστήσω την οικογένεια, τους φίλους και τα κοντινά μου πρόσωπα που στάθηκαν δίπλα μου με τον δικό τους τρόπο και με στήριξαν όλα αυτά τα χρόνια.

Η ολοκλήρωση της πτυχιακής εργασίας δεν αποτελεί για εμένα μόνο το τέλος μιας ακαδημαϊκής υποχρέωσης, αλλά την ολοκλήρωση μιας μακράς προσωπικής προσπάθειας.

Περιεχόμενα

Πρόλογος.....	iv
Περίληψη.....	v
Abstract	vi
Ευχαριστίες	vii
Περιεχόμενα	viii
Κατάλογος Εικόνων	xii
Κεφάλαιο 1ο: Δημιουργία Περιβάλλοντος	1
1.1 Εισαγωγή.....	1
1.2 React Native	1
1.3 Σχέση της React Native με τη JavaScript.....	1
1.4 Συγκριτική ανάλυση των Swift και Kotlin και αιτιολόγηση της επιλογής τεχνολογικής προσέγγισης.....	2
1.5 Expo	2
1.6 Δημιουργία του Project	3
1.7 Δοκιμάζοντας την Εφαρμογή σε Πραγματική Συσκευή.....	4
1.8 Δοκιμάζοντας την Εφαρμογή στο Android Emulator.....	4
1.9 React Hooks	4
1.9.1 sideEffects	5
1.9.2 useCallback	5
1.10 Επίλογος.....	5
Κεφάλαιο 2ο: Ανάπτυξη της Εφαρμογής	7
2.1 Εισαγωγή.....	7
2.2 Επιλογή Γραμματοσειράς και Εικόνας κατά την Έναρξη.....	8
2.3 Εισαγωγή και Χρήση του Συστήματος Πλοήγησης.....	10
2.3.1 Εισαγωγή των bottom tabs	11
2.3.2 React Native Vector Icons	11
2.3.3 Διαχωρισμός του NavigationContainer σε Ξεχωριστό Αρχείο.....	12
2.4 Δημιουργία της Οθόνης Συνομιλίας.....	13
2.4.1 Πληκτρολόγιο Συσκευής που Καλύπτει το Πλαίσιο Κειμένου.....	15
2.5 Σύστημα Εγγραφής / Αυθεντικοποίησης Χρήστη	15
2.5.1 Εναλλαγή ανάμεσα σε οθόνη σύνδεσης ή εγγραφής.....	17
2.5.2 Χρήση λογότυπου του τμήματος στην οθόνη σύνδεσης.....	18
2.6 Έλεγχος Εγκυρότητας Φόρμας.....	18
2.6.1 Έλεγχος διεύθυνσης email και κωδικού πρόσβασης.....	20

2.6.2 useReducer Hook	21
2.6.3 Εμφάνιση μηνύματος λανθασμένης εισαγωγής στοιχείων.....	23
2.6.4 Στιγμιότυπα Οθόνης με Χρήση του Android Emulator	24
2.7 Εγγραφή Χρήστη.....	25
2.8 Επίλογος.....	26
Κεφάλαιο 3ο: Σύστημα Back-End	28
3.1 Εισαγωγή.....	28
3.2 Google Firebase.....	29
3.3 Στέλνοντας τα Δεδομένα Χρήστη στην Βάση Δεδομένων	31
3.4 Διαχείριση Κατάστασης της Εφαρμογής με το Εργαλείο Redux	32
3.4.1 Χρήση Redux	32
3.4.2 Εισαγωγή των reducers στο store	33
3.4.3 Πρόσβαση στα δεδομένα που βρίσκονται στην κατάσταση (state) της εφαρμογής.....	35
3.5 Λειτουργίες της Οθόνης Σύνδεσης.....	39
3.5.1 Αποσύνδεση Χρήστη Όταν Λήξει το Token.....	41
3.6 Επίλογος.....	41
Κεφάλαιο 4ο: Οθόνη Ρυθμίσεων	43
4.1 Εισαγωγή.....	43
4.2 Τίτλος οθόνης και λοιπές λειτουργίες	44
4.3 Ενημέρωση Φωτογραφίας Προφίλ.....	47
4.4 Επίλογος.....	53
Κεφάλαιο 5ο: Αναζήτηση Χρηστών, Δημιουργία Συνομιλίας	55
5.1 Εισαγωγή.....	55
5.2 Δημιουργία Επιλογής	55
5.3 Ανάκτηση δεδομένων από τους αποθηκευμένους χρήστες	60
5.3.1 Εμφάνιση μηνύματος σε καινούργιο διάλογο	61
5.4 Δημιουργία του διαλόγου όταν σταλεί μήνυμα.....	61
5.5 Ανάκτηση Χρηστών Όταν Ξεκινάει η Εφαρμογή	65
5.5.1 Εμφάνιση Λίστας Συνομιλιών	66
5.5.2 Ρύθμιση Αποθηκευμένων Χρηστών στην Σύνδεση	67
5.5.3 Εμφάνιση Ενεργών Συνομιλιών	67
5.6 Επίλογος.....	69
Κεφάλαιο 6ο: Αποστολή Μηνυμάτων.....	70
6.1 Εισαγωγή.....	70
6.2 Μηνύματα κειμένου	71
6.2.1 Ενημέρωση Μηνυμάτων στο State.....	71

6.2.2 Εμφάνιση Μηνυμάτων	72
6.3 Επιπλέον Λειτουργίες Μηνυμάτων	73
6.3.1 Άνοιγμα του Μενού	73
6.3.2 Αγαπημένα Μηνύματα.....	74
6.3.3 Ενημέρωση Αγαπημένων Μηνυμάτων στο State	75
6.4 Απάντηση Σε Μηνύματα	77
6.5 Αποστολή Μηνυμάτων Εικόνας	80
6.5.1 Λήψη Εικόνας από Κάμερα	82
6.6 Επίλογος	83
Κεφάλαιο 7ο: Ομαδικές Συνομιλίες.....	85
7.1 Εισαγωγή.....	85
7.2 Προσθήκη λειτουργιών	85
7.3 Προσθήκη χρηστών στην συνομιλία	86
7.3.1 Εμφάνιση επιλεγμένων χρηστών.....	87
7.4 Λειτουργία Επιλογής Create.....	87
7.5 Επίλογος	88
Κεφάλαιο 8ο: Ρυθμίσεις Συνομιλιών	91
8.1 Εισαγωγή.....	91
8.2 Ρυθμίσεις Προσωπικών Συνομιλιών	91
8.3 Ρυθμίσεις Ομαδικών Συνομιλιών	95
8.3.1 Ενημέρωση Ονόματος Ομαδικής Συνομιλίας	96
8.3.2 Εμφάνιση Συμμετεχόντων Συνομιλίας.....	96
8.3.3 Αφαίρεση Χρήστη από Συνομιλία.....	97
8.3.4 Αποχώρηση του Εαυτού μας Από Ομαδική	100
8.4 Εμφάνιση Όλων των Χρηστών σε Διαφορετική Οθόνη.....	100
8.5 Προσθήκη Νέων Χρηστών στην Ομαδική	103
8.5.1 Λειτουργία Επιλογής Προσθήκης	103
8.6 Προβολή Αγαπημένων Μηνυμάτων	106
8.7 Επίλογος	107
Κεφάλαιο 9ο: Αποστολή Ειδοποιήσεων	110
9.1 Εισαγωγή.....	110
9.2 Διαμόρφωση Εφαρμογής για Ειδοποιήσεις.....	110
9.3 Ειδοποιήσεις Κατά την Αποστολή Μηνύματος Κειμένου και Εικόνας	111
9.4 Επίλογος	112
Κεφάλαιο 10ο: Συμπεράσματα ή/και προτάσεις βελτίωσης.....	113

ΒΙΒΛΙΟΓΡΑΦΙΑ	114
Επίσημα Τεχνικά Έγγραφα & Ιστοσελίδες.....	114

Κατάλογος Εικόνων

Εικόνα 2.1 - Οθόνη συνομιλίας μετά τις μετατροπές.....	14
Εικόνα 2.2 - Οθόνη σύνδεσης χρήστη.....	24
Εικόνα 2.3 - Οθόνη εγγραφής χρήστη.....	24
Εικόνα 2.4 - Ο Χρήστης έχει εισάγει όλα τα στοιχεία του λανθασμένα	25
Εικόνα 3.1 - Εισαγωγή λανθασμένων στοιχείων.....	40
Εικόνα 4.1 - Storage στον φάκελο profilePics του Firebase	51
Εικόνα 4.2 - Αποθηκευμένες εικόνες του φακέλου profilePics του Storage.....	51
Εικόνα 4.3 - Δεδομένα χρήστη στον κόμβο users με το μοναδικό id.....	52
Εικόνα 4.4 - Οθόνη ρυθμίσεων του συνδεδεμένου χρήστη	53
Εικόνα 5.1 - Οθόνη αναζήτησης χρηστών	58
Εικόνα 5.2 - Δυναμική αναζήτηση με τα αποτελέσματα.....	59
Εικόνα 5.3 - Μήνυμα καλωσορίσματος σε νέα συνομιλία.....	63
Εικόνα 5.4 - Νέα συνομιλία στον κόμβο chats με τα δεδομένα και τους users που αποτελούνται από τον άλλον χρήστη και εμάς.	64
Εικόνα 5.5 - Κόμβος userChats με τα id του χρήστη και την λίστα με τους επιμέρους διαλόγους που συμμετέχει.	64
Εικόνα 5.6 - Id χρηστών, κάποιος μετέχει σε δύο διαλόγους ενώ οι υπόλοιποι χρήστες σε έναν.	64
Εικόνα 5.7 - Κεντρική οθόνη και η λίστα με τους ενεργούς διαλόγους.....	68
Εικόνα 6.1 - Κόμβος messages με τα δεδομένα του κάθε μηνύματος	71
Εικόνα 6.2 - Κόμβος userLikedMessages, το id του χρήστη, το id του διαλόγου, id του μηνύματος και τα δεδομένα του.	75
Εικόνα 6.3 - Επισήμανση μηνυμάτων ως αγαπημένα και ημερομηνία αποστολής του.	76
Εικόνα 6.4 - Παρατεταμένο πάτημα στο μήνυμα και εμφάνιση μενού επιλογών.....	78
Εικόνα 6.5 - Απάντηση σε υφιστάμενο μήνυμα και εμφάνισή του στο κάτω μέρος της οθόνης.	79
Εικόνα 6.6 - Φάκελος εικόνων διαλόγου chatImages στο storage του Firebase.	80
Εικόνα 6.7 - Αποθήκευση εικόνας στο πεδίο imageUrl στον κόμβο messages.	81
Εικόνα 6.8 - Αποστολή εικόνας στο παράθυρο διαλόγου.	81
Εικόνα 6.9 – Απάντηση με εικόνα σε υφιστάμενο μήνυμα.	82
Εικόνα 7.1 - Δημιουργία νέας ομαδικής συνομιλίας με επιλογή χρηστών και την κατάλληλη ένδειξη.	89
Εικόνα 7.2 - Οθόνη ομαδικής συνομιλίας με τα ονόματα ορισμένων συμμετεχόντων.....	90
Εικόνα 8.1 - Καρτέλα στοιχείων χρήστη μέσα από τις ρυθμίσεις συνομιλίας	94
Εικόνα 8.2 - Επιλογή διαγραφής συμμετέχοντα ομαδικής συνομιλίας.	99
Εικόνα 8.3 - Μηνύματα προσθήκης / διαγραφής κάποιου χρήστη.	99
Εικόνα 8.4 - Λίστα με όλους τους συμμετέχοντες της ομαδικής συνομιλίας.....	102
Εικόνα 8.5 – Αναζήτηση και προσθήκη νέων ατόμων σε ομαδικό διάλογο.....	105
Εικόνα 8.6 - Ενημέρωση συνομιλίας μετά την προσθήκη πολλαπλών νέων χρηστών.	105
Εικόνα 8.7 - Ολοκληρωμένη οθόνη ρυθμίσεων με τις επιλογές, "Add users", "View all" και "Liked messages".	108
Εικόνα 8.8 - Λίστα με όλα τα αγαπημένα μηνύματα του χρήστη.	109
Εικόνα 8.9 - Ολοκληρωμένη οθόνη ρυθμίσεων του εγγεγραμμένου χρήστη.	109

Κεφάλαιο 1ο: Δημιουργία Περιβάλλοντος

1.1 Εισαγωγή

Σε αυτό το κεφάλαιο θα εξηγηθούν οι τεχνολογίες που θα χρησιμοποιηθούν για την ανάπτυξη της εφαρμογής. Η React Native σαν κύρια γλώσσα προγραμματισμού και το expo που χειρίζεται όλο το περιβάλλον στο οποίο θα στηθεί η εφαρμογή καθώς και τα πλεονεκτήματα που προσφέρει. Τέλος, θα εξηγηθεί πως η εφαρμογή μπορεί να δοκιμαστεί σε πραγματική συσκευή και το λογισμικό που χρησιμοποιήθηκε.

1.2 React Native

Η React Native αποτελεί ένα ανοικτού κώδικα πλαίσιο ανάπτυξης εφαρμογών κινητών συσκευών, το οποίο αναπτύχθηκε από τη Meta και επιτρέπει τη δημιουργία εφαρμογών για Android και iOS με χρήση JavaScript και της βιβλιοθήκης React. Από την πρώτη του έκδοση το 2015, έχει εξελιχθεί σε μία από τις πιο διαδεδομένες τεχνολογίες για cross-platform ανάπτυξη mobile εφαρμογών.

Η βασική του αρχιτεκτονική βασίζεται στην ιδέα της ανάπτυξης ενός ενιαίου κώδικα (single codebase), ο οποίος μπορεί να μεταγλωττιστεί και να εκτελεστεί σε διαφορετικά λειτουργικά συστήματα. Σε αντίθεση με άλλες προσεγγίσεις, η React Native χρησιμοποιεί εγγενή (native) στοιχεία του κάθε λειτουργικού συστήματος, γεγονός που προσφέρει υψηλότερη απόδοση και εμπειρία χρήσης αντίστοιχη με αυτή των native εφαρμογών.

Η ανάπτυξη στη React Native βασίζεται στη φιλοσοφία των επαναχρησιμοποιήσιμων συστατικών (components). Κάθε στοιχείο της διεπαφής χρήστη δομείται ως ανεξάρτητη μονάδα, η οποία μπορεί να συνδυαστεί με άλλες για τη δημιουργία λειτουργιών. Η προσέγγιση αυτή ενισχύει τη συντηρησιμότητα του κώδικα, διευκολύνει την επαναχρησιμοποίηση και επιτρέπει την ευκολότερη επέκταση της εφαρμογής.

Επιπλέον, η React Native υιοθετεί μια ιδιαίτερη προσέγγιση για τη δημιουργία διεπαφών χρήστη, όπου η κατάσταση της εφαρμογής (state) καθορίζει την απεικόνιση της εκάστοτε οθόνης. Με τον τρόπο αυτό, κάθε αλλαγή στα δεδομένα οδηγεί αυτόματα στην ανανέωση της διεπαφής, χωρίς να απαιτείται χειροκίνητος χειρισμός του DOM, όπως συμβαίνει σε παραδοσιακές web τεχνολογίες.

Ένα από τα σημαντικότερα πλεονεκτήματα της React Native είναι η δυνατότητα κοινής χρήσης μεγάλου μέρους του κώδικα μεταξύ διαφορετικών πλατφορμών. Αυτό μειώνει σημαντικά τον χρόνο ανάπτυξης και το κόστος συντήρησης, ενώ παράλληλα επιτρέπει την ταυτόχρονη υποστήριξη πολλαπλών λειτουργικών συστημάτων από μία ενιαία ομάδα ανάπτυξης.

Τέλος, το οικοσύστημα της React Native είναι ιδιαίτερα πλούσιο, παρέχοντας πληθώρα βιβλιοθηκών και εργαλείων για λειτουργίες όπως η πλοήγηση μεταξύ οθονών, η επικοινωνία μεταξύ διαφόρων API's, η διαχείριση κατάστασης και η ενσωμάτωση υπηρεσιών όπως το Firebase.

Για τον λόγο αυτό, αποτελεί μια ιδιαίτερα κατάλληλη επιλογή για την ανάπτυξη σύγχρονων εφαρμογών κινητών συσκευών με υψηλές απαιτήσεις λειτουργικότητας και επεκτασιμότητας.

1.3 Σχέση της React Native με τη JavaScript

Η React Native βασίζεται στη γλώσσα προγραμματισμού JavaScript, η οποία αποτελεί το κύριο μέσο ανάπτυξης της λογικής των εφαρμογών που υλοποιούνται με τη συγκεκριμένη τεχνολογία. Όλη η ροή

εκτέλεσης της εφαρμογής, η διαχείριση της κατάστασης καθώς και η αλληλεπίδραση με τα δεδομένα υλοποιούνται μέσω JavaScript ή παραλλαγών της, όπως η TypeScript.

Η JavaScript λειτουργεί ως το βασικό επίπεδο προγραμματισμού, μέσω του οποίου ο προγραμματιστής ορίζει τη συμπεριφορά της εφαρμογής, ενώ η απόδοση των γραφικών στοιχείων και η διαχείριση της διεπαφής χρήστη πραγματοποιούνται μέσω του μηχανισμού του React Native, ο οποίος μεταφράζει τον JavaScript κώδικα σε εγγενή (native) στοιχεία του εκάστοτε λειτουργικού συστήματος.

Η εκτέλεση του JavaScript κώδικα πραγματοποιείται μέσω ειδικού runtime περιβάλλοντος, όπως το Hermes, το οποίο επιτρέπει την επικοινωνία με τα native modules της συσκευής μέσω ενός ενδιάμεσου επιπέδου διασύνδεσης.

Ο μηχανισμός αυτός καθιστά εφικτή την αλληλεπίδραση μεταξύ του JavaScript επιπέδου και των εγγενών λειτουργιών των πλατφορμών Android και iOS, χωρίς να απαιτείται ξεχωριστή υλοποίηση για κάθε λειτουργικό σύστημα.

Με τον τρόπο αυτό, η JavaScript αποτελεί τον κεντρικό άξονα ανάπτυξης στο React Native, καθώς καθορίζει τη λογική της εφαρμογής, ενώ το πλαίσιο αναλαμβάνει τη μετατροπή αυτής της λογικής σε εγγενή στοιχεία διεπαφής χρήστη, εξασφαλίζοντας ταυτόχρονα συμβατότητα μεταξύ διαφορετικών πλατφορμών και υψηλό βαθμό επαναχρησιμοποίησης κώδικα.

1.4 Συγκριτική ανάλυση των Swift και Kotlin και αιτιολόγηση της επιλογής τεχνολογικής προσέγγισης

Οι Swift και Kotlin αποτελούν τις βασικές γλώσσες προγραμματισμού για την ανάπτυξη (native) εφαρμογών στις δύο κυρίαρχες πλατφόρμες κινητών συσκευών. Συγκεκριμένα, η Swift χρησιμοποιείται στο οικοσύστημα της Apple για την ανάπτυξη εφαρμογών σε iOS, iPadOS και macOS, ενώ η Kotlin αποτελεί την επίσημη γλώσσα ανάπτυξης για το λειτουργικό σύστημα Android.

Και οι δύο γλώσσες παρέχουν υψηλό επίπεδο απόδοσης και άμεση πρόσβαση στις δυνατότητες του εκάστοτε λειτουργικού συστήματος, επιτρέποντας την ανάπτυξη αποδοτικών και βελτιστοποιημένων εφαρμογών. Ωστόσο, η χρήση τους προϋποθέτει την ανάπτυξη ξεχωριστού κώδικα για κάθε πλατφόρμα, γεγονός που αυξάνει την πολυπλοκότητα της ανάπτυξης, τον απαιτούμενο χρόνο υλοποίησης και το κόστος συντήρησης ενός έργου που στοχεύει σε πολλαπλά λειτουργικά συστήματα.

Αντιθέτως, οι σύγχρονες cross-platform τεχνολογίες επιτρέπουν την ανάπτυξη εφαρμογών με ενιαία βάση κώδικα, η οποία μπορεί να εκτελείται τόσο σε Android όσο και σε iOS. Η προσέγγιση αυτή μειώνει σημαντικά τον χρόνο ανάπτυξης και διευκολύνει τη συντήρηση και την επέκταση της εφαρμογής, χωρίς να απαιτείται ξεχωριστή υλοποίηση για κάθε πλατφόρμα.

Για τους παραπάνω λόγους υιοθετήθηκε η χρήση της React Native ως τεχνολογίας cross-platform ανάπτυξης. Η επιλογή αυτή επιτρέπει την ταχύτερη υλοποίηση της εφαρμογής ανταλλαγής μηνυμάτων.

1.5 Expo

Το **Expo** αποτελεί ένα ανοικτού κώδικα framework και πλατφόρμα ανάπτυξης εφαρμογών κινητών συσκευών, η οποία βασίζεται στην React Native. Στόχος του είναι η απλοποίηση της διαδικασίας ανάπτυξης εφαρμογών για λειτουργικά συστήματα Android και iOS, μέσω ενός ολοκληρωμένου συνόλου εργαλείων και υπηρεσιών που μειώνουν σημαντικά την ανάγκη χρήσης Swift ή Kotlin.

Παρέχει ένα πλήρες **Software Development Kit (Expo SDK)**, το οποίο περιλαμβάνει έτοιμες βιβλιοθήκες για την αξιοποίηση βασικών δυνατοτήτων των κινητών συσκευών. Ενδεικτικά,

υποστηρίζονται λειτουργίες όπως η χρήση κάμερας, η διαχείριση ειδοποιήσεων (push notifications), ο γεωεντοπισμός, η αποθήκευση δεδομένων και η επεξεργασία πολυμέσων.

Ένα από τα σημαντικότερα χαρακτηριστικά του Expo είναι ότι επιτρέπει την ανάπτυξη εφαρμογών χωρίς την απαίτηση εγκατάστασης και παραμετροποίησης εργαλείων όπως το Android Studio ή το Xcode για τις βασικές διαδικασίες ανάπτυξης και δοκιμής. Αυτό καθιστά το περιβάλλον ιδιαίτερα φιλικό προς τον προγραμματιστή και μειώνει σημαντικά την πολυπλοκότητα της διαδικασίας.

Επιπλέον, προσφέρει την εφαρμογή **Expo Go**, μέσω της οποίας είναι δυνατή η άμεση εκτέλεση της εφαρμογής σε πραγματικές συσκευές. Ο προγραμματιστής μπορεί να σαρώσει έναν κώδικα QR και να δοκιμάζει σε πραγματικό χρόνο τις αλλαγές που πραγματοποιεί, επιταχύνοντας έτσι τον κύκλο ανάπτυξης και ελέγχου της εφαρμογής.

Παράλληλα, το Expo παρέχει το **Expo CLI**, ένα εργαλείο γραμμής εντολών για τη δημιουργία, διαχείριση και εκτέλεση λειτουργιών για τη δημιουργία τελικών εκδόσεων της εφαρμογής για Android και iOS.

1.6 Δημιουργία του Project

Η χρήση του Node.js είναι απαραίτητη για την ανάπτυξη εφαρμογών με το Expo framework, καθώς αποτελεί το περιβάλλον εκτέλεσης στο οποίο βασίζεται το Expo CLI και το οικοσύστημα διαχείρισης εντολών npm. Μέσω του Node.js επιτρέπεται η δημιουργία, εκτέλεση και διαχείριση της εφαρμογής, καθώς και η λειτουργία του τοπικού development server που υποστηρίζει τη διαδικασία ανάπτυξης και δοκιμής της εφαρμογής.

Για την δημιουργία του περιβάλλοντος της εφαρμογής χρησιμοποιείται η εντολή **npm create-expo-app@latest** μέσα από το command line του λειτουργικού συστήματος. Ωστόσο, είναι σημαντικό να βρισκόμαστε μέσα στον φάκελο πριν εκτελεστεί η παραπάνω εντολή. Σε αυτόν τον φάκελο θα αποθηκευτεί όλη η δομή και τα αρχεία της εφαρμογής. Ακολουθώντας τα βήματα δίνουμε ένα μοναδικό όνομα στην εφαρμογή όπως και κάποιες επιλογές τύπου αρχείων. Στην συγκεκριμένη δουλειά χρησιμοποιήθηκε η blank επιλογή, σύμφωνα με την οποία το project έχει όσα λιγότερα προ-εγκατεστημένα αρχεία σε javascript. Οτιδήποτε χρειαστεί θα προστίθεται στην πορεία. Αφού γίνει η εγκατάσταση της εφαρμογής κάνουμε αλλαγή μέσα στον συγκεκριμένο φάκελο. Εκεί γίνεται η εκτέλεση της εντολής **npm expo start** που ξεκινάει τον server εκτέλεσης της εφαρμογής.

VsCode Text Editor

Θα χρησιμοποιηθεί το visual studio code σαν το κύριο πρόγραμμα συγγραφής. Οτιδήποτε άλλο είναι αποδεκτό και εξαρτάται από την προτίμηση του κάθε προγραμματιστή. Μέσω του vsCode ανοίγουμε τον φάκελο με όλο το project όπου βρίσκεται η δομή των αρχείων όπως έχει δημιουργηθεί αυτόματα.

Το αρχείο App.js είναι το κύριο μέρος όπου και εκτελείται ο κώδικας.

Το αρχείο package.json περιέχει πληροφορίες σχετικά με τα πακέτα που έχουμε κάνει εγκατάσταση και τις αντίστοιχες εκδόσεις. Μπορεί να διαφέρει σημαντικά από χρήστη σε χρήστη ανάλογα με τον χρόνο εγκατάστασης τους καθώς γίνονται αναβαθμίσεις σε νεότερα πακέτα.

Κατά την μεταφορά αρχείων δεν περιλαμβάνεται ο φάκελος με το όνομα node_modules διότι πρόκειται για φάκελο που περιέχει μεγάλο αριθμό αρχείων. Προτιμάται να γίνεται χωριστά ανά περίπτωση η εγκατάσταση του συγκεκριμένου φακέλου με την εντολή **npm install**. Η εντολή θα κάνει εγκατάσταση τα 'dependencies' για όλα τα πακέτα που έχουν χρησιμοποιηθεί σύμφωνα με ό,τι βρίσκεται στο package.json, μαζί με τα νούμερα εκδόσεων. Ο φάκελος node_modules δεν περιλαμβάνεται και στην σελίδα του project στο github.

1.7 Δοκιμάζοντας την Εφαρμογή σε Πραγματική Συσκευή

Η έναρξη του περιβάλλοντος ανάπτυξης με την εντολή **npm expo start** στο command line που παρέχει το VsCode δίνει κάποιες επιλογές. Μία από αυτές είναι ένα QR Code. Κάνοντας scan κάποιος μπορεί να τρέξει την εφαρμογή σε πραγματική συσκευή αφού πρώτα γίνει εγκατάστασή της εφαρμογής Expo Go σε αυτήν.

Ότι αλλαγή γίνεται στον κώδικα απεικονίζεται απευθείας στην οθόνη του κινητού.



Για να εκτελεστούν σωστά και να γίνει η αναγνώριση της κινητής συσκευής θα πρέπει να είναι και το laptop και το κινητό συνδεδεμένα στο ίδιο WiFi.

1.8 Δοκιμάζοντας την Εφαρμογή στο Android Emulator

Εγκατάσταση του Android Studio, χωρίς όμως να χρησιμοποιηθεί για την πραγματική δημιουργία κώδικα. Θα γίνει χρήση μόνο του Android Emulator. Δημιουργούμε μία εικονική συσκευή από την επιλογή 'Create Virtual Device'. Δίνεται μια λίστα με διάφορες συσκευές αλλά πρέπει να επιλέξουμε μία που έχει δίπλα το εικονίδιο του Google Play Store. Υποδεικνύει πως μέσω αυτού μπορεί κάποιος να κατεβάσει την εφαρμογή Expo Go εικονικά μέσω της κινητής συσκευής. Μετά από λίγα βήματα παραμετροποίησης που επιλέγουμε και την έκδοση Android η οθόνη του κινητού εμφανίζεται σαν μία πραγματική συσκευή.

1.9 React Hooks

Η λειτουργία τους είναι να επιτρέπουν στα React Components να έχουν πρόσβαση στο state της εφαρμογής και άλλα χαρακτηριστικά της React. State είναι δεδομένα που αναφέρονται στην συγκεκριμένη επανάληψη του κάθε component ή αλλιώς η συγκεκριμένη κατάσταση που βρίσκεται το κάθε component. Είναι σαν τα instances σε μία κλάση σε γλώσσες όπως η Java, όπου μπορούμε να δημιουργήσουμε ένα instance μια κλάσης με το όνομα User. Το συγκεκριμένο instance μπορεί να έχει ένα πεδίο name, age, birthday οτιδήποτε. Με απλά λόγια το state είναι data που ανταποκρίνεται στο συγκεκριμένο instance του συγκεκριμένου component στο οποίο δουλεύουμε.

useState: function της React το οποίο καλούμε και επιστρέφει ένα array. Το πρώτο στοιχείο αυτού είναι το ίδιο state object και το δεύτερο στοιχείο που επιστρέφει είναι το function που χρησιμοποιούμε για να ενημερώσουμε το state object. Αυτή η δομή ονομάζεται Array Destructuring, όπου παίρνουμε τις δύο τιμές που επιστρέφει το useState και τις θέτουμε στις δύο αυτές μεταβλητές για να μπορούμε να τις χρησιμοποιήσουμε στην συνέχεια.

Γενικά, δίνουμε ένα όνομα στην μεταβλητή state και μετά το function που ενημερώνει το αντίστοιχο state. Στο useState θέτουμε την αρχική τιμή του state.

Κάθε φορά που καλούμε το function που ενημερώνει το state γίνεται νέο render του component και μετά γίνεται η ενημέρωση του state object.

1.9.1 sideEffects

Λειτουργίες που δεν αναφέρονται στο κύριο σκοπό ενός component. Ένα παράδειγμα μπορεί να είναι το αίτημα σε βάση να επιστρέψει κάποια δεδομένα, τα στοιχεία χρήστη, μία φωτογραφία ή να θέσουμε έναν μετρητή που θα μετράει αντίστροφα μέχρι να γίνει κάτι. Γίνεται με την χρήση του useEffect της React, ένα function στο οποίο δηλώνουμε την λειτουργία που θέλουμε να εκτελεστεί. Στέλνοντας αίτημα σε μία βάση δεδομένων είναι κάτι που γίνεται μία φορά και ενδέχεται να μην χρειάζεται να γίνει ξανά το ίδιο αίτημα. Για να αποτραπεί κάτι τέτοιο κάθε useEffect δέχεται σαν παράμετρο την λίστα των εξαρτήσεων, dependency list, που καθορίζει κάθε πότε θα εκτελεστεί. Κενή λίστα σημαίνει εκτέλεση μόνο μία φορά. Λίστα με μία μεταβλητή που αλλάζει σημαίνει θα εκτελεστεί όταν η αναφερόμενη μεταβλητή αλλάζει τιμή.

1.9.2 useCallback

Χρησιμοποιείται για να περιλαμβάνει functions και ορίζει στην React πότε θα πρέπει να εκτελέσει το συγκεκριμένο function κάθε φορά που γίνεται render ενός component. Αποτρέπει τις αχρείαστες εκτελέσεις του function. Σαν πρώτη παράμετρο δέχεται το σώμα του κώδικα και σαν δεύτερη τα dependencies, μία λίστα που ορίζει πότε θα εκτελεστεί το συγκεκριμένο callback. Εάν αυτό που έχει δηλωθεί στη λίστα των dependencies δεν αλλάξει τότε δεν θα εκτελεστεί και ο κώδικας. Επίσης το useCallback κρατάει ένα αντίγραφο του function στο οποίο αναφέρεται, το κάνει caching, έτσι γνωρίζει εάν αυτό έχει αλλάξει σε κάθε render του component.

1.10 Επίλογος

Στο παρόν κεφάλαιο παρουσιάστηκαν οι βασικές τεχνολογίες και τα εργαλεία που χρησιμοποιήθηκαν για την ανάπτυξη της εφαρμογής ανταλλαγής μηνυμάτων. Αρχικά, αναλύθηκε η React Native ως πλαίσιο ανάπτυξης εφαρμογών κινητών συσκευών και παρουσιάστηκαν τα βασικά χαρακτηριστικά της, με έμφαση στη δυνατότητα ανάπτυξης εφαρμογών για διαφορετικές πλατφόρμες μέσω κοινής βάσης κώδικα και στη χρήση επαναχρησιμοποιήσιμων components.

Στη συνέχεια, εξετάστηκε η σχέση της React Native με τη JavaScript και ο τρόπος με τον οποίο η γλώσσα χρησιμοποιείται για την υλοποίηση της λογικής και της διαχείρισης της κατάστασης της εφαρμογής. Παράλληλα, παρουσιάστηκε η διαφορά μεταξύ της native ανάπτυξης με Swift και Kotlin και της cross-platform προσέγγισης που υιοθετεί η συγκεκριμένη εφαρμογή, αιτιολογώντας την επιλογή της React Native ως βασικής τεχνολογίας ανάπτυξης.

Ακολούθως, παρουσιάστηκε το Expo, το οποίο χρησιμοποιήθηκε ως βασικό περιβάλλον ανάπτυξης πάνω από τη React Native, καθώς και τα εργαλεία που παρέχει για τη δημιουργία, εκτέλεση και δοκιμή της εφαρμογής. Περιγράφηκε επίσης η διαδικασία δημιουργίας του project, η χρήση του Node.js και του Visual Studio Code, καθώς και η δομή βασικών αρχείων και εξαρτήσεων του έργου.

Τέλος, παρουσιάστηκαν οι τρόποι δοκιμής της εφαρμογής τόσο σε πραγματική κινητή συσκευή μέσω του Expo Go όσο και σε Android Emulator. Το κεφάλαιο ολοκληρώθηκε με την παρουσίαση των βασικών React Hooks που χρησιμοποιούνται στην ανάπτυξη, όπως τα useState, useEffect και useCallback, τα οποία συμβάλλουν στη διαχείριση της κατάστασης, στην εκτέλεση ενεργειών ως αποτέλεσμα αλλαγών και στη διαχείριση συναρτήσεων μέσα στα components.

Συνολικά, οι τεχνολογίες και οι μηχανισμοί που παρουσιάστηκαν αποτελούν τη βάση πάνω στην οποία υλοποιήθηκε η εφαρμογή. Η κατανόηση του τρόπου λειτουργίας τους είναι απαραίτητη για την παρουσίαση των επιμέρους λειτουργιών και της αρχιτεκτονικής της εφαρμογής που ακολουθεί στα επόμενα κεφάλαια.

Κεφάλαιο 2ο: Ανάπτυξη της Εφαρμογής

2.1 Εισαγωγή

Στο παρόν κεφάλαιο παρουσιάζεται η διαδικασία ανάπτυξης των βασικών λειτουργιών και της διεπαφής χρήστη της εφαρμογής ανταλλαγής μηνυμάτων. Μετά την παρουσίαση των βασικών τεχνολογιών που χρησιμοποιούνται για την υλοποίηση της εφαρμογής, ακολουθεί η πρακτική εφαρμογή των γνώσεων αυτών μέσω της σταδιακής δημιουργίας των επιμέρους οθονών, των επαναχρησιμοποιήσιμων στοιχείων διεπαφής και του συστήματος πλοήγησης.

Αρχικά παρουσιάζεται η διαδικασία προσαρμογής της εμφάνισης της εφαρμογής, μέσω της εισαγωγής εξωτερικής γραμματοσειράς και της δημιουργίας μιας αρχικής οθόνης φόρτωσης. Η χρήση της οθόνης SplashScreen επιτρέπει την ομαλή μετάβαση από την εκκίνηση της εφαρμογής στην κύρια διεπαφή της, ενώ παράλληλα πραγματοποιείται η φόρτωση απαραίτητων πόρων, όπως οι γραμματοσειρές που χρησιμοποιούνται στα διάφορα στοιχεία της εφαρμογής.

Στη συνέχεια αναλύεται η υλοποίηση του συστήματος πλοήγησης μεταξύ των διαφορετικών οθονών. Για τον σκοπό αυτό χρησιμοποιείται η βιβλιοθήκη React Navigation, μέσω της οποίας δημιουργείται ένας μηχανισμός πλοήγησης βασισμένος τόσο στη λογική της στοίβας οθονών όσο και στη χρήση καρτελών στο κάτω μέρος της εφαρμογής. Παράλληλα, παρουσιάζεται ο τρόπος οργάνωσης του συστήματος πλοήγησης σε ξεχωριστά αρχεία, με στόχο τη βελτίωση της δομής και της συντηρησιμότητας του πηγαίου κώδικα.

Ένα σημαντικό μέρος του κεφαλαίου αφορά τη δημιουργία της βασικής οθόνης συνομιλίας. Στην οθόνη αυτή υλοποιούνται τα βασικά στοιχεία που απαιτούνται για την ανταλλαγή μηνυμάτων, όπως το πεδίο εισαγωγής κειμένου, τα εικονίδια επιλογής πολυμέσων και κάμερας, καθώς και ο μηχανισμός αλλαγής της διεπαφής ανάλογα με το αν ο χρήστης έχει εισαγάγει κάποιο μήνυμα. Επιπλέον, εξετάζεται η προσαρμογή της οθόνης στις διαφορετικές διαστάσεις των κινητών συσκευών και η αντιμετώπιση προβλημάτων που προκύπτουν από την εμφάνιση του πληκτρολογίου.

Ακολουθώς παρουσιάζεται η ανάπτυξη του συστήματος εγγραφής και σύνδεσης των χρηστών. Για τον σκοπό αυτό δημιουργούνται ξεχωριστές φόρμες για τη διαδικασία δημιουργίας λογαριασμού και τη διαδικασία σύνδεσης, καθώς και επαναχρησιμοποιήσιμα στοιχεία διεπαφής που χρησιμοποιούνται σε περισσότερα από ένα σημεία της εφαρμογής. Η προσέγγιση αυτή συμβάλλει στη μείωση της επανάληψης του κώδικα και στη δημιουργία μιας περισσότερο οργανωμένης αρχιτεκτονικής.

Ιδιαίτερη έμφαση δίνεται επίσης στον έλεγχο εγκυρότητας των δεδομένων που εισάγει ο χρήστης στις φόρμες. Παρουσιάζεται η διαδικασία ελέγχου των στοιχείων, όπως το όνομα, η διεύθυνση ηλεκτρονικού ταχυδρομείου και ο κωδικός πρόσβασης, καθώς και η εμφάνιση κατάλληλων μηνυμάτων σε περίπτωση λανθασμένης εισαγωγής δεδομένων. Για τη διαχείριση της κατάστασης των φορμών χρησιμοποιείται το Hook `useReducer` της React, το οποίο επιτρέπει την κεντρική και οργανωμένη διαχείριση πολλαπλών πεδίων και της εγκυρότητας ολόκληρης της φόρμας.

Τέλος, παρουσιάζεται η διαδικασία προετοιμασίας των δεδομένων του χρήστη για την εγγραφή και την αυθεντικοποίησή του στην εφαρμογή. Οι πληροφορίες που εισάγονται στις φόρμες αποθηκεύονται στην κατάσταση των αντίστοιχων `components` και στη συνέχεια προετοιμάζονται ώστε να μπορούν να χρησιμοποιηθούν από τον μηχανισμό αυθεντικοποίησης και να αποσταλούν στην υπηρεσία που θα αναλάβει τη διαχείριση των χρηστών.

Συνολικά, το κεφάλαιο αυτό παρουσιάζει τη σταδιακή μετάβαση από τη βασική δομή της εφαρμογής στη δημιουργία ενός ολοκληρωμένου περιβάλλοντος χρήστη, το οποίο περιλαμβάνει μηχανισμούς

πλοήγησης, οθόνες συνομιλίας, φόρμες εγγραφής και σύνδεσης, έλεγχο εγκυρότητας δεδομένων και οργανωμένη διαχείριση της κατάστασης της εφαρμογής. Οι λειτουργίες αυτές αποτελούν τη βάση για την υλοποίηση των επόμενων χαρακτηριστικών και τη σύνδεση της εφαρμογής με υπηρεσίες αποθήκευσης και διαχείρισης δεδομένων.

2.2 Επιλογή Γραμματοσειράς και Εικόνας κατά την Έναρξη

Γίνεται χρήση γραμματοσειράς μέσω του google fonts. Έχει επιλεγθεί η γραμματοσειρά με το όνομα Fira Sans όπου κατεβαίνει ένας φάκελος με αρχεία που έχουν κατάληξη (.ttf). Δημιουργία φακέλου στην δομή του project με το όνομα fonts και εξαγωγή αυτών των αρχείων.

SplashScreen είναι ένας τρόπος να εμφανίσουμε μία εικόνα την στιγμή που φορτώνει η εφαρμογή και την στιγμή που έχει ολοκληρωθεί η διαδικασία γίνεται η αφαίρεσή της. Η εγκατάσταση γίνεται με την χρήση της εντολής

npx expo install expo-splash-screen

Η εισαγωγή της γραμματοσειράς παίρνει ένα μικρό διάστημα για να φορτωθεί. Στην διάρκεια αυτή θα εμφανίζεται μία εικόνα (SplashScreen). Όταν ολοκληρωθεί και είναι διαθέσιμα θα γίνει η αφαίρεσή της και θα μπορεί ο χρήστης να συνεχίσει στην εφαρμογή.

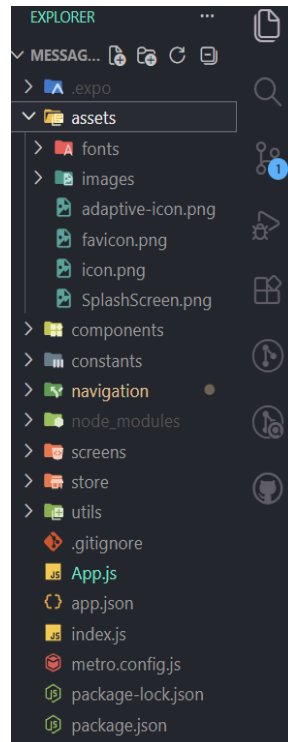
Η εικόνα είναι τυχαία και τοποθετείται μέσα στον φάκελο assets με το όνομα SplashScreen.png. Μετά, γίνεται μια ρύθμιση του configuration στο αρχείο app.json ώστε να αλλάξει το path που θα διαβάζει την εικόνα.

```
"splash": {
  "image": "./assets/SplashScreen.png",
  "resizeMode": "cover",
  "backgroundColor": "#ffffff"
```

Στις συσκευές Android ενδέχεται να υπάρχει ένα κενό ανάμεσα στα δύο άκρα της οθόνης. Για να διορθωθεί αυτό αλλάζουμε το resizeMode σε 'cover'. Όπως υπονοεί και το όνομα θα το αλλάξει ώστε να καλύπτει ολόκληρη την οθόνη.

Για την εισαγωγή της γραμματοσειράς έγινε εγκατάσταση το πακέτου expo-fonts με την εντολή (**npx expo install expo-font**). Με την χρήση import στο αρχείο App.js μπορούμε να χρησιμοποιήσουμε την γραμματοσειρά που έχουμε εισάγει. Σε useEffect θα καλέσουμε το function loadAsync που προσφέρεται μέσω του Font Object. Πρόκειται για ασύγχρονη διαδικασία της JavaScript, έτσι θα χρησιμοποιηθεί το async/await, καθώς πρέπει να γίνει διαχείριση ενός Promise. Σε ένα useEffect είναι προτιμότερο να μην χρησιμοποιείται το async keyword. Θα χρησιμοποιηθεί σε διαφορετικό function και θα το καλέσουμε στο τέλος. Στην ασύγχρονη διαδικασία με το όνομα prepare θα γίνει η φόρτωση των γραμματοσειρών, στην ουσία δίνοντας το path αυτών στον φάκελο που έχουν αποθηκευτεί.

Η δομή των φακέλων και του τελικού project έχει την εξής μορφή:



Η μεταβλητή state με το όνομα `appIsLoaded` κρατάει εάν η εφαρμογή έχει φορτώσει ή όχι. Η αρχική της τιμή ορίζεται ως `false` και η αλλαγή γίνεται από το state function `setAppIsLoaded`. Όταν ολοκληρωθεί η διαδικασία με τις γραμματοσειρές θέτουμε `setAppIsLoaded(true)`;

Ο κώδικας όλων των παραπάνω θα είναι ο ακόλουθος:

```
export default function App() {
  const [appIsLoaded, setAppIsLoaded] = useState(false);

  useEffect(() => {
    const prepare = async function () {
      try {
        await Font.loadAsync({
          "black": require('./assets/fonts/FiraSans-Black.ttf'),
          "blackItalic": require('./assets/fonts/FiraSans-BlackItalic.ttf'),
          "bold": require('./assets/fonts/FiraSans-Bold.ttf'),
          "boldItalic": require('./assets/fonts/FiraSans-BoldItalic.ttf'),
          "italic": require('./assets/fonts/FiraSans-Italic.ttf'),
          "light": require('./assets/fonts/FiraSans-Light.ttf'),
          "lightItalic": require('./assets/fonts/FiraSans-LightItalic.ttf'),
          "medium": require('./assets/fonts/FiraSans-Medium.ttf'),
          "mediumItalic": require('./assets/fonts/FiraSans-MediumItalic.ttf'),
          "regular": require('./assets/fonts/FiraSans-Regular.ttf'),
          "thin": require('./assets/fonts/FiraSans-Thin.ttf'),
          "thinItalic": require('./assets/fonts/FiraSans-ThinItalic.ttf')
        });
      } catch(err) {
        console.error(err);
      } finally {
        setAppIsLoaded(true);
      }
    };

    prepare();
  }, []); // Run only on the first render
```

2.3 Εισαγωγή και Χρήση του Συστήματος Πλοήγησης

Χρήση του React Navigation. Πρόκειται για ένα πακέτο που χειρίζεται την μεταφορά ανάμεσα στις οθόνες της εφαρμογής και τις τοποθετεί σε μορφή στοιβάς, δηλαδή την μία πάνω στην άλλη. Με άλλα λόγια κάνει push την μία οθόνη και όταν ο χρήστης επιστρέψει την προηγούμενη την αφαιρεί σε μορφή pop.

Κάτι που χρησιμοποιηθεί επίσης είναι τα bottom-tabs του Stack Navigator. Πρόκειται για τα εικονίδια στο κάτω μέρος μίας εφαρμογής που όταν πατάει ο χρήστης μεταφέρεται στην αντίστοιχη οθόνη αλλάζοντας View.

Για την εγκατάσταση του πακέτου ακολουθείται το επίσημο εγχειρίδιο που εγκαθιστά και τα κατάλληλα dependencies ώστε να υπάρχει πλήρη συμβατότητα.

npm install @react-navigation/native

Στην εγκατάσταση του κάθε πακέτου πρώτη λέξη χρησιμοποιείται συνήθως το npm και expo και ακολουθεί το όνομα του εκάστοτε πακέτου που θέλουμε να εισάγουμε. Το πρόθεμα expo δουλεύει σε περιβάλλοντα όπου το project έχει στηθεί κατά αυτό τον τρόπο (expo projects), διαφορετικά δεν μπορεί να χρησιμοποιηθεί. Φροντίζει ώστε να είναι συμβατές όλες οι εκδόσεις των πακέτων μεταξύ τους και να μην υπάρχει κάποια ασυμφωνία που μπορεί να οδηγήσει σε προβλήματα. Ωστόσο για ορισμένα πακέτα δεν είναι απαραίτητο να γίνει κάτι τέτοιο.

npm install @react-navigation/native

Το πακέτο safe-area-context λαμβάνει υπόψη το μέγεθος της οθόνης της συσκευής που τρέχουμε την εφαρμογή για να μην ξεφύγει το περιεχόμενο από τα επιτρεπόμενα όρια. Κάτι τέτοιο μπορεί να είναι το πάνω μέρος της οθόνης όπου κάθε κινητό έχει συνήθως διαφορετική μορφή η μπορεί η κάμερα να βρίσκεται μέσα στην οθόνη.

npm install @react-navigation/stack

Εγκαθιστά το Stack Navigation μέρος.

npm install @react-native-gesture-handler @react-native-masked-view/masked-view

Η χρήση του γίνεται με το στοιχείο NavigationContainer που θα δείξουμε μετέπειτα με παράδειγμα.

Δημιουργία του Stack Navigator με το createNativeStackNavigator το οποίο και ορίζουμε σαν ένα Stack Object. Το root component θα είναι το Stack.Navigator και μέσα σε αυτό ορίζουμε την κάθε μία οθόνη (screen) που θέλουμε να δημιουργηθεί με το Stack.Screen.

Στην δομή του project δημιουργείται ένας φάκελος (screens) που περιέχει όλα τα screens που θα προστίθενται. Η οθόνη ChatListScreen είναι αυτή που θα περιέχει όλες τις συνομιλίες που έχει πάρει μέρος ο χρήστης και θα εμφανίζεται αμέσως μόλις συνδεθεί στην εφαρμογή.

Κάθε screen δηλώνεται στο property component του κάθε Stack.Screen που δημιουργούμε.

```
<Stack.Screen
  name='ChatScreen'
  component={ChatScreen}
  options={{
    headerTitle: '',
    headerTitleStyle: { fontWeight: 'bold' },
    headerBackTitle: 'Go back',
    headerTitleAlign: 'center'
  }}
/>
```

Η οθόνη ChatSettingsScreen που δημιουργείται στον φάκελο screens θα περιέχει τα settings της κάθε συνομιλίας. Προστίθεται και αυτή η οθόνη στο Stack Navigator μέσα στο Stack.Screen όπου το property component θα δείχνει στο αρχείο ChatSettingsScreen.js

Η πλοήγηση σε διαφορετικό screen γίνεται με τα εργαλεία που προσφέρει το React Navigation πακέτο. Ο τρόπος που έχουμε πρόσβαση σε αυτά είναι μέσω του props property που περνάει αυτόματα σε κάθε component που δημιουργούμε. Κάθε φορά που δημιουργείται ένα καινούργιο screen μέσα στο

Stack.Screen το React Navigation περνάει στο component property τα props. Μέσω αυτού έχουμε πρόσβαση στο αρχείο που δουλεύουμε.

Το σύστημα τύπου Stack μόλις γίνει πλοήγηση στην επόμενη οθόνη βάζει το συγκεκριμένο View πάνω στην στοίβα και ταυτόχρονα δημιουργείται αυτόματα ένα button με το οποίο μπορεί να επιστρέψει ο χρήστης στην προηγούμενη οθόνη.

Υπάρχει η δυνατότητα και αλλαγή στα options της κάθε οθόνης όπου μπορεί κάποιος να αλλάξει τίτλο, εμφάνιση.

2.3.1 Εισαγωγή των bottom tabs

Από την γραμμή εντολών γίνεται η εγκατάσταση των bottom tabs με την εντολή

npm install @react-navigation/bottom-tabs

Η λειτουργία του είναι ίδια με αυτή του Stack Navigator, με την έννοια ότι δημιουργούμε ένα BottomTabNavigator object και έπειτα δηλώνουμε κάθε στοιχείο με την μορφή Tab.Navigator.

Το TabNavigator που θα εισαχθεί θα λειτουργεί ως ένα component στην κύρια οθόνη. Μέσα στο TabNavigator είναι που θα προστεθεί το κάθε screen. Η πρώτη οθόνη που γίνεται αναφορά θα είναι το ChatListScreen και η δεύτερη το SettingsScreen. Το TabNavigator είναι ένα ξεχωριστό component που εμπεριέχει την λειτουργία του κάτω μέρους της οθόνης. Με την σειρά του θα προστεθεί στο Stack Navigator σαν component του Home Screen.

Αυτό που έγινε είναι να δηλωθεί το Stack Navigator σαν το κύριο επίπεδο Navigator της εφαρμογής. Μέσα του υπάρχει το Tab Navigator. Σε αυτό αν δηλωθεί ότι δεν θέλουμε να φαίνεται ο τίτλος τότε θα εφαρμοστεί σε όλες τις οθόνες που βρίσκονται μέσα του. Διαφορετικά πρέπει να πηγαίνουμε και να το κάνουμε σε κάθε screen χωριστά. Θέτουμε το property headerTitle να είναι ένα κενό string. Βεβαίως, θα υπάρχουν κάποιες περιπτώσεις που θα εμφανίζεται ο τίτλος αλλά θα γίνει στην κάθε οθόνη που χρειάζεται ξεχωριστά.

2.3.2 React Native Vector Icons

Είναι ένας εύκολος τρόπος για να χρησιμοποιήσουμε εικόνες από διάφορους προμηθευτές σε ένα μόνο πακέτο. Από την σελίδα των icons επιλέγεται οποιοδήποτε εικονίδιο επιθυμεί κάποιος καθώς και το όνομα προμηθευτή του αντίστοιχου πακέτου. Για την αλλαγή εικονιδίου στο tab που αναφέρεται στην λίστα με τις συνομιλίες προστίθεται το property tabBarIcon που δέχεται ένα component και μέσα του το όνομα του πακέτου. Κάθε tabBarIcon δέχεται αυτόματα ένα props object που περιέχει κάποια στοιχεία με το χρώμα, το μέγεθος και αν είναι επιλεγμένο.

Παράδειγμα δημιουργίας των εικονιδίων στο bottom-tabs

```
options={{
  tabBarLabel: 'Chats',
  tabBarLabelPosition: 'beside-icon',
  tabBarIcon: ({ color, size }) => ( // const { color, size } = props
    <Ionicons name="chatbubbles-outline" size={size} color={color} />
  )
}}
```

Ενδεικτικό κομμάτι κώδικα με ολόκληρη την δημιουργία των tabs.

```
const TabNavigator = function () {
  return (
    <Tab.Navigator screenOptions={{
      headerTitle: '',
      headerShadowVisible: false
    }}>
      <Tab.Screen
        name='ChatList'
        component={ChatListScreen}
        options={{
          tabBarLabel: 'Chats',
          tabBarLabelPosition: 'beside-icon',
          tabBarIcon: ({ color, size }) => ( // const { color, size } = props;
            <Ionicons name="chatbubbles-outline" size={size} color={color} />
          )
        }}
      />
      <Tab.Screen
        name='Settings'
        component={SettingsScreen}
        options={{
          tabBarLabel: 'Settings',
          tabBarLabelPosition: 'beside-icon',
          tabBarIcon: ({ color, size }) => ( // props is passed, then we destructure the object
            <SimpleLineIcons name="settings" size={size} color={color} />
          )
        }}
      />
    </Tab.Navigator>
  );
};
```

2.3.3 Διαχωρισμός του NavigationContainer σε Ξεχωριστό Αρχείο

Για την καλύτερη διαχείριση των αρχείων και της πλοήγησης θα γίνει διαχωρισμός του navigation σε ένα ξεχωριστό αρχείο με το όνομα AppNavigator αφού πρώτα δημιουργηθεί ένας νέος φάκελος με το όνομα navigation. Εκεί μέσα βρίσκεται όλο το navigation σύστημα της εφαρμογής. Το AppNavigator είναι ένα ξεχωριστό component με την σειρά του που επιστρέφει το JSX code. Με αυτό τον τρόπο έχουμε εξάγει όλη την λειτουργικότητα σε ξεχωριστό αρχείο και κρατάμε καθαρό το βασικό αρχείο (App.js) στο οποίο θα πρέπει να υπάρχει το component AppNavigator, στο σημείο που βρισκόταν πριν ο κώδικας που αντιγράφηκε.

Επιπλέον θα γίνει άλλη μία μετατροπή με την μεταφορά του Stack Navigator σε ξεχωριστό αρχείο με ονομασία MainNavigator.js που θα βρίσκεται στον φάκελο navigation. Έτσι στο AppNavigator εκεί που βρισκόταν ο κώδικας που μετακινήθηκε θα μπει στην θέση του το MainNavigator στοιχείο.

Μετά από αυτό ο MainNavigator περιέχει μόνο το Stack Navigator που δίνει όλο το view με τα bottom tabs. Ονομάστηκε έτσι γιατί περιλαμβάνει όλες τις σελίδες που θα βλέπει ο χρήστης μόλις συνδεθεί, η κύρια εμπειρία της εφαρμογής θα περιλαμβάνεται σε αυτές τις σελίδες. Εάν ο χρήστης δεν έχει συνδεθεί θα γίνεται παραπομπή σε άλλη οθόνη. Κάτι τέτοιο θα προστεθεί μέσα στο AppNavigator που θα κρίνει σε πια οθόνη θα ανακατευθύνει τον χρήστη με τις κατάλληλες ενέργειες.

Διαχωρίζοντας κάθε ξεχωριστό navigator σε δικό του αρχείο το κάνει πολύ πιο εύκολο να δουλέψει κάποιος.

2.4 Δημιουργία της Οθόνης Συνομιλίας

Πρόκειται για μία οθόνη (screen) που προστίθεται στον φάκελο screens. Θα πρέπει να εισαχθεί στον navigator σαν ένα νέο Stack.Screen με το όνομα ChatScreen.

```
<Stack.Screen
  name='ChatScreen'
  component={ChatScreen}
  options={{
    headerTitle: '',
    headerTitleStyle: { fontWeight: 'bold' },
    headerBackTitle: 'Go back',
    headerTitleAlign: 'center'
  }}
/>
```

Στην οθόνη αυτή θα υπάρχει μία εικόνα που θα λειτουργεί σαν φόντο της συνομιλίας. Η εισαγωγή της γίνεται με το ImageBackground της React Native. Για να λειτουργήσει και εμφανιστεί θα πρέπει να οριστεί πως θα πιάνει όλο το εύρος της οθόνης. Στο κάτω μέρος της οθόνης θα προστεθούν δύο εικονίδια, ένα με το εικονίδιο συν ' + ' από όπου ο χρήστης όταν το πατάει θα ανοίγει το μενού για να επιλέξει μία φωτογραφία από την συλλογή της συσκευής του. Το άλλο εικονίδιο θα είναι αυτό της κάμερας που θα ανοίγει την κάμερα της συσκευής, θα μπορεί να αποτυπώσει μία φωτογραφία και μετέπειτα να την στείλει στην συνομιλία.

Ενδιάμεσα από τα δύο εικονίδια θα υπάρχει ένα πλαίσιο στο οποίο θα γίνεται η εισαγωγή κειμένου για αποστολή στην συνομιλία. Θα είναι σε μορφή TextInput.

Σε πολλές περιπτώσεις όταν προστεθούν νέα στοιχεία σε μία οθόνη, στα δύο άκρα της παρατηρείται το φαινόμενο να γίνεται απόκρυψη μέρους τους. Οφείλεται στο πλαίσιο της οθόνης και τα περιθώρια που δίνονται αυτόματα από την κάθε συσκευή. Μία ασφαλής λύση είναι η εισαγωγή του SafeAreaView, ενός πακέτου που έρχεται με το όνομα React Native Safe Area Context.

```
<SafeAreaView
  edges={['right', 'left', 'bottom']}
  style={styles.container}
/>
```

Ο τρόπος για εισαγωγή εικονιδίων στην θέση ενός κουμπιού δεν γίνεται με την χρήση του Native Element Button της React Native. Δεν επιτρέπεται η εισαγωγή τους μέσα σε αυτό. Υπάρχει εναλλακτική με το στοιχείο TouchableOpacity, δηλαδή ένα στοιχείο που μπορεί να το πατήσει ο χρήστης. Όταν θέλουμε να φτιάξουμε ένα στοιχείο που είναι touchable χρησιμοποιούμε αυτή την τεχνική.

Ένα στοιχείο που δεν είναι touchable μπορεί να το μετατρέψει σε touchable καθώς περιέχει ένα property που χειρίζεται ενέργειες όταν γίνει ένα κλικ ή ακουμπήσει ο χρήστης το σημείο εκείνο στην οθόνη. Η λειτουργία που υποστηρίζει είναι το onPress και μπορεί να δεχτεί ένα function που εκτελεί μία ενέργεια. Μέσα σε αυτό προστίθεται ένα εικονίδιο από την βιβλιοθήκη vector icons που αναπαριστά το σύμβολο 'συν' καθώς θα ανοίγει την συλλογή της κινητής συσκευής.

Η ίδια διαδικασία γίνεται για το δεύτερο εικονίδιο, αυτό της κάμερας, επιλέγοντας από το πακέτο το εικονίδιο.

Στην δομή του project έχει δημιουργηθεί νέος φάκελος με το όνομα colors. Εκεί, υπάρχουν όλα τα χρώματα σε μορφή hex όπου θα προστίθενται και άλλα. Παρέχει την δυνατότητα να υπάρχουν όλα σε ένα μέρος, έτσι αν θέλουμε να κάνουμε μία αλλαγή σε έναν κωδικό τότε θα αντικατοπτρίζεται σε όλα τα σημεία που έχει χρησιμοποιηθεί μέσα στην εφαρμογή.

Η δημιουργία των εικονιδίων ενδεικτικά:

```

<TouchableOpacity
  style={styles.mediaButton}
  onPress={pickImage}
>
  <Entypo name="plus" size={24} color={colors.blue} />
</TouchableOpacity>

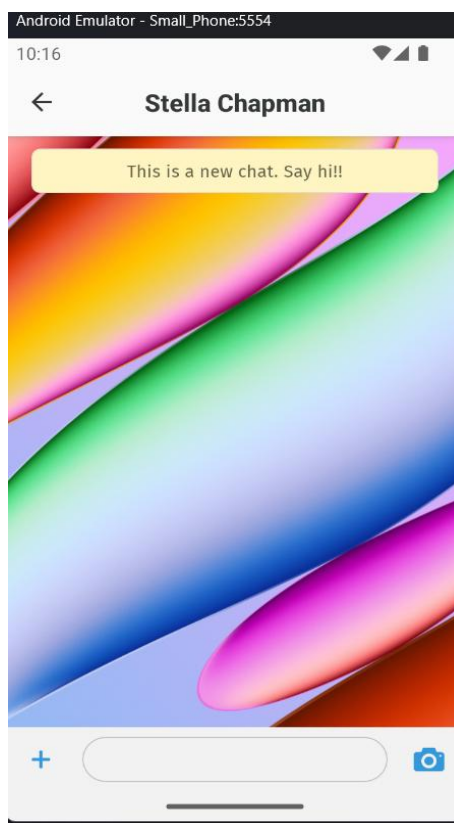
```

Εφαρμογή styling στο πλαίσιο κειμένου TextInput για την επιθυμητή μορφοποίηση:

```

textbox: {
  flex: 1,
  borderWidth: 1,
  borderRadius: 50,
  borderColor: colors.lightGrey,
  marginHorizontal: 15,
  paddingHorizontal: 12,
},

```



Εικόνα 2.1 - Οθόνη συνομιλίας μετά τις μετατροπές

Σε πολλές εφαρμογές αποστολής μηνυμάτων όταν ο χρήστης πληκτρολογεί στο πλαίσιο το κείμενο που θέλει να στείλει τότε το δεξί εικονίδιο αλλάζει απευθείας σε κάτι που υποδηλώνει πως μπορεί να το αποστείλει. Για να γίνει αυτό χρησιμοποιείται ένα state variable 'messageText' μαζί με το

setMessageText που ορίζει πότε θα αλλάξει η τιμή του πρώτου. Εάν λοιπόν υπάρχει κείμενο, ο χρήστης έχει πληκτρολογήσει κάτι, μέσα στο TextInput τότε θα εμφανίζεται διαφορετικό εικονίδιο. Εάν είναι κενό θα παραμένει αυτό της κάμερας.

Την αποστολή του μηνύματος την χειρίζεται το onPress από το στοιχείο με το εικονίδιο 'send' που δέχεται ένα function με όνομα sendMessage. Η αποστολή μπορεί επίσης να γίνει με το κουμπί submit από το πληκτρολόγιο της συσκευής το οποίο χειριζόμαστε από το property onSubmitEditing που επίσης δέχεται το sendMessage. Στην συνέχεια θα προστεθεί και η υπόλοιπη λειτουργικότητα.

2.4.1 Πληκτρολόγιο Συσκευής που Καλύπτει το Πλαίσιο Κειμένου

Σε πραγματική συσκευή όταν εμφανίζεται το πληκτρολόγιο για την εισαγωγή μηνύματος αυτό καλύπτει το πλαίσιο και ο χρήστης δεν μπορεί να δει τι πληκτρολογεί. Η React Native προσφέρει ένα στοιχείο με το όνομα KeyboardAvoidingView που λύνει αυτό το πρόβλημα. Θα πρέπει να τεθεί ως parent element όλων των υπολοίπων στοιχείων της οθόνης καθώς και δύο ακόμη property που ορίζουν την συμπεριφορά και το αντιστάθμισμα προς τα πάνω.

```
<KeyboardAvoidingView
  style={{ flex: 1 }}
  behavior={ Platform.OS === 'ios' ? 'padding' : 'height' }
  keyboardVerticalOffset={100}
>
```

2.5 Σύστημα Εγγραφής / Αυθεντικοποίησης Χρήστη

Η λειτουργία εγγραφής και ταυτοποίησης χρήστη θα γίνεται βάσει μίας μεταβλητής που θα ελέγχει εάν αυτός έχει επαληθεύσει τα στοιχεία του ή όχι. Εάν τα στοιχεία είναι ορθά θα μεταφέρεται στην κύρια οθόνη της εφαρμογής (MainNavigator), εάν δεν είναι θα εμφανίζεται νέο screen που θα δίνει την δυνατότητα εγγραφής. Η διαδικασία ελέγχου θα γίνεται στο αρχείο AppNavigator.

Η οθόνη που θα εμφανίζει τις επιλογές δημιουργείται στον φάκελο screens με ονομασία AuthScreen. Η χρήση του SafeAreaView βοηθάει ώστε το περιεχόμενο να μην καλύπτεται από την εγκόπη της κάμερας της συσκευής, μένει μέσα στα επιτρεπτά όρια εμφάνισης.

Κατά την διάρκεια της ανάπτυξης αξιοποιείται ένα custom component που τακτοποιεί την απόσταση (spacing) του περιεχομένου, δηλαδή τα κενά από τα άκρα της εκάστοτε οθόνης.

Ο φάκελος components έχει όλα τα αρχεία (components) που θα εμφανίζονται και στην συνέχεια της ανάπτυξης. Είναι επαναχρησιμοποιήσιμα κομμάτια κώδικα και αξιοποιούνται όπου κρίνεται απαραίτητο.

Το component PageContainer προσφέρει στοιχεία styling καθώς και χρώμα στο background. Το props και children είναι properties που περνάνε αυτόματα μέσω των React component. Children είναι οτιδήποτε βρίσκεται ανάμεσα στα tags του PageContainer, οτιδήποτε βρίσκεται ενδιάμεσα στα tags είναι προσβάσιμο μέσω του children. Αυτό σημαίνει πως μπορούμε να χρησιμοποιήσουμε το PageContainer (μαζί με το styling) και να περάσουμε ότι περιεχόμενο επιθυμούμε σε κάθε νέο screen που θα φτιάξουμε.

```
const PageContainer = props => {
  return <View style={{ ...styles.container, ...props.style }}>
    { props.children }
  </View>
};
```

Το στοιχείο `TextInput` της `React Native` για εισαγωγή κειμένου δεν προσφέρει πολλές επιλογές για μορφοποίηση, με την έννοια να υπάρχει ένα εικονίδιο και δίπλα να φαίνεται το κείμενο που εισάγεται. Δημιουργία νέου component με όνομα `Input`. Εισαγωγή ετικέτας `'label'` και `'TextInput'` καθώς και τα αντίστοιχα στυλ για το καθένα. Το εικονίδιο εισάγεται μέσω των `vector icons`. Κείμενο και εικονίδιο μπορούν να είναι στην ίδια γραμμή γιατί φτιάξαμε ένα custom component.

Για να μπορεί να γίνει εισαγωγή οποιουδήποτε πακέτου εικονιδίων και του αντίστοιχου προμηθευτή θα πρέπει στο `Input` component να δηλωθεί ένα γενικευμένο property με όνομα `icon` και `iconPack`. Έτσι, μπορούμε στο αρχείο που θέλουμε να χρησιμοποιήσουμε το συγκεκριμένο component να περάσουμε οποιοδήποτε συνδυασμό επιθυμούμε. Το ίδιο μπορεί να γίνει και με το μέγεθος του εικονιδίου.

```
{ // As long as the props.icon property exists then render this
  props.icon && <props.iconPack
    name={props.icon}
    size={props.iconSize || 20}
    style={styles.icon}
  />
}
```

Θα προστεθεί το χαρακτηριστικό που θα εμφανίζει μήνυμα λάθους αν για παράδειγμα έχει εισαχθεί λανθασμένο email ή όνομα. Το συγκεκριμένο θα προστεθεί μέσα σε ένα νέο `View` και θα αποτυπώνεται κάτω από το `Input` μόνο εάν έχει περαστεί το property `errorText` στο μέρος όπου θα αξιοποιηθεί αυτό το `Input` component.

```
{
  props.errorText &&
  <View style={styles.errorContainer}>
    <Text style={styles.errorText}>{props.errorText[0]}</Text>
  </View>
}
```

Στην φόρμα εισαγωγής των στοιχείων χρήστη θα προστεθούν και τα υπόλοιπα πεδία με την ίδια ακριβώς λογική. Το `Input` component θα αναπαριστά ένα πλαίσιο εισαγωγής για το επίθετο, το email και τον κωδικό προτίμησης του χρήστη.

Στις περισσότερες φόρμες εγγραφής τέτοιου τύπου όταν συμπληρωθούν όλα τα πεδία υπάρχει ένα κουμπί επιβεβαίωσης με το οποίο υποβάλλεται αυτή η φόρμα. Σε ένα ξεχωριστό component για να μπορεί να γίνει διαμόρφωση με πολλούς περισσότερους τρόπους από αυτούς που προσφέρει το native `Button` στοιχείο. Στον φάκελο `components` δημιουργία αρχείου `SubmitButton` και θα είναι επαναχρησιμοποιήσιμο σε διάφορα μέρη της εφαρμογής.

Η εμφάνισή του θα καθορίζεται εάν είναι απενεργοποιημένο από το property (`disabled`).

```
const SubmitButton = function (props) {
  const enabledBgColor = props.color || colors.primary;
  const disabledBgColor = colors.lightGrey;
  const bgColor = props.disabled ? disabledBgColor : enabledBgColor;
```

Τίτλος, χρώμα κειμένου κατά τον ίδιο τρόπο.

```
return (
  <TouchableOpacity
    onPress={props.disabled ? () => {} : props.onPress}
    style={{
      ...styles.button,
      ...props.style,
      ... { backgroundColor: bgColor }
    }}
  >
    <Text style={{ color: props.disabled ? colors.grey : 'white' }}>
      { props.title }
    </Text>
  </TouchableOpacity>
);
```

2.5.1 Εναλλαγή ανάμεσα σε οθόνη σύνδεσης ή εγγραφής

Η λειτουργικότητα των πλαισίων κειμένου των Inputs καθώς και το κουμπί υποβολής θα μεταφερθούν σε νέο ξεχωριστό component με όνομα SignUpForm.

Ένα react component δεν μπορεί να επιστρέψει πολλά στοιχεία μαζί, πρέπει να είναι μέσα σε ένα (parent) στοιχείο. Για παράδειγμα ένα View που μέσα του όμως μπορούν να υπάρχει αρκετά στοιχεία. Αυτό που επιστρέφεται από ένα τέτοιο function θα πρέπει να είναι ένα και μοναδικό. Τα React fragments είναι τρόπος για να επιστρέφονται πολλά στοιχεία από ένα function.

Η διαδικασία επαναλαμβάνεται για την φόρμα σύνδεσης που χρειάζονται δύο πεδία, διεύθυνση email και κωδικός πρόσβασης. Ένα νέο component SignInForm που θα αναφέρεται σε αυτή την οθόνη.

```
{
  isSignUp ?
    <SignUpForm /> :
    <SignInForm />
}
```

Χρειαζόμαστε έναν τρόπο ώστε να εναλλάσσονται οι δύο αυτές οθόνες. Στο αρχείο AuthScreen η state μεταβλητή isSignUp καθορίζει πια θα εμφανίζεται.

Κουμπί τύπου `TouchableOpacity` χειρίζεται την εναλλαγή ανάμεσα στην οθόνη σύνδεσης και εγγραφής. Η λειτουργία του όταν ο χρήστης πατήσει σε αυτό είναι να αλλάζει την μεταβλητή `state` μέσω της μεθόδου `setIsSignUp` και να ενημερώνει την παραπάνω κατάσταση. Από την στιγμή που αλλάζει η τιμή της μεταβλητής βάσει της συνθήκης που αναφέρθηκε πιο πάνω θα απεικονιστεί και η κατάλληλη οθόνη.

```
<TouchableOpacity
  onPress={() => setIsSignUp(prevState => !prevState)}
  style={styles.linkContainer}
>
  <Text style={styles.link}>
    { `Switch to ${isSignUp ? 'Sign In' : 'Sign Up'}` }
  </Text>
</TouchableOpacity>
```

2.5.2 Χρήση λογότυπου του τμήματος στην οθόνη σύνδεσης

Η εικόνα με το λογότυπο του τμήματος θα γίνει πάνω από τα πεδία συμπλήρωσης στοιχείων του χρήστη. Θα γίνει εισαγωγή της από το αρχείο και έπειτα απεικόνιση σε στοιχείο `Image` της `React Native`.

```
<View style={styles.imageContainer}>
  <Image
    style={styles.image}
    source={logo}
  />
</View>
```

Εάν ο χρήστης βρίσκεται σε μικρότερη συσκευή τότε δεν μπορεί να δει όλο το περιεχόμενο ούτε μπορεί να κάνει ‘scrolling’ ώστε να εμφανιστεί ολόκληρη η οθόνη. Επίσης, τα πεδία κρύβονται πίσω από το πληκτρολόγιο όταν αυτό εμφανιστεί στην συσκευή.

Λύση είναι το στοιχείο `ScrollView` της `React Native` που ουσιαστικά είναι ένα `View` το οποίο επιτρέπει το ‘scrolling’ όπως υποδηλώνει το όνομά του. Μπαίνει γύρο από τα υπόλοιπα στοιχεία, κατά κάποιο τρόπο τα συμπεριλαμβάνει. Όπως και πριν το πληκτρολόγιο καλύπτει κάποια πεδία και ο χρήστης δεν μπορεί να δει τι πληκτρολογεί. Η λύση βρίσκεται στο `KeyboardAvoidingView` και την μετατόπιση `keyboardVerticalOffset`.

2.6 Έλεγχος Εγκυρότητας Φόρμας

Ο έλεγχος εγκυρότητας θα γίνει με την χρήση πακέτου `validate JavaScript`. Πρόκειται για ένα πακέτο που προσφέρει έλεγχο για διαφορετικούς τύπους εισαγωγής όπως αριθμών, αριθμούς πιστωτικών καρτών, διευθύνσεις ηλεκτρονικού ταχυδρομείου, ημερομηνίες, εάν είναι πεδίο περιλαμβάνει μόνο αριθμούς ή γράμματα.

Η εγκατάσταση γίνεται με την εντολή **`npm install validate.js`**

Η διαδικασία επικύρωσης θα γίνεται όταν ο χρήστης κάνει αλλαγή μέσα στο πεδίο και η μέθοδος που θα το χειρίζεται αυτό είναι η `inputChangedHandler` στο αρχείο `SignUpForm`. Κάθε `Input` θα δέχεται ένα property `onInputChanged` που θα πυροδοτεί την μέθοδο `inputChangedHandler`. Για να εκτελεστεί θα πρέπει να γίνει χρήση του στο `Input component` μέσα στο οποίο το ‘περνάμε’ από τα props.

Το στοιχείο native `TextInput` έχει δική του λειτουργία με όνομα `onChangeText` και εκτελείται κάθε φορά που γίνεται αλλαγή σε αυτό. Περνάμε την μέθοδο με όνομα `onChangeText` που εξ'ορισμού δίνει αυτό που έχει πληκτρολογήσει ο χρήστης (`text`).

Στο `Input` component επίσης δίνεται ακόμη ένα πεδίο με όνομα `id` που αναφέρεται στο όνομα του δηλαδή σε αυτό μπορούμε να έχουμε πρόσβαση μέσω του `props.id`

```
const onChangeText = text => {
  setValue(text); // updating the state variable
  props.onInputChanged(props.id, text);
};
```

Αυτή η μεθοδολογία δίνει την δυνατότητα να ξέρουμε πιο πεδίο αλλάζει καθώς και την τιμή του. Χρειάζεται γιατί βάσει του τύπου του πεδίου εισαγωγής θα γίνεται ο έλεγχος εγκυρότητας και με διαφορετικό τρόπο.

Σύμφωνα με το εγχειρίδιο θα πρέπει να γίνει δήλωση κάποιων περιορισμών (`constraints`) που είναι το `presence` δηλαδή αν θα πρέπει να υπάρχει τιμή και η μορφή του (`format`). Βασίζεται σε `regular expressions` (`regex`) και έχει την μορφή `"[a-z0-9]+"`. Το `flag` επιτρέπει τον έλεγχο και των κεφαλαίων γραμμάτων, (`case sensitive`) ενώ χωρίς αυτό θα έπρεπε να δηλωθούν χωριστά. Το `message` είναι το μήνυμα λάθους που επιστρέφεται και μπορεί να μετατραπεί σε αυτό που επιθυμούμε, επίσης η δήλωσή του είναι προαιρετική.

Ο έλεγχος του πρώτου ονόματος και του επίθετου ώστε να μην είναι άδεια πεδία και αν περιέχουν μόνο γράμματα. Τα κεφαλαία γράμματα περνούν τον έλεγχο εξίσου. Οτιδήποτε άλλο αποτυγχάνει να περάσει. Σε νέο φάκελο `utils` και αρχείο `validationConstraints` θα μεταφερθεί όλη η λειτουργικότητα καθώς πρόκειται για επαναλαμβανόμενο κώδικα.

```
export const validateString = function (id, value) {
  const constraints = {
    presence: { allowEmpty: false }
  };

  if (value !== '') {
    constraints.format = {
      pattern: '[a-z0-9]+',
      flags: 'i',
      message: 'Value can only contain letters'
    }
  }

  const validationResult = validate({ [id]: value }, { [id]: constraints });
  return validationResult && validationResult[id];
};
```

Στην κονσόλα με δοκιμές φαίνεται πως όταν το πεδίο είναι έγκυρο να επιστρέφει `undefined` από το `validation`. Εάν είναι `undefined` ξέρουμε ότι είναι έγκυρο. Όταν δεν είναι έγκυρο παίρνουμε πίσω ένα `object` που περιέχει το `id` σαν το `key` του πεδίου που κάνουμε έλεγχο και μέσα το μήνυμα λάθους. Δηλώνοντας

`return validationResult && validationResult[id]` έχουμε έτοιμο το μήνυμα λάθους και μπορούμε να το εμφανίσουμε στην οθόνη.

2.6.1 Έλεγχος διεύθυνσης email και κωδικού πρόσβασης

Με την ίδια λογική η μέθοδος ελέγχου θα είναι όπως και πριν. Το όνομα της θα είναι `validateEmail`. Οι περιορισμοί παραμένουν ίδιοι ότι δεν μπορεί να είναι άδειο το πεδίο. Η βιβλιοθήκη αναφέρει πως με την χρήση του

`constraints.email = true` ελέγχει αυτόματα αν πρόκειται για διεύθυνση email.

```
const constraints = {
  presence: { allowEmpty: false }
};

if (value !== '') constraints.email = true;
```

Για την κωδικό πρόσβασης θα οριστεί η μέθοδος `validatePassword` και απλά θα γίνει έλεγχος του μήκους του (`length`). Θέλουμε να είναι το λιγότερο 6 χαρακτήρες καθώς και το μήνυμα λάθους που θα εμφανίζεται όταν δεν ικανοποιείται αυτή η συνθήκη.

```
const constraints = {
  presence: { allowEmpty: false }
};

if (value !== '') {
  constraints.length = {
    minimum: 6,
    message: 'must be at least 6 characters'
  }
}
```

Γίνονται κάποιες αλλαγές στο Input component για να δέχεται properties που κρύβουν τον κωδικό καθώς πληκτρολογεί ο χρήστης και τα γράμματα να είναι πεζά αντί για κεφαλαία. Περνάνε από τα μέρη που καλούμε το Input με τα ονόματα:

`autoCapitalize='none'`

`secureTextEntry` έχει την τιμή `true`. Αυτό σημαίνει όταν αρχίσει να πληκτρολογεί ο χρήστης τον κωδικό αυτός θα παραμένει κρυφός.

`keyboardType='email-address'` στο email πεδίο. Όταν ανοίγει το πληκτρολόγιο του κινητού εμφανίζει το σύμβολο '@' που είναι χρήσιμο για την ευκολότερη συμπλήρωσή του.

Τα ίδια βήματα ακολουθούνται και στον φόρμα σύνδεσης χρήση (`SignInForm`).

Οι διαδικασίες στα δύο αυτά αρχεία είναι παρόμοιες, πράγμα που σημαίνει πως ο κώδικας επαναλαμβάνεται. Για αυτό σε ξεχωριστό φάκελο μέσα στα `utils` με το όνομα `actions` φτιάχνεται ένα νέο αρχείο `formActions` που θα περιλαμβάνει τον ενιαίο κώδικα και από εδώ θα χρησιμοποιείται όπου είναι απαραίτητο.

Η μέθοδος που τα περιέχει είναι η `validateInput` και θα γίνει χρήση της στην φόρμα σύνδεσης (`SignInForm`) αλλά και στην φόρμα εγγραφής (`SignUpForm`).

```
import {
  validateEmail,
  validateLength,
  validatePassword,
  validateString
} from '../validationConstraints';

export const validateInput = function (inputId, inputValue) {
  if (inputId === 'firstName' || inputId === 'lastName')
    return validateString(inputId, inputValue);

  else if (inputId === 'email')
    return validateEmail(inputId, inputValue);

  else if (inputId === 'password')
    return validatePassword(inputId, inputValue);
}
```

2.6.2 useReducer Hook

Το `useReducer` αποτελεί ένα από τα βασικά Hooks της βιβλιοθήκης React και χρησιμοποιείται για τη διαχείριση της κατάστασης (state) ενός component, ιδιαίτερα όταν αυτή χαρακτηρίζεται από αυξημένη πολυπλοκότητα ή όταν η μεταβολή της εξαρτάται από πολλαπλές ενέργειες. Το `useReducer` προσφέρει μια περισσότερο οργανωμένη και επεκτάσιμη προσέγγιση όταν η κατάσταση περιλαμβάνει πολλαπλές ιδιότητες ή όταν απαιτείται η εκτέλεση σύνθετης λογικής για την ενημέρωσή της. Η λειτουργία του `useReducer` βασίζεται σε δύο κύρια στοιχεία: τη συνάρτηση **reducer** και τη συνάρτηση **dispatch**. Η `reducer` είναι υπεύθυνη για τον καθορισμό του τρόπου με τον οποίο θα μεταβληθεί η κατάσταση της εφαρμογής, ενώ η `dispatch` χρησιμοποιείται για την αποστολή μιας ενέργειας (action) προς τη `reducer`. Κάθε ενέργεια περιλαμβάνει πληροφορίες σχετικά με το είδος της μεταβολής που πρέπει να πραγματοποιηθεί, επιτρέποντας στη `reducer` να επιστρέψει τη νέα κατάσταση του component.

Αντί να τροποποιείται το υπάρχον αντικείμενο, η `reducer` επιστρέφει πάντοτε ένα νέο αντικείμενο κατάστασης που ενσωματώνει τις απαιτούμενες αλλαγές. Η προσέγγιση αυτή επιτρέπει στη React να εντοπίζει αποτελεσματικά τις μεταβολές και να πραγματοποιεί μόνο τις απαραίτητες ανανεώσεις της διεπαφής χρήστη (re-render), βελτιώνοντας τόσο την απόδοση όσο και την αξιοπιστία της εφαρμογής. Η χρήση του `useReducer` παρουσιάζει σημαντικά πλεονεκτήματα σε εφαρμογές με σύνθετη λογική διαχείρισης δεδομένων καθώς συγκεντρώνει όλη τη λογική ενημέρωσης της κατάστασης σε μία κεντρική συνάρτηση, γεγονός που καθιστά τον κώδικα περισσότερο οργανωμένο και ευανάγνωστο.

Θα υπάρχουν δύο τιμές, μία θα είναι το **formState** και η δεύτερη τιμή θα είναι η μέθοδος που το ενημερώνει με όνομα `dispatchFormState`. Ο λόγος που ονομάζεται έτσι είναι γιατί αποστέλλει ένα action σε μία συνάρτηση που θα ενημερώσει το `formState` και θα επιστρέψει το ενημερωμένο state. Το `useReducer` δέχεται δύο παραμέτρους με την πρώτη να είναι η μέθοδος που καλείται για να ενημερώσει το state (`reducer`). Η `reducer` παίρνει δύο παραμέτρους ένα state που είναι η τρέχουσα κατάσταση και ένα action που στην ουσία είναι δεδομένα που δέχεται αυτή η μέθοδος. Η `reducer` επίσης δέχεται τα δεδομένα της αρχικής κατάστασης (`initialState`).

```
const [formState, dispatchFormState] = useReducer(reducer, initialState);
```

Το `initialState` θα έχει πεδία (`properties`) που θα καταγράφουν εάν αυτά που έχει εισάγει ο χρήστης είναι έγκυρα ή όχι όπως και ένα ακόμη που θα κρατάει εάν ολόκληρη η φόρμα είναι έγκυρη (`formIsValid`). Το τελευταίο μπορεί να χρησιμοποιηθεί στο `SubmitButton` απευθείας και θα κρίνει εάν το κουμπί υποβολής της φόρμας θα είναι απενεργοποιημένο ή ενεργοποιημένο.

```
inputValidities: {
  firstName: false,
  lastName: false,
  email: false,
  password: false
},
formIsValid: false
```

Με την κλήση της `dispatchFormState` που στην ουσία είναι σα να καλείται η συνάρτηση `reducer` (που ενημερώνει το `formState`) θα περάσουμε ένα αντικείμενο (`object`) με τα δεδομένα.

```
dispatchFormState({ inputId, validationResult: result, inputValue });
```

```
let updatedFormIsValid = true;

for (const key in updatedValidities) {
  if (updatedValidities[key] !== undefined) {
    updatedFormIsValid = false;
    break;
  }
};
```

Τα τρία αυτά δεδομένα θα περαστούν σαν παράμετροι στη συνάρτηση `reducer`, δηλαδή θα είναι διαθέσιμα σε αυτήν. Με την χρήση τους θα έχουμε ένα ενημερωμένο αντικείμενο με όλα τα `inputValidities` βάσει του πεδίου στο οποίο γίνεται ο έλεγχος. Κατά αυτόν τον τρόπο θα εξετάσει τα τρέχοντα πεδία του `state` και σε επόμενο βήμα ενημερώνει αυτό που την στιγμή συμπληρώνεται από τον χρήστη. Εάν βρεθεί ένα πεδίο που είναι `false` τότε ολόκληρη η φόρμα δεν είναι έγκυρη γιατί αν οποιοδήποτε από αυτά δεν είναι έγκυρο τότε αρκεί για να την καταστήσει ως άκυρη. Ο λόγος που η συνθήκη ελέγχει εάν δεν είναι `undefined` είναι γιατί το `validationResult` θα είναι `undefined` όταν το πεδίο που εισάγεται από τον χρήστη είναι έγκυρο.

```
const updatedValidities = {
  ...state.inputValidities,
  [inputId]: validationResult
  // When we want to use a var
  // Take all of these from th
  // whatever that inputId is
};
```

Τέλος, η `reducer` θα επιστρέψει τα ενημερωμένα δεδομένα `inputValidities`.

```
return {
  inputValues: updatedValues,
  inputValidities: updatedValidities,
  formIsValid: updatedFormIsValid
};
```

Η λειτουργικότητα της `reducer` θα μεταφερθεί σε ξεχωριστό αρχείο για να μπορεί να χρησιμοποιηθεί και άλλα μέρη της εφαρμογής. Δημιουργία νέου φακέλου `reducers` μέσα στον `utils` και το όνομα του αρχείου είναι `formReducer`.

Τα βήματα που παρουσιάστηκαν παραπάνω έγιναν για την φόρμα εγγραφής του χρήστη (SignUpForm). Το ίδιο θα γίνει και για την φόρμα σύνδεσης (login) του χρήστη στο αρχείο (SignInForm). Στο σημείο της επικύρωσης παραλείπονται τα πεδία (firstName, lastName) και ο έλεγχος θα γίνει μόνο για την διεύθυνση ταχυδρομείου καθώς και τον κωδικό πρόσβασης.

2.6.3 Εμφάνιση μηνύματος λανθασμένης εισαγωγής στοιχείων

Σε κάθε Input component θα προστεθεί ένα `errorText` property και αυτό με την σειρά του το χειρίζεται το ίδιο Input. Ο παρακάτω έλεγχος αφορά το πεδίο email.

```
errorText={formState.inputValidities['email']}
```

Τα μηνύματα λάθους που επιστρέφονται από τη συνάρτηση validation είναι σε μορφή Array γιατί μπορεί να είναι πολλαπλά σφάλματα, έτσι θέλουμε αυτό που βρίσκεται στην πρώτη θέση αυτής της διάταξης.

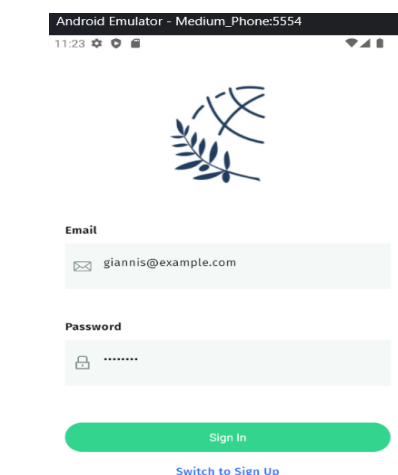
```
<Text style={styles.errorText}>{props.errorText[0]}</Text>
```

Το ίδιο θα επαναληφθεί για το πεδίο password:

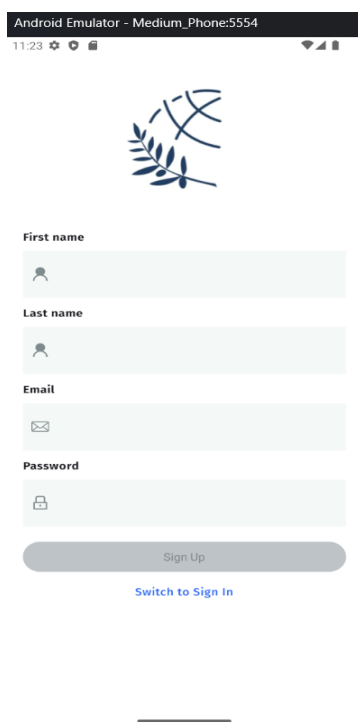
```
errorText={formState.inputValidities['password']}
```

Στο αρχείο SignUpForm η ίδια διαδικασία για όλα τα πεδία εισαγωγής συμπεριλαμβανομένου το πρώτου ονόματος (firstName) και επίθετου (lastName).

2.6.4 Στιγμιότυπα Οθόνης με Χρήση του Android Emulator



Εικόνα 2.2 - Οθόνη σύνδεσης χρήστη



Εικόνα 2.3 - Οθόνη εγγραφής χρήστη



The screenshot shows an Android emulator interface. At the top, the status bar displays 'Android Emulator - Medium_Phone:5554' and the time '11:31'. Below the status bar is a blue header with a white logo of a stylized leaf. The main content area contains a sign-up form with the following fields and errors:

- First name:** The input field contains '2334something'. Below it, a red error message reads: 'First name Value can only contain letters'.
- Last name:** The input field contains '33last'. Below it, a red error message reads: 'Last name Value can only contain letters'.
- Email:** The input field contains 'someEmail'. Below it, a red error message reads: 'Email is not a valid email'.
- Password:** The input field contains four dots '....'. Below it, a red error message reads: 'Password must be at least 6 characters'.

At the bottom of the form, there is a grey 'Sign Up' button and a blue link that says 'Switch to Sign In'.

Εικόνα 2.4 - Ο Χρήστης έχει εισάγει όλα τα στοιχεία του λανθασμένα

2.7 Εγγραφή Χρήστη

Η συνάρτηση που θα φροντίσει για την επαλήθευση του χρήστη είναι η `authHandler`. Δηλώνεται στο αρχείο `SignUpForm` και `SignInForm`. Ενεργοποιείται όταν πατηθεί το κουμπί υποβολής της φόρμας (Submit button) αφού δηλωθεί στο `prop onPress`.

Ο χειρισμός θα περαστεί σε νέο αρχείο `authActions` στον φάκελο `utils` και `actions`. Η μέθοδος `signUp` που δημιουργείται εκεί θα καλείται μέσω του `authHandler`, συνεπώς όταν πατηθεί το κουμπί υποβολής. Οι τιμές των εισαχθέντων στοιχείων θα είναι αποθηκευμένες σε ξεχωριστά πεδία που θα ενημερώνονται καθώς αλλάζει η κατάσταση (state). Δηλώνονται με αρχικές τιμές το κενό string.

```
const initialState = {
  inputValues: {
    firstName: '',
    lastName: '',
    email: '',
    password: ''
  },
```

```
  return {
    inputValues: updatedValues,
    inputValidities: updatedValidities,
    formIsValid: updatedFormIsValid
  };
```

Έπειτα θα πρέπει να ενημερωθεί η reducer συνάρτηση ώστε να δέχεται αυτά τα πεδία και να τα επιστρέφει πίσω ενημερωμένα με τις νέες τιμές. (formReducer)

```
const updatedValues = {
  ...state.inputValues,
  [inputId]: inputValue
};
```

Οι ενημερωμένες τιμές θα είναι οι παράμετροι στην κλήση της μεθόδου signUp που μετέπειτα θα στέλνει αυτά τα δεδομένα στην βάση.

Το αρχείο SignInForm θα ενημερωθεί και αυτό με την σειρά του με τις αρχικές τιμές inputValues για τα πεδία email και password.

Μετά από τα παραπάνω, τα δεδομένα που έχει εισάγει ο χρήστης στα πεδία συμπλήρωσης είναι 'αποθηκευμένα' στο state. Τώρα μπορούν να χρησιμοποιηθούν για να ξεκινήσει η εγγραφή του.

2.8 Επίλογος

Στο παρόν κεφάλαιο παρουσιάστηκε η διαδικασία ανάπτυξης των βασικών στοιχείων της διεπαφής και της λειτουργικότητας της εφαρμογής ανταλλαγής μηνυμάτων. Η ανάπτυξη πραγματοποιήθηκε σταδιακά, ξεκινώντας από την προσαρμογή της εμφάνισης της εφαρμογής και συνεχίζοντας με την υλοποίηση του συστήματος πλοήγησης, των βασικών οθονών και των μηχανισμών αλληλεπίδρασης με τον χρήστη.

Αρχικά πραγματοποιήθηκε η εισαγωγή εξωτερικής γραμματοσειράς και η δημιουργία μιας οθόνης SplashScreen, η οποία εμφανίζεται κατά την εκκίνηση της εφαρμογής και παραμένει ενεργή μέχρι να ολοκληρωθεί η φόρτωση των απαραίτητων πόρων. Με τον τρόπο αυτό δημιουργείται μια περισσότερο οργανωμένη διαδικασία αρχικοποίησης της εφαρμογής και βελτιώνεται η συνολική εμπειρία του χρήστη.

Στη συνέχεια υλοποιήθηκε το σύστημα πλοήγησης με τη χρήση της βιβλιοθήκης React Navigation. Η συνδυασμένη χρήση του Stack Navigator και του Bottom Tab Navigator επέτρεψε τη δημιουργία μιας οργανωμένης δομής πλοήγησης μεταξύ των διαφορετικών οθονών της εφαρμογής. Παράλληλα, η

μεταφορά της λειτουργικότητας των Navigators σε ξεχωριστά αρχεία συνέβαλε στη βελτίωση της δομής του πηγαίου κώδικα και στη διευκόλυνση της μελλοντικής επέκτασης της εφαρμογής.

Ιδιαίτερη σημασία είχε η δημιουργία της οθόνης συνομιλίας, η οποία αποτελεί ένα από τα βασικότερα στοιχεία της εφαρμογής. Μέσω της χρήσης στοιχείων όπως τα TextInput, TouchableOpacity, ImageBackground και SafeAreaView δημιουργήθηκε το αρχικό περιβάλλον ανταλλαγής μηνυμάτων. Επιπλέον, εφαρμόστηκαν μηχανισμοί για την προσαρμογή της διεπαφής στην εμφάνιση του πληκτρολογίου της συσκευής, με στόχο τη διατήρηση της σωστής λειτουργίας της οθόνης σε διαφορετικές συνθήκες χρήσης.

Σημαντικό μέρος της ανάπτυξης αποτέλεσε επίσης η δημιουργία του συστήματος εγγραφής και σύνδεσης των χρηστών. Για την υλοποίηση των σχετικών οθονών δημιουργήθηκαν επαναχρησιμοποιήσιμα components, όπως το PageContainer, το Input και το SubmitButton. Η χρήση επαναχρησιμοποιήσιμων components επέτρεψε τη μείωση της επανάληψης κώδικα και τη διατήρηση μιας κοινής μορφοποίησης μεταξύ διαφορετικών οθονών της εφαρμογής.

Παράλληλα, υλοποιήθηκε μηχανισμός ελέγχου εγκυρότητας των δεδομένων που εισάγει ο χρήστης στις φόρμες. Μέσω της χρήσης της βιβλιοθήκης validate.js πραγματοποιείται έλεγχος των πεδίων εισαγωγής, ενώ σε περίπτωση λανθασμένων δεδομένων εμφανίζονται κατάλληλα μηνύματα σφάλματος. Η διαδικασία αυτή συμβάλλει στη βελτίωση της αξιοπιστίας της εφαρμογής, καθώς περιορίζει την πιθανότητα αποστολής μη έγκυρων ή ελλιπών δεδομένων.

Για τη διαχείριση της κατάστασης των φορμών χρησιμοποιήθηκε το Hook useReducer της React. Η συγκεκριμένη προσέγγιση επέτρεψε την οργανωμένη διαχείριση πολλαπλών πεδίων και των αντίστοιχων καταστάσεων εγκυρότητας. Μέσω της χρήσης μιας κεντρικής reducer συνάρτησης, η λογική ενημέρωσης της κατάστασης συγκεντρώνεται σε ένα συγκεκριμένο σημείο, γεγονός που καθιστά τον κώδικα περισσότερο οργανωμένο και διευκολύνει τη συντήρηση και την επέκτασή του.

Τέλος, τα δεδομένα που εισάγει ο χρήστης στις φόρμες εγγραφής και σύνδεσης αποθηκεύονται στην κατάσταση της εφαρμογής και προετοιμάζονται για χρήση από τον μηχανισμό αυθεντικοποίησης. Με την ολοκλήρωση των παραπάνω λειτουργιών έχει δημιουργηθεί η βασική υποδομή της διεπαφής χρήστη και του συστήματος διαχείρισης των στοιχείων του χρήστη.

Συνεπώς, το δεύτερο κεφάλαιο αποτελεί ένα σημαντικό στάδιο της συνολικής ανάπτυξης της εφαρμογής, καθώς θέτει τις βάσεις για την υλοποίηση των επόμενων λειτουργιών.

Κεφάλαιο 3ο: Σύστημα Back-End

3.1 Εισαγωγή

Στο παρόν κεφάλαιο παρουσιάζεται αναλυτικά η υλοποίηση των μηχανισμών που σχετίζονται με τη διαχείριση των χρηστών, την αυθεντικοποίηση και τη διατήρηση της κατάστασης της εφαρμογής. Οι συγκεκριμένες λειτουργίες αποτελούν βασικό μέρος της εφαρμογής ανταλλαγής μηνυμάτων, καθώς είναι απαραίτητο να υπάρχει ένας αξιόπιστος μηχανισμός μέσω του οποίου κάθε χρήστης θα μπορεί να δημιουργεί και να διαχειρίζεται τον προσωπικό του λογαριασμό, να πραγματοποιεί ασφαλή σύνδεση και να έχει πρόσβαση στα προσωπικά του δεδομένα.

Για την υλοποίηση των παραπάνω λειτουργιών αξιοποιείται η πλατφόρμα Firebase, η οποία παρέχει τις απαραίτητες υπηρεσίες για την αυθεντικοποίηση των χρηστών και την αποθήκευση των δεδομένων της εφαρμογής. Αρχικά, παρουσιάζεται η διαδικασία ενσωμάτωσης του Firebase στο περιβάλλον ανάπτυξης της εφαρμογής, το οποίο βασίζεται σε React Native και Expo. Στη συνέχεια, αναλύεται η διαδικασία δημιουργίας νέου λογαριασμού, η επικοινωνία με την υπηρεσία Authentication του Firebase, καθώς και η διαχείριση πιθανών σφαλμάτων που μπορεί να προκύψουν κατά την εγγραφή.

Παράλληλα, εξετάζεται η αποθήκευση των επιπλέον πληροφοριών του χρήστη στη Realtime Database. Μετά την επιτυχή δημιουργία ενός λογαριασμού, τα δεδομένα του χρήστη καταχωρούνται στη βάση δεδομένων και συσχετίζονται με το μοναδικό αναγνωριστικό (UID) που παρέχεται από το Firebase. Με αυτόν τον τρόπο επιτυγχάνεται η οργάνωση και η εύκολη ανάκτηση των δεδομένων που σχετίζονται με κάθε χρήστη.

Ιδιαίτερη σημασία έχει επίσης η διαχείριση της κατάστασης της εφαρμογής. Για τον σκοπό αυτό χρησιμοποιείται η βιβλιοθήκη Redux, μέσω της οποίας δημιουργείται ένα κεντρικό store στο οποίο αποθηκεύονται και διαχειρίζονται δεδομένα που πρέπει να είναι διαθέσιμα σε διαφορετικές οθόνες της εφαρμογής. Παρουσιάζεται η δημιουργία του store και του authSlice, καθώς και η χρήση των actions, της useDispatch και της useSelector για την ενημέρωση και την ανάκτηση των δεδομένων της κατάστασης.

Επιπλέον, εξετάζεται το ζήτημα της διατήρησης της κατάστασης μετά την επανεκκίνηση της εφαρμογής. Επειδή η κατάσταση που διαχειρίζεται η Redux δεν διατηρείται αυτόματα μετά τον τερματισμό ή την επανεκκίνηση της εφαρμογής, χρησιμοποιείται το AsyncStorage για την τοπική αποθήκευση των απαραίτητων στοιχείων αυθεντικοποίησης. Με τον τρόπο αυτό υλοποιείται μηχανισμός αυτόματης σύνδεσης, μέσω του οποίου ελέγχεται η ύπαρξη και η εγκυρότητα των αποθηκευμένων δεδομένων και, εφόσον είναι δυνατή η επαναχρησιμοποίησή τους, ο χρήστης παραμένει συνδεδεμένος χωρίς να απαιτείται εκ νέου εισαγωγή των διαπιστευτηρίων του.

Τέλος, παρουσιάζεται η λειτουργία της οθόνης σύνδεσης και ο τρόπος με τον οποίο αυτή συνεργάζεται με το Firebase, τη Redux και το AsyncStorage. Αναλύεται επίσης η διαχείριση λανθασμένων διαπιστευτηρίων και η διαδικασία αυτόματης αποσύνδεσης όταν το token αυθεντικοποίησης παύσει να είναι έγκυρο. Συνεπώς, το κεφάλαιο αποσκοπεί στην παρουσίαση της συνολικής αρχιτεκτονικής που υιοθετήθηκε για τη διαχείριση της ταυτότητας και της κατάστασης του χρήστη, καθώς και στον τρόπο με τον οποίο οι επιμέρους τεχνολογίες συνεργάζονται για την ομαλή και συνεπή λειτουργία της εφαρμογής.

3.2 Google Firebase

Στο firebase θα αποθηκεύονται όλες οι πληροφορίες που αφορούν τον χρήστη. Θα είναι η βάση δεδομένων και για την υπόλοιπη εφαρμογή για τα μηνύματα, εικόνες και άλλα. Περιέχει σε πραγματικό χρόνο (real time) βάση δεδομένων όπως επίσης αποθηκευτικό χώρο που μπορεί να αξιοποιηθεί για εικόνες και άλλα αρχεία. Πρόκειται για ένα αξιόπιστο εργαλείο που κάτοχός του είναι η Google έτσι κάποιος μπορεί να είναι σίγουρος πως έχει πολύ καλή υποστήριξη. Τέλος, αποτελεί σημαντικό κομμάτι της δουλειάς πολλών χρηστών.

Από το firebase console δημιουργούμε νέο project με το όνομα που θέλουμε. Δίνει από μόνο του ένα μοναδικό αναγνωριστικό για το project καθώς και την επιλογή για την περιοχή που θα βρίσκεται ο server της βάσης. Με αυτό τον τρόπο δημιουργείται το βασικό project. Από κάτω του θα φτιαχτεί η εφαρμογή (app), δηλαδή μπορούν να υπάρχουν ξεχωριστές εφαρμογές για συσκευή android ή για συσκευή IOS. Επειδή χρησιμοποιείται η React native, δομείται μόνο μία φορά και θα απεικονίζεται σε όλες τις πλατφόρμες, δεν είναι απαραίτητο να γίνει αυτός ο διαχωρισμός. Θα χρησιμοποιηθεί η επιλογή για δημιουργία project web.

Η εγκατάσταση του πακέτου Firebase γίνεται με την εντολή

npm install firebase

Δίνονται πληροφορίες σχετικά με τις ρυθμίσεις παραμέτρων της εφαρμογής που χρειάζονται για να υπάρχει πρόσβαση στην βάση. Είναι προσωπικά και μοναδικά για χρήστη. Η σύνδεση θα γίνεται με το παρακάτω απόσπασμα.

```
// Import the functions you need from the SDKs you need
import { initializeApp } from "firebase/app";
// TODO: Add SDKs for Firebase products that you want to use
// https://firebase.google.com/docs/web/setup#available-libraries

// Your web app's Firebase configuration
const firebaseConfig = {
  apiKey: "AIzaSyCveVXGRTWOf7aiFuss0p74l3mtP0KQqLI",
  authDomain: "thesisproject-70868.firebaseio.com",
  databaseURL: "https://thesisproject-70868-default-rtdb.firebaseio.com",
  projectId: "thesisproject-70868",
  storageBucket: "thesisproject-70868.firebaseio.com",
  messagingSenderId: "551698389000",
  appId: "1:551698389000:web:c4f27b9552af69eeacd9bd"
};

// Initialize Firebase
const app = initializeApp(firebaseConfig);
```

Η εφαρμογή όπως έχει αναφερθεί είναι στημένη σε περιβάλλον Expo που βοηθάει να στηθεί μια εφαρμογή σε React Native. Το Expo χρησιμοποιεί ένα bundler που λέγεται Metro. Το bundler είναι ένα εργαλείο ανάπτυξης εφαρμογών που συνδυάζει πολλά αρχεία JavaScript μαζί και τα μετατρέπει σε ένα το οποίο είναι έτοιμο προς χρήση στον browser. Στην προκειμένη περίπτωση θα εκτελείται και μέσω αυτού θα γίνεται η απεικόνιση στην κινητή συσκευή.

Οι ρυθμίσεις παραμέτρων του firebase θα χρησιμοποιούνται σε διάφορα σημεία συνεχώς κατά την ανάπτυξη, συνεπώς θα μετακινηθούν σε ξεχωριστό αρχείο για ευκολότερη πρόσβαση και χρήση του. Θα βρίσκεται στον φάκελο utils με όνομα αρχείου firebaseHelper.js.

```
// Import the functions you need from the SDKs you need
import { initializeApp } from "firebase/app";

export const getFirebaseApp = function () {
  // TODO: Add SDKs for Firebase products that you want to use
  // https://firebase.google.com/docs/web/setup#available-libraries

  // Your web app's Firebase configuration
  const firebaseConfig = {
    apiKey: "AIzaSyCveVXGRTW0f7aiFuss0p74l3mtP0KQqLI",
    authDomain: "thesisproject-70868.firebaseio.com",
    databaseURL: "https://thesisproject-70868-default-rtdb.europe-west1.firebaseio.com",
    projectId: "thesisproject-70868",
    storageBucket: "thesisproject-70868.firebaseio.com",
    messagingSenderId: "551698389000",
    appId: "1:551698389000:web:c4f27b9552af69eeacd9bd"
  };

  // Initialize Firebase
  return initializeApp(firebaseConfig);
};
```

Σειρά έχουν οι ενέργειες του χρήστη καθώς θα κάνει εγγραφή. Η συνάρτηση `signUp` θα χειρίζεται την λειτουργία αυτή. Η σύνδεση με την βάση γίνεται με την χρήση του βοηθητικού αρχείου `firebaseHelper` και την κλήση της συνάρτησης `getFirebaseApp`. Η `getAuth` επιστρέφει το authentication instance, στην ουσία επιτρέπει να επικοινωνήσουμε με την εφαρμογή που βρίσκεται στην κονσόλα του firebase.

Η συνάρτηση `createUserWithEmailAndPassword` έρχεται από τον φάκελο `auth` του firebase και δημιουργεί έναν χρήστη με διεύθυνση ταχυδρομείου (`email`) και κωδικό πρόσβασης (`password`). Ο χρήστης θα εμφανιστεί στην κονσόλα του firebase μαζί με τα δύο στοιχεία του.

Η πλατφόρμα προσφέρει πολλά παραδείγματα χειρισμού σφαλμάτων, για παράδειγμα όταν γίνει εισαγωγή ενός email που υπάρχει ήδη. Μάλιστα δίνεται ο κωδικός του εκάστοτε σφάλματος που μπορεί να προκύψει. Γνωρίζοντας τον κωδικό σφάλματος είναι δυνατή η διαχείριση του λάθους με διαφορετικό τρόπο εμφανίζοντας κατάλληλο μήνυμα στον χρήστη.

```
catch (error) {
  const errorCode = error.code;
  let message = 'Something went wrong.';

  if (errorCode === 'auth/email-already-in-use')
    message = 'This email is already in use';

  throw new Error(message);
}
```

Ο χειρισμός του σφάλματος θα γίνει μετέπειτα. Δίνοντας στην μεταβλητή `message` το παραπάνω μήνυμα καλύπτονται όλες οι υπόλοιπες περιπτώσεις. Εάν κάπως συμβεί ένα σφάλμα, τότε για όλες θα εμφανίζεται το συγκεκριμένο μήνυμα χωρίς να χρειαστεί να εξεταστούν ξεχωριστά.

Καλώντας την συνάρτηση `signUp` υπάρχει το ενδεχόμενο να οδηγήσει σε λάθος το οποίο πρέπει και να χειριστούμε ή αλλιώς όπως λέμε “να πιάσουμε το error”. Το error προέρχεται από το αρχείο `authActions` όπου ορίζεται η `signUp` και είναι ότι περιλαμβάνει το `message` μέσα στο `catch block`.

Σε μία state μεταβλητή `error` θα περιλαμβάνονται τα σφάλματα του οποίου την τιμή θα αλλάζει η συνάρτηση `setError`. Ο χειρισμός του θα γίνεται σε ξεχωριστό `useEffect`.

```
useEffect(() => {
  if (error)
    Alert.alert('An error occurred', error, [{ text: 'Okay' }]);
}, [error]); // Run only anytime an error changes
```

Όταν ο χρήστης πατήσει το κουμπί για εγγραφή θα υπάρχει κάποια μικρή καθυστέρηση μέχρι να ολοκληρωθεί η ενέργεια. Κατά το διάστημα αυτό θα εμφανίζεται ένδειξη φόρτωσης των δεδομένων. Η state μεταβλητή `isLoading` θα χειρίζεται αυτή την ενέργεια και η συνάρτηση `setIsLoading` θα αλλάζει την κατάστασή της. Η `setIsLoading` θα παίρνει τιμή μέσα στην `authHandler` μέθοδο και θα εναλλάσσεται σε `true` ή `false` στο `try/catch` block. Όσον αφορά την χρήση του θα γίνεται στο σημείο που δηλώνεται το κουμπί υποβολής της φόρμας. Εκεί ελέγχεται το `isLoading` state και απεικονίζεται είτε η ένδειξη φόρτωσης είτε το κουμπί το ίδιο.

```
{
  isLoading ?
  <ActivityIndicator
    size={'small'}
    color={colors.primary}
    style={{ marginTop: 10 }}
  /> :
  <SubmitButton
    title='Sign Up'
    onPress={authHandler}
    style={{ marginTop: 20 }}
    disabled={!formState.formIsValid}
  />
}
```

3.3 Στέλνοντας τα Δεδομένα Χρήστη στην Βάση Δεδομένων

Ως τώρα ο χρήστης μπορεί να πιστοποιήσει την γνησιότητά του στην εφαρμογή αλλά υπάρχει περιορισμός στις πληροφορίες που μπορούν να αποθηκευτούν καθώς δεν γίνεται να περαστούν τιμές εύκολα. Θέλουμε να αποθηκεύσουμε περισσότερα δεδομένα αξιοποιώντας τις δυνατότητες που παρέχει το `realtime database` του `firebase`. Όταν γίνει η εγγραφή θα δημιουργείται μία καινούργια καταχώρηση για τον χρήστη στο `Realtime Database`.

Η λογική είναι να δημιουργείται ένας κόμβος με το όνομα `users` και να γίνεται η αποθήκευση όλων των χρηστών σε αυτόν. Η καταχώρηση του χρήστη θα συμβαίνει την στιγμή που η εγγραφή του έχει γίνει επιτυχώς.

Για κάθε νέα εγγραφή δημιουργείται από το `firebase` αυτόματα ένα μοναδικό αναγνωριστικό χρήστη (`uid`) το οποίο μπορούμε να προσπελάσουμε.

Δηλώνεται συνάρτηση με την οποία θα καταχωρηθούν τα στοιχεία του χρήστη στην βάση δεδομένων με όνομα `createUser` στο αρχείο `authActions`. Το `userData` είναι όλα τα στοιχεία που θα αποθηκευτούν.

```
const firstName = `${firstName} ${lastName}`.toLowerCase();
const userData = {
  firstName,
  lastName,
  firstName,
  email,
  userId,
  signUpDate: new Date().toISOString()
};
```

```
const dbRef = ref(getDatabase());
```

Δημιουργεί αναφορά στο μέρος όπου βρίσκεται η εφαρμογή στην βάση δεδομένων. Το dbRef είναι το root level της βάσης και η αναφορά στο child ο κόμβος users που ακολουθείται από το μοναδικό αναγνωριστικό userId του κάθε χρήστη.

```
const childRef = child(dbRef, `users/${userId}`);
```

Επόμενο βήμα είναι να τοποθετηθούν τα δεδομένα στον κόμβο users που έχει δημιουργηθεί στο κεντρικό μέρος της εφαρμογής. Η μέθοδος set δέχεται ως παραμέτρους την αναφορά που δημιουργήθηκε πιο πάνω και τα δεδομένα που θα αποθηκευτούν στην θέση εκείνη.

Η συνάρτηση createUser θα κληθεί στην signUp και θα δημιουργήσει τον χρήστη.

```
await set(childRef, userData);
```

3.4 Διαχείριση Κατάστασης της Εφαρμογής με το Εργαλείο Redux

Χρειάζεται ένας τρόπος ώστε η εφαρμογή να «θυμάται» τα δεδομένα που χρησιμοποιήθηκαν όταν δημιουργήθηκε ο λογαριασμός για να κρατάει τον χρήστη συνδεδεμένο. Προς το παρόν όταν οδηγηθεί ο χρήστης στο κύριο μέρος της εφαρμογής, αφού έχει συνδεθεί, δεν υπάρχει κάποιος τρόπος να κρατάει ποιος έχει συνδεθεί καθώς τα δεδομένα δεν αποθηκεύονται κάπου. Συνεπώς χρειάζεται μία μέθοδος που θα αναλαμβάνει αυτή την λειτουργία και είναι η Redux.

Η Redux είναι μία βιβλιοθήκη της JavaScript που ουσιαστικά διαχειρίζεται την κατάσταση της εφαρμογής συνολικά, one big state application. Προηγουμένως χρησιμοποιήθηκε η useReducer για την κατάσταση της φόρμας (state) και μόνο για αυτήν. Τώρα χρειάζεται ένα state για όλη την εφαρμογή. Αυτό σημαίνει ότι σε οποιαδήποτε οθόνη και να βρεθεί ο χρήστης θα υπάρχει η δυνατότητα πρόσβασης αυτής της αποθήκης δεδομένων (data store) από όπου μπορούν να ανακτηθούν οι τιμές. Για παράδειγμα, όταν βρεθεί κανείς στην οθόνη συνομιλίας και θέλει να ανακτήσει τα δεδομένα χρήστη που έχει συνδεθεί, όπως το όνομά του ή τα μηνύματα που έχει στείλει.

Με λίγα λόγια θα υπάρχει ένα κεντρικό state για όλη την εφαρμογή στο οποίο θα υπάρχει πρόσβαση από οποιαδήποτε σελίδα.

Η εγκατάσταση της βιβλιοθήκης σύμφωνα με το εγχειρίδιο γίνεται με την εντολή:

```
npm install @reduxjs/toolkit react-redux
```

3.4.1 Χρήση Redux

Δημιουργία αποθήκης store σε νέο φάκελο με όνομα store και ένα αρχείο store.js. Εδώ θα βρίσκεται η αποθήκη της Redux και το store θα είναι το πρώτο επίπεδο διαχείρισης κατάστασης (state) της εφαρμογής.

```
import { configureStore } from "@reduxjs/toolkit";
```

Ορίζεται ένα store με την μέθοδο configureStore και εκεί θα δηλώνονται όλοι οι reducers της εφαρμογής. Οι reducers στην ουσία είναι κομμάτια κατάστασης (slices of state), properties και ενέργειες (actions) που ενημερώνουν το state.

Επόμενο βήμα είναι στο κεντρικό αρχείο App.js στο σημείο που επιστρέφεται ο κώδικας JSX να καλυφθεί ολόκληρη η δήλωση σε ένα Provider στοιχείο της React – Redux. Με την Redux οτιδήποτε βρίσκεται μέσα σε Provider στοιχείο θα έχει πρόσβαση στο state της εφαρμογής. Δεν είναι απαραίτητο να δηλωθεί σαν στοιχείο πρώτου επιπέδου, μπορούν να δηλωθούν μεμονωμένα σελίδες στον Provider

και μόνο αυτές θα έχουν πρόσβαση στο state. Αλλά επειδή θέλουμε ολόκληρη η εφαρμογή να χρησιμοποιεί το state ο Provider τοποθετείται σαν πρώτο στοιχείο.

Ακόμη, θα πρέπει να οριστεί ένα property store που θα δέχεται το store αρχείο που δημιουργήθηκε πριν με το configureStore και τους reducers.

```
return (
  <Provider store={store} stabilityCheck='never'>
    <SafeAreaProvider
      style={styles.container}
      onLayout={onLayout}
    >
      <MenuProvider>
        <AppNavigator />
      </MenuProvider>
    </SafeAreaProvider>
  </Provider>
);
```

3.4.2 Εισαγωγή των reducers στο store

Όπως ειπώθηκε και πριν οι reducers είναι κομμάτια που περιγράφουν την κατάσταση και μπορούν να περιέχουν μεθόδους που ενημερώνουν αυτό το state. Θα δημιουργηθεί το πρώτο slice μέσα στον φάκελο store με ονομασία authSlice.

```
import { createSlice } from "@reduxjs/toolkit";
```

Με την χρήση της createSlice ορίζεται το authSlice reducer με τιμές που δίνουν στα πεδία αρχικές τιμές. Μετά, δηλώνονται οι reducers που απαρτίζουν το συγκεκριμένο slice. Κάθε reducer είναι μία μέθοδος που δέχεται δύο παραμέτρους με την πρώτη να είναι η τρέχουσα κατάσταση (current state) και η δεύτερη ένα action που περιέχει δεδομένα. Αυτό που γίνεται είναι να ενημερωθεί το state.

```
const authSlice = createSlice({
  name: 'auth',
  initialState: {
    token: null,
    userData: null, // state h
    didTryAutoLogin: false
  },
});
```

```
reducers: {
  authenticate: (state, action) => {
    const { payload } = action;
    state.token = payload.token;
    state.userData = payload.userData;
    state.didTryAutoLogin = true;
  },
}
```

Σε κάθε αρχείο slice που δημιουργείται θα γίνεται η εξαγωγή των reducer και functions για να είναι διαθέσιμα και σε άλλα αρχεία κατά την ανάπτυξη.

```
export const authenticate = authSlice.actions.authenticate;
export default authSlice.reducer;
```

Το authSlice που δημιουργήθηκε θα πρέπει να οριστεί μέσα στην κεντρική αποθήκη store. Με τον τρόπο αυτό έχει διαμορφωθεί ο auth reducer (authSlice) που θα περιέχει όλες τις μεθόδους που χρειάζονται για πρόσβαση και ενημέρωση του state αυθεντικότητας χρήστη.

```
export const store = configureStore({
  reducer: {
    auth: authSlice,
```

Στο SignUpForm υπάρχει η authHandler και εκεί καλείται η signUp μέθοδος απευθείας, αυτό όμως που θα γίνει είναι να πάρουμε τα αποτελέσματα που επιστρέφει και να τα στείλουμε στο state δηλαδή στο κεντρικό data store. Αυτό που θα γίνει αποκαλείται dispatch. Το dispatch είναι ένα object και καλεί την μέθοδο useDispatch από την βιβλιοθήκη react-redux.

```
const dispatch = useDispatch();
```

Όταν θέλουμε να ενημερώσουμε το state στην ουσία αποστέλλουμε ενέργειες (actions) και μπορούμε να το κάνουμε χρησιμοποιώντας αυτή τη μέθοδο. Η useDispatch λοιπόν δίνει μία μέθοδο που μπορούμε να καλέσουμε οποιαδήποτε στιγμή για να ενημερώσουμε το state. Με αυτή τη λογική η signUp θα κληθεί μέσω του dispatch. Η παράμετρος που δέχεται η dispatch θα πρέπει να είναι μία ενέργεια, ωστόσο η signUp επιστρέφει ένα Promise καθώς πρόκειται για ασύγχρονη διαδικασία. Θα διαμορφωθεί κατάλληλα ώστε να μην είναι ασύγχρονη αλλά να επιστρέφει η ίδια μία μέθοδο που θα είναι ένα Promise και θα δέχεται σαν παράμετρο ένα αντικείμενο dispatch. Το αποτέλεσμα είναι μία μέθοδος που όταν κληθεί επιστρέφει ένα action.

```
const action = signUp(
  formState.inputValues.firstName,
  formState.inputValues.lastName,
  formState.inputValues.email,
  formState.inputValues.password
);
```

```
await dispatch(action);
```

Το παραπάνω δεν θα ενεργοποιήσει την λειτουργικότητα της signUp γιατί πλέον επιστρέφει μία ενέργεια τύπου dispatch, η οποία δεν θα εκτελεστεί μέχρι να κληθεί το παραπάνω action. Όταν το action κληθεί σε κάποια ενέργεια τότε στην πραγματικότητα θα εκτελεστεί όλη η λειτουργία της signUp μεθόδου. Αυτό που πρέπει να γίνει είναι να περαστεί σαν παράμετρος το action μέσα στην μέθοδο dispatch.

Με την παραπάνω ενέργεια γίνεται το dispatch του παραπάνω action, που με την σειρά του περνάει σαν παράμετρος και δέχεται μία dispatch μέθοδο γιατί η signUp επιστρέφει το dispatch. Αυτό μπορεί να χρησιμοποιηθεί για να ενημερωθεί το state γιατί όπως ειπώθηκε η dispatch είναι υπεύθυνη για αυτή την διαδικασία. Χρησιμοποιείται λοιπόν η dispatch μέθοδος και μέσα της καλείται σαν παράμετρος η authenticate του authSlice που δέχεται ένα token και το userData.

Το token έρχεται μαζί με τα στοιχεία του χρήστη που έχει συνδεθεί και βρίσκεται στο stsTokenManager property που επιστρέφεται με το αποτέλεσμα result.

```
const { uid, stsTokenManager } = result.user;
const { accessToken, expirationTime } = stsTokenManager;
```

Το userData το παίρνουμε από την κλήση της συνάρτησης createUser που δημιουργεί τον χρήστη στην βάση δεδομένων.

```
const userData = await createUser(firstName, lastName, email, uid);
```

Έπειτα, θα περαστούν σαν παράμετροι στην authenticate του authSlice.

```
dispatch(authenticate({ token: accessToken, userData }));
```

Συνοπτική εξήγηση:

Έχουμε το store που είναι το state όλης της εφαρμογής και έχει το authSlice, μετέπειτα θα έχει και άλλα slices. Το slice είναι ένας σωρός από reducer λογική στο ίδιο αρχείο. Περιέχει το initialState, δηλαδή το state πριν κάνουμε οτιδήποτε και κάποιες μεθόδους οι οποίες μπορούν να ενημερώσουν το state. Στο αρχείο SignUpForm αντί να καλέσουμε την signUp απευθείας την κάνουμε dispatch σαν ένα action. Αυτό γίνεται χρησιμοποιώντας το dispatch γιατί ο μόνος τρόπος να ενημερωθεί το state είναι αποστέλλοντας actions και για να αποσταλούν αυτά τα actions θα πρέπει να χρησιμοποιηθεί η dispatch που μας δίνει η useDispatch. Στην ουσία κάνουμε dispatch την signUp μέθοδο γιατί έχει γίνει μια αλλαγή σε αυτήν ώστε τώρα να επιστρέφει ένα action που είναι dispatch. Όταν λοιπόν καλέσουμε το action χρησιμοποιώντας την dispatch αυτό το action αυτόματα θα περαστεί στην μέθοδο ώστε να μπορεί να χρησιμοποιηθεί σε αυτήν για να ενημερώσει το state. Καλούμε την μέθοδο authenticate που μας δίνει η authSlice και περνάμε τα δεδομένα token και userData. Token και userData λοιπόν είναι διαθέσιμα στο action της authenticate μέσω του payload. Το payload θα έχει το token και το userData.

3.4.3 Πρόσβαση στα δεδομένα που βρίσκονται στην κατάσταση (state) της εφαρμογής

Η πρόσβαση στο state γίνεται με την χρήση της μεθόδου useSelector που προσφέρει η βιβλιοθήκη redux της react. Για την ενημέρωση του state χρησιμοποιείται η useDispatch και για την ανάκτηση δεδομένων η useSelector.

Όταν ανακτηθούν τα δεδομένα του state είναι διαθέσιμα για όλη την διάρκεια που η εφαρμογή είναι ενεργή. Όταν γίνει ανανέωση της εφαρμογής το state θα είναι άδαιο και πάλι. Αυτό λοιπόν που χρειάζεται είναι ένας τρόπος για να αποθηκεύεται έτσι ώστε όταν η εφαρμογή ξεκινήσει και πάλι, εάν ο χρήστης είναι συνδεδεμένος, να συνεχίσει από το σημείο που είχε μείνει.

Την στιγμή που θα γίνει η ανάκτηση της κατάστασης εφαρμογής ο χρήστης θα μεταφερθεί στην επόμενη σελίδα.

Τον έλεγχο και κατ' επέκταση την πλοήγηση στις οθόνες τις χειρίζεται το αρχείο AppNavigator στο οποίο υπάρχει η μεταβλητή isAuth που καθορίζει σε πια σελίδα θα μεταφερθεί ο χρήστης ανάλογα με την τιμή που θα πάρει. Οι τιμές που παίρνει είναι δύο true ή false.

Με την useSelector γίνεται η πρόσβαση στο state. Δέχεται το τρέχον state και επιστρέφει το αντίστοιχο slice που είναι επιθυμητό.

```
const isAuth = useSelector(state => state.auth.token !== null && state.auth.token !== '');
```

Γίνεται ανάκτηση του state και συγκεκριμένα εάν υπάρχει token που να έχει δημιουργηθεί για τον χρήστη που έκανε εγγραφή στην εφαρμογή. Εάν λοιπόν υπάρχει αυτό το token, δηλαδή δεν είναι κενό, τότε το isAuth θα γίνει true και ο navigator θα μεταφέρει τον χρήστη στην κύρια οθόνη της εφαρμογής (MainNavigator).

```
{ isAuth && <MainNavigator /> }
```

Όταν γίνει ανανέωση της εφαρμογής ο χρήστης αποσυνδέεται, και ο λόγος είναι γιατί δεν αποθηκεύονται τα δεδομένα που έχει εισάγει. Χρειάζεται ένας τρόπος ώστε αυτά τα δεδομένα να

διατηρούνται έτσι όταν γίνεται επανεκκίνηση της εφαρμογής ο χρήστης να παραμένει συνδεδεμένος. Θα χρησιμοποιηθεί το Async Storage της React Native που στην ουσία δίνει έναν αποθηκευτικό χώρο ο οποίος είναι προσπελάσιμος μετά από κάθε επανεκκίνηση της εφαρμογής.

Εγκατάσταση πακέτου: **npm i @react-native-async-storage/async-storage**

Τα στοιχεία που πρέπει να διατηρηθούν είναι το αναγνωριστικό χρήστη (uid), το token καθώς και την ημερομηνία λήξης του token γιατί τα tokens που προσφέρει το firebase λήγουν μετά από μία ώρα και χρειάζεται ανανέωση της εφαρμογής ή ο χρήστης πρέπει να συνδεθεί ξανά με τα διαπιστευτήριά του.

Η μέθοδος που θα αποθηκεύει τα δεδομένα είναι η `saveDataToStorage`.

```
const saveDataToStorage = (token, userId, expiryDate) => {
  AsyncStorage.setItem('userData', JSON.stringify({
    // JSON.stringify takes an object and then makes it into a string
    // object directly we have to store a string value
    token,
    userId,
    expiryDate: expiryDate.toISOString()
  }));
};
```

Η χρήση της θα γίνει μέσα στην `signUp` αφού ενημερωθεί το state.

Ο λόγος που χρησιμοποιείται μόνο το user Id και όχι όλα τα δεδομένα που αφορούν τον χρήστη είναι γιατί κάθε φορά που αυτός συνδέεται στην εφαρμογή θα γίνεται αίτημα στην βάση για να επιστρέψει το νέο user Id. Με το user Id λοιπόν θα γίνεται ανάκτηση των νέων δεδομένων κάθε φορά που ο χρήστης ανοίγει την εφαρμογή.

```
saveDataToStorage(accessToken, uid, expiryDate);
```

Από δω και έπειτα όταν ανανεώνεται η εφαρμογή τα τρία δεδομένα (`accessToken`, `uid`, `expiryDate`) θα είναι διαθέσιμα μέσω του αποθηκευτικού χώρου Async Storage.

Για να γίνει έλεγχος της λειτουργίας και ότι τα δεδομένα διατηρούνται κατάλληλα θα δημιουργηθεί μια καινούργια οθόνη `StartUpScreen` που θα ελέγχει τον αποθηκευτικό χώρο και αν υπάρχουν δεδομένα εκεί θα τα χρησιμοποιεί για να συνδέσει τον χρήστη.

Στο `AppNavigator` θα υπάρχει μεταβλητή που θα κρατάει αν ο χρήστης προσπάθησε να συνδεθεί αυτόματα.

```
const didTryAutoLogin = useSelector(state => state.auth.didTryAutoLogin);
```

Αν δεν έχει συνδεθεί και έχει προσπαθήσει να συνδεθεί αυτόματα τότε θα εμφανιστεί η οθόνη σύνδεσης (`SignInForm`). Η άλλη περίπτωση είναι να μην έχει συνδεθεί και δεν έχει ακόμη προσπαθήσει να συνδεθεί αυτόματα τότε θα οδηγηθεί στο `StartUpScreen`.

```
{ isAuth && <MainNavigator /> }
{ !isAuth && didTryAutoLogin && <AuthScreen /> }
{ !isAuth && !didTryAutoLogin && <StartUpScreen /> }
```

Στο `StartUpScreen` αρχείο θα γίνει έλεγχος εάν υπάρχει διαθέσιμο token, δηλαδή αν υπάρχει η δυνατότητα ανάκτησης των δεδομένων που έχουν αποθηκευτεί προηγουμένως στο AsyncStorage με κλειδί όνομα 'userData'.

Σε ένα side effect (`useEffect`) που καλεί την μέθοδο `tryLogin` γίνεται ο έλεγχος.

```
const storedAuthInfo = await AsyncStorage.getItem('userData');
```

Η Dispatch τροποποιεί το state και αυτό γίνεται αποστέλλοντας actions με την μέθοδο useDispatch. Η dispatch θα κληθεί για να ενημερώσει το action setDidTryAutoLogin το οποίο θα πρέπει να ενημερωθεί στο state καθώς είναι αυτό που καθορίζει πια σε οθόνη θα οδηγηθεί ο χρήστης κατά την σύνδεση (στο AppNavigator). Πρώτα όμως θα πρέπει να οριστεί το action στο authSlice με έναν reducer και όνομα setDidTryAutoLogin που θα αλλάζει την τιμή του σε αληθή (true). Η αρχική κατάσταση initialState ενημερώνεται ώστε να δίνει αρχική τιμή στο didTryAutoLogin.

```
setDidTryAutoLogin: (state, action) => {
  state.didTryAutoLogin = true;
}
```

Σε περίπτωση που δεν υπάρχουν δεδομένα στο storage τότε θα ενημερώνεται το didTryAutoLogin με χρήση της dispatch.

```
// When there is no data in the storage
if (!storedAuthInfo) {
  dispatch(setDidTryAutoLogin());
  return;
}
```

Στο σημείο αυτό που δεν υπάρχουν δεδομένα στο storage το didTryAutoLogin θα γίνει αληθές (true) και θα μεταφέρει τον χρήστη στην οθόνη σύνδεσης (signInForm).

Εάν περάσει τον παραπάνω έλεγχο σημαίνει ότι υπάρχουν δεδομένα στο storage και θα τον κρατήσει συνδεδεμένο στην εφαρμογή.

Όσον αφορά εάν το token έχει λήξει, στην περίπτωση που το expiryDate του token είναι στο παρελθόν ή δεν υπάρχει token για κάποιο λόγο ή user Id, τότε σημαίνει πως ο χρήστης προσπάθησε να συνδεθεί αυτόματα (το didTryAutoLogin γίνεται αληθές) και θα τον μεταφέρει στην οθόνη σύνδεσης.

```
// If there is a token in the storage but it has expired
if (expiryDate ≤ new Date() || !token || !userId) {
  // If any of these are true we will mark it we have tried to autoLogin
  dispatch(setDidTryAutoLogin());
  return;
}
```

Η επόμενη περίπτωση είναι να υπάρχει token στο storage, δεν έχει λήξει και είναι διαθέσιμο για χρήση.

Θα γίνει ανάκτηση των δεδομένων του χρήστη με βάση το συγκεκριμένο userId. Στον φάκελο utils και actions θα υπάρχει ένα αρχείο userActions που θα περιέχει την λειτουργία. Η μέθοδος που θα καλέσουμε είναι η getUserData. Δίνει μία αναφορά στην βάση δεδομένων στον φάκελο users στην θέση userId που περνάει σαν παράμετρος σε αυτή.

Η μέθοδος getUserData στο αρχείο userActions:

```
export const getUserData = async function (userId) {
  try {
    const app = getFirestoreApp();
    const dbRef = ref(getDatabase(app));
    const userRef = child(dbRef, `users/${userId}`);

    const snapshot = await get(userRef); // snapshot is how you get data from firebase
    return snapshot.val();
  }
  catch (error) {
    console.error(error);
  }
};
```

Γίνεται χρήση της στο StartUpScreen και επιστρέφει τα δεδομένα χρήστη με το μοναδικό αναγνωριστικό του (user Id).

```
const userData = await getUserData(userId);
```

Για να ενημερωθεί το state με τα δεδομένα του θα γίνει και πάλι η χρήση του dispatch η οποία καλεί την authenticate και τα αποθηκεύει στην εφαρμογή.

```
dispatch(authenticate({ token: token, userData }));
```

Πλέον αυτό σημαίνει πως αν γίνει ανανέωση της εφαρμογής θα κρατήσει τον χρήστη συνδεδεμένο διότι «γνωρίζει» πως υπάρχουν τα δεδομένα του κρατημένα και δεν θα τον μεταφέρει στην οθόνη σύνδεσης. Κάνει έλεγχο του storage την στιγμή που φαίνεται στην οθόνη η ένδειξη φορτώματος (loading indicator από το StartUpScreen) ανακτά τα δεδομένα του και τον συνδέει αυτόματα.

3.5 Λειτουργίες της Οθόνης Σύνδεσης

Οι λειτουργίες της θα βρίσκονται στην μέθοδο SignIn στο αρχείο authActions, τρόπος παρόμοιος με αυτόν της οθόνης εγγραφής. Αυτή τη φορά θα δέχεται σαν παράμετρο την διεύθυνση ταχυδρομείου και τον κωδικό πρόσβασης. Από την βιβλιοθήκη auth του firebase θα γίνει χρήση της μεθόδου **signInWithEmailAndPassword** που συνδέει τον χρήστη με αυτά τα δύο διαπιστευτήρια.

```
const result = await signInWithEmailAndPassword(auth, email, password);
```

Στο userActions βρίσκεται η μέθοδος getUserData που επιστρέφει τα δεδομένα αυτού.

```
const userData = await getUserData(uid);
```

Τα δεδομένα μαζί με το token χρησιμοποιούνται για την ενημέρωση του state με την dispatch μέθοδο καθώς και η saveDataToStorage που τα αποθηκεύει στο κεντρικό storage της εφαρμογής.

```
dispatch(authenticate({ token: accessToken, userData }));
saveDataToStorage(accessToken, uid, expiryDate);
```

Θα πρέπει να προστεθούν στην SignInForm οθόνη η authHandler μέθοδος που εκτελείται μόλις ο χρήστης πατήσει το κουμπί σύνδεσης.

```
const authHandler = useCallback(async () => {
  try {
    setIsLoading(true);

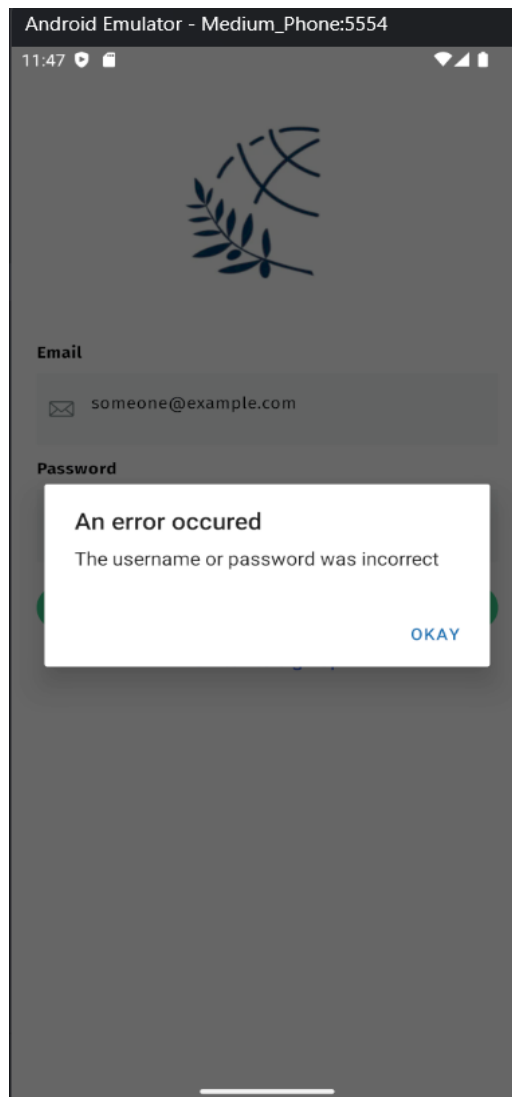
    const action = signIn(
      formState.inputValues.email,
      formState.inputValues.password
    );
    setError(null);
    await dispatch(action);
  }
  catch (error) {
    setError(error.message);
    setIsLoading(false);
  }
}, [dispatch, formState]);
```

Για την διαχείριση λαθών θα καλυφθούν δύο σενάρια, το πρώτο είναι να μην βρεθεί ο χρήστης και το δεύτερο να έχει εισάγει λάθος κωδικό πρόσβασης.

Αν λοιπόν προσπαθήσει ο χρήστης να συνδεθεί με λάθος διαπιστευτήρια θα λάβει στην οθόνη ειδοποίηση με το αντίστοιχο μήνυμα σφάλματος.

Μέρος κώδικα που χειρίζεται το λάθος:

```
catch (error) {  
  const errorCode = error.code;  
  console.log(error.code);  
  
  let message = 'Something went wrong.';  
  
  if (errorCode === 'auth/invalid-credential' || errorCode === 'auth/user-not-found')  
    message = 'The username or password was incorrect';  
  
  throw new Error(message);  
}
```



Εικόνα 3.1 - Εισαγωγή λανθασμένων στοιχείων

3.5.1 Αποσύνδεση Χρήστη Όταν Λήξει το Token

Με το πέρας κάποιου χρονικού διαστήματος ο χρήστης θα αποσυνδέεται αυτόματα όταν λήξει το token του firebase. Η μέθοδος `userLogout` στο `authActions` θα χειρίζεται την ενέργεια. Θα έχει πρόσβαση στο state και όταν γίνει η αποσύνδεση θα ‘καθαρίζει’ τα δεδομένα που βρίσκονται στο storage. Θα αφαιρεί ένα timer που έχει οριστεί και θα γίνεται `dispatch` της ενέργειας αποσύνδεσης. Αυτό το action ορίζεται στο `authSlice` σαν νέο reducer που θα μεταβάλλει το state και στην ουσία αλλάζει όλα τα δεδομένα στις τιμές που είχαν πριν ο χρήστης συνδεθεί, στην αρχική τους κατάσταση.

```
logout: (state, action) => {
  state.token = null;
  state.userData = null;
  state.didTryAutoLogin = false;
},
```

Οι ενέργειες της `userLogout` με αποσπάσματα σε μορφή κώδικα:

```
AsyncStorage.clear();
clearTimeout(timer);
dispatch(logout());
```

Για να ολοκληρωθεί θα κληθεί η `userLogout` στην μέθοδο `signIn` όπου θα οριστεί και ο μετρητής που θα μετράει αντίστροφα μέχρι να λήξει το token. Τα tokens του Firebase λήγουν μετά από μία ώρα.

```
timer = setTimeout(() => {
  dispatch(userLogout(userData));
}, millisecondsUntilExpiry);
```

Ο μετρητής θα πρέπει να προστεθεί και στην μέθοδο `signUp` για να γίνεται η ίδια διαδικασία όταν νέος χρήστης κάνει εγγραφή στην εφαρμογή.

3.6 Επίλογος

Στο παρόν κεφάλαιο αναλύθηκε η υλοποίηση των βασικών μηχανισμών που απαιτούνται για τη διαχείριση των χρηστών και της κατάστασης της εφαρμογής. Η διαδικασία ξεκίνησε με την ενσωμάτωση της πλατφόρμας Firebase, η οποία αποτέλεσε το βασικό εργαλείο για την υλοποίηση της αυθεντικοποίησης και την αποθήκευση των δεδομένων των χρηστών. Μέσω της υπηρεσίας Authentication πραγματοποιείται η δημιουργία και η επαλήθευση των λογαριασμών, ενώ η Realtime Database χρησιμοποιείται για την αποθήκευση πρόσθετων πληροφοριών που σχετίζονται με κάθε χρήστη.

Στο πλαίσιο της διαδικασίας εγγραφής υλοποιήθηκε η δημιουργία νέου χρήστη με χρήση διεύθυνσης ηλεκτρονικού ταχυδρομείου και κωδικού πρόσβασης. Παράλληλα, παρουσιάστηκε η αποθήκευση των δεδομένων του χρήστη στη βάση δεδομένων, χρησιμοποιώντας ως βασικό αναγνωριστικό το μοναδικό UID που παρέχεται από το Firebase. Η προσέγγιση αυτή επιτρέπει την οργάνωση των δεδομένων με τρόπο ώστε κάθε καταχώρηση να αντιστοιχεί με σαφήνεια στον αντίστοιχο λογαριασμό χρήστη.

Στη συνέχεια, η χρήση της βιβλιοθήκης Redux επέτρεψε την κεντρική διαχείριση της κατάστασης της εφαρμογής. Μέσω του store και του authSlice συγκεντρώνονται οι πληροφορίες που αφορούν την αυθεντικοποίηση, ενώ οι μηχανισμοί dispatch και useSelector επιτρέπουν, αντίστοιχα, την αποστολή ενεργειών για την ενημέρωση του state και την ανάκτηση των απαιτούμενων δεδομένων από αυτό. Η συγκεκριμένη αρχιτεκτονική διευκολύνει την επικοινωνία μεταξύ των διαφορετικών οθονών και περιοχών της εφαρμογής, καθώς τα απαραίτητα δεδομένα μπορούν να είναι διαθέσιμα κεντρικά χωρίς να απαιτείται η επαναλαμβανόμενη μεταφορά τους μεταξύ των επιμέρους components.

Ωστόσο, η διατήρηση της κατάστασης αποκλειστικά στη Redux δεν επαρκεί στην περίπτωση επανεκκίνησης της εφαρμογής, καθώς το περιεχόμενο του store επαναφέρεται στην αρχική του κατάσταση. Για την αντιμετώπιση του συγκεκριμένου περιορισμού αξιοποιήθηκε το AsyncStorage, στο οποίο αποθηκεύονται τα απαραίτητα στοιχεία για την επαναφορά της συνεδρίας του χρήστη. Η διαδικασία αυτή περιλαμβάνει τον έλεγχο ύπαρξης αποθηκευμένων δεδομένων, την επαλήθευση της χρονικής ισχύος του token και, εφόσον αυτό παραμένει έγκυρο, την ανάκτηση των δεδομένων του χρήστη από τη βάση δεδομένων. Με τον τρόπο αυτό επιτυγχάνεται η αυτόματη επανασύνδεση του χρήστη και αποφεύγεται η ανάγκη επανεισαγωγής των στοιχείων σύνδεσης σε κάθε εκκίνηση της εφαρμογής.

Επιπλέον, υλοποιήθηκε η διαδικασία κανονικής σύνδεσης του χρήστη, η οποία βασίζεται στην υπηρεσία Authentication του Firebase και συμπληρώνεται από την ανάκτηση των αντίστοιχων δεδομένων από τη Realtime Database. Παράλληλα, προβλέφθηκε η διαχείριση περιπτώσεων λανθασμένων διαπιστευτηρίων, ώστε ο χρήστης να ενημερώνεται κατάλληλα σε περίπτωση αποτυχίας της διαδικασίας αυθεντικοποίησης.

Τέλος, ολοκληρώθηκε ο μηχανισμός διαχείρισης της λήξης του token. Όταν το χρονικό διάστημα ισχύος του token παρέλθει, πραγματοποιείται αποσύνδεση του χρήστη, καθαρισμός των αποθηκευμένων δεδομένων και επαναφορά της κατάστασης της εφαρμογής στην αρχική της μορφή. Με τον τρόπο αυτό διασφαλίζεται ότι η εφαρμογή δεν θα συνεχίσει να θεωρεί έναν χρήστη αυθεντικοποιημένο όταν τα στοιχεία της συνεδρίας του δεν είναι πλέον έγκυρα.

Συνολικά, οι μηχανισμοί που παρουσιάστηκαν στο κεφάλαιο διαμορφώνουν το βασικό υπόβαθρο για τη διαχείριση της ταυτότητας και της κατάστασης των χρηστών. Η συνεργασία των Firebase, Redux και AsyncStorage επιτρέπει στην εφαρμογή να διαχειρίζεται οργανωμένα τη δημιουργία λογαριασμών, τη σύνδεση και αποσύνδεση, την αποθήκευση και ανάκτηση δεδομένων, καθώς και τη διατήρηση της συνεδρίας του χρήστη μεταξύ διαδοχικών εκκινήσεων. Με την ολοκλήρωση αυτών των μηχανισμών δημιουργείται η απαραίτητη βάση πάνω στην οποία μπορούν να αναπτυχθούν οι λειτουργίες που αφορούν την κύρια χρήση της εφαρμογής, όπως η διαχείριση των χρηστών και η ανταλλαγή μηνυμάτων.

Κεφάλαιο 4ο: Οθόνη Ρυθμίσεων

4.1 Εισαγωγή

Στο παρόν κεφάλαιο παρουσιάζεται η υλοποίηση της οθόνης ρυθμίσεων της εφαρμογής, μέσω της οποίας ο χρήστης έχει τη δυνατότητα να διαχειρίζεται και να τροποποιεί βασικές πληροφορίες που σχετίζονται με το προφίλ του. Η συγκεκριμένη λειτουργικότητα αποτελεί σημαντικό μέρος της εφαρμογής, καθώς παρέχει στον χρήστη μεγαλύτερο έλεγχο σχετικά με τα προσωπικά στοιχεία που εμφανίζονται και αποθηκεύονται σε αυτήν. Παράλληλα, η οθόνη ρυθμίσεων συγκεντρώνει λειτουργίες που αφορούν τόσο την ενημέρωση των στοιχείων του λογαριασμού όσο και τη διαχείριση της συνεδρίας του χρήστη.

Αρχικά, παρουσιάζεται η διαμόρφωση της οθόνης και η δημιουργία κοινών στοιχείων διεπαφής, όπως ο τίτλος της σελίδας και το περιβάλλον στο οποίο τοποθετούνται τα επιμέρους στοιχεία. Για τη διατήρηση της ομοιομορφίας μεταξύ των διαφορετικών οθονών της εφαρμογής αξιοποιούνται επαναχρησιμοποιήσιμα components, ενώ τα πεδία εισαγωγής βασίζονται σε στοιχεία που έχουν ήδη χρησιμοποιηθεί στις οθόνες εγγραφής και σύνδεσης. Παράλληλα, εφαρμόζεται μηχανισμός επικύρωσης των δεδομένων που εισάγει ο χρήστης, ώστε οι τροποποιήσεις να πραγματοποιούνται σύμφωνα με τους κανόνες που έχουν καθοριστεί για κάθε πεδίο.

Στη συνέχεια, αναλύεται ο τρόπος με τον οποίο τα ήδη αποθηκευμένα στοιχεία του χρήστη ανακτώνται από το global state της εφαρμογής και χρησιμοποιούνται για την αρχική συμπλήρωση των πεδίων της οθόνης. Ο χρήστης μπορεί να τροποποιήσει στοιχεία όπως το όνομα, τη διεύθυνση ηλεκτρονικού ταχυδρομείου και τις πληροφορίες που αφορούν το προφίλ του. Οι αλλαγές ελέγχονται πριν αποθηκευτούν και η δυνατότητα αποθήκευσης ενεργοποιείται μόνο όταν έχει πραγματοποιηθεί κάποια ουσιαστική μεταβολή στα υπάρχοντα δεδομένα. Μετά την επιτυχή ενημέρωση της βάσης δεδομένων, το global state ενημερώνεται επίσης, ώστε τα νέα δεδομένα να είναι άμεσα διαθέσιμα σε ολόκληρη την εφαρμογή.

Ένα επιπλέον χαρακτηριστικό που παρουσιάζεται στο κεφάλαιο αφορά τη δυνατότητα αλλαγής της φωτογραφίας προφίλ. Για την επιλογή της εικόνας από τη συσκευή αξιοποιείται η βιβλιοθήκη Expo Image Picker, μέσω της οποίας πραγματοποιείται ο απαραίτητος έλεγχος δικαιωμάτων και παρέχεται πρόσβαση στη συλλογή φωτογραφιών της συσκευής. Η επιλεγμένη εικόνα αρχικά χρησιμοποιείται τοπικά και στη συνέχεια μεταφορτώνεται στο Firebase Storage, ώστε να είναι διαθέσιμη και μετά την επανεκκίνηση της εφαρμογής.

Παράλληλα, παρουσιάζεται η διαδικασία μετατροπής και μεταφόρτωσης της εικόνας σε κατάλληλη μορφή, καθώς και η δημιουργία μοναδικού αναγνωριστικού για κάθε αρχείο εικόνας. Μετά την επιτυχή μεταφόρτωση, το URL της εικόνας αποθηκεύεται στα δεδομένα του αντίστοιχου χρήστη στη βάση δεδομένων. Με τον τρόπο αυτό η φωτογραφία συνδέεται μόνιμα με το προφίλ του χρήστη και μπορεί να ανακτηθεί οποιαδήποτε στιγμή. Επιπλέον, το URL της νέας εικόνας μεταφέρεται και στο global state της εφαρμογής, ώστε η αλλαγή να αντικατοπτρίζεται άμεσα και στις υπόλοιπες οθόνες.

Τέλος, εξετάζεται η λειτουργία αποσύνδεσης του χρήστη, κατά την οποία τα δεδομένα της συνεδρίας αφαιρούνται από τον τοπικό αποθηκευτικό χώρο και η κατάσταση της εφαρμογής επανέρχεται στην αρχική της μορφή. Παράλληλα, προστίθενται μηχανισμοί ενημέρωσης του χρήστη για την ολοκλήρωση ενεργειών, όπως η επιτυχής αποθήκευση των αλλαγών ή η αναμονή κατά τη μεταφόρτωση μιας εικόνας.

Το κεφάλαιο επικεντρώνεται στην ολοκλήρωση των λειτουργιών διαχείρισης του προφίλ, συνδυάζοντας στοιχεία της διεπαφής χρήστη με τους μηχανισμούς αποθήκευσης και διαχείρισης δεδομένων που παρουσιάστηκαν στο προηγούμενο κεφάλαιο. Η συνεργασία μεταξύ React Native, Redux και των υπηρεσιών του Firebase επιτρέπει την άμεση ενημέρωση και συνεπή διαχείριση των στοιχείων του χρήστη σε ολόκληρη την εφαρμογή.

4.2 Τίτλος οθόνης και λοιπές λειτουργίες

Η κάθε οθόνη στην εφαρμογή δεν θα χρησιμοποιηθεί τον τίτλο που προσφέρει το header bar αλλά έναν custom. Θα είναι ένα custom component με ονομασία PageTitle.

```
import { StyleSheet, Text, View } from 'react-native';
import colors from "../constants/colors";

export default function PageTitle (props) {
  return <View style={styles.container}>
    <Text style={styles.text}>{props.text}</Text>
  </View>
};

const styles = StyleSheet.create({
  container: {
    marginBottom: 10
  },
  text: {
    fontSize: 28,
    color: colors.onyx,
    fontFamily: 'bold',
    letterSpacing: 0.3
  }
});
```

Στο Settings Screen θα γίνει χρήση του PageTitle και θα περαστεί ένα text property με όνομα 'Settings'. Για να γίνει ομορφότερη η σελίδα και να υπάρχει συνοχή σε όλες τις οθόνες θα χρησιμοποιηθεί το PageContainer σαν πρώτο στοιχείο που δίνει λευκό φόντο και περιθώρια στα άκρα.

Θα υπάρχουν πεδία στα οποία ο χρήστης θα μπορεί να ενημερώσει το όνομά του, την διεύθυνση ηλεκτρονικού ταχυδρομείου και πληροφορίες σχετικές με αυτόν. Καθένα από αυτά θα είναι ένα Input component που έχει χρησιμοποιηθεί και προηγουμένως κατά την ανάπτυξη.

Η επικύρωση θα γίνεται όπως και στις φόρμες εγγραφής / σύνδεσης όπου τον χειρισμό αναλαμβάνει η inputChangedHandler. Καλείται η validateInput και μετά γίνεται dispatch ένα action. Η dispatchFormState στέλνει την ενέργεια στον reducer που κάνει τον έλεγχο εάν η φόρμα είναι έγκυρη και επιστρέφει το νέο state αυτής. Φυσικά είναι απαραίτητη και η αρχική κατάσταση των πεδίων initialState που προστίθεται και ένα ακόμη στοιχείο με όνομα about. Επειδή προστέθηκε το about θα πρέπει να γίνει το ίδιο και για την επικύρωσή του στο αρχείο formActions στην μέθοδο validateInput. Στην περίπτωση λοιπόν που πρόκειται για το about πεδίο θα το χειρίζεται η validateLength καθώς για αυτό το πεδίο δεν ενδιαφέρει τι θα εισάγει ο χρήστης παρά μόνο το μήκος του κειμένου να είναι στα επιτρεπτά όρια. Θα ορίζεται κατά την κλήση της εάν επιτρέπεται να είναι άδειο πεδίο (allowEmpty), θα είναι ένα προαιρετικό πεδίο δηλαδή θα είναι αποδεκτό να υπάρχει κενό κείμενο. Σαν παράμετροι θα υπάρχουν το ελάχιστο και μέγιστο οι οποίοι θα λαμβάνονται υπόψη μόνο στην περίπτωση που δεν επιτρέπεται το πεδίο να είναι κενό, δηλαδή θα πρέπει να υπάρχει κείμενο.

Απόσπασμα από validateLength του πεδίου about:

```
if (!allowEmpty || value !== '') {
  constraints.length = {};

  if (minLength !== null) constraints.length.minimum = minLength;

  if (maxLength !== null) constraints.length.maximum = maxLength;
}
```

Καθώς και η κλήση της κατά τον έλεγχο της validateInput:

```
else if (inputId === 'about')
  return validateLength(inputId, inputValue, 0, 150, true);
```

Συνεπώς, κατά την χρήση το μήνυμα λάθους θα εμφανιστεί μόνο στην περίπτωση που εισαχθούν περισσότεροι από 150 χαρακτήρες.

Στο σημείο αυτό έχουμε ήδη τα δεδομένα χρήστη αποθηκευμένα στο state της εφαρμογής. Στην μέθοδο signIn (authActions) για παράδειγμα γίνεται η ανάκτηση των δεδομένων από το firebase και ακριβώς μετά το dispatch ενός action στην authenticate που θέτει την αρχική κατάσταση με τα δεδομένα που έχουν αποσταλεί. Το state έχει το userData property που περιέχει όλα τα δεδομένα για τον χρήστη που έχει συνδεθεί. Σε αυτά πλέον υπάρχει πρόσβαση από το useSelector που παρέχει η React Redux, με άλλα λόγια το useSelector δίνει πρόσβαση στο state της εφαρμογής.

```
const userData = useSelector(state => state.auth.userData);
```

Όταν γίνεται η μετάβαση στην οθόνη με τα settings τα πεδία θα είναι ήδη συμπληρωμένα με τα στοιχεία του χρήστη που έχει εισάγει κατά την εγγραφή του και όπως ειπώθηκε βρίσκονται αποθηκευμένα στο state. Σε κάθε Input θα δοθεί ένα property initialValue που δέχεται κάθε πεδίο χωριστά με το αντίστοιχο στοιχείο και ο χειρισμός του γίνεται στο Input component. Εκεί δηλώνεται μία state μεταβλητή με αρχική τιμή props.initialValue που θα αλλάζει το TextInput και θα ενημερώνεται από την setValue με το περιεχόμενο που υπάρχει σε κάθε πεδίο.

Θα υπάρχει ένα κουμπί το οποίο όταν πατήσει ο χρήστης θα αποθηκεύει τις αλλαγές που έχουν γίνει. Στην αρχική κατάσταση θα είναι απενεργοποιημένο (formIsValid: false). Καθώς πληκτρολογεί νέα στοιχεία και γίνεται ο έλεγχος ότι αυτά είναι αποδεκτά και ακολουθούν τους κανόνες που έχουν οριστεί στο validation, θα ενεργοποιείται ή θα απενεργοποιείται ανάλογα με την περίπτωση.

Η saveHandler θα χειρίζεται την αποθήκευση των δεδομένων στην βάση και θα εκτελείται μόλις ο χρήστης πατήσει την επιλογή 'Save'. Σε αυτήν γίνεται η χρήση της μεθόδου updateSignedInUserData (αρχείο authActions) που ενημερώνει την αντίστοιχη θέση στην βάση δεδομένων βάσει του userId που περνάει σαν παράμετρος και του newData που περιέχει το δεδομένα που είναι προς ενημέρωση.

```
const dbRef = ref(getDatabase());
const childRef = child(dbRef, `users/${userId}`);
await update(childRef, newData);
```

Η χρήση της στην saveHandler:

```
setIsLoading(true);
await updateSignedInUserData(userData.userId, updatedValues);
```

Θα προστεθεί ένα κουμπί αποσύνδεσης από την εφαρμογή 'Logout' το οποίο όταν το πατήσει ο χρήστης θα γίνεται έξοδος. Στο οπτικό κομμάτι θα μεταφέρεται στις αρχικές οθόνες σύνδεσης, ωστόσο στο πίσω μέρος θα γίνεται ενέργεια στο state καθώς με το `dispatch` θα στέλνεται μία ενέργεια (`action`) για την μεταβολή αυτού του state. Αυτή η ενέργεια θα καθαρίζει τα στοιχεία του χρήστη.

Στο αρχείο `authActions` υπάρχει η `userLogout` μέθοδος που σβήνει τα στοιχεία του Async Storage και κάνει `dispatch` ένα `action` στο state καλώντας την `logout reducer` που με την σειρά της επαναφέρει τα δεδομένα στην αρχική τους κατάσταση δηλαδή πριν ο χρήστης συνδεθεί στην εφαρμογή.

Επειδή καλείται ένα `action` για την ενημέρωση του state θα πρέπει να χρησιμοποιηθεί το `dispatch` και πάλι. Η `useDispatch` δίνει μία μέθοδο που επιτρέπει να γίνεται `dispatch` πολλών `actions` προκειμένου να γίνει αυτή η ενημέρωση.

Δημιουργία του κουμπιού αποσύνδεσης:

```
<SubmitButton
  title='Logout'
  onPress={() => dispatch(userLogout(userData))}
  style={{ marginTop: 20 }}
  color={colors.red}
/>
```

Μέχρι τώρα έχει γίνει η ενημέρωση των δεδομένων και αυτές οι αλλαγές φαίνονται στην βάση, αυτό που δεν έχει γίνει όμως είναι η ενημέρωση του state καθώς όπως έγινε και προηγουμένως υπάρχει ένα `global state` για ολόκληρη την εφαρμογή που καταγράφει τα δεδομένα του χρήστη με χρήση της `react redux`. Αυτό πρέπει να ενημερωθεί με τα καινούργια δεδομένα.

Στο `authSlice` δημιουργείται ένα νέο `action` με όνομα `updateLoggedInUserData`. Αυτή η μέθοδος θα παίρνει τα υπάρχοντα δεδομένα και θα τα συγχωνεύει με τα νέα δεδομένα που έρχονται με την κλήση αυτής της μεθόδου.

```
updateLoggedInUserData: (state, action) => {
  state.userData = { ...state.userData, ...action.payload.newData }
```

Η χρήση της γίνεται στην μέθοδο `saveHandler` αμέσως μετά την ενημέρωση της βάσης δεδομένων.

```
dispatch(updateLoggedInUserData({ newData: updatedValues }));
```

Όταν τελειώσει η διαδικασία ενημέρωσης της βάσης και αυτής του state με τα νέα δεδομένα θα εμφανίζεται στην οθόνη μήνυμα ότι όλα έγιναν επιτυχώς. Ο χειρισμός του θα γίνεται από την state μεταβλητή `showSuccessMessage` και η μεταβολή αυτής από την μέθοδο `setShowSuccessMessage`.

```
const [showSuccessMessage, setShowSuccessMessage] = useState(false);
```

Θα εμφανίζεται σαν ένα απλό κείμενο 'Saved!' και θα εξαφανίζεται από την οθόνη μετά από τρία δευτερόλεπτα.

```
{
  showSuccessMessage && <Text style={{ color: colors.red }}>Saved!</Text>
}

setTimeout(() => setShowSuccessMessage(false), 3000);
```

Το κουμπί αποθήκευσης των δεδομένων 'Save' θα εμφανίζεται μόνο όταν γίνονται αλλαγές σε κάποιο από τα τέσσερα πεδία. Αν δεν έχει γίνει κάποια αλλαγή θα είναι κρυφό. Για να γίνει αυτό θα συγκρίνονται οι τιμές των ενημερωμένων πεδίων με αυτές του αρχικού state, δηλαδή με τις τιμές που έχουν τα πεδία όταν ο χρήστης συνδεθεί στην εφαρμογή. Η μέθοδος `hasChanges` θα κάνει τον έλεγχο και θα επιστρέφει αληθές ή ψευδές και βάσει αυτού θα γίνεται παρακάτω η απεικόνιση του `SubmitButton` 'Save'.

```
const hasChanges = function () {
  const currentValues = formState.inputValues;

  return currentValues.firstName !== firstName ||
    currentValues.lastName !== lastName ||
    currentValues.email !== email ||
    currentValues.about !== about;
};
```

Η χρήση της μεθόδου στο κουμπί αποθήκευσης:

```
hasChanges() && <SubmitButton
  title='Save'
  onPress={saveHandler}
  style={{ marginTop: 20 }}
  disabled={!formState.formIsValid}
/>
```

4.3 Ενημέρωση Φωτογραφίας Προφίλ

Στην ίδια οθόνη θα προστεθεί το χαρακτηριστικό ώστε ο χρήστης να κάνει αλλαγή αν το επιθυμεί και να ενημερώνει την φωτογραφία προφίλ του. Η υλοποίησή του θα γίνει με την δημιουργία ενός `ProfileImage` component που όταν το πατήσει ο χρήστης θα ανοίγει τις φωτογραφίες της συσκευής του και θα επιλέγει την φωτογραφία που θέλει να ενημερώσει.

Με την προσθήκη και αυτού του στοιχείου στην οθόνη settings θα υπάρχουν αρκετά στοιχεία που θα απεικονίζονται στην κινητή συσκευή. Το πρόβλημα είναι πως σε συσκευές με μικρότερη οθόνη ενδέχεται να μην φαίνεται όλο το περιεχόμενο. Το `ScrollView` που προστίθεται στο `SettingsScreen`, και περικλείει μέσα του όλα τα στοιχεία αυτής της οθόνης, παρέχει την δυνατότητα σε πιο μικρές οθόνες όταν δεν φτάνει το εύρος τους να εμφανίζουν μία μπάρα στα δεξιά με την οποία ο χρήστης μπορεί να κάνει το λεγόμενο 'scrolling' και να δει τα υπόλοιπα μέρη. Το στυλ που θα οριστεί σε αυτό θα δίνεται από το `contentContainerStyle` που εφαρμόζεται σε όλο το περιεχόμενο (`children`) που βρίσκεται μέσα στο `ScrollView`.

Κάθε φορά που θέλουμε να γίνει απεικόνιση της εικόνας χρήστη θα χρησιμοποιείται το `ProfileImage` component. Αφού οριστεί το στοιχείο `Image` που θα εμφανίζει την εικόνα, η διαμόρφωση του στυλ του θα γίνεται με τέτοιο τρόπο ώστε το ύψος και μήκος της να είναι διαφορετικό ανάλογα με την οθόνη που βρίσκεται ο χρήστης. Έτσι, το στυλ της εικόνας θα ορίζεται από τις ιδιότητες που θα οριστούν γενικά

για αυτό συν αυτές που θα περνάνε από την χρήση του ProfileImage το οποίο θα δέχεται το props.size για το ύψος και μήκος αντίστοιχα.

```
<Image
  style={{ ... styles.image, ...{ width: props.size, height: props.size } }}
  source={image}
/>
```

Θα προστεθεί ένα εικονίδιο δίπλα στην εικόνα που θα δείχνει στον χρήστη πως μπορεί να κάνει κλικ σε αυτό και όταν το πατήσει θα ανοίγουν οι φωτογραφίες της συσκευής.

Συνήθως τα στοιχεία που απεικονίζονται στην οθόνη δημιουργούνται στο στοιχείο View. Ωστόσο, το συγκεκριμένο δεν δίνει την δυνατότητα στον χρήστη να μπορεί να ‘πατήσει’ σε αυτό και να εκτελεστεί μία ενέργεια. Για αυτό η React Native προσφέρει το στοιχείο TouchableOpacity που όταν τοποθετηθεί φαίνεται ότι είναι εκτελέσιμο για να γνωρίζει ο χρήστης ότι μπορεί να κάνει μία ενέργεια. Αυτό έρχεται με ένα property onPress που εκτελεί μία μέθοδο. Η μέθοδος που θα περαστεί είναι η pickImage και θα χειρίζεται το άνοιγμα των φωτογραφιών.

Η βιβλιοθήκη Expo Image Picker ανοίγει την συλλογή φωτογραφιών για αυτή την συσκευή.

Όπως πάντα η πρώτη ενέργεια που γίνεται είναι η εγκατάσταση της βιβλιοθήκης που θα εγκαταστήσει τα ανάλογα πακέτα:

npm expo install expo-image-picker

Σε πολλές εφαρμογές και λειτουργίες του κινητού πριν γίνει η εκτέλεσή τους ζητείται από την χρήστη να δώσει άδεια. Η άδεια αφορά στο τι θα είναι επιτρεπτό να γίνει όπως η χρήση της κάμερας, του μικροφώνου, τοποθεσίας ή στην συγκεκριμένη περίπτωση το άνοιγμα της συλλογής φωτογραφιών. Όλη η λειτουργικότητα θα γίνει σε ένα βοηθητικό αρχείο imagePickerHelper (φάκελος utils). Οι μέθοδοι που θα περιέχει θα μπορούν να χρησιμοποιηθούν και σε άλλα αρχεία.

Η launchImagePicker από όποιο αρχείο και να κληθεί θα ανοίγει την συλλογή. Το πρώτο βήμα είναι να δοθεί εξουσιοδότηση από τον χρήστη, το χειρίζεται η μέθοδος checkMediaPermissions. Εάν πράγματι εκτελείται από κινητή συσκευή (δεν είναι web application) τότε η μέθοδος requestMediaLibraryPermissionsAsync της ImagePicker κάνει τον έλεγχο αν έχει εξουσιοδότηση. Στην περίπτωση που δεν έχει άδεια ρωτάει τον χρήστη αν επιθυμεί να την δώσει. Η μέθοδος checkMediaPermissions είναι ασύγχρονη διαδικασία διότι χειρίζεται ένα Promise. Στην JavaScript ένα Promise μπορεί να έχει δύο δυνατές καταστάσεις, είτε να έχει επιλυθεί επιτυχώς (resolved) είτε να έχει απορριφθεί (rejected). Στην συγκεκριμένη περίπτωση εάν επιλυθεί επιτυχώς σημαίνει πως υπάρχει εξουσιοδότηση, εάν απορριφθεί τότε δεν έχει δοθεί.

```
const checkMediaPermissions = async () => {
  if (Platform.OS !== 'web') {
    const permissionResult = await ImagePicker.requestMediaLibraryPermissionsAsync();
    if (permissionResult.granted === false)
      return Promise.reject('We need permission to access your photos');
    // Reject the Promise, its async so returns a Promise
  }

  return Promise.resolve();
};
```

Η checkMediaPermissions θα κληθεί στην μέθοδο launchImagePicker. Εάν έχει δοθεί άδεια θα εκτελεστεί το υπόλοιπο κομμάτι του κώδικα, διαφορετικά θα γίνει έξοδος από την μέθοδο. Η launchImageLibraryAsync της ImagePicker βιβλιοθήκης ανοίγει την συλλογή με τις φωτογραφίες και δέχεται κάποιες επιλογές που έχουν να κάνουν με τον τύπο αρχείων που θα εμφανίζονται όπως εικόνες,

βίντεο. Ενδιαφερόμαστε αποκλειστικά για φωτογραφίες οπότε θα αγνοήσει αρχεία τύπου βίντεο. Υπάρχει επιλογή ώστε ο χρήστης να μπορεί να ‘κόψει την εικόνα’ πριν την επιλέξει. Η μέθοδος που ανοίγει την συλλογή καθώς και τα options:

```
const result = await ImagePicker.launchImageLibraryAsync({
  // mediaTypes: ImagePicker.MediaTypeOptions.Images, (Deprecated)
  mediaTypes: ['images'],
  allowsEditing: true, // allow the user to crop the image before they press submit
  aspect: [1, 1],
  quality: 1
});
```

Μπορεί να επιστρέψει ένα προσωρινό uri (Uniform Resource Identifier) που είναι ουσιαστικά η θέση της εικόνας στην συσκευή. Αυτό μπορεί να χρησιμοποιηθεί για να ενημερωθεί η εικόνα του χρήστη και να εμφανιστεί στην οθόνη.

Το παρακάτω επιστρέφεται από την κλήση της μεθόδου launchImagePicker:

```
if (!result.canceled) {
  return result.assets[0].uri;
}
```

Στο component ProfileImage, η μέθοδος launchImagePicker θα κληθεί από την pickImage όπου και θα δώσει την διαδρομή στην εικόνα. Επόμενο βήμα είναι να οριστεί η εικόνα και θα γίνει με την state μεταβλητή image της οποίας η τιμή θα αλλάζει σύμφωνα με την setImage.

```
const [image, setImage] = useState(source);
```

Όταν ο χρήστης επιλέξει εικόνα και αυτή ανέβει στην βάση δεδομένων θα περνάει ένα uri property στο ProfileImage component που θα περιέχει την επιλεγμένη εικόνα. Η παρακάτω γραμμή κάνει τον έλεγχο αν υπάρχει αυτό το uri property τότε θα χρησιμοποιηθεί αυτό διαφορετικά μία γενική εικόνα χρήστη.

```
const source = props.uri ? { uri: props.uri } : userImage;
```

Σταδιακά όταν η εικόνα αποθηκευτεί θα γίνει χρήση του απομακρυσμένου uri.

Προς το παρόν η λειτουργία της εικόνας δουλεύει, ωστόσο όταν γίνει ανανέωση της εφαρμογής αυτή θα χαθεί και αυτό διότι δεν αποθηκεύεται κάπου. Το Firebase εκτός του Realtime Database και Authentication παρέχει μία ακόμη επιλογή το Storage που δίνει την δυνατότητα αποθήκευσης εικόνων, αρχείων ακόμη και βίντεο.

Η αποθήκευση της εικόνας θα γίνει από την μέθοδο uploadImageAsync (imagePickerHelper αρχείο) όπου πραγματοποιείται ένα αίτημα XMLHttpRequest. Το blob αναφέρεται ως Binary Large Object και είναι ένας τρόπος αποθήκευσης μεγάλων αρχείων όπως εικόνων, βίντεο. Θα είναι ένα αίτημα web (web request) με ασύγχρονο τρόπο (Promise) και θα έχει δύο καταστάσεις, είτε θα εκπληρωθεί επιτυχώς (resolved) είτε θα απορριφθεί (rejected). Γίνεται η διαμόρφωση ορισμένων παραμέτρων που είναι απαραίτητες για να σταλεί το αίτημα και στο τέλος αποστέλλεται.

Για το παρακάτω απόσπασμα κώδικα έγινε χρήση παραδείγματος από την σελίδα:

<https://github.com/expo/examples/blob/master/with-firebase-storage-upload/App.js>

```
const blob = await new Promise((resolve, reject) => {
  const xhr = new XMLHttpRequest();
  xhr.onload = function () {
    resolve(xhr.response);
  };

  xhr.onerror = function (e) {
    console.log(e);
    reject(new TypeError('Network request failed'));
  };

  xhr.responseType = 'blob';
  xhr.open('GET', uri, true);
  xhr.send();
});
```

Το παραπάνω αίτημα δίνει πρόσβαση στην εικόνα που βρίσκεται τοπικά και την επιστρέφει σε μορφή 'blob' μετατρέποντάς την στην μορφή που χρειάζεται για να είναι έτοιμη να ανέβει στο Storage του Firebase καθώς αυτό δέχεται αποκλειστικά αυτού του είδους τα αρχεία.

Στο Storage θα υπάρχει ένας φάκελος με όνομα ProfilePics και εκεί θα αποθηκεύονται όλες οι εικόνες. Επειδή όλοι οι χρήστες μπορούν να ανεβάσουν τις δικές τους εικόνες θα πρέπει τα ονόματα που θα χρησιμοποιηθούν εκεί να είναι μοναδικά, δηλαδή να δοθεί ένα ξεχωριστό αναγνωριστικό για κάθε εικόνα. Αυτό θα γίνει με την χρήση εξωτερικής βιβλιοθήκης uuid (universal unique identifier). Όταν χρησιμοποιηθεί θα δώσει ένα id που είναι μοναδικό, κανένας ποτέ δεν θα μπορέσει να δημιουργήσει όμοιό του.

Η εγκατάσταση της βιβλιοθήκης γίνεται με την εντολή: **npm i react-native-uuid**

Η εισαγωγή του γίνεται στο imagePickerHelper με τον εξής τρόπο:

```
import uuid from 'react-native-uuid';
```

Στην συνέχεια της μεθόδου θα χρειαστεί αναφορά στο Storage καθώς και στον φάκελο profilePics ο οποίος δεν υπάρχει ακόμη αλλά θα δημιουργηθεί μόλις ανέβει η εικόνα. Πρόκειται για την τοποθεσία της μέσα στην βάση δεδομένων στον φάκελο που μόλις δημιουργήθηκε με το αντίστοιχο αναγνωριστικό (uuid) που δίνεται αυτόματα. Η uploadBytesResumable θα τοποθετήσει την εικόνα σε μορφή blob στην αντίστοιχη τοποθεσία που έχει δοθεί. Τέλος, η μέθοδος θα επιστρέψει την τοποθεσία (url) της εικόνας στο storage του Firebase.

```
const storageRef = ref(getStorage(app), `${pathFolder}/${uuid.v4()}`);
// Will give us the location we want to upload this image to

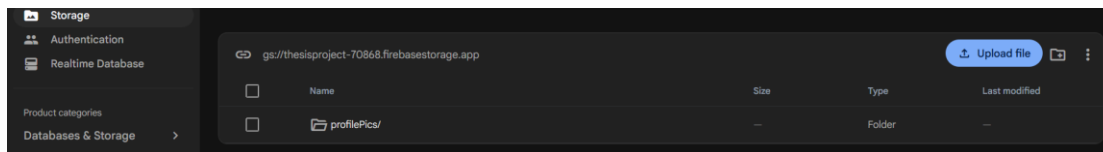
await uploadBytesResumable(storageRef, blob);

blob.close();

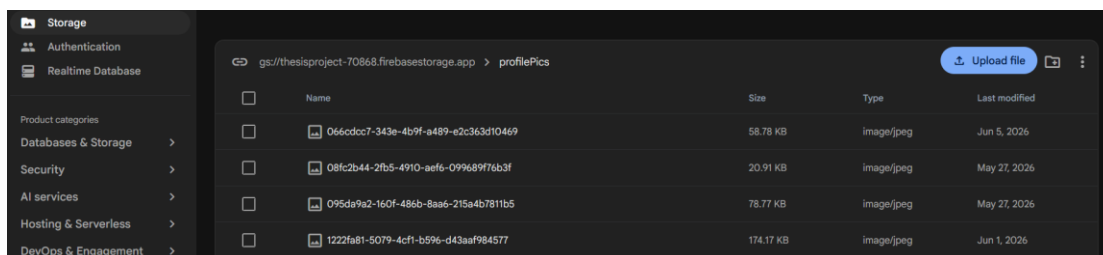
// Return the url of the uploaded file
return await getDownloadURL(storageRef);
```

Μπορεί να γίνει χρήση της στο ProfileImage component και με το url της εικόνας στην βάση που επιστρέφει η μέθοδος μπορεί να τεθεί η νέα εικόνα που επέλεξε ο χρήστης. Πλέον όταν επιλεγθεί εικόνα

από την συλλογή της συσκευής αυτή θα βρίσκεται στο storage σε φάκελο ProfilePics όπου κάθε μία θα έχει το δικό της μοναδικό όνομα που έχει δοθεί από το uuid.



Εικόνα 4.1 - Storage στον φάκελο profilePics του Firebase



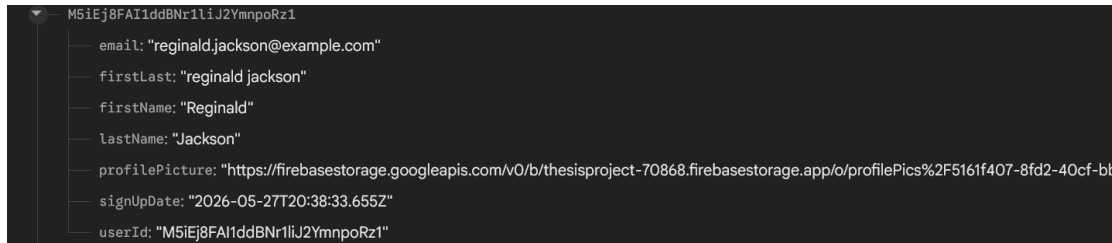
Εικόνα 4.2 - Αποθηκευμένες εικόνες του φακέλου profilePics του Storage

Αν και εικόνα βρίσκεται στον server στην πραγματικότητα θα πρέπει να αποθηκευτεί επίσης στα δεδομένα του χρήστη που βρίσκονται στην βάση. Από προηγούμενες ενέργειες υπάρχει η μέθοδος `updateSignedInUserData ( authActions )` που ενημερώνει τα αρχεία του χρήστη στην βάση με οποιαδήποτε δεδομένα ορίσουμε. Το ίδιο μπορεί να γίνει και με την νέα φωτογραφία προφίλ.

```
const newData = { profilePicture: uploadUrl };
await updateSignedInUserData(userId, newData);
```

Στον κόμβο λοιπόν `users` ανάμεσα στα πεδία `firstName`, `email` κλπ θα υπάρχει ένα νέο όνομα `profilePicture`.

Στιγμιότυπο με τα δεδομένα χρήστη:



Εικόνα 4.3 - Δεδομένα χρήστη στον κόμβο users με το μοναδικό id

Μπορούμε να πάρουμε το νέο ProfilePicture Url και να το περάσουμε σαν prop uri στο ProfileImage component. Αυτό θα είναι το νέο source που θα αξιοποιεί το Image στοιχείο.

Ακόμη και μετά από ανανέωση της εφαρμογής ο χρήστης θα βλέπει την ίδια εικόνα γιατί πλέον ανακτάται από τα δεδομένα του στην βάση καθώς δεν είναι κάτι που έχει οριστεί τοπικά.

Μπορεί τα δεδομένα να βρίσκονται στην βάση αλλά στην πραγματικότητα έχουμε φροντίσει να αποθηκεύονται τοπικά στο state της εφαρμογής. Έχει αναφερθεί ήδη πως υπάρχει ένα καθολικό state (global state), που το διαχειρίζεται η βιβλιοθήκη React Redux, και στις σελίδες που υλοποιούμε μπορούμε να έχουμε πρόσβαση στα δεδομένα του χρήστη χωρίς να χρειάζεται να στέλνουμε αιτήματα προς το firebase. Αυτή η μέθοδος είναι ταχύτερη και προσβάσιμη οποιαδήποτε στιγμή.

Τα state δεδομένα λοιπόν του χρήστη δεν είναι ακόμη ενημερωμένα με το νέο πεδίο που προστέθηκε, θα έχουν την παλαιότερη εικόνα πριν γίνει η αλλαγή. Αυτό σημαίνει πως σε διαφορετική οθόνη για παράδειγμα θα εμφανίζεται η προηγούμενη. Συνεπώς, θα πρέπει να ενημερωθεί το συνολικό state.

Η μέθοδος updateLoggedInUserData (authSlice) δέχεται τα νέα δεδομένα και ενημερώνει το αρχικό state με τα δεδομένα του χρήστη (userData). Το state ενημερώνεται με την μέθοδο dispatch της Redux αποστέλλοντας ένα action. Το νέο στοιχείο θα είναι το profilePicture που περνάει σαν παράμετρος newData.

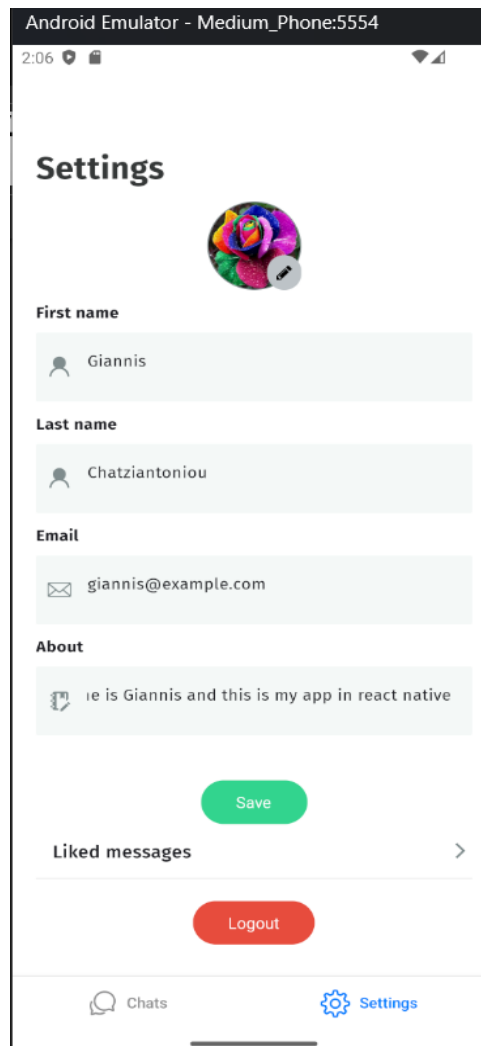
```
dispatch(updateLoggedInUserData({ newData })); // Same as newData: newData
```

Σαν ένα τελευταίο χαρακτηριστικό που θα προστεθεί είναι ένδειξη πως γίνεται η μεταφόρτωση της εικόνας στην βάση. Κατά την διάρκεια θα εμφανίζεται ένα loading indicator και μόλις τελειώσει η διαδικασία θα εμφανίζεται η εικόνα.

```

{
  isLoading ?
  <View height={props.size} width={props.size} style={styles.loadingContainer}>
    <ActivityIndicator size={'small'} color={colors.primary} />
  </View>
  :
  <Image
    style={{ ...styles.image, ...{ width: props.size, height: props.size } }}
    source={image}
  />
}

```



Εικόνα 4.4 - Οθόνη ρυθμίσεων του συνδεδεμένου χρήστη

4.4 Επίλογος

Στο παρόν κεφάλαιο παρουσιάστηκε η υλοποίηση της οθόνης ρυθμίσεων και των βασικών λειτουργιών που σχετίζονται με τη διαχείριση του προφίλ του χρήστη. Η συγκεκριμένη οθόνη συγκεντρώνει σε ένα ενιαίο περιβάλλον τις απαραίτητες δυνατότητες για την τροποποίηση των προσωπικών στοιχείων, την ενημέρωση της φωτογραφίας προφίλ και την αποσύνδεση από την εφαρμογή. Με αυτόν τον τρόπο, ο χρήστης μπορεί να διαχειρίζεται τα στοιχεία του χωρίς να απαιτείται πρόσβαση σε διαφορετικά σημεία της εφαρμογής.

Αρχικά, υλοποιήθηκε η δυνατότητα εμφάνισης και τροποποίησης των βασικών δεδομένων του χρήστη. Τα δεδομένα που έχουν ήδη ανακτηθεί κατά τη διαδικασία σύνδεσης είναι διαθέσιμα μέσω του global state της Redux και χρησιμοποιούνται για την αρχική συμπλήρωση των αντίστοιχων πεδίων. Με την αξιοποίηση των μηχανισμών επικύρωσης που είχαν αναπτυχθεί προηγουμένως, πραγματοποιείται έλεγχος των νέων τιμών πριν αυτές αποθηκευτούν. Ιδιαίτερη περίπτωση αποτελεί το πεδίο πληροφοριών του χρήστη, για το οποίο εφαρμόζεται έλεγχος μήκους και επιτρέπεται να παραμείνει κενό, καθώς πρόκειται για προαιρετική πληροφορία.

Παράλληλα, υλοποιήθηκε μηχανισμός ελέγχου των αλλαγών που πραγματοποιεί ο χρήστης. Η χρήση της μεθόδου ελέγχει τις τροποποιήσεις και επιτρέπει την εμφάνιση του κουμπιού αποθήκευσης μόνο όταν υπάρχει πραγματική διαφορά μεταξύ των αρχικών και των νέων δεδομένων. Μετά την επιτυχή αποθήκευση, πραγματοποιείται όχι μόνο η ενημέρωση των δεδομένων στη βάση Firebase, αλλά και η

αντίστοιχη ενημέρωση του global state. Με αυτόν τον τρόπο αποφεύγεται η εμφάνιση προηγούμενων δεδομένων σε άλλες οθόνες της εφαρμογής και διατηρείται η συνέπεια μεταξύ της βάσης δεδομένων και της τρέχουσας κατάστασης της εφαρμογής. Επιπλέον, παρέχεται κατάλληλη ένδειξη προς τον χρήστη ώστε να γνωρίζει ότι η διαδικασία αποθήκευσης ολοκληρώθηκε επιτυχώς.

Σημαντικό μέρος της υλοποίησης αποτέλεσε και η προσθήκη της δυνατότητας αλλαγής της φωτογραφίας προφίλ. Μέσω του Expo Image Picker ο χρήστης μπορεί να επιλέξει μία εικόνα από τη συλλογή της συσκευής του, αφού προηγουμένως δοθεί η απαραίτητη άδεια πρόσβασης. Η διαδικασία υλοποιήθηκε με τρόπο ώστε να μπορεί να επαναχρησιμοποιηθεί και σε άλλα σημεία της εφαρμογής, μέσω βοηθητικών συναρτήσεων που συγκεντρώνουν τη σχετική λειτουργικότητα.

Η επιλεγμένη εικόνα μετατρέπεται στην κατάλληλη μορφή και μεταφορτώνεται στο Firebase Storage. Για την αποφυγή προβλημάτων που θα μπορούσαν να προκύψουν από την ύπαρξη αρχείων με ίδιο όνομα χρησιμοποιείται μοναδικό αναγνωριστικό UUID για κάθε εικόνα. Μετά την ολοκλήρωση της μεταφόρτωσης, το URL της εικόνας αποθηκεύεται στα δεδομένα του χρήστη στη Realtime Database, ενώ παράλληλα ενημερώνεται το global state. Επομένως, η νέα φωτογραφία δεν αποτελεί πλέον ένα προσωρινό δεδομένο της συσκευής, αλλά συνδέεται με τον λογαριασμό του χρήστη και μπορεί να ανακτηθεί και μετά από επανεκκίνηση της εφαρμογής. Κατά τη διάρκεια της μεταφόρτωσης χρησιμοποιείται επίσης ένδειξη φόρτωσης, ώστε ο χρήστης να ενημερώνεται για την εξέλιξη της διαδικασίας.

Επιπλέον, ολοκληρώθηκε η λειτουργία αποσύνδεσης, η οποία αποτελεί απαραίτητο στοιχείο για τη διαχείριση της συνεδρίας. Με την ενεργοποίησή της διαγράφονται τα σχετικά δεδομένα από το AsyncStorage και αποστέλλεται η αντίστοιχη ενέργεια στη Redux, με αποτέλεσμα την επαναφορά του authentication state στην αρχική του κατάσταση. Παράλληλα, η εφαρμογή μεταφέρει τον χρήστη στις οθόνες που αφορούν τη σύνδεση, ολοκληρώνοντας με αυτόν τον τρόπο τον κύκλο διαχείρισης της συνεδρίας.

Συνολικά, η υλοποίηση που παρουσιάστηκε στο κεφάλαιο επεκτείνει τους μηχανισμούς αυθεντικοποίησης και διαχείρισης κατάστασης που αναλύθηκαν προηγουμένως και τους αξιοποιεί σε ένα ολοκληρωμένο περιβάλλον διαχείρισης προφίλ. Η σύνδεση της διεπαφής της εφαρμογής με το global state της Redux και τις υπηρεσίες αποθήκευσης του Firebase επιτρέπει την άμεση και συνεπή ενημέρωση των δεδομένων του χρήστη. Παράλληλα, η δυνατότητα αλλαγής της φωτογραφίας προφίλ και η υποστήριξη μόνιμης αποθήκευσής της ενισχύουν τη λειτουργικότητα και την πληρότητα της εφαρμογής. Με την ολοκλήρωση των παραπάνω λειτουργιών έχει δημιουργηθεί ένα ολοκληρωμένο σύστημα διαχείρισης του προφίλ, πάνω στο οποίο μπορούν να βασιστούν οι επόμενες λειτουργίες της εφαρμογής, οι οποίες αφορούν την κύρια επικοινωνία και αλληλεπίδραση μεταξύ των χρηστών.

Κεφάλαιο 5ο: Αναζήτηση Χρηστών, Δημιουργία Συνομιλίας

5.1 Εισαγωγή

Στο παρόν κεφάλαιο παρουσιάζεται η διαδικασία υλοποίησης του μηχανισμού δημιουργίας, ανάκτησης και διαχείρισης των συνομιλιών της εφαρμογής. Η συγκεκριμένη λειτουργικότητα αποτελεί βασικό τμήμα μιας εφαρμογής ανταλλαγής μηνυμάτων, καθώς επιτρέπει στον χρήστη να εντοπίζει άλλους εγγεγραμμένους χρήστες, να δημιουργεί νέους διαλόγους και να αποκτά πρόσβαση στις συνομιλίες στις οποίες ήδη συμμετέχει.

Αρχικά αναλύεται η προσθήκη της δυνατότητας δημιουργίας νέας συνομιλίας μέσα από την κεντρική οθόνη εμφάνισης των διαλόγων. Για τον σκοπό αυτό δημιουργείται κατάλληλη επιλογή πλοήγησης, μέσω της οποίας ο χρήστης μεταφέρεται σε ξεχωριστή οθόνη αναζήτησης. Στην οθόνη αυτή υλοποιείται μηχανισμός δυναμικής αναζήτησης εγγεγραμμένων χρηστών, ο οποίος επικοινωνεί με τη βάση δεδομένων και επιστρέφει τα αντίστοιχα αποτελέσματα χωρίς να απαιτείται η χειροκίνητη ενεργοποίηση της αναζήτησης μέσω ξεχωριστού κουμπιού.

Στη συνέχεια παρουσιάζεται η απεικόνιση των αποτελεσμάτων αναζήτησης και η επιλογή ενός χρήστη για την έναρξη νέας συνομιλίας. Παράλληλα, εξετάζεται η αξιοποίηση του κεντρικού state της εφαρμογής για την προσωρινή αποθήκευση των δεδομένων των χρηστών που έχουν ήδη ανακτηθεί από τη βάση. Με αυτό τον τρόπο περιορίζονται τα επαναλαμβανόμενα αιτήματα προς το Firebase και επιτρέπεται η επαναχρησιμοποίηση των δεδομένων σε διαφορετικές οθόνες της εφαρμογής.

Έπειτα αναλύεται η διαδικασία δημιουργίας ενός νέου διαλόγου. Ο διάλογος δεν αποθηκεύεται άμεσα με τη μετάβαση στην οθόνη συνομιλίας, αλλά δημιουργείται όταν αποσταλεί το πρώτο μήνυμα. Με αυτόν τον τρόπο αποφεύγεται η δημιουργία κενών συνομιλιών στη βάση δεδομένων. Κατά τη διαδικασία αυτή δημιουργούνται οι απαραίτητες εγγραφές τόσο για τα δεδομένα της ίδιας της συνομιλίας όσο και για τη συσχέτισή της με τους χρήστες που συμμετέχουν σε αυτή.

Επίσης, γίνεται ανάκτηση των ενεργών συνομιλιών κατά την είσοδο του χρήστη στο κύριο περιβάλλον της εφαρμογής. Τα δεδομένα των συνομιλιών ανακτώνται από το Firebase, αποθηκεύονται στο Redux state και καθίστανται διαθέσιμα στις επιμέρους οθόνες χωρίς να απαιτείται νέα ανάκτηση σε κάθε σύνδεση. Παράλληλα, υλοποιείται η διαχείριση των listeners της βάσης δεδομένων, ώστε η εφαρμογή να ενημερώνεται όταν πραγματοποιούνται μεταβολές και να πραγματοποιείται ο καθαρισμός τους όταν δεν απαιτούνται πλέον.

Τέλος, παρουσιάζεται η δημιουργία της λίστας ενεργών συνομιλιών, η ταξινόμησή τους σύμφωνα με την τελευταία χρονική στιγμή ενημέρωσης και η μετάβαση από τη λίστα στην αντίστοιχη οθόνη διαλόγου. Με τις παραπάνω λειτουργίες ολοκληρώνεται ένας βασικός μηχανισμός διαχείρισης συνομιλιών, ο οποίος συνδέει τη διεπαφή χρήστη με τα αποθηκευμένα δεδομένα και το κεντρικό state της εφαρμογής.

5.2 Δημιουργία Επιλογής

Η οθόνη εμφάνισης όλων των συνομιλιών είναι το αρχείο ChatListScreen όπου και θα προστεθεί το νέο κουμπί. Θα γίνει χρήση του πακέτου React Navigation Header Buttons που προσφέρει έναν εύκολο τρόπο να προσθέσουμε νέα κουμπιά και φροντίζει την διαμόρφωσή τους (styling). Είναι δυνατή η

δημιουργία components χωρίς αυτό το πακέτο, όμως θα πρέπει να φροντίσουμε για όλη την εμφάνισή τους με τον κίνδυνο να μην ταιριάζουν με τα προκαθορισμένα χαρακτηριστικά της κάθε συσκευής (Android / IOS). Το πακέτο θα το κάνει αυτόματα πράγμα που το καθιστά ευκολότερο στην ανάπτυξη της εφαρμογής.

Η εγκατάσταση γίνεται με την εντολή : **npm i react-navigation-header-buttons**

Η δημιουργία του θα γίνει σε ένα useEffect που θα εκτελεστεί μόνο μία φορά. Από την μεταβλητή props και του navigation που έρχεται αυτόματα από το React Native Navigation πακέτο θα τεθούν τα options για αυτό το κουμπί. Προστίθεται το κύριο στοιχείο HeaderButtons που ορίζει ένα component HeaderComponent που με την σειρά του χειρίζεται την εμφάνιση του κουμπιού. Σύμφωνα με το εγχειρίδιο θα πρέπει να περάσουμε ένα custom component που ονομάζεται CustomHeaderButton (φάκελος components).

Το CustomHeaderButton χρησιμοποιεί το HeaderButton στοιχείο του πακέτου όπου δίνοντας ορισμένα options δημιουργεί το κουμπί.

```
const CustomHeaderButton = props => {
  return <HeaderButton
    { ...props }
    IconComponent={Ionicons}
    iconSize={23}
    color={props.color ?? colors.blue}
    // ?? means if props.color is null or empty then use colors.blue instead
  />
};

export default CustomHeaderButton;
```

Αυτό θα ανοίγει νέα οθόνη με όνομα NewChatScreen (screens φάκελος). Θα εμφανίζεται σαν modal που θα έχει κουμπί για κλείσιμο αλλιώς ο χρήστης δεν θα μπορεί να φύγει από την σελίδα. Ο τίτλος θα έχει το όνομα 'New Chat'.

```
useEffect(() => {
  props.navigation.setOptions({
    headerLeft: () => {
      return <HeaderButtons HeaderButtonComponent={CustomHeaderButton}>
        <Item
          title='Close'
          onPress={() => props.navigation.goBack()}
          buttonStyle={{
            backgroundColor: colors.blueGray,
            color: colors.alabasterWhite,
            paddingHorizontal: 20,
            paddingVertical: 9,
            borderRadius: 20
          }}
        />
      </HeaderButtons>
    },
  });
}, [props.navigation]);
```

Για να γίνει χρήση της οθόνης NewChatScreen θα πρέπει να προστεθεί στο MainNavigator της εφαρμογής. Γίνεται διαχωρισμός των οθονών σε τμήματα (groups) και είναι κάτι που προσφέρει το React Navigation για την καλύτερη οργάνωσή τους. Οι οθόνες που βρίσκονται μέσα στο ίδιο group μοιράζονται τα ίδια options, συνεπώς δεν χρειάζεται να γίνονται αλλαγές σε κάθε οθόνη ξεχωριστά.

Η μετάβαση στην νέα οθόνη θα γίνεται από το ChatListScreen όταν πατήσει ο χρήστης το κουμπί στο πάνω δεξί μέρος της οθόνης.

Το κουμπί 'New Chat' μαζί με την λειτουργία μετάβασης:

```
useEffect(() => {
  props.navigation.setOptions({
    headerRight: () => {
      return <HeaderButtons HeaderButtonComponent={CustomHeaderButton}>
        <Item
          title='New chat'
          iconName='create-outline'
          onPress={() => props.navigation.navigate('NewChat')}
          // The name in navigate is whatever we called it in the MainNavigator screens
        />
      </HeaderButtons>
    }
  });
}, []);
```

Στην νέα οθόνη που μεταφέρεται ο χρήστης θα υπάρχει πλαίσιο αναζήτησης (search box) που θα μπορεί να πληκτρολογεί το όνομα αυτού που θέλει να ξεκινήσει διάλογο. Η περιοχή κάτω από αυτό θα εμφανίζει τα αποτελέσματα της αναζήτησης όταν προστεθεί η λειτουργία. Όταν ανοίξει για πρώτη φορά την οθόνη ή δεν έχει γίνει ακόμη αναζήτηση θα εμφανίζεται αντίστοιχο μήνυμα όπως και παρόμοια ειδοποίηση στην περίπτωση που δεν έχουν βρεθεί αποτελέσματα.

Θα υπάρχουν τρεις συνθήκες, καταστάσεις, βάσει των οποίων θα γίνεται η εμφάνιση του περιεχομένου. Μία όταν δεν έχει πραγματοποιηθεί ακόμη αναζήτηση, η επόμενη όταν έχει γίνει αλλά δεν υπάρχουν αποτελέσματα και η τελευταία όταν υπάρχουν αποτελέσματα και εμφανιστούν στην οθόνη. Η εισαγωγή του ονόματος γίνεται στο search box και αυτό το στοιχείο (TextInput) έχει onChangeText που καλεί μέθοδο κάθε φορά που αλλάζει το περιεχόμενό του. Η μεταβλητή state searchTerm θα καταγράφει το κείμενο που έχει εισαχθεί και η setSearchTerm θα ενημερώνει την τιμή κάθε φορά που γίνεται αλλαγή. Η αναζήτηση θα γίνεται αυτόματα κάθε φορά που ο χρήστης πληκτρολογεί χωρίς να χρειάζεται να πατήσει κάποιο κουμπί.

Σε ένα useEffect θα προστεθεί η λειτουργία και ο κώδικας σε αυτό θα εκτελείται κάθε φορά που αλλάζει το searchTerm. Ορίζεται ένα χρονόμετρο (timer) που μετά από ένα πολύ μικρό διάστημα (0,5 δευτερόλεπτα) θα εκτελεί την αναζήτηση. Κάθε φορά που εισάγεται ένα γράμμα δημιουργείται και ένα νέο timer, έτσι θα πρέπει να υπάρχει μία αναφορά σε αυτό ώστε κάθε φορά που αλλάζει το searchTerm (γίνεται re-render) να εξαλείφεται ο προηγούμενος timer. Η αναφορά στον timer θα χρησιμοποιηθεί στην μέθοδο clearTimeout. Την στιγμή που ο χρήστης σταματήσει να πληκτρολογεί δεδομένα τότε αρχίζει η αντίστροφη μέτρηση του μισού δευτερολέπτου και γίνεται η αναζήτηση.

Η μέθοδος searchUsers (αρχείο userActions) θα χειρίζεται την αναζήτηση και θα καλύπτει την περίπτωση που ο χρήστης εισάγει κεφαλαίους χαρακτήρες.

Η αναζήτηση στο Firebase δουλεύει διαφορετικά καθώς η βάση δεδομένων είναι noSql, δηλαδή ανακτά δεδομένα διαφορετικά από την παραδοσιακή δομή των σχεσιακών βάσεων δεδομένων που βασίζεται σε πίνακες. Η δομή του είναι στην ουσία σαν ένα JSON αντικείμενο.

Θα υπάρχει αναφορά στον κόμβο users όπου βρίσκονται όλοι οι χρήστες και η αναζήτηση θα γίνεται με βάση το πεδίο 'firstLast'. Θα ξεκινάει στον χαρακτήρα που εισάγεται και θα τελειώνει σε οποιοδήποτε άλλο χαρακτήρα. Παράδειγμα, ο συμβολισμός "Fre" θα εμφανίσει και τις καταγραφές "Fred", "Freddy", "Frey".

```
const queryRef = query(
  userRef, orderByChild('firstLast'), startAt(searchTerm), endAt(searchTerm + '\uf8ff')
);
```

Η μέθοδος searchUsers πλέον θα κληθεί και θα επιστρέψει τα αποτελέσματα της αναζήτησης ενώ μπορεί να περιέχει περισσότερους από έναν εγγεγραμμένους χρήστες. Με αυτά θα ενημερωθεί η state

μεταβλητή `users` περνώντας στην μέθοδο `setUsers` τα αποτελέσματα. Μέχρι να ολοκληρωθεί η διαδικασία επιστροφής των δεδομένων θα εμφανίζεται ένδειξη (loading indicator).

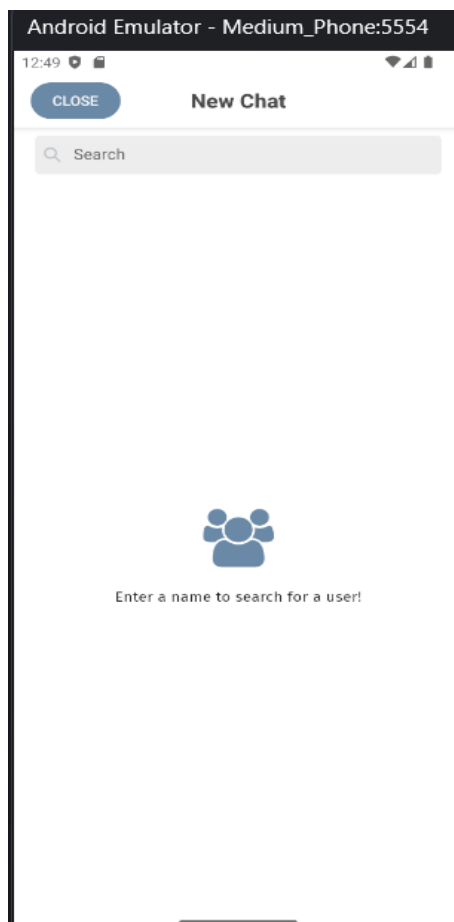
Η απεικόνιση των αποτελεσμάτων αναζήτησης θα γίνει στο στοιχείο `FlatList` της `React Native`. Το `FlatList` παρέχει ένα πλαίσιο (container), χρησιμοποιείται όταν δεν είναι γνωστό πόσα στοιχεία θα υπάρχουν σε αυτό, συνήθως όταν υπάρχει ένας μεγάλος αριθμός αποτελεσμάτων. Εκτελείται για κάθε ένα αντικείμενο του `dataset` και το `userId` που επιστρέφει μπορεί να χρησιμοποιηθεί για την απεικόνιση των λεπτομερειών του χρήστη. Τα δεδομένα χρήστη βρίσκονται στο `users` object.

Το νέο component `DataItem` θα εμφανίζει όλες τις λεπτομέρειες των χρηστών για τους οποίους έχει γίνει αναζήτηση. Κάθε χρήστης θα είναι ένα στοιχείο που όταν επιλεγεί θα ξεκινάει νέος διάλογος μαζί του. Θα πραγματοποιείται δηλαδή μία ενέργεια πάνω σε αυτό, έτσι θα είναι ένα `TouchableWithoutFeedback` στοιχείο της `React Native`.

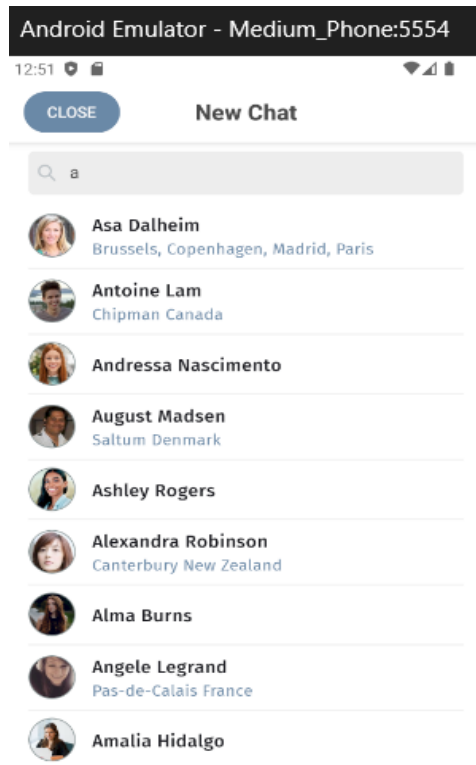
Το `DataItem` component λοιπόν θα χρησιμοποιηθεί στο `FlatList` και θα χειρίζεται την εμφάνιση του ονόματος, την φωτογραφία προφίλ χρήστη καθώς και επιπλέον πληροφορίες εάν έχουν προστεθεί. Το `ProfileImage` εμφάνιζε την εικόνα στην οθόνη `settings` όπου υπήρχε και ένα εικονίδιο που έδειχνε στον χρήστη πως μπορεί να ανανεώσει την φωτογραφία προφίλ του. Στην οθόνη που βρισκόμαστε τώρα δεν θέλουμε αυτό το εικονίδιο και το `ProfileImage` θα είναι αυτό που θα καθορίζει πότε θα εμφανίζεται με την μεταβλητή `showEditButton`.

```
const showEditButton = props.showEditButton && props.showEditButton === true;
```

Αλλαγή θα γίνει και στο `settings` screen που θα περαστεί σαν prop το `showEditButton` με την τιμή `true` γιατί εκεί θέλουμε πράγματι να εμφανίζεται.



Εικόνα 5.1 - Οθόνη αναζήτησης χρηστών



Εικόνα 5.2 - Δυναμική αναζήτηση με τα αποτελέσματα

Αυτό που θα γίνεται είναι όταν πατήσουμε σε ένα από τα αποτελέσματα θα μεταφερόμαστε σε νέα σελίδα διαλόγου με αυτόν τον χρήστη. Ο χρήστης βεβαίως που θα έχει συνδεθεί (εμείς) δεν θα συμπεριλαμβάνεται στα αποτελέσματα της αναζήτησης.

```
delete usersResult[userData.userId];
```

Η μέθοδος `userPressed` στο `newChatScreen` θα χειρίζεται όταν γίνεται κλικ πάνω σε όνομα χρήστη. Στο `FlatList` και στο component `DataItem` θα προστεθεί το property `onPress` με αυτή την μέθοδο. Ο χρήστης θα μεταφέρεται στην προηγούμενη οθόνη (`ChatListScreen`) μαζί με δεδομένα (`selectedUserId`). Το `onPress` το χειριζόμαστε αναλόγως στο `DataItem`. Πλέον υπάρχει πρόσβαση στο `selectedUserId` στην οθόνη `ChatListScreen`.

```
const selectedUser = props.route?.params?.selectedUserId;
```

Περιέχει το `userId` του επιλεγμένου χρήστη εάν βεβαίως υπάρχει.

Το `useEffect` (στο `ChatListScreen`) θα χειρίζεται την πλοήγηση στην οθόνη διαλόγου (`ChatScreen`) ενώ θα μεταφέρει μαζί και κάποια δεδομένα που θα είναι οι χρήστες της νέας συνομιλίας, δηλαδή ο επιλεγμένος χρήστης και εμείς καθώς είμαστε μέρος της συνομιλίας.

Μετά την πλοήγηση στον νέο διάλογο (`ChatScreen`) μπορούμε να έχουμε πρόσβαση στα δεδομένα που είναι ένας πίνακας με όλους τους χρήστες της συνομιλίας και με αυτά τα δεδομένα θα δημιουργηθεί ο διάλογος. Το ένα `userId` είναι το δικό μας και το άλλο ανήκει στον χρήστη με τον οποίο συνομιλούμε. Με αυτά τα `userId` θα γίνει η ανάκτηση των δεδομένων για τον κάθε χρήστη. Για τον χρήστη που έχει συνδεθεί (εμείς) είναι ευκολότερο γιατί υπάρχει ήδη στο `state` της εφαρμογής ενώ για τον άλλον θα πρέπει να γίνει αίτημα προς την βάση δεδομένων πράγμα που δεν είναι ιδανικό. Κατά την αναζήτηση έχει γίνει ήδη αίτημα προς το `Firebase` για τα δεδομένα του χρήστη και δεν θέλουμε στο `chat page` να γίνει νέα κλήση. Είναι εφικτό να περνούσαν στην νέα οθόνη ολόκληρα τα δεδομένα αντί για το `userId` αλλά η μεταφορά μεγάλων αντικειμένων είναι κάτι που δεν προτείνεται. Στην πραγματικότητα το εγχειρίδιο του `React Navigation` αναφέρει συγκεκριμένα πως κάτι τέτοιο δεν πρέπει να γίνεται αλλά να χρησιμοποιείται το `state` για την ανάκτηση.

Αυτό λοιπόν που πρέπει να γίνει είναι για κάθε χρήστη που βρίσκεται να αποθηκεύεται στο `state` της εφαρμογής. Θα υπάρχει ξεχωριστός κόμβος στο `state` (`storedUsers`) και κάθε ένας για τον οποίο γίνεται αναζήτηση θα αποθηκεύεται εκεί. Θα δημιουργηθεί ένα νέο `user slice` που θα ενημερώνει τους αποθηκευμένους χρήστες και θα προστεθεί στο κεντρικό `store` (`userSlice`). Στο σημείο που γίνεται η αναζήτηση (`NewChatScreen`) θα γίνει η ενημέρωση του `state` με το `dispatch` που επιτρέπει να αποστέλλονται ενέργειες και μπορούν να τροποποιήσουν το `state`.

```
// We're passing data in the userSlice with the name newUsers
dispatch(setStoredUsers({ newUsers: usersResult }));
```

Πλέον μπορεί να γίνει η ανάκτηση των αποθηκευμένων χρηστών από το `state` σε οποιαδήποτε οθόνη της εφαρμογής όπου δημιουργείται τοπικά μία βάση δεδομένων με όλους τους χρήστες που έχει γίνει αναζήτηση.

5.3 Ανάκτηση δεδομένων από τους αποθηκευμένους χρήστες

Πλέον κάθε φορά που αναζητούμε έναν χρήστη και παίρνουμε τα αποτελέσματα αυτός αποθηκεύεται στο `storedUsers slice` μέρος του `state`. Αυτό σημαίνει όταν πάμε στον διάλογο με αυτό το άτομο μπορεί να γίνει χρήση του `userId` που περνάει στο `chat` και να ανακτήσουμε τα δεδομένα του χωρίς να χρειάζεται να γίνει αίτημα στην βάση δεδομένων. Επίσης, όταν γίνεται η μεταφορά στο παράθυρο διαλόγου περνάει το `newChatData` που περιέχει έναν πίνακα με το δικό μας `id` και το `id` του ατόμου που ξεκινήσαμε την συνομιλία.

Κατά την παραμονή στην συγκεκριμένη οθόνη θα υπάρχει `state` μεταβλητή `chatUsers` που θα καταγράφει όλους τους χρήστες της συνομιλίας.

Ενημερώνεται ο τίτλος του διαλόγου που θα είναι το όνομα του χρήστη με τον οποίο συνομιλούμε. Στην μέθοδο `getChatTitleFromName` θα βρεθεί το `id` του άλλου χρήστη, όχι το δικό μας, καθώς υπάρχουν δύο `id`.

```
const getChatTitleFromName = () => {
  // Get the id of the other person
  const otherUserId = chatUsers.find(uid => uid !== userData.userId);
  const otherUserData = storedUsers[otherUserId]; // Get it from storedUsers in position we want

  return otherUserData && `${otherUserData.firstName} ${otherUserData.lastName}`;
};
```

Το όνομα λοιπόν του χρήστη μπορεί να χρησιμοποιηθεί στο navigation για να εμφανιστεί σαν τίτλος του διαλόγου.

5.3.1 Εμφάνιση μηνύματος σε καινούργιο διάλογο

Όταν ξεκινάει καινούργιος διάλογος θα εμφανίζεται μήνυμα (Bubble) που θα καλωσορίζει τους χρήστες. Θα είναι νέο component και ονομάζεται έτσι γιατί θα μοιάζει με φούσκα μέσα στην οποία θα εμφανίζονται όλου του είδους τα μηνύματα. Τα μηνύματα στο παράθυρο διαλόγου μπορεί να είναι τύπου συστήματος, ο χρήστης πρόσθεσε έναν άλλον χρήστη, ο χρήστης αποχώρησε από τον διάλογο όπως και μηνύματα που στέλνουμε εμείς και αντίστοιχα λαμβάνουμε από τον άλλον χρήστη που συνομιλούμε.

Αφού δημιουργηθεί το Bubble component θα γίνει χρήση του στο Chat Screen (μέσα στο στοιχείο ImageBackground) που είναι όλο το περιεχόμενο της οθόνης. Στο PageContainer component θα βρίσκεται ολόκληρος ο διάλογος, οποιαδήποτε μηνύματα θα εμφανίζονται εκεί μέσα.

Για να ξέρουμε πως βρισκόμαστε σε νέο διάλογο και να εμφανίσουμε το μήνυμα θα δημιουργηθεί state μεταβλητή για το chatId και θα ελέγχουμε για αυτό όπου είναι απαραίτητο.

```
const [chatId, setChatId] = useState(props.route?.params?.chatId);

{
  !chatId && <Bubble text='This is a new chat. Say hi!!' type='system' />
}
```

Θα υπάρχει μία δήλωση switch που θα εφαρμόζει το ανάλογο στυλ σύμφωνα με τον τύπο του μηνύματος. Ο τύπος (type) είναι κάτι που περνάει από τα props και θα δηλώνεται εκεί που γίνεται χρήση του Bubble.

```
const bubbleStyle = { ...styles.container };
const textStyle = { ...styles.text };

switch (type) {
  case 'system':
    textStyle.color = '#65644a';
    bubbleStyle.backgroundColor = colors.beige;
    bubbleStyle.alignItems = 'center';
    bubbleStyle.marginTop = 10;
    break;
```

Όταν θα είναι τύπου system το μήνυμα θα προστίθενται τα παραπάνω στοιχεία στυλ. Το ίδιο θα γίνεται για μηνύματα σφάλματος (error). Στο αρχείο Bubble θα υπάρχουν κοινά στοιχεία στυλ για όλα τα Bubbles που θα μοιράζονται μεταξύ τους και έπειτα θα γίνονται μετατροπές σύμφωνα με τον τύπο που θα ορίζεται από το type.

5.4 Δημιουργία του διαλόγου όταν σταλεί μήνυμα

Στο πλαίσιο κειμένου όταν εισαχθεί μήνυμα και σταλεί τότε θα δημιουργείται το chat, όταν αυτό βέβαια είναι νέος διάλογος. Μέχρι αυτό το σημείο ακόμη και όταν μεταφερθεί ο χρήστης στην νέα οθόνη το chat δεν έχει δημιουργηθεί. Το chat θα υπάρξει όταν στην πραγματικότητα σταλεί το πρώτο μήνυμα.

Εάν επιστρέψει στην αρχική οθόνη χωρίς να στείλει κάτι το chat δεν θα δημιουργηθεί ποτέ. Αυτό σημαίνει ότι στην μέθοδο `sendMessage ( ChatScreen )` θα προστεθεί η λογική για αυτή την λειτουργία. Το `chatId` θα είναι κάτι που περνάει σε αυτό τον διάλογο, ωστόσο εάν είναι η πρώτη φορά σε αυτό το (id) δεν θα έχει τιμή. Σημαίνει πως είναι το πρώτο chat και πρέπει να δημιουργηθεί. Το αρχείο `chatActions (utils/actions)` θα χειρίζεται την δημιουργία του chat και η μέθοδος `createChat` θα στέλνει δεδομένα με τους χρήστες, ποιος το δημιούργησε, ποιος ανανέωσε τελευταία το chat, την ημερομηνία που στάλθηκε το πρώτο μήνυμα και τότε ενημερώθηκε. Μετά την σύνδεση στο Firebase θα γίνει push των νέων δεδομένων, θα έχει πρόσβαση στον κόμβο 'chats' που δεν υπάρχει ακόμη αλλά θα δημιουργηθεί όταν σταλεί το πρώτο μήνυμα. Στην ουσία χρησιμοποιείται η μέθοδος `push` του `firebase/database` και προσδιορίζεται η τοποθεσία που θα γίνει η ενέργεια, τον κόμβο `chats` δηλαδή μαζί με τα δεδομένα.

```
const newChat = await push(child(dbRef, 'chats'), newChatData);
```

Θα προστεθεί νέα καταχώριση στην βάση δεδομένων για τα 'userChats'. Δύο κόμβοι θα υπάρχουν ένας για τα 'chats', εκεί που θα πηγαίνουν όλα τα δεδομένα του κάθε διαλόγου, αλλά θα υπάρχει και ακόμη ένας ξεχωριστά ο 'userChats' ο οποίος θα έχει καταχωρήσεις για κάθε χρήστη και θα καταγράφει τα chats που αυτοί έχουν πάρει μέρος. Ο επιπλέον κόμβος Θα κάνει ευκολότερο να βρούμε σε ποιους διαλόγους ο συγκεκριμένος χρήστης ανήκει.

Η μορφή του κόμβου θα είναι το όνομα `userChats` και θα ακολουθεί το `userId` του χρήστη που κοιτάμε.

```
const chatUsers = newChatData.users;
for (let i = 0; i < chatUsers.length; i++) {
  const userId = chatUsers[i];
  await push(child(dbRef, `userChats/${userId}`), newChat.key);
}

return newChat.key;
```

Το `newChat.key` θα είναι το key για την νέα καταχώριση που προστέθηκε. Το Firebase κάθε φορά που γίνεται χρήση της μεθόδου `push` δημιουργεί ένα μοναδικό id για αυτό. Η μέθοδος `createChat` θα επιστρέψει αυτό το key και μπορεί να χρησιμοποιηθεί στο `ChatScreen` (μέθοδος `sendMessage`).

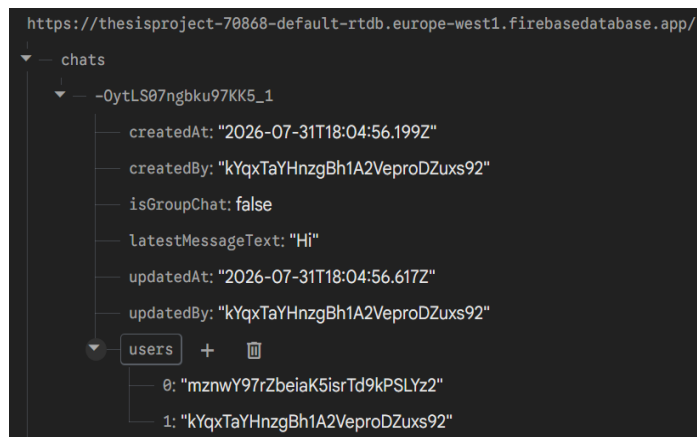
Αν λοιπόν δεν υπάρχει id σημαίνει πως βρισκόμαστε σε νέο διάλογο και τότε θα δημιουργηθεί ένας καινούργιος.

```
// No chat Id. We're on a new chat. Create the chat
id = await createChat(userData.userId, props.route.params.newChatData);
setChatId(id);
```



Εικόνα 5.3 - Μήνυμα καλωσορίσματος σε νέα συνομιλία

Την στιγμή που θα σταλεί το πρώτο μήνυμα σε καινούργιο chat και εξαφανιστεί το Bubble με το μήνυμα καλωσορίσματος τότε είναι που δημιουργείται και το id του. Ταυτόχρονα στο Firebase θα υπάρχουν δύο κόμβοι, chats και userChats.



Εικόνα 5.4 - Νέα συνομιλία στον κόμβο chats με τα δεδομένα και τους users που αποτελούνται από τον άλλον χρήστη και εμάς.



Εικόνα 5.5 - Κόμβος userChats με τα id του χρήστη και την λίστα με τους επιμέρους διαλόγους που συμμετέχει.

Κόμβος userChats που έχει έναν κόμβο για κάθε χρήστη αυτού του διαλόγου. Ο κάθε κόμβος αποτελείται από το userId των δύο χρηστών αυτού του chat και ακολουθεί το id του chat που παίρνουν μέρος. Για ευκολότερη κατανόηση είναι τα δύο id που βρίσκονται στους users του chats node.

Εάν αναζητήσουμε άλλον χρήστη και ξεκινήσουμε καινούργιο διάλογο μαζί του στον κόμβο chats θα προσθέσει καινούργιο chatId με τα νέα στοιχεία. Επίσης, στον userChats στον δικό μας κόμβο θα υπάρχει και το νέο chatId αφού πήραμε μέρος σε ακόμη έναν διάλογο και ακόμη ένας χρήστης με το ίδιο chatId.



Εικόνα 5.6 - Id χρηστών, κάποιος μετέχει σε δύο διαλόγους ενώ οι υπόλοιποι χρήστες σε έναν.

Κόμβος userChats, για κάθε χρήστη (αυτά είναι τα userId's) καταγράφονται όλες οι συνομιλίες (chatId) που έχουν λάβει μέρος. Έχουμε δύο chatId's στον δικό μας κόμβο γιατί παίρνουμε μέρος σε δύο συνομιλίες.

5.5 Ανάκτηση Χρηστών Όταν Ξεκινάει η Εφαρμογή

Δημιουργούμε τον διάλογο με την έννοια ότι στέλνουμε μήνυμα και αυτός φτιάχνεται στο πίσω μέρος. Όταν ξεκινήσει η εφαρμογή την πρώτη φορά θα γίνεται αίτημα στην βάση δεδομένων και θα ανακτούμε όλες τις συνομιλίες του χρήστη που έχει συνδεθεί. Αφού έχουμε τις συνομιλίες από το Firebase θα τις αποθηκεύσουμε στο state της εφαρμογής. Αυτό σημαίνει πως θα μπορούμε να τις εμφανίσουμε στην αρχική οθόνη όπως επίσης όταν ψάξουμε για έναν χρήστη και πατήσουμε πάνω του να ξέρουμε αν έχουμε συνομιλία μαζί του και να την φορτώσουμε αντί να φτιάξουμε καινούργια.

Η μεθοδολογία θα γίνει στο MainNavigator γιατί εκεί βρίσκεται το κύριο μέρος της εφαρμογής όταν γίνει η σύνδεση και η ανάκτηση θα γίνεται από την βάση δεδομένων και τον κόμβο userChats.

Σε useEffect θα γίνεται το αίτημα στο Firebase και συγκεκριμένα αναφορά στον κόμβο userChats και το δικό μας userId.

```
const userChatsRef = child(dbRef, `userChats/${userData.userId}`);
```

Η μέθοδος **onValue** θα εκτελέσει το αίτημα και για κάθε εκτέλεση θα επιστρέφει ένα snapshot. Η μορφή που έχει είναι key – value με το key να έχει τιμή που παράγει αυτόματα το Firebase και το value είναι το chatId ή αλλιώς του διαλόγου που είμαστε μέρος του το οποίο και μας χρειάζεται. Θα κρατήσουμε σε έναν πίνακα μόνο αυτές τις τιμές.

Η **onValue** θα εκτελείται κάθε φορά που γίνεται αλλαγή στον userChats κόμβο, δηλαδή όταν δημιουργείται καινούργιος διάλογος και αυτόματα φτιάχνει έναν listener όπως ονομάζεται από το Firebase. Εάν κλείσει η εφαρμογή ή γίνει αποσύνδεση θα πρέπει να γίνει αφαίρεση αυτού του listener γιατί αλλιώς θα πυροδοτείτε συνεχώς. Όταν υπάρχουν listeners αυτού του τύπου θα πρέπει να γίνεται απεγγραφή καλώντας την μέθοδο **off** που δίνει το Firebase. Θα υπάρχουν αρκετοί listeners γιατί θα γίνονται αιτήματα προς την βάση και για άλλες λειτουργίες, έτσι θα πρέπει κάθε φορά να γίνεται απεγγραφή. Αντί να γίνεται διαγραφή χωριστά για την κάθε περίπτωση θα υπάρχει ένας πίνακας refs που θα περνάνε όλοι οι listeners και στο τέλος του useEffect όταν θα κλείσει η λειτουργία του MainNavigator θα διαγράφονται όλοι οι listeners με την μέθοδο **off**.

Ως τώρα έχουμε τα έχουμε τα userChats που είναι τα id των chat στα οποία συμμετέχει ο χρήστης, ωστόσο δεν περιέχουν τα πραγματικά δεδομένα της συνομιλίας. Τα δεδομένα είναι διαθέσιμα στον κόμβο chats στον οποίο βρίσκονται τα chatId και τα δεδομένα των συνομιλιών. Έτσι, θα γίνει επανάληψη για κάθε διάλογο που συμμετέχει ο χρήστης και από εκεί η ανάκτηση των δεδομένων.

```
const chatRef = child(dbRef, `chats/${chatId}`);
```

Πρόσβαση στον κόμβο chats στην θέση chatId για το κάθε chat ξεχωριστά και ανάκτηση των πραγματικών δεδομένων των συνομιλιών.

Επειδή δεν είναι γνωστό πόσοι διάλογοι υπάρχουν και η επανάληψη γίνεται ένα προς ένα διάλογο θα καταγράφεται πόσοι έχουν φορτωθεί και όταν έχουν ολοκληρωθεί όλοι θα γίνει ενημέρωση του state. Θα είναι αποθηκευμένοι στο state της εφαρμογής και από εκεί θα εμφανιστούν στην κύρια οθόνη.

```
onValue(chatRef, (chatSnapshot) => {
  chatsFoundCount++;

  const data = chatSnapshot.val();
```

Είναι στην διάθεσή μας τα πραγματικά δεδομένα για κάθε συνομιλία που συμμετέχει ο χρήστης και θα πρέπει να αποθηκευτούν στο Redux state της εφαρμογής. Θα γίνει μέσω του store με ακόμη ένα slice (chatSlice). Η ενημέρωση θα γίνεται από το dispatch αφού έχουν ανακτηθεί όλα τα chats, τότε θα σταλεί ένα action στην μέθοδο setChatsData.

Συνοπτικά τα βήματα είναι, όταν τελειώσει η φόρτωση της εφαρμογής η πρώτη ενέργεια είναι η ανάκτηση των chatId's όλων των συνομιλιών που μετέχει ο χρήστης. Αμέσως μετά γίνεται επανάληψη ένα προς ένα για όλα και επιστρέφονται τα πραγματικά δεδομένα αυτών των διαλόγων. Όταν έχουμε στην διάθεσή μας όλα τα δεδομένα (όλων των συνομιλιών) γίνεται dispatch αυτών και σώζεται το chatData στο state. Αυτό σημαίνει πως οπουδήποτε αλλού στην εφαρμογή υπάρχει πρόσβαση στα στοιχεία των διαλόγων.

Όταν φορτώνεται ο MainNavigator, το κύριο μέρος της εφαρμογής, το πρώτο πράγμα που γίνεται είναι να ανακτήσει τους διάλογους, θα υπάρχει ένδειξη 'loading' που θα χάνεται όταν ολοκληρωθεί η διαδικασία ενημέρωσης του state και γίνει dispatch το action.

Η state μεταβλητή isLoading θα καταγράφει την κατάσταση του loading. Ακόμη και αν ο χρήστης δεν μετέχει σε κάποιον διάλογο θα πρέπει να αλλάζει η τιμή της για να εμφανιστεί η αρχική οθόνη.

5.5.1 Εμφάνιση Λίστας Συνομιλιών

Στην κύρια οθόνη (ChatListScreen) θα γίνεται η εμφάνιση των συνομιλιών που μετέχει ο χρήστης. Τα δεδομένα τους ανακτώνται από το state και είναι αποθηκευμένα στην μεταβλητή userChats. Γίνεται χρήση του στοιχείου FlatList της React Native καθώς αυτή είναι η μεθοδολογία για την εμφάνιση δυναμικού περιεχομένου. Το property renderItem είναι μέθοδος που καλείται για κάθε στοιχείο και επιστρέφει ένα component.

Τα δεδομένα του userChats έχουν την μορφή Object δεν είναι πίνακας (Array), αν και υπάρχουν όλα τα στοιχεία που χρειαζόμαστε. Χρήσιμες είναι μόνο οι τιμές για κάθε στοιχείο που θα εμφανιστεί και βρίσκονται στην μεταβλητή userChats αφού προηγείται μετατροπή τους σε μορφή πίνακα. Δεν είναι ιδιαίτερα χρήσιμο γιατί αυτό που θέλουμε είναι τα ονόματα των συνομιλητών και δεν υπάρχουν σε αυτό το στοιχείο. Ωστόσο, το chatData έχει τα δεδομένα σε κάθε επανάληψη του FlatList και εκεί υπάρχει ένα property **users** που έχει την μορφή πίνακα και περιέχει τα userId των ατόμων που είναι μέρος του συγκεκριμένου διαλόγου (εμείς και ακόμη ένας χρήστης).

```
<FlatList
  data={userChats}
  renderItem={({itemData}) => {
    // A function which is called on every element to return some component
    const chatData = itemData.item;
    const chatId = chatData.key;
```

Η πρώτη ενέργεια είναι να βρεθεί το id του άλλου χρήστη καθώς αυτού τα στοιχεία θα εμφανιστούν. Θα ψάξουμε το **chatData.users** (array με userId's) για το πρώτο στοιχείο id που δεν είναι ίδιο με του χρήστη που είναι συνδεδεμένος (εμάς δηλαδή).

5.5.2 Ρύθμιση Αποθηκευμένων Χρηστών στην Σύνδεση

Ανάκτηση δεδομένων για κάθε χρήστη όταν γίνεται η σύνδεση και αποθήκευση στο state της εφαρμογής. Στο MainNavigator στο σημείο που ανακτούμε τα δεδομένα, το data εκτός από τα υπόλοιπα στοιχεία περιέχει και ένα users property που είναι οι χρήστες του συγκεκριμένου διαλόγου. Λόγω αυτού μπορούμε να έχουμε πρόσβαση στον πίνακα users και να ψάξουμε τα userId's.

Υπάρχουν ήδη οι αποθηκευμένοι χρήστες που ανακτώνται από το state στην μεταβλητή storedUsers και θα γίνει έλεγχος πως δεν υπάρχει ήδη ο χρήστης σε αυτό. Εάν ο πίνακας storedUsers έχει ήδη μία καταχώρηση για το userId που κοιτάμε σημαίνει πως έχουμε βρει ήδη αυτό τον χρήστη προηγουμένως και δεν θα γίνει καμία ενέργεια. Στην περίπτωση που δεν ισχύει η παραπάνω συνθήκη σημαίνει ότι στο storedUsers δεν υπάρχει καταχώρηση στο userId που γίνεται έλεγχος, τότε θα γίνει ανάκτηση δεδομένων αυτού του χρήστη και θα προστεθεί στο state της εφαρμογής. Η αναφορά θα είναι στον κόμβο users και το userId του χρήστη που βρίσκονται όλα τα στοιχεία του και θα ενημερωθεί το state με ενέργεια (action) στην μέθοδο setStoredUsers. Έχουμε ανακτήσει τα chats που ο χρήστης συμμετέχει και για κάθε ένα από αυτά κάνουμε επανάληψη στον πίνακα data.users (τα άτομα που έχει το chat) όπου και παίρνουμε τα δεδομένα. Η ενέργεια αυτή γίνεται στο MainNavigator γιατί οποιαδήποτε στιγμή στην εφαρμογή σε όλες τις οθόνες αν θέλουμε να έχουμε πρόσβαση στα δεδομένα κάποιου χρήστη αυτά θα είναι διαθέσιμα.

```
data.users.forEach(userId => {
  if (storedUsers[userId]) return;

  const userRef = child(dbRef, `users/${userId}`);

  get(userRef).then(userSnapshot => {
    const userSnapshotData = userSnapshot.val();
    dispatch(setStoredUsers({ newUsers: { userSnapshotData } }));
    // userSlice is expecting an object because we're iterating on its values
    // so we pass the property newUsers with the value of userSnapshotData as an object
  });
});
```

5.5.3 Εμφάνιση Ενεργών Συνομιλιών

Αφού υπάρχουν διαθέσιμοι οι αποθηκευμένοι χρήστες όταν ο χρήστης συνδέεται στην εφαρμογή, σημαίνει πως στην οθόνη ChatListScreen μπορεί να γίνει η πρόσβαση σε αυτούς και η εμφάνιση στο FlatList. Η πρόσβαση γίνεται από το storedUsers που έχει έναν πίνακα με όλους τους αποθηκευμένους χρήστες και συγκεκριμένα στο userId οποιουδήποτε χρήστη εκτός από αυτόν που έχει συνδεθεί στην εφαρμογή.

```
const otherUserId = chatData.users.find(uid => uid !== userData.userId);
const otherUser = storedUsers[otherUserId];
```

Θα χρησιμοποιηθεί το DataItem component για την εμφάνιση των δεδομένων, το PageContainer για την μορφοποίηση του φόντου και των διαστημάτων καθώς και το PageTitle που θα ορίζει τον τίτλο της οθόνης.

Από δω και πέρα κάθε φορά που θα προστίθεται νέα συνομιλία θα εμφανίζεται στην λίστα. Η ταξινόμησή τους θα γίνεται με βάση την ημερομηνία που στάλθηκε το τελευταίο μήνυμα, δηλαδή ο διάλογος που ενημερώθηκε πιο πρόσφατα θα εμφανίζεται πρώτος στην λίστα. Οι διάλογοι στον κόμβο chats της βάσης δεδομένων έχουν ένα updatedAt πεδίο.

Αφού εμφανιστούν σε μορφή λίστας θα προστεθεί η λειτουργία όταν ο χρήστης πατήσει πάνω σε έναν από αυτούς να μεταφέρεται στην αντίστοιχη συνομιλία. Θα χρειαστεί το chatId, να δηλωθεί το onPress στο DataItem που χειρίζεται την πλοήγηση στο ChatScreen και θα περάσει σε αυτή την οθόνη το chatId.

```
const chatId = chatData.key;
```

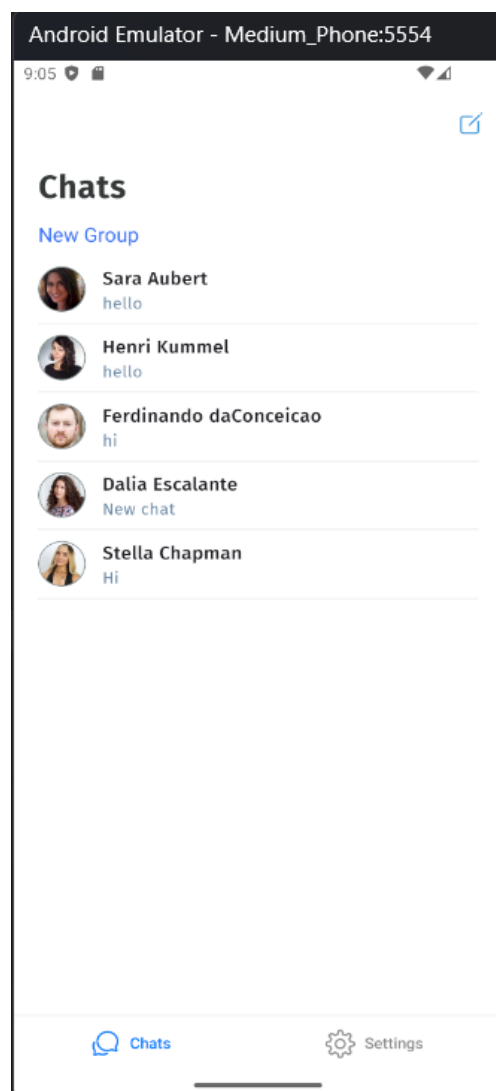
```
onPress={() => props.navigation.navigate('ChatScreen', { chatId })}
```

Στην οθόνη συνομιλίας (ChatScreen) η μεταβλητή chatData παίρνει τιμή μόνο όταν πρόκειται για νέα συνομιλία όμως θέλουμε να γίνεται το ίδιο για έναν υπάρχον διάλογο. Η ανάκτηση θα προέρχεται από το state που εκεί υπάρχουν αποθηκευμένα αυτά τα δεδομένα.

```
const storedChats = useSelector(state => state.chats.chatsData);
```

Αυτό που κάνουμε είναι να ελέγξουμε αν υπάρχει chatId τότε το chatData θα προέρχεται από τους αποθηκευμένους διαλόγους στο state διαφορετικά θα είναι νέος διάλογος.

```
// Users array passed from the ChatListScreen, contains our userId and the selectedUser
const chatData = (chatId && storedChats[chatId]) || props.route?.params?.newChatData || {};
```



Εικόνα 5.7 - Κεντρική οθόνη και η λίστα με τους ενεργούς διαλόγους.

5.6 Επίλογος

Στο παρόν κεφάλαιο παρουσιάστηκε η ανάπτυξη του βασικού μηχανισμού δημιουργίας και διαχείρισης των συνομιλιών της εφαρμογής ανταλλαγής μηνυμάτων. Αρχικά υλοποιήθηκε η δυνατότητα δημιουργίας νέου διαλόγου μέσω της κεντρικής οθόνης των συνομιλιών, παρέχοντας στον χρήστη τη δυνατότητα να μεταβαίνει σε ξεχωριστή οθόνη αναζήτησης και να εντοπίζει άλλους εγγεγραμμένους χρήστες της εφαρμογής.

Για την πραγματοποίηση της αναζήτησης δημιουργήθηκε μηχανισμός δυναμικής ενημέρωσης των αποτελεσμάτων, ο οποίος ενεργοποιείται καθώς ο χρήστης πληκτρολογεί. Παράλληλα, εφαρμόστηκε χρονική καθυστέρηση πριν από την εκτέλεση του αιτήματος, ώστε να περιορίζονται οι περιττές αναζητήσεις προς τη βάση δεδομένων. Τα αποτελέσματα εμφανίζονται δυναμικά στην οθόνη και επιτρέπουν την επιλογή του επιθυμητού χρήστη για την έναρξη μιας νέας συνομιλίας.

Σημαντικό μέρος της υλοποίησης αποτέλεσε η διαχείριση των δεδομένων των χρηστών μέσω του Redux state. Τα στοιχεία που έχουν ήδη ανακτηθεί αποθηκεύονται κεντρικά και μπορούν να χρησιμοποιηθούν από διαφορετικές οθόνες χωρίς να απαιτείται η εκτέλεση νέου αιτήματος προς το Firebase. Με τον τρόπο αυτό μειώνεται η επανάληψη των λειτουργιών ανάκτησης και διευκολύνεται η πρόσβαση στα δεδομένα σε ολόκληρη την εφαρμογή.

Στη συνέχεια παρουσιάστηκε η δημιουργία νέας συνομιλίας κατά την αποστολή του πρώτου μηνύματος. Η επιλογή αυτή εξασφαλίζει ότι ένας διάλογος αποθηκεύεται μόνο όταν υπάρχει πραγματική δραστηριότητα μεταξύ των χρηστών και αποτρέπει τη δημιουργία κενών εγγράφων στη βάση δεδομένων. Για κάθε νέα συνομιλία δημιουργούνται κατάλληλες καταχωρίσεις τόσο στον κόμβο που περιλαμβάνει τα δεδομένα των διαλόγων όσο και στον κόμβο που συνδέει κάθε χρήστη με τις συνομιλίες στις οποίες συμμετέχει.

Παράλληλα, αναλύθηκε η διαδικασία ανάκτησης των συνομιλιών του συνδεδεμένου χρήστη κατά την εκκίνηση του κύριου περιβάλλοντος της εφαρμογής. Αρχικά ανακτώνται τα αναγνωριστικά των συνομιλιών και στη συνέχεια τα πλήρη δεδομένα κάθε διαλόγου. Μετά την ολοκλήρωση της διαδικασίας, τα δεδομένα αποθηκεύονται στο Redux store, καθιστώντας τα διαθέσιμα σε όλες τις σχετικές οθόνες.

Επιπλέον, πραγματοποιείται η ανάκτηση και αποθήκευση των δεδομένων των χρηστών που συμμετέχουν στις συνομιλίες. Έτσι, οι πληροφορίες που απαιτούνται για την εμφάνιση των ενεργών διαλόγων είναι διαθέσιμες τοπικά στο state και δεν χρειάζεται να ανακτώνται εκ νέου κάθε φορά που εμφανίζεται η σχετική οθόνη.

Η υλοποίηση ολοκληρώνεται με την εμφάνιση της λίστας των ενεργών συνομιλιών, οι οποίες ταξινομούνται σύμφωνα με την τελευταία χρονική στιγμή ενημέρωσης, ώστε οι πιο πρόσφατες να εμφανίζονται πρώτες. Μέσω της επιλογής ενός διαλόγου από τη λίστα, ο χρήστης μεταφέρεται στην αντίστοιχη οθόνη συνομιλίας, όπου ανακτώνται τα απαραίτητα δεδομένα από το ήδη ενημερωμένο state της εφαρμογής.

Συνολικά, οι λειτουργίες που αναπτύχθηκαν στο συγκεκριμένο κεφάλαιο δημιουργούν την απαραίτητη υποδομή για την ολοκληρωμένη διαχείριση των συνομιλιών.

Κεφάλαιο 6ο: Αποστολή Μηνυμάτων

6.1 Εισαγωγή

Στο παρόν κεφάλαιο παρουσιάζεται η υλοποίηση του βασικού μηχανισμού ανταλλαγής μηνυμάτων της εφαρμογής, καθώς και η ανάπτυξη πρόσθετων λειτουργιών για την επικοινωνία μεταξύ των χρηστών. Έχοντας προηγουμένως διαμορφωθεί η απαραίτητη υποδομή για τη δημιουργία, την αποθήκευση και την ανάκτηση των συνομιλιών, το επόμενο στάδιο αφορά τη διαχείριση του πραγματικού περιεχομένου που ανταλλάσσεται σε κάθε διάλογο.

Αρχικά αναλύεται η διαδικασία αποστολής και αποθήκευσης μηνυμάτων κειμένου. Κάθε μήνυμα συσχετίζεται με τη συγκεκριμένη συνομιλία στην οποία ανήκει και συνοδεύεται από τις απαραίτητες πληροφορίες για τον αποστολέα και τον χρόνο αποστολής του. Παράλληλα, ενημερώνονται τα δεδομένα της αντίστοιχης συνομιλίας, ώστε να αποτυπώνεται η τελευταία δραστηριότητα και να είναι δυνατή η σωστή παρουσίασή της στην εφαρμογή.

Στη συνέχεια εξετάζεται η ανάκτηση των μηνυμάτων από τη βάση δεδομένων και η ενσωμάτωσή τους στο κεντρικό state της εφαρμογής. Η συγκεκριμένη προσέγγιση επιτρέπει την άμεση πρόσβαση στα δεδομένα των μηνυμάτων από τις επιμέρους οθόνες και συμβάλλει στην αποτελεσματικότερη διαχείρισή τους. Παράλληλα, μέσω της αξιοποίησης των μηχανισμών παρακολούθησης μεταβολών του Firebase, η εφαρμογή μπορεί να ενημερώνει δυναμικά το περιεχόμενο των συνομιλιών όταν αποστέλλονται νέα μηνύματα.

Ιδιαίτερη σημασία δίνεται επίσης στην παρουσίαση των μηνυμάτων στην οθόνη συνομιλίας. Η εμφάνισή τους διαφοροποιείται ανάλογα με τον αποστολέα, ώστε να διακρίνονται οπτικά τα μηνύματα του συνδεδεμένου χρήστη από εκείνα του συνομιλητή του. Επιπλέον, υλοποιείται κατάλληλος μηχανισμός ενημέρωσης σε περίπτωση αποτυχίας αποστολής, παρέχοντας στον χρήστη σχετική ένδειξη σφάλματος.

Πέρα από τη βασική ανταλλαγή μηνυμάτων, στο κεφάλαιο αναπτύσσονται πρόσθετες δυνατότητες αλληλεπίδρασης. Μέσω ενός αναδυόμενου μενού επιλογών παρέχεται η δυνατότητα εκτέλεσης ενεργειών πάνω σε συγκεκριμένα μηνύματα, όπως η αντιγραφή του περιεχομένου τους, η επισήμανσή τους ως αγαπημένα και η απάντηση σε προηγούμενο μήνυμα. Για την υποστήριξη των λειτουργιών αυτών πραγματοποιούνται οι αντίστοιχες μεταβολές τόσο στη διεπαφή όσο και στη δομή των δεδομένων που αποθηκεύονται στο Firebase και στο state της εφαρμογής.

Τέλος, παρουσιάζεται η επέκταση της λειτουργίας ανταλλαγής μηνυμάτων μέσω της υποστήριξης εικόνων. Ο χρήστης μπορεί να επιλέξει μια υπάρχουσα εικόνα από τη συλλογή της συσκευής ή να πραγματοποιήσει νέα λήψη μέσω της κάμερας. Οι εικόνες αποθηκεύονται στο Firebase Storage και στη συνέχεια ενσωματώνονται στον αντίστοιχο διάλογο ως μηνύματα, επιτρέποντας την ανταλλαγή περιεχομένου μεταξύ των χρηστών.

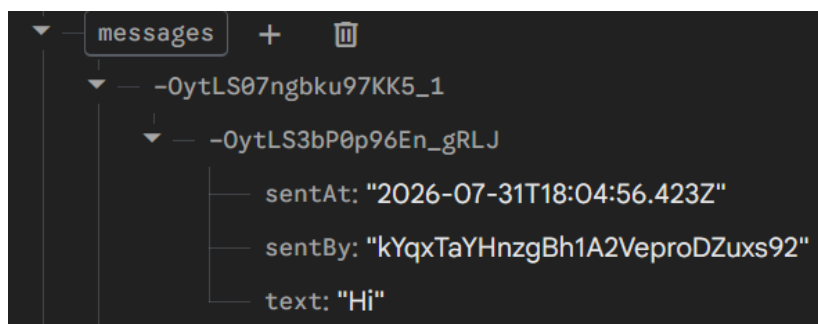
Συνολικά, το κεφάλαιο επικεντρώνεται στην ολοκλήρωση του βασικού μηχανισμού επικοινωνίας της εφαρμογής και στην προσθήκη λειτουργιών που καθιστούν την ανταλλαγή μηνυμάτων περισσότερο ολοκληρωμένη και διαδραστική.

6.2 Μηνύματα κειμένου

Μέχρι αυτό το σημείο έχει υλοποιηθεί η πλοήγηση σε υπάρχον διάλογο ωστόσο δεν υπάρχει ακόμη η δυνατότητα αποστολής πραγματικών μηνυμάτων. Όταν λοιπόν ο χρήστης πατήσει το κουμπί αποστολής θα στέλνει το μήνυμα και αυτό θα προστίθεται στην βάση δεδομένων. Η λειτουργία θα προστεθεί στην μέθοδο sendMessage όπου θα καλείται η sendMessage (αρχείο chatActions). Θα υπάρχει αναφορά στον κόμβο messages και από κάτω του το αντίστοιχο chatId.

```
const messagesRef = child(dbRef, `messages/${chatId}`);
```

Ο κόμβος messages βέβαια δεν υπάρχει ακόμη γιατί δεν έχει σταλεί κάποιο μήνυμα, μόλις όμως σταλεί το πρώτο μήνυμα τότε θα δημιουργηθεί. Ταυτόχρονα θα σταλούν και δεδομένα στην βάση δεδομένων που θα είναι ποιος έστειλε το μήνυμα, πότε στάλθηκε αυτό και περιεχόμενό του. Η μέθοδος push θα στείλει τα δεδομένα στην περιοχή αναφοράς messagesRef. Κατά την χρήση της sendMessage ανάμεσα στις παραμέτρους υπάρχει η messageText που περιέχει το κείμενο που εισάγεται, προέρχεται από την state μεταβλητή και αλλάζει κάθε φορά που ο χρήστης πληκτρολογεί. Όταν πατηθεί το κουμπί αποστολή στην βάση θα δημιουργηθεί ο κόμβος messages, μέσα του chatId και κάτω από αυτό τα μηνύματα.



Εικόνα 6.1 - Κόμβος messages με τα δεδομένα του κάθε μηνύματος

Εάν σταλεί μήνυμα σε διαφορετικό χρήστη στον ίδιο κόμβο θα προστεθεί νέο chatId με τα αντίστοιχα δεδομένα.

Την στιγμή που γίνονται οι αλλαγές στον κόμβο messages ταυτόχρονα θα ενημερωθεί και ο κόμβος chats στα πεδία updatedAt, updatedBy και θα προστεθεί ένα νέο πεδίο latestMessageText που θα είναι το τελευταίο μήνυμα που έχει σταλεί.

6.2.1 Ενημέρωση Μηνυμάτων στο State

Για να εμφανιστούν τα μηνύματα στην οθόνη συνομιλίας θα πρέπει να υπάρχει η δυνατότητα ανάκτησής τους. Μπορεί να γίνει από την βάση δεδομένων, να γίνεται αίτημα προς αυτή και να επιστρέφονται τα μηνύματα, ωστόσο ο καλύτερη μεθοδολογία είναι αυτή που χρησιμοποιήθηκε και σε άλλες λειτουργίες της εφαρμογής. Όταν λοιπόν συνδεθεί ο χρήστης θα ανακτώνται όλα τα μηνύματα από όλες τις συνομιλίες και θα προστίθενται στο state της εφαρμογής, έτσι σε αυτό το σημείο ξέρουμε πως μπορούμε να έχουμε πρόσβαση σε αυτά τα μηνύματα.

Στο MainNavigator εκεί που γίνονται όλες οι κλήσεις προς την βάση δεδομένων θα προστεθεί άλλη μία για τα μηνύματα. Για κάθε ένα chatId που έχει βρεθεί παίρνουμε τους χρήστες (users) αυτού του chat και τώρα θα πάρουμε τα μηνύματα για κάθε ένα από αυτά. Έπειτα θα αποστέλλεται μία ενέργεια στο

messages reducer που δημιουργείται στο store με όνομα messagesSlice. Η messagesSlice θα έχει την μέθοδο setChatMessages που ενημερώνει το state με τα μηνύματα.

```
reducers: {
  setChatMessages: (state, action) => {
    const existingMessages = state.messagesData;

    // The two properties that we've passed in to this function
    const { chatId, messagesData } = action.payload;

    // Update existing messages
    existingMessages[chatId] = messagesData;

    state.messagesData = existingMessages;
  },
},
```

Θα γίνει dispatch αυτού του action (MainNavigator).

```
// dispatch an action to our messages reducer
dispatch(setChatMessages({ chatId, messagesData }));
```

Αυτό που θα συμβεί είναι όταν η εφαρμογή φορτώνεται θα επαναφέρει όλα τα δεδομένα μηνυμάτων για κάθε μία συνομιλία και θα τα αποθηκεύει τοπικά.

6.2.2 Εμφάνιση Μηνυμάτων

Η οθόνη συνομιλιών είναι προς το παρόν κενή, εκεί θα εμφανίζονται όλου του είδους τα μηνύματα. Αρχικά, θα είναι η περίπτωση μηνύματος λάθους όταν ένα μήνυμα αποτύχει να παραδοθεί. Η state μεταβλητή errorBannerText θα χειρίζεται το μήνυμα και θα περνάει στο Bubble component σαν τύπος 'error' για να δώσει την κατάλληλη μορφοποίηση. Οτιδήποτε αποτύχει στην μέθοδο sendMessage θα εκτελεστεί το catch μέρος της, θα παίρνει τιμή η setErrorBannerText και το μήνυμα σφάλματος θα εξαφανίζεται μετά από πέντε δευτερόλεπτα.

Πριν μπορέσουμε να εμφανίσουμε τα μηνύματα χρειάζεται να ανακτήσουμε αυτά που αφορούν τον συγκεκριμένο διάλογο. Υπάρχουν ήδη στην διάθεσή μας τα μηνύματα από όλες τις συνομιλίες που συμμετέχουμε στην μεταβλητή chatMessages και προέρχονται από το γεγονός ότι είναι αποθηκευμένα στο state της εφαρμογής. Στην πραγματικότητα χρειάζονται μόνο αυτά της συγκεκριμένης συνομιλίας με χρήση του chatId.

```
// chatMessages contains all the messages only for this chat and it's in an array
const chatMessages = useSelector(state => {
  // chatId might not always have a value
  if (!chatId) return [];

  const chatMessagesData = state.messages.messagesData[chatId]; // is an object

  if (!chatMessagesData) return [];

  const messageList = [];

  // Loop over the chatMessagesData object
  // key is each message under the current chatId and will contain the data
  for (const key in chatMessagesData) {
    const message = chatMessagesData[key]; // message will now contain all the pr

    // We want to remember this key so we can access this message later if we wan
    messageList.push({
      key, // we're adding the key property as well, same as key: key
      ...message // spread operator, creating a copy of the message above
    });
  }

  return messageList;
});
```

Τώρα που υπάρχουν τα μηνύματα για αυτή την συνομιλία η απεικόνιση θα γίνει σε στοιχείο FlatList εάν υπάρχει ήδη id για αυτήν γιατί εάν είμαστε σε καινούργιο διάλογο δεν έχει δημιουργηθεί το chat συνεπώς δεν υπάρχει και chatId. Δεν υπάρχουν και διαθέσιμα μηνύματα άρα δεν υπάρχει λόγος να γίνει ανάκτησή τους.

Θέλουμε να ξέρουμε αν είναι το δικό μας η το μήνυμα του χρήστη που συνομιλούμε γιατί κάθε περίπτωση θα έχει διαφορετική διαμόρφωση και θα βρίσκεται στο δεξί ή αριστερό μέρος της οθόνης. Γίνεται έλεγχος λοιπόν αν είναι δικό μας μήνυμα ή άλλου και ο λόγος είναι γιατί ο τύπος του μηνύματος θα περάσει στο Bubble και θα οριστεί νέο στυλ για τον τύπο 'myMessage' και 'theirMessage'.

Τα μηνύματα που δεν έχουμε στείλει είναι αυτά του χρήστη που συνομιλούμε και για να γίνει αυτό χρειάζονται βεβαίως μηνύματα εκείνων. Ο τρόπος είναι να γίνει αποσύνδεση από τον λογαριασμό και σύνδεση με τα στοιχεία του άλλου προσώπου ή να το άνοιγμα άλλης συσκευής προσομοίωσης. Αν σταλεί μήνυμα θα εμφανιστεί απευθείας ταυτόχρονα και στις δύο συσκευές και αυτό οφείλεται στο γεγονός ότι οι listeners του Firebase (MainNavigator), στο σημείο που γίνεται η αναφορά στον κόμβο messages, η μέθοδος onValue 'βλέπει' οποιαδήποτε αλλαγή στον συγκεκριμένο κόμβο. Όταν στείλουμε νέο μήνυμα από διαφορετικό χρήστη πυροδοτεί απευθείας την παραπάνω λειτουργία γιατί υπήρχε αλλαγή και στην ουσία θα προσθέσει αυτό το μήνυμα στο state της εφαρμογής για αυτό και το βλέπουμε ταυτόχρονα και στις δύο συσκευές. Η εφαρμογή θα εμφανίσει νέα μηνύματα αυτόματα χωρίς να χρειαστεί περαιτέρω ενέργεια.

6.3 Επιπλέον Λειτουργίες Μηνυμάτων

Όταν ο χρήστης πατήσει παρατεταμένα πάνω σε μήνυμα θα εμφανίζεται ένα pop-up μενού με κάποιες επιλογές όπως να γίνει 'like', αντιγραφή κειμένου, και απάντηση στο συγκεκριμένο μήνυμα.

Θα γίνει χρήση του πακέτου React Native Pop-up-Menu και η εγκατάστασή του γίνεται με την εντολή `npm i react-native-popup-menu`.

Για να χρησιμοποιηθεί σύμφωνα με το εγχειρίδιο θα πρέπει το στοιχείο MenuProvider της βιβλιοθήκης να προστεθεί γύρω από την κύρια εφαρμογή, στο στοιχείο AppNavigator (App.js).

Το πρώτο πράγμα που πρέπει να γίνει είναι τα στοιχεία των μηνυμάτων (Bubble) να μπορούν να δέχονται κλικ γιατί προς το παρόν είναι στοιχεία View, που σημαίνει πως δεν γίνεται να εκτελέσουν κάποια ενέργεια. Όταν πρόκειται για δικό μας μήνυμα ή μήνυμα του άλλου χρήστη θα αλλάζουν σε στοιχεία TouchableWithoutFeedback που έχουν μία ενέργεια onLongPress και ενεργοποιείται όχι με ένα απλό πάτημα αλλά παρατεταμένα.

6.3.1 Άνοιγμα του Μενού

Ακολουθώντας το εγχειρίδιο χρειάζεται η προσθήκη του στοιχείου Menu από την βιβλιοθήκη, το MenuTrigger και ακολουθούν τα MenuItem μέσα στα οποία δηλώνονται οι πραγματικές επιλογές του μενού. Το μενού θέλουμε να ανοίγει όταν ο χρήστης πατήσει παρατεταμένα στο μήνυμα, λειτουργία που χειρίζεται το onLongPress. Θα δημιουργηθεί μία αναφορά στο στοιχείο Menu με το hook useRef της React, όπου το Menu είναι το root στοιχείο και θα δηλωθεί το ref. Λέμε πως η αναφορά σε αυτό το στοιχείο μπορεί να σωθεί στο menuRef και χρησιμοποιώντας την έχουμε πρόσβαση σε όλες τις μεθόδους του Menu, συγκεκριμένα την openMenu. Στο openMenu χρειάζεται να προσδιοριστεί η επιλογή που θέλουμε να εμφανιστεί με ένα id γιατί η συγκεκριμένη μέθοδος μπορεί να ανοίξει οποιοδήποτε μενού στην οθόνη. Για το id θα χρησιμοποιηθεί η βιβλιοθήκη uuid όπως και σε προηγούμενη λειτουργία της εφαρμογής.

Η useRef θα δώσει ένα νέο id και θα υπάρχει για όλη την διάρκεια λειτουργίας το Menu, δεν θα δημιουργήσει νέο id.

```
const id = useRef(uuid.v4());
// useRef remembers values and make them exist for the whole life time of the element
```

Όσα μηνύματα υπάρχουν στην οθόνη συνομιλίας τόσα id θα δημιουργηθούν και δεν θα αλλάζουν στην πορεία. Το id θα δηλωθεί στο Menu στοιχείο και στην μέθοδο openMenu αντίστοιχα.

```
<Container
  onPress={() => menuRef.current.props.ctx.menuActions.openMenu(id.current)}
  style={{ width: '100%' }}
>
```

```
<Menu name={id.current} ref={menuRef}>
```

Με το μενού πλέον να εμφανίζεται όταν πατήσει ο χρήστης πάνω σε μήνυμα θα προστεθούν οι επιλογές με την πρώτη να είναι η αντιγραφή κειμένου (copy the text). Όταν επιλεγθεί θα αντιγράψει το κείμενο του μηνύματος στο clipboard και θα γίνει χρήση του πακέτου Expo Clipboard με την εντολή `npm install expo-clipboard`

Δίπλα από κάθε επιλογή θα υπάρχουν εικονίδια που θα υποδηλώνουν την λειτουργία τους. Θα δημιουργηθεί νέο component MenuItem που θα επιστρέφει ένα MenuOption, δηλαδή κάθε επιλογή του μενού. Θα ορίζει το κείμενο και το εικονίδιο που θα εμφανίζεται δίπλα του όπως και την λειτουργία που θα εκτελείται με την εκάστοτε επιλογή.

Το επόμενο MenuItem θα είναι η προσθήκη στα αγαπημένα (like message) που θα αλλάζει ανάμεσα σε δύο εικονίδια ανάλογα εάν το μήνυμα έχει προστεθεί στα αγαπημένα ή όχι.

6.3.2 Αγαπημένα Μηνύματα

Αφορά τις ενέργειες που θα εκτελούνται όταν ο χρήστης επιλέξει να προσθέσει το μήνυμα στα αγαπημένα.

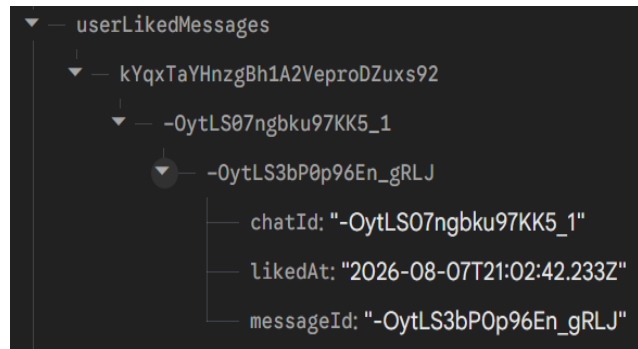
Στο αρχείο chatActions θα οριστεί η μέθοδος likeMessage, θα στέλνει αίτημα στην βάση δεδομένων και θα το επισημαίνει ως αγαπημένο και θα μπορεί να τα δει μόνο ο ενεργός χρήστης. Κανένας άλλος ο χρήστης δεν θα έχει πρόσβαση σε αυτά μόνο αυτός που τα έχει επιλέξει ως αγαπημένα για αυτό θα υπάρχει ξεχωριστός κόμβος, για κάθε χρήστη, στο Firebase που θα βρίσκονται αυτού του τύπου τα μηνύματα. Η αναφορά στην βάση θα βρίσκεται στον κόμβο με όνομα userLikedMessages, το userId του χρήστη, το chatId που παίρνει μέρος και τέλος το messageId.

```
const childRef = child(dbRef, `userLikedMessages/${userId}/${chatId}/${messageId}`);
```

Ο λόγος που δομείται έτσι είναι γιατί προσφέρει έναν γρήγορο τρόπο για να αποκτήσουμε πρόσβαση στα αγαπημένα μηνύματα του χρήστη στέλνοντας αίτημα στην συγκεκριμένη τοποθεσία της βάσης όπως και όλα τα μηνύματα που έχουν οριστεί έτσι για ολόκληρο τον διάλογο. Η μέθοδος get του Firebase στέλνει ένα αίτημα κάθε φορά και δεν ενεργοποιείται κάθε φορά που γίνεται κάποια αλλαγή όπως η onValue που χρησιμοποιήθηκε σε προηγούμενη λειτουργία.

Εάν λοιπόν κάτι υπάρχει στην τοποθεσία που ψάχνουμε τότε θα γίνεται η αφαίρεσή του ενώ αντίθετα εάν δεν υπάρχει τότε θα προστίθεται. Η χρήση της μεθόδου likeMessage θα γίνεται στο Bubble στο σημείο που ορίζεται το δεύτερο MenuItem (Like Message), και οι παράμετροι στην μέθοδο προέρχονται από το ίδιο component όπως αυτό ορίζεται στην οθόνη ChatScreen. Έτσι, καλούμε αυτή την μέθοδο πηγαίνει στην βάση, κάνει έλεγχο της τοποθεσίας, εάν δεν βρει κάποιο μήνυμα θα το τοποθετήσει σαν καινούργιο.

Η μέθοδος set θα εντάξει τα δεδομένα messageId, chatId και likedAt στην ανωτέρω τοποθεσία.



Εικόνα 6.2 - Κόμβος userLikedMessages, το id του χρήστη, το id του διαλόγου, id του μηνύματος και τα δεδομένα του.

Αφού προστέθηκε στην βάση δεδομένων αν ο χρήστης πατήσει πάλι την ίδια επιλογή σημαίνει πως θα πρέπει να γίνει η αντίθετη ενέργεια (un-like). Η αναφορά για το συγκεκριμένο μήνυμα υπάρχει ήδη στην τοποθεσία και θα πρέπει να γίνει η αφαίρεσή του με την μέθοδο remove που έρχεται με το Firebase. Το αποτέλεσμα θα είναι να αφαιρέσει τον κόμβο που είχε δημιουργηθεί με όλα τα στοιχεία του ενώ αν ο χρήστης πατήσει και πάλι την ίδια επιλογή θα τον φτιάξει εκ νέου.

6.3.3 Ενημέρωση Αγαπημένων Μηνυμάτων στο State

Θα πρέπει να γίνει ενημέρωση της οθόνης ώστε να εμφανίζει τις αλλαγές στον χρήστη, συγκεκριμένα να αλλάζει σε Unlike Message, το εικονίδιο να είναι διαφορετικό και τέλος να υπάρχει ένδειξη δίπλα στο μήνυμα που να υποδεικνύει πως βρίσκεται στα αγαπημένα του. Για να γίνει κάτι τέτοιο θα πρέπει να ανακτηθούν τα αγαπημένα μηνύματα και να αποθηκευτούν στο state της εφαρμογής για ευκολότερη πρόσβαση. Στο messagesSlice θα υπάρχει νέος reducer addLikedMessage που θα προσθέτει τα μηνύματα και ο removeLikedMessage που θα τα αφαιρεί αντίστοιχα από το state. Η πρώτη θα προσθέτει ένα αγαπημένο, η δεύτερη θα το αφαιρεί και δουλεύουν ένα μήνυμα την φορά αλλά πρέπει να οριστούν και τα αρχικά αγαπημένα μηνύματα όταν η εφαρμογή ξεκινάει και τα παίρνουμε όλα για αυτόν τον χρήστη με τον reducer setLikedMessages.

Ο MainNavigator είναι το μέρος που ορίζεται το αρχικό state για όλες τις λειτουργίες όταν ξεκινάει η εφαρμογή και εκεί θα γίνει το ίδιο για τα αγαπημένα μηνύματα. Θα οριστεί μία αναφορά στον κόμβο userLikedMessages και συγκεκριμένα στο μέρος userId του χρήστη που είναι συνδεδεμένος και θα γίνεται dispatch μίας ενέργειας στην μέθοδο setLikedMessages. Τα δεδομένα που δέχεται η μέθοδος είναι αυτά που έχουν επιστραφεί μετά το αίτημα στην τοποθεσία της βάσης.

```

const userLikedMessagesRef = child(dbRef, `userLikedMessages/${userData.userId}`);
refs.push(userLikedMessagesRef);
onValue(userLikedMessagesRef, querySnapshot => {
  const likedMessages = querySnapshot.val() ?? {};
  // If querySnapshot.val() doesn't exist or is empty then set it to an empty object
  dispatch(setLikedMessages({ likedMessages }));
});
  
```

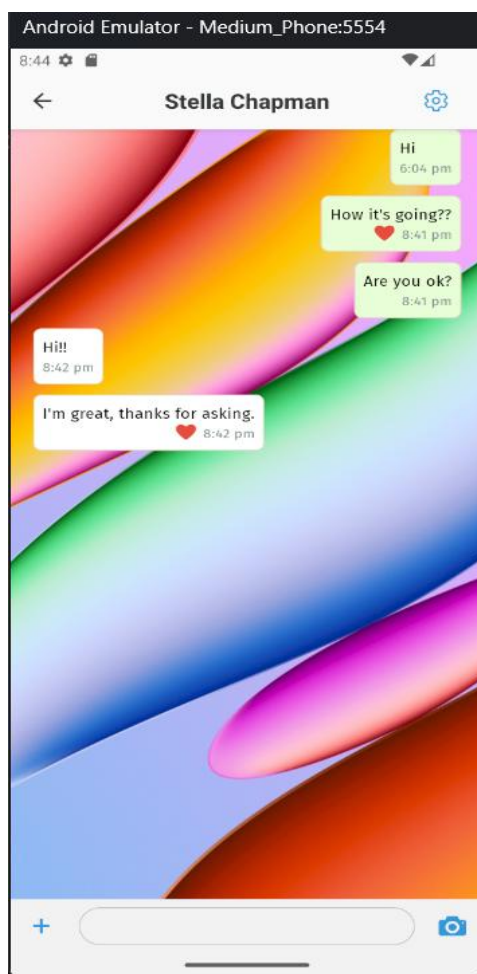
Πλέον μπορούμε να ορίσουμε ως like και unlike τα μηνύματα και ορίζεται η αρχική τιμή τους μέσω του state όταν φορτώνεται η εφαρμογή που σημαίνει πως μπορούν να ανακτηθούν σε οποιοδήποτε σημείο όπως στο Bubble Component και στην μεταβλητή likedMessages. Μπορεί να χρησιμοποιηθεί για να γίνει έλεγχος αν ένα μήνυμα είναι liked ή unliked και να ενημερώσει την οθόνη αντίστοιχα με διαφορετική επιλογή στο μενού ή να προσθέσει εικονίδιο δίπλα του.

Η μεταβλητή isUserMessage καταγράφει αν πρόκειται για μήνυμα που έχει στείλει ο χρήστης και αν ισχύει (είναι true) θα γίνει έλεγχος των αγαπημένων μηνυμάτων από το state (likedMessages) στην

θέση `messageId` αν περιέχει κάτι. Αν δεν υπάρχει κάτι εκεί (είναι `undefined`) σημαίνει πως το μήνυμα δεν βρίσκεται στα αγαπημένα ενώ αν υπάρχουν δεδομένα (δεν είναι `undefined`) ισχύει το αντίθετο. Η καταγραφή θα βρίσκεται στην μεταβλητή `isLiked` και μπορεί να χρησιμοποιηθεί για να εμφανίζεται το αντίστοιχο μήνυμα και εικονίδιο στις επιλογές του μενού.

Όταν ο χρήστης προσθέσει μήνυμα στα αγαπημένα η επιλογή του μενού θα αλλάζει σε 'unlike' γιατί στον `MainNavigator` παρατηρούνται οποιεσδήποτε αλλαγές και αν υπάρχουν ενημερώνει από την αρχή όλα τα μηνύματα του `state` (`likedMessages`).

Θα υπάρχει εικονίδιο δίπλα από το μήνυμα αν αυτό έχει επισημανθεί ως αγαπημένο όπως και η ημερομηνία που στάλθηκε αυτό. Η ημερομηνία θα περνάει σαν δεδομένο σε αυτό το `Bubble` μήνυμα καθώς υπάρχει το πεδίο `sentAt` που περιέχει την ημερομηνία που στάλθηκε και θα γίνει μετατροπή στην μορφή που θα εμφανίζεται.



Εικόνα 6.3 - Επισήμανση μηνυμάτων ως αγαπημένα και ημερομηνία αποστολής του.

6.4 Απάντηση Σε Μηνύματα

Θα προστεθεί επιπλέον επιλογή στο μενού που επιτρέπει την απάντηση σε ένα μήνυμα και θα υπάρχει ένα τμήμα στο κάτω μέρος της οθόνης που θα φαίνεται σε ποιον χρήστη γίνεται η απάντηση.

Επειδή θα γίνεται εμφάνιση / απόκρυψη ενός τμήματος στο κάτω μέρος θα χρειαστεί μία state μεταβλητή για αυτό με όνομα `replyingTo` (`ChatScreen`). Στο `Bubble Component` που επιστρέφεται στην οθόνη συνομιλίας θα οριστεί μία ενέργεια `setReply` που θα αλλάζει την state μεταβλητή μέσω της μεθόδου `setReplyingTo`.

Αυτό που θα συμβεί είναι χρησιμοποιηθεί η `setReply` μέσα στο `Bubble` και εκεί θα δημιουργηθεί μια επιλογή στο μενού που όταν την πατήσει ο χρήστης θα καλέσει αυτή την μέθοδο (`setReply`) που με την σειρά της θα θέσει την μεταβλητή `replyingTo` (`setReplyingTo`).

Θα εμφανίζεται ένα μικρό μέρος που θα εμφανίζει το μήνυμα που απαντάται και θα είναι σε ξεχωριστό component `ReplyTo`. Η `ReplyTo` θα δέχεται τρία δεδομένα από τα props και θα είναι το κείμενο που θα φαίνεται, ο χρήστης που πηγαίνει η απάντηση και μία μέθοδος (`onCancel`) που θα ακυρώνει την απάντηση.

Η χρήση του component γίνεται στην οθόνη `ChatScreen`. Το κείμενο θα είναι η τιμή της `replyingTo` state μεταβλητής που περιέχει τα δεδομένα ολόκληρου του μηνύματος. Στα συγκεκριμένα δεδομένα υπάρχει ένα πεδίο `sentBy` που είναι το `userId` που θα χρησιμοποιηθεί για να ανακτηθεί ο συγκεκριμένος χρήστης από τους αποθηκευμένους στο state της εφαρμογής (`storedUsers`).

Με το τμήμα απάντησης του μηνύματος να εμφανίζεται θα πρέπει να υπάρχει και ένας τρόπος για να μπορεί ο χρήστης να το ακυρώσει ή αλλιώς να το κλείσει. Θα προστεθεί ένα εικονίδιο με ένδειξη ακύρωσης και θα είναι στοιχείο `TouchableOpacity` καθώς θα προκαλεί μία ενέργεια όταν πατηθεί. Η ενέργεια του `onCancel` θα είναι να ενημερώσει την `replyingTo` state μεταβλητή.

Με τις παραπάνω ενέργειες έχει οριστεί ο τρόπος απάντησης μηνυμάτων ωστόσο όταν πατηθεί το κουμπί αποστολή δεν συμβαίνει κάποια ενέργεια. Θα πρέπει το `id` του μηνύματος που απαντάται να τοποθετηθεί μαζί με τα δεδομένα του μηνύματος που στέλνονται στην βάση δεδομένων. Στην μέθοδο `sendMessage` της οθόνης `ChatScreen` και συγκεκριμένα στην `sendTextMessage` θα υπάρχει ακόμη μία παράμετρος που θα είναι το `id` του μηνύματος που γίνεται απάντηση (`replyingTo.key`). Μόλις σταλεί η απάντηση τότε το μέρος θα εξαφανίζεται γιατί τίθεται το `setReplyingTo` με `null`. Οι αντίστοιχες αλλαγές θα πρέπει να γίνουν και στην μέθοδο `sendTextMessage` όπου αν το `replyTo` έχει τιμή θα προστεθεί αυτό στα δεδομένα, εάν δεν έχει κάποια τιμή που σημαίνει πως δεν απαντάμε σε κάποιο μήνυμα θα αγνοήσει εκείνο το σημείο.

Εφόσον ξέρουμε πως το μήνυμα πρόκειται για απάντηση θα πρέπει να ενημερωθεί και η οθόνη για να δείχνει στον χρήστη πως απαντάει σε αυτό. Ένα μήνυμα είναι απάντηση όταν έχει το πεδίο `replyTo` στην βάση δεδομένων. Στο σημείο που γίνεται η εμφάνισή του (στο `Bubble`) θα υπάρχει το `replyingTo` που θα βρίσκει το μήνυμα που απαντάει ο χρήστης από αυτά που βρίσκονται στο state της εφαρμογής. Το `replyingTo` που περνάει στο `Bubble` θα έχει τα δεδομένα του μηνύματος και θα χρησιμοποιηθεί στο component για να βρεθεί ο χρήστης στον οποίο απαντάμε.

```
const replyingToUser = replyingTo && storedUsers[replyingTo.sentBy];
```

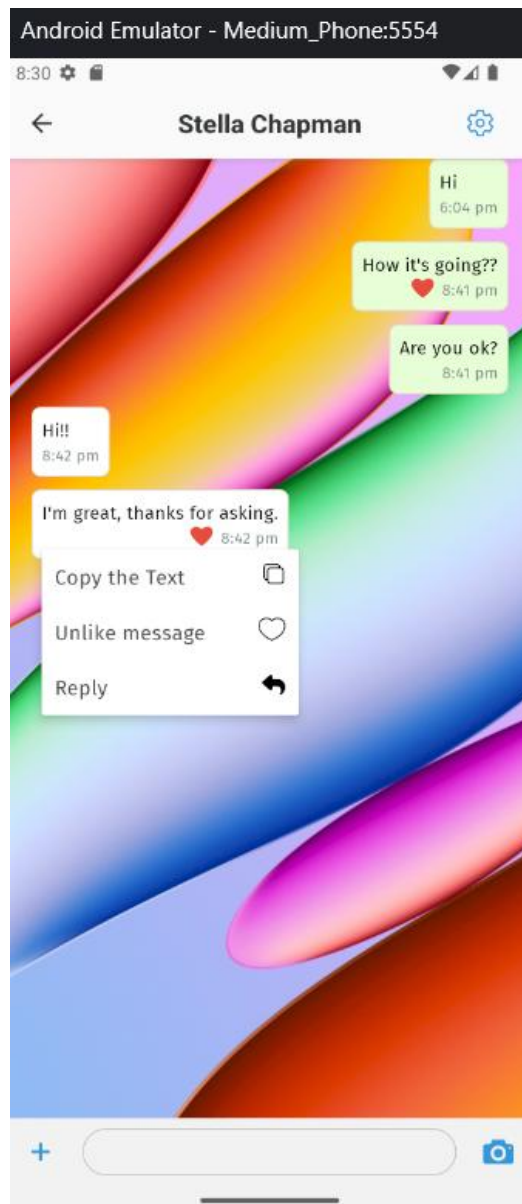
Το `replyingToUser` θα περιέχει τα δεδομένα χρήστη του προσώπου που έστειλε το μήνυμα που απαντάμε. Βάσει αυτού θα εμφανίζεται το περιεχόμενο της απάντησης με χρήση του ίδιου `Bubble component` που ο τύπος του θα είναι `'reply'`, το κείμενο θα προέρχεται από τα δεδομένα του `replyingTo` και το όνομα του χρήστη (`replyingToUser`) του οποίου το μήνυμα απαντάται που θα φαίνεται ακριβώς πάνω από το `Bubble`.

```

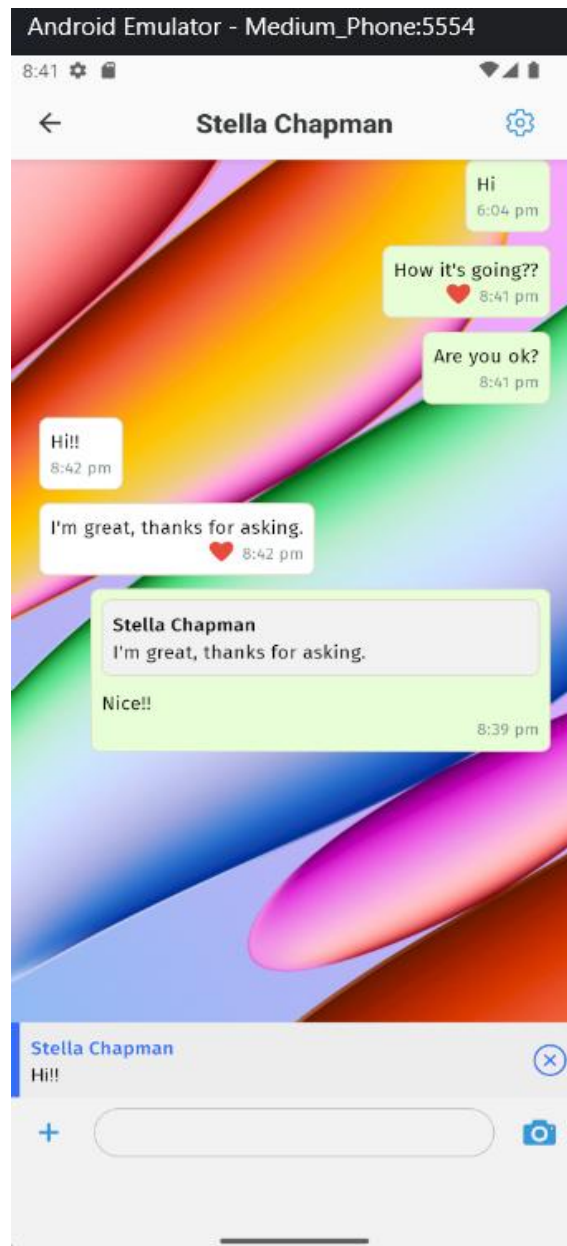
{ name && type === 'info' && <Text style={styles.name}>{name}</Text> }

{
  replyingToUser &&
  <Bubble
    type='reply'
    text={replyingTo.text}
    name={` ${replyingToUser.firstName} ${replyingToUser.lastName}`}
  />
}

```



Εικόνα 6.4 - Παρατεταμένο πάτημα στο μήνυμα και εμφάνιση μενού επιλογών.



Εικόνα 6.5 - Απάντηση σε υφιστάμενο μήνυμα και εμφάνισή του στο κάτω μέρος της οθόνης.

6.5 Αποστολή Μηνυμάτων Εικόνας

Στην οθόνη συνομιλίας υπάρχουν δύο κουμπιά, το ένα ανοίγει την συλλογή εικόνων της συσκευής, το άλλο την κάμερα και όταν αυτά χρησιμοποιηθούν επιτρέπουν την αποστολή εικόνων στον διάλογο.

Για την επιλογή στο αριστερό κάτω μέρος της οθόνης με το εικονίδιο (+) που ανοίγει την συλλογή θα καλείται η μέθοδος `pickImage`. Θα υπάρχει state μεταβλητή που θα καταγράφει την εικόνα που έχει επιλέξει ο χρήστης. Θα γίνεται κλικ στο (+) θα ανοίγει την συλλογή με τις εικόνες και όταν επιλεγεί μία από αυτές θα εμφανίζεται στο κάτω μέρος προσωρινά ώστε να μπορεί να γίνει εισαγωγή κειμένου και έπειτα να αποσταλεί. Η μεταβλητή `tempImageUri` θα είναι το μονοπάτι της εικόνας όπως αυτή είναι αποθηκευμένη στην συσκευή πριν γίνει η αποστολή της στην βάση δεδομένων. Δεν είναι το uri που θα ανέβει στο Firebase αλλά με αυτό που θα γίνει η διαδικασία της αποθήκευσης του στην βάση. Η μέθοδος `pickImage` θα κάνει χρήση της `launchImagePicker` που υπάρχει ήδη από την σελίδα `settings` και το uri που επιστρέφει θα δίνει τιμή στην state μεταβλητή.

Από την συλλογή φωτογραφιών και όταν γίνει επιλογή η εικόνα θα εμφανίζεται σε παράθυρο `pop-up` και όταν πατήσει ο χρήστης αποστολή τότε θα αποστέλλεται στον διάλογο. Εκτός από την επιλογή αποστολή θα υπάρχει και αυτή της ακύρωσης. Για την λειτουργία του `pop-up` θα αξιοποιηθεί το πακέτο `Awesome Alerts` της `React Native` και η εγκατάσταση γίνεται με την εντολή:

`npm i react-native-awesome-alerts`

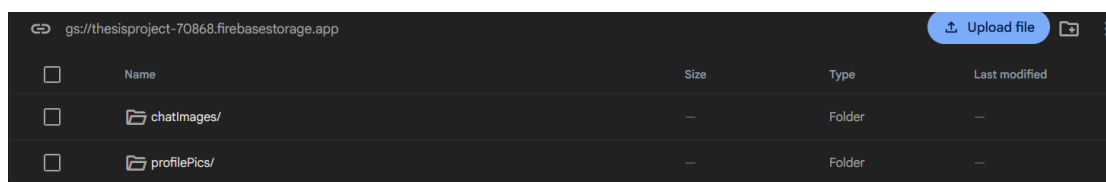
Το `Alert` θα πρέπει να προστεθεί στην σελίδα `ChatScreen` σαν ξεχωριστό στοιχείο. Τα πεδία εξηγούνται στο εγχειρίδιο και έρχονται μαζί με το πακέτο.

Το στοιχείο `AwesomeAlert` θα είναι κρυφό εξ αρχής και θα εμφανίζεται μόλις το `tempImageUri` αποκτήσει κάποια τιμή. Πλέον, μόλις επιλεγεί η εικόνα θα εμφανίζεται στο `pop-up`, με την επιλογή `Send` θα ανεβαίνει στην βάση ενώ αν πατηθεί το `Cancel` η σε μέρος εκτός του παραθύρου θα ακυρώνεται. Η ενέργεια υπάρχει ήδη στο πακέτο με όνομα `onConfirmedPressed` και θα καλεί την μέθοδο `uploadImage`.

`onConfirmedPressed={uploadImage}`

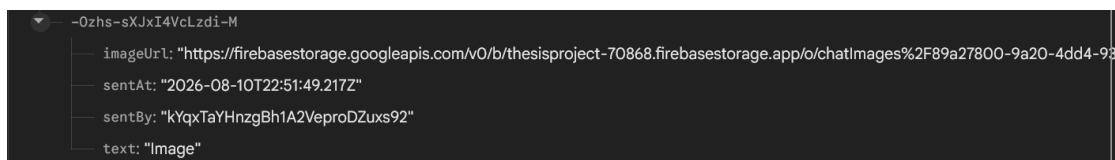
Στην `uploadImage` θα υπάρχει ένδειξη φόρτωσης της διαδικασίας με την state μεταβλητή `isLoading` και θα καλείται η `uploadImageAsync` που τοποθετεί την εικόνα στην βάση και επιστρέφει σύνδεσμο που δείχνει την θέση της στο Firebase. Οι εικόνες αποθηκεύονται στο `Firebase storage`. Οι εικόνες που προέρχονται από διάλογο θα αποθηκεύονται σε νέο φάκελο `chatImages` και θα καθορίζεται από την παράμετρο `isChatImage` που είναι προαιρετική.

Η `uploadImage` θα δημιουργήσει έναν φάκελο `chatImages` στο `storage` του `Firebase` και εκεί θα βρίσκονται μόνο οι εικόνες που στέλνονται από διαλόγους.



Εικόνα 6.6 - Φάκελος εικόνων διαλόγου `chatImages` στο `storage` του `Firebase`.

Το τελευταίο που θα γίνει για την λειτουργία του φορτώματος της εικόνας είναι να πάρουμε την εικόνα που βρίσκεται στο `storage` από το `uploadUrl` και να φτιάξουμε το μήνυμα με αυτό. Υπάρχει ήδη η μέθοδος για μηνύματα τύπου κειμένου αλλά χρειάζεται και για μηνύματα τύπου εικόνων για αυτό στο `chatActions` θα βρίσκεται η `sendImage` η οποία θα κληθεί στην `uploadImage`. Το αποτέλεσμα θα είναι να τοποθετηθεί η εικόνα σαν μήνυμα στην βάση δεδομένων στον κόμβο `messages` για τον συγκεκριμένο διάλογο και θα περιέχει `text` σαν `Image`, το `imageUrl`.



Εικόνα 6.7 - Αποθήκευση εικόνας στο πεδίο imageUrl στον κόμβο messages.

Αφού έγινε η μεθοδολογία για το ανέβασμα των εικόνων στο Realtime Database θα πρέπει να εμφανίζονται και στο παράθυρο του διαλόγου.

Στο Bubble component θα υπάρχει ένα imageUrl που θα περνάει μέσω του ChatScreen (στην εμφάνιση του Bubble) και τα δεδομένα του θα είναι το νέο imageUrl που βρίσκεται σε κάθε μήνυμα αποστολής εικόνων. Μετά θα γίνει χρήση του imageUrl (στο Bubble component).

```

{
  imageUrl &&
  <Image source={{ uri: imageUrl }} style={styles.image} />
}

```

Εάν ο χρήστης επιλέξει απάντηση σε ένα μήνυμα από το μενού μπορεί να το κάνει και με εικόνα από την επιλογή (+) και να διαλέξει από την συλλογή εικόνων. Στο τμήμα της απάντησης δεν θα εμφανίζεται ολόκληρη η εικόνα αλλά η λέξη Image.



Εικόνα 6.8 - Αποστολή εικόνας στο παράθυρο διαλόγου.



Εικόνα 6.9 – Απάντηση με εικόνα σε υφιστάμενο μήνυμα.

6.5.1 Λήψη Εικόνας από Κάμερα

Το κουμπί με το εικονίδιο της κάμερας θα επιτρέπει στον χρήστη να κάνει λήψη φωτογραφιών και επιπλέον θα ανεβαίνουν απευθείας στην βάση δεδομένων. Αυτό το χαρακτηριστικό θα λειτουργεί αποκλειστικά σε πραγματική συσκευή και όχι στον προσομοιωτή.

Στο αρχείο `imagePickerHelper` η μέθοδος `openCamera` θα ανοίγει την εφαρμογή της κάμερας. Θα ζητάει την άδεια του χρήστη για να αποκτήσει πρόσβαση στην κάμερα και στην συνέχεια θα καλείται η `launchCameraAsync`.

```
export const openCamera = async () => {
  const permissionResult = await ImagePicker.requestCameraPermissionsAsync();

  if (permissionResult.granted === false) {
    console.log('No permission to access the camera.');
```

```
    return;
  }

  const result = await ImagePicker.launchCameraAsync({
    mediaTypes: ['images'],
    allowsEditing: true,
    aspect: [1, 1],
    quality: 1
  });

  if (!result.canceled)
    return result.assets[0].uri;
};
```

Η χρήση της θα γίνει στο ChatScreen στην μέθοδο takePhoto που επιστρέφει το uri της εικόνας στην συσκευή και χρησιμοποιείται για να δώσει τιμή στην state μεταβλητή tempImageUri. Η takePhoto θα εκτελείται μόλις ο χρήστης πατήσει το εικονίδιο της κάμερας.

6.6 Επίλογος

Στο παρόν κεφάλαιο παρουσιάστηκε η ανάπτυξη του βασικού μηχανισμού ανταλλαγής μηνυμάτων. Η υλοποίηση επεκτάθηκε πέρα από την απλή αποστολή κειμένου, περιλαμβάνοντας την ανάκτηση και εμφάνιση των μηνυμάτων, τη διαχείριση πρόσθετων ενεργειών πάνω σε αυτά, τη δυνατότητα απάντησης, καθώς και την ανταλλαγή εικόνων μεταξύ των χρηστών.

Αρχικά υλοποιήθηκε η αποστολή μηνυμάτων κειμένου και η αποθήκευσή τους στη βάση δεδομένων. Κάθε μήνυμα συνδέεται με τον αντίστοιχο διάλογο και συνοδεύεται από πληροφορίες που προσδιορίζουν τον αποστολέα και τον χρόνο αποστολής. Παράλληλα, με κάθε νέο μήνυμα ενημερώνονται τα σχετικά στοιχεία της συνομιλίας, επιτρέποντας στην εφαρμογή να διατηρεί πληροφορίες σχετικά με την τελευταία δραστηριότητά της.

Για την αποτελεσματικότερη διαχείριση των δεδομένων πραγματοποιήθηκε η ενημέρωση των μηνυμάτων στο κεντρικό state της εφαρμογής. Κατά την ανάκτησή τους από το Firebase, τα δεδομένα αποθηκεύονται μέσω του αντίστοιχου μηχανισμού διαχείρισης κατάστασης και μπορούν στη συνέχεια να χρησιμοποιηθούν από την οθόνη συνομιλίας. Η αξιοποίηση των listeners του Firebase επιτρέπει παράλληλα την άμεση ενημέρωση της εφαρμογής όταν πραγματοποιείται κάποια μεταβολή στα δεδομένα των μηνυμάτων. Ως αποτέλεσμα, ένα νέο μήνυμα μπορεί να εμφανίζεται άμεσα στις συσκευές χωρίς να απαιτείται χειροκίνητη ανανέωση της συνομιλίας.

Η εμφάνιση των μηνυμάτων προσαρμόστηκε ανάλογα με τον αποστολέα τους, ώστε ο χρήστης να μπορεί να διακρίνει τα εξερχόμενα από τα εισερχόμενα μηνύματα. Παράλληλα, πραγματοποιήθηκε κατάλληλος χειρισμός των περιπτώσεων αποτυχίας αποστολής, μέσω της εμφάνισης προσωρινού μηνύματος σφάλματος στην οθόνη.

Στη συνέχεια, η λειτουργικότητα των μηνυμάτων επεκτάθηκε μέσω της προσθήκης ενός μενού ενεργειών, το οποίο ενεργοποιείται με παρατεταμένη επιλογή ενός μηνύματος. Μέσω του συγκεκριμένου τρόπου παρέχεται η δυνατότητα αντιγραφής του κειμένου, επισήμανσης ενός μηνύματος ως αγαπημένου και απάντησης σε συγκεκριμένο μήνυμα. Η αποθήκευση των αγαπημένων μηνυμάτων πραγματοποιείται ξεχωριστά για κάθε χρήστη, διατηρώντας την πληροφορία προσωπική και επιτρέποντας την ανάκτησή της κατά την επόμενη εκτέλεση της εφαρμογής.

Η λειτουργία απάντησης σε μήνυμα υλοποιήθηκε με τη συσχέτιση του νέου μηνύματος με το μήνυμα στο οποίο αναφέρεται. Μέσω της κατάλληλης αποθήκευσης του αναγνωριστικού του αρχικού μηνύματος και της αντίστοιχης προσαρμογής της διεπαφής, ο χρήστης μπορεί να γνωρίζει σε ποιο

περιεχόμενο απαντά τόσο πριν από την αποστολή όσο και κατά την εμφάνιση της απάντησης στον διάλογο.

Τέλος, η ανταλλαγή περιεχομένου επεκτάθηκε με την προσθήκη υποστήριξης μηνυμάτων εικόνας. Ο χρήστης έχει τη δυνατότητα να επιλέξει εικόνα από την τοπική συλλογή της συσκευής ή να πραγματοποιήσει νέα λήψη μέσω της κάμερας. Πριν από την αποστολή παρέχεται η δυνατότητα προεπισκόπησης και επιβεβαίωσης, ενώ στη συνέχεια η εικόνα αποθηκεύεται στο Firebase Storage και η σχετική πληροφορία περνάει στα δεδομένα του αντίστοιχου μηνύματος. Με αυτόν τον τρόπο, η εφαρμογή υποστηρίζει τόσο την ανταλλαγή κειμένου όσο και περιεχομένου εικόνων.

Συνολικά, με τις λειτουργίες που αναπτύχθηκαν στο συγκεκριμένο κεφάλαιο ολοκληρώνεται ένα σημαντικό μέρος του συστήματος επικοινωνίας της εφαρμογής. Η αποστολή και λήψη μηνυμάτων σε πραγματικό χρόνο, η κεντρική διαχείριση των δεδομένων, η δυνατότητα αλληλεπίδρασης με υπάρχοντα μηνύματα και η υποστήριξη εικόνων συνθέτουν ένα ολοκληρωμένο περιβάλλον ανταλλαγής περιεχομένου.

Κεφάλαιο 7ο: Ομαδικές Συνομιλίες

7.1 Εισαγωγή

Στο παρόν κεφάλαιο παρουσιάζεται η επέκταση της λειτουργικότητας της εφαρμογής μέσω της υλοποίησης ομαδικών συνομιλιών. Έχοντας ήδη αναπτυχθεί η δημιουργία διαλόγων μεταξύ δύο χρηστών, καθώς και οι βασικές λειτουργίες αποστολής και διαχείρισης μηνυμάτων, το επόμενο στάδιο αφορά τη δυνατότητα συμμετοχής περισσότερων χρηστών στον ίδιο διάλογο. Η συγκεκριμένη προσθήκη επιτρέπει τη δημιουργία οργανωμένων ομάδων ανταλλαγής μηνυμάτων.

Αρχικά παρουσιάζεται η προσθήκη της κατάλληλης επιλογής στην κεντρική οθόνη της εφαρμογής, μέσω της οποίας ο χρήστης μπορεί να ξεκινήσει τη διαδικασία δημιουργίας μιας νέας ομαδικής συνομιλίας. Για την υλοποίηση επαναχρησιμοποιείται η ήδη υπάρχουσα οθόνη δημιουργίας νέου διαλόγου, η οποία προσαρμόζεται κατάλληλα ώστε να διαφοροποιεί την απλή συνομιλία μεταξύ δύο χρηστών από την ομαδική συνομιλία. Παράλληλα, προστίθεται η δυνατότητα καθορισμού ονόματος για την ομάδα και εφαρμόζονται οι απαραίτητοι έλεγχοι πριν από την ενεργοποίηση της επιλογής δημιουργίας.

Στη συνέχεια αναλύεται η επιλογή πολλαπλών χρηστών που θα συμμετέχουν στη νέα συνομιλία. Τα αποτελέσματα της αναζήτησης προσαρμόζονται ώστε κάθε χρήστης να μπορεί να επιλεγεί ή να αφαιρεθεί από τη λίστα των συμμετεχόντων. Η κατάσταση των επιλεγμένων χρηστών διατηρείται κατά τη διάρκεια της διαδικασίας, ενώ η οθόνη ενημερώνεται οπτικά με τους επιλεγέντες χρήστες.

Επιπλέον, γίνεται η απεικόνιση των επιλεγμένων συμμετεχόντων σε ξεχωριστό τμήμα της οθόνης. Μέσω της συγκεκριμένης λειτουργίας ο χρήστης έχει τη δυνατότητα να ελέγχει τα άτομα που πρόκειται να συμμετέχουν στην ομάδα και, εφόσον θέλει, να αφαιρεί κάποιο από αυτά πριν από την ολοκλήρωση της διαδικασίας. Η δημιουργία της ομαδικής συνομιλίας επιτρέπεται μόνο εφόσον έχει καθοριστεί όνομα και έχει επιλεγεί ένας αριθμός συμμετεχόντων.

Τέλος, αναλύεται η διαδικασία μετάβασης στην οθόνη της νέας ομαδικής συνομιλίας και η μεταφορά των απαραίτητων δεδομένων, όπως το όνομα και οι συμμετέχοντες. Επίσης, πραγματοποιούνται οι απαιτούμενες προσαρμογές στην παρουσίαση του διαλόγου, ώστε στις ομαδικές συνομιλίες να εμφανίζεται ως τίτλος το όνομα της ομάδας και να προσδιορίζεται ο αποστολέας κάθε μηνύματος. Εξετάζεται επίσης ο έλεγχος ύπαρξης προηγούμενης συνομιλίας κατά την επικοινωνία μεταξύ δύο χρηστών, ώστε να αποφεύγεται η δημιουργία περιττών διπλότυπων διαλόγων.

7.2 Προσθήκη λειτουργιών

Στην κεντρική οθόνη, εκεί που εμφανίζονται όλες οι συνομιλίες που μετέχει ο χρήστης, θα υπάρχει ένα κουμπί που θα δημιουργεί ομαδικούς διαλόγους. Υπάρχει ήδη το κουμπί που ξεκινάει συνομιλία με ένα μόνο άτομο, για τις ομαδικές όμως θα υπάρχει ξεχωριστό.

Θα είναι στοιχείο `TouchableOpacity` διότι θα εκτελείται ενέργεια όταν ενεργοποιηθεί. Δεν θα χρησιμοποιηθεί το στοιχείο `Button` της `React Native` γιατί απεικονίζεται διαφορετικά σε συσκευές `Android` και `iOS`, έτσι αν δημιουργήσουμε το δικό μας θα δείχνει το ίδιο σε όλες τις συσκευές ανεξαρτήτως. Η λειτουργία `onPress` θα ανοίγει νέα οθόνη (`NewChatScreen`), όπως γινόταν και με την δημιουργία διαλόγου με ένα άτομο, ωστόσο θα περνάει διαφορετικά δεδομένα (`isGroupChat`). Όταν γίνει η πλοήγηση στην οθόνη θα ξέρουμε πως πρόκειται για ομαδική ομιλία.

Η οθόνη επαναχρησιμοποιείται, πριν δημιουργούσε διάλογο με ένα άτομο, και θα δίνει την δυνατότητα στον χρήστη να φτιάχνει ομαδικούς διαλόγους. Θα γίνει λήψη της παραμέτρου `isGroupChat` που περνάει στην οθόνη κατά την διάρκεια της πλοήγησης και μετά θα χρησιμοποιηθεί για να ενημερωθεί ο τίτλος. Όσο δημιουργείται συνομιλία από την επιλογή 'New Group' η νέα σελίδα θα έχει τίτλο 'Add Participants'.

Θα υπάρχει πεδίο που θα μπορεί ο χρήστης να δώσει όνομα στην ομαδική, ένα κουμπί πάνω δεξιά που θα την δημιουργεί και θα ενεργοποιείται μόνο όταν έχει οριστεί ένα όνομα. Αν δεν έχει δοθεί θα μένει απενεργοποιημένο.

Από προηγούμενη λειτουργία έχει οριστεί στο navigation το κουμπί 'Close' στο μέρος 'headerLeft' και καταλαμβάνει το αριστερό μέρος του μενού πλοήγησης. Θα γίνει το ίδιο για το 'headerRight' με το νέο κουμπί να εμφανίζεται μόνο όταν πρόκειται για ομαδική συνομιλία (`isGroupChat`), ενώ όταν δεν είναι (ομαδική) αυτό θα παραμένει κρυφό γιατί στην περίπτωση αυτή πατώντας στο όνομα του χρήστη θα δημιουργήσει τον διάλογο χωρίς να χρειάζεται περαιτέρω ενέργεια. Ο τίτλος της νέας επιλογής θα είναι 'Create'. Θα παραμένει απενεργοποιημένο αν δεν υπάρχει τιμή στο πεδίο γιατί θέλουμε το όνομα της συνομιλίας να είναι υποχρεωτικό.

Η state μεταβλητή `chatName` θα περιέχει το όνομα και η `isGroupChatDisabled` θα ενεργοποιεί / απενεργοποιεί το νέο στοιχείο αναλόγως και θα αλλάζει την εμφάνισή του. Για να ενεργοποιηθεί θα συμβαίνει αυτόματα με την εισαγωγή κειμένου στο πεδίο και αρκεί να ενημερωθεί η `setChatName` δηλαδή να δώσει τιμή στο `chatName` που με την σειρά του θα αλλάζει το `isGroupChatDisabled`.

7.3 Προσθήκη χρηστών στην συνομιλία

Μέχρι στιγμής μπορούμε να αναζητήσουμε χρήστη και όταν πατήσουμε πάνω σε αυτόν δημιουργεί τον διάλογο μαζί του. Αντί για αυτό θέλουμε όταν γίνει επιλογή να 'θυμάται' ποιος χρήστης επιλέχθηκε και θα δημιουργεί το chat μόνο όταν πατήσουμε το 'Create'. Κάθε χρήστης είναι ένα `DataItem` στο οποίο θα γίνει μετατροπή σε τύπου 'checkbox' μόνο όταν πρόκειται για group chat και σε αυτό το component θα προστεθεί η λειτουργία να φαίνεται ένα check στα δεξιά του. Στο `DataItem` αν είναι τύπου checkbox θα γίνει εισαγωγή εικονιδίου και με βάση το `isChecked` που ελέγχει αν το στοιχείο είναι επιλεγμένο θα έχει διαφορετική μορφοποίηση. Κατά την αναζήτηση χρήστη λοιπόν θα φαίνεται ένας άδειος κύκλος σαν ένδειξη πως μπορούμε να το επιλέξουμε και όταν πράγματι γίνει κλικ το `isChecked` θα αλλάξει σε true και θα εφαρμόσει τις επιπλέον επιλογές μορφοποίησης.

Θα διατηρείται πίνακας με τους χρήστες που έχουν επιλεγθεί (`selectedUsers`), όταν γίνει κλικ σε έναν θα τον προσθέτει στην διάταξη, εάν γίνει ξανά θα τον αφαιρεί. Μία ακόμη παράμετρος που θα ενεργοποιεί / απενεργοποιεί το κουμπί 'Create' είναι να έχουν επιλεγεί χρήστες.

Η μέθοδος `userPressed` ενεργοποιείται στην περίπτωση που πατηθεί το εικονίδιο του χρήστη και μας μεταφέρει σε άλλη οθόνη όταν δεν πρόκειται για group chat. Όταν όμως φτιάχνουμε ομαδικό chat η συμπεριφορά θα είναι να γίνεται check του checkbox και το αντίθετο. Στην ουσία θα ενημερώνεται ο πίνακας `selectedUsers` προσθέτοντας ή αφαιρώντας τον χρήστη.

```
if (isGroupChat) {
  const newSelectedUsers = selectedUsers.includes(userId) ?
    selectedUsers.filter(id => id !== userId) :
    selectedUsers.concat(userId);

  // Update the selectedUsers state variable
  setSelectedUsers(newSelectedUsers);
}
```

Αφού υπάρχουν οι επιλεγμένοι χρήστες απομένει να ενημερωθούν τα checkboxes για να δείχνουν τις όποιες αλλαγές. Η συνθήκη θα προστεθεί στο DataItem και θα ελέγχει αν ο πίνακας selectedUsers περιέχει το συγκεκριμένο userId όπου και θα είναι checked. Με την συγκεκριμένη προσθήκη η επιλογή Create θα ενεργοποιείται όταν έχουν επιλεγεί χρήστες και ταυτόχρονα έχει δοθεί όνομα στο group chat. Εάν ένα από τα δύο λείπει αυτόματα θα γίνεται ανενεργό.

7.3.1 Εμφάνιση επιλεγμένων χρηστών

Με την λίστα επιλεγμένων χρηστών να έχει οριστεί (selectedUsers) θα γίνει η εμφάνισή τους για να φαίνεται ποιοι έχουν επιλεγεί ή ακόμη και να αφαιρεθούν αν δεν θέλουμε το συγκεκριμένο άτομο στον διάλογο.

Στην οθόνη (NewChatScreen) στο πεδίο που αναγράφεται το όνομα του chat θα υπάρχει ξεχωριστό View και σε αυτό ένα FlatList που θα εμφανίζει την επιλεγμένη λίστα με τους χρήστες. Σε κάθε επανάληψη του FlatList θα εμφανίζεται ένα ProfileImage component που θα είναι η εικόνα προφίλ του κάθε χρήστη και τα δεδομένα θα προέρχονται από την μεταβλητή storedUsers που έχει όλους τους χρήστες που βρίσκονται στο state της εφαρμογής. Οι storedUsers ενημερώνονται σε κάθε νέα αναζήτηση χρήστη ο οποίος και προστίθεται.

Ένας χρήστης μπορεί να αφαιρεθεί πατώντας το checkbox ή ολόκληρο το DataItem. Θα γίνεται το ίδιο και από την λίστα καθώς θα υπάρχει εικονίδιο που θα δείχνει πως επιτρέπεται τέτοια ενέργεια. Κάθε φορά που πατάμε σε χρήστη της λίστας το onPress θα εκτελεί την userPressed (ProfileImage) στο σημείο του κώδικα που βγάζει τον επιλεγμένο χρήστη από τον πίνακα selectedUsers.

Η ένδειξη θα είναι εικονίδιο στην εικόνα προφίλ του χρήστη και θα γίνει περνώντας το property showRemoveButton στο ProfileImage στην οθόνη NewChatScreen. Σε αυτό το component θα βρίσκεται στην μεταβλητή showRemoveButton και βάσει αυτού θα εμφανίζεται το εικονίδιο με την ένδειξη (x).

```
const showRemoveButton = props.showRemoveButton && props.showRemoveButton === true;

{
  showRemoveButton && !isLoading &&
  <View style={styles.removeIconContainer}>
    <FontAwesome name='close' size={15} color={colors.alabasterWhite} />
  </View>
}
```

7.4 Λειτουργία Επιλογής Create

Όταν ο χρήστης πατήσει στο κουμπί Create θα μεταφέρεται σε νέο παράθυρο διαλόγου. Στο headerRight που υπάρχει το κουμπί θα γίνεται η πλοήγηση στην κύρια οθόνη ChatListScreen και θα μεταφέρει σαν δεδομένα τους επιλεγμένους χρήστες του διαλόγου (selectedUsers) και το όνομα του chat (chatName) και εκεί η ανάκτησή τους θα γίνεται στις μεταβλητές selectedUserList και chatName αντίστοιχα για να μπορούν να χρησιμοποιηθούν. Το useEffect στην οθόνη ChatListScreen χειρίζεται την πλοήγηση στον διάλογο και θα οριστούν οι χρήστες στην chatUsers μεταβλητή, υποθέτουμε πως θα είναι selectedUserList ενώ αν δεν υπάρχει η λίστα θα είναι ένας επιλεγμένος χρήστης δηλαδή πρόκειται για διάλογο με ένα άτομο. Τα δεδομένα που θα περάσουν στην οθόνη ChatScreen είναι οι users που αφορά τους χρήστες που συμμετέχουν στον διάλογο και αν είναι ομαδική συνομιλία όπως και το όνομα του chat αν αυτό υπάρχει.

```

navigationProps = {
  newChatData: {
    users: chatUsers,
    isGroupChat: selectedUserList !== undefined,
    // chatName
  }
};

if (chatName) navigationProps.newChatData.chatName = chatName;

```

Όταν έχουμε ήδη ξεκινήσει συνομιλία με έναν χρήστη και αναζητήσουμε ξανά το ίδιο όνομα θα δημιουργήσει νέο διάλογο εκ νέου. Το επιθυμητό είναι να ανοίγει την υπάρχουσα συνομιλία και όχι καινούργια. Για να λυθεί το πρόβλημα στο ChatListScreen θα γίνει έλεγχος αν στις συνομιλίες που παίρνει μέρος ο χρήστης (userChats) υπάρχει μία με τον χρήστη που έχει επιλέξει κατά την αναζήτηση. Αν λοιπόν το chat υπάρχει ήδη θα χρησιμοποιήσουμε τα δεδομένα του και θα σταλεί στην νέα οθόνη το chatId της συνομιλίας μέσω του navigationProps .

```

chatData = userChats.find(chat => !chat.isGroupChat && chat.users.includes(selectedUser));

if (chatData) {
  navigationProps = { chatId: chatData.key };
}

```

Το αποτέλεσμα θα είναι ο χρήστης να μεταφερθεί σε υπάρχουν διάλογο στην οθόνη (ChatScreen) όπου και αξιοποιείται το chatId που μεταφέρθηκε με τα δεδομένα της πλοήγησης.

Όταν πρόκειται για ομαδική δεν θα εμφανίζεται το όνομα κάποιου χρήστη σαν τίτλος αλλά το όνομα της συνομιλίας. Σε διάλογο με ένα άτομο θα συνεχίσει να χρησιμοποιείται το όνομα αυτού σαν τίτλος. Επίσης, κάθε μήνυμα της ομαδικής θα υποδεικνύει το όνομα του χρήστη που το έστειλε ενώ όταν συνομιλούμε με ένα άτομο δεν χρειάζεται κάτι τέτοιο διότι είναι προφανές. Στο ChatScreen που δημιουργούνται τα μηνύματα (Bubble) θα οριστεί χαρακτηριστικό 'name'

```

name={
  !chatData.isGroupChat || isOwnMessage ? undefined : name
}

const sender = message.sentBy && storedUsers[message.sentBy];
// sentBy contains the userId not the user data, so we need
// to retrieve that data from the storedUsers
const name = sender && `${sender.firstName} ${sender.lastName}`;

```

7.5 Επίλογος

Στο παρόν κεφάλαιο παρουσιάστηκε η επέκταση του συστήματος επικοινωνίας της εφαρμογής με την προσθήκη της δυνατότητας δημιουργίας και διαχείρισης ομαδικών συνομιλιών. Η συγκεκριμένη λειτουργικότητα βασίστηκε σε λειτουργίες που είχαν ήδη αναπτυχθεί για τις συνομιλίες μεταξύ δύο χρηστών, οι οποίοι προσαρμόστηκαν κατάλληλα ώστε να υποστηρίζεται η συμμετοχή πολλαπλών χρηστών στον ίδιο διάλογο.

Αρχικά προστέθηκε στην κεντρική οθόνη η δυνατότητα έναρξης μιας νέας ομαδικής συνομιλίας. Για τον σκοπό αυτό αξιοποιήθηκε η οθόνη δημιουργίας νέου διαλόγου, αποφεύγοντας τη δημιουργία μιας ανεξάρτητης οθόνης με παρόμοια λειτουργικότητα. Μέσω της κατάλληλης παραμέτρου πλοήγησης γίνεται η διάκριση μεταξύ προσωπικής και ομαδικής συνομιλίας και η αντίστοιχη προσαρμογή της οθόνης.

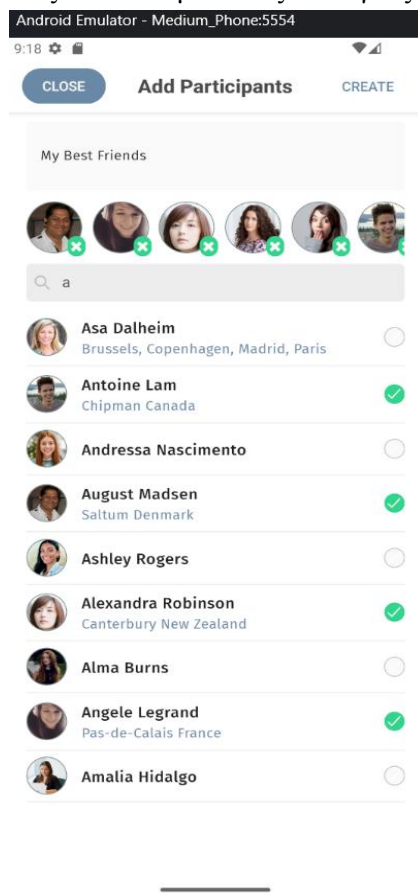
Στη συνέχεια υλοποιήθηκε η επιλογή των συμμετεχόντων της ομάδας. Τα αποτελέσματα αναζήτησης προσαρμόστηκαν ώστε να παρέχουν ένδειξη για την κατάσταση επιλογής κάθε χρήστη. Παράλληλα, δημιουργήθηκε κατάλληλο τμήμα στην οθόνη στο οποίο εμφανίζονται οι συμμετέχοντες που έχουν ήδη επιλεγεί, παρέχοντας τη δυνατότητα αφαίρεσής τους πριν από την ολοκλήρωση της δημιουργίας της ομάδας.

Για την ολοκλήρωση της διαδικασίας προστέθηκε η επιλογή δημιουργίας της ομαδικής συνομιλίας, η οποία ενεργοποιείται μόνο όταν πληρούνται οι απαραίτητες προϋποθέσεις. Ειδικότερα, απαιτείται η εισαγωγή ονόματος για τη συνομιλία και η επιλογή συμμετεχόντων. Με αυτόν τον τρόπο αποτρέπεται η δημιουργία περιττών ομαδικών διαλόγων και διασφαλίζεται ότι τα βασικά δεδομένα της ομάδας έχουν καθοριστεί πριν από τη μετάβαση στην οθόνη συνομιλίας.

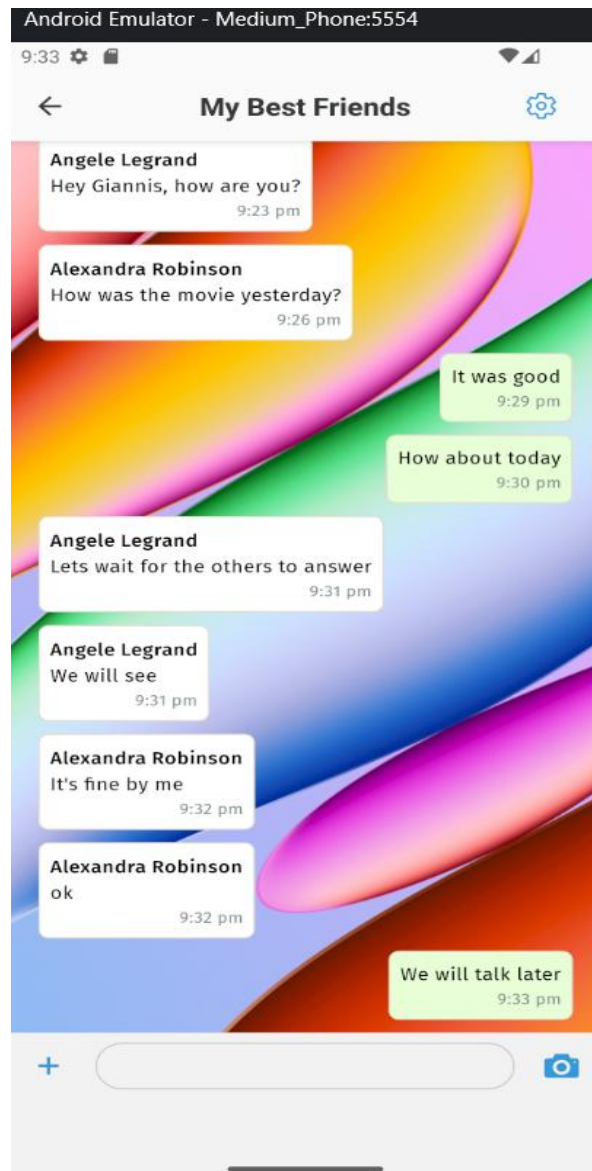
Ακόμη, προσαρμόστηκε η διαδικασία πλοήγησης και μεταφοράς δεδομένων, ώστε η οθόνη συνομιλίας να λαμβάνει τις πληροφορίες που απαιτούνται για τη διαχείριση μιας ομάδας. Στις ομαδικές συνομιλίες χρησιμοποιείται ως τίτλος το όνομα που έχει οριστεί κατά τη δημιουργία τους, ενώ στα επιμέρους μηνύματα εμφανίζεται το όνομα του αποστολέα, διευκολύνοντας τη διάκριση μεταξύ των διαφορετικών συμμετεχόντων.

Τέλος, προβλέφθηκε η περίπτωση κατά την οποία ο χρήστης επιχειρεί να ξεκινήσει εκ νέου προσωπική συνομιλία με άτομο με το οποίο υπάρχει ήδη ενεργός διάλογος. Μέσω ελέγχου των υφιστάμενων συνομιλιών εντοπίζεται η αντίστοιχη καταχώριση και χρησιμοποιείται το υπάρχον αναγνωριστικό της, αντί να δημιουργείται ένας νέος διάλογος με τους ίδιους συμμετέχοντες.

Η προσθήκη των ομαδικών συνομιλιών επεκτείνει ουσιαστικά τις δυνατότητες επικοινωνίας της εφαρμογής και διαμορφώνεται ένα περισσότερο ολοκληρωμένο σύστημα ανταλλαγής μηνυμάτων, το οποίο υποστηρίζει τόσο προσωπικούς όσο και ομαδικούς διαλόγους.



Εικόνα 7.1 - Δημιουργία νέας ομαδικής συνομιλίας με επιλογή χρηστών και την κατάλληλη ένδειξη.



Εικόνα 7.2 - Οθόνη ομαδικής συνομιλίας με τα ονόματα ορισμένων συμμετεχόντων.

Κεφάλαιο 8ο: Ρυθμίσεις Συνομιλιών

8.1 Εισαγωγή

Στο παρόν κεφάλαιο παρουσιάζεται η ανάπτυξη των λειτουργιών διαχείρισης των συνομιλιών της εφαρμογής. Μετά την υλοποίηση των προσωπικών και ομαδικών διαλόγων, καθώς και των βασικών μηχανισμών ανταλλαγής μηνυμάτων, ακολουθεί η προσθήκη δυνατοτήτων μέσω των οποίων ο χρήστης μπορεί να διαχειρίζεται τις πληροφορίες και τους συμμετέχοντες μιας συνομιλίας. Οι λειτουργίες αυτές διαφοροποιούνται ανάλογα με το αν πρόκειται για προσωπικό ή ομαδικό διάλογο.

Αρχικά, εξετάζεται η περίπτωση των προσωπικών συνομιλιών και η δημιουργία ξεχωριστής καρτέλας για την παρουσίαση των στοιχείων του συνομιλητή. Μέσω της συγκεκριμένης οθόνης παρέχεται πρόσβαση σε πληροφορίες όπως το όνομα, η εικόνα προφίλ και τα πρόσθετα στοιχεία του χρήστη. Παράλληλα, υλοποιείται η δυνατότητα εντοπισμού των κοινών ομαδικών συνομιλιών στις οποίες συμμετέχουν οι δύο χρήστες και παρέχεται η δυνατότητα άμεσης μετάβασης σε αυτές.

Στη συνέχεια παρουσιάζονται οι ρυθμίσεις των ομαδικών συνομιλιών, οι οποίες περιλαμβάνουν τη διαχείριση βασικών χαρακτηριστικών της ομάδας. Συγκεκριμένα, υλοποιείται η δυνατότητα αλλαγής της εικόνας και του ονόματος της συνομιλίας, με τις αντίστοιχες μεταβολές να αποθηκεύονται στη βάση δεδομένων και να γίνονται διαθέσιμες στα υπόλοιπα σημεία της εφαρμογής. Για την επεξεργασία του ονόματος αξιοποιείται η υφιστάμενη λογική διαχείρισης και επικύρωσης δεδομένων των φορμών, ακολουθώντας παρόμοια προσέγγιση με προηγούμενες λειτουργίες της εφαρμογής.

Παρουσιάζεται η διαδικασία εμφάνισης των χρηστών που αποτελούν μέλη της ομάδας, η μετάβαση στις προσωπικές καρτέλες τους, καθώς και η δυνατότητα αφαίρεσης ενός συμμετέχοντα. Παράλληλα, υλοποιείται η δυνατότητα οικειοθελούς αποχώρησης του συνδεδεμένου χρήστη από την ομάδα. Οι συγκεκριμένες ενέργειες συνοδεύονται από τις απαραίτητες ενημερώσεις στη βάση δεδομένων.

Επιπλέον, αντιμετωπίζεται η περίπτωση ομαδικών συνομιλιών με μεγάλο αριθμό συμμετεχόντων. Για τον σκοπό αυτό δημιουργείται ξεχωριστή οθόνη προβολής δεδομένων, μέσω της οποίας είναι δυνατή η παρουσίαση της πλήρους λίστας των χρηστών. Η ίδια οθόνη σχεδιάζεται με δυνατότητα επαναχρησιμοποίησης και για διαφορετικούς τύπους δεδομένων της εφαρμογής.

Στην συνέχεια αναλύεται η διαδικασία προσθήκης νέων χρηστών σε μια ήδη υπάρχουσα ομαδική συνομιλία. Κατά την αναζήτηση αποκλείονται οι χρήστες που αποτελούν ήδη μέλη της ομάδας, ενώ μετά την επιλογή των νέων συμμετεχόντων ενημερώνονται οι αντίστοιχες εγγραφές της βάσης δεδομένων. Ακόμη, δημιουργούνται μηνύματα μέσα στη συνομιλία, ώστε οι συμμετέχοντες να πληροφορούνται για τις μεταβολές στη σύνθεση της ομάδας.

Τέλος, παρουσιάζεται η δυνατότητα συγκεντρωτικής προβολής των μηνυμάτων που έχουν επισημανθεί ως αγαπημένα. Η λειτουργία παρέχεται τόσο για μια συγκεκριμένη ομαδική συνομιλία όσο και μέσω των προσωπικών ρυθμίσεων του χρήστη, όπου μπορούν να συγκεντρωθούν αγαπημένα μηνύματα από διαφορετικούς διαλόγους.

8.2 Ρυθμίσεις Προσωπικών Συνομιλιών

Θα υπάρχουν δύο εκδοχές σχετικά με τις ρυθμίσεις στις συνομιλίες. Η πρώτη περίπτωση θα είναι σε ομαδική συνομιλία όπου θα υπάρχει ένα κουμπί και θα μεταφέρει τον χρήστη στις ρυθμίσεις. Η δεύτερη θα αφορά την συνομιλία με ένα άτομο όπου θα υπάρχει το ίδιο κουμπί αλλά θα μεταβαίνει σε καρτέλα επικοινωνίας με τα στοιχεία του.

Θα γίνει πρώτα η περίπτωση για συνομιλία με ένα άτομο και έπειτα για τις ομαδικές.

Αρχικά, θα υπάρχει το κουμπί που θα μεταφέρει τον χρήστη σε αυτή την οθόνη. Στον τίτλο του navigation (ChatScreen) στο useEffect θα προστεθεί headerRight που θα το εισάγει στο δεξί μέρος της κεφαλίδας και θα επιστρέφει στοιχείο HeaderButtons. Ορίζεται custom component με όνομα CustomHeaderButton που εμφανίζει το στοιχείο. Το onPress θα χειρίζεται την μεταφορά στην αντίστοιχη οθόνη αφού γίνει έλεγχος (isGroupChat) αν πρόκειται για ομαδική ή ατομική συνομιλία. Στην προκειμένη που αφορά την ατομική θα μεταφέρεται στην οθόνη με όνομα Contact και θα είναι νέο αρχείο με όνομα ContactScreen. Το ContactScreen είναι η προσωπική καρτέλα του χρήστη με τον οποίο συνομιλούμε. Πρέπει να προστεθεί στο κύριο μέρος πλοήγησης της εφαρμογής (MainNavigator) στο Stack Navigator όπου και θα είναι νέο στοιχείο (Stack Screen) με όνομα Contact.

Στην οθόνη ContactScreen θα ανακτάται ο αντίστοιχος χρήστης από τους αποθηκευμένους στο state της εφαρμογής (storedUsers) και θα εμφανίζονται η εικόνα προφίλ του, το όνομά του αλλά για να γίνει αυτό χρειάζονται τα δεδομένα του. Απαραίτητο είναι και το id του χρήστη που περνάει στην οθόνη κατά την πλοήγηση όπου ορίζονται το όνομα της οθόνης και μετά τα δεδομένα (uid). Η μεταβλητή chatUsers περιλαμβάνει τους χρήστες του διαλόγου και θα ψάξουμε αυτόν που δεν έχει το δικό μας id. Εφόσον βρισκόμαστε σε αυτό το σημείο σημαίνει πως δεν είναι ομαδική συνομιλία που σημαίνει πως υπάρχουν μόνο δύο χρήστες, εμείς και το άλλο πρόσωπο.

```
props.navigation.navigate('Contact', {
  uid: chatUsers.find(uid => uid !== userData.userId)
})
```

Με το id του άλλου χρήστη διαθέσιμο (uid) μπορεί πλέον να ανακτηθεί στην σελίδα ContactScreen. Από τους αποθηκευμένους χρήστες στο state (storedUsers) θα ψάξουμε αυτόν βάσει του παραπάνω id, στην σελίδα του οποίου έχουμε μεταφερθεί.

```
// currentUser is the user's page we're currently on
const currentUser = storedUsers[props.route.params.uid];
// We called it uid when we navigated from the ChatScreen
```

Θα προστεθεί το component ProfileImage στην σελίδα για την φωτογραφία του χρήστη και το όνομά του χρησιμοποιώντας το PageTitle component. Το μέρος 'about' με προσωπικές πληροφορίες θα εμφανίζεται εφόσον υπάρχει.

```
<View style={styles.topContainer}>
  <ProfileImage
    uri={currentUser.profilePicture}
    size={80}
    style={{ marginBottom: 20 }}
  />

  <PageTitle text={` ${currentUser.firstName} ${currentUser.lastName}`} />
  {
    currentUser.about &&
    <Text style={styles.about} numberOfLines={2}>{currentUser.about}</Text>
  }
</View>
```

Ένα άλλο χαρακτηριστικό που θα υπάρχει στην οθόνη είναι οι κοινοί διάλογοι του χρήστη. Θα είναι οι συνομιλίες που λαμβάνουμε εμείς μέρος και ταυτόχρονα μετέχει και ο άλλος χρήστης. Για να υλοποιηθεί χρειάζονται δύο πράγματα, οι συνομιλίες που συμμετέχουμε εμείς οι ίδιοι, τις συνομιλίες του άλλου χρήστη και τέλος θα γίνει διασταύρωσή τους για κοινά στοιχεία.

Οι δικοί μας διάλογοι βρίσκονται στο state της εφαρμογής στην μεταβλητή storedChats. Αφού βρεθούν και αυτοί του άλλου χρήστη θα γίνει η αποθήκευσή τους στην state μεταβλητή commonChats.

Η μέθοδος getUserChats (userActions) θα επιστρέφει τις συνομιλίες του χρήστη όπου και θα γίνεται αναφορά στον κόμβο userChats του Firebase. Μπορεί να χρησιμοποιηθεί πλέον στην οθόνη ContactScreen και επειδή φορτώνει δεδομένα σαν επιπλέον ενέργεια μπορεί να οριστεί σε ένα useEffect. Γίνεται έλεγχος αν για κάθε ένα id διαλόγου του άλλου χρήστη υπάρχει το αντίστοιχο id στους δικούς μας αποθηκευμένους διαλόγους (storedChats) και η συνθήκη να είναι ομαδικός διάλογος (isGroupChat) γιατί δεν θέλουμε να συμπεριλαμβάνεται το τρέχον προσωπικό chat. Το αποτέλεσμα θα είναι ένας πίνακας με chat Id's και θα δώσει τιμή στην state μεταβλητή commonChats.

```
const currentUserChats = await getUserChats(currentUser.userId);
setCommonChats(Object.values(currentUserChats)
  .filter(cid => storedChats[cid] && storedChats[cid].isGroupChat));
```

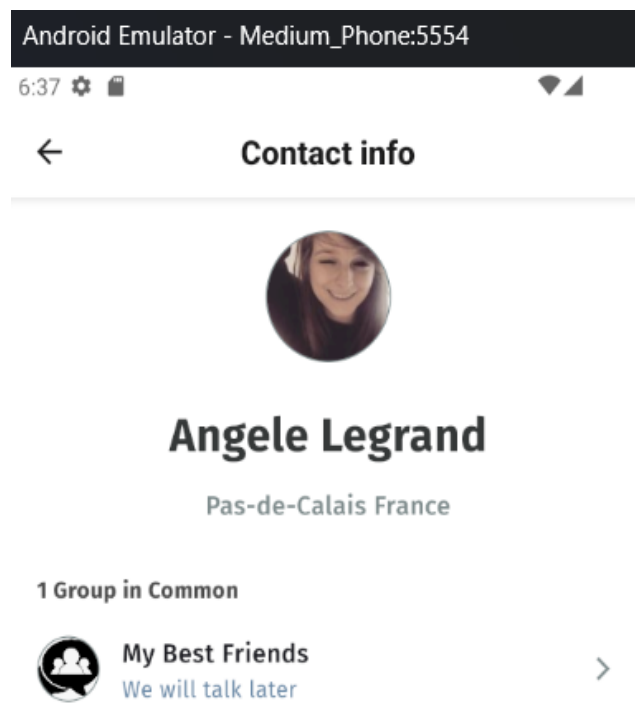
Πλέον η μεταβλητή commonChats έχει ενημερωθεί και περιέχει τις κοινές συνομιλίες που συμμετέχουμε εμείς και ο άλλος χρήστης. Με αυτά τα δεδομένα θα γίνει η εμφάνιση των διαλόγων. Αν υπάρχουν κοινές συνομιλίες θα προστεθούν δύο μέρη, το ένα μέρος θα είναι ο τίτλος (Common Chats) και το άλλο μέρος τα στοιχεία των ίδιων των chats. Θα βρίσκονται μέσα σε React Fragments που επιτρέπουν την επιστροφή πολλαπλών στοιχείων. Χωρίς αυτά επιτρέπεται η επιστροφή ενός parent στοιχείου και τα δύο στοιχεία θα πρέπει να βρίσκονται μέσα σε ένα View (parent).

Ο χρήστης θα μπορεί να πατάει πάνω στους διαλόγους που εμφανίζονται και θα μεταφέρεται σε αυτούς. Θα υπάρχει ένδειξη που θα του δείχνει αυτή την ενέργεια και θα γίνεται περνώντας το property (type = 'link') στο DataItem component. Στο ίδιο component θα χειρίζεται σαν διαφορετική περίπτωση τύπου 'link' και θα εμφανίζει εικονίδιο στο δεξί μέρος του διαλόγου.

```
{
  type === 'link' &&
  <View>
    <Icons name="chevron-forward-sharp" size={20} color={colors.grey} />
  </View>
}
```

Η λειτουργία της πλοήγησης στην αντίστοιχη συνομιλία θα γίνεται από το onPress στην οθόνη ChatScreen και θα μεταφέρει το id του υπάρχοντος chat (chatId) το οποίο και χειρίζεται η οθόνη προορισμού. Θα αξιοποιηθεί η μέθοδος push του navigation που κάνει 'push' νέα οθόνη στην στοίβα του Stack Navigation.

```
{
  commonChats.map(cid => {
    const chatData = storedChats[cid];
    return <DataItem
      key={cid}
      title={chatData.chatName}
      subTitle={chatData.latestMessageText}
      type='link'
      onPress={() => props.navigation.push('ChatScreen', {
        chatId: cid
      })}
      image={chatData.chatImage}
    />
  })
}
```



Εικόνα 8.1 - Καρτέλα στοιχείων χρήστη μέσα από τις ρυθμίσεις συνομιλίας.

8.3 Ρυθμίσεις Ομαδικών Συνομιλιών

Σε οθόνη ομαδικής συνομιλίας θα προστεθούν οι λειτουργίες όταν πατηθεί η επιλογή των ρυθμίσεων. Αυτές θα είναι στην οθόνη με όνομα ChatSettingsScreen. Η πλοήγηση θα γίνεται από την οθόνη ChatScreen, στο σημείο που δημιουργείται το εικονίδιο, και συγκεκριμένα στην περίπτωση που είναι ομαδικός διάλογος (isGroupChat) ενώ θα μεταφέρει μαζί και το id του διαλόγου.

```
props.navigation.navigate('ChatSettings', { chatId })
```

Με το chatId που περνάει στην οθόνη ChatSettings θα το χρησιμοποιήσουμε για την ανάκτηση των δεδομένων του chat. Αυτά βρίσκονται στο state της εφαρμογής (chatData) αλλά θέλουμε για το συγκεκριμένο chatId.

```
const chatId = props.route.params.chatId;
// Get the chats data from the state, but only for the chatId we've received
const chatData = useSelector(state => state.chats.chatsData[chatId] || {});
```

Το component PageTitle θα εμφανίζει τον τίτλο της οθόνης με όνομα 'Chat Settings' και το ProfileImage θα κάνει το ίδιο για την φωτογραφία της ομαδικής η οποία θα μπορεί να αλλάζει.

Στο ProfileImage θα περνάει το chatId και userId, μέσα σε αυτό το component θα εξάγονται και θα χρησιμοποιούνται στην μέθοδο pickImage, συγκεκριμένα στην uploadImageAsync που ανεβάζει την εικόνα στην βάση δεδομένων ανάλογα αν πρόκειται για εικόνα ομαδικής συνομιλίας ή ατομικής (isChatImage = true). Από την ομαδική που βρίσκόμαστε και ανεβάσουμε φωτογραφία αυτή θα αποθηκεύεται στον κόμβο chatImages. Θα γίνει μετατροπή στην μέθοδο uploadImageAsync για να χειρίζεται και αυτή την περίπτωση, όπου αν περάσει chatId θα είναι αληθής η συνθήκη (chatId !== undefined).

Όταν είναι ομαδική εικόνα θα ενημερώνονται τα δεδομένα της συνομιλίας στην βάση δεδομένων. Η μέθοδος updateChatData (chatActions) θα χειρίζεται την ενημέρωση και θα δέχεται σαν παραμέτρους το id του chat, τον χρήστη που έκανε την ενημέρωση και τα δεδομένα που θα σταλούν. Η αναφορά θα είναι στον κόμβο chats στο αντίστοιχο chatId.

```
if (chatId) { // handles if it is a chat image
  await updateChatData(chatId, userId, { chatImage: uploadUrl });
}
```

Αφού ανέβει στο Firebase θα πρέπει να χρησιμοποιηθεί το νέο πεδίο (chatImage) της εικόνας στην βάση για να εμφανιστεί στην οθόνη (ChatSettingsScreen) περνώντας το uri στο ProfileImage. Η φωτογραφία της ομαδικής θα πρέπει να ενημερωθεί στην κύρια οθόνη που φαίνονται όλοι οι διάλογοι (ChatListScreen) καθώς και στην καρτέλα στοιχείων του χρήστη (ContactScreen) στο αντίστοιχο DataItem component και να περαστεί το νέο πεδίο chatImage.

```
if (isGroupChat) {
  title = chatData.chatName;
  image = chatData.chatImage;
}
```

8.3.1 Ενημέρωση Ονόματος Ομαδικής Συνομιλίας

Οι χρήστες θα μπορούν να αλλάζουν το όνομα της συνομιλίας. Για το πεδίο που θα γίνεται η εισαγωγή ονόματος θα χρησιμοποιηθεί το Input component όπως και σε προηγούμενα μέρη της εφαρμογής. Όπως και στην οθόνη ρυθμίσεων του προσωπικού μας προφίλ (SettingsScreen) έτσι και τώρα θα είναι η ίδια λογική όπου θα υπάρχει ένας reducer που αναλαμβάνει το state της φόρμας.

Αρχικά, θα προστεθεί η αρχική κατάσταση του πεδίου (initialState) με το τρέχον όνομα διαλόγου (chatName).

```
const initialState = {
  inputValues: { chatName: chatData.chatName },
  inputValidities: { chatName: undefined },
  formIsValid: false
};
```

Ο reducer (formReducer) που χειρίζεται τις ενημερωμένες τιμές, αξιολογεί αν είναι έγκυρες (valid) ή όχι και τέλος επιστρέφει το νέο state με αυτές τις τιμές. Η μέθοδος inputChangedHandler καλείται κάθε φορά που αλλάζει το πεδίο εισαγωγής (Input) και καταγράφει αν η τιμή που έχουμε εισάγει είναι έγκυρη ενώ ο έλεγχος γίνεται μέσω της validateInput. Θα προστεθεί στο Input component και στην συνέχεια θα ενεργοποιεί την ιδιότητα onChangeText του στοιχείου TextInput, δηλαδή κάθε φορά που αλλάζει το κείμενο.

Για να αλλάξει το όνομα του διαλόγου και να αποθηκευτεί, ο χρήστης θα πρέπει να πατήσει την επιλογή 'Save changes'. Θα είναι τύπου SubmitButton component και θα εκτελεί την μέθοδο saveHandler που μέσω της updateChatData θα αλλάζει τα δεδομένα στην βάση, στην συγκεκριμένη περίπτωση το chatName. Σε περίπτωση επιτυχούς ενημέρωσης θα υπάρχει μήνυμα και θα εξαφανίζεται μετά από δύο δευτερόλεπτα.

Το κουμπί 'Save changes' θα εμφανίζεται μόνο όταν υπάρχουν αλλαγές στο όνομα του διαλόγου. Όταν δεν υπάρχουν αλλαγές δεν θέλουμε να φαίνεται στον χρήστη και αυτό θα το καθορίζει η μέθοδος hasChanges που ελέγχει αν το chatName έχει μεταβληθεί.

8.3.2 Εμφάνιση Συμμετεχόντων Συνομιλίας

Θα εμφανίζονται οι συμμετέχοντες του διαλόγου, θα υπάρχει η δυνατότητα αφαίρεσης κάποιου όπως και ένα κουμπί που θα δίνει την δυνατότητα προσθήκης επιπλέον χρηστών.

Θα είναι ένα μικρό τμήμα στην σελίδα και θα βρίσκεται σε στοιχείο View όπου θα αναγράφεται ο αριθμός των συμμετεχόντων. Το κουμπί θα είναι τύπου DataItem component που θα δέχεται ένα icon και θα αναπαριστά την ένδειξη συν (+). Το icon θα πρέπει να προστεθεί στο DataItem με την κατάλληλη μορφοποίηση ενώ το component εξυπηρετεί την λειτουργία ενός κουμπιού (type = button).

Για την εμφάνιση των συμμετεχόντων θα γίνει από τα δεδομένα του διαλόγου (chatData), συγκεκριμένα από τον πίνακα users που περιέχει τα id των ατόμων και από τους αποθηκευμένους χρήστες του state (storedUsers). Για κάθε ένα χρήστη του πίνακα users εφαρμόζεται η μέθοδος map και θα επιστρέψει ένα στοιχείο τύπου DataItem στο οποίο θα υπάρχει ένδειξη (link) που θα μεταφέρει στην καρτέλα επικοινωνίας με τα στοιχεία του κάθε ατόμου. Η πλοήγηση στην καρτέλα θα εκτελείται από το onPress στην οθόνη ContactScreen και θα μεταφέρει δεδομένα το id του κάθε χρήστη (uid). Θα γίνεται για όλους τους συμμετέχοντες εκτός από εμάς.

```

{
  chatData.users.slice(0, 4).map(uid => {
    const currentUser = storedUsers[uid];
    return <DataItem
      key={uid}
      image={currentUser.profilePicture}
      title={` ${currentUser.firstName} ${currentUser.lastName}`}
      subTitle={currentUser.about}
      type={uid !== userData.userId && 'link'}
      onPress={() => uid !== userData.userId &&
        props.navigation.navigate('Contact', { uid, chatId })
      }
    />
  })
}

```

8.3.3 Αφαίρεση Χρήστη από Συνομιλία

Με την πλοήγηση στην καρτέλα στοιχείων του χρήστη θα προστεθεί επιλογή στην οθόνη που θα μπορεί να τον αφαιρεί από την συνομιλία. Θα είναι ένα κουμπί που θα αναγράφει 'Remove from chat' και θα εκτελεί την παραπάνω ενέργεια. Η επιλογή θα φαίνεται μόνο στις ρυθμίσεις των ομαδικών συνομιλιών και δεν θα υπάρχει στους διαλόγους με ένα πρόσωπο.

Για να γίνει αυτό θα πρέπει να περάσει στην οθόνη (ContactScreen) το chat id ώστε να ξέρουμε από πια συνομιλία θα γίνει η αφαίρεσή του. Από το προηγούμενο παράδειγμα στην οθόνη ChatSettingsScreen που χειρίζεται την πλοήγηση στην τρέχουσα (ContactScreen) μαζί με το uid θα περάσει και το chatId. Με το chatId διαθέσιμο μπορούν να βρεθούν τα δεδομένα για το συγκεκριμένο chat.

```

const chatId = props.route.params.chatId;
const chatData = chatId && storedChats[chatId];

```

Η επιλογή διαγραφής θα είναι τύπου SubmitButton component με τίτλο 'Remove from chat' και θα εκτελεί την μέθοδο removeFromChat. Η μέθοδος θα αφαιρεί τον χρήστη και αμέσως θα μας στέλνει στην προηγούμενη οθόνη. Θα βρίσκεται στο αρχείο chatActions με όνομα removeUserFromChat.

Στην removeUserFromChat θα παίρνουμε το id του χρήστη που θέλουμε να διαγράψουμε. Τους νέους χρήστες χωρίς τον συγκεκριμένο που βρίσκονται στο chatData.users πίνακα και θα γίνει διάκριση σε αυτούς που δεν έχουν το id του. Όποιος χρήστης έχει αυτό το id, που θα είναι μόνο ένας, θα αφαιρεθεί.

```

const newUsers = chatData.users.filter(uid => uid !== userToRemoveId);
// filters the users array of this chat based on anyone who doesn't have the userToRemoveId
// Whoever has the userToRemoveId (one user) will be removed

```

Με τους νέους χρήστες θα ενημερωθούν τα δεδομένα της συνομιλίας στην βάση δεδομένων με την μέθοδο updateChatData που υπάρχει ήδη και συγκεκριμένα το πεδίο users. Επίσης, θα πρέπει να αφαιρεθεί ο συγκεκριμένος διάλογος από αυτούς που καταγράφονται για τον χρήστη στην βάση δεδομένων καθώς καθένας έχει όλα τα chats που συμμετέχει στον κόμβο userChats. Εάν δεν γίνει η ενημέρωση η συνομιλία θα εξακολουθεί να εμφανίζεται στην κεντρική οθόνη στην λίστα μαζί με τις υπόλοιπες.

Ο κόμβος userChats περιέχει το id όλων των χρηστών και για κάθε έναν ξεχωριστά υπάρχει μία λίστα με όλες τις συνομιλίες στις οποίες μετέχουν.

Οπότε το παραπάνω chat θα πρέπει να διαγραφεί και από αυτή την λίστα. Αρχικά θα πρέπει να ανακτήσουμε την λίστα με τα chats καθενός που γίνεται με την μέθοδο getUserChats (userActions) η οποία και επιστρέφει το αποτέλεσμα. Με την λίστα διαθέσιμη θα γίνει επανάληψη σε όλα τα στοιχεία της μέχρι να βρούμε αυτό που αντιστοιχεί στον διάλογο που μας ενδιαφέρει. Εάν βρεθεί αυτό το

στοιχείο η μέθοδος που θα το διαγράφει είναι η `deleteUserChat ( userActions )` η οποία δημιουργεί αναφορά στον κόμβο `userChats`, στο αντίστοιχο `id` του χρήστη έπειτα το `key` του `chat` (η τιμή του είναι το `chatId`) και καλεί την μέθοδο `remove` του `Firebase` που εκτελεί την διαγραφή.

```
export const deleteUserChat = async (userId, key) => {
  try {
    const app = getFirebaseApp();
    const dbRef = ref(getDatabase(app));
    const chatRef = child(dbRef, `userChats/${userId}/${key}`);

    await remove(chatRef);
  }
  catch (error) {
    console.log(error);
    throw error;
  }
};
```

Η `deleteUserChat` θα χρησιμοποιηθεί στην μέθοδο `removeUserFromChat`.

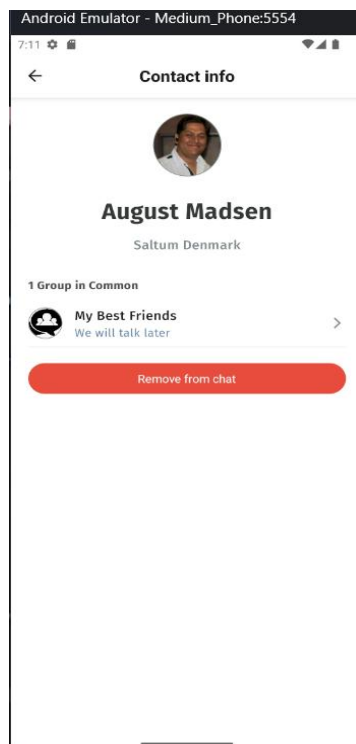
Όπως αναφέρθηκε ήδη η μέθοδος που θα εκτελείται όταν πατηθεί το κουμπί διαγραφής είναι η `removeFromChat` και σε αυτήν θα γίνει κλήση της `removeUserFromChat` για να ολοκληρωθεί η διαδικασία.

Όσον αφορά την δομή της βάσης δεδομένων στον κόμβο `userChats` στο αντίστοιχο `id` του χρήστη παρατηρείται να μειώνεται η λίστα κατά ένα στοιχείο ενώ το ίδιο γίνεται ταυτόχρονα και στον κόμβο `chats` όπου στους `users` του `chat` υπάρχει ένας λιγότερος.

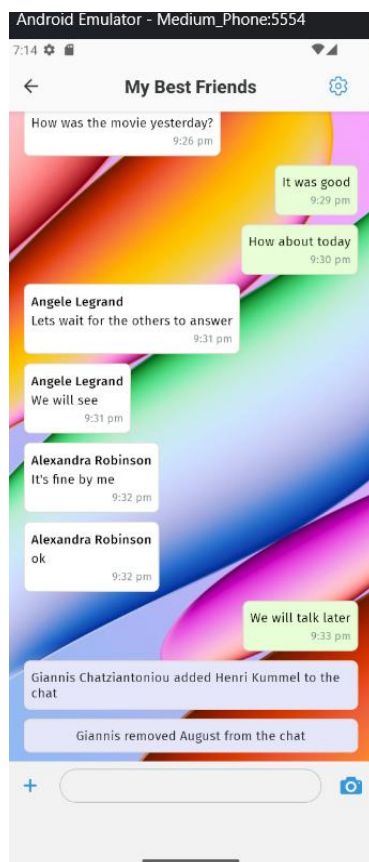
Αφού γίνει η αφαίρεση του χρήστη θα εμφανίζεται μήνυμα στον κύριο διάλογο ώστε να ξέρουν οι υπόλοιποι συμμετέχοντες πως έγινε κάποια ενέργεια.

Η υλοποίηση θα βρίσκεται στην μέθοδο `removeUserFromChat ( chatActions )` και μόλις γίνει η διαγραφή του χρήστη από τον διάλογο θα καλείται μέθοδος που θα στέλνει μήνυμα στην συνομιλία. Σκοπός της θα είναι να γνωστοποιεί στους υπόλοιπους χρήστες μηνύματα όπως όταν κάποιος φεύγει από την συνομιλία, όταν ένας αφαίρεσε κάποιον χρήστη ή όταν προστεθεί σε αυτήν. Θα έχει το όνομα `sendInfoMessage` όπου θα περνάει νέα παράμετρο στην `sendMessage` με όνομα `type` `'info'`. Τα μηνύματα τέτοιου τύπου θα έχουν διαφορετική μορφοποίηση από τα υπόλοιπα και η αλλαγή θα γίνεται στο `Bubble component ( case 'info' )`.

Για να εφαρμοστούν οι αλλαγές πρέπει στην οθόνη που υλοποιείται (`ChatScreen`) στο στοιχείο `FlatList` που γίνεται η εμφάνιση των μηνυμάτων (`Bubble`) να γίνει τροποποίηση ως προς τον τύπο του μηνύματος (`messageType`) και τότε τα μηνύματα θα αναπαρίστανται κατάλληλα.



Εικόνα 8.2 - Επιλογή διαγραφής συμμετέχοντα ομαδικής συνομιλίας.



Εικόνα 8.3 - Μηνύματα προσθήκης / διαγραφής κάποιου χρήστη.

8.3.4 Αποχώρηση του Εαυτού μας Από Ομαδική

Στην οθόνη ρυθμίσεων της ομαδικής συνομιλίας θα προστεθεί ένα κουμπί που θα μας επιτρέπει να αποχωρήσουμε από την συνομιλία. Στην οθόνη ChatSettingsScreen θα υπάρχει το SubmitButton component, θα αναπαριστά την επιλογή με τίτλο 'Leave chat' και όταν το πατήσουμε θα εκτελεί την μέθοδο leaveChat. Εάν αποχωρήσουμε από το chat δεν θέλουμε να πάμε πίσω γιατί θα μας μεταφέρει στην οθόνη συνομιλίας του, αντί αυτού θέλουμε να μεταφερθούμε στην κεντρική οθόνη με όλους τους διαλόγους με την μέθοδο popToTop του navigation.

```
await removeUserFromChat(userData, userData, chatData);

props.navigation.popToTop();
```

Η removeUserFromChat θα δέχεται τα δικά μας στοιχεία (userData) με την επόμενη παράμετρο να είναι τα δεδομένα του χρήστη που θέλουμε να αφαιρεθεί. Επειδή θέλουμε να αποχωρήσουμε από το chat τα δεδομένα αυτά θα είναι πάλι τα δικά μας. Στην μέθοδο removeUserFromChat θα γίνει αλλαγή ώστε στην περίπτωση που φύγουμε εθελοντικά να δείχνει το αντίστοιχο μήνυμα.

```
const messageText = userLoggedInData.userId === userToRemoveData.userId ?
  `${userLoggedInData.firstName} left the chat` :
  `${userLoggedInData.firstName} removed ${userToRemoveData.firstName} from the chat`;

await sendInfoMessage(chatData.key, userLoggedInData.userId, messageText);
```

8.4 Εμφάνιση Όλων των Χρηστών σε Διαφορετική Οθόνη

Μία κατάσταση που θα πρέπει να καλυφθεί είναι όταν σε μία συνομιλία υπάρχουν πολλά άτομα και δεν χωράνε στην οθόνη ρυθμίσεων με αποτέλεσμα να χάνονται στο κάτω μέρος της. Θα περιορίσουμε τον αριθμό των ατόμων που εμφανίζονται στην λίστα αλλά θα υπάρχει και μία ξεχωριστή επιλογή που θα τους εμφανίζει όλους.

Στην σελίδα ChatSettingsScreen όπου και γίνεται η εμφάνιση των χρηστών, στα δεδομένα chatData.users που βρίσκονται όλοι, θα εφαρμοστεί η μέθοδος slice και θα επιλέξουμε τους τέσσερις πρώτους. Όταν είναι πάνω από τέσσερις θα υπάρχει συνθήκη και θα εμφανίζει ένα κουμπί τύπου DataItem component, ενώ αν είναι λιγότεροι δεν θα φαίνεται στην οθόνη. Η επιλογή θα μεταφέρει σε νέα σελίδα και σε αυτήν θα καταγράφονται.

Το κουμπί θα έχει το όνομα 'View all' και θα είναι τύπου 'link' υποδεικνύοντας στον χρήστη πως μπορεί να μεταβεί σε άλλη οθόνη. Η νέα σελίδα θα είναι η DataListScreen και θα πρέπει να προστεθεί στο σύστημα πλοήγησης του MainNavigator, συγκεκριμένα στο Stack.Navigator ως νέο Stack.Screen με όνομα DataList.

Η πλοήγηση θα εκτελείται από το κουμπί παραπάνω και θα μεταφέρει μαζί δεδομένα που θα αξιοποιηθούν στην οθόνη προορισμού. Τα δεδομένα θα είναι ο τίτλος, οι συμμετέχοντες του διαλόγου, ο τύπος που θα εμφανιστεί, και το id της συνομιλίας chatId.

```
onPress={() => props.navigation.navigate('DataList',
  { title: 'Participants', data: chatData.users, type: 'users', chatId }
)}
```

Στην οθόνη DataListScreen αφού ανακτηθούν τα δεδομένα της πλοήγησης θα οριστεί ο τίτλος της σελίδας σε useEffect και τη χρήση του navigation.setOptions.

```
const { title, data, type, chatId } = props.route.params;

useEffect(() => {
  props.navigation.setOptions({ headerTitle: title });
}, [title]);
```

Η εμφάνιση των ατόμων θα γίνει με το στοιχείο FlatList και τα δεδομένα θα προέρχονται από το data που είχε περάσει στην οθόνη κατά την μεταφορά. Ο τύπος (type) θα είναι 'users' γιατί από την οθόνη ChatSettingsScreen περνάει αυτή η παράμετρος. Ορίζεται το type διότι σταδιακά θέλουμε και άλλου τύπου δεδομένα στην οθόνη όπως τα αγαπημένα μηνύματα οπότε χρειάζεται να προσδιορίζουμε ποιος είναι ο τύπος για να εμφανίζεται το αντίστοιχο περιεχόμενο.

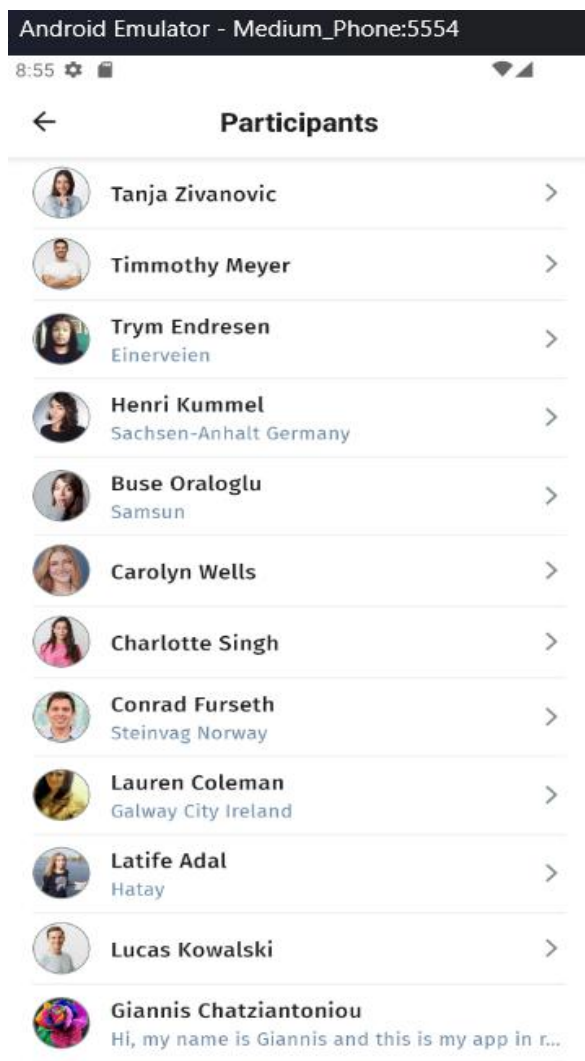
Εφόσον τα δεδομένα που δέχεται η οθόνη είναι τα id των χρηστών (τύπου users) θα δοθούν και οι ανάλογες τιμές στις μεταβλητές που ορίζονται. Με το id του τρέχον χρήστη θα ανακτηθούν τα δεδομένα του από τους αποθηκευμένους που βρίσκονται στο state της εφαρμογής (storedUsers). Έχοντας στην διάθεσή μας τα δεδομένα του χρήστη, τα πεδία που έχουν οριστεί θα πάρουν τιμές και θα χρησιμοποιηθούν στο DataItem component το οποίο και θα εμφανίσει κάθε ένα χρήστη ξεχωριστά.

Πατώντας πλέον στην επιλογή 'View all' ο χρήστης της εφαρμογής θα βλέπει όλους τους συμμετέχοντες της ομαδικής σε μία λίστα. Θα προστεθεί το χαρακτηριστικό όταν πατηθεί το στοιχείο ενός χρήστη να μεταβαίνει στην καρτέλα με τα στοιχεία του. Εάν πρόκειται για το δικό μας προφίλ δεν θα υπάρχει αυτή η δυνατότητα και αρκεί να ελεγχθεί το id να μην είναι ίδιο με το τρέχον στοιχείο.

```
const isLoggedInUser = uid === userData.userId;
```

Η λειτουργία της πλοήγησης θα πραγματοποιείται από το πεδίο onPress που θα περνάει στο DataItem και δεν θα κάνει καμία ενέργεια αν είμαστε εμείς ενώ αν είναι οποιοσδήποτε άλλος θα πηγαίνει στην οθόνη ContactScreen και θα περνάει σε αυτήν τα δεδομένα uid και chatId. Το uid αναφέρεται στο id του χρήστη στου οποίου τη καρτέλα γίνεται η πλοήγηση και το chatid είναι το id της ομαδικής (τα κλειδιά με αυτά τα ονόματα γιατί η ContactScreen χρησιμοποιεί τα ανάλογα).

```
onPress = isLoggedInUser ? undefined :
  () => props.navigation.navigate('Contact', { uid, chatId });
```



Εικόνα 8.4 - Λίστα με όλους τους συμμετέχοντες της ομαδικής συνομιλίας.

8.5 Προσθήκη Νέων Χρηστών στην Ομαδική

Στο κουμπί ‘Add users’ που έχει δημιουργηθεί ήδη (ChatSettingsScreen) θα προστεθεί η λειτουργία και όταν το πατήσει ο χρήστης θα ανοίγει την οθόνη που γίνεται η αναζήτηση νέων ατόμων. Στο onPress θα γίνεται η πλοήγηση στην οθόνη (NewChatScreen) από τη μέθοδο του navigation και θα περνάνε κάποια δεδομένα κατά την μεταφορά σε αυτή. Τα δεδομένα θα είναι οι ενεργοί χρήστες της συνομιλίας (existingUsers) ώστε να μπορέσουμε να τους αγνοήσουμε κατά την αναζήτηση, εφόσον είναι ήδη μέρος του διαλόγου, και τέλος το id (chatId) γιατί αν θέλουμε να προσθέσουμε νέους χρήστες θα πρέπει να γνωρίζουμε ποια είναι αυτή η συνομιλία. Επιπλέον, θα περάσει και το πεδίο isGroupChat γιατί γίνεται χρήση του στην οθόνη προορισμού (NewChatScreen) και καθορίζει τον τίτλο της σελίδας.

```
onPress={() => props.navigation.navigate('NewChat', {
  isGroupChat: true, existingUsers: chatData.users, chatId
})}

headerTitle: isGroupChat ? 'Add Participants' : 'New Chat'
```

Το existingUsers θα περιέχει τα id όλων των ατόμων που είναι ήδη μέρος του διαλόγου και βάσει αυτού θα γίνεται ο έλεγχος ώστε κατά την αναζήτηση να μην εμφανίζονται στα αποτελέσματα αυτοί οι χρήστες. Αρκεί να προστεθεί η συνθήκη στο FlatList που εμφανίζει την λίστα με τα ονόματα για τον τρέχοντα χρήστη και να τον αποκλείσει αν πράγματι υπάρχει ήδη.

```
// If the existingUsers contains that userId we just return
if (existingUsers && existingUsers.includes(userId)) return;
```

Όταν είμαστε σε υπάρχουσα ομαδική συνομιλία το πεδίο που εισάγεται το όνομα της θα είναι κρυφό. Χρειάζεται το chatId από τις παραμέτρους που περνάνε κατά την μεταφορά σε αυτή την οθόνη και αν δεν υπάρχει το chatId τότε υποθέτουμε πως είμαστε σε νέα συνομιλία. Σε ένα από τα σημεία που μπορεί να χρησιμοποιηθεί η παραπάνω υπόθεση είναι στο κουμπί που βρίσκεται στο δεξί μέρος της οθόνης ώστε να εμφανίζει διαφορετικό τίτλο αν πρόκειται για νέα συνομιλία ή μία υπάρχουσα. Εάν δεν είναι νέο chat ο τίτλος του κουμπιού θα είναι ‘Add’ αντί για ‘Create’.

```
const isNewChat = !chatId; // If there is no chatId we're assuming it's a new chat
title={isNewChat ? 'Create' : 'Add'}
```

Επίσης, όταν πατηθεί το παραπάνω κουμπί η πλοήγηση θα γίνεται σε διαφορετική οθόνη γιατί αν είναι νέα συνομιλία θα μεταβαίνει στην οθόνη ChatListScreen πρώτα, εάν όμως βρισκόμαστε σε υπάρχουσα θα γυρνάμε πίσω στην οθόνη ChatSettingsScreen γιατί από εκεί είχε γίνει η αρχική μετάβαση.

```
const screenName = isNewChat ? 'ChatList' : 'ChatSettings';
```

8.5.1 Λειτουργία Επιλογής Προσθήκης

Με την εύρεση και επιλογή χρήστη ή χρηστών όταν πατηθεί το κουμπί προσθήκη ‘Add’ θα γίνεται η μεταφορά στην προηγούμενη οθόνη ChatSettingsScreen και μαζί θα περνάει η λίστα με τα επιλεγθέντα άτομα (selectedUsers). Στην οθόνη ChatSettingsScreen θα γίνει η ανάκτησή τους και ο χειρισμός τους σε useEffect. Η λίστα selectedUsers περιέχει τα id των επιλεγμένων χρηστών, ωστόσο χρειαζόμαστε τα δεδομένα τους για την εμφάνιση στοιχείων τους στο παράθυρο της συνομιλίας που μπορούν να βρεθούν από τους αποθηκευμένους χρήστες στο state της εφαρμογής (storedUsers).

Πραγματοποιείται επανάληψη για κάθε ένα id του πίνακα `selectedUsers`, όσοι είναι οι χρήστες που προστίθενται, και με βάση τους `storedUsers` του state αποθηκεύονται τα δεδομένα του καθενός στον πίνακα με όνομα `selectedUserData`.

Αφού πλέον πατήσουμε 'Add' κάποιον χρήστη και περάσουν τα δεδομένα του στην οθόνη των ρυθμίσεων (`ChatSettingsScreen`) θα πρέπει αυτά να προστεθούν στην βάση δεδομένων. Την ενέργεια θα χειρίζεται η μέθοδος `addUsersToChat` (`chatActions`) και θα προσθέτει την τρέχουσα ομιλία στον κόμβο `userChats` για τον κάθε ένα χρήστη. Ο κόμβος `userChats` περιέχει όλα τα id των χρηστών και για τον καθένα ξεχωριστά μία λίστα με όλες τις συνομιλίες στις οποίες συμμετέχει. Σε αυτή την λίστα θα προστεθεί και το τρέχον `chatId` ώστε το chat να εμφανίζεται στις συμμετοχές του.

Την προσθήκη στην λίστα με τις υπάρχουσες συνομιλίες του χρήστη θα εκτελεί η μέθοδος `addUserChat` (`userActions`) με την `push` του `Firebase` και στην συνέχεια θα κληθεί στην προηγούμενη μέθοδο `addUsersToChat`.

Έχει ενημερωθεί ο κόμβος `userChats` για τον αντίστοιχο χρήστη και απομένει να γίνει το ίδιο και για τον κόμβο `chats` που περιέχει τα δεδομένα της κάθε συνομιλίας καθώς και τους `users`. Στους `users` βρίσκονται όλοι οι συμμετέχοντες και τώρα θα προστεθούν και οι επιλεγμένοι νέοι χρήστες με την `updateChatData`.

```
export const addUsersToChat = async (userLoggedInData, usersToAddData, chatData) => {
  const existingUsers = Object.values(chatData.users);
  // The user Id's of everyone who is already part of the chat
  const newUsers = [];

  let userAddedName = '';

  usersToAddData.forEach(async userToAdd => {
    const userToAddId = userToAdd.userId;

    if (existingUsers.includes(userToAddId)) return;

    newUsers.push(userToAddId);

    await addUserChat(userToAddId, chatData.key);

    userAddedName = `${userToAdd.firstName} ${userToAdd.lastName}`;
  });

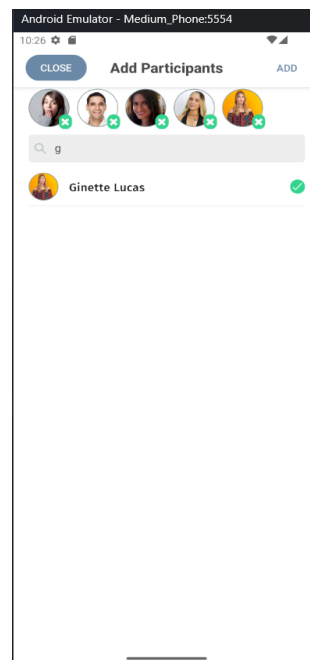
  if (newUsers.length === 0) return;

  await updateChatData(
    chatData.key, userLoggedInData.userId, { users: existingUsers.concat(newUsers) }
  );
};
```

Τελευταία λειτουργία είναι η εμφάνιση μηνυμάτων στην οθόνη ώστε οι υπόλοιποι χρήστες να ενημερώνονται όταν κάποιος έχει προστεθεί στην συνομιλία.

Στο αρχείο `chatActions`, στην `addUsersToChat` μετά την `updateChatData` μέθοδο που ενημερώνει τα δεδομένα στην βάση θα δημιουργηθεί το κατάλληλο μήνυμα. Για να πάρουμε τα στοιχεία ενός χρήστη χρειαζόμαστε δεδομένα και ο πίνακας `newUsers` που έχει οριστεί περιέχει τα id και όχι τα δεδομένα του. Θα δημιουργηθεί η μεταβλητή `userAddedName` που θα αποθηκεύει το όνομά του και μπορούμε να έχουμε πρόσβαση για την εμφάνιση του μηνύματος. Η τιμή της `userAddedName` θα αλλάζει σύμφωνα με τον αντίστοιχο χρήστη για τον οποίο γίνεται η επανάληψη. Το μήνυμα θα βρίσκεται στην μεταβλητή `messageText` και θα προσαρμοστεί κατάλληλα ώστε σε περίπτωση προσθήκης περισσότερων από ενός ατόμου να εμφανίζει μαζί και τον αριθμό τους. Ο τύπος του θα είναι 'info' γιατί αφορά μήνυμα πληροφορίας και για αυτό θα κληθεί η μέθοδος `sendInfoMessage` και μέσω της `sendMessage` θα ενημερώσει τους κόμβους `messages` και `chats` με τα δεδομένα.

```
const moreUsersMessage = newUsers.length > 1 ? `and ${newUsers.length - 1} others` : '';
const messageText = `${userLoggedInData.firstName} ${userLoggedInData.lastName} added $
{userAddedName} ${moreUsersMessage} to the chat`;
await sendInfoMessage(chatData.key, userLoggedInData.userId, messageText);
```



Εικόνα 8.5 – Αναζήτηση και προσθήκη νέων ατόμων σε ομαδικό διάλογο.



Εικόνα 8.6 - Ενημέρωση συνομιλίας μετά την προσθήκη πολλαπλών νέων χρηστών.

8.6 Προβολή Αγαπημένων Μηνυμάτων

Θα προστεθεί λειτουργία που θα εμφανίζει τα αγαπημένα μηνύματα της συνομιλίας. Ένα κουμπί με τίτλο 'Liked Messages' θα μεταφέρει τον χρήστη σε άλλη οθόνη (DataListScreen) όπου θα μπορεί να δει όλα τα μηνύματα που έχουν οριστεί ως αγαπημένα για τη συγκεκριμένη συνομιλία.

Χρειάζεται να βρεθούν τα αγαπημένα μηνύματα από το store της εφαρμογής στην οθόνη ChatSettingsScreen όπως και να δημιουργηθεί η επιλογή που θα βλέπει ο χρήστης στην οθόνη του. Για το κουμπί θα χρησιμοποιηθεί το DataItem component με τίτλο 'Liked messages', ο τύπος του θα είναι 'link' και η επιλογή onPress θα πραγματοποιεί πλοήγηση στην οθόνη με όνομα DataListScreen. Κατά την μεταφορά θα περνάνε τα δεδομένα title που θα ορίζουν τον τίτλο στην οθόνη μεταφοράς, το data που θα περιέχει τα αγαπημένα μηνύματα που ανακτήθηκαν από το store και το type. Με τα παραπάνω στοιχεία θα γίνει η κατάλληλη ενημέρωση στην οθόνη για την εμφάνιση των μηνυμάτων.

Η εμφάνιση θα γίνει και πάλι σε στοιχείο FlatList στο οποίο θα υπάρχει συνθήκη βάσει του τύπου (type = 'messages') καθώς αυτή περνάει σαν παράμετρος και θα εμφανίζεται το μήνυμα που έχουμε πατήσει ως αγαπημένο. Για να γίνει αυτό χρειάζονται τα δεδομένα των μηνυμάτων από το store της εφαρμογής στην μεταβλητή messagesData η οποία θα περιέχει όλα τα μηνύματα που έχει στείλει ο συγκεκριμένος χρήστης. Ωστόσο, θέλουμε αυτά που ανήκουν στο συγκεκριμένο chat που βρίσκεται και το αγαπημένο μήνυμα (messagesForChat).

Έχοντας τα μηνύματα αυτής της συνομιλίας θα βρεθούν τα αγαπημένα χρησιμοποιώντας το messageId μέσω του οποίου μπορεί να ανακτηθεί και ο αποστολέας του γιατί κάθε μήνυμα στην βάση δεδομένων περιέχει το πεδίο sentBy με το id αυτού που έστειλε το μήνυμα. Τα παραπάνω στοιχεία θα χρησιμοποιηθούν για την ενημέρωση των πεδίων που περνάνε στο DataItem.

```

else if (type === 'messages') {
  const likeData = itemData.item;
  const { chatId, messageId } = likeData;
  const messagesForChat = messagesData[chatId];
  // All of the messages that belong to this particular chat that this m
  // belongs to

  if (!messagesForChat) return;

  const messageData = messagesForChat[messageId];
  // We have all the data for that particular message
  const sender = messageData.sentBy && storedUsers[messageData.sentBy];
  // Retrieving the user data for the user that sent this message
  const name = sender && `${sender.firstName} ${sender.lastName}`;

  key = messageId;
  title = name;
  subTitle = messageData.text;
  itemType = '';
  onPress = () => {};
}

return <DataItem
  key={key}
  onPress={onPress}
  image={image}
  title={title}
  subTitle={subTitle}
  type={itemType}

```

Η επιλογή εμφάνισης των αγαπημένων μηνυμάτων θα προστεθεί και στην οθόνη ρυθμίσεων του προσωπικού μας προφίλ. Στο αρχείο SettingsScreen που αφορά την συγκεκριμένη οθόνη θα υπάρχει το DataItem component με τίτλο 'Liked messages' και όταν το πατήσει ο χρήστης θα τον μεταφέρει όπως και πριν στην οθόνη DataListScreen.

Στις ρυθμίσεις της ομαδικής συνομιλίας η πρόσβαση στα αγαπημένα γινόταν για το συγκεκριμένο id του chat. Τώρα που αφορά το δικό μας προφίλ και μόνο θέλουμε να εμφανίσουμε όλα τα μηνύματα από όλους τους διαλόγους που έχουν οριστεί ως αγαπημένα. Η likedMessages μεταβλητή περιέχει όλα τα

chatId και τα αντίστοιχα μηνύματα που έχουν οριστεί ως αγαπημένα για τον καθένα διάλογο σε μορφή Object. Η μορφή δεν είναι εφαρμόσιμη και θα πρέπει να γίνει επεξεργασία στην likedMessages με επανάληψη σε όλα τα id των chat (chats μεταβλητή) από όπου θα ανακτηθούν όλα τα αγαπημένα και θα αποθηκεύονται στον πίνακα result. Το αποτέλεσμα θα επιστρέφεται από την μεταβλητή sortedLikedMessages και θα το μεταφέρει κατά την πλοήγηση στην οθόνη DataListScreen όπου και θα αξιοποιηθεί για την εμφάνιση των αγαπημένων μηνυμάτων.

```
const sortedLikedMessages = useMemo(() => {
  let result = [];

  const chats = Object.values(likedMessages);
  chats.forEach(chat => {
    const chatMessages = Object.values(chat);
    result = result.concat(chatMessages);
  });

  return result;
}, [likedMessages]);
```

```
<DataItem
  type='link'
  title='Liked messages'
  hideImage={true}
  onPress={() => props.navigation.navigate('DataList', {
    title: 'Liked messages',
    data: sortedLikedMessages,
    type: 'messages'
  })}
/>
```

8.7 Επίλογος

Στο παρόν κεφάλαιο παρουσιάστηκε η υλοποίηση ενός συνόλου λειτουργιών που αφορούν τη διαχείριση των προσωπικών και ομαδικών συνομιλιών.

Αρχικά, υλοποιήθηκε η καρτέλα στοιχείων για τις προσωπικές συνομιλίες, μέσω της οποίας εμφανίζονται οι διαθέσιμες πληροφορίες του συνομιλητή. Τα δεδομένα ανακτώνται από το state της εφαρμογής και χρησιμοποιούνται για την παρουσίαση της εικόνας προφίλ, του ονόματος και των πρόσθετων πληροφοριών του χρήστη. Επιπλέον, προστέθηκε η δυνατότητα εντοπισμού των κοινών ομαδικών συνομιλιών των δύο χρηστών, καθώς και η απευθείας μετάβαση σε αυτές.

Για τις ομαδικές συνομιλίες δημιουργήθηκε ξεχωριστή οθόνη ρυθμίσεων, στην οποία υπάρχουν οι βασικές λειτουργίες διαχείρισης της ομάδας. Μέσω αυτής παρέχεται η δυνατότητα αλλαγής της εικόνας και του ονόματος της συνομιλίας. Οι μεταβολές ενημερώνουν τα αντίστοιχα δεδομένα στο Firebase, με αποτέλεσμα οι νέες πληροφορίες να μπορούν να χρησιμοποιηθούν στις διαφορετικές οθόνες όπου παρουσιάζεται η συγκεκριμένη συνομιλία.

Οι χρήστες μιας ομαδικής συνομιλίας εμφανίζονται στην οθόνη και υπάρχει πρόσβαση στις προσωπικές καρτέλες τους. Υλοποιήθηκε η δυνατότητα αφαίρεσης ενός συμμετέχοντα, με ταυτόχρονη ενημέρωση τόσο των δεδομένων της συνομιλίας όσο και των εγγραφών που συνδέουν τον συγκεκριμένο χρήστη με τους διαλόγους στους οποίους συμμετέχει. Με αυτόν τον τρόπο διατηρείται η συνέπεια μεταξύ των διαφορετικών τμημάτων της βάσης δεδομένων.

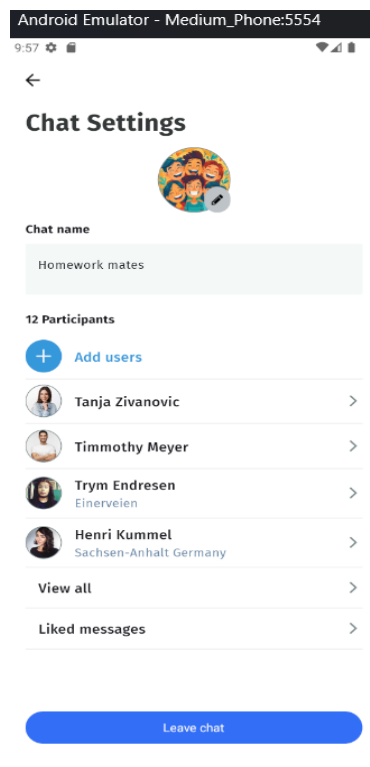
Επιπλέον, προστέθηκε η δυνατότητα αποχώρησης του ίδιου του χρήστη από μια ομαδική συνομιλία. Μετά την ολοκλήρωση της ενέργειας πραγματοποιούνται οι απαραίτητες ενημερώσεις και ο χρήστης

επιστρέφει στην κεντρική λίστα των συνομιλιών. Οι μεταβολές στη σύνθεση μιας ομάδας συνοδεύονται από κατάλληλα μηνύματα στον διάλογο, ώστε οι υπόλοιποι συμμετέχοντες να ενημερώνονται για την προσθήκη, την αφαίρεση ή την αποχώρηση κάποιου μέλους.

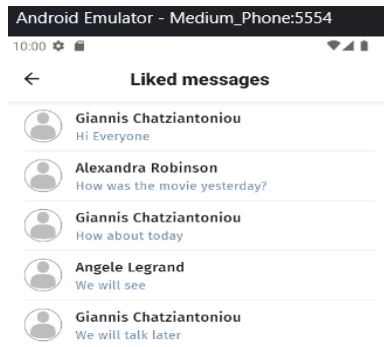
Για την αποτελεσματικότερη παρουσίαση ομάδων με μεγάλο αριθμό συμμετεχόντων δημιουργήθηκε ξεχωριστή οθόνη προβολής λίστας δεδομένων. Στην κύρια οθόνη ρυθμίσεων εμφανίζεται περιορισμένος αριθμός χρηστών, ενώ μέσω της σχετικής επιλογής παρέχεται πρόσβαση στην πλήρη λίστα. Η συγκεκριμένη οθόνη σχεδιάστηκε με δυνατότητα διαχείρισης διαφορετικών τύπων περιεχομένου, επιτρέποντας την επαναχρησιμοποίησή της και σε άλλες λειτουργίες.

Στη συνέχεια γίνεται η προσθήκη νέων χρηστών σε υπάρχουσες ομαδικές συνομιλίες. Κατά τη διαδικασία αναζήτησης λαμβάνονται υπόψη τα υφιστάμενα μέλη της ομάδας, ώστε να μην είναι δυνατή η εκ νέου προσθήκη τους. Μετά την επιλογή νέων συμμετεχόντων ενημερώνονται τόσο η λίστα χρηστών της συνομιλίας όσο και οι αντίστοιχες εγγραφές συμμετοχής κάθε νέου μέλους. Η δημιουργία ενημερωτικών μηνυμάτων επιτρέπει την άμεση ενημέρωση των αλλαγών στους υπόλοιπους συμμετέχοντες.

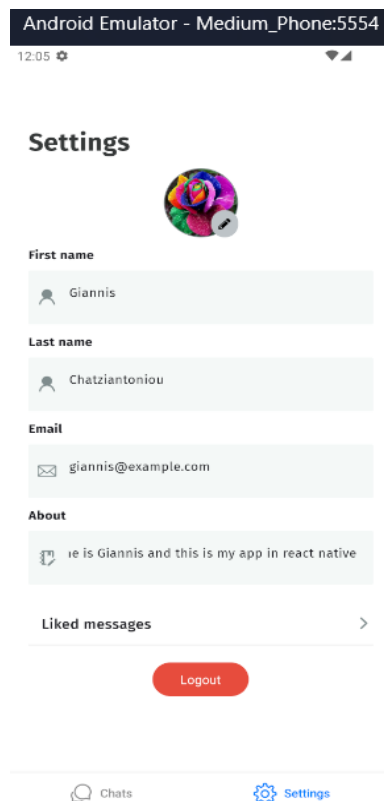
Τέλος, προστέθηκε η δυνατότητα προβολής των αγαπημένων μηνυμάτων. Ο χρήστης μπορεί να εμφανίζει τα μηνύματα που έχει επισημάνει ως αγαπημένα για μια συγκεκριμένη συνομιλία, ενώ αντίστοιχη δυνατότητα παρέχεται μέσω των προσωπικών ρυθμίσεων για το σύνολο των συνομιλιών του. Η αξιοποίηση της κοινής οθόνης παρουσίασης δεδομένων επιτρέπει την οργανωμένη εμφάνιση του συγκεκριμένου περιεχομένου χωρίς την ανάγκη δημιουργίας διαφορετικής οθόνης για κάθε περίπτωση. Συνολικά, οι λειτουργίες που αναπτύχθηκαν στο κεφάλαιο ολοκληρώνουν τη διαχείριση των προσωπικών και ομαδικών συνομιλιών.



Εικόνα 8.7 - Ολοκληρωμένη οθόνη ρυθμίσεων με τις επιλογές, "Add users", "View all" και "Liked messages".



Εικόνα 8.8 - Λίστα με όλα τα αγαπημένα μηνύματα του χρήστη.



Εικόνα 8.9 - Ολοκληρωμένη οθόνη ρυθμίσεων του εγγεγραμμένου χρήστη.

Κεφάλαιο 9ο: Αποστολή Ειδοποιήσεων

9.1 Εισαγωγή

Η λειτουργία των ειδοποιήσεων αποτελεί σημαντικό χαρακτηριστικό μιας εφαρμογής ανταλλαγής μηνυμάτων, καθώς επιτρέπει στον χρήστη να πληροφορείται για νέα δραστηριότητα στις συνομιλίες του χωρίς να απαιτείται να βρίσκεται τη συγκεκριμένη χρονική στιγμή στην αντίστοιχη οθόνη διαλόγου. Για την ανάπτυξη της συγκεκριμένης λειτουργικότητας αξιοποιείται το πακέτο Expo Notifications, μέσω του οποίου πραγματοποιείται η διαμόρφωση της εφαρμογής για τη λήψη απομακρυσμένων ειδοποιήσεων.

Στη συνέχεια αναλύεται η αποθήκευση των push tokens στη βάση δεδομένων και η συσχέτισή τους με τον αντίστοιχο χρήστη. Η δομή της εφαρμογής λαμβάνει υπόψη το ενδεχόμενο ένας λογαριασμός να χρησιμοποιείται από περισσότερες από μία συσκευές και, κατά συνέπεια, να διαθέτει πολλαπλά tokens. Για τον λόγο αυτό τα tokens οργανώνονται ανά χρήστη, ώστε να είναι δυνατή η αποστολή της ειδοποίησης σε όλες τις συσκευές στις οποίες είναι ενεργός ο συγκεκριμένος λογαριασμός.

Παράλληλα, εξετάζεται η διαχείριση των tokens κατά τη διαδικασία σύνδεσης, εγγραφής και αποσύνδεσης. Κατά την είσοδο ενός χρήστη στην εφαρμογή το token της συγκεκριμένης συσκευής αποθηκεύεται στα δεδομένα του, ενώ κατά την αποσύνδεση αφαιρείται. Με τον τρόπο αυτό γίνεται σωστή αντιστοίχιση μεταξύ συσκευών και λογαριασμών και αποφεύγεται η διατήρηση του token μιας συσκευής σε χρήστη ο οποίος δεν είναι πλέον συνδεδεμένος σε αυτή.

Η διαδικασία εφαρμόζεται τόσο στα μηνύματα κειμένου όσο και στα μηνύματα εικόνας. Για κάθε συνομιλία προσδιορίζονται οι χρήστες που πρέπει να ενημερωθούν και ανακτώνται τα αντίστοιχα tokens των συσκευών τους, εξαιρώντας τον αποστολέα του μηνύματος από τους παραλήπτες της ειδοποίησης. Η ίδια λογική μπορεί να εφαρμοστεί τόσο σε προσωπικούς όσο και σε ομαδικούς διαλόγους.

Τέλος, αναλύεται ο χειρισμός της αλληλεπίδρασης του χρήστη με μια ειδοποίηση.

Το κεφάλαιο επικεντρώνεται στην ενσωμάτωση ενός ολοκληρωμένου μηχανισμού ειδοποιήσεων, ο οποίος συνδέεται με το σύστημα αυθεντικοποίησης, τη βάση δεδομένων και την ανταλλαγή μηνυμάτων βελτιώνοντας τη συνολική εμπειρία χρήσης της εφαρμογής.

9.2 Διαμόρφωση Εφαρμογής για Ειδοποιήσεις

Για την αποστολή ειδοποιήσεων θα χρησιμοποιηθεί το πακέτο Expo Notifications. Οι ειδοποιήσεις λειτουργούν αποκλειστικά σε πραγματική συσκευή και όχι σε προσομοιωτές, οπότε για τον πειραματισμό χρειάζεται φυσική συσκευή. Οι τοπικές ειδοποιήσεις (local notifications) λειτουργούν και σε προσομοιωτή αλλά δεν θα χρησιμοποιηθούν στην εφαρμογή. Τοπικές ειδοποιήσεις είναι αυτές που υπάρχουν στην ίδια την συσκευή, αποστέλλονται και προγραμματίζονται αποκλειστικά από αυτή όπως ένα χρονόμετρο. Το χρονόμετρο θα στείλει ειδοποίηση σε μελλοντική ημερομηνία όταν ο χρόνος θα έχει τελειώσει όπως και μία εφαρμογή ημερολογίου όταν πλησιάζει ένα ραντεβού. Οι απομακρυσμένες ειδοποιήσεις (push notifications) αποστέλλονται από μία συσκευή σε κάποια άλλη, έτσι η συσκευή κάποιου άλλου μπορεί να στείλει στην δική μας ειδοποίηση.

Η εγκατάσταση του πακέτου ειδοποιήσεων γίνεται με την εντολή `npx expo install expo-notifications` και `npx expo install expo-device`.

Σύμφωνα με το εγχειρίδιο θα πρέπει να γίνει η εισαγωγή των δύο πακέτων στο αρχείο MainNavigator. Όλη η λειτουργικότητα θα βρίσκεται σε `useEffect` το οποίο θα εκτελείται μόνο μία φορά και όχι

συνέχεια. Καταχωρεί την εφαρμογή για να δέχεται ειδοποιήσεις με την μέθοδο `registerForPushNotificationsAsync` η οποία με την σειρά της κάνει έλεγχο αν πρόκειται για συσκευή Android και ορίζει τις ανάλογες ρυθμίσεις. Ζητάει την άδεια από την χρήστη ώστε η εφαρμογή να μπορεί να στέλνει ειδοποιήσεις και σε περίπτωση που δεν έχει αποδεχτεί εμφανίζει μήνυμα στην οθόνη. Εάν γίνει αποδοχή δημιουργείται token που μπορεί να αξιοποιηθεί για την δημιουργία της ειδοποίησης. Με την κλήση της `registerForPushNotificationsAsync` ο χειρισμός του token γίνεται από την μέθοδο `then` και δίνει τιμή στην state μεταβλητή `expoPushToken`.

Η μέθοδος `addNotificationResponseReceivedListener` εκτελείται όταν πατήσει ο χρήστης στην ειδοποίηση που έχει λάβει και στην ουσία περιλαμβάνει την απάντηση (`response`). Το κομμάτι του `return` από το `useEffect` εκτελείται πάντα όταν ολοκληρωθούν οι διεργασίες του και καταργεί τις ενέργειες `event` που είχαν δημιουργηθεί από τις ειδοποιήσεις.

Στο αρχείο `authActions` θα υπάρχει η μέθοδος `storePushToken` που θα δημιουργεί τα tokens για την συγκεκριμένη συσκευή. Κάθε συσκευή έχει το δικό της token που σημαίνει ότι αν ο χρήστης χρησιμοποιεί δύο διαφορετικές συσκευές τότε θα υπάρχουν και δύο tokens. Με την κατασκευή του token αυτό θα αποστέλλεται στην βάση δεδομένων στον κόμβο `users` και για το `id` του κάθε χρήστη θα υπάρχει κόμβος με όνομα `pushTokens` όπου και αποθηκεύεται.

Η μέθοδος `storePushToken` που δημιουργεί και αποθηκεύει τα tokens θα καλείται στην `signIn` που εκτελείται μόλις ο χρήστης συνδεθεί στην εφαρμογή. Αυτό σημαίνει πως μόλις γίνει η σύνδεση αυτόματα θα ορίζεται και token για τον συγκεκριμένο χρήστη. Το ίδιο θα γίνει και για την `signUp` μέθοδο που εκτελείται κατά την εγγραφή. Το αποτέλεσμα είναι όταν συνδέεται ο χρήστης από κινητό στον κόμβο `users` στο δικό του `id` να υπάρχει φάκελος με όνομα `pushTokens` και καταχωρημένο έναν αριθμό `pushToken`. Αν ο ίδιος συνδεθεί από tablet θα προστεθεί νέο `pushToken` καθώς πρόκειται για διαφορετική συσκευή.

Με τα tokens του χρήστη να αποθηκεύονται όταν γίνεται εγγραφή ή σύνδεσή του θα πρέπει εξίσου να αφαιρούνται κατά την αποσύνδεση από την εφαρμογή. Αυτή η ενέργεια πρέπει να γίνει γιατί αν ένας χρήστης αποσυνδεθεί και συνδεθεί εκ νέου κάποιος άλλος αυτός θα έχει το `push token` του προηγούμενου χρήστη (αν δεν το αφαιρέσουμε).

Η μέθοδος `removePushToken` θα εκτελεί την διαγραφή των token. Χρειάζονται όλα τα `pushToken` του χρήστη από την βάση δεδομένων ενέργεια που θα χειρίζεται η `getUserPushTokens` η οποία θα δημιουργεί αναφορά στον κόμβο `users`, στο `id` του χρήστη, στον φάκελο `pushTokens` και θα επιστρέφει το στιγμιότυπο από την συγκεκριμένη τοποθεσία. Το αποτέλεσμα που επιστρέφει μπορεί πλέον να χρησιμοποιηθεί στην `removePushToken` όπου θα γίνεται έλεγχος και αν το υπάρχον token είναι το ίδιο με της συσκευής τότε θα αφαιρείται και τέλος θα ενημερώνεται η τοποθεσία στην βάση με τα νέα δεδομένα.

Για να ολοκληρωθεί η διαδικασία η `removePushToken` θα προστεθεί στην μέθοδο `userLogout` που ενεργοποιείται μόλις ο χρήστης πατήσει το κουμπί αποσύνδεσης (`Logout`).

9.3 Ειδοποιήσεις Κατά την Αποστολή Μηνύματος Κειμένου και Εικόνας

Τα `pushToken` του χρήστη είναι πλέον αποθηκευμένα ενώ αφαιρούνται όταν αποσυνδεθεί από την εφαρμογή, μπορούν να αξιοποιηθούν για την αποστολή ειδοποιήσεων.

Η μέθοδος `sendPushNotificationForUsers` (`chatActions`) θα χειρίζεται την αποστολή των ειδοποιήσεων και θα δέχεται έναν πίνακα με τους χρήστες που θα σταλούν αυτές για παράδειγμα σε μία ομαδική συνομιλία ενώ αν δεν πρόκειται για ομαδική θα είναι μόνο ένα άτομο. Για όλους του χρήστες που μετέχουν στον διάλογο θα γίνει επανάληψη, θα ανακτηθούν τα tokens του καθένα και τότε θα σταλεί η

ειδοποίηση. Συνήθως θα υπάρχει ένα token, για το κινητό του χρήστη, αλλά είναι πιθανό να υπάρχουν πολλαπλές συσκευές δηλαδή περισσότερα tokens για αυτό θα γίνει επανάληψη σε όλα.

Για το καθένα token θα γίνει ένα αίτημα web (web request) στο url που δίνεται από το εγχειρίδιο και θα μεταφέρει δεδομένα. Η `sendPushNotificationForUsers` θα χρησιμοποιηθεί στην μέθοδο `sendTextMessage` και τις ειδοποιήσεις θα τις λάβουν όλοι οι υπόλοιποι χρήστες εκτός αυτού που έστειλε το ίδιο το μήνυμα.

Η άλλη περίπτωση κατά την οποία θα στέλνεται ειδοποίηση είναι με την αποστολή μηνύματος εικόνας. Η μέθοδος που θα εκτελεί την λειτουργία είναι η `sendImage` και θα καλεί την `sendPushNotificationForUsers`.

Όταν εμφανίζεται η ειδοποίηση ο χρήστης θα μπορεί να πατήσει πάνω της και να μεταφερθεί στην συνομιλία. Θα πρέπει να περάσει το `id` της συνομιλίας (`chatId`) στην `sendPushNotificationForUsers` μέθοδο όπως και στα δύο σημεία που καλείται για την αποστολή μηνύματος και εικόνας. Στο `MainNavigator` υπάρχει το σημείο που γίνεται ο χειρισμός της απάντησης (`response`) το οποίο είναι ένα `object` που περιέχει στοιχεία σχετικά με την ειδοποίηση. Ανάμεσα στα στοιχεία υπάρχει και το `data` με το `chatId` που περνάει κατά την δημιουργία του αιτήματος. Αρκεί να αποκτήσουμε πρόσβαση σε αυτό και να το χρησιμοποιήσουμε για την μεταφορά στην αντίστοιχη συνομιλία.

9.4 Επίλογος

Στο παρόν κεφάλαιο παρουσιάστηκε η ενσωμάτωση του μηχανισμού ειδοποιήσεων στην εφαρμογή, με σκοπό την ενημέρωση των χρηστών όταν πραγματοποιείται αποστολή νέου μηνύματος.

Πραγματοποιήθηκε η απαραίτητη διαμόρφωση της εφαρμογής για τη χρήση απομακρυσμένων ειδοποιήσεων μέσω του `Expo Notifications`. Κατά τη διαδικασία αυτή πραγματοποιείται ο έλεγχος των απαραίτητων δικαιωμάτων και δημιουργείται το `push token` της συσκευής, το οποίο μπορεί στη συνέχεια να χρησιμοποιηθεί για την αποστολή ειδοποιήσεων προς αυτή.

Για την διαχείριση των `tokens` δημιουργήθηκε κατάλληλη δομή στη βάση δεδομένων, μέσω της οποίας κάθε `token` συσχετίζεται με τον αντίστοιχο χρήστη. Η υλοποίηση υποστηρίζει την ύπαρξη περισσότερων από ενός `tokens` για τον ίδιο λογαριασμό, καλύπτοντας έτσι την περίπτωση κατά την οποία ο χρήστης χρησιμοποιεί την εφαρμογή από διαφορετικές συσκευές.

Η αποθήκευση των `tokens` προστέθηκε στις διαδικασίες εγγραφής και σύνδεσης, ώστε η συσκευή να καταχωρείται αυτόματα όταν ο χρήστης αποκτά πρόσβαση στην εφαρμογή. Αντίστοιχα, κατά την αποσύνδεση πραγματοποιείται αφαίρεση του `token` της συγκεκριμένης συσκευής από τα δεδομένα του λογαριασμού. Με τη διαδικασία αυτή διατηρείται η σωστή αντιστοίχιση των ενεργών συσκευών με τους χρήστες της εφαρμογής.

Όταν αποστέλλεται ένα μήνυμα κειμένου ή εικόνας, προσδιορίζονται οι υπόλοιποι συμμετέχοντες της συνομιλίας και ανακτώνται τα `tokens` που αντιστοιχούν στις συσκευές τους. Η ειδοποίηση αποστέλλεται στους κατάλληλους παραλήπτες χωρίς να συμπεριλαμβάνεται ο ίδιος ο αποστολέας του μηνύματος.

Τέλος, προστέθηκε η δυνατότητα αλληλεπίδρασης με τις εισερχόμενες ειδοποιήσεις. Το αναγνωριστικό της αντίστοιχης συνομιλίας συμπεριλαμβάνεται στα δεδομένα της ειδοποίησης και αξιοποιείται όταν ο χρήστης την επιλέξει. Με αυτόν τον τρόπο είναι δυνατή η άμεση μετάβαση στον διάλογο από τον οποίο προήλθε το μήνυμα.

Κεφάλαιο 10ο: Συμπεράσματα ή/και προτάσεις βελτίωσης

Η παρούσα πτυχιακή εργασία επικεντρώθηκε στον σχεδιασμό και την ανάπτυξη μιας εφαρμογής ανταλλαγής μηνυμάτων για κινητές συσκευές, με αξιοποίηση του React Native και του Expo. Βασικός στόχος αποτέλεσε η δημιουργία ενός λειτουργικού συστήματος επικοινωνίας, στο οποίο συνδυάζονται διαφορετικοί μηχανισμοί διαχείρισης χρηστών, συνομιλιών, μηνυμάτων και δεδομένων πραγματικού χρόνου. Το Firebase αξιοποιήθηκε για την αυθεντικοποίηση των χρηστών, την αποθήκευση και τον συγχρονισμό των δεδομένων, καθώς και για την αποθήκευση περιεχομένου. Το Redux Toolkit χρησιμοποιήθηκε για την κεντρική διαχείριση της κατάστασης της εφαρμογής, ενώ το React Navigation ανέλαβε την οργάνωση και τη διαχείριση της πλοήγησης μεταξύ των διαφορετικών οθονών.

Το τελικό σύστημα υποστηρίζει ένα ευρύ σύνολο λειτουργιών που υπάρχουν σε σύγχρονες εφαρμογές ανταλλαγής μηνυμάτων. Μεταξύ αυτών περιλαμβάνονται η εγγραφή και σύνδεση χρηστών, η διαχείριση προσωπικού προφίλ, η αναζήτηση χρηστών, η δημιουργία προσωπικών και ομαδικών συνομιλιών, καθώς και η αποστολή μηνυμάτων κειμένου και εικόνας. Υλοποιήθηκαν δυνατότητες απάντησης σε συγκεκριμένα μηνύματα, επισημάνσης αγαπημένων μηνυμάτων, διαχείρισης συμμετεχόντων σε ομαδικούς διαλόγους και αποστολής push notifications.

Η ανάπτυξη των λειτουργιών ανέδειξε τη σημασία των components καθώς αυτή συνέβαλε στη διατήρηση μιας πιο οργανωμένης δομής κώδικα.

Παρότι η εφαρμογή καλύπτει τις βασικές λειτουργίες ενός συστήματος ανταλλαγής μηνυμάτων, υπάρχουν αρκετές δυνατότητες ανάπτυξης και βελτίωσής της. Μία μελλοντική επέκταση θα μπορούσε να αφορά τη μετάβαση από τη JavaScript στην TypeScript. Η TypeScript επεκτείνει τη JavaScript προσθέτοντας στατικό σύστημα τύπων και θα μπορούσε να συμβάλει στη καλύτερη περιγραφή της δομής των δεδομένων που χρησιμοποιούνται στην εφαρμογή. Σε ένα σύστημα όπου δηλώνονται διαφορετικά αντικείμενα, όπως δεδομένα χρηστών, συνομιλιών, μηνυμάτων και παραμέτρων πλοήγησης, η χρήση τύπων θα μπορούσε να διευκολύνει τον άμεσο εντοπισμό σφαλμάτων.

Άλλη δυνατότητα θα ήταν η επιλογή και αλλαγή του χρωματικού θέματος της εφαρμογής από τον ίδιο τον χρήστη. Θα μπορούσαν, για παράδειγμα, να υπάρχουν διαφορετικά προκαθορισμένα χρωματικά θέματα ή η δυνατότητα επιλογής βασικών χρωμάτων για στοιχεία όπως το φόντο και τα μηνύματα συνομιλίας. Θα μπορούσε να συνδυαστεί και με υποστήριξη φωτεινού και σκοτεινού θέματος. Επιπλέον, η εφαρμογή θα μπορούσε να υποστηρίζει περισσότερους τύπους αρχείων. Εκτός από τα μηνύματα κειμένου και τις εικόνες, θα μπορούσε να προστεθεί δυνατότητα αποστολής ηχητικών μηνυμάτων και βίντεο. Θα μπορούσαν να προστεθούν λειτουργίες επεξεργασίας και διαγραφής ήδη αποσταλμένων μηνυμάτων.

Σαν τελευταία πρόταση θα ήταν χρήσιμη η πραγματοποίηση δοκιμών σε διαφορετικές φυσικές συσκευές και εκδόσεις λειτουργικών συστημάτων Android και iOS. Μέσω αυτών των δοκιμών θα μπορούσαν να αξιολογηθούν με μεγαλύτερη ακρίβεια η απόδοση, η συμβατότητα και η συνολική εμπειρία χρήσης της εφαρμογής.

ΒΙΒΛΙΟΓΡΑΦΙΑ

Επίσημα Τεχνικά Έγγραφα & Ιστοσελίδες

[1] **Meta Platforms. (2026). React Native Architecture Overview**

Native Documentation [Online]. Available: <https://reactnative.dev/docs/environment-setup>

[2] **Expo Core Team. (2026). Expo Application Services (EAS)**

Expo Docs [Online]. Available: <https://docs.expo.dev/>

[3] **OpenJS Foundation, (2026). Node.js Official Documentation.**

[Online]. Available: <https://nodejs.org/en>

[4] **Expo SplashScreen. A library that provides access to controlling the visibility**

behavior of native splash screen. Available: <https://docs.expo.dev/versions/v54.0.0/sdk/splash-screen/>

[5] **Google Fonts. Computer font and web font service owned by Google.**

[Online]. Available: <https://fonts.google.com/specimen/Fira+Sans>

[6] **React Navigation / Stack Navigator. Routing and navigation for React Native and Web apps.**

[7version]. Available: <https://reactnavigation.org/docs/stack-navigator>

<https://reactnavigation.org/docs/getting-started/>

[7] **React Navigation / Bottom Tabs.**

Available: <https://reactnavigation.org/docs/bottom-tab-navigator/>

[8] **Expo Vector Icons. Various types of icons in your Expo app, including react native vector icons.**

Available: <https://docs.expo.dev/guides/icons/>

<https://icons.expo.fyi/Index>

[9] **JavaScript Validation Package – Validate JS. A declarative way of validating javascript objects.**

Available: <https://validatejs.org/>

[10] **Firebase Realtime Database – Google. Store and sync data with NoSQL cloud database.**

Available: <https://firebase.google.com/docs/database>

[11] **Redux Toolkit. The standard way to write Redux logic.**

Available: <https://redux-toolkit.js.org/tutorials/quick-start>

[12] **React Native Async Storage. An asynchronous, unencrypted, persistent key-value storage solution for your React Native application.**

Available: <https://github.com/react-native-async-storage/async-storage#readme>

[13] **Expo ImagePicker. A library that provides access to the system's UI for selecting images and videos from the phone's library or taking a photo with the camera.**

Available: <https://docs.expo.dev/versions/latest/sdk/imagepicker/>

[14] React Native Universal Unique Identifier (uuid). A Universally Unique Identifier (UUID) URN Namespace.

Available: <https://www.npmjs.com/package/react-native-uuid?activeTab=dependencies>

[15] React Navigation Header Buttons. Render buttons in the navigation bar and handle the styling.

Available: <https://reactnavigation.org/docs/header-buttons>
<https://github.com/vonovak/react-navigation-header-buttons>

[16] React Native Popup – Menu. Extensible popup menu component for React Native, Android and iOS.

Available: <https://www.npmjs.com/package/react-native-popup-menu>

[17] Expo Clipboard. A universal library that allows getting and setting Clipboard content.

Available: <https://docs.expo.dev/versions/latest/sdk/clipboard/>

[18] React Native Awesome Alerts. Awesome alerts for React Native, works with iOS and Android.

Available: <https://github.com/rishabhhatia/react-native-awesome-alerts>

[19] Expo Notifications. A library that provides an API to fetch push notification tokens and to present, schedule, receive and respond to notifications.

Available: <https://docs.expo.dev/versions/latest/sdk/notifications/>