ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ (ΤΕΙ)

ΔΥΤΙΚΗΣ ΜΑΚΕΔΟΝΙΑΣ

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

«Δημιουργία εφαρμογής σε κινητή συσκευή για θέματα υγείας σε περιβάλλον iOS»

Πτυχιακή εργασία

του

Αριστερίδη Αλέξανδρου

Επιβλέπων: ***Σινάτκας Ιωάννης***

Καθηγητής

Καστοριά, Δεκέμβριος 2018

ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ (ΤΕΙ)

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

Πτυχιακή εργασία

«Δημιουργία εφαρμογής σε κινητή συσκευή για θέματα υγείας σε περιβάλλον iOS»

του

Αριστερίδη Αλέξανδρου

Επιβλέπων: ***Σινάτκας Ιωάννης***

Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την

Σινάτκας Ιωάννης	Φωτιάδης Δημήτριος	Αγγελής Στυλιανός
Καθηγητης	Επίκουρος Καθηγητής	Ε.Τ.Ε.Π

Καστοριά, Δεκέμβριος 2018

Ευχαριστίες

Ευχαριστώ την σύντροφο μου Αριστούλα για την σημαντική υποστήριξή της κατά την διάρκεια εκπόνησης αυτής της πτυχιακής εργασίας.

Περίληψη

Η αλματώδης τεχνολογική εξέλιξη της εποχής μας, με την πληθώρα επιλογών στον τρόπο επικοινωνίας και ενημέρωσης, έχει οδηγήσει σε μία έκρηξη χρηστών έξυπνων τηλεφώνων (smartphones) ακόμα και στις αναπτυσσόμενες χώρες, καθώς και σε ραγδαία ανάπτυξη νέων διαθέσιμων εφαρμογών για τους σκοπούς αυτούς και όχι μόνο.

Αντικείμενο αυτής της πτυχιακής εργασίας είναι η σχεδίαση και ανάπτυξη μιας διαδραστικής τηλεϊατρικής εφαρμογής, ένα chat app για θέματα υγείας με το όνομα «BoolevardAid» για την υποστήριξη της τηλεϊατρικής, έτσι ώστε να βοηθήσει στην πράξη χρήζοντες ιατρικής βοήθειας λειτουργώντας σαν προσωπικός βοηθός για τους χρήστες και την ιατρική κοινότητα. Απο την μία μεριά, η εφαρμογή θα είναι διαθέσιμη σε επαγγελματίες υγείας, εθνικούς οργανισμούς και λειτουργούς της δημόσιας διοίκησης και από την άλλη μεριά σε πολίτες και συγκεκριμένα σε ευάλωτες κοινωνικές ομάδες, αλλά και τους θεράποντες ιατρούς. Στόχος είναι η αποθήκευση και προώθηση ιατρικών δεδομένων, η απομακρυσμένη παρακολούθηση ασθενών και η άμεση επικοινωνία μεταξύ τους. Η συγκεκριμένη εφαρμογή τέλος προορίζεται για τις κινητές συσκευές που χρησιμοποιούν το λειτουργικό σύστημα iOS της Apple.

Η εφαρμογή θα παρέχει την δυνατότητα στον χρήστη να δημιουργεί λογαριασμό ανώνυμα ή επώνυμα και να μπορεί να συνδεθεί σε πραγματικό χρόνο (real time) με το υποκείμενο που τον ενδιαφέρει τηρώντας παράλληλα τους βασικούς κανόνες απορρήτου και προστασίας των προσωπικών του δεδομένων. Για την αλληλεπίδραση αυτή χρησιμοποιήθηκε το πρόγραμμα Firebase της Google, το οποίο μας δίνει την δυνατότητα να δημιουργήσουμε μια βάση δεδομένων από χρήστες καθώς και τα κρυπτογραφημένα μηνύματά τους, το οποίο εν τέλει συνδέουμε με το project μας. Τέλος ο χρήστης θα έχει την δυνατότητα να δει με ποιους άλλους χρήστες έχει συνομιλήσει αλλά και να ανατρέξει στο ιστορικό της συνομιλίας του.

Λέξεις Κλειδιά: iOS, XcodeApple, εφαρμογή, chat, application, BoulevardAid, τηλεϊατρική, Apple, Λειτουργικό κινητής συσκευής, Εφαρμογή ανταλλαγής μηνυμάτων

Abstract

The rapid technological progress of our era, with the many choices in the ways of communication and information, has as a result that many people use smartphones even at the growing countries, and new available applications are developed rapidly for those or other purposes.

The current labour deals with the design and the development of an interactive telemedicine application, a chat app for medical issues with the name «BoolevardAid» about the support of the telemedicine, in order to be supported people who need medical support, and which will be useful for the users such as for the doctors. On the one hand, the application will be available for practitioners, national organizations and officers of the public administration, on the other hand for the citizens, specifically for the vulnerable groups and the doctors. The purpose of the application is to save and propel medical data, medical monitor remotely the patients such as to communicate directly each other. The current application is designed for the portable telephones that use the Apple's operating system, iOS.

The application gives the opportunity to the user to create an account anonymously or eponymously and to be able to log-in in real time with the person that he is interested in, abiding also by the basic privacy and personal data's protection rules. The Google's programme named Firebase was used for this interaction, which give us the ability to create a user's data basis such as their encrypted messages, and which (programme) we connect with our project. After all, the user has the ability to see with whom he has talked to and go back to his chat's record.

Keywords: iOS, XcodeApple, chat, ApplicationSoftware, BoulevardAid, telemedicine, MacOS, Apple, mobileOS, ChatApp

Λίστα Πινάκων

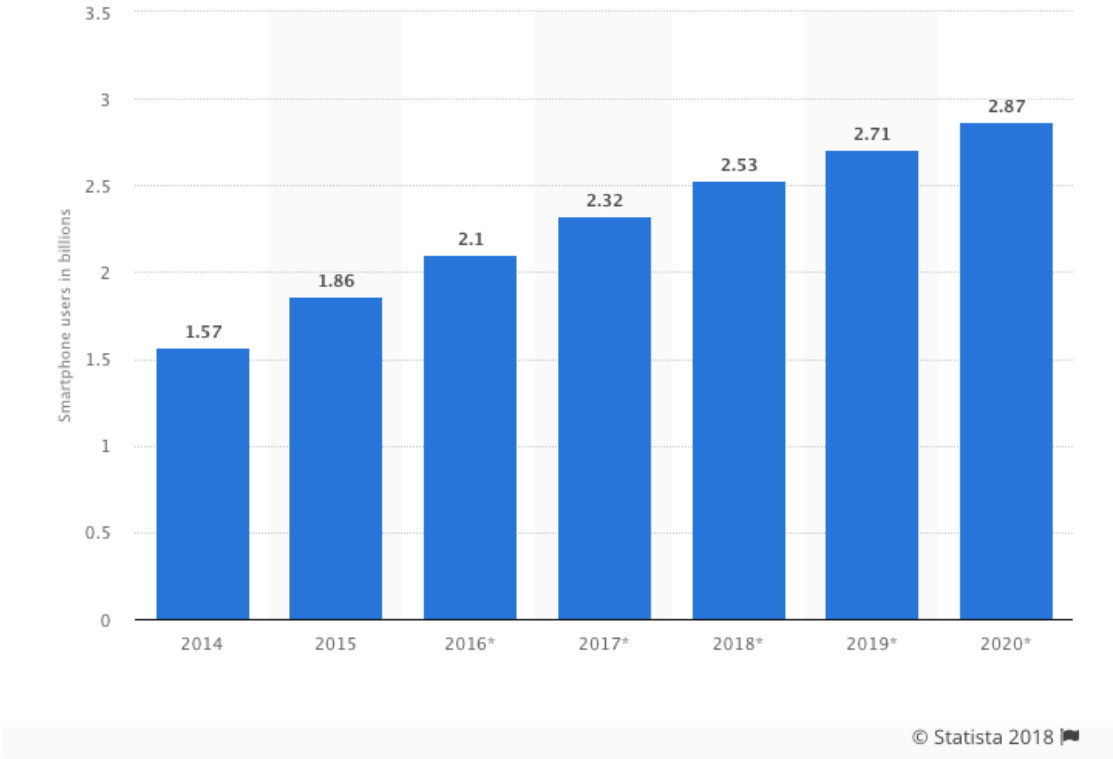
Λίστα Εικόνων

Περιεχόμενα

Εισαγωγή

Στην όλο και αυξανόμενη ανάγκη για άμεση επικοινωνία οι μεγάλες εταιρίες τεχνολογίας έκαναν τεράστια άλματα προς αυτήν την κατεύθυνση μέσα σε λίγα μόλις χρόνια, ειδικότερα δε στον τομέα του hardware αλλά και του software, μετατρέποντας έτσι τα κινητά τηλέφωνα σε μικρούς υπολογιστές, υποκαθιστώντας και μερικές φορές αντικαθιστώντας τους desktop υπολογιστές. Η αποκλειστική χρήση ενός έξυπνου τηλεφώνου (smartphone) από πολλούς αλλά και η τάση των χρηστών και της αγοράς για αλλαγή πλεύσης και εστίασης στο mobile marketing σε σχέση με τους συμβατικούς υπολογιστές, κατέστησε ελκυστικό από την αρχή το project της πτυχιακής. Το ακριβές θέμα της πτυχιακής σε συνδυασμό με την προοπτική των δυνατοτήτων που πιθανώς θα προκύψουν μέσα από αυτή την εφαρμογή (application) σε θέματα υγείας, έκανε αρκετά εύκολη την επιλογή του.

Στο κεφάλαιο που ακολουθεί θα γίνει μια εισαγωγή στο λειτουργικό σύστημα, στις κινητές πλατφόρμες, στις εφαρμογές και μια αναδρομή στην ιστορία του λειτουργικού συστήματος σε φορητές συσκευές. Στο δεύτερο κεφάλαιο θα γνωρίσουμε και θα αναπτύξουμε συνοπτικά τις iOS τεχνολογίες. Έπειτα στο τρίτο κεφάλαιο θα παρουσιαστεί η διαδικασία σχεδίασης, ανάπτυξης και υλοποίησης της εφαρμογής. Τέλος στο τέταρτο κεφάλαιο περιγράφονται τα συμπεράσματα από την δημιουργία αυτής της εφαρμογής και οι μελλοντικές επεκτάσεις που αυτή επιδέχεται.



Πίνακας 1.1

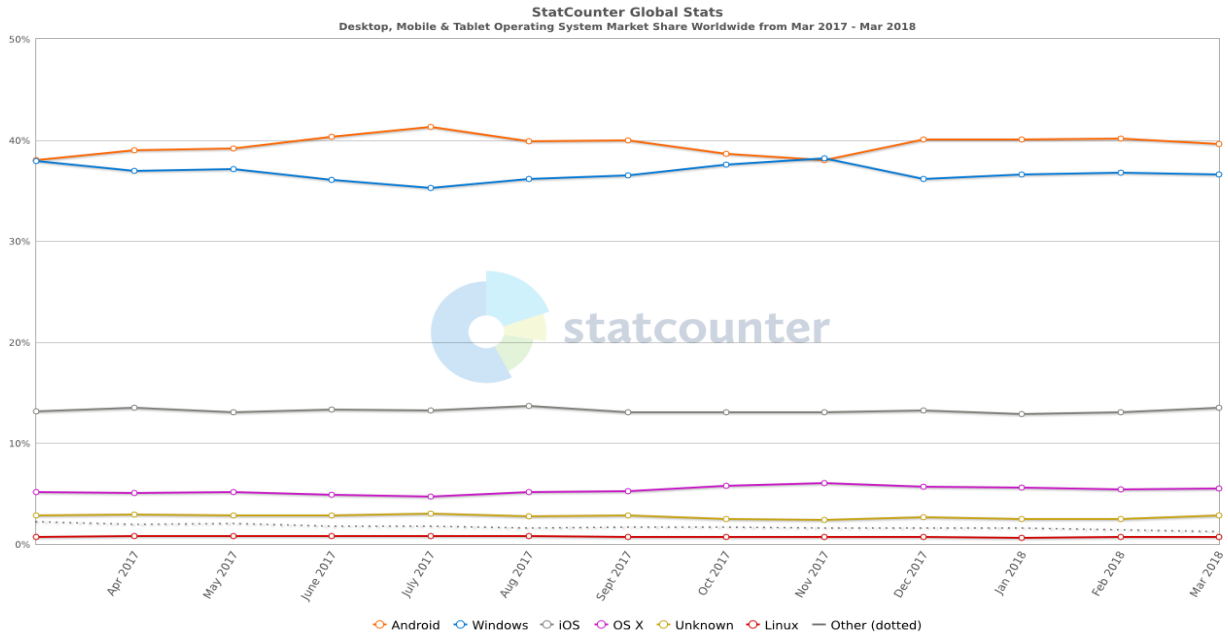
Αριθμός χρηστών έξυπνων τηλεφώνων παγκοσμίως

1.1 Λειτουργικό σύστημα κινητής συσκευής

Το λειτουργικό σύστημα κινητής συσκευής (mobile OS) είναι ένα λειτουργικό σύστημα για τηλέφωνα, tablets, έξυπνα ρολόγια ή άλλες κινητές συσκευές. Παρόλο που κάποιοι υπολογιστές όπως λόγου χάρη τα laptops, ανήκουν στην κατηγορία των κινητών, δηλαδή μεταφέρονται εύκολα όπως οι άλλες κινητές συσκευές, το λειτουργικό σύστημα που χρησιμοποιούν δεν λογίζεται σαν λειτουργικό σύστημα κινητής συσκευής γιατί εξαρχής σχεδιάστηκε για επιτραπέζιους υπολογιστές και δεν έχουν ή δεν χρειάζονται ειδικά χαρακτηριστικά κινητής συσκευής. Το λειτουργικό σύστημα μιας κινητής συσκευής συνδυάζει χαρακτηριστικά ενός λειτουργικού συστήματος desktop υπολογιστή με άλλα χαρακτηριστικά, που είναι χρήσιμα για τις κινητές συσκευές πολλά από τα οποία είναι απαραίτητα για μια σύγχρονη κινητή συσκευή όπως για παράδειγμα η οθόνη αφής, το Bluetooth, το δίκτυο κινητής τηλεφωνίας, το Wi-Fi, το

GPS καθώς και πολλά άλλα, ωστόσο το λειτουργικό σύστημα απαιτεί πολύ λιγότερους υπολογιστικούς πόρους σε σχέση με έναν desktop υπολογιστή. [1]

Το λογισμικό αυτό ξεκινάει όταν η συσκευή ενεργοποιείται και επιτρέπει στις κινητές συσκευές να το εγκαθιστούν καθώς και να λειτουργούν εφαρμογές και προγράμματα, μέσω της απεικόνισης εικονιδίων σε μια οθόνη, της παρουσίασης πληροφοριών και της παροχής πρόσβασης στις εφαρμογές -απεικονίζοντας μία οθόνη συνήθως από εικονίδια, παρουσιάζοντας πληροφορίες και παρέχοντας πρόσβαση στις εφαρμογές. Το software διαχειρίζεται το κινητό και το ασύρματο δίκτυο καθώς και την πρόσβαση στην κινητή συσκευή. Τα περισσότερα λειτουργικά συστήματα για κινητές συσκευές έχουν δεσμευτεί σε συγκεκριμένο hardware με μικρή δυνατότητα ευελιξίας. [2] Τα δημοφιλέστερα λειτουργικά συστήματα είναι το iOS της Apple, το Android της Google και τα Windows Phone της Microsoft.



Πίνακας 1.2

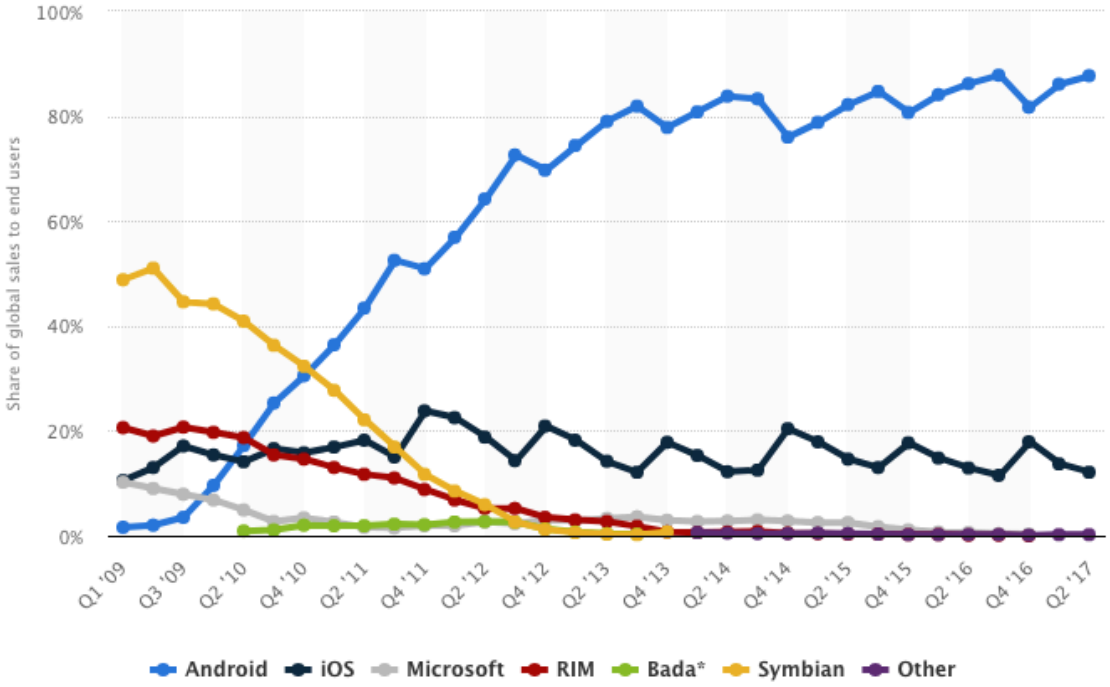
Χρήση λειτουργικών συστημάτων

Αυτή η νέα γενιά λειτουργικών συστημάτων έχει ρίξει το βάρος στην ανάπτυξη εφαρμογών από τρίτες εταιρίες αλλά και από ανεξάρτητους προγραμματιστές. Αυτός είναι και ο βασικός λόγος για τον οποίο όλα τα νέα λειτουργικά συστήματα προμηθεύουν με τα δικά τους λογισμικά

ανάπτυξης εφαρμογών οποιονδήποτε θέλει να ασχοληθεί και ενημερώνουν τους επίδοξους προγραμματιστές για τους τρόπους ανάπτυξης σε αυτά.

1.2 Ιστορία λειτουργικών συστημάτων κινητών συσκευών

Στις αρχές του 1990 η Psion κυκλοφόρησε το Psion Series 3 PDA, μία μικρή υπολογιστική συσκευή που υποστήριζε εφαρμογές γραμμένες από τον χρήστη σε ένα λειτουργικό σύστημα που ονομαζόταν EPOC. Οι επόμενες εκδόσεις του EPOC μετεξελίχθηκαν σε Symbian, ένα λειτουργικό σύστημα το οποίο χρησιμοποίησαν στα κινητά τους τηλέφωνα η Nokia, η Motorola και η Ericsson [3]. Το 1996 η Palm κυκλοφόρησε το Pilot 1000 τρέχοντας το Palm OS που ήταν εύκολο στη χρήση του παρέχοντας μια σουίτα από βασικές εφαρμογές όπως ημερολόγιο, αριθμομηχανή και ατζέντα. Στις επόμενες εκδόσεις του το Palm OS αναβαθμίστηκε ώστε να υποστηρίζει έξυπνα τηλέφωνα όπως επίσης εφαρμογές και λειτουργίες από third party κατασκευαστές και προγραμματιστές. [4] Το 2000 η Microsoft κυκλοφόρησε το PocketPC 2000 με λειτουργικό το Windows CE το οποίο το 2003 μετονομάστηκε σε Windows Mobile. Το 2007 η Apple παρουσίασε το iPhone με λειτουργικό το iPhone OS, βασισμένο σε Unix, το οποίο το 2010 μετονομάστηκε σε iOS εισάγωντας ένα ισχυρό και παράλληλα πρωτοποριακό User Interface για τον χρήστη. Μαζί εγκαινιάστηκε και το ηλεκτρονικό κατάστημα εφαρμογών το iOS App Store το οποίο έδινε την δυνατότητα να κατεβάσεις εφαρμογές είτε της Apple είτε από third party κατασκευαστές έχοντας ήδη λανσάρει το δικό της Software Development Kit [5]. Τον επόμενο χρόνο παρουσιάστηκε από την Google το Android, λειτουργικό βασισμένο σε Linux το οποίο ακριβώς επειδή είναι open source λειτουργικό και όχι σαν το iOS, το οποίο είναι closed source, έγινε ευρύτατα αποδεκτό και από τους ανεξάρτητους προγραμματιστές αλλά και από τις άλλες κατασκευάστριες εταιρίες συσκευών τηλεφωνίας οι οποίες αμέσως υιοθέτησαν το λειτουργικό σύστημα αυτό. [6]

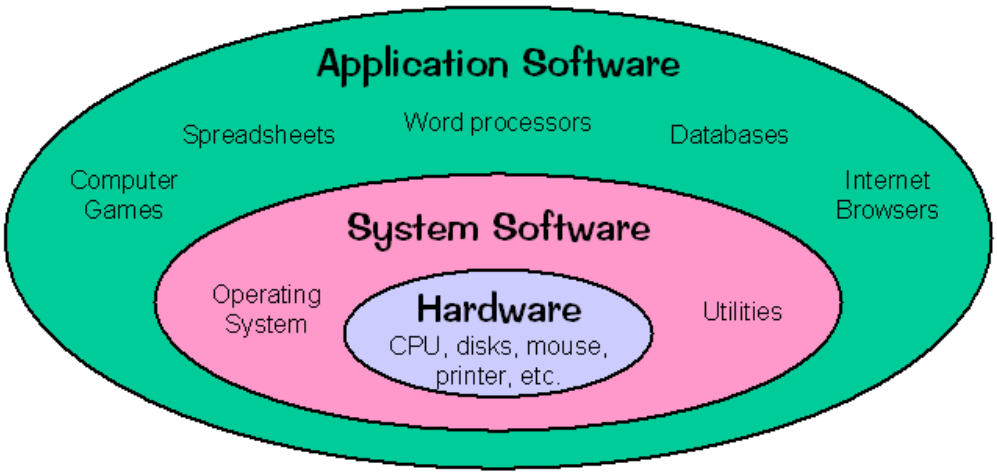


Πίνακας 1.3
Μερίδιο στην αγορά iOS

1.3 Λειτουργικό σύστημα

Το λειτουργικό συστήματος (systems software) αποτελείται από προγράμματα και υπηρεσίες του κατασκευαστή, τα οποία είναι σχεδιασμένα αποκλειστικά για να διαχειρίζονται το υλικό ενός υπολογιστή ή μιας κινητής συσκευής καθώς επίσης προκειμένου να παρέχεται μια πλατφόρμα, στην οποία θα μπορούν να τρέξουν άλλα λογισμικά όπως το λειτουργικό σύστημα αλλά και τα application software. Στόχος του είναι η επικοινωνία με τα εσωτερικά μέρη της συσκευής όπως είναι ο σκληρός δίσκος, η μνήμη RAM, οι μικροεπεξεργαστές κ.α., έτσι ώστε να διευκολύνεται ο χρήστης. Συγκεκριμένα περιλαμβάνει όλους τους drivers που είναι απαραίτητοι για αυτήν την επικοινωνία, δηλαδή με απλούστερα λόγια είναι η διεπαφή(interface) μεταξύ του χρήστη και του hardware. Το λειτουργικό σύστημα μιας συσκευής δεν είναι μόνο ένα από τα σημαντικότερα systems software της συσκευής αλλά και το σύστημα που χρησιμοποιείται πιο συχνά. Πρόκειται λοιπόν για το λογισμικό που εκτελείται στο παρασκήνιο και ενώνει τα χωριστά, φυσικά μέρη της συσκευής, προκειμένου να παρέχεται μια απρόσκοπτη και συνεχής εμπειρία στον χρήστη (user

experience). Μερικές από τις λειτουργίες του είναι η μεταφορά δεδομένων μεταξύ της μνήμης και των δίσκων στον σκληρό δίσκο καθώς και το γραφικό περιβάλλον διεπαφής του χρήστη (GUI), το οποίο είναι αποκλειστικά και μόνο αποτέλεσμα του λειτουργικού συστήματος (OS) στην συσκευή. Το λειτουργικό σύστημα είναι ανεξάρτητο τόσο από τον χρήστη όσο και από τα application software που σημαίνει ότι δεν εξαρτάται άμεσα από αυτούς τους παράγοντες για να λειτουργήσει. Τέλος πολλά από τα systems software περιλαμβάνουν το BIOS και το υλικολογισμικό της συσκευής, το οποίο διευκολύνει τον χρήστη να αλληλοεπιδράσει με την συσκευή. [7]



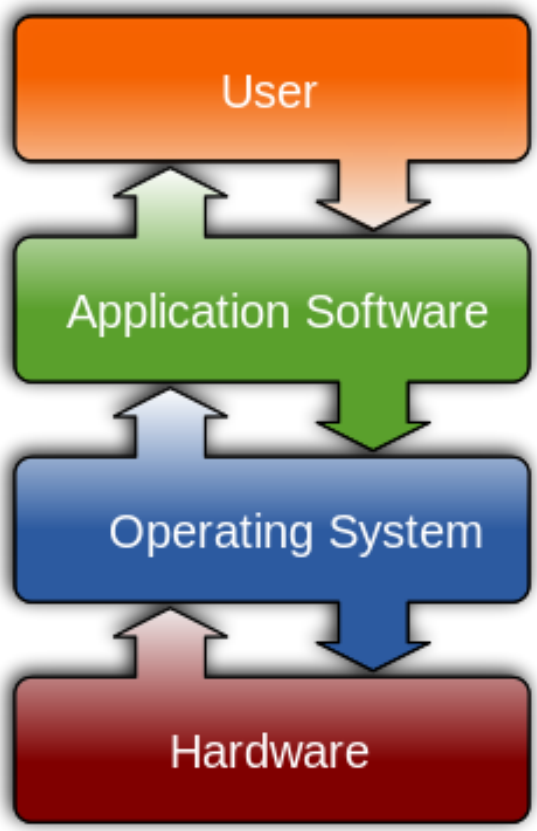
Εικόνα 1.1

Λειτουργικό σύστημα vs λειτουργικό εφαρμογής

1.4 Εφαρμογές (application software)

Application software ή αλλιώς applications ή πιο σύντομα και πιο διαδεδομένα apps, είναι τα πιο κοινά προγράμματα που εκτελούνται στο προσκήνιο μια συσκευής. Ένα mobile app είναι ένα πρόγραμμα υπολογιστή, το οποίο είναι σχεδιασμένο να τρέχει σε μια κινητή συσκευή [8]. Έχουν την τάση να εκτελούν χρήσιμες εργασίες που δεν σχετίζονται όμως με την συντήρηση της συσκευής, την εκκίνηση του συστήματος ή την επικοινωνία με το υλικό. Οι εφαρμογές εξαρτώνται άμεσα από το systems software το οποίο τις βοηθά να επικοινωνήσουν με τα φυσικά

μέρη της συσκευής και δεν μπορούν να λειτουργήσουν χωρίς αυτό. Εάν μπορούσαμε να απεικονίσουμε το application software, αυτό θα ήταν στην κορυφή της πυραμίδας και θα λειτουργούσε πάνω από το systems software, όντας το πιο ορατό στοιχείο για τον χρήστη, ενώ το systems software θα παρέμενε απαρατήρητο στο παρασκήνιο.



Εικόνα 1.2
Επίπεδα του Apple iOS

Όταν χρειάζεται να επικοινωνήσει μια εφαρμογή με το hardware, το systems software επικοινωνεί με το hardware για λογαριασμό της εφαρμογής και μεταδίδει οποιαδήποτε πληροφορία σε αυτό. Στον αντίποδα, όποια πληροφορία χρειάζεται το hardware θα περάσει από το systems software στο application software. Οι εφαρμογές είναι οι πιο οικείες μορφές λογισμικού και συνοδεύονται από μεγάλη ποικιλία διαφόρων τύπων. Συνήθως μπορούν να είναι προσβάσιμες μέσω ενός γραφικού περιβάλλοντος διεπαφής του χρήστη, το οποίο ανήκει στο λειτουργικό σύστημα, συνήθως πατώντας πάνω στην εφαρμογή. Μερικές από τις πιο δημοφιλείς εφαρμογές που συναντάμε πιο συχνά, έχουν να κάνουν με αριθμητικές πράξεις, επεξεργασίας

κειμένου, επεξεργασίας φωτογραφιών κ.α., οι οποίες συνήθως παρέχονται δωρεάν από τους κατασκευαστές των συσκευών [9]. Θα μπορούσε όμως να παρατηρήσει κάποιος ότι πολλές από τις νέες εφαρμογές που δημιουργούνται, έχουν να κάνουν με την καθημερινή δραστηριότητα των ανθρώπων και την βελτίωση αυτής. Δεδομένου ότι τη σημερινή εποχή τα smartphones παίζουν σημαντικό ρόλο στη ζωή μας και αποτελούν αναπόσπαστο κομμάτι της καθημερινότητάς μας, τα ηλεκτρονικά καταστήματα εφαρμογών σχετικά με τα διάφορα λειτουργικά συστήματα και τις κινητές συσκευές που υπάρχουν είναι εύκολα προσβάσιμα στον χρήστη, ο οποίος, είτε δωρεάν είτε επί πληρωμή και μάλιστα με πολύ απλό τρόπο, έχει τη δυνατότητα αναζήτησης και κατεβάσματος εφαρμογών που καλύπτουν τις ανάγκες του από μία τεράστια δεξαμενή. Αντιλαμβανόμενες την επιρροή τους σε πολλές πτυχές της ζωής μας, όλο και περισσότερες εταιρίες επενδύουν προς αυτήν την κατεύθυνση με αποτέλεσμα ο ανταγωνισμός να είναι τεράστιος και πρώτος κερδισμένος ο καταναλωτής.

2 Τεχνολογίες iOS

2.1 Λειτουργικό σύστημα iOS

Το iOS, πρώην iPhone OS, είναι ένα λειτουργικό σύστημα το οποίο δημιουργήθηκε και αναπτύχθηκε από την εταιρεία τεχνολογίας Apple αποκλειστικά για τις δικές της κινητές συσκευές όπως είναι το iPhone, το iPad, το iPod και το iWatch. Επίσημα παρουσιάστηκε το έτος 2007 για την υποστήριξη του iPhone - έξυπνου τηλεφώνου που κυκλοφόρησε, ωστόσο αναβαθμίστικε στην συνέχεια για να υποστηρίξει και τις υπόλοιπες κινητές συσκευές τις εταιρείας. Το περιβάλλον χρήστη στο iOS είναι βασισμένο στο στυλ του άμεσου χειρισμού (direct manipulation), στην αλληλεπίδραση δηλαδή του χρήστη με την συσκευή, το οποίο περιλαμβάνει μία συνεχή παρουσίαση αντικειμένων καθώς και διαχείρισης αυτών μέσω πολλαπλής αφής (multi-touch) πάνω στην οθόνη της συσκευής, όπως είναι το πάτημα και το σύρσιμο (switching) που έχουν συγκεκριμένες λειτουργίες στο λειτουργικό. Επίσης οι εσωτερικοί αισθητήρες που έχει εισάγει στις συσκευές και έχει προσαρμόσει στο λειτουργικό της η Apple, η οποία θα πρέπει να παρατηρήσουμε ότι αποτελεί μία καινοτόμο τεχνολογία,

χρησιμοποιούνται απο πολλές εφαρμογές που υπακουούν σε συγκεκριμένες κινήσεις, όπως είναι η περιστροφή και το γρήγορο κούνημα της συσκευής, με την δυνατότητα να χρησιμοποιούν τα προϊόντα της ακόμα και άτομα με ειδικές ανάγκες. [10]

2.1.1 iOS runtime system

Κατά την εκκίνηση μιας iOS συσκευής το λειτουργικό σύστημα ξεκινάει να τρέχει ένα σέτ προγραμμάτων που ανήκουν στο συστήμα. Αυτό το σύνολο των προγραμμάτων το οποίο είναι το iOS runtime σύστημα τρέχει συνεχώς στο παρασκήνιο και διαχειρίζεται κάθε εφαρμογή που εκτελείται. Ουσιαστικά η εφαρμογή δεν είναι τίποτα περισσότερο από ένα εκτελέσιμο πρόγραμμα (όπως ένα .exe πρόγραμμα στα Windows) που εκτελείται στην συσκευή και αλληλεπιδρά με το iOS runtime σύστημα. Η αρχική οθόνη σε μία iOS συσκευή απλά εμφανίζει εικονίδια για όλα αυτά τα εκτελέσιμα προγράμματα. Όταν πιεστεί το εικονίδιο το λειτουργικό σύστημα εκκινεί το εκτελέσιμο αρχείο που αντιστοιχεί στο εικονίδιο και προκαλεί την εκτέλεση του προγράμματος στην συσκευή. [11]

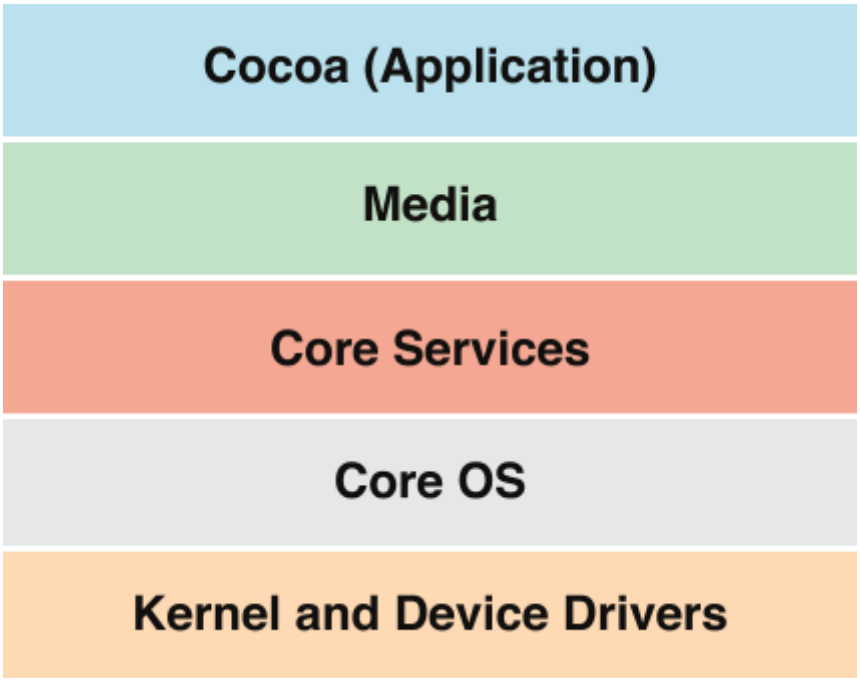
2.2 Διαστρωματωμένη αρχιτεκτονική του iOS

Είναι χρήσιμο να δούμε το λειτουργικό iOS ως ένα σύνολο στρωμάτων. Τα χαμηλότερα στρώματα του συστήματος παρέχουν τις βασικές υπηρεσίες στις οποίες βασίζεται όλο το λογισμικό. Τα ανώτερα επίπεδα περιέχουν πιο εξελιγμένες υπηρεσίες και τεχνολογίες, στις οποίες βασίζονται ή συμπληρώνονται τα κατώτερα επίπεδα. Όσο χαμηλότερο είναι το επίπεδο τόσο πιο εξειδικευμένες είναι οι υπηρεσίες που παρέχει. Γενικά οι τεχνολογίες σε ανώτερα στρώματα ενσωματώνουν τεχνολογίες χαμηλότερου επιπέδου για την σωστή παροχή κοινών συμπεριφορών στις εφαρμογές, για αυτό τον λόγο είναι προτιμότερη η χρήση του ανώτατου επιπέδου προγραμματιστικής διεπαφής που ανταποκρίνεται στους στόχους της εφαρμογής. [12]

2.2.1 Τα επίπεδα διαστρωματωμένης αρχιτεκτονικής σε iOS

- Η Cocoa (Application), είναι το επίπεδο που περιλαμβάνει τεχνολογίες για την δημιουργία διεπαφής χρήστη μιας εφαρμογής, προκειμένου να υπάρχει ανταπόκριση στα συμβάντα των χρηστών και τη διαχείριση της συμπεριφοράς των εφαρμογών.

- Το Media επίπεδο περιλαμβάνει εξειδικευμένες τεχνολογίες αναπαραγωγής, εγγραφής και επεξεργασίας οπτικοακουστικών μέσων καθώς και προβολής γραφικών 2D και 3D.
- Το Core Services στρώμα περιέχει τις θεμελιώδεις υπηρεσίες και τεχνολογίες συστήματος που χρησιμοποιούν οι εφαρμογές, από την επικοινωνία δικτύου χαμηλού επιπέδου μέχρι την μορφοποίηση των δεδομένων.
- Το Core OS επίπεδο ορίζει ποιες διεπαφές προγραμματισμού σχετίζονται με το υλικό και την δικτύωση, συμπεριλαμβανωμένων των διεπαφών για την εκτέλεση εργασιών υπολογισμού υψηλής απόδοσης σε CPU και GPU μιας συσκευής.
- Το Kernel and Device Drivers επίπεδο αποτελείται από το περιβάλλον του Mach (Kernel), τους οδηγούς συσκευών, τις λειτουργίες της βιβλιοθήκης BSD (libSystem) και άλλα στοιχεία χαμηλού επιπέδου. Επίσης περιλαμβάνει υποστήριξη για συστήματα αρχείων, δικτύωση, ασφάλεια, επικοινωνία μεταξύ διαδικασιών, γλώσσες προγραμματισμού, προγράμματα οδήγησης συσκευών και επεκτάσεις στον πυρήνα. [13]



Εικόνα 1.3
Διάγραμμα λειτουργικού εφαρμογής

2.3 iOS εφαρμογές(apps)

Μία iOS εφαρμογή είναι ένα μεταγλωττισμένο εκτελέσιμο αρχείο μαζί με ένα σύνολο υποστηρικτικών αρχείων σε ένα bundle, σε ένα κατάλογο δηλαδή με συγκεκριμένη δομή και επέκταση αρχείου (file extension), επιτρέποντας σε σχετικά αρχεία να ομαδοποιηθούν μεταξύ τους σε ένα ενιαίο στοιχείο. Το application bundle αποτελεί ένα package για εγκατάσταση σε μία συσκευή ή μεταφόρτωση στο App Store. Μία εφαρμογή περιέχει μια σειρά από διαφορετικούς τύπους αρχείων τα οποία χρησιμοποιούνται τόσο κατά την χρόνο μεταγλώττισης όσο και κατά την διάρκεια εκτέλεσης. Αυτά τα αρχεία περιλαμβάνουν

- Το info.plist αρχείο το οποίο περιέχει πληροφορίες σχετικά με τις γλώσσες τις οποίες πρόκειται να υποστηρίξει η εφαρμογή, το ID της εφαρμογής και τις απαιτούμενες επιλογές διαμόρφωσης όπως είναι οι υποστηριζόμενοι τύποι διεπαφών (iPhone, iPad και Universal) καθώς και διαθέσιμους προσανατολισμούς (πορτραίτο, upside down κλπ)
- Κανένα ή περισσότερα interface builder αρχεία με . xib επέκταση τα οποία περιέχουν user interface οθόνες και αντικαθιστούν τα προηγούμενα . nib αρχεία.
- Κανένα ή περισσότερα αρχεία εικόνας με επέκταση . xcassets, τα οποία αποθηκεύουν ομαδοποιημένα σχετικά μεταξύ τους εικονίδια σε διαφορετικά μεγέθη όπως το εικονίδιο της εφαρμογής και γραφικά εμφάνισης στην οθόνη, αντικαθιστώντας τα . icns αρχεία.
- Κανένα ή περισσότερα Storyboard αρχεία με . storyboard επέκταση για την γραφική απεικόνιση και συντονισμό μεταξύ διαφορετικών οθονών μιας εφαρμογής.
- Ένα ή περισσότερα . swift αρχεία τα οποία περιλαμβάνουν τον κώδικα της εφαρμογής. [14]

2.4 iOS SDK(Software Development Kit)

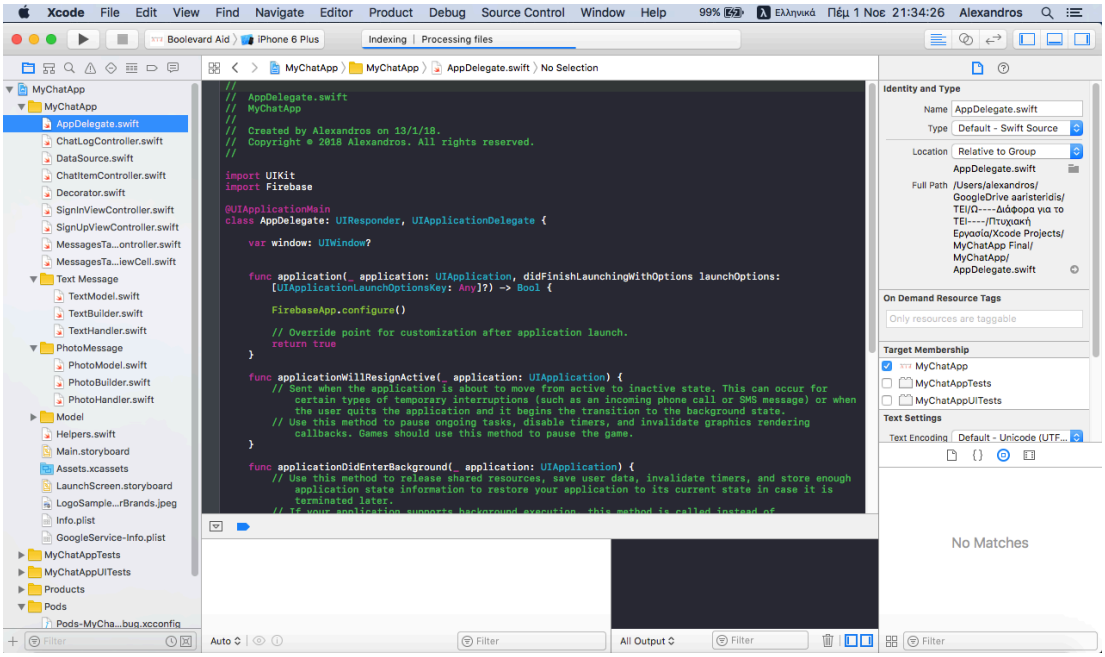
Το iOS software development kit είναι κάποια εργαλεία ανάπτυξης λογισμικού σχεδιασμένα από την Apple που επιτρέπουν σε προγραμματιστές την ανάπτυξη εφαρμογών για το

συγκεκριμένο λειτουργικό σύστημα. Το SDK είναι διαθέσιμο δωρεάν για τους χρήστες Mac OS μέσω του Mac App Store, το λειτουργικό σύστημα δηλαδή που διαθέτει η Apple για τους υπολογιστές της. Το SDK περιέχει ένα σύνολο από εργαλεία τα οποία δίνουν την δυνατότητα στους προγραμματιστές να έχουν πρόσβαση σε λειτουργίες και υπηρεσίες των συσκευών iOS, όπως χαρακτηριστικά του hardware και του software. Περιέχει επίσης προσομοιωτές για κάθε συσκευή της Apple για την οποία προορίζεται η εφαρμογή, σχετικά με την εμφάνιση και αίσθηση της συσκευής κατα την ανάπτυξη της εφαρμογής. Οι νέες εκδόσεις iOS συνοδεύονται απο νέες εκδόσεις SDK. Σχετικά με μια σειρά απο δυνατότητες όπως ο έλεγχος της εφαρμογής, τεχνική υποστήριξη και διανομή της εφαρμογής στο App Store οι προγραμματιστές υποχρεούνται να εγγραφούν στο Apple Developer Program με συνδρομή 86€ κατ'ετος. Μαζί με το Xcode, το πρόγραμμα που περιέχει το SDK, η Apple και το iOS SDK βοηθάει τους προγραμματιστές να γράψουν iOS εφαρμογές στις επίσημες υποστηριζόμενες γλώσσες προγραμματισμού όπως η Swift και η Objective-C. Πολλές ακόμα εταιρίες έχουν δημιουργήσει εργαλεία που σου δίνουν την δυνατότητα για ανάπτυξη μιας iOS εφαρμογής, οι οποίες χρησιμοποιούν τις αντίστοιχες δικές τους γλώσσες προγραμματισμού όχι όμως με τα ίδια αποτελέσματα. Για τον λόγο αυτό, προκειμένου να βοηθήσουν τους προγραμματιστές, κολοσσοί της τεχνολογίας έχουν αναπτύξει τα δικά τους SDK για τις εταιρίες τους, προκειμένου να υπάρχει καλύτερη εμπειρία χρήστη, με κερδισμένους και τους ίδιους και τους developers, οι οποίοι το μοναδικό πράγμα που χρειάζεται να κάνουν, είναι η εισαγωγή του SDK στο project τους. [15]

2.4.1 Εισαγωγή στο Xcode IDE

Το Xcode είναι ένα ολοκληρωμένο περιβάλλον ανάπτυξης λογισμικού (IDE) σχεδιασμένο να τρέχει σε MacOS το οποίο περιλαμβάνει μια σουίτα εργαλείων ανάπτυξης λογισμικού και ανήκει στο iOS SDK το οποίο παρέχεται από την Apple. Περιέχει δηλαδή σε ένα πακέτο λογισμικού τα εργαλεία εκείνα που απαιτούνται για την δημιουργία μιας εφαρμογής (text editor, μεταγλωττιστή και Interface Builder) προκειμένου να παράγονται εκτελέσιμα αρχεία. Τα εργαλεία αυτά είναι υπεύθυνα για την διαχείριση ολόκληρης της ροής εργασιών, από την δημιουργία μιας εφαρμογής έως τη δοκιμή της, την βελτιστοποίηση και την τελική υποβολή της στο App Store, συγκεντρωμένα σε ένα πακέτο λογισμικού χωρίς οι εφαρμογές να είναι ανεξάρτητες μεταξύ τους

και χωρίς να συνδέονται μεταξύ τους μέσω scripts όπως συμβαίνει σε άλλα περιβάλλοντα ανάπτυξης λογισμικού. Είναι με απλά λόγια μια εφαρμογή που παράγει εφαρμογές. [16]



Εικόνα 1.4
Γραφικό περιβάλλον Xcode

Το Xcode πρωτοκυκλοφόρησε το έτος 2003 σαν Project Builder από την Apple με σκοπό την ανάπτυξη MacOS εφαρμογών αλλά στην συνέχεια, και συγκεκριμένα το έτος 2004, εξελίχθηκε και μετονομάστηκε σε Xcode. Στις μέρες μας είναι διαθέσιμο μέσω του Mac App Store δωρεάν, υποστηρίζει δε ανάπτυξη εφαρμογών όχι μόνο για MacOS αλλά και για iOS, watchOS και tvOS. Ο όρος Xcode χρησιμοποιείται με δύο τρόπους και συγκεκριμένα αποτελεί το όνομα του προγράμματος το οποίο χρησιμοποιείται για την επεξεργασία και την δημιουργία μιας εφαρμογής ενώ επιπρόσθετα αποτελεί το όνομα μια ολόκληρης σουίτας βοηθητικών προγραμμάτων, τα οποία περιέχουν μεταξύ των υπολοίπων προγραμμάτων και το Xcode σαν πρόγραμμα αυτούσιο όπως αναλύθηκε παραπάνω. [17]

Το Xcode υποστηρίζει πηγαίο κώδικα για μια πληθώρα γλωσσών προγραμματισμού όπως είναι η Swift, Objective-C, C++, C, Python, Java κλπ καθώς και την δυνατότητα οι εφαρμογές που παράγει να υποστηρίζονται από πολλούς και διαφορετικούς επεξεργαστές. Τέλος ένα από τα μεγαλύτερα πλεονεκτήματα του Xcode είναι ότι μέσω του Simulator που διαθέτει υπάρχει η

δυνατότητα για λειτουργία προσομείωσης της εφαρμογής σε όλα τα στάδια κατασκευής της με οπτική απεικόνιση του app καθώς επίσης και συλλογής στατιστικών δεδομένων του app τα οποία αποδεικνύονται χρήσιμα για τους προγραμματιστές. [18]

2.4.2 AppDelegate

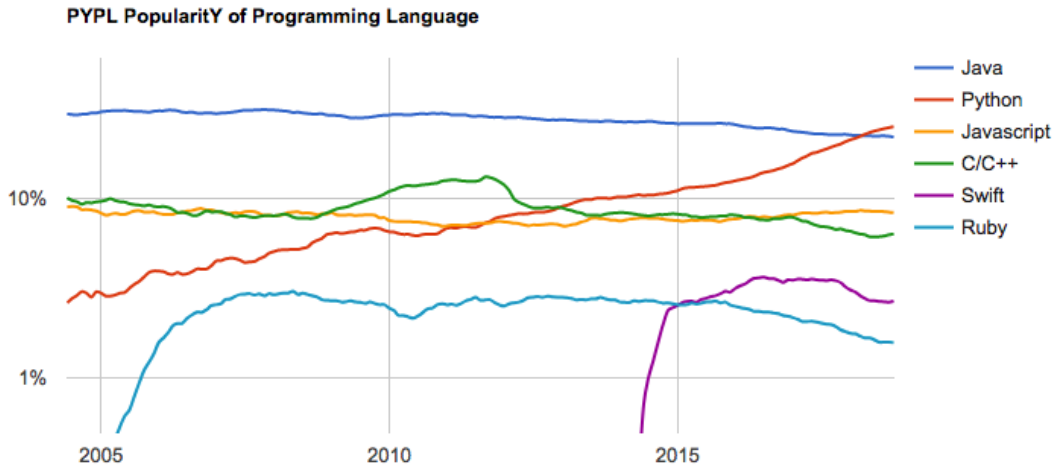
Κατά την διάρκεια κατασκευής μιας iOS εφαρμογής στο Xcode το app συνδέεται με το βασικό κύριο πρόγραμμα καθώς και με ένα συγκεκριμένο στοιχείο της εφαρμογής που παράγεται από το Xcode IDE το appdelegate. Το κύριο πρόγραμμα μαζί με το appdelegate χρησιμεύουν ως η διεπαφή μεταξύ του app και του iOS runtime. Αυτά τα στοιχεία ασχολούνται με τα συμβάντα στο User Interface όπως η επαφή του χρήστη με την οθόνη και τα συμβάντα συστήματος. Κυριότερη όμως λειτουργία του appdelegate είναι η επικοινωνία με το λειτουργικό σύστημα για βασικές λειτουργίες της εφαρμογής μέσω της κλήσης συγκεκριμένων μεθόδων του appdelegate όπως για παράδειγμα ο έλεγχος σχετικά με το αν σταμάτησε η εκτέλεση του app, αν η εφαρμογή θα πρέπει τερματιστεί κλπ. [19]

2.4.3 Εισαγωγή στην Swift

Η Swift είναι μία αντικειμενοστραφής γλώσσα προγραμματισμού που παρουσιάστηκε από την Apple τον Ιούνιο του έτους 2014 στο παγκόσμιο συνέδριο των προγραμματιστών της και αντικατέστησε την Objective-C που χρησιμοποιούσε μέχρι τότε, συνεχίζοντας όμως να λειτουργεί παράλληλα με αυτή. Στο αντίστοιχο συνέδριο του 2015 η Apple έκανε μια εξίσου σημαντική ανακοίνωση μαζί με την παρουσίαση της Swift 2 και συγκεκριμένα την μετεξέλιξη της γλώσσας σε open source διαθέτοντας το https://swift.org/ για την κοινότητά της. Το έτος 2016 έκανε την εμφάνισή της η Swift 3 περιέχοντας θεμελιώδεις αλλαγές στην ίδια την γλώσσα όπως και στην βασική βιβλιοθήκη της Swift, όμως με ένα βασικό μειονέκτημα, το οποίο ήταν ότι δεν ήταν συμβατή με τις προηγούμενες εκδόσεις της γλώσσας. Ένας από τους βασικούς στόχους της Swift 3 ήταν να είναι συμβατή και σε άλλες πλατφόρμες έτσι ώστε ο κώδικας που γραφόταν σε μία πλατφόρμα όπως είναι το MacOS να είναι συμβατός και σε άλλη πλατφόρμα, όπως λόγου χάρη στην Linux.

Εικόνα 1.5
Λογότυπο Swift

Κατά την διάρκεια της συγγραφής της παρούσας πτυχιακής εργασίας η Apple έχει κυκλοφορήσει την Swift 4. Πρωταρχικός σκοπός του μεταγλωττιστή της Swift 4 είναι η συμβατότητα με την Swift 3. Αυτό μας επιτρέπει να γράψουμε project σε Swift 3 ή Swift 4 ή και στις δύο με τον μεταγλωττιστή της Swift 4.



Πίνακας 1.4
Η Swift σε σχέση με τις άλλες γλώσσες προγραμματισμού

2.4.3.1 Η Swift σε συνάρτηση με την Objective-C

Όταν η Apple πρωτοσύστησε στο κοινό την Swift, την παρουσίασε σαν την Objective-C χωρίς την C. Η Objective-C, η οποία είναι ένα υπερσύνολο της C που της προσφέρει

αντικειμενοστραφείς δυνατότητες προγραμματισμού και την αναβαθμίζει σε δυναμική γλώσσα προγραμματισμού. Αυτό σήμαινε ότι με την Objective-C η Apple θα δεσμευόταν να διατηρήσει την συμβατότητά της με την C, πράγμα όμως που θα περιόριζε τις βελτιώσεις που θα μπορούσε να κάνει στην γλώσσα. Από την στιγμή που η Swift δεν είχε τέτοιου είδους περιορισμούς σχετικά με την συμβατότητα με την C, η Apple ήταν ελεύθερη να προσθέσει οποιοδήποτε χαρακτηριστικό πίστευε ότι θα μπορούσε να βελτιστοποιήσει την γλώσσα. Έτσι λοιπόν η Apple είχε την δυνατότητα να συμπεριλάβει στην Swift τα καλύτερα χαρακτηριστικά από τις πιο διαδεδομένες γλώσσες προγραμματισμού στις μέρες μας όπως η Objective-C, η Java, η Python, η Ruby και πολλές άλλες. Όταν αναφέρουμε ότι η Swift είναι μία δυναμική γλώσσα προγραμματισμού εννοούμε ότι υπάρχει η δυνατότητα στους τύπους της να μεταβληθούν κατά την διάρκεια του χρόνου εκτέλεσής τους. Αυτό περιλαμβάνει είτε την προσθήκη νέων προσαρμοσμένων τύπων είτε την αλλαγή/επέκταση στους υπάρχοντες τύπους. Παρόλο που υπάρχουν πολλές ομοιότητες μεταξύ των δύο υπάρχουν αντίστοιχα και σημαντικές διαφορές. Η σύνταξη και η μορφοποίηση της Swift είναι πολύ πιο κοντά με την Python από ότι με την Objective-C αλλά η Apple κράτησε τα άγκιστρα (curly braces) μετατρέποντάς τα απαραίτητα σε δηλώσεις ελέγχου όπως είναι το if και το while, με τα οποία εξαλείφονται τα σφάλματα. [20]

2.4.4 Swift standard library

Η Standard library της Swift ορίζει μια βάση από κανόνες λειτουργιών για την συγγραφή μιας εφαρμογής σε γλώσσα Swift. Η βιβλιοθήκη αυτή ορίζει τους θεμελιώδεις τύπους δεδομένων όπως είναι το Int, το String και το Double. Επίσης προσδιορίζει ποιες συλλογές, global functions και πρωτόκολλα είναι σύμφωνα με αυτούς τους τύπους δεδομένων. [21]

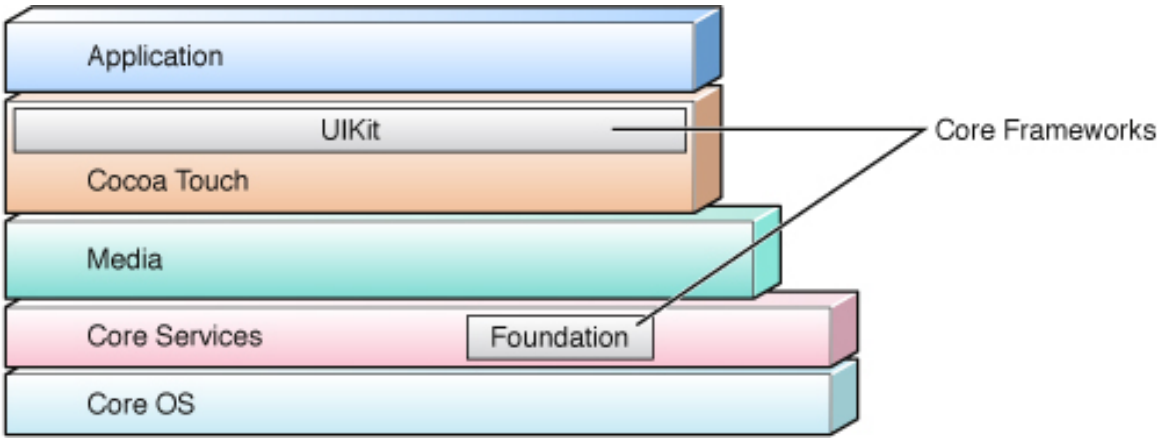
2.5 Software frameworks (Πλαίσια)

Για την διευκόλυνση των προγραμματιστών κατά την ανάπτυξη μιας εφαρμογής εταιρίες ανάπτυξης λογισμικού αλλά και ανεξάρτητοι προγραμματιστές αναπτύσσουν τις περισσότερες από τις διεπαφές του συστήματος, οργανωμένες σε ειδικά πακέτα που ονομάζονται πλαίσια (frameworks). Ένα framework είναι ένας κατάλογος που περιέχει μια δυναμική κοινή βιβλιοθήκη λογισμικού και τους πόρους (όπως αρχεία επικεφαλίδας, εικόνες κ.α.) που χρειάζονται για την υποστήριξη εφαρμογών για ένα συγκεκριμένο περιβάλλον, λειτουργεί δηλαδή ως

υποστηρικτικός σκελετός στην κατασκευή μιας εφαρμογής. Ο σκοπός του σχεδιασμού των frameworks είναι να μειωθούν τα προβλήματα που δημιουργούνται κατά την ανάπτυξη εφαρμογών και αυτό επιτυγχάνεται με την χρήση προκατασκευασμένου-ελεγμένου κώδικα, ενώ μπορεί να γίνει κοινή η χρήση του μεταξύ διαφορετικών ενοτήτων μίας εφαρμογής ελαχιστοποιώντας πιθανά σφάλματα. Προκειμένου να χρησιμοποιηθεί κάποιο framework απαιτείται η σύνδεση της εφαρμογής με αυτό, όπως γίνεται με οποιαδήποτε κοινή βιβλιοθήκη. Η σύνδεση αυτή δίνει την απαραίτητη πρόσβαση στα χαρακτηριστικά του framework. [22] Τα frameworks σαν έννοια δεν έχουν μεγάλη διαφορά από μία βιβλιοθήκη, η κυριότερη δε διαφορά τους έγκειται στον σκοπό για τον οποίο προορίζονται. Από τη μία, τα frameworks εμπεριέχουν μια πληθώρα επιλογών από σύνθετες λειτουργίες, από την άλλη δε οι βιβλιοθήκες προορίζονται να εκπληρώσουν μικρότερες και συγκεκριμένες λειτουργίες. Συμπερασματικά, η χρήση των frameworks αποσκοπεί στο ξεκλείδωμα όλων των χαρακτηριστικών και των λειτουργιών του υλικολογισμικού για το οποίο διατίθενται. [23]

2.5.1 Cocoa framework

Το Cocoa είναι η native διεπαφή προγραμματισμού εφαρμογών (Application Programming Interface) της Apple για το λειτουργικό της σύστημα Mac OS. Για τα υπόλοιπα λειτουργικά συστήματα της Apple (iOS, tvOS, watchOS) υπάρχει μία άλλη παρόμοια διεπαφή προγραμματισμού εφαρμογών η Cocoa Touch βασισμένη στην εργαλειοθήκη του Cocoa API, ακολουθώντας και τα δύο την Model View Controller (MVC), η οποία είναι αρχιτεκτονική λογισμικού προσφέροντας όμως διαφορετικά χαρακτηριστικά το καθένα. Το Cocoa Touch είναι ένα user interface (UI) framework κυρίως γραμμένο σε γλώσσα Objective-C το οποίο επιτρέπει την χρήση του hardware και των λειτουργιών του, οι οποίες είναι όμως διαφορετικές στις συσκευές με λειτουργικό iOS και Mac OS, κυρίως λόγο των υπολογιστικών πόρων, οι οποίοι είναι περιορισμένοι στις κινητές συσκευές σε σχέση με desktop υπολογιστές. Το Cocoa framework και το Cocoa Touch framework μπορούμε να τα δούμε σαν δύο μεγάλα frameworks τα οποία είναι διαιρεμένα σε πολλά μικρότερα frameworks. [24]



Εικόνα 1.6
Cocoa frameworks μέσα στο διάγραμμα λογισμικού iOS εφαρμογής

Τα σημαντικότερα frameworks που περιέχει το Cocoa API είναι το Foundation Kit framework, το Application Kit framework και το User Interface Kit framework τα οποία περιλαμβάνονται στο αρχείο κεφαλίδας (header file) cocoa.h όπως επίσης και οι βιβλιοθήκες και τα frameworks που εμπεριέχουν αυτά, όπως για παράδειγμα η C standard library και το Objective-C runtime. [25]

- Foundation Kit framework-το οποίο ανήκει και στο Cocoa και στο Cocoa Touch- εμπεριέχει την NSObject κλάση για τον προσδιορισμό της συμπεριφοράς του object. Επίσης διαθέτει κλάσεις για βασικούς τύπους, αποθήκευση δεδομένων, πρόσβαση σε φακέλους του συστήματος, επεξεργασία κειμένου, inter-app ειδοποιήσεις κ.α.
- AppKit framework είναι ένα Cocoa framework για την ανάπτυξη εφαρμογών σε Mac OS GUIs (Γραφικό περιβάλλον χρήστη). Το App Kit παρέχει controls, παράθυρα, μενού, κουμπιά, pannels, πλαισια κειμένου καθώς και γραφικά, push notifications, λειτουργίες πρόσβασης σε AMEA κ.α.
- UIKit framework είναι το αντίστοιχο AppKit framework του Cocoa Touch για ανάπτυξη σε iOS GUIs κινητών συσκευών επιπλέον δε διαθέτει multi-touch interface control το οποίο είναι απαραίτητο για mobile apps, event handling για αισθητήρες κίνησης κ.α.

List of Cocoa Touch frameworks

Cocoa Touch Layer	AssetsLibrary	OpenAL	CoreLocation	Social
AddressBookUI	AudioToolbox	OpenGLES	CoreMedia	StoreKit
EventKitUI	AudioUnit	Photos	CoreMotion	SystemConfig-uration
GameKit	CoreAudio	QuartzCore	CoreTelephony	UIAutomation
MapKit	CoreGraphics	SceneKit	EventKit	WebKit
MessageUI	CoreImage	SpriteKit	Foundation	
Notification-Center	CoreMIDI		HealthKit	
PhotosUI	CoreText	***Core Services Layer***	HomeKit	***Core OS Layer***
Twitter	CoreVideo	Accounts	JavaScriptCore	Accelerate
UIKit	GLKit	AdSupport	MobileCore-Services	CoreBluetooth
iAd	GameController	AddressBook	Multipeer-Connectivity	ExternalAccessory
	ImageIO	CFNetwork	NewsstandKit	LocalAuthen-tication
Media Layer	MediaAccess-ibility	CloudKit	PassKit	Security
AVFoundation	MediaPlayer	CoreData	QuickLook	System
	Metal	CoreFoundation		

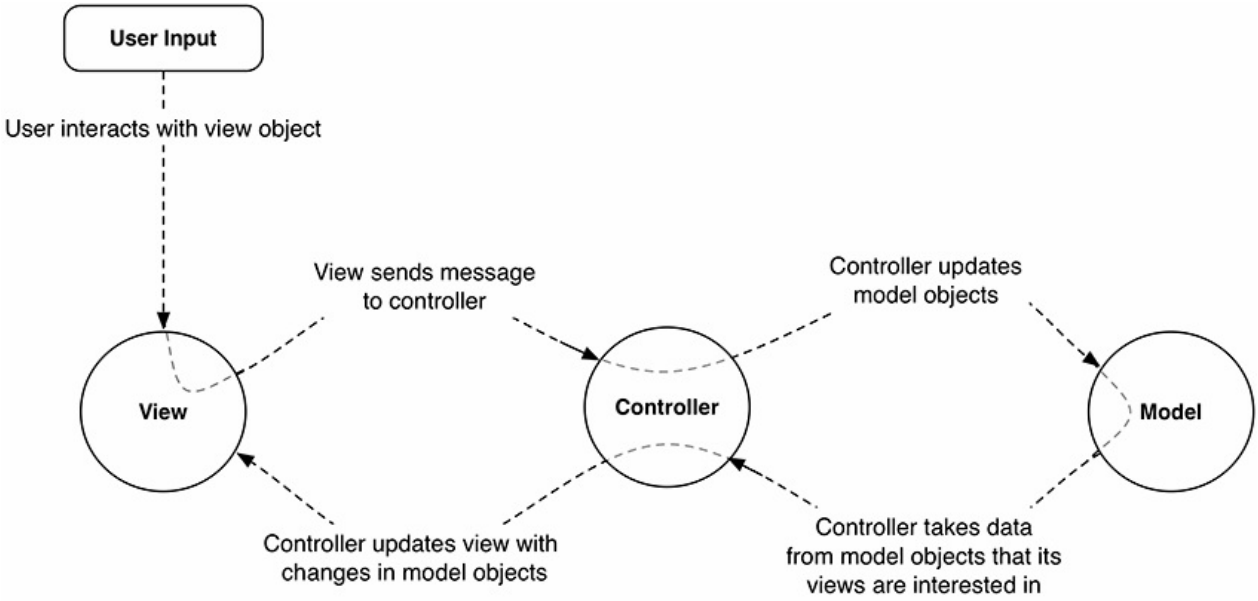
Εικόνα 1.7
Λίστα Cocoa Touch frameworks

2.6 Model View Controller (MVC)

Για την αντιμετώπιση μικρών και μεγάλων προβλημάτων κατά την ανάπτυξη λογισμικού οι προγραμματιστές οδηγήθηκαν στην ανάγκη δημιουργίας μιας συγκεκριμένης δομής ενός προγράμματος, ένα αρχιτεκτονικό πρότυπο (architectural pattern). [26] Η λύση ήταν η διαίρεση ενός προγράμματος σε μικρότερα τμήματα και αποθήκευση κοινών τμημάτων σε ξεχωριστά αρχεία. Με αυτόν τον τρόπο είναι πιο εύκολο να βρούμε την ενότητα του προγράμματος η οποία πρέπει να τροποποιηθεί σε περίπτωση σφάλματος όπως επίσης και την δυνατότητα ταυτόχρονης συνεργασίας στο ίδιο πρόγραμμα δύο ή περισσοτέρων προγραμματιστών, οι οποίοι μπορούν να ασχολούνται με διαφορετικά αρχεία του προγράμματος. [27] Ως εκ τούτου, καταλήγουμε στο συμπέρασμα ότι με την διαίρεση ενός προγράμματος σε ξεχωριστά αρχεία είναι πιο εύκολο το σύνολό του να παραμείνει οργανωμένο.

Το πρότυπο είναι αρκετά εύκολο να γίνει κατανοητό, και συγκεκριμένα ο κώδικας ενός προγράμματος MVC διαχωρίζεται σε επίπεδο δεδομένων-Model layer, ένα User Interface επίπεδο-View layer και ένα τρίτο επίπεδο-Controller layer το οποίο συνδέει τα άλλα δύο επίπεδα.

Αυτό λοιπόν επιτρέπει έναν λογικό διαχωρισμό μέσα στον κώδικα με βάση την λειτουργία του. [26]



Εικόνα 1.8
Model View Controller πρότυπο

2.6.1 Ανάλυση MVC προτύπου

Το User Interface (View) είναι αυτό που βλέπει ο χρήστης. Σκοπός κάθε UI είναι να εμφανίζει πληροφορίες και να δέχεται δεδομένα και εντολές από τον χρήστη. Τα παλαιότερα χρόνια οι προγραμματιστές δημιουργούσαν UI γράφοντας κώδικα. Παρόλο που μπορούμε να το κάνουμε και με την γλώσσα Swift, αυτό είναι ιδιαιτέρως χρονοβόρο και επιρρεπές σε σφάλματα και πιθανόν να μην συμπεριφέρεται με τον ίδιο τρόπο ανάμεσα σε διαφορετικούς προγραμματιστές. Το ιδανικότερο είναι η σχεδίαση του UI οπτικά, κάτι που μας επιτρέπει να κάνουμε το Xcode απλά σχεδιάζοντας αντικείμενα στο UI όπως κουμπιά, πλαίσια κειμένου και μενού, που το Xcode δημιουργεί αυτόματα χωρίς σφάλματα UI. Όταν δημιουργείται ένα UI μέσω Xcode στην πραγματικότητα χρησιμοποιείται το Cocoa framework για την δημιουργία του. Από μόνο του όμως ένα UI μπορεί να είναι όμορφο αλλά δεν μπορεί να κάνει τίποτα. Προκειμένου να κάνουμε ένα πρόγραμμα να μπορεί να κάνει κάτι χρήσιμο, πρέπει να γράψουμε κώδικα για την εκτέλεση μίας συγκεκριμένης λειτουργίας.

Το Model είναι τελείως απομονωμένο από το View (UI). Αυτό διευκολύνει την τροποποίηση του UI χωρίς να επηρεαστεί το Model και αντίστροφα. Διατηρώντας διαφορετικά τμήματα του προγράμματος όσο γίνεται πιο απομονωμένα από άλλα μέρη του προγράμματος μειώνεται η πιθανότητα εμφάνισης σφαλμάτων κατά την τροποποίηση ενός προγράμματος. Από την στιγμή που το Model είναι πάντα απομονωμένο από το View χρειάζεται ένας Controller. Όταν ένας χρήστης επιλέγει μια εντολή ή εισάγει δεδομένα στο UI ο Controller παίρνει τις πληροφορίες από το View και τις μεταβιβάζει στο Model. Το Model ανταποκρίνεται σε αυτά τα δεδομένα ή εντολές και υπολογίζει ένα νέο αποτέλεσμα, το οποίο στέλνει στον Controller και τέλος εκείνο το προωθεί πίσω στο View ώστε να εμφανιστεί στο UI προκειμένου ο χρήστης να το δει. Καθ' όλη την διάρκεια αυτής της διαδικασίας, View και Model παραμένουν τελείως ξεχωριστά το ένα από το άλλο. [27]

2.7 Native development

Η γλώσσα Swift και η γλώσσα Objective-C σε συνδυασμό με τα Cocoa frameworks αλλά και την χρήση του Xcode μας επιτρέπει την ανάπτυξη κάθε δυνατής εφαρμογής. Οι εφαρμογές που έχουν αναπτυχθεί με αυτές τις τεχνολογίες ονομάζονται native εφαρμογές. Υπάρχουν και άλλα εργαλεία όμως ανάπτυξης εφαρμογών-μη Apple's frameworks- που συγκαταλέγονται στην hybrid τεχνολογία, και συγκεκριμένα κάποια SDK's frameworks όπως το Titanium Appcelerator με το οποίο γράφεις σε JavaScript, το PhoneGap με το οποίο γράφεις σε JavaScript, HTML και CSS κ.α. Τα πλεονεκτήματα αυτών των μη native frameworks είναι τα εξής

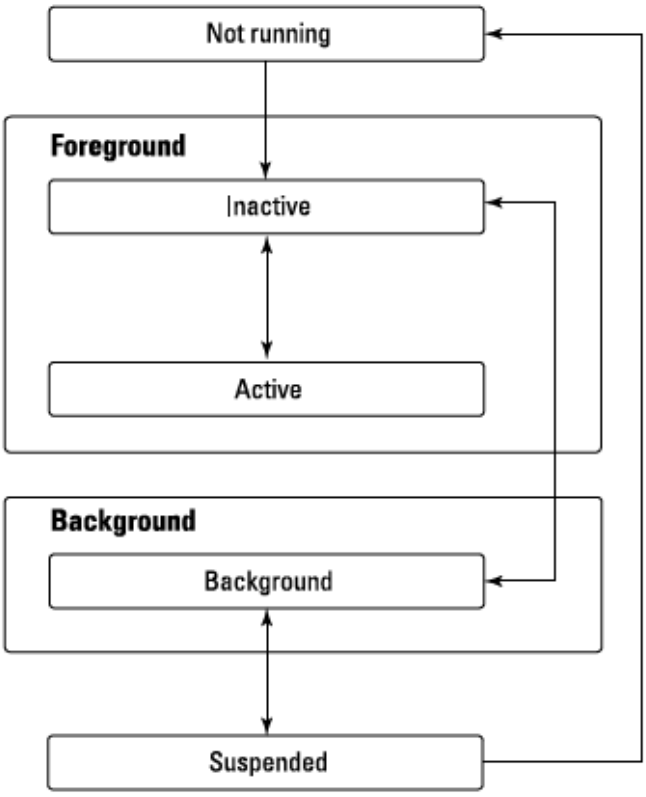
- Με αυτά τα frameworks η διαδικασία ανάπτυξης εφαρμογών είναι ταχύτερη από ότι με native app frameworks
- Με την χρήση αυτών των frameworks μπορούμε να αναπτύξουμε εφαρμογές που είναι συμβατές με διαφορετικές πλατφόρμες (cross-platform apps)

Ωστόσο τα μη native frameworks έχουν και ένα πολύ μεγάλο μειονέκτημα, καθότι εργαλεία από τρίτους κατασκευαστές δεν είναι σίγουρο ότι θα υποστηρίζουν τα νέα χαρακτηριστικά (πιθανόν όχι και όλα τα τρέχοντα) από το υλικολογισμικό της Apple. Γι' αυτό τον λόγο είναι ουσιώδης αυτός ο διαχωρισμός μεταξύ native και hybrid τεχνολογίας, καθότι αν και είναι πιο κοστοβόρος και χρονοβόρος, η διάρκεια ζωής των native apps είναι μεγαλύτερη. Επιπλέον όμως

η native τεχνολογία μας δίνει την δυνατότητα να αναπτύξουμε τις ικανότητές μας σε αυτόν τον ραγδαία αναπτυσσόμενο χώρο του mobile. [28]

2.8 Κύκλος λειτουργίας ενός iOS app

Μία iOS εφαρμογή ακολουθεί έναν τυπικό κύκλο λειτουργίας. Στην αρχή το app είναι απλά ένα εκτελέσιμο στοιχείο, δεν τρέχει και περιμένει υπομονετικά μέχρι ο χρήστης να πατήσει στο εικονίδιο. Όταν η εφαρμογή ξεκινάει περνάει από πολλά στάδια αρχικοποιήσεων. Κατά την διάρκεια αυτής της διαδικασίας το app βρίσκεται σε ανενεργή κατάσταση. Η εφαρμογή πράγματι τρέχει στο προσκήνιο αλλά δεν λαμβάνει γεγονότα και έτσι δεν αλληλοεπιδρά με τίποτα και κανέναν. Στην συνέχεια η εφαρμογή μεταβαίνει στην ενεργή κατάσταση, η οποία αποτελεί την χρήσιμη κατάσταση μιας εφαρμογής.



Εικόνα 1.9
Στάδια λειτουργίας ενός iOS app

Κάποια στιγμή, κυρίως όταν κάποιο άλλο app ξεκινάει, το iOS runtime βάζει την εφαρμογή στο παρασκήνιο. Σε εκείνο το σημείο η εφαρμογή είναι σε κατάσταση παρασκηνίου. Τα

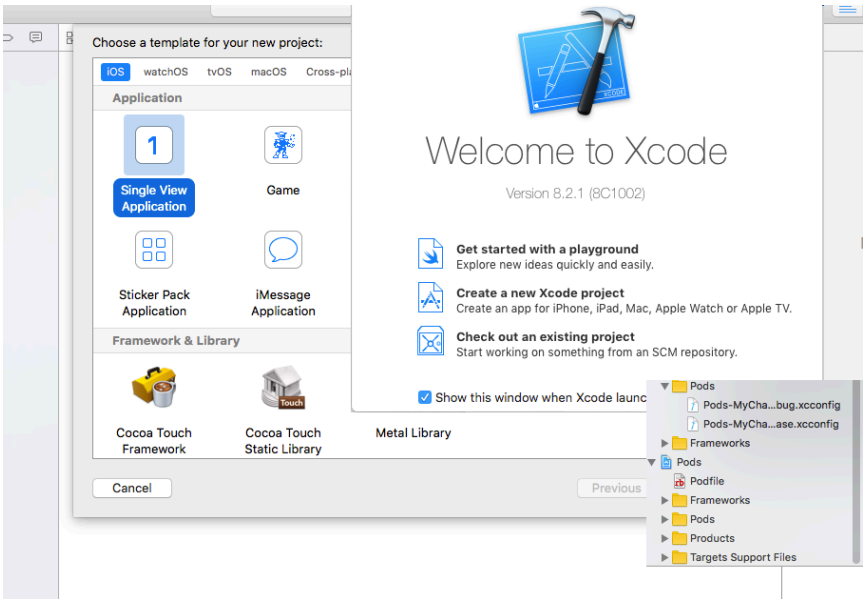
περισσότερα apps μένουν σ' αυτή την κατάσταση για μικρό χρονικό διάστημα μέχρι να μεταβληθεί η κατάστασή τους και να ανασταλεί η λειτουργία τους. Υπάρχει όμως και η πιθανότητα ένα app να ζητήσει επιπλέον χρόνο για να ολοκληρώσει κάποιες διαδικασίες (π.χ. το κλείσιμο ενός εγγράφου στο σημείο που βρίσκεται). Επιπλέον υπάρχουν και τα apps που προορίζονται για το παρασκήνιο και ξεκινούν και παραμένουν σε αυτήν την κατάσταση, τα οποία τέλος μπορούν να λαμβάνουν γεγονότα ακόμα και αν δεν έχουν ορατό User Interface. [29]

3 Υλοποίηση της εφαρμογής Boolevard Aid

Πριν κάνουμε οτιδήποτε θα πρέπει να εγκαταστήσουμε το CocoaPods το οποίο θα χρειαστούμε κατά την διάρκεια κατασκευής αυτής της εφαρμογής. Ανοίγουμε λοιπόν το τερματικό στον υπολογιστή και γράφουμε την παρακάτω εντολή $ sudo gem install cocoapods. Το CocoaPods είναι ένας διαχειριστής εξαρτήσεων των βιβλιοθηκών σε Xcode projects, δηλαδή κατανέμει, επεξεργάζεται και εισάγει third party κώδικα και βιβλιοθήκες στο project. Αυτές οι εξαρτήσεις ορίζονται σε ένα αρχείο κειμένου που ονομάζεται Podfile. [30]

3.1 Δημιουργία project-Εισαγωγή frameworks

Δημιουργούμε ένα καινούργιο project στο Xcode, ανοίγουμε το Xcode και επιλέγουμε Create a new Xcode project→iOS Single View Application και στην συνέχεια προσθέτουμε το όνομα του project μας, BoolevardAid. Έπειτα κλείνουμε το Xcode και μέσω του terminal οδηγούμαστε στον φάκελο όπου αποθηκεύτηκε το project.

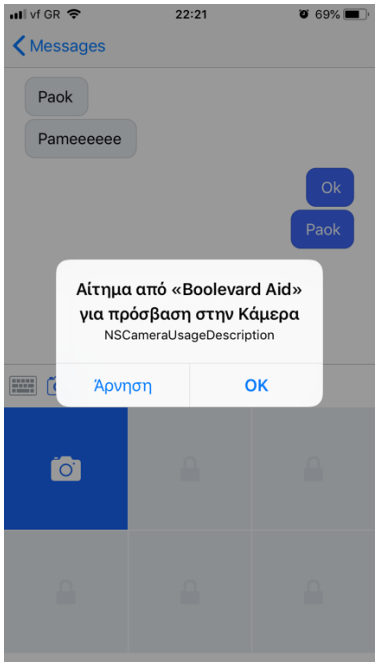


Εικόνα 1.10

Νέο Xcode project-εισαχθέντα frameworks

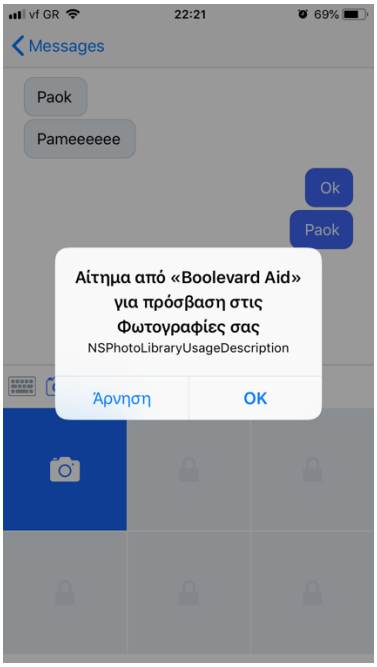
Με την εντολή $ pod init δημιουργούμε ένα podfile αρχείο και με την $ nano podfile εντολή επεξεργαζόμαστε το αρχείο. Εισάγουμε σε αυτό τις παρακάτω εντολές, $ pod 'Chatto', '= 3.2.0' και $ pod 'ChattoAdditions', '= 3.2.0' που βρίσκουμε στο github.com/badoo/Chatto. Πρόκειται για ένα framework το οποίο προορίζεται για την κατασκευή chat applications γραμμένο σε Swift το οποίο κατασκευάστηκε από την badooTech τεχνολογική κοινότητα, βασιζόμενοι στην εφαρμογή κοινωνικής δικτύωσης badoo. Εγκαθιστούμε το podfile στο project με την εντολή $ pod install και όταν ξανανοίγουμε το project μας βλέπουμε στην αριστερή στήλη να έχουν εισαχθεί τα pods. Τέλος θα πρέπει να μορφοποιήσουμε το Info.plist για τη χρήση και εικόνων στην εφαρμογή και να εισάγουμε τις ιδιότητες

Privacy – Camera Usage Description	NSCameraUsageDescription	Ενημέρωση χρήστη για πρόσβαση του app στην κάμερα της συσκευής
Privacy – Photo Library Usage Description	NSPhotoLibraryUsageDescription	Ενημέρωση χρήστη για πρόσβαση του app στις φωτογραφίες της συσκευής



Εικόνα 1.11

Αίτημα πρόσβασης στην κάμερα της συσκευής



Εικόνα 1.12

Αίτημα πρόσβασης στις εικόνες της συσκευής

3.2 Βασικό μέρος εφαρμογής

3.2.1 Δημιουργία πλαισίου κειμένου

Αρχικά θα εισάγουμε στο project ένα πλαίσιο κειμένου στο κάτω μέρος της οθόνης με την βοήθεια του framework Chatto το οποίο έχει δημιουργήσει από πριν από το ChatInputBar. Το ChatInpuBar είναι ένα text view για τον χρήστη ώστε να εισάγει κείμενο και ταυτόχρονα ένα κουμπί send ώστε να αποστέλλει το κείμενο. Για αρχή μετονομάζουμε την κλάση View Controller σε ChatLogController ο οποίος γίνεται ο αρχικός View Controller, ομοίως και το όνομα του αρχείου swift.

Εντολές	**Χρήση**
import Chatto import ChattoAdditions	Εισαγωγή Frameworks

Αλλάζουμε την superclass UIViewController σε BaseChatViewController που ανήκει στο Chatto για τις ανάγκες της εφαρμογής μας ώστε να αποκτήσουμε τις λειτουργίες που περιέχονται σε αυτήν. Η κλάση ChatLogController γίνεται subclass της BaseChatViewController.

Εντολές	**Χρήση**
override func createPresenterBuilders()-> [ChatItemType: [ChatItemPresenterBuilderProtocol]]{ return [ChatItemType: [ChatItemPresenterBuilderProtocol]]()}	Πρέπει να κάνουμε override αυτήν την function από την superclass αλλιώς η εφαρμογή κρασάρει. Είναι απαραίτητη για την λειτουργία της κλάσης έστω και αν επιστρέφουμε έναν κενό κατάλογο.
override func createChatInputView()->UIView { let inputbar = ChatInputBar.loadNib()}	Αυτη την function θα χρησιμοποιήσουμε για δημιουργήσουμε το ChatInputBar. Φορτώνει το αρχείο Nib το οποίο περιέχει το Input Bar, μια κάτω στήλη η οποία περιέχει έναν κειμενοφράφο και εχει φτιαχτεί απο το Chatto.

ώστε να δημιουργήσουμε το Chat Input Bar, το πλαίσιο δηλαδή εκείνο που θα χρησιμοποιείται κατά την συνομολία. Προς το παρόν το μόνο που βλέπουμε αν τρέξουμε το app είναι μια λευκή οθόνη με έναν κειμενογράφο και ένα πλαίσιο που γράφεις αλλα δεν κάνεις τίποτα άλλο.

Στην συνέχεια θα μορφοποιήσουμε την εμφάνιση του Chat Input Bar. Μέσα στην func createChatInputView θα εισάγουμε

Εντολές	**Χρήση**
var appearance = chatInputBarAppearance() appearance.sendButtonApperance.title = “Send”	Μορφοποιούμε το κουμπί send
appearance.textInputApperance.placeholderText = “Type a message”	Μορφοποιούμε το πλαίσιο κειμένου στο Input Bar ώστε στο λευκό πλαίσιο να εμφανίζει το Type a message

Θα πρέπει όμως για να εμφανιστούν αυτά τα στοιχεία στην οθόνη μας να συνδέσουμε την μεταβλητή appearance με το Input Bar και αυτό θα γίνει μέσω της δήλωσης ενός presenter

Εντολές	**Χρήση**
var presenter: BasicChatInputBarPresenter!	Δήλωση presenter μέσα στο ChatLogController

self.presenter = BasicChatInputPresenter(chatInputBar: inputbar, chatInputItems :[ChatInputItemProtocol](), chatInputBarAppearance: appearance)	Μέσα στην func createChatInputView καλούμε τον presenter property και μέσα στις παράμετρους σαν chatInputBar βάζουμε το inputbar που εχουμε δηλώσει πιο πριν, chatInputItems θα το δούμε παρακάτω, προς το παρών επιστρέφουμε κενή την μεταβλητή. Στην τελευταία παράμετρο εισάγουμε την μεταβλητή appearance την οποία είχαμε δηλώσει παραπάνω.

Έτσι ο presenter θα παρουσιάσει τις αλλαγές στην εμφάνιση του InputBar πριν ολοκληρωθεί η εκτέλεση της εντολής return inputbar.

3.2.2 Λειτουργία κουμπιού send

Για την λειτουργία ενός κουμπιού Send θα δημιουργήσουμε μία function με το όνομα handleSend με το TextChatInputItem να είναι υπεύθυνο για την διαχείριση του κουμπιού send.

Εντολές	**Χρήση**
func handleSend() -> TextChatInputItem { let item = TextChatInputItem() item.textInputHandler = { text in } return item }	Η function θα επιστρέφει ένα τύπο TextChatInputItem. Όποτε ο χρήστης πατάει send θα ενεργοποιείται η παράμετρος κειμένου στο item.textInputHandler, το οποίο είναι ότι έχει γράψει στο text view. Στο τέλος επιστρέφεται το item τύπου TextChatInputItem

Θα πρέπει να συνδεθεί όμως το TextChatInputItem με το InputBar και αυτό θα γίνει όπως και πριν μέσω του presenter. Στις παραμέτρους του BasicChatInpurPresenter θα προσθέσουμε το TextChatInputItem και έτσι θα γίνει self.presenter = BasicChatInputPresenter(chatInputBar: inputbar, chatInputItems :[handleSend()], chatInputBarAppearance: appearance). Συνεπώς κάθε φορά που θα πατιέται το send θα εκτελείται το handleSend.

3.3 Δημιουργία μηνύματος κειμένου

Ο μόνος τρόπος για την δημιουργία ενός μηνύματος κειμένου είναι μέσω ενός Model το οποίο κληρονομεί τις ιδιότητες από την superclass TextMessageModel του Chatto. Οπότε δημιουργούμε έναν φάκελο με το όνομα Text Message και ένα swift αρχείο μέσα σ'αυτό με το όνομα TextModel.

Εντολές	Χρήση
import Chatto import ChattoAdditions	Εισαγωγή frameworks

class TextModel:
TextMessageModel<MessageModel> {
}
	Δημιουργούμε μία subclass TextModel η οποία ανήκει στην superclass TextMessageModel με ένα γενικού τύπου MessageModel protocol που απαιτείται

override init(messageModel: MessageModel, text: String){ }	Μέσα στην κλάση TextModel αρχικοποιούμε τα μηνύματα κειμένου μέσω του override init που κληρονομεί και αρχικοποιεί τα μηνύματα από την προκατασκευασμένη TextMessageModel
super.init(messageModel: messageModel, text: text)	Καλούμε το super.init για να καλέσουμε την αρχικοποίηση καθώς έτσι είναι δομημένη η TextMessageModel για να αποφύγουμε errors

Έπειτα θα πρέπει να δηλώσουμε στην function handleSend τα μηνύματα κειμένου

Εντολές	Χρήση
let message = MessageModel(uid:"", senderId:"",type:"", isIncoming:false, date:Date(), status:nil)	Δήλωση μηνύματος κειμένου
let textMessage = TextModel(messageModel :message, text: text)	Δημιουργία μηνύματος μέσω του TextModel, αρχικοποιούμε ένα textMessage ίσο με ένα TextModel με παράμετρους το message που δηλώσαμε απο πάνω και παράμετρο text.

3.3.1 Αποστολή μηνύματος και ρύθμιση δεδομένων

Δημιουργούμε ένα νέο αρχείο swift με το όνομα DataSource και εισάγουμε όπως και πριν τα frameworks Chatto.

Εντολές	Χρήση
class DataSource: ChatDataSourceProtocol	Δήλωση κλάσης DataSource που κληρονομεί methods και properties από το ChatDataSourceProtocol πρωτόκολλο

var chatItems: [ChatItemProtocol] {
 	
}
	Όπως μας επιτάσσει η κλάση καλούμε για αρχή το chatItem και σαν τιμή θα επιστρέφεται ένας πίνακας ChatItemProtocol, δηλαδή ένας πίνακας με τα μηνήματα που πρόκειται να εμφανιστούν

Γ’αυτόν τον πίνακα θα δημιουργήσουμε μία κλάση δική του. Φτιάχνουμε ένα swift αρχείο με όνομα ChatItemsController και εισάγουμε και τα frameworks.

Εντολές	**Χρήση**
class ChatItemsController {}	Δήλωση κλάσης ChatItemsController
var items = [ChatItemProtocol]()	Μέσα στην κλάση δήλωση μεταβλητής items που είναι ένας πίνακας ChatItemProtocol

Στην κλάση DataSource εισάγουμε

Εντολές	**Χρήση**
var controller = ChatItemController()	Εισαγωγή κλάσης ChatItemsController στο DataSource
var initialMessages = [ChatItemProtocol]()	Μέσα στην κλάση δήλωση μεταβλητής initialMessages που είναι ένας πίνακας ChatItemProtocol
return controller.items	Επιστρέφεται όπως αναλύθηκε παραπάνω η τιμή του items που είναι ένας πίνακας ChatItemProtocol

var hasMoreNext: Bool{
return false}
var hasMorePrevious: Bool{
return false}
	Και τα δύο έχουν να κάνουν με την σελιδοποίηση (pagination) που θα αναλυθεί παρακάτω και είναι απαιτητά από το ChatDataSourceProtocol

func loadNext() { } func loadPrevious() { }	Και τα δύο έχουν να κάνουν με την σελιδοποίηση (pagination) και είναι απαιτητά από το ChatDataSourceProtocol
func adjustNumberOfMessages(preferedMaxCount: Int?,focusPosition:Double, completion{(Bool)} -> Void) { completion(false)}	Πολύ σημαντική function η οποία εχει να κάνει με την απόδοση και είναι απαιτητή και αυτή από το ChatDataSourceProtocol
var delegate: ChatDataSourceDelegateProtocol?	Καλούμε την delegate η οποία κάθε φορά που εκτελείται ενημερώνει το View collection (layer).

Κάθε φορά που θα πατάμε Send δημιουργήται ένα TextMessage το οποίο χρειάζεται το chatItems επειδή όμως είναι computed property και όχι stored property δεν μπορεί να το διαχειριστεί, πρέπει να το περνάμε και να το προσθέτουμε μέσω του πίνακα items και από την στιγμή που το items είναι η επιστρεφόμενη τιμή του chatItems το chatItems θα πολλαπλασιάζεται. Στο DataSource εισάγουμε

Εντολές	Χρήση
func addMessage(message: ChatItemProtocol){}	Εισαγωγή function

ώστε να το καλέσουμε στο handleSend στο ChatLogController

Εντολές	Χρήση
var dataSource = DataSource()	Εισαγωγή κλάσης DataSource στο ChatLogController
self?.dataSource.addMessage(message: textMessage)	Καλούμε την function μέσα στο handleSend
func handleSend() -> TextChatInputItem { let item = TextChatInputItem() item.textInputHandler = { [weak self] text in } return item }	Προσθέτουμε το weak self στην κλάση ώστε να μην έχουμε πρόβλημα με την μνήμη και να μπορέσει στο τέλος να κάνει αποδέσμευση μνήμης αλλιώς κρασάρει

Στην ChatItemsController κλάση εισάγουμε

Εντολές	Χρήση
func insertItem (message: ChatItemProtocol){ self.items.append(message)}	Εισαγωγή function

και στην DataSource στην function addMessage καλούμε αυτήν την function

Εντολές	Χρήση
self.controller.insertItem(message: message)	Καλούμε την function

Πρόκειται για μία στάνταρ διαδικασία και για να γίνει κατανοητό με την λογική της MVC αρχιτεκτονικής κάθε φορά που πατάμε το κουμπί Send δημιουργήται ένα textMessage μέσω του Model που δημιουργήσαμε, στην handleSend function του ChatLogController→καλεί την function addMessage στην DataSource →η οποία με την σειρά της ενεργοποιεί την function insertItem στο ChatItemsController προσθέτοντας το μήνυμα στον πίνακα items→το οποίο στην τελική αυξάνει το chatItems στο DataSource. Τέλος συνδέουμε το ChatLogController με το DataSource

Εντολές	Χρήση

self.chatDataSource = self.dataSource	Σύνδεση ChatLogController με DataSource

3.3.2 Μορφοποίηση μηνυμάτων

Δημιουργούμε ένα αρχείο swift με όνομα Decorator και εισάγουμε και τα frameworks που δουλεύουμε για να προσδώσουμε στα μηνύματα τα οπτικά χαρακτηριστικά που επιθυμούμε.

Εντολές	**Χρήση**
class Decorator: ChatItemsDecoratorProtocol { }	Δήλωση κλάσης Decorator
func decorateItems(_chatItems: [ChatItemProtocol]) -> [DecoratedChatItem]{ }	Είναι η κλάση σύμφωνα με το ChatItemsDecoratorProtocol την οποία πρέπει να καλέσουμε

Ακολούθως μέσα στο ChatLogController θα συνδέσουμε το ChatLogController με το Decorator

Εντολές	**Χρήση**
var decorator = Decorator()	Εισαγωγή κλάσης Decorator στο ChatLogController
self.chatItemsDecorator = self.decorator	Είναι η κλάση σύμφωνα με το ChatItemsDecoratorProtocol την οποία πρέπει να καλέσουμε

Η μορφοποίηση του μηνύματος θα πρέπει να γίνεται κάθε φορά που ο πίνακας chatItems αυξάνεται, οπότε στο addMessage εισάγουμε

Εντολές	**Χρήση**
self.delegate?.chatDataSourceDidUpdate(self)	Το delegate ειδοποιεί το Decorator οτι αυξήθηκε ο πίνακας chatItems και ενεργοποιείται το decoratedItems

Πρέπει να μετατρέψουμε όλα τα textMessage στον πίνακα chatItems σε decorated textMessage πριν εμφανιστούν στο View.

Εντολές	**Χρήση**
var decoratedItems = [DecoratedChatItem]()	Δηλώνουμε έναν πίνακα decoratedItems τύπου DecoratedChatItem μέσα στην decoratedItems function

for item in chatItems { let decoratedItem = DecoratedChatItem(chatItem: item, decorationAttributes: ChatItemDecorationAttributes(bottomMargin: 10,showTail: false,canShowAvatar: false)) decoratedItems.append(decoratedItem)}	Θα προσπελάσουμε κάθε item στον πίνακα chatItems και κάθε μετατρεπούμενο item θα προσθέτεται στον πίνακα decoratedItems
return decoratedItems	Επιστρέφει τον πίνακα decoratedItems με τα μετατρέψιμα item

3.3.3 *Μετατροπή μηνυμάτων κειμένου σε ViewModel*

Για να εμφανιστεί στο Collection View το κάθε μήνυμα θα πρέπει να μετατραπεί σε View Model. Έτσι δημιουργούμε ένα αρχείο swift με το όνομα TextBuilder και εισάγουμε σ'αυτό τα γνωστά frameworks.

Εντολές	**Χρήση**
class TetxBuilder: ViewModelBuilderProtocol { }	Δήλωση κλάσης η οποία θα κάνει μετατροπή κάθε decorated μηνύματος σε View Model, subclass του ViewModelBuilderProtocol

class ViewModel:
TextMessageViewModel<TextModel> {
}
	Δήλωση κλάσης της οποίας η subclass δέχεται υποχρεωτικά σαν παράμετρο έναν γενικού τύπου text message model οπότε εμείς βάζουμε το δικό μας TextModel - Θα δέχεται τα μετατρέψιμα μηνύματα από την TextBuilder και θα τα εμφανίζει

override init(textMessage: TextModel, messageViewModel:MessageViewModelProtocol){}	Όπως επιτάσει η superclass TextMessageViewModel πρέπει να κάνουμε override το initializer για να αποκτήσουμε πρόσβαση σε αυτό
super.init(textMessage: TextMessage, messageViewModel: messageViewModel) {}	Μέσα στο override καλούμε την αρχικοποίηση της superclass
let defaultBuilder = MessageViewModelDefaultBuilder()	Την κάνουμε τοπική μεταβλητή γιατί προκαλεί errors στην function

func canCreateViewModel(fromModel
decoratedTextMessage: Any) -> Bool {
return decoratedTextMessage is TextModel
}
	Μέσα στην κλάση TetxBuilder καλούμε την function canCreateViewModel η οποία ελέγχει αν το Model του decorated μηνύματος είναι τύπου TextModel και αν είναι επιστρέφει true και συνεχίζει στην από κάτω function

func createViewModel(_decoratedTextMessage: TextModel) -> ViewModel{}	Function για την μετατροπή του decorated μηνύματος από TextModel σε ViewModel
let textmessageViewModel = ViewModel(textMessage: decoratedTextMessage,	Μετατρέπει τα decorated μηνύματα από TextModel σε ViewModel

messageViewModel: defaultBuilder. createMessageViewModel(message: decoratedTextMessage))	
return textmessageViewModel	Επιστρέφουμε το ViewModel και εμφανίζεται στο Collection View

Ακολούθως θα πρέπει όπως και στις προηγούμενες περιπτώσεις να συνδέσουμε το TextBuilder με τον Controller, έτσι μέσα στο ChatLogController στην function createPresenterBuilders

Εντολές	**Χρήση**
let textMessageBuilder = TextMessageBuilder(viewModelBuilder :TextBuilder(),interactionHandler: TextHandler())	Σύνδεση TextBuilder με ChatLogController η οποία μας υποχρεώνει στην δημιουργία ενός TextHandler μαζί με τον TextBuilder

Δημιουργούμε ένα αρχείο swift με όνομα TextHandler και εισάγουμε και τα frameworks Chatto.

Εντολές	**Χρήση**

class TextHandler:
BaseMessageInteractionHandlerProtocol {
}
	Δημιουργία κλάσης TextHandler και θα κληρονομεί ιδιότητες από το προτόκολλο BaseMessageInteractionHandlerProtocol το οποίο αν περιηγηθούμε σε αυτό μας υποχρεώνει να καλέσουμε συγκεκριμένα functions όλα τύπου ViewModel

func userDidTapOnFailIcon(viewModel: ViewModel,failIconView: UIView){}	Ειδοποιεί εάν ο χρήστης πάτησε πάνω στην εικόνα προειδοποίησης αποτυχίας αποστολής
func userDidTapOnAvatar(viewModel: ViewModel){}	Ειδοποιεί εάν ο χρήστης πάτησε πάνω στην εικόνα χρήστη
func userDidTapOnBubble(viewModel: ViewModel){}	Ειδοποιεί εάν ο χρήστης πάτησε πάνω στην κείμενο
func userDidBeginLongPressBubble(viewModel: ViewModel){}	Ειδοποιεί εάν ο χρήστης πάτησε συνεχιζόμενα πάνω στο κείμενο
func userDidEndLongPressBubble(viewModel: ViewModel){}	Ειδοποιεί εάν ο χρήστης σταμάτησε το συνεχιζόμενο πάτημα στο κείμενο

Τέλος για την σύνδεση του ChatLogController με το TextBuilder χρειάζεται η επιστροφή ενός ChatItemType όπως επιτάσσει η createPresenterBuilders function. Για τον λόγο αυτό δηλώνουμε μια μεταβλητή chatItemType μέσα στο TextModel

Εντολές	Χρήση
static let chatItemType = “text”	Δημιουργία μιας μεταβλητής chatItemType, μετα από παρότρυνση του compiler σε static let

Επίσης κατά την δημιουργία του μηνύματος στο handleSend αλλάζουμε το type του message σε TextModel.chatItemType. Στο ChatLogController στην function createPresenterBuilders

Εντολές	Χρήση
return [TextModel.chatItemType: [textMessageBuilder]]	Σύνδεση ChatLogController με TextBuilder καθώς μόνο τα μηνύματα τύπου TextModel.chatItemType μπορούν να έχουν πρόσβαση στο TextBuilder, αφού πρώτα μορφοποιηθούν στο Decorator

Τέλος για να μπορέσει η εφαρμογή να χρησιμοποιειθεί χωρίς να κρασάρει θα πρέπει κατά την δημιουργία του μηνύματος στο handleSend, κάθε μήνυμα να πέρνει ένα μοναδικό ID. Για να διορθωθεί αυτό θα πρέπει μέσα στο handleSend

Εντολές	Χρήση
let date = Date()	Η μεταβλητή date παίρνει την ημερομηνία και ώρα
let double = Double(date.timeIntervalSinceReferenceDate)	Μετατροπή της date μεταβλητής σε double χρησιμοποιώντας το χρονικό διάστημα από την ημερομηνία αναφοράς
let senderID = “me”	Δήλωση στατικής μεταβλητής

Τώρα το double σε συνδυασμό με το senderID δίνει στο κάθε μήνυμα ένα μοναδικό uid. Έτσι το property message γίνεται

```
let message = MessageModel(uid: “(\(double,senderID))”,senderId: senderID,type: TextModel.chatItemType,isIncoming: false,date: date,status: .success)
```

3.3.4 Διαδρομή μηνύματος

Κάθε μήνυμα που στέλνουμε αρχικοποιείται με το TextModel και καλεί την function addMessage -> εκείνη με την σειρά της την insertItem και αυτή προσθέτει στο τέλος του πίνακα items το μήνυμα -> το οποίο το προωθεί στον πίνακα chatItems ->έπειτα πάλι στην function addMessage καλείται το delegate -> το οποίο ειδοποιεί το Decorator->και εκείνο παίρνει το

μήνυμα από τον chatItems πίνακα και μορφοποιεί το μήνυμα-> προσθέτει το μήνυμα στο τέλος του πίνακα decoratedItems->τελικός προορισμός του μηνύματος το TextBuilder ώστε να μετατραπεί σε ViewModel εκτελώντας τις δύο functions, αν είναι τύπου TextModel μετατρέπεται σε ViewModel και εμφανίζεται στο Collection View.

3.4 Προσθήκη δυνατότητας αποστολής εικόνων

Με την ίδια λογική όπως με τα μηνύματα κειμένου έτσι και με την ανταλλαγή φωτογραφιών πρέπει να δημιουργήσουμε μια function αντίστοιχη με την handleSend η οποία κάθε φορά που ο χρήστης πατάει σε μία φωτογραφία να δημιουργεί μήνυμα εικόνας.

Εντολές	**Χρήση**
func handlePhoto() -> PhotosChatInputItem{ }	Δημιουργία κλάσης η οποία επιστρέφει ένα αντικεκείμενο τύπου PhotosChatInputItem
let item = PhotosChatInputItem(presentingController: self)	Αρχικοποιούμε την μεταβλητή item ωστε να είναι τύπου PhotosChatInputItem με View Controller αυτόν πάνω στον οποίο δουλεύουμε
item.photoInputHandler = { photo in }	Επίσης η μεταβλητή item δέχεται σαν είσοδο μόνο παράμετρο τύπου photo
return item	Επιστροφή item τύπου PhotosChatInputItem

Τέλος εισάγουμε στον presenter την function handlePhoto ώστε να εμφανίζεται στο inputBar

```
self.presenter = BasicChatInputPresenter(chatInputBar: inputbar, chatInputItems
:[handleSend(),handlePhoto()], chatInputBarAppearance: appearance)
```

3.4.1 Δημιουργία μηνύματος εικόνας

Ακριβώς όπως και στα μημύματα κειμένου για να αρχικοποιήσουμε ένα μήνυμα κειμένου χρησιμοποιούμε το TextModel έτσι και στα μηνύματα εικόνας χρειαζόμαστε ένα PhotoModel. Δημιουργούμε ένα καινούργιο αρχείο swift με όνομα PhotoModel, εισάγωντας τα frameworks και δημιουργώντας μία κλάση PhotoModel.

Εντολές	**Χρήση**

class PhotoModel:
PhotoMessageModel<MessageModel> {
}
	Δημιουργούμε μία subclass PhotoModel η οποία ανήκει στην superclass PhotoMessageModel με ένα γενικού τύπου MessageModel protocol που απαιτείται

override init(messageModel: MessageModel, imageSize: CGSize,image: UIImage){ }	Μέσα στην κλάση PhotoModel αρχικοποιούμε τα μηνύματα εικόνας μέσω του override init που κληρονομεί και αρχικοποιεί τα μηνύματα από την προκατασκευασμένη PhotoMessageModel
super.init(messageModel: messageModel, imageSize: imageSize, image: image)	Καλούμε το super.init για να καλέσουμε την αρχικοποίηση καθώς έτσι είναι δομημένη η PhotoMessageModel

Η function handlePhoto γίνεται

Εντολές	**Χρήση**
let date = Date()	Η μεταβλητή date παίρνει την ημερομηνία και ώρα
let double = Double(date.timeIntervalSinceReferenceDate)	Μετατροπή της date μεταβλητής σε double χρησιμοποιώντας το χρονικό διάστημα από την ημερομηνία αναφοράς
let senderID = “me”	Δήλωση στατικής μεταβλητής
let message = MessageModel(uid: “(\(double,senderID))”,senderId: senderID,type: “”,isIncoming: false,date: date,status: .success).	Δήλωση μηνύματος εικόνας
let photoMessage = PhotoModel(messageModel :message, imageSize: photo.size,image: photo)	Δημιουργία μηνύματος μέσω του PhotoModel, αρχικοποιούμε ένα photoMessage ίσο με ένα PhotoModel με παράμετρους το message που δηλώσαμε απο πάνω το μέγεθος της εικόνας και την ίδια την εικόνα.
self?.dataSource.addMessage(message: photoMessage)	Καλούμε την function μέσα στο handlePhoto
item.photoInputHandler = { [weak self] photo in }	Προσθέτουμε το weak self στην κλάση ώστε να μην έχουμε πρόβλημα με την μνήμη και να μπορέσει στο τέλος να κάνει αποδέσμευση μνήμης αλλιώς κρασάρει

3.4.2 Μετατροπή μηνυμάτων εικόνας σε ViewModel

Για να λειτουργήσει θα πρέπει να φτιάξουμε ένα builder ακριβώς όπως κάναμε και για τα μηνύματα κειμένου έτσι και για τα μηνύματα εικόνας θα πρέπει να μετατραπούν από decorated σε ViewModel. Δημιουργούμε ένα swift αρχείο με όνομα PhotoBuilder και εισάγουμε τα frameworks.

Εντολές	**Χρήση**

class PhotoBuilder: ViewModelBuilderProtocol { }	Δήλωση κλάσης η οποία θα κάνει μετατροπή κάθε decorated μηνύματος σε View Model, subclass του ViewModelBuilderProtocol
class photoViewModel: PhotoMessageViewModel<PhotoModel> { }	Δήλωση κλάσης της οποίας η subclass δέχεται υποχρεωτικά σαν παράμετρο έναν γενικού τύπου photo message model οπότε εμείς βάζουμε το δικό μας PhotoModel-Θα δέχεται τα μετατρέψιμα μηνύματα από την PhotoBuilder και θα τα εμφανίζει
override init(photoMessage: PhotoModel, messageViewModel:MessageViewModelProtocol) { }	Όπως επιτάσει η superclass PhotoMessageViewModel πρέπει να κάνουμε override το initializer για να αποκτήσουμε πρόσβαση σε αυτό
super.init(photoMessage: photoMessage, messageViewModel: messageViewModel) {}	Μέσα στο override καλούμε την αρχικοποίηση της superclass
let defaultBuilder = MessageViewModelDefaultBuilder()	Την κάνουμε τοπική μεταβλητή γιατί προκαλεί errors στην function

func canCreateViewModel(fromModel decoratedPhotoMessage: Any) -> Bool {
return decoratedPhotoMessage is PhotoModel
}
	Μέσα στην κλάση PhotoBuilder καλούμε την function canCreateViewModel η οποία ελέγχει αν το Model του decorated μηνύματος είναι τύπου PhotoModel και αν είναι επιστρέφει true και συνεχίζει στην από κάτω function

func createViewModel(_decoratedPhotoMessage: PhotoModel) -> photoViewModel{}	Function για την μετατροπή του decorated μηνύματος από PhotoModel σε ViewModel
let photoMessageViewModel = photoViewModel(photoMessage: decoratedPhotoMessage, messageViewModel: defaultBuilder. createMessageViewModel(message: decoratedPhotoMessage))	Μετατρέπει τα decorated μηνύματα εικόνας από PhotoModel σε ViewModel
return photoMessageViewModel	Επιστρέφουμε το ViewModel και εμφανίζεται στο Collection View

Όπως ακριβώς και με το TextBuilder έτσι και με το PhotoBuilder κατά την σύνδεσή του με το ChatLogController μας υποχρεώνει να δημιουργήσουμε ένα PhotoHandler. Έτσι δημιουργούμε το swift αρχείο με όνομα PhotoHandler και εισάγουμε τα frameworks.

Εντολές	**Χρήση**
class PhotoHandler: BaseMessageInteractionHandlerProtocol { }	Δημιουργία κλάσης PhotoHandler και θα κληρονομεί ιδιότητες από το προτόκολλο BaseMessageInteractionHandlerProtocol το οποίο αν περιηγηθούμε σε αυτό μας υποχρεώνει

	να καλέσουμε συγκεκριμένα functions όλα τύπου ViewModel
func userDidTapOnFailIcon(viewModel: photoViewModel,failIconView: UIView){ }	Ειδοποιεί εάν ο χρήστης πάτησε πάνω στην εικόνα προειδοποίησης αποτυχίας αποστολής
func userDidTapOnAvatar(viewModel: photoViewModel){}	Ειδοποιεί εάν ο χρήστης πάτησε πάνω στην εικόνα χρήστη
func userDidTapOnBubble(viewModel: photoViewModel){}	Ειδοποιεί εάν ο χρήστης πάτησε πάνω στην κείμενο
func userDidBeginLongPressBubble(viewModel: photoViewModel){}	Ειδοποιεί εάν ο χρήστης πάτησε συνεχιζόμενα πάνω στην εικόνα
func userDidEndLongPressBubble(viewModel: photoViewModel){}	Ειδοποιεί εάν ο χρήστης σταμάτησε το συνεχιζόμενο πάτημα στην εικόνα

Στο PhotoModel εισάγουμε

Εντολές	**Χρήση**
static let chatItemType = “photo”	Δημιουργία μιας μεταβλητής chatItemType, μετα από παρότρυνση του compiler σε static let

Σύνδεση PhotoBuilder με ChatLogController, εισάγουμε στην function createPresenterBuilders

Εντολές	**Χρήση**
let photoPresenterBuilder = PhotoMessageBuilder(viewModelBuilder :PhotoBuilder(),interactionHandler: PhotoHandler())	Σύνδεση PhotoBuilder με ChatLogController
return [PhotoModel.chatItemType: [photoPresenterBuilder]]	Σύνδεση ChatLogController με PhotoBuilder καθώς μόνο τα μηνύματα τύπου PhotoModel.chatItemType μπορούν να έχουν πρόσβαση στο PhotoBuilder, αφού πρώτα μορφοποιηθούν στο Decorator

Τέλος στην function handlePhoto εισάγουμε τον τύπο του μηνύματος και έτσι διαμορφώνεται

let message = MessageModel(uid: “(\(double,senderID))”,senderId: senderID,type: PhotoModel.chatItemType,isIncoming: false,date: date,status: .success)

3.5 Δημιουργία User Interfaces

Δημιουργούμε δύο αρχεία swift με ονόματα SignInViewController και SignUpViewController τα οποία δημιουργούν αυτομάτως μέσα στα αρχεία τους μια κλάση με το όνομα αρχείου τους έχοντας σαν superclass την UiViewController.

3.5.1 Sign in UI

Στο Main.storyboard αρχείο εισάγουμε έναν View Controller με storyboardID:SIGNIN. Έπειτα εισάγουμε-> ένα ImageView μέσα στον View Controller και προσθέτουμε από την αριστερή στήλη στο images την εικόνα με το λογότυπο μας. Στην συνέχεια προσθέτουμε ->δύο TextFields και τα μετονομάζουμε σε Email και Password με secure entry ιδιότητα. Τέλος εισάγουμε->δύο Buttons με όνομα SignIn και SignUp με ένα Label δίπλα στο SignUp οτι πρόκειται για χρήστες οι οποίοι δεν έχουν λογαριασμό.

Στην συνέχεια συνδέουμε τον View Controller στο αρχείο swift που φτιάξαμε δίνοντας συγκεκριμένα την κλάση που δημιουργήθηκε με το αρχείο, στην προκειμένη SignInViewController στο πάνω μέρος δεξιά του Xcode. Τέλος συνδέουμε ένα ένα τα αντικείμενα που συνοδέουν τον View Controller στο αρχείο μας με drag&drop και ονοματίζοντάς τα όπως τα ονοματήσαμε κατά την δημιουργία τους προσδιδοντας στα Buttons την ιδιότητα Action. Εισάγουμε στην function SignUp

Εντολές	**Χρήση**
let controller = storyboard?.instantiateViewController(withIdentifier: “SIGNUP”) as! SignUpViewController	Αρχικοποιούμε την μεταβλητή controller ίση με τον View Controller SIGNUP
self.navigationController?.show(table,sender:nil)	Με το πάτημα του κουμπιού SignUp παρουσιάζεται η View Controller SIGNUP

3.5.2 Sign up UI

Με την ίδια ακριβώς διαδικασία που φτιάξαμε τον View Controller SIGNIN θα δημιουργήσουμε και τον View Controller με storyboardID:SIGNUP. Η μόνη διαφορά είναι οτι θα φτιάξουμε ένα ακόμα TextField με όνομα FullName και δύο Button ένα με όνομα SignUp και ένα με όνομα Back. Συνδέουμε τον View Controller με τον SignUpViewController όπως ακριβώς και τον View Controller SIGNIN μαζί με τα αντικείμενά του. Στην function Back

Εντολές	Χρήση
self.dismiss(animated: true,completion: nil)	Με το πάτημα του κουμπιού Back απελευθερώνουμε απλά τον View Controller SIGNUP και αυτό μας επιστρέφει στην αρχική οθόνη

3.5.3 Message UI

Δημιουργούμε ένα swift αρχείο με όνομα MessagesTableViewController και ένα swift αρχείο με όνομα MessagesTableViewCell. Στην πρώτη περίπτωση δημιουργείται μία κλάση MessagesTableViewController με superclass UIViewController η οποία κληρονομεί στοιχεία από FUICollectionDelegate, UITableViewDelegate και UITableViewDataSource. Στην δεύτερη περίπτωση δημιουργείται μια κλάση MessagesTableViewCell με superclass UITableViewCell.

Εισάγουμε ένα Navigation Controller στο Main.storyboard και ορίζουμε αυτόν ως τον αρχικό View Controller και κάνουμε τον View Controller SIGNIN root-ViewController. Στην συνέχεια εισάγουμε ένα νέο View Controller με storyboardID:table και μέσα σε αυτόν τοποθετούμε ένα TableView. Έπειτα προσαρμόζουμε τον TableView ώστε να καταλάβει όλον τον View Controller και εισάγουμε ->ένα NavigationItem με τίτλο Messages, -> δύο Buttons με όνομα SignOut και μεταβάλουμε τις ιδιότητες του δεύτερου ώστε να γίνει SystemItemType ίσο με Add και να αποκτήσει το σύμβολο + με την προοπτική της πρόσθεσης νέων χρηστών. Τέλος στο TableView εισάγουμε-> ένα TableViewCell και -> τρία Labels ώστε να αντιπροσωπεύει το πρώτο το όνομα του συνομιλητή μας, το δεύτερο την προεσκόπιση του τελευταίου μηνύματος και το τρίτο την ημερομηνία του τελευταίου σταλθέντος μηνύματος. Κλείνωντας συνδέουμε όπως και παραπάνω με τους View Controllers, τον View Controller και τα αντικείμενά του με το MessagesTableViewController και το TableViewCell και τα αντικείμενα του με το MessagesTableViewCell.

Στην function SignUp του SignUpViewController

Εντολές	Χρήση
let table = self.storyboard.instantiateViewController(with identifier: “table”) as!	Αν είναι επιτιχυμένο το SignUp παρουσιάζεται η View Controller table

MessagesTableViewController self.navigationController?.show(table,sender:nil)	

Το ίδιο και αν είναι επιτυχυμένο το SignIn, στην SignIn function

Εντολές	**Χρήση**
let table = self.storyboard.instantiateViewController(with identifier: “table”) as! MessagesTableViewController self.navigationController?.show(table,sender:nil)	Αν είναι επιτιχυμένο το SignIN παρουσιάζεται η View Controller table

3.5.4 Μορφοποίηση των UI

Για να καλύτερη εμπειρία του χρήστη με την εφαρμογή θα πρέπει τα UI να μορφοποιηθούν κατάλληλα. Για αρχή θα πρέπει να διεθετύσουμε το ζήτημα με το πληκτρολόγιο που κρύβει τα TextFields στο οποίο θέλουμε να εισάγουμε κείμενο. Δημιουργούμε ένα αρχείο Swift με όνομα Helpers και εισάγουμε

Εντολές	**Χρήση**
import UIKIt	Εισαγωγή framework για το UI
extension UIViewController{ }	Ορίζουμε μία επέκταση του UIViewController
func showingKeyboard(notification: notification){ }	Μια function η οποία θα ενημερώνει τον τύπο notification

if let keyboardHeight = (notification.userInfo[
UIKeyboardFrameBeginInUserInfoKey] as?
NSValue).cgRectValue.height{
}
	Με το userInfo παίρνουμε τις πληροφορίες απο το Notification, μας δίνει το πλαίσιο του πληκτρολογίου σαν NSValue το οποίο το κάνουμε μετατροπή σε cgRectValue

self.view.frame.origin.y = -keyBoardHeight	Τέλος λέμε οτι κάτω μέρος του UI θα ξεκινάει στο πάνω μέρος του πληκτρολογίου
func hidingKeyboard(){ }	Μια function η οποία θα ενημερώνει τον τύπο notification
self.view.frame.origin.y = 0	Μεταφέρει το πληκτρολόγιο στο κάτω μέρος της οθόνης και ουσιαστικά κλείνει το πληκτρολόγιο

Στην SignInViewController εισάγουμε

Εντολές	Χρήση
NotificationCenter.default.addObserver(observer: self,selector: (showingKeybord), name: NSNotification.Name(rawValue: “UIKeybordWillShowNotification”),object: nil)	Κάθε φορά που εμφανίζεται το πληκτρολόγιο ενεργοποιείται το UIKeybordWillShow, ακολούθως ενεργοποιείται το Notification και εκείνο καλεί την function showingKeyboard
NotificationCenter.default.addObserver(observer: self,selector: (hidingKeyboard), name: NSNotification.Name(rawValue: “UIKeybordWillHideNotification”),object: nil)	Κάθε φορά που απομακρύνεται το πληκτρολόγιο ενεργοποιείται το UIKeybordHideShow, ακολούθως ενεργοποιείται το Notification και εκείνο καλεί την function hidingKeyboard
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?)){ self.view.endEditing(true)}	Κάθε φορά που θα πατάμε σε ένα μέρος της οθόνης εκτός του πλαισίου του πληκτρολογίου το πληκτρολόγιο θα απομακρύνεται

Το ίδιο πρέπει να κάνουμε και με την SignUpViewController εφόσον έχει και εκείνη TextFields

Εντολές	Χρήση
NotificationCenter.default.addObserver(observer: self,selector: (showingKeybord), name: NSNotification.Name(rawValue: “UIKeybordWillShowNotification”),object: nil)	Κάθε φορά που εμφανίζεται το πληκτρολόγιο ενεργοποιείται το UIKeybordWillShow, ακολούθως ενεργοποιείται το Notification και εκείνο καλεί την function showingKeyboard
NotificationCenter.default.addObserver(observer: self,selector: (hidingKeyboard), name: NSNotification.Name(rawValue: “UIKeybordWillHideNotification”),object: nil)	Κάθε φορά που απομακρύνεται το πληκτρολόγιο ενεργοποιείται το UIKeybordHideShow, ακολούθως ενεργοποιείται το Notification και εκείνο καλεί την function hidingKeyboard
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?)){ self.view.endEditing(true)}	Κάθε φορά που θα πατάμε σε ένα μέρος της οθόνης εκτός του πλαισίου του πληκτρολογίου το πληκτρολόγιο θα απομακρύνεται

3.6 Firebase

Το Firebase είναι μία πλατφόρμα ανάπτυξης εφαρμογών απο την Google που μας βοηθάει να βελτιώσουμε και να αναπτύξουμε την εφαρμογή μας, συνδυάζοντας πιστοποίηση χρήστη, βάση δεδομένων σε πραγματικό χρόνο και αποθήκευση δεδομένων.

3.6.1 Εισαγωγή Firebase στο project

Στο https://console.firebase.google.com/u/0/ θα δημιουργήσουμε έναν λογαριασμό ώστε να συνδεθούμε στο Firebase και ακολούθως θα δώσουμε το όνομα του project BoolevardAid και την χώρα προορισμού του project μας. Κατά την δημιουργία του project καλούμαστε να

εισάγουμε το iOS bundle ID μας που είναι com.AlexandrosAristeridis.BoolevardAid και ακολούθως μας προτρέπει να κατεβάσουμε το GoogleService-Info.plist και να το εισάγουμε στο project μας στο Xcode με drag&drop. Τέλος πλοηγούμαστε στον φάκελο του project μας και εγκαθηστούμε τα frameworks στο υπάρχων podfile , $ pod 'Firebase/Core', $ pod 'Firebase/Auth', $ pod 'Firebase/Database', $ pod 'FirebaseUI/Database', $ pod 'Firebase/Storage', $ pod 'Kingfisher' και $ pod 'SwiftyJSON' κάνοντας $ pod install. Στο AppDelegate αρχείο μας εισάγουμε

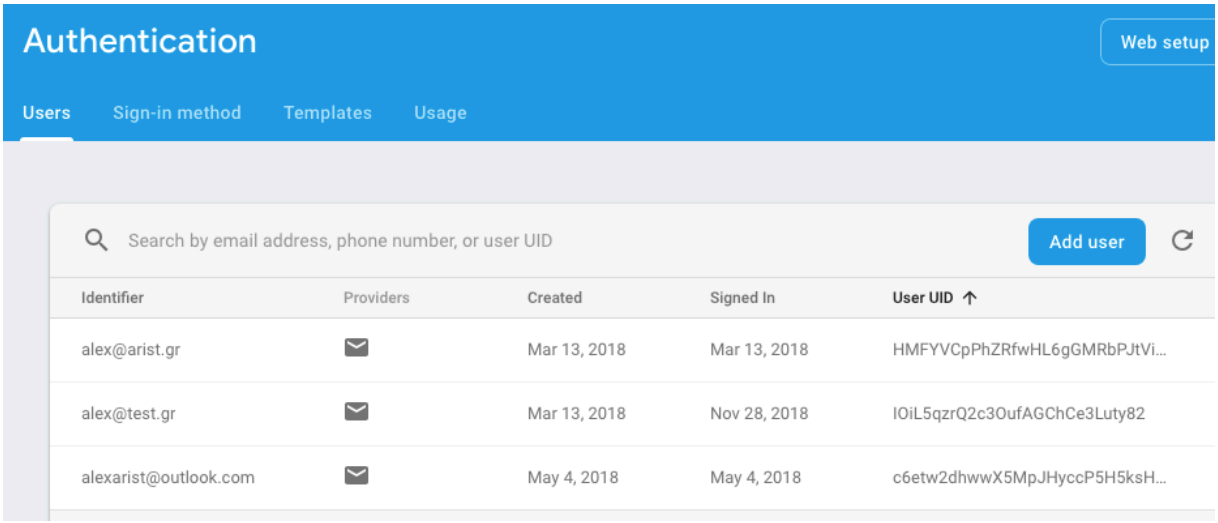
Εντολές	**Χρήση**
import Firebase	Εισαγωγή framework
FirebaseApp.configure()	Μας υποχρεώνει κατά την δημιουργία του Firebase project να το καλέσουμε στο AppDelegate

3.6.2 Δημιουργία χρήστη με το Firebase

Για ενεργοποίηση της δυνατότητας που μας δίνει το Firebase να συνδεθούμε με ένα email και έναν κωδικό πρόσβασης θα πρέπει στη consola του Firebase στο SignIn method να ενεργοποιήσουμε το email/password. Στο SignUpViewController θα εισάγουμε στην SignUp function

Εντολές	**Χρήση**
import FirebaseAuth	Εισαγωγή framework για την πιστοποίηση χρήστη
guard let email =email.text, let password = password.text,let fullname = fullname.text else return	Ανάγνωση και αποθήκευση σε μεταβλητές που με την χρήση guard έχουν πρόσβαση στο κείμενο του .text των τριών labels που απαιτούνται για την δημιουργία νέου χρήστη
Auth.auth().createUser(withEmail: email, password: password)	Δημιουργία χρήστη στο firebase χρησιμοποιώντας τα στοιχεία από τις μεταβλητές
{[weak self] (user,error) in if let error = error { self?.alert(message: error.localizedDescription) return }	Αν προκύψει error κατά την δημιουργία χρήστη καλεί την function alert
print("success")	Αλλιώς εμφάνιση επιτυχής δημιουργία χρήστη

Πλέον κάθε νέος χρήστης εμφανίζεται στην Authentication σελίδα του Firebase αποκτώντας ένα μοναδικό αναγνωριστικό χρήστη (User Uid). [31]



Εικόνα 1.13
Εγγεγραμένοι χρήστες στο firebase

3.6.3 Σύνδεση χρήστη με το Firebase

Παρόμοια με την δημιουργία χρήστη είναι και η σύνδεση ενός χρήστη. Στο SignInViewController θα εισάγουμε στην SignIn function

Εντολές	**Χρήση**
import FirebaseAuth	Εισαγωγή framework για την πιστοποίηση χρήστη
guard let email =email.text, let password = password.text else return	Ανάγνωση και αποθήκευση σε μεταβλητές που με την χρήση guard έχουν πρόσβαση στο κείμενο του .text των δύο labels που απαιτούνται για την σύνδεση ενός χρήστη
Auth.auth().signIn (withEmail: email, password: password)	Σύνδεση ενός χρήστη στο firebase χρησιμοποιώντας τα στοιχεία από τις μεταβλητές
{[weak self] (user,error) in if let error = error { self?.alert(message: error.localizedDescription) return }	Αν προκύψει error κατά την σύνδεση ενός χρήστη καλεί την function alert
print(“success”)	Αλλιώς εμφάνιση επιτυχής σύνδεση χρήστη

3.6.4 Εγκατάσταση βάσης δεδομένων σε πραγματικό χρόνο(Real time database)

Για κάθε φορά που κάποιος χρήστης κάνει SignUp θα πρέπει να ενημερώνεται η βάση δεδομένων με το όνομα και το emal του. Στο SignUpViewController θα εισάγουμε στην SignUp function [32]

Εντολές	**Χρήση**
import FirebaseDatabase	Εισαγωγή framework για την βάση δεδομένων
Database.database().reference().child("Users"). child(user.uid).updateChildValues(["email": email,"name":fullname])	Στην σελίδα Database του Firebase αποκτάμε πρόσβαση στην βάση δεδομένων με σημείο αναφοράς στον κατάλογο το όνομα του project μας μαζι με έναν μοναδικό κωδικό, μετά στον υποκατάλογο Users με τους χρήστες, και στον υποκατάλογο Users μέσα στο μοναδικό User Id είναι καταχωρημένα το email και το name του μοναδικού χρήστη
let changeRequest = user!.CreateProfileChangeRequest() changeRequest.displayName = fullname changeRequest.commitChanges(completion:nil)	Κάθε φορά που κάνουμε εγγραφή νέου χρήστη κρατάει το fullname και επίσης ζητάμε έγκριση να το χρησιμοποήσουμε

3.6.5 Εισαγωγή Alerts και δυνατότητα προσθήκης νέου χρήστη

Δημιουργία swift αρχείου με όνομα Me

Εντολές	**Χρήση**
import FirebaseAuth	Εισαγωγή framework
class Me{ static var uid:String { return Auth.auth().currentUser!.uid}}	Δήλωση κλάσης για την ευκολότερη πρόσβαση στο User ID μας μέσω static var για ταχύτερη πρόσβαση στην μεταβλητή

Στο αρχείο Helpers εισάγουμε

Εντολές	**Χρήση**
func alert(message: String) { }	Δήλωση function alert η οποία δέχεται ένα μήνυμα σαν παράμετρο
let alertController =UIAlertController(title:nil, message: message,preferredStyle: .alert)	Δημιουργία alert ώστε να εμφανίζει το μήνυμα
let okAction = UIAlertAction(title: "OK",style: .default,handler:nil))	Απομάκρυνση του alert όταν ο χρήστης πατάει πάνω στο alert

self.presenter(alertController, animated:true,completion:nil)	Εμφάνιση alert

Στο MessagesTableViewController

Εντολές	**Χρήση**
import SwiftyJSON	Εισαγωγή framework για την ανάλυση του JSON προτύπου που παράγει η βάση δεδομένων μας
import FirebaseDatabase	Εισαγωγή framework FirebaseDatabase για πρόσβαση στην βάση δεδομένων
import Chatto	Εισαγωγή framework
func addContact(email:String) {}	Δήλωση function εισαγωγής email στις επαφές
Database.database().reference().child(“Users”).observeSingleEvent(of: .value,with : { [weak self](snapshot) in	Θα πάρουμε το στιγμιότυπο από τους Users
let snaphot = JSON(snapshot.value as Any).dictionaryValue	Παίρνουμε την τιμή του στιγμιότυπου χρησιμοποιώντας το JSON κατάλογο
if let index = snapshot.index(where: { (key,value) ->Bool	Αν το index υπάρχει, αν ο δείκτης δηλαδή στον κατάλογο υπάρχει
return value[“email”].stringValue == email	Επιστροφή του index αν το email που εισήγαγε ο χρήστης είναι το ίδιο με το email του index στην βάση δεδομένων
let allUpdates = ["/Users/\(Me.uid)/Contacts/\ (snapshot[index].key)" : (["email": snapshot[index].value["email"].stringValue , "name": snapshot[index].value["name"].stringValue]), "/Users/\(snapshot[index].key)/Contacts /\(Me.uid)" : (["email": Auth.auth().currentUser!.email!, "name": Auth.auth().currentUser!.displayName!])]	Ενημέρωση της βάσης δεδομένων ώστε όταν κάποιος χρήστης εισάγει μία επαφή, αυτομάτως ο ένας να είναι στις επαφές του άλλου χρήστη. Χρησιμοποιούμε την διαδρομή των δύο καταλόγων για να έχουμε πρόσβαση στο email και fullname των δύο καταλόγων
Database.database().reference(). updateChildValues(allUpdates)	Ενημέρωση και των δύο καταλόγων
self?.alert(message: ”success”)}	Εισαγωγή alert για επιτυχημένη εισαγωγή επαφής
else { self?.alert(message: ”no such email”) }	Δεν υπάρχει τέτοιο email στην βάση δεδομένων

Εικόνα 1.14

Βάση δεδομένων firebase με τις επαφές των χρηστών

Εντολές	**Χρήση**
func presentAlert() { }	Δήλωση function alert
let alertController =UIAlertController(title:"Email?", message: "Please write the email",preferredStyle: .alert)	Δημιουργούμε ένα alert με μήνυμα παρακαλώ γράψτε το email σας
alert.Controller.addTextField { (textField) in textField.placeholder = "Email" }	Δίνουμε στο alert την δυνατότητα ενός TextField στο παλίσιο το οποίο μπορούμε να εισάγουμε κείμενο
let cancelAction = UIAlertAction(title: "Cancel",style: .cancel,handler:nil)	Δυνατότητα ακύρωσης της εισαγωγής email
let confirmAction = UIAlertAction(title: "Confirm",style: .default)	Δυνατότητα στον χρήστη επιβεβαίωσης του email που έχει εισάγει
{[weak self] in if let email = alertController.TextFields[0].text{ }}	Αν το TextField που εισάγαμε έχει κάποιο κείμενο
self?.addcontact(email:email)	Καλεί την function εισαγωγής επαφής
alertController.addAction(cancelAction)	Κλήση alert cancelAction
alertController.addAction(confirmAction)	Κλήση alert confirmAction
self.present(alertController,animated:true, completion: nil)	Παρουσίαση του alert

Στην Function addContact εισάγουμε

Εντολές	**Χρήση**
self.presentAlert()	Καλούμε το alert

3.6.6 Σύνδεση UI με την βάση δεδομένων

Στο MessagesTableViewController

Εντολές	**Χρήση**
import FirebaseDatabaseUI	Εισαγωγή framework για την σύνδεση του User Interface με την βάση δεδομένων
let Contacts = FUISortedArray(query: Database.database().reference().child("Users").child(Me.uid).child("Contacts"),delegate: nil)/*sorted arry to sort contacts by date sends message*/{ (lhs,rhs) -> ComparisonResult in let lhs = Date(timeIntervalSinceReferenceDate: JSON(lhs.value as Any)["lastMessage"]["date"].doubleValue)/*left hand size*/ let rhs = Date(timeIntervalSinceReferenceDate: JSON(rhs.value as Any)["lastMessage"]["date"].doubleValue)/*right hand size*/ return rhs.compare(lhs)	Σύνδεση επαφών στην βάση δεδομένων με το User Interface ταξινομημένη όμως με βάση την ημερομηνίας αποστολής ή λήψης μηνύματος με την επαφή, οπότε και τα μηνύματα να είναι ταξινομημένα στο Table View με βάση την ημερομηνία.
self.Contacts.observeQuery()	Ελέγχει συνεχώς τον κατάλογο Contacts για αλλαγές και τον συγχρονίζει συνεχώς με το FUIArray, δηλαδή με το Table View
self.Contacts.delegate = self	Κατα τον έλεγχο στον κατάλογο Contacts μέσω του delegate καλεί τις delegate functions από την extension MessageTableViewController. Το delegate με την ιδιότητα του object που έχει ενημερώνει το UITableViewDelegate και εκείνο όλα τα objects delegate στον κώδικα
self.tableView.delegate = self self.tableView.dataSource = self	Απαραίτητη εντολή για να λειτουργήσουν οι function του Table View

Επίσης εισάγουμε μια προκατασκευασμένη function για την μορφή και εμφάνιση της ημερομηνίας στα κελιά με την συνομιλία

Εντολές	Χρήση

func dateFormatter(timestamp: Double?) -> String? {
if let timestamp = timestamp {
let date = Date(timeIntervalSinceReferenceDate: timestamp)
let dateFormatter = DateFormatter()
let timeSinceDateInSeconds = Date().timeIntervalSince(date)
let secondsInDays: TimeInterval = 24*60*60
if timeSinceDateInSeconds > 7 * secondsInDays {
dateFormatter.dateFormat = "dd/MM/yy"
}else if timeSinceDateInSeconds > secondsInDays
{dateFormatter.dateFormat = "EEE"
}else {dateFormatter.dateFormat = "h:mm a"}
return dateFormatter.string(from: date)
}else {return nil}
	Εισάγουμε την προκατασκευασμένη function για το πόση ώρα πέρασε από το τελευταίο μήνυμα που στείλαμε ή λάβαμε

Δήλωση delegate functions του FUIArray με την χρήση extension στην κλάση MessageTableViewController για την εισαγωγή νέων λειτουργιών

Εντολές	Χρήση
extension MessageTableViewController { }	Δήλωση delegate functions του FUIArray, των σειρών δηλαδή μέσα στο Table View
func array(_ array: FUICollection,didAdd object: Any,at index: UInt{ self.tableView.insertRows(at: [IndexPath(row. Int(index)section[0],with: .automatic)}	Οταν κληθεί αυτή η function καλείται να εισάγει μία σειρά στο Table View με την επαφή που προστέθηκε στην βάση δεδομένων

func array(_ array: FUICollection,didMove object: Any,from fromIndex: UInt,to toIndex: UInt {
self.tableView.insertRows(at: [IndexPath(row. Int(toIndex)section[0],with: .automatic)
self.tableView.deleteRows(at: [IndexPath(row. Int(fromIndex)section[0],with: .automatic)
}
	Οταν κληθεί αυτή η function καλείται να ενημερώσει στο Table View οτι μία επαφή μετακινήθηκε από την βάση δεδομένων, να εισάγει μία σειρά στο Table View με την επαφή που μετακινήθηκε στην βάση δεδομένων και να διαγράψει την σειρά στο Table View με την επαφή που ήταν πριν μετακινηθεί

func array(_ array: FUICollection,didRemove object: Any,at index: UInt{
self.tableView.deleteRows(at: [IndexPath(row. Int(index)section[0],with: .automatic
}
	Οταν κληθεί αυτή η function καλείται να ενημερώσει στο Table View οτι μία επαφή αφαιρέθηκε από την βάση δεδομένων και να διαγράψει την σειρά στο Table View με την επαφή που διεγράφει

func array(_ array: FUICollection,didChange object: Any,at index: UInt{ self.tableView.reloadRows(at: [IndexPath(row. Int(index)section[0],with: .automatic}	Οταν κληθεί αυτή η function καλείται να ενημερώσει στο Table View τις αλλαγές της επαφής που βρίσκεται στην βάση δεδομένων

Δήλωση delegate functions του Table View μέσα στην extension MessageTableViewController

Εντολές	**Χρήση**
func tableView(_tableView:UITableView, numberOfRowsInSection section: Int) -> Int { return Int(self.contacts.count) }	Functions του Table View η οποία επιστρέφει τον αριθμό των σειρών του FUIArray
func tableView(_tableView:UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell { let cell =tableView.dequeueReusableCell(withIdentifier: “cell”,for: indexPath) let info = JSON(Contacts[(UInt(indexPath.row))] as? Datasnapshot)?.value as Any).dictionaryObject cell.Name.text= info?[”name”] as? String lastMessage.text = info["lastMessage"]?["text"].string cell.lastMessageDate.text = dateFormatter(timestamp: info["lastMessage"]?["date"].double) return cell }	Οταν κληθεί αυτή η function παίρνει όλες τις πληροφορίες απο την βάση δεδομένων και ενημερώνει το κελί για την συγκεκριμένη επαφή καθώς και μία προεσκόπιση του τελευταίου μηνύματος που έχει ανταλάξει ο χρήστης με την επαφή. Τέλος δίνει στο κελί την ημερομηνία της τελευταίας αποστολής ή λήψης μηνύματος

3.6.7 Αποστολή μηνυμάτων μέσω του Firebase

Στην function HandleSend απο την στιγμή που έχουμε συνδέσει το Firebase αλλάζουμε

Εντολές	**Χρήση**
let senderID = Me.uid	Δήλωση μεταβλητής senderID ίσο με το User ID με το οποίο έχουμε συνδεθεί
let messageUID = “(\(double)”) + senderID. replaxingOccurances(of: “.”,with: “ ”)	Με την δήλωση μεταβλητής messageUID παίρνουμε το uid του μηνύματος επειδή όμως το Firebase δεν αναγνωρίζει τους ειδικούς χαρακτήρες όπως η τελεία την αντικαθιστούμε με κενό

let message = MessageModel(uid: messageUID, senderId: senderId, type: TextModel.chatItemType, isIncoming: false, date: date, status: .sending)	Δήλωση μηνύματος κειμένου, αλλάζουμε από nil σε sending. Όταν κληθεί η function addMessage το μηνυμά μας θα είναι σε φάση sending και όταν περάσει από το Firebase τότε θα μας ειδοποιήσει για success sending

Στο TextModel ορίζουμε το status του μηνύματος για να μην έχουμε errors

Εντολές	**Χρήση**

var status: MessageStatus {
get {
return self._messageModel.status
} set {
self._messageModel.status = newValue
}
	Το status θα μπορεί δέχεται και από εμάς τιμές από την στιγμή που δεν είναι μόνο getter αλλά και setter

Δημιουργούμε μια function στην DataSource

Εντολές	**Χρήση**
func updateTextMessage(uid: String, status: MessageStatus){	Δημιουργία function για την εισαγωγή status ενός μηνύματος σε sending
if let index = self.controller.items.index(where: { (message) -> Bool in return message.uid == uid }) {	Βρίσκουμε το index του uid του μηνύματος από τον πίνακα items. Αν το uid του μηνύματος είναι ίδιο με το uid στο Firebase
message.status = .status self.delegate?.chatDataSourceDidUpdate(self)	Τότε το status αλλάζει σε success και μετά καλούμε το delegate

Δήλωση delegate function του Table View μέσα στην extension MessageTableViewController

Εντολές	**Χρήση**
func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath)	Η κλάση αυτή ενεργοποιείται μόλις επιλέξεις μια γραμμή στο Table View επιστρέφοντάς μας το indexPath της γραμμής που εχουμε επιλέξει
let uid = (Contacts[UInt(indexPath.row)] as? DataSnapshot)?.key	Αποθηκεύει σε μία μεταβλητή το User ID του συνομιλητή

Στο ChatLogController εισάγουμε

Εντολές	**Χρήση**

import FirebaseDatabase	Εισαγωγή framework για την επικοινωνία με την βάση δεδομένων
var userUID = String	Δήλωση μεταβλητής
func sendOnLineTextMessage(text: String, uid: String, double: Double, senderId: String){	Δημιουργία function για την αλληλεπίδραση των μηνυμάτων με το Firebase
let message = ["text": text, "uid": uid, "date": double, "senderId": senderId, "status": "success","type": TextModel.chatItemType] as [String : Any]	Αποθηκεύουμε σε έναν πίνακα το μήνυμα γιατί μόνο έτσι μπορει να ενημερώνεται ο κατάλογος
let childUpdates = ["User-messages”/\(senderId)/\(self.userUID)/\ (uid)": message,//"User-messages”/\(self.userUID)/\(senderId)/\(uid)": message/,"Users/\(Me.uid)/Contacts/\(self.userUID)/lastMessage": message,"Users/\(self.userUID)/Contacts/\(Me.uid)/lastMessage": message]	Αφού πρώτα δημιουργούμε τον κατάλογο User messages, ενημερώνουμε και στους δύο χρήστες τους καταλόγους με τα μηνύματα που ανταλλάσουν, μέσω paths
Database.database().reference().updateChildValues (childUpdates) { [weak self] (error, _) in if error != nil { self?.dataSource.updateTextMessage(uid: uid, status: .failed) return }self?.dataSource.updateTextMessage(uid: uid, status: .success)}	Ενημερώνει την βάση δεδομένων με το μήνυμα. Μόλις ενημερώσει την βάση δεδομένων και είναι επιτυχημένη αλλάζει το status του μηνύματος από sending σε success. Αν δεν μπορέσει να ενημερώσει την βάση δεδομένων θα προσδώσει στο μήνυμα το status failed αλλιώς success
deinit{ }	Κάθε φορά φορά που θα βγαίνουμε από τον View Controller θα εκτελείται αυτή η εντολή και θα αποδεσμεύει κλάσεις, properties, methods κ.α. για την ορθή λειτουργία και καθαρισμό της εφαρμογής

Εικόνα 1.15

Βάση δεδομένων firebase του χρήστη με τα μηνύματα και τα στοιχεία τους

Τέλος καλούμε την function sendOnLineTextMessage στην handleSend

self?.sendOnLineTextMessage (text: text, uid: messageUID, double: double, senderId: senderId)

3.6.8 Φόρτωση μηνυμάτων απο το Firebase

Στην MessagesTableViewController εισάγουμε

Εντολές	**Χρήση**
Database.database().reference().child("User-messages").child(Me.uid).keepSynced(true)	Με βάση το User ID κρατάμε όλα τα μηνύματα συγχρονισμένα, αποθηκευμένα σε ένα offline cache και έχουμε πρόσβαση χωρίς internet
let reference = Database.database().reference().child("User-messages").child(Me.uid).child(uid!).queryLimited(toLast: 51)	Πρόσβαση στα τελευταία 51 μηνύματα με βάση του User ID που συνδεθήκαμε και το User ID του συνομιλιτή
self.tableView.isUserInteractionEnabled = false	Κάθε φορά που κάνουμε κλικ σε ένα κελί απενεργοποιούμε ολόκληρο το υπόλοιπο Table View
reference.observeSingleEvent(of: .value, with: { [weak self] (snapshot) in	Αποθηκεύουμε σε έναν πίνακα σαν JSON τα values στοιχεία του μηνύματος ταξινομημένα με βάση την ημερομηνία, μέσω της εντολής observe η οποία θέλει λίγα δευτερόλεπτα να

let messages = Array(JSON(snapshot.value as Any).dictionaryValue.values).sorted(by: { (lhs, rhs) -> Bool in return lhs["date"].doubleValue < rhs["date"].doubleValue })	παρατηρήσει τα μηνύματα γι'αυτο απενεργοποιούμε ολόκληρο το υπόλοιπο Table View μην τυχόν με δεύτερο πάτημα πάνω στο κελί ξανακληθούν functions όπως η didSelect και κρασάρει η εφαρμογή
print(messages)	Εκτυπώνει τα μηνύματα
let chatlog = ChatLogController() chatlog.userUID = uid! chatlog.dataSource = DataSource(initialMessages: converted,uid: uid!)	Σύνδεση με τον ChatLogController
self?.tableView.isUserInteractionEnabled = true	Επανενεργοποίηση του Table View

Στο Helpers

Εντολές	**Χρήση**
import Chatto import ChattoAdditions import SwiftyJSON	Εισαγωγή Frameworks
extension NSObject { func convertToChatItemProtocol(messages: [JSON]) -> [ChatItemProtocol] {	Μετατροπή πίνακα μηνυμάτων JSON σε πίνακα ChatItemProtocol. Μόνο έτσι μπορούν να εισαχθούν στο DataSource
convertedMessages = messages.map({ (message) -> ChatItemProtocol in	Πρόσβαση σε κάθε μήνυμα μέσω του map εισάγωντας σε κάθε μήνυμα την επισρεφόμενη τιμή δηλαδή ChatItemProtocol
let senderId = message["senderId"].stringValue let model = MessageModel(uid: message["uid"].stringValue, senderId: senderId, type: message["type"].stringValue, isIncoming: senderId == Me.uid ? false : true, date: Date(timeIntervalSinceReferenceDate: message["date"].doubleValue), status: message["status"] == "success" ? MessageStatus.success : MessageStatus.sending)	Πρόσβαση στα values του μηνύματος
let textMessage = TextModel(messageModel: model, text: message["text"].stringValue)	Αρχικοποίηση μηνύματος μέσω του TextModel
return convertedMessages **let converted = self!.convertToChatItemProtocol(messages: messages)**	Επιστροφή του πίνακα με τα μετατρέψιμα μηνύματα

Τέλος το καλούμε μέσα στην function didSelectRow

```
let converted = self!.convertToChatItemProtocol( messages: messages)
```

3.6.9 Αρίθμηση μηνυμάτων (pagination)

Στην ChatItemsController θα εισάγουμε μία function

Εντολές	**Χρήση**
import SwiftyJSON	Εισαγωγή framework
var loadMore = false	Δίνουμε λογική τιμή στην μεταβλητή

func loadIntoItemsArray(messagesNeeded: Int, moreToLoad: Bool)
{for index in stride(from: initialMessages.count - items.count, to: initialMessages.count - items.count - messagesNeeded, by: -1){
self.items.insert(initialMessages[index - 1], at: 0)
self.loadMore = moreToLoad
}
	Για να παραμένει το loadMore true και να εκτυπώνει μηνύματα θα πρέπει μέσα στην MessagesTableViewController να λέμε στο Firebase φόρτωσε 51 μημύματα ούτως ώστε σε περίπτωση που τα μηνύματα στο σύνολο είναι 52 να εκτυπώσει μία φορά από το initialMessages 50 μηνύματα και στην συνέχεια να είναι πάλι true το loadMore και να εκτυπώσει και τα υπόλοιπα δύο

func loadPrevious(Completion: @escaping completeLoading){ Database.database().reference().child("User-messages").child(Me.uid).child(userUID). queryEnding(atValue: nil, childKey: self.items.first?.uid).queryLimited(toLast: 52).observeSingleEvent(of: .value, with: { [weak self] (snapshot) in var messages = Array(JSON(snapshot.value as Any).dictionaryValue.values).sorted(by: { (lhs, rhs) -> Bool in return lhs["date"].doubleValue < rhs["date"].doubleValue}) messages.removeLast()	Ελέγχουμε στο Firebase με το User ID του συνομιλιτή μας, παίρνουμε το message UID του τελευταίου μηνύματος που εμφανίστηκε στο Collection View, το βρίσκουμε στην βάση δεδομένων και φορτώνει τα υπόλοιπα μηνύματα, αν υπάρχουν, από αυτό το μήνυμα και κάτω. Τέλος ταξινομούμε τα μηνύματα με βάση την ημερομηνία και διαγράφουμε το τελευταίο μήνυμα γιατί από την στιγμή που θα ξεκινήσουμε το loading από το message UID του τελευταίου μηνύματος που εμφανίστηκε δεν θέλουμε να εμφανιστεί δύο φορές το ίδιο μήνυμα.
self?.loadMore = messages.count > 50 let converted =	Αν τα μηνύματα είναι περισσότερα απο 50 καλείται το loadMore
self!.convertToChatItemProtocol(messages: messages)for index in stride(from: converted.count, to: converted.count - min(messages.count, 50), by: -1) { self?.items.insert(converted[index - 1], at: 0)}	Μετατρέπουμε τα μηνύματα σε ChatItemProtocol ξεκινώντας από το τελευταίο μήνυμα προς το πρώτο
typealias completeLoading = () -> Void	Δήλωση Completion
Completion() messages.filter({ (message) -> Bool in return message["type"].stringValue == PhotoModel.chatItemType	Καλούμε την standar διαδικασία Completion() όπου το delegate θα κληθεί μόνο όταν τελείωσει το loading των μηνυμάτων ώστε να μην κλειθεί

}).forEach({ (message) in self?.parseURLs(UID_URL: (key: message["uid"].stringValue, value: message["image"].stringValue))})	κάτι πριν ολοκληρωθεί η διαδικασία και κρασάρει η εφαρμογή

Στην DataSource εισάγουμε

Εντολές	**Χρήση**
var currentlyLoading = false	Δίνουμε λογική τιμή στην μεταβλητή
init(initialMessages:[ChatItemProtocol], uid: String) { self.controller.initialMessages = initialMessages self.controller.userUID = uid self.controller.loadIntoItemsArray(messagesNeed ed: min(initialMessages.count, 50),moreToLoad: initialMessages.count > 50)	Εκτυπώνει το πολύ 50 μηνύματα από τον πίνακα messages κάθε φορά

var hasMoreNext: Bool{
return false }
var hasMorePrevious: Bool{
return controller.loadMore }
	Αν το το loadMore είναι true καλείται η function loadPrevious

func loadPrevious() {
if currentlyLoading == false {
currentlyLoading = true
controller.loadPrevious(){
self.delegate?.chatDataSourceDidUpdate(self,
updateType: .pagination)
self.currentlyLoading = false
}
	Αν την ώρα που δεν φορτώνονται μηνύματα κάνει το currentlyLoading true δεν θα μην κληθεί ο observe πολλές φορές και δεν θα εκτυπώσει διπλά τα μηνύματα. Θα πρέπει να εκτελεστεί όταν τελειώσει η loadPrevious του ChatItemsController, οπότε όσο κάνουμε scrolling δεν θα μπαίνει στην if γιατί πρέπει να εκτελεστεί μία φορά μόνο. Όταν ολοκληρωθεί το Loading όσο κάνουμε scrolling θα μπαίνει στην if και θα κάνει paginate τα μηνύματα

func adjustWindow(){
self.items.removeFirst(200)
self.loadMore = true
}
	Τέλος αφαιρούνται τα πρώτα 200 μηνύματα ώστε να μην είναι φορτωμένη όλη η συνομιλία μας και κάνει loadMore ίσο με true. Όσο είναι ίσο με true θα κάνει pagination

3.6.10 Εισερχόμενα μηνύματα

Στο ChatLogController

Εντολές	**Χρήση**
import SwiftyJSON import FirebaseDatabaseUI	Εισαγωγή frameworks

var MessagesArray: FUIArray!	Δήλωση μεταβλητής τύπου FUIArray

Στην function didSelect εισάγουμε

Εντολές	**Χρήση**
chatlog.MessagesArray = FUIArray(query: Database.database().reference().child("User-messages").child(Me.uid).child(uid!).queryStarting(atValue: nil, childKey: converted.last?.uid), delegate: nil)	Παρατηρούμε στο Firebase στα μηνύματά μας με βάση τον συνομιλητή, αν στο τελευταίο μήνυμα που παρατηρούμε προστέθηκε κάποιο άλλο μήνυμα

Στο ChatLogController στην viewDidLoad

Εντολές	**Χρήση**
self.MessagesArray.observeQuery() self.MessagesArray.delegate = self	Παρατηρούμε στο Firebase το τελευταίο μήνυμα με βάση τον συνομιλητή και έπειτα καλούμε τις delegate functions

extension ChatLogController {
func array(_ array: FUICollection, didAdd object: Any, at index: UInt) {
let message = JSON((object as! DataSnapshot).value as Any)
	Δημιουργούμε μία delegate function όπου καλείται όταν ο observe εντοπίσει ένα καινούργιο μήνυμα και μας δίνει το value του νέου μηνύματος σε JSON

let contains = self.dataSource.controller.items.contains { (collectionViewMessage) -> Bool in return collectionViewMessage.uid == message["uid"].stringValue}	Ελέγχουμε αν το μήνυμα που θα πάρουμε απο την βάση δεδομένων υπάρχει ήδη στο Collection View
if contains == false { let model = MessageModel(uid: message["uid"].stringValue, senderId: senderId, type: type, isIncoming: senderId == Me.uid ? false : true, date: Date(timeIntervalSinceReferenceDate: message["date"].doubleValue), status: message["status"] == "success" ? MessageStatus.success : MessageStatus.sending)	Αν δεν περιέχεται το μήνυμα στο Collection View μετατρέπουμε το JSON σε TextModel και προσθέτουμε το μήνυμα καλώντας την addMessage

3.6.11 Αποστολή και λήψη μηνυμάτων εικόνας

Την ίδια διαδικασία που εφαρμόσαμε με τα μηνύματα κειμένου θα εφαρμόσουμε και με τα μηνύματα εικόνας. Στο ChatLogController δημιουργούμε μία function

Εντολές	Χρήση
import Kingfisher import FirebaseStorage	Εισαγωγή frameworks για την αποθήκευση δεδομένων στην βάση δεδομένων και μετατροπή των δεδομένων σε images
func uploadToStorage(photo: photoModel){}	Δημιουργία function με παράμετρο photoModel
let imageName = photo.uid	Αρχικοποίηση imageName

Στο handlePhoto [33]

Εντολές	Χρήση
self.uploadToStorage(photo: photoMessage)	Καλούμε την function uploadToStorage
let storage = Storage.storage().reference().child("images").child(imageName)	Δήλωνουμε στην μεταβλητή storage το path όπου θα αποθηκεύουμε τις εικόνες στον κατάλογο images
let data = UIImagePNGRepresentation(photo.image)	Μετατέπουμε την εικόνα σε png δεδομένα εικόνας

storage.putData(data!, metadata: nil) { [weak self] (metadata, error) in
if let imageURL = metadata?.downloadURL()?.absoluteString {
self?.sendOnlineImageMessage(photoMessage: photo, imageURL: imageURL)
} else {
self?.dataSource.updatePhotoMessage(uid: photo.uid, status: .failed)}}
	Αποθηκεύουμε την εικόνα στο storage του Firebase. Αν δεν υπάρχουν errors και μας επιστρέφεται ένα URL τύπου String με την εικόνα, ενημερώνει το URL στο Firebase αλλιώς ενημερώνει οτι απέτυχε

func sendOnlineImageMessage(photoMessage: PhotoModel, imageURL: String) {	Δήλωση κλάσης πανομοιότυπη με την sendOnlineTextMessage
let message = ["image": imageURL, "uid": photoMessage.uid, "date": photoMessage.date.timeIntervalSinceReferenceDate, "senderId": photoMessage.senderId, "status": "success","type": PhotoModel.chatItemType] as [String : Any]	Παρόμοια με την sendOnlineTextMessage μόνο που αλλάζουμε σε PhotoModel και το text γίνεται image καθώς παίρνουμε το imageURL
let childUpdates = ["User-messages/\(photoMessage.senderId)/\(self.userUID)/\(photoMessage.uid)": message,/*updates mynode*/"User-messages/\(self.userUID)/\(photoMessage.senderId)/\(photoMessage.uid)": message/*updates his message	Πάλι σαν την sendOnlineTextMessage, αλλάζουμε το path σε uid του μηνύματος της εικόνας

node*/,"Users/\(Me.uid)/Contacts/\(self.userUID)/lastMessage": message,"Users/\(self.userUID)/Contacts/\(Me.uid)/lastMessage": message] Database.database().reference().updateChildValues(childUpdates) { [weak self] (error, _) in if error != nil {	

Στο DataSource

Εντολές	**Χρήση**
func updatePhotoMessage(uid: String, status: MessageStatus){ if let index = self.controller.items.index(where: { (message) -> Bool in return message.uid == uid }) { let message = self.controller.items[index] as! PhotoModel message.status = status self.delegate?.chatDataSourceDidUpdate(self)}	Δημιουργούμε παρόμοια function με την updateTextMessage και αλλάζουμε από TextModel σε PhotoModel

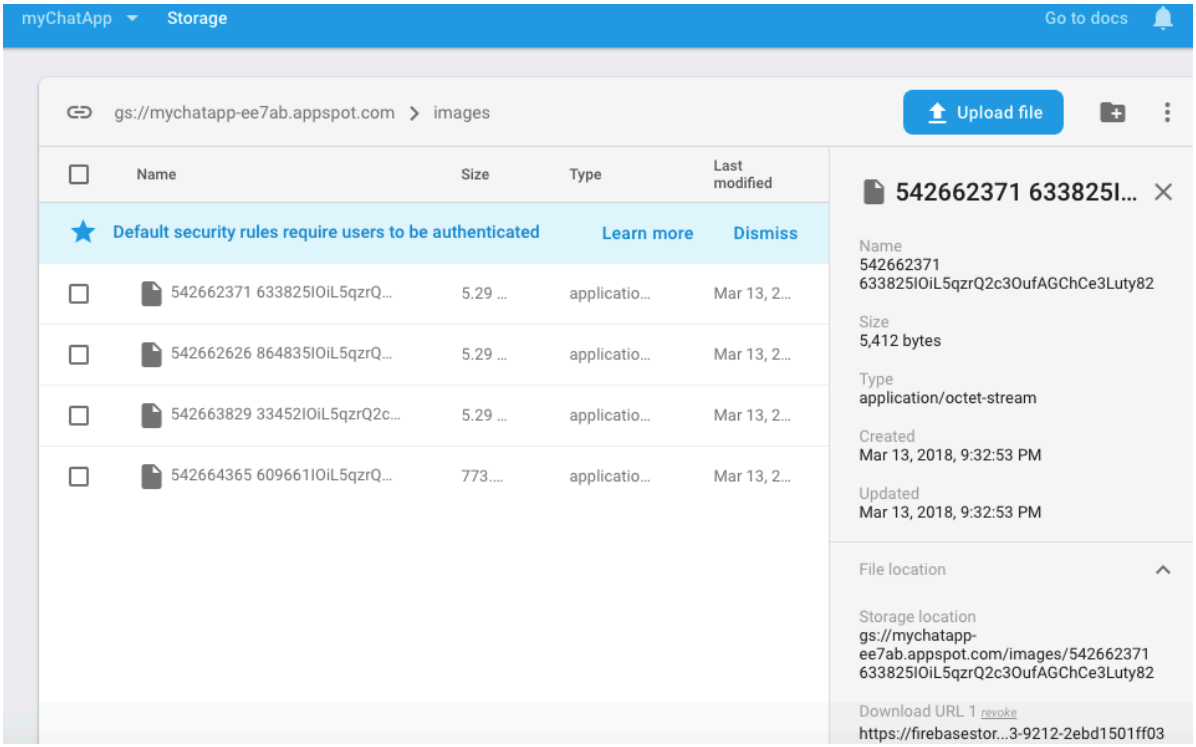
Στο PhotoModel

Εντολές	**Χρήση**
var status: MessageStatus { get { return self._messageModel.status } set { self._messageModel.status = newValue}	Το status θα μπορεί δέχεται και από εμάς τιμές από την στιγμή που δεν είναι μόνο getter αλλά και setter, όπως και για το textMessage στο TextModel

Στην uploadToStorage καλούμε τις functions

Εντολές	**Χρήση**
self?.dataSource.updatePhotoMessage(uid: photoMessage.uid, status: .failed) return} self?.dataSource.updatePhotoMessage(uid: photoMessage.uid, status: .success)}	Καλούμε τις functions αν υπάρχει error το status του μηνύματος εικόνας γίνεται failed αλλιώς success
if type == TextModel.chatItemType { let textMessage = TextModel(messageModel: model, text: message["text"].stringValue)	Έλεγχος το μήνυμα τι τύπος είναι, αν είναι TextModel δημιουργεί το textMessage και το εισάγει στην addMessage αλλιώς αν είναι

self.dataSource.addMessage(message: textMessage)
} else if type == PhotoModel.chatItemType {
KingfisherManager.shared.retrieveImage(with: URL(string: message["image"].stringValue)!, options: nil, progressBlock: nil, completionHandler: { [weak self] (image, error, _, _) in
if error != nil {
self?.alert(message: "error receiving image from user")} else {
let photoMessage = PhotoModel(messageModel: model, imageSize: image!.size, image: image!)
self?.dataSource.addMessage(message: photoMessage)}
	PhotoModel χρησιμοποιούμε το Kingfisher για να έχουμε πρόσβαση στην εικόνα απο το URL και εάν δεν έχουμε κάποιο error κατά την διαδικασία ανάκτησης της εικόνας, εισάγει στην addMessage το photoMessage



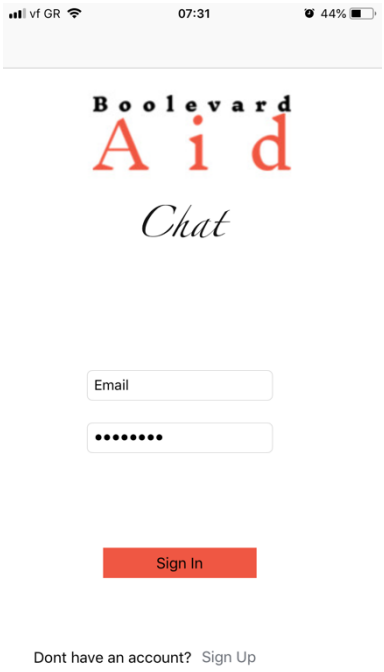
Εικόνα 1.16

Βάση δεδομένων firebase με τις εικόνες από τα μηνύματα εικόνας

3.7 Οθόνες εφαρμογής

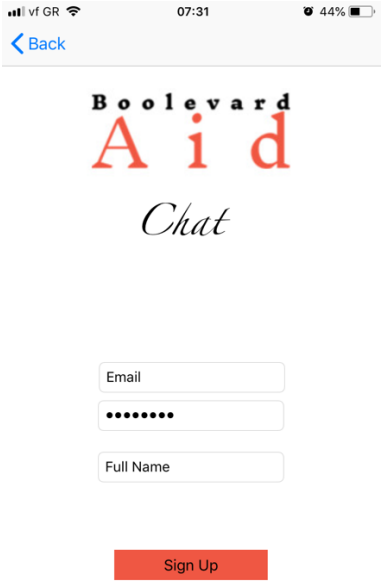
Εικόνα 1.17

Εικόνα εφαρμογής στην επιφάνεια εργασίας της συσκευής



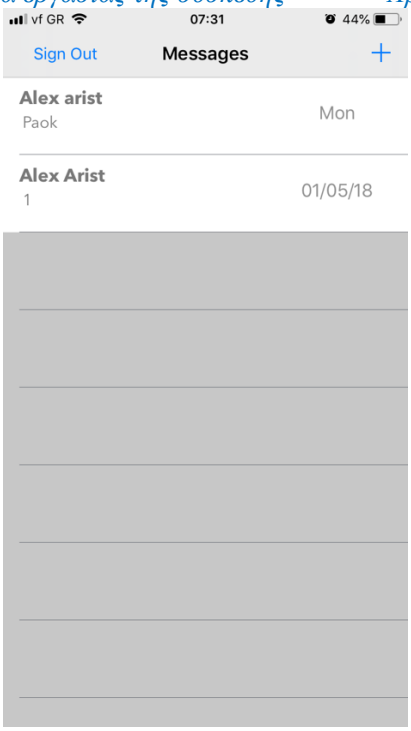
Εικόνα 1.18

Αρχική οθόνη-οθόνη σύνδεσης στο app



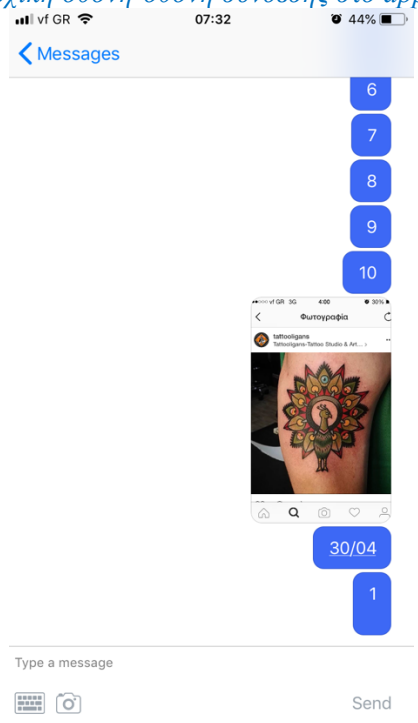
Εικόνα 1.19

Οθόνη εγγραφής στο app



Εικόνα 1.20

Οθόνη συνομιλιών-επαφών



Εικόνα 1.21

Οθόνη συνομιλίας

4 Συμπεράσματα & επεκτάσεις εφαρμογής

Η δημιουργία αυτής της πτυχιακής εργασίας δημιούργησε πολλαπλά οφέλη στον κατασκευαστή της. Καταρχάς η εξοικείωση με την γλώσσα προγραμματισμού Swift, η οποία θεωρείται ανερχόμενη δύναμη στον τομέα κατασκευής εφαρμογών (όλων των mobile αλλά και web εφαρμογών) και η διαπίστωση των ιδιαιτεροτήτων προγραμματισμού ενός προγράμματος αποτελεί ένα σπουδαίο εφόδιο στην συλλογή μου για τυχόν αναζήτηση εργασίας στον τομέα ανάπτυξης τεχνολογικών εφαρμογών. Επίσης δεν πρέπει να παραράται ότι η εμβάθυνση στα frameworks παίζει σημαντικότατο ρόλο στην εξοικείωση με την γλώσσα προγραμματισμού για την οποία προορίζονται, καθώς από τη μία έχεις μια ισχυρή εργαλειοθήκη διαθέσιμη να την εντάξεις στο πρόγραμμα που σχεδιάζεις και να την διαχειριστείς με τον τρόπο που επιθυμείς και από την άλλη η σιγουριά που σου δίνει για την απόδοσή της η συγκεκριμένη εργαλειοθήκη είναι εξίσου σημαντική καθώς δεν χρειάζεται να ανησυχείς για το αν θα προκύψουν προγραμματιστικά σφάλματα αλλά και προβλήματα συμβατότητας, κάτι που σε γλιτώνει από πολλές ώρες ανάλυσης και επίλυσης του κάθε προβλήματος. Κατά δεύτερον, η γνωριμία με το Xcode IDE και η δυνατότητα παραγωγής προγραμμάτων που σου δίνουν τα εργαλεία του, προσφέρουν μία σημαντική τεχνογνωσία για την μελλοντική επαγγελματική εμπειρία.

Οι μελλοντικές επεκτάσεις της εφαρμογής Boolevard Aid περιστρέφονται γύρω από τέσσερις άξονες. Ο πρώτος έχει να κάνει με την βάση δεδομένων, και συγκεκριμένα με την δημιουργία μίας ισχυρής βάσης δεδομένων με την συμμετοχή της δημόσιας διοίκησης, Εθνικών Οργανισμών, Νοσοκομείων και Μη Κυβερνητικών οργανώσεων έτσι ώστε να αναπτυχθεί η τηλεϊατρική σε μεγάλο βαθμό, με αποτέλεσμα μία δημόσια υπηρεσία να είναι αποκλειστικά επιφορτισμένη με την τηλεϊατρική. Στον δεύτερο άξονα ενσωματώνονται όλες εκείνες οι λειτουργίες για την επικοινωνία μεταξύ των χρηστών, όπως είναι η δυνατότητα βιντεοκλήσης αλλά και η αποστολή βίντεο για την διευκόλυνση τόσο ως προς την εξέταση αλλά και ως προς την θεραπεία των ασθενών. Ο τρίτος άξονας περιλαμβάνει την διαδικασία ειδοποιήσεων (notifications) που θα παράγει η εφαρμογή κατά την διάρκεια συγκεκριμένων συμβάντων όπως

για παράδειγμα η λήψη ενός εισερχόμενου μηνύματος, η σύνδεση ενός χρήστη από τις επαφές κλπ. Τέλος ο τέταρτος άξονας έχει να κάνει με την δυνατότητα σύνδεσης νέων χρηστών μέσω τρίτων προυπάρχωντων λογαριασμών σε εταιρίες τεχνολογίας όπως η Google, Facebook κ.α.

Εν κατακλείδι η δημιουργία αυτής της εφαρμογής που ευελπιστώ να χρησιμοποιηθεί σαν ένα σημαντικό επιπλέον εργαλείο στην φαρέτρα της ϊατρικής κοινότητας και των ασθενών τους, θα έχει ως αποτέλεσμα την εκμηδένιση των κοινωνικών ανισοτήτων, και θα αποτελέσει μια τεράστια προσωπική επιτυχία και ηθική ικανοποίηση.

Κώδικας εφαρμογής

```
//
//  AppDelegate.swift
//  MyChatApp
//
//  Created by Alexandros on 13/1/18.
//  Copyright © 2018 Alexandros. All rights reserved.
//
import UIKit
import Firebase
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
   var window: UIWindow?
   func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {
      FirebaseApp.configure()
      // Override point for customization after application launch.
      return true
   }
   func applicationWillResignActive(_ application: UIApplication) {
      // Sent when the application is about to move from active to inactive state. This can occur for certain types of temporary
interruptions (such as an incoming phone call or SMS message) or when the user quits the application and it begins the
transition to the background state.
      // Use this method to pause ongoing tasks, disable timers, and invalidate graphics rendering callbacks. Games should use
this method to pause the game.
   }
   func applicationDidEnterBackground(_ application: UIApplication) {
      // Use this method to release shared resources, save user data, invalidate timers, and store enough application state
information to restore your application to its current state in case it is terminated later.
```

```
    // If your application supports background execution, this method is called instead of applicationWillTerminate: when the user quits.
  }
  func applicationWillEnterForeground(_ application: UIApplication) {
    // Called as part of the transition from the background to the active state; here you can undo many of the changes made on entering the background.
  }

  func applicationDidBecomeActive(_ application: UIApplication) {
    // Restart any tasks that were paused (or not yet started) while the application was inactive. If the application was previously in the background, optionally refresh the user interface.
  }
  func applicationWillTerminate(_ application: UIApplication) {
    // Called when the application is about to terminate. Save data if appropriate. See also applicationDidEnterBackground:.
  }}
```

```
    //
    //  ViewController.swift
    //  MyChatApp
    //
    //  Created by Alexandros on 13/1/18.
    //  Copyright © 2018 Alexandros. All rights reserved.
    //
    import UIKit
    import Chatto
    import ChattoAdditions
    import FirebaseAuth
    import FirebaseDatabase
    import FirebaseDatabaseUI
    import SwiftyJSON
    import FirebaseStorage
    import Kingfisher
    class ChatLogController: BaseChatViewController, FUICollectionDelegate {
      var presenter : BasicChatInputBarPresenter!
      var dataSource : DataSource!
      var decorator = Decorator()
      var userUID = String()//use it that way to remove optional from firebase id's instead userUID: String!
      var MessagesArray: FUIArray!
      override func createPresenterBuilders() -> [ChatItemType : [ChatItemPresenterBuilderProtocol]] {
        let textMessageBuilder = TextMessagePresenterBuilder(viewModelBuilder: TextBuilder(), interactionHandler: TextHandler())
        let photoPresenterBuilder = PhotoMessagePresenterBuilder(viewModelBuilder: PhotoBuilder(), interactionHandler: PhotoHandler())
        return [TextModel.chatItemType: [textMessageBuilder],PhotoModel.chatItemType: [photoPresenterBuilder]]
      }
      override func createChatInputView() -> UIView {
        let inputbar = ChatInputBar.loadNib()
        var apperance = ChatInputBarAppearance()
        apperance.sendButtonAppearance.title = "Send"
        apperance.textInputAppearance.placeholderText = "Type a message"
        self.presenter = BasicChatInputBarPresenter(chatInputBar: inputbar, chatInputItems: [handleSend(),handlePhoto()], chatInputBarAppearance: apperance)
        return inputbar
```

```
        }
        func handleSend() -> TextChatInputItem {
            let item = TextChatInputItem()
            item.textInputHandler = { [weak self] text in
                let date = Date()
                let double = Double(date.timeIntervalSinceReferenceDate)
                let senderId = Me.uid
                let messageUID = ("\(double)" + senderId).replacingOccurrences(of: ".", with: " ")//timestamp first then the id
because it will sorted by date and not by the sender id
                let message = MessageModel(uid: messageUID, senderId: senderId, type: TextModel.chatItemType, isIncoming:
false, date: date, status: .success)
                let textMessage = TextModel(messageModel: message, text: text)
      self?.dataSource.addMessage(message: textMessage)
                self?.sendOnLineTextMessage(text: text, uid: messageUID, double: double, senderId: senderId)}
            return item
        }
        func handlePhoto() -> PhotosChatInputItem {
            let item = PhotosChatInputItem(presentingController: self)
            item.photoInputHandler = { [weak self] photo in
                let date = Date()
                let double = Double(date.timeIntervalSinceReferenceDate)
                let senderId = Me.uid
                let messageUID = ("\(double)" + senderId).replacingOccurrences(of: ".", with: " ")
                let message = MessageModel(uid: messageUID, senderId: senderId, type: PhotoModel.chatItemType, isIncoming:
false, date: date, status: .sending)
                let photoMessage = PhotoModel(messageModel: message, imageSize: photo.size, image: photo)
                self?.dataSource.addMessage(message: photoMessage)
                self?.uploadToStorage(photo: photoMessage)
            }
            return item
        }
        override func viewDidLoad() {
            super.viewDidLoad()
            self.chatDataSource = self.dataSource
            self.chatItemsDecorator = self.decorator
            self.constants.preferredMaxMessageCount = 300
            self.MessagesArray.observeQuery()
            self.MessagesArray.delegate = self//to trigger delagate functions
      /*print(userUID) if userUID has optional in front check it*/
            // Do any additional setup after loading the view, typically from a nib.
        }
      func sendOnLineTextMessage(text: String, uid: String, double: Double, senderId: String){
            let message = ["text": text, "uid": uid, "date": double, "senderId": senderId, "status": "success","type":
TextModel.chatItemType] as [String : Any]//updates our message node
            let childUpdates = ["User-messages/\(senderId)/\(self.userUID)/\(uid)": message,/*updates mynode*/"User-
messages/\(self.userUID)/\(senderId)/\(uid)": message/*updates his message
node*/,"Users/\(Me.uid)/Contacts/\(self.userUID)/lastMessage":
message,"Users/\(self.userUID)/Contacts/\(Me.uid)/lastMessage": message]

            Database.database().reference().updateChildValues(childUpdates) { [weak self] (error, _) in//updates the message
to our database
                if error != nil {
      self?.dataSource.updateTextMessage(uid: uid, status: .failed)
                    return
            }
```

```
                self?.dataSource.updateTextMessage(uid: uid, status: .success)}
        }
            func uploadToStorage(photo: PhotoModel) {//it make some time to firebase/Storage to storage image
                let imageName = photo.uid
                let storage = Storage.storage().reference().child("images").child(imageName)
                let data = UIImagePNGRepresentation(photo.image)
                    storage.putData(data!, metadata: nil) { [weak self] (metadata, error) in
                      if let imageURL = metadata?.downloadURL()?.absoluteString {
                         self?.sendOnlineImageMessage(photoMessage: photo, imageURL: imageURL)
                       } else {
                       self?.dataSource.updatePhotoMessage(uid: photo.uid, status: .failed)
        }
                    }
        }
            func sendOnlineImageMessage(photoMessage: PhotoModel, imageURL: String) {
        let message = ["image": imageURL, "uid": photoMessage.uid, "date":
photoMessage.date.timeIntervalSinceReferenceDate, "senderId": photoMessage.senderId, "status": "success","type":
PhotoModel.chatItemType] as [String : Any]
        let childUpdates = ["User-messages/\(photoMessage.senderId)/\(self.userUID)/\(photoMessage.uid)":
message,/*updates mynode*/"User-messages/\(self.userUID)/\(photoMessage.senderId)/\(photoMessage.uid)":
message/*updates his message node*/,"Users/\(Me.uid)/Contacts/\(self.userUID)/lastMessage":
message,"Users/\(self.userUID)/Contacts/\(Me.uid)/lastMessage": message]
        Database.database().reference().updateChildValues(childUpdates) { [weak self] (error, _) in//updates the message to
our database
                if error != nil {
        self?.dataSource.updatePhotoMessage(uid: photoMessage.uid, status: .failed)
                    return
                }
                self?.dataSource.updatePhotoMessage(uid: photoMessage.uid, status: .success)
            }
        }
          deinit {
            print(" Chatlog Deinitialized")//απενεγοποίηση,κλεισιμο αρχειου και απελευθερωση μνημης,θα πρεπει να υπαρχει
σε καθε view controller
          }
            /*override func didReceiveMemoryWarning() {
            super.didReceiveMemoryWarning()
            // Dispose of any resources that can be recreated.
          }*/
        }
        extension ChatLogController {//delagate function
            func array(_ array: FUICollection, didAdd object: Any, at index: UInt) {
                let message = JSON((object as! DataSnapshot).value as Any)
                let senderId = message["senderId"].stringValue
                let type = message["type"].stringValue
                let contains = self.dataSource.controller.items.contains { (collectionViewMessage) -> Bool in
                    return collectionViewMessage.uid == message["uid"].stringValue
            }
                if contains == false {
                let model = MessageModel(uid: message["uid"].stringValue, senderId: senderId, type: type, isIncoming: senderId ==
Me.uid ? false : true, date: Date(timeIntervalSinceReferenceDate: message["date"].doubleValue), status: message["status"]
== "success" ?  MessageStatus.success : MessageStatus.sending)
                    if type == TextModel.chatItemType {
                        let textMessage = TextModel(messageModel: model, text: message["text"].stringValue)
                        self.dataSource.addMessage(message: textMessage)
```

```
            } else if type == PhotoModel.chatItemType {
                KingfisherManager.shared.retrieveImage(with: URL(string: message["image"].stringValue)!, options: nil,
progressBlock: nil, completionHandler: { [weak self] (image, error, _, _) in
                    if error != nil {
                    self?.alert(message: "error receiving image from user")
                    } else {
                    let photoMessage = PhotoModel(messageModel: model, imageSize: image!.size, image: image!)
                    self?.dataSource.addMessage(message: photoMessage)
                    }
                })
    }} }}
```

```
    //
    //  ChatItemController.swift
    //  MyChatApp
    //
    //  Created by Alexandros on 13/1/18.
    //  Copyright © 2018 Alexandros. All rights reserved.
    //

    import Foundation
    import Chatto
    import ChattoAdditions
    import FirebaseDatabase
    import SwiftyJSON

    class ChatItemsController: NSObject {
    var items = [ChatItemProtocol]()//controls at DataSource file var chatItems
        var initialMessages = [ChatItemProtocol]()
        var loadMore = false
        var userUID : String!
        typealias completeLoading = () -> Void
    func loadIntoItemsArray(messagesNeeded:Int,moreToLoad: Bool) {
    for index in stride(from: initialMessages.count - items.count, to: initialMessages.count - items.count - messagesNeeded,
by: -1){
                self.items.insert(initialMessages[index - 1], at: 0)
            self.loadMore = moreToLoad
          }
        }
        func insertItem(message: ChatItemProtocol){
    self.items.append(message)
        }
        func loadPrevious(Completion: @escaping completeLoading){
          Database.database().reference().child("User-messages").child(Me.uid).child(userUID).queryEnding(atValue: nil,
childKey: self.items.first?.uid).queryLimited(toLast: 52).observeSingleEvent(of: .value, with: { [weak self] (snapshot) in
            var messages = Array(JSON(snapshot.value as Any).dictionaryValue.values).sorted(by: { (lhs, rhs) -> Bool in
                return lhs["date"].doubleValue < rhs["date"].doubleValue
            })
            messages.removeLast()
            self?.loadMore = messages.count > 50
            let converted = self!.convertToChatItemProtocol(messages: messages)
            for index in stride(from: converted.count, to: converted.count - min(messages.count, 50), by: -1) {
            self?.items.insert(converted[index - 1], at: 0)
```

```
                }
                Completion()
                messages.filter({ (message) -> Bool in
                    return message["type"].stringValue == PhotoModel.chatItemType
                }).forEach({ (message) in
                    self?.parseURLs(UID_URL: (key: message["uid"].stringValue, value: message["image"].stringValue))
                })
            })
        }
            func adjustWindow(){
                self.items.removeFirst(200)
                self.loadMore = true//it must be always true to paginate
            }}
```

```
//
//  DataSource.swift
//  MyChatApp
//
//  Created by Alexandros on 13/1/18.
//  Copyright © 2018 Alexandros. All rights reserved.
//

import Foundation
import Chatto
import ChattoAdditions
class DataSource : ChatDataSourceProtocol{
    weak var delegate: ChatDataSourceDelegateProtocol?
        var controller = ChatItemsController()
    var currentlyLoading = false
init(initialMessages:[ChatItemProtocol], uid: String) {
        self.controller.initialMessages = initialMessages
        self.controller.userUID = uid
        self.controller.loadIntoItemsArray(messagesNeeded: min(initialMessages.count, 50),moreToLoad: initialMessages.count
> 50)
        NotificationCenter.default.addObserver(self, selector: #selector(updateLoadingPhoto), name:
NSNotification.Name(rawValue: "updateImage"), object: nil)
        }
        var chatItems: [ChatItemProtocol]  {//the messages that show up in the collection view
return controller.items
    }
var hasMoreNext: Bool{
return false
    }
    var hasMorePrevious: Bool{
return controller.loadMore
}
    func loadNext() {
    }
    func loadPrevious() {
        if currentlyLoading == false {
            currentlyLoading = true
        controller.loadPrevious(){//it goes to ChatItemController  loadPrevious first and after here for the delagate
 self.delegate?.chatDataSourceDidUpdate(self, updateType: .pagination)
```

```
            self.currentlyLoading = false
        }   }}
     func addMessage(message: ChatItemProtocol){
        self.controller.insertItem(message: message)
        self.delegate?.chatDataSourceDidUpdate(self)
}
func updateTextMessage(uid: String, status: MessageStatus){
        if let index = self.controller.items.index(where: { (message) -> Bool in
            return message.uid == uid
        }) {
            let message = self.controller.items[index] as! TextModel
            message.status = status
            self.delegate?.chatDataSourceDidUpdate(self)
        }}
    func updatePhotoMessage(uid: String, status: MessageStatus){
        if let index = self.controller.items.index(where: { (message) -> Bool in
            return message.uid == uid
        }) {
            let message = self.controller.items[index] as! PhotoModel
            message.status = status
            self.delegate?.chatDataSourceDidUpdate(self)
        }}
    @objc func updateLoadingPhoto(notification: Notification) {
        let info = notification.userInfo as! [String: Any]
        let image = info["image"] as! UIImage
        let uid = info["uid"] as! String
        if let index = self.controller.items.index(where: { (message) -> Bool in
            return message.uid == uid
        }) {
        let item = self.controller.items[index] as! PhotoModel
            let model = MessageModel(uid: item.uid, senderId: item.senderId, type: item.type, isIncoming: item.isIncoming, date:
item.date, status: item.status)
            let photoMessage = PhotoModel(messageModel: model, imageSize: image.size, image: image)
            self.controller.items[index] = photoMessage
            self.delegate?.chatDataSourceDidUpdate(self)
}}
func adjustNumberOfMessages(preferredMaxCount: Int?, focusPosition: Double, completion: (Bool) -> Void) {
        if focusPosition > 0.9{
        self.controller.adjustWindow()//auto pagination
        completion(true)
        }
        else {
            completion(false)
        }}
    deinit {
        NotificationCenter.default.removeObserver(self, name: NSNotification.Name(rawValue: "updateImage"), object: nil)
    }}
```

```
//
//  Decorator.swift
//  MyChatApp
//
//  Created by Alexandros on 13/1/18.
//  Copyright © 2018 Alexandros. All rights reserved.
//
```

```
import Foundation
import Chatto
import ChattoAdditions
class Decorator: ChatItemsDecoratorProtocol {
   func decorateItems(_ chatItems: [ChatItemProtocol]) -> [DecoratedChatItem] {
      var decoratedItems = [DecoratedChatItem]()
            for (index, item) in chatItems.enumerated() {
      let nextMessage: ChatItemProtocol? = (index + 1 < chatItems.count) ? chatItems[index + 1] : nil
      let bottomMargin = separationAfterItem(current: item, next: nextMessage)
      let decoratedItem = DecoratedChatItem (chatItem: item, decorationAttributes:
ChatItemDecorationAttributes(bottomMargin: bottomMargin, showsTail: false, canShowAvatar: false))
      decoratedItems.append(decoratedItem)
      }
return decoratedItems
}
func separationAfterItem(current: ChatItemProtocol?,next: ChatItemProtocol?) -> CGFloat {
guard let next = next else { return 0 }
      let currentMessage = current as? MessageModelProtocol
      let nextMessage = next as? MessageModelProtocol
      if currentMessage?.senderId != nextMessage?.senderId {
         return 10
      } else {
return 3
}  }}
```

```
//
//  Helpers.swift
//  MyChatApp
//
//  Created by Alexandros on 23/1/18.
//  Copyright © 2018 Alexandros. All rights reserved.
//

import Foundation
import UIKit
import Chatto
import ChattoAdditions
import SwiftyJSON
import Kingfisher

extension UIViewController {
   func showingKeyboard(Notification : Notification) {
      if let kyeboardHeigth = (Notification.userInfo?[UIKeyboardFrameBeginUserInfoKey] as? NSValue)?.cgRectValue.height {
         self.view.frame.origin.y = -kyeboardHeigth
      } }
   func hidingKeyabord(){
   self.view.frame.origin.y = 0
     }
   func alert(message: String){
   let alertController = UIAlertController(title: "Boolevart Aid Alert", message: message, preferredStyle: .alert)
   let okAction = UIAlertAction(title: "OK", style: .default, handler: nil)
      alertController.addAction(okAction)
      self.present(alertController, animated: true, completion: nil)
}}
extension NSObject {
```

```
    func convertToChatItemProtocol(messages: [JSON]) -> [ChatItemProtocol] {
        var convertedMessages = [ChatItemProtocol]()
    convertedMessages = messages.map({ (message) -> ChatItemProtocol in//.map its the same with for loop
            let senderId = message["senderId"].stringValue
            let model = MessageModel(uid: message["uid"].stringValue, senderId: senderId, type: message["type"].stringValue,
isIncoming: senderId == Me.uid ? false : true, date: Date(timeIntervalSinceReferenceDate: message["date"].doubleValue),
status: message["status"] == "success" ?  MessageStatus.success : MessageStatus.sending)
            if message["type"].stringValue == TextModel.chatItemType {
                let textMessage = TextModel(messageModel: model, text: message["text"].stringValue)
                return textMessage
            } else {
let loading = #imageLiteral(resourceName: "loading")
                let photoMessage = PhotoModel(messageModel: model, imageSize: loading.size, image: loading)
                return photoMessage
            }})
      /*    for message in messages {
          let senderId = message["senderId"].stringValue
            let model = MessageModel(uid: message["uid"].stringValue, senderId: senderId, type: message["type"].stringValue,
isIncoming: senderId == Me.uid ? false : true, date: Date(timeIntervalSinceReferenceDate: message["date"].doubleValue),
status: message["status"] == "success" ?  MessageStatus.success : MessageStatus.sending)
            let textMessage = TextModel(messageModel: model, text: message["text"].stringValue)
            convertedMessages.append(textMessage)
            } */
return convertedMessages
}
func parseURLs(UID_URL: (key: String, value: String)) {
          let uid = UID_URL.key
          let imageURL = UID_URL.value
          KingfisherManager.shared.retrieveImage(with: URL(string: imageURL)!, options: nil, progressBlock: nil) { (image, error, _,
_) in
            if let image = image {
                NotificationCenter.default.post(name: NSNotification.Name(rawValue: "updateImage"), object: nil, userInfo:
["image": image,"uid": uid])
}}
          class deiniti {
            deinit {
                print("deinit helpers")}
}}}
```

```
//
//  Me.swift
//  MyChatApp
//
//  Created by Alexandros on 13/2/18.
//  Copyright © 2018 Alexandros. All rights reserved.
//

import Foundation
import FirebaseAuth
class Me {
    static var uid: String {
        return (Auth.auth().currentUser?.uid)!
}}
```

```
//
//  MessagesTableViewCell.swift
//  MyChatApp
//
//  Created by Alexandros on 28/1/18.
//  Copyright © 2018 Alexandros. All rights reserved.
//

import UIKit
class MessagesTableViewCell: UITableViewCell {
@IBOutlet weak var Name: UILabel!
   @IBOutlet weak var lastMessageDate: UILabel!
   @IBOutlet weak var lastMessage: UILabel!
     override func awakeFromNib() {
      super.awakeFromNib()
      // Initialization code
   }
override func setSelected(_ selected: Bool, animated: Bool) {
      super.setSelected(selected, animated: animated)
      // Configure the view for the selected state
   }
   deinit {
      print("deinit TableViewcell")
   }}
```

```
//
//  MessagesTableViewController.swift
//  MyChatApp
//
//  Created by Alexandros on 28/1/18.
//  Copyright © 2018 Alexandros. All rights reserved.
//

import UIKit
import FirebaseAuth
import SwiftyJSON
import FirebaseDatabase
import FirebaseDatabaseUI//Συνδέει το table view με την database
import Chatto
class MessagesTableViewController: UIViewController, FUICollectionDelegate, UITableViewDelegate, UITableViewDataSource
{
let Contacts = FUISortedArray(query:
Database.database().reference().child("Users").child(Me.uid).child("Contacts"),delegate: nil)/*sorted arry to sort contacts by
date sends message*/{ (lhs,rhs) -> ComparisonResult in
      let lhs = Date(timeIntervalSinceReferenceDate: JSON(lhs.value as Any)["lastMessage"]["date"].doubleValue)/*left hand
size*/
      let rhs =  Date(timeIntervalSinceReferenceDate: JSON(rhs.value as Any)["lastMessage"]["date"].doubleValue)/*right
hand size*/
      return rhs.compare(lhs)//table sorts by the newest message
   }
     @IBOutlet weak var tableView: UITableView!
   override func viewDidLoad() {
      super.viewDidLoad()
      self.Contacts.observeQuery()
      self.Contacts.delegate = self//calls when observeQuery,func didadd,didmove etc
```

```
        self.tableView.delegate = self
        self.tableView.dataSource = self//καλουμε αυτην και την απο πανω ωστε να μπορουμε να τις αλλαζουμε
        Database.database().reference().child("User-messages").child(Me.uid).keepSynced(true)
        // Do any additional setup after loading the view.
    }
override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }

    @IBAction func Add(_ sender: Any) {
        self.presentAlert()
    }
    @IBAction func SignOut(_ sender: Any) {
        try! Auth.auth().signOut()
        _ = self.navigationController?.popToRootViewController(animated: true)
}
    deinit {
        print("SignOut Deinitialized")
    }
    func presentAlert() {
        let alertController = UIAlertController(title: "Email?", message: "Please write the email:", preferredStyle: .alert)
        alertController.addTextField { (textField) in
            textField.placeholder = "email"
        }
    let cancelAction = UIAlertAction(title: "Cancel", style: .cancel, handler: nil)
        let confirmAction = UIAlertAction(title: "Confirm", style: .default) { [weak self] (_) in
            if let email = alertController.textFields?[0].text {
            self?.addContact(email: email)}
        }
          alertController.addAction(cancelAction)
            alertController.addAction(confirmAction)
        self.present(alertController, animated: true,completion: nil)
    }
    func addContact(email: String) {
        Database.database().reference().child("Users").observeSingleEvent(of: .value, with: { [weak self](snapshot) in
            let snapshot = JSON(snapshot.value as Any).dictionaryValue
            if let index = snapshot.index(where: { (key, value) -> Bool in
                return value["email"].stringValue == email
            }) {
                let allUpdates = ["/Users/\(Me.uid)/Contacts/\(snapshot[index].key)" : (["email":
snapshot[index].value["email"].stringValue, "name":
snapshot[index].value["name"].stringValue]),"/Users/\(snapshot[index].key)/Contacts/\(Me.uid)" : (["email":
Auth.auth().currentUser!.email!, "name": Auth.auth().currentUser!.displayName!])]
                Database.database().reference().updateChildValues(allUpdates)
                self?.alert(message: "Success adding Contact")
            } else {
            self?.alert(message: "No such email")
            }
        })
    }}
extension MessagesTableViewController {
    func array(_ array: FUICollection, didAdd object: Any, at index: UInt) {
        self.tableView.insertRows(at: [IndexPath(row: Int(index), section: 0)], with: .automatic)
    }
```

```
func array(_ array: FUICollection, didMove object: Any, from fromIndex: UInt, to toIndex: UInt) {
      self.tableView.insertRows(at: [IndexPath(row: Int(toIndex), section: 0)], with: .automatic)
      self.tableView.deleteRows(at: [IndexPath(row: Int(fromIndex), section: 0)], with: .automatic)
   }
      func array(_ array: FUICollection, didRemove object: Any, at index: UInt) {
      self.tableView.deleteRows(at: [IndexPath(row: Int(index), section: 0)], with: .automatic)
   }
func array(_ array: FUICollection, didChange object: Any, at index: UInt) {
      self.tableView.reloadRows(at: [IndexPath(row: Int(index), section: 0)], with: .none)
}
func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
      return Int(self.Contacts.count)
   }
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
      let cell = tableView.dequeueReusableCell(withIdentifier: "cell", for: indexPath) as! MessagesTableViewCell
      let info = JSON((Contacts[(UInt(indexPath.row))] as? DataSnapshot)?.value as Any).dictionaryValue
      cell.Name.text = info["name"]?.stringValue
      cell.lastMessage.text = info["lastMessage"]?["text"].string
      cell.lastMessageDate.text = dateFormatter(timestamp: info["lastMessage"]?["date"].double)
      return cell
   }
func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {//Μολις επιλεξεις μια γραμμη στο table
ενεργοποιειται η κλαση αυτη,δινοντας μας το indexPath της γραμμης που εχουμε επιλεξει

      let uid = (Contacts[UInt(indexPath.row)] as? DataSnapshot)?.key//path of the uid
      let reference = Database.database().reference().child("User-messages").child(Me.uid).child(uid!).queryLimited(toLast:
51)
      self.tableView.isUserInteractionEnabled = false
reference.observeSingleEvent(of: .value, with: { [weak self] (snapshot) in
let messages = Array(JSON(snapshot.value as Any).dictionaryValue.values).sorted(by: { (lhs, rhs) -> Bool in
            return lhs["date"].doubleValue < rhs["date"].doubleValue
         })
print(messages)
         let converted = self!.convertToChatItemProtocol(messages: messages)
         let chatlog = ChatLogController()
         chatlog.userUID = uid!
         chatlog.dataSource = DataSource(initialMessages: converted,uid: uid!)
         chatlog.MessagesArray = FUIArray(query: Database.database().reference().child("User-
messages").child(Me.uid).child(uid!).queryStarting(atValue: nil, childKey: converted.last?.uid), delegate: nil)//observes new
messages
         self?.navigationController?.show(chatlog, sender: nil)//shows ChatLogController
         self?.tableView.deselectRow(at: indexPath, animated: true)
         self?.tableView.isUserInteractionEnabled = true
         messages.filter({ (message) -> Bool in
            return message["type"].stringValue == PhotoModel.chatItemType
         }).forEach({ (message) in
            self?.parseURLs(UID_URL: (key: message["uid"].stringValue, value: message["image"].stringValue))
         })})}
   func dateFormatter(timestamp: Double?) -> String? {//Ποση ωρα/μερα περασε απο το τελευταιο μηνυμα
if let timestamp = timestamp {
         let date = Date(timeIntervalSinceReferenceDate: timestamp)
         let dateFormatter = DateFormatter()
         let timeSinceDateInSeconds = Date().timeIntervalSince(date)
         let secondsInDays: TimeInterval = 24*60*60
         if timeSinceDateInSeconds > 7 * secondsInDays {
```

```
        dateFormatter.dateFormat = "dd/MM/yy"
}else if timeSinceDateInSeconds > secondsInDays {
        dateFormatter.dateFormat = "EEE"
        }else {
        dateFormatter.dateFormat = "h:mm a"
}
        return dateFormatter.string(from: date)
}else {
      return nil
      }}}
```

```
//
//  PhotoBuilder.swift
//  MyChatApp
//
//  Created by Alexandros on 16/1/18.
//  Copyright © 2018 Alexandros. All rights reserved.
//

import Foundation
import Chatto
import ChattoAdditions
class photoViewModel: PhotoMessageViewModel<PhotoModel>{
 override init(photoMessage: PhotoModel, messageViewModel: MessageViewModelProtocol) {
      super.init(photoMessage: photoMessage, messageViewModel: messageViewModel)
  }}
class PhotoBuilder: ViewModelBuilderProtocol{
  let defaultBuilder = MessageViewModelDefaultBuilder()
  func canCreateViewModel(fromModel decoratedPhotoMessage: Any) -> Bool {
     return decoratedPhotoMessage is PhotoModel
  }
  func createViewModel(_ decoratedPhotoMessage: PhotoModel) -> photoViewModel {
  let photoMessageViewModel = photoViewModel(photoMessage: decoratedPhotoMessage, messageViewModel:
defaultBuilder.createMessageViewModel(decoratedPhotoMessage))
    return photoMessageViewModel}
}
```

```
//
//  PhotoHandler.swift
//  MyChatApp
//
//  Created by Alexandros on 16/1/18.
//  Copyright © 2018 Alexandros. All rights reserved.
//
import Foundation
import Chatto
import ChattoAdditions

class PhotoHandler: BaseMessageInteractionHandlerProtocol{
func userDidTapOnFailIcon(viewModel: photoViewModel, failIconView: UIView) {
      print("tap on fail")
  }
  func userDidTapOnAvatar(viewModel: photoViewModel) {
     print("tap on avatar")
```

```
}
   func userDidTapOnBubble(viewModel: photoViewModel) {
      //print(photoViewModel)
      print("tap on bubble")
}
   func userDidBeginLongPressOnBubble(viewModel: photoViewModel) {
      print("beeing long press")
}
   func userDidEndLongPressOnBubble(viewModel: photoViewModel) {
      print("end long press")
}}
```

```
//
//  PhotoModel.swift
//  MyChatApp
//
//  Created by Alexandros on 16/1/18.
//  Copyright © 2018 Alexandros. All rights reserved.
//

import Foundation
import Chatto
import ChattoAdditions
class PhotoModel: PhotoMessageModel<MessageModel> {
   static let chatItemType = "photo"
override init(messageModel: MessageModel, imageSize: CGSize, image: UIImage) {
      super.init(messageModel: messageModel, imageSize: imageSize, image: image)
   }
    var status: MessageStatus {
get {
return self._messageModel.status
} set {
self._messageModel.status = newValue
     }}}
```

```
//
//  SignInViewController.swift
//  MyChatApp
//
//  Created by Alexandros on 22/1/18.
//  Copyright © 2018 Alexandros. All rights reserved.
//

import UIKit
import FirebaseAuth
import FirebaseDatabase
class SignInViewController: UIViewController {

   @IBOutlet weak var password: UITextField!
   @IBOutlet weak var email: UITextField!
   override func viewDidLoad() {
      super.viewDidLoad()
          NotificationCenter.default.addObserver(self, selector: #selector(showingKeyboard), name:
NSNotification.Name(rawValue: "UiKeyboardWillShowNotification"), object: nil)
```

```
      NotificationCenter.default.addObserver(self, selector: #selector(hidingKeyabord), name: NSNotification.Name(rawValue:
"UiKeybordWillHideNotification"), object: nil)//hides keyboard οταν πατας σε αλλο σημειο της οθόνης
      // Do any additional setup after loading the view.
   }
      override func didReceiveMemoryWarning() {
      super.didReceiveMemoryWarning()
      // Dispose of any resources that can be recreated.
   }
   @IBAction func SignIn(_ sender: Any) {
      guard let email = email.text, let password = password.text else {return}
      Auth.auth().signIn(withEmail: email, password: password) { [weak self] (user, error) in
         if let error = error {
            self?.alert(message: error.localizedDescription)
            return
         }
         let table = self?.storyboard?.instantiateViewController(withIdentifier: "table") as! MessagesTableViewController
         self?.navigationController?.show(table, sender: nil)

         print("success SignIn")
      }
}
@IBAction func SignUp(_ sender: Any) {
      let controller = storyboard?.instantiateViewController(withIdentifier: "SIGNUP") as! SignUpViewController
      self.navigationController?.show(controller, sender: nil)
   }
   override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
      self.view.endEditing(true) }
deinit {
      print("deinit SignIn")
   }}
```

```
//
//  SignUpViewController.swift
//  MyChatApp
//
//  Created by Alexandros on 22/1/18.
//  Copyright © 2018 Alexandros. All rights reserved.
//

import UIKit
import FirebaseAuth
import FirebaseDatabase
class SignUpViewController: UIViewController {

   @IBOutlet weak var fullname: UITextField!
   @IBOutlet weak var password: UITextField!
   @IBOutlet weak var email: UITextField!
   override func viewDidLoad() {
      super.viewDidLoad()
      NotificationCenter.default.addObserver(self, selector: #selector(showingKeyboard), name:
NSNotification.Name(rawValue: "UiKeyboardWillShowNotification"), object: nil)
      NotificationCenter.default.addObserver(self, selector: #selector(hidingKeyabord), name: NSNotification.Name(rawValue:
"UiKeybordWillHideNotification"), object: nil)
      // Do any additional setup after loading the view.
   }
```

```
override func didReceiveMemoryWarning() {
      super.didReceiveMemoryWarning()
      // Dispose of any resources that can be recreated.
   }
   @IBAction func SignUp(_ sender: Any) {
      guard let email = email.text , let password = password.text , let fullname = fullname.text
         else {return}
      Auth.auth().createUser(withEmail: email, password: password) { [weak self] (user, error) in
         if let error = error {
            self?.alert(message: error.localizedDescription)
            return
         }
         Database.database().reference().child("Users").child(user!.uid).updateChildValues(["email": email, "name": fullname])
         let changeRequest = user?.createProfileChangeRequest()
         changeRequest?.displayName = fullname
         changeRequest?.commitChanges(completion: nil)
         let table = self?.storyboard?.instantiateViewController(withIdentifier: "table") as! MessagesTableViewController
         self?.navigationController?.show(table, sender: nil)
         print("Success SignUp")
      } }
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
      self.view.endEditing(true)
   }
deinit {
      print("deinit SignOut")
   }}
```

```
//
//  TextBuilder.swift
//  MyChatApp
//
//  Created by Alexandros on 15/1/18.
//  Copyright © 2018 Alexandros. All rights reserved.
//

import Foundation
import Chatto
import ChattoAdditions

class ViewModel: TextMessageViewModel<TextModel>{
override init (textMessage: TextModel,messageViewModel: MessageViewModelProtocol){
    super.init(textMessage: textMessage, messageViewModel: messageViewModel)
} }
class TextBuilder: ViewModelBuilderProtocol {
      let defaultBuilder = MessageViewModelDefaultBuilder()
   func canCreateViewModel(fromModel decoratedTextMessage: Any) -> Bool {
      return decoratedTextMessage is TextModel
   }
   func createViewModel(_ decoratedTextMessage: TextModel) -> ViewModel {
      let textMessageViewModel = ViewModel(textMessage: decoratedTextMessage, messageViewModel:
defaultBuilder.createMessageViewModel(decoratedTextMessage))
      return textMessageViewModel
   }}
```

```
//
//  TextHandler.swift
//  MyChatApp
//
//  Created by Alexandros on 15/1/18.
//  Copyright © 2018 Alexandros. All rights reserved.
//

import Foundation
import Chatto
import ChattoAdditions
class TextHandler: BaseMessageInteractionHandlerProtocol{

   func userDidTapOnFailIcon(viewModel: ViewModel, failIconView: UIView) {
      print("tap on fail")
   }
   func userDidTapOnAvatar(viewModel: ViewModel) {
      print("tap on avatar")
   }
   func userDidTapOnBubble(viewModel: ViewModel) {
      print(viewModel.text)
      print("tap on bubble")
}
   func userDidBeginLongPressOnBubble(viewModel: ViewModel) {
      print("beeing long press")
   }
   func userDidEndLongPressOnBubble(viewModel: ViewModel) {
      print("end long press")
}}
```

```
//
//  TextModel.swift
//  MyChatApp
//
//  Created by Alexandros on 13/1/18.
//  Copyright © 2018 Alexandros. All rights reserved.
//

import Foundation
import Chatto
import ChattoAdditions

class TextModel : TextMessageModel<MessageModel> {
   static let chatItemType = "text"
   override init(messageModel: MessageModel,text : String){
   super.init(messageModel : messageModel,text : text)
   }
var status: MessageStatus {
get {
return self._messageModel.status
} set {
self._messageModel.status = newValue
      }}}
```

Βιβλιογραφία

[1] Mobile operating system,Wikipedia,https://en.wikipedia.org/wiki/Mobile_operating_system [πρόσβαση 02/12/2018]

[2] What is mobile OS,https://searchmobilecomputing.techtarget.com/definition/mobile-operating-system [πρόσβαση 02/12/2018]

[3] Wikipedia,https://en.wikipedia.org/wiki/History_of_operating_systems [πρόσβαση 02/12/2018]

[4] PalmOS,https://en.wikipedia.org/wiki/Palm_OS [πρόσβαση 02/12/2018]

[5] iOS,https://en.wikipedia.org/wiki/IOS [πρόσβαση 02/12/2018]

[6] Android,https://en.wikipedia.org/wiki/Android_(operating_system) [πρόσβαση 02/12/2018]

[7] Wikibooks,https://en.wikibooks.org/wiki/Introduction_to_Computer_Information_Systems/System_Software - Systems_Software [πρόσβαση 02/12/2018]

[8] Mobile App,https://en.wikipedia.org/wiki/Mobile_app [πρόσβαση 02/12/2018]

[9] Wikibooks,https://en.wikibooks.org/wiki/Introduction_to_Computer_Information_Systems/System_Software [πρόσβαση 02/12/2018]

[10] Wikipedia,https://en.wikipedia.org/wiki/IOS - Home_screen [πρόσβαση 02/12/2018]

[11] Dr Rajiv Rammath,"Beginning iOS Programming for dummies", John Wiley & Sons Inc 2014

[12] Apple developer, https://developer.apple.com/library/content/documentation/MacOSX/Conceptual/OSX_Technology_Overview/About/About.html - //apple_ref/doc/uid/TP40001067-CH204-TPXREF101 [πρόσβαση 02/12/2018]

[13] Apple developer, https://developer.apple.com/library/documentation/OSX_Technology_Overview/About/About.html - [πρόσβαση 02/12/2018]

[14] Dr. Alex Blewitt," Swift Essentials", Packt Publishing 2014

[15] Wikipedia,https://en.wikipedia.org/wiki/IOS_SDK [πρόσβαση 02/12/2018]

[16] Quora,https://www.quora.com/What-is-Xcode-and-why-do-I-need-it [πρόσβαση 02/12/2018]

[17] Matt Neuburg, "*iOS 9 Programming Fundamentals with Swift Second Edition*", O'Reilly Media Inc. 2016

[18] Wikipedia,https://en.wikipedia.org/wiki/Xcode - Xcode_1.0_-_Xcode_2.x_(before_iOS_support) [πρόσβαση 02/12/2018]

[19] CocoaPods,https://guides.cocoapods.org/using/getting-started.html [πρόσβαση 02/12/2018]

[20] Jon Hoffman. ”*Mastering Swift 4*”,Packt Publishing 2017

[21] Jon Hoffman. ”*Mastering Swift 4*”,Packt Publishing 2017

[22] Wikipedia, https://en.wikipedia.org/wiki/Application_framework [πρόσβαση 02/12/2018]

[23] Matthew Knott,”*Beginning Xcode”*,Paul Manning 2014

[24] Paul Deitel,”*Swift for programmers*”,Pearson Education Inc. 2015

[25] Wikipedia,https://en.wikipedia.org/wiki/Cocoa_(API) [πρόσβαση 02/12/2018]

[26] Jan Tiano,”Learning Xcode 8”, Packt Publishing 2016

[27] Wallace Wang,”Swift OS X Programming for Absolute Beginners”,Apress Media 2015

[28] Jesse Feiler,”Swift for Dummies”,John Wiley & Sons Inc 2015

[29] Dr Rajiv Rammath,”Beginning iOS Programming for dummies”, John Wiley & Sons Inc 2014

[30] CocoaPods,https://guides.cocoapods.org/using/getting-started.html [πρόσβαση 02/12/2018]

[31] Firebase,https://firebase.google.com/docs/reference/swift/firebaseauth/api/reference/Classes/Auth [πρόσβαση 02/12/2018]

[32] Firebase,https://firebase.google.com/docs/reference/swift/firebasedatabase/api/reference/Classes [πρόσβαση 02/12/2018]

[33] Firebase,https://firebase.google.com/docs/reference/swift/firebasestorage/api/reference/Classes [πρόσβαση 02/12/2018]