



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

Διαδικτυακή εφαρμογή για τον δυναμικό συνεπιβατισμό μεταφορικών μέσων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΤΟΥ

ΓΙΑΚΟΥΜΑΚΗ ΕΥΣΤΡΑΤΙΟΥ

Επιβλέπων: Παναγιώτης Τσανάκας
Καθηγητής ΕΜΠ

Αθήνα, Οκτώβριος 2024



Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Διαδικτυακή εφαρμογή για τον δυναμικό συνεπιβατισμό μεταφορικών μέσων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΓΙΑΚΟΥΜΑΚΗ ΕΥΣΤΡΑΤΙΟΥ

Επιβλέπων: Παναγιώτης Τσανάκας
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 31η Οκτωβρίου 2024.

(Υπογραφή)

(Υπογραφή)

(Υπογραφή)

.....
Παναγιώτης Τσανάκας
Καθηγητής ΕΜΠ

.....
Βασίλειος Βεσκούκης
Καθηγητής ΕΜΠ

.....
Γεώργιος Αλεξανδρίδης
Επίκουρος Καθηγητής ΕΚΠΑ

Αθήνα, Οκτώβριος 2024



Copyright © – All rights reserved. Με την επιφύλαξη παντός δικαιώματος.
Ευστράτιος Γιακουμάκης, 2024.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα.

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

ΔΗΛΩΣΗ ΜΗ ΛΟΓΟΚΛΟΠΗΣ ΚΑΙ ΑΝΑΛΗΨΗΣ ΠΡΟΣΩΠΙΚΗΣ ΕΥΘΥΝΗΣ

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

(Υπογραφή)

.....

Ευστράτιος Γιακουμάκης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

31 Οκτωβρίου 2024

Περίληψη

Εδώ και πολλά χρόνια έχει παρατηρηθεί το φαινόμενο οι φοιτητές που προσέρχονται στην Πολυτεχνειούπολη του Εθνικού Μετσοβίου Πολυτεχνείου μέσω των συγκοινωνιών να συγκεντρώνονται σε ορισμένες κομβικές στάσεις, οδηγώντας σε συνωστισμό και διαταράσσοντας τη γύρω κυκλοφορία. Αρκετοί ωστόσο φοιτητές χρησιμοποιούν το ιδιωτικό τους όχημα για τη μετακίνησή του στο Πολυτεχνείο, και εφόσον πορεύονται κατά μήκος των συγκεκριμένων οδών εμφανίζεται η ευκαιρία επιβίβασης στο ΙΧ κάποιων εκ των φοιτητών που περιμένουν στη στάση, ελαφρύνοντας έτσι τη συμφόρηση στον χώρο. Στην εργασία αυτή θα παρουσιάσουμε μία κινητή εφαρμογή η οποία καθιστά δυνατή την επικοινωνία των φοιτητών που βρίσκονται στο όχημά τους με τους φοιτητές που βρίσκονται στη στάση, αναλαμβάνοντας το ταίριασμα οδηγών και πεζών, ενθαρρύνοντας τον συνεπιβατισμό, και συμβάλλοντας έτσι στην επίλυση του προβλήματος. Για την ανάπτυξη της εφαρμογής χρησιμοποιήσαμε ένα tech stack που περιλαμβάνει το framework του Flutter στο frontend, και τον συνδυασμό ενός server Node.js με μία βάση δεδομένων MySQL στο backend, με την επικοινωνία των δύο μερών να συντονίζεται μέσω του πρωτοκόλλου WebSocket. Η εφαρμογή έχει δοκιμαστεί ενδελεχώς σε κλειστό περιβάλλον, και έχοντας διαπιστωθεί ότι λειτουργεί ικανοποιητικά, είναι πλέον έτοιμη για πιλοτική δοκιμή σε πραγματικές συνθήκες με μετέπειτα deployment στις υπηρεσίες του App Store και Google Play.

Λέξεις-κλειδιά — carpooling, ride-hailing, mobile app, Flutter, Node.js

Abstract

One of the problems that the National Technical University of Athens currently faces concerns the severe crowding observed at specific bus stops that students use during their commute to the University, a phenomenon that disrupts traffic and puts students' lives at risk. At the same time, a large portion of students commutes to the University using their own personal vehicle, and therefore an opportunity presents itself whereby student-drivers can make a brief stop near the bus stop and let students board their vehicle, lessening the amount of crowding in the area. In the present paper we will be showcasing a mobile app we developed in order to automatically pair drivers with their potential passengers and facilitate communication between them, with the aim of encouraging the aforementioned ridesharing, and providing a solution to the crowding issue. The app was developed using the Flutter framework for the frontend, and a combination of the Node.js runtime and a MySQL database for the backend, with communication between the two ends made possible using the WebSocket protocol. After thorough closed testing, the app has been deemed fully functional and ready for deployment to the App Store and Play Store in order to perform further open beta testing and eventually release to production.

Keywords— carpooling, ride-hailing, mobile app, Flutter, Node.js

Ευχαριστίες

Ευχαριστώ τον καθηγητή και κοσμήτορα της σχολής κ. Παναγιώτη Τσανάκα για την ευκαιρία που μου έδωσε μέσω αυτής της διπλωματικής, καθώς και για τη καθοδήγηση που μου προσέφερε καθ' όλη τη διάρκεια ανάπτυξης της εφαρμογής. Επίσης ευχαριστώ τον Κωνσταντίνο Αθανασιάδη για την τεχνική υποστήριξη και διαχείριση λοιπών θεμάτων και τον Άγγελο Κολαίτη για την διαχείριση του server και της υπηρεσίας ταυτοποίησης. Τέλος, ευχαριστώ την οικογένειά μου και τους φίλους μου, που ήταν πάντα στο πλευρό μου.

Περιεχόμενα

1	Εισαγωγή	5
1.1	Το πρόβλημα	5
1.2	Συνεπιβατισμός	6
1.3	Η λειτουργία της εφαρμογής	7
2	Flutter	11
2.1	Cross-platform	11
2.2	Αρχιτεκτονική	12
2.3	Widgets	13
2.4	UI architecture	14
2.5	State Management	16
3	Node.js	20
3.1	JavaScript	20
3.2	Χαρακτηριστικά	21
3.3	TypeScript	23
3.4	Node.js	24
4	WebSocket	28
5	Frontend	32
5.1	Λειτουργίες	32
5.2	Παρουσίαση κώδικα	37
5.2.1	Αρχική οθόνη	38
5.2.2	Λειτουργία πεζού	42
5.2.3	Λειτουργία οδηγού	48
5.2.4	Σύνδεση WebSocket	56

6 Backend	59
6.1 Παρουσίαση κώδικα	64
7 Συμπεράσματα	73
A' Εντοπισμός ακατάλληλου περιεχομένου μέσω μηχανικής όρασης	75
B' Authentication	77
B'.1 OAuth και OpenID Connect	77
B'.2 Browser authentication	78
B'.3 Native app authentication	79
Γ' Βάση Δεδομένων	83
Δ' Ιστοσελίδες	86

Κεφάλαιο 1

Εισαγωγή

1.1 Το πρόβλημα

Το πρόβλημα το οποίο καλούμαστε να λύσουμε αφορά τον συνωστισμό που παρατηρείται συχνά στη στάση λεωφορείου του *Στρατιωτικού Νοσοκομείου* που βρίσκεται στη λεωφόρο Κατεχάκης, κοντά στη σχολή του Ε.Μ.Π. Ένα σημαντικό ποσοστό των φοιτητών προσέρχεται καθημερινά στη σχολή χρησιμοποιώντας τα μέσα μαζικής μεταφοράς, και συγκεκριμένα το λεωφορείο 242 που μεταφέρει άτομα από τη στάση απευθείας στη σχολή. Η στάση κατέχει κομβική θέση λόγω του μεγάλου αριθμού φοιτητών που τη χρησιμοποιούν, ωστόσο το γεγονός αυτό έχει ως αποτέλεσμα την αδυναμία επαρκούς κάλυψης των αναγκών των επιβατών από τα δρομολόγια των λεωφορείων, και επομένως τη συγκέντρωση πλήθους στην περιοχή.

Η συγκέντρωση πλήθους με τη σειρά της σημαίνει ότι οι φοιτητές αναγκάζονται να σταθούν εκτός του πεζοδρομίου και εντός της λεωφόρου, διαταράσσοντας την κυκλοφορία και θέτοντας την ασφάλειά τους σε κίνδυνο. Εκτός αυτού, πολλές φορές ο αριθμός φοιτητών είναι αρκετά μεγάλος ώστε να υπερβαίνει τη χωρητικότητα του κάθε λεωφορείου που φτάνει στη στάση, με αποτέλεσμα οι επιβαίνοντες στο λεωφορείο να συνωστίζονται σε επικίνδυνο βαθμό, και οι φοιτητές που αδυνατούν να επιβιβαστούν να χρειάζεται να αναμένουν το επόμενο λεωφορείο. Η κατάσταση είναι χειρότερη κατά τις ώρες αιχμής, ενώ δυσχεραίνεται περαιτέρω όταν παρατηρείται κυκλοφοριακή συμφόρηση ή καθυστερήσεις στα δρομολόγια, αλλά και κατά τη διάρκεια της εξεταστικής περιόδου όπου η προσέλευση των φοιτητών είναι αυξημένη.

Ταυτόχρονα με τους φοιτητές που μεταβαίνουν στη σχολή μέσω των ΜΜΜ,

ένας αριθμός φοιτητών χρησιμοποιεί οχήματα ΙΧ για τη μεταφορά του. Έχει παρατηρηθεί επομένως το φαινόμενο οι φοιτητές αυτοί να σταματάνε το όχημά τους στη στάση για να επιτρέψουν σε συμφοιτητές τους να εισέλθουν στο όχημα και να μεταβούν μαζί στη σχολή, μειώνοντας κατά ένα μικρό βαθμό τον συνωστισμό.

Η εφαρμογή που αναπτύξαμε έχει στόχο την αξιοποίηση αυτής ακριβώς της συμπεριφοράς προκειμένου να καταπολεμηθεί πιο στοχευμένα και αποτελεσματικά το παραπάνω πρόβλημα. Συγκεκριμένα η εφαρμογή συντονίζει το ταίριασμα των εν κινήσει οδηγών με τους πεζούς που αναμένουν στη στάση, προσφέροντας με αυτόν τον τρόπο ένα πιο οργανωμένο πλαίσιο στο οποίο μπορεί να συμβεί ο συνεπιβατισμός ο οποίος έχει αναπτυχθεί οργανικά στις δεδομένες συνθήκες. Παράλληλα, η εφαρμογή συμβάλλει στην ανάπτυξη διαπροσωπικών σχέσεων μεταξύ των φοιτητών, καθώς φέρνει σε επαφή άτομα τα οποία δεν γνωρίζονται απαραίτητα μεταξύ τους, ενισχύοντας έτσι το κλίμα αλληλεγγύης και συναδελφικότητας στη σχολή.

1.2 Συνεπιβατισμός

Ο όρος *συνεπιβατισμός* (carpooling στα Αγγλικά) αναφέρεται στη χρήση ενός οχήματος ΙΧ από πολλά άτομα για την άφιξη σε έναν κοινό προορισμό. Στην πιο συνηθισμένη περίπτωση, ο συνεπιβατισμός περιλαμβάνει την εκ των προτέρων συνεννόηση μεταξύ του επιβάτη και του οδηγού πριν την έναρξη της διαδρομής, προκειμένου να αποφασιστεί ο ακριβής προορισμός, η ώρα συνάντησης, και το σημείο επιβίβασης. Επομένως, αυτό το είδος συνεπιβατισμού εν γένει χαρακτηρίζεται από μειωμένη ευελιξία, και περιορίζεται κυρίως σε φίλους, ή συνεργάτες.

Μία άλλη μορφή συνεπιβατισμού αναπτύχθηκε στη Washington DC τη δεκαετία του 1970 και έπειτα εξαπλώθηκε και σε άλλες πόλεις της Αμερικής και του κόσμου, το λεγόμενο casual carpooling ή “slugging”.

Σε ορισμένους αυτοκινητοδρόμους των πόλεων αυτών τα οχήματα ΙΧ με περισσότερους από έναν επιβάτη είχαν πρόσβαση σε λωρίδες ταχείας κυκλοφορίας και μειωμένα τέλη διοδίων, γεγονός που έδινε κίνητρο σε οδηγούς να δέχονται πεζούς στο όχημά τους και να τους μεταφέρουν στον προορισμό τους[1]. Καθώς η δραστηριότητα αυτή ωφελούσε αμφότερους πεζούς και οδηγούς, μειώνοντας εκτός των άλλων το χρόνο μετακίνησης και για τους δύο,

το casual carpooling άνθισε στις πόλεις αυτές, με τη γρήγορη καθιέρωση ημιεπίσημων “στάσεων” όπου οι πεζοί ακόμη και σήμερα μπορούν να αναμένουν τους διερχόμενους οδηγούς. Αυτό το είδος συνεπιβατισμού δίνει τη δυνατότητα σε άτομα άγνωστα μεταξύ τους να μοιραστούν το όχημα άμεσα και χωρίς την ανάγκη προηγούμενης συνεννόησης, ωστόσο είναι περιορισμένο ως προς τα σημεία συνάντησης, καθώς αυτά είναι λιγοστά και καθορισμένα.

Με την έλευση των κινητών εφαρμογών η δημοφιλία του συνεπιβατισμού αυξήθηκε ραγδαία, μέσω των λεγόμενων ride-hailing apps.

Μέσω των εφαρμογών αυτών οι χρήστες έχουν πλέον τη δυνατότητα να αναζητήσουν επί τόπου οδηγό, οπουδήποτε και αν βρίσκονται. Οι εφαρμογές παρουσιάζουν όλους τους κοντινούς διαθέσιμους οδηγούς στον χρήστη, ο οποίος έχει την ελευθερία να επιλέξει όχι μόνο τον οδηγό που θα τον παραλάβει, αλλά και τον ακριβή προορισμό του, αίροντας πρακτικά όλους τους περιορισμούς που αναφέρθηκαν προηγουμένως. Ωστόσο, οι ανέσεις αυτές που προσφέρονται στον πεζό επιβάλλουν συχνά τη χρέωση της υπηρεσίας, η οποία ουσιαστικά μετατρέπεται σε υπηρεσία ταξί. Πράγματι, σχεδόν όλες οι διαθέσιμες εφαρμογές ride-hailing είναι εμπορικές. Η πρώτη εμπορική εταιρεία που εισήλθε στον τομέα των ride-hailing εφαρμογών ήταν η Uber, η οποία το 2010 δημιούργησε την επώνυμη εφαρμογή της, και η οποία διατηρεί μέχρι και σήμερα το μεγαλύτερο μερίδιο της αγοράς σε παγκόσμιο επίπεδο.

1.3 Η λειτουργία της εφαρμογής

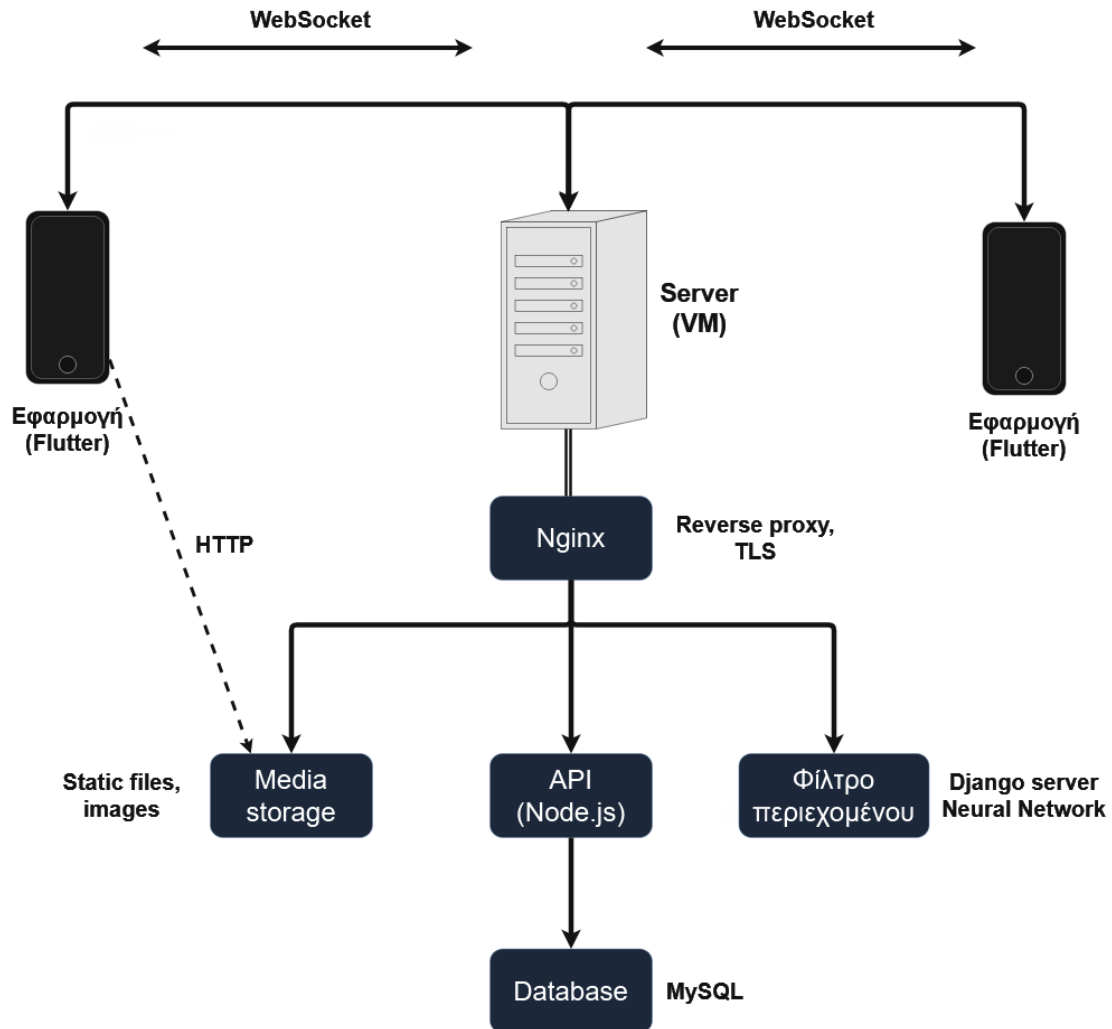
Η εφαρμογή που αναπτύξαμε πλησιάζει περισσότερο στη φιλοσοφία της το casual carpooling, αλλά ενσωματώνει και χαρακτηριστικά από τις εφαρμογές ride-hailing.

Το σημείο συνάντησης καθώς και ο τελικός προορισμός είναι καθορισμένα: ο πεζός πρέπει να μεταβεί στη στάση του Στρατιωτικού Νοσοκομείου για να μπορέσει να αναζητήσει οδηγούς μέσω της εφαρμογής, ενώ ο προορισμός είναι προφανώς η σχολή του πολυτεχνείου. Μόλις ένας οδηγός ενεργοποιήσει την εφαρμογή και εισέλθει εντός μίας ακτίνας μερικών χιλιομέτρων από τη στάση, ενημερώνεται για την ύπαρξη πεζών που περιμένουν, και δηλώνει την πρόθεση του να τους δεχτεί στο όχημά του. Οι πεζοί με τη σειρά τους ενημερώνονται άμεσα μόλις κάποιος οδηγός δεχτεί να τους παραλάβει, και “κλειδώνουν” τη θέση τους στο όχημα. Το μήνυμα για τον διαθέσιμο οδηγό στέλνεται ταυτό-

χρονα σε πολλαπλούς πεζούς, και η αντίστοιχη θέση παραδίδεται σε εκείνον που απάντησε πιο γρήγορα στην ειδοποίηση.

Ύστερα από την επιτυχή αντιστοίχιση πεζού και οδηγού, ο κάθε χρήστης παρουσιάζεται με τα στοιχεία του άλλου ώστε να είναι δυνατή η αναγνώριση κατά τη συνάντηση (όνομα, φωτογραφίες, αριθμός κυκλοφορίας), ενώ παράλληλα παρουσιάζεται το στίγμα του στο χάρτη για τον προσδιορισμό της θέσης και του χρόνου άφιξης. Μόλις ο οδηγός παραλάβει τους πεζούς από τη στάση κατευθύνονται μαζί προς τη σχολή, και κατά την άφιξή τους σε αυτή, η εφαρμογή τερματίζει αυτόματα τη συνεδρία. Τέλος, παρέχεται η ευκαιρία αξιολόγησης τόσο από πλευράς του οδηγού όσο και από πλευράς του επιβάτη.

Όλος ο κώδικας για την εφαρμογή είναι διαθέσιμος στο GitHub. Ο κώδικας της εφαρμογής κινητού βρίσκεται στον σύνδεσμο [2], και ο κώδικας του server βρίσκεται στον σύνδεσμο [3].



Σχήμα 1.1: Σχηματικό της βασικής αρχιτεκτονικής του συνόλου της εφαρμογής. Η εφαρμογή κινητού του χρήστη αποτελεί το frontend μέρος. Το backend συνιστάται από όλες τις υπηρεσίες του server, οι οποίες περιλαμβάνουν το βασικό API με το οποίο επικοινωνεί το frontend, την βάση δεδομένων, το κομμάτι που χειρίζεται το ανέβασμα και τη λήψη αρχείων πολυμέσων, και την υπηρεσία εντοπισμού ακατάλληλου περιεχομένου. Η επικοινωνία με το backend γίνεται μέσω του πρωτοκόλλου WebSocket, ενώ διαμεσολαβεί ένας server Nginx ο οποίος δέχεται όλα τα αιτήματα και πραγματοποιεί το κατάλληλο routing στη κάθε υπηρεσία

Κεφάλαιο 2

Flutter¹

Το Flutter είναι ένα framework ανάπτυξης cross-platform εφαρμογών για Android, iOS και Web που δημιουργήθηκε και αναπτύσσεται από την Google. Η Google παρουσίασε για πρώτη φορά το 2015 μία πειραματική έκδοση του Flutter υπό την ονομασία “Sky”, αναφέροντας ως πρωταρχικό στόχο του νέου αυτού framework την εύκολη και γρήγορη ανάπτυξη αποδοτικών mobile εφαρμογών [6]. Το Flutter είναι άρρηκτα συνδεδεμένο με τη γλώσσα προγραμματισμού Dart, η οποία αναπτύσσεται επίσης από την Google.



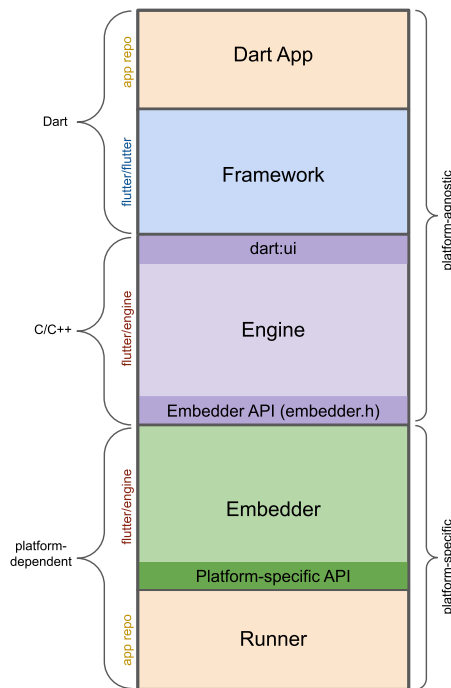
Σχήμα 2.1: Το λογότυπο του Flutter

2.1 Cross-platform

Ως framework ανάπτυξης cross-platform εφαρμογών, το Flutter καθιστά δυνατή τη συγγραφή ενιαίου κώδικα για όλες τις υποστηριζόμενες πλατφόρμες. Ο προγραμματιστής αρκεί να αναπτύξει την εφαρμογή μία φορά στη γλώσσα Dart, αγνοώντας σε μεγάλο βαθμό τις λεπτομέρειες υλοποίησης της κάθε πλατφόρμας, και αφήνοντας το Flutter να χειριστεί όλες τις διαδικασίες που είναι απαραίτητες για τη λειτουργία της εφαρμογής στην εκάστοτε συσκευή.

Η cross-platform λειτουργία επιτυγχάνεται εν μέρει μέσα από τη συμπερίληψη στο Flutter ειδικού rendering engine (Skia/Impeller) ο οποίος χειρίζεται εξ ολοκλήρου τη σωστή εμφάνιση των στοιχείων του UI ανεξαρτήτως της

¹Πηγή για τις πληροφορίες του κεφαλαίου αποτελούν τα επίσημα docs του Flutter [4, 5] εκτός όπου αναγράφεται διαφορετικά



Σχήμα 2.2: Σχηματικό διάγραμμα της αρχιτεκτονικής δομής μίας εφαρμογής Flutter. Πηγή: [4]

πλατφόρμας. Έτσι οι εφαρμογές Flutter δε χρησιμοποιούν τα native UI components του κάθε λογισμικού συστήματος, αλλά αντιθέτως «ζωγραφίζουν» τα δικά τους components πάνω σε έναν χαμβά, pixel ανά pixel. Αυτή η μέθοδος απεικόνισης προσφέρει περισσότερη ομοιομορφία στην εμφάνιση των εφαρμογών σε όλες τις πλατφόρμες, ξεχωρίζοντας το Flutter από άλλα frameworks που χρησιμοποιούν native components όπως το React Native, το οποίο αποτελεί τον κύριο ανταγωνιστή του στον τομέα των cross-platform εφαρμογών.

2.2 Αρχιτεκτονική

Μια εφαρμογή Flutter αποτελείται από τα παρακάτω αρχιτεκτονικά επίπεδα:

- **Runner και Embedder:** Αποτελούν τη βάση της εφαρμογής και τον δίαυλο επικοινωνίας με το λογισμικό σύστημα. Είναι υπεύθυνα για την εκκίνηση της εφαρμογής, την ενημέρωσή της σχετικά με όλα τα system events, και τη δημιουργία του βάθρου πάνω στο οποίο ο engine θα εμφανίσει τα περιεχόμενα της εφαρμογής. Εφόσον τα δύο αυτά επίπεδα

είναι άμεσα συνυφασμένα με το λογισμικό, ο σχετικός κώδικας είναι προσαρμοσμένος ειδικά για την εκάστοτε πλατφόρμα (πχ. χρησιμοποιείται Java/C++ για Android, και Objective-C για iOS)

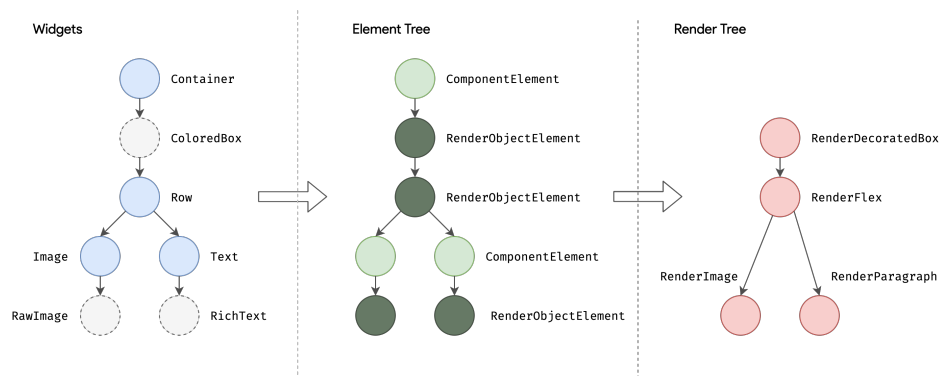
- **Engine:** Αποτελεί τον πυρήνα μίας εφαρμογής Flutter. Συνιστάται από ένα runtime το οποίο εκτελεί τον κώδικα της εφαρμογής που είναι γραμμένος σε Dart, και χειρίζεται παράλληλα την εμφάνιση όλων των γραφικών στην οθόνη. Εφόσον επικοινωνεί με το λογισμικό διαμέσου του embedder, ο engine είναι ανεξάρτητος της πλατφόρμας (platform-agnostic)
- **Framework/Dart App:** Αποτελούν το υψηλού επιπέδου τμήμα της εφαρμογής. Το Flutter framework περιέχει όλα τα components με τα οποία ο προγραμματιστής θα συνθέσει την εφαρμογή, καθώς και εργαλεία για τον χειρισμό υπηρεσιών όπως τα animations και το gesture detection. Το τελευταίο επίπεδο είναι το μέρος της εφαρμογής που σχεδιάζεται εξ ολοκλήρου από τον developer, και περιέχει όλη τη λογική μαζί με το UI, όπως αυτά έχουν οριστεί από τον προγραμματιστή. Τόσο το Flutter framework όσο και ο κώδικας της εφαρμογής είναι γραμμένα στη γλώσσα Dart.

2.3 Widgets

Η βασική μονάδα όλων των στοιχείων του UI στο Flutter είναι το widget. Ένα widget περιγράφει ένα μέρος του UI της εφαρμογής, όπως ένα κουμπί, κείμενο, ή εικονίδιο.

Πέρα από τα εμφανή λειτουργικά στοιχεία, στα widgets περιλαμβάνονται και στοιχεία που ελέγχουν την διάταξη/layout της εφαρμογής, όπως τα widget στοίχισης, padding, και δημιουργίας στηλών/γραμμών. Η εφαρμογή έχει ως ρίζα ένα βασικό widget το οποίο αποτελεί το υπόβαθρο του συνόλου της διεπαφής, και όλα τα στοιχεία του UI δομούνται μέσα από την σύνθεση και την εμφώλευση απλότερων widget. Ο κεντρικός αυτός ρόλος των widget δικαιολογεί το σλόγκαν που έχει υιοθετήσει το Flutter, “*everything is a widget*”.

Το σύνολο των widget που απαρτίζουν μία εφαρμογή Flutter έχει τη δομή ενός δέντρου, με το κάθε widget να διαθέτει έναν γονιό (το widget μέσα στο οποίο είναι εμφωλεμένο) και ενδεχόμενα παιδιά (τα widget τα οποία περιέχει).



Σχήμα 2.3: Δείγμα της ιεραρχίας των widgets με τη μορφή δέντρων. Πηγή: [4]

Στην πραγματικότητα το Flutter δεν ορίζει εσωτερικά το δέντρο των widget απευθείας, αλλά δομεί το λεγόμενο element tree του οποίου οι κόμβοι αντιστοιχούν στα widgets που ορίζονται στον κώδικα, μαζί με όσα μεταβατικά widgets βρίσκονται ενδιάμεσά τους. Επιπλέον, δεν ορίζεται μόνο ένα δένδρο, αλλά δύο· για λόγους απόδοσης, εκτός από το element tree ορίζεται και ένα render tree το οποίο περιέχει πληροφορίες για το rendering (δηλαδή τη σχεδίαση) των αντίστοιχων widget στην οθόνη.

Τα δέντρα αυτά αποτελούν το πιο σημαντικό μέρος του Flutter καθώς όλες οι διαδικασίες ενημέρωσης του UI στηρίζονται στην διάτρεξή τους και στην τροποποίηση των κόμβων τους.

“Flutter is, at its core, a series of mechanisms for efficiently walking the modified parts of trees, converting trees of objects into lower-level trees of objects, and propagating changes across these trees.”

Flutter docs[4]

2.4 UI architecture

Όλες οι εφαρμογές που διαθέτουν γραφικό περιβάλλον χρειάζονται έναν τρόπο να ανανεώνουν τα στοιχεία του UI τους σύμφωνα με τις δράσεις του χρήστη, και γενικότερα σύμφωνα με τις αλλαγές στην εσωτερική κατάσταση της εφαρμογής.

Ένας απλοϊκός τρόπος υλοποίησης του παραπάνω περιλαμβάνει την αρχική δημιουργία του UI, και έπειτα την συνεχή ανανέωση των χαρακτηριστικών των στοιχείων του μέσω του κώδικα. Αυτή η μέθοδος γρήγορα γίνεται μη βιώσιμη

για εφαρμογές που διαθέτουν κάποια στοιχειώδη πολυπλοκότητα, καθώς η μίξη κώδικα/λογικής και UI/εμφάνισης οδηγεί σε ανοργάνωτο κώδικα ο οποίος είναι επιρρεπής σε σφάλματα και είναι δύσκολα επεκτάσιμος.

Μία από τις πρώτες λύσεις στο πρόβλημα αυτό εμφανίστηκε υπό τη μορφή του μοτίβου *Model-View-Controller (MVC design pattern)*. Σύμφωνα με το MVC pattern, ο κώδικας της εφαρμογής που αφορά το UI χωρίζεται γενικά σε τρία μέρη, το model, το view και τον controller[7]:

- Το model αφορά το μέρος του κώδικα το οποίο περιγράφει τη δομή των δεδομένων που παρουσιάζονται στο UI, και αποτελεί την “πηγή αλήθειας” για τα δεδομένα αυτά. Είναι ουσιαστικά η εσωτερική κατάσταση της εφαρμογής, και έχει πλήρη άγνοια για την κατάσταση του UI.
- Το view έχει να κάνει αποκλειστικά με το μέρος του κώδικα που περιγράφει το UI της εφαρμογής, και είναι το κομμάτι με το οποίο αλληλεπιδρά άμεσα ο χρήστης. Εξαρτάται από τα δεδομένα που βρίσκονται στο αντίστοιχο model, και επομένως πρέπει να παρακολουθεί συνεχώς το model για αλλαγές ώστε να ενημερώνει την εμφάνισή του αναλόγως.
- Ο controller αποτελείται από κώδικα που λειτουργεί ως διαμεσολαβητής μεταξύ model και view. Δέχεται τα inputs του χρήστη από το view και ενημερώνει κατάλληλα το model για τις αλλαγές που επιφέρει η κάθε κίνηση του χρήστη στην κατάσταση της εφαρμογής, ενώ μπορεί ενδεχόμενως να ενημερώσει και άμεσα το view.

Με τον διαχωρισμό του κώδικα σε επιμέρους τμήματα, το καθένα με την δική του αρμοδιότητα, επιτυγχάνεται το λεγόμενο *separation of concerns* και συγκεκριμένα *separation of content and presentation*[8]: το UI αποζεύεται από την λογική του προγράμματος, γεγονός που οδηγεί σε πιο εύκολα κατανοητό και επεκτάσιμο κώδικα. Το μοντέλο MVC είναι από τα παλαιότερα του είδους και εμφανίζει πολλαπλές παραλλαγές, οι οποίες είναι μέχρι και σήμερα σε εξαιρετικά ευρεία χρήση.

Το Flutter είναι ένα *reactive framework*² το οποίο επίσης προσπαθεί να ενσωματώσει το *separation of concerns* στο UI των εφαρμογών, υιοθετώντας όμως μία διαφορετική φιλοσοφία από το MVC.

²Τα reactive frameworks βασίζονται στη φιλοσοφία της open-source βιβλιοθήκης *React* που αναπτύσσεται από την εταιρία Meta (Facebook) από το 2013.

Στο Flutter το UI της εφαρμογής περιγράφεται ορίζοντας τα widgets που το αποτελούν. Τα widgets είναι immutable, και η ακριβής εμφάνισή τους είναι συνάρτηση της κατάστασης/state της εφαρμογής την στιγμή δημιουργίας τους, με την μέθοδο `build()` να είναι αρμόδια για τη δόμηση του κάθε widget δεδομένου του state. Επομένως, η αλλαγή της κατάστασης της εφαρμογής πρέπει να συνοδεύεται από την εκ νέου δημιουργία των widgets τα οποία πρέπει να αλλάξουν, δηλαδή από την κλήση της μεθόδου `build()`. Ο ρόλος του Flutter είναι ουσιαστικά η στρατηγική ενημέρωση του widget tree (συγκεκριμένα του render tree) όποτε αυτή είναι απαραίτητη, με όσο το δυνατόν περισσότερο αποδοτικό τρόπο και χωρίς περιττές ανανεώσεις των widgets, έτσι ώστε να επιτυγχάνονται υψηλές επιδόσεις.

2.5 State Management

Έχοντας κάνει λόγο για την σημασία της κατάστασης/state στη λειτουργία της εφαρμογής, διακρίνουμε δύο ευρείες κατηγορίες του state εντός του πλαισίου του Flutter: ephemeral και application[9].

Η εφήμερη κατάσταση (αλλιώς local/UI state) είναι η κατάσταση ενός μεμονωμένου widget, οι οποία περιορίζεται εντός του widget αυτού και δεν αλληλεπιδρά με άλλα μέρη της εφαρμογής. Η κατάσταση αυτή είναι συνήθως απλή, και αφορά κυρίως την εμφάνιση του widget και όχι κάποια σύνθετη δομή δεδομένων. Η κλάση `StatefulWidget` αναπαριστά ένα widget που διαθέτει ephemeral state, και συνοδεύεται από την αντίστοιχη κλάση `State` η οποία περιέχει ακριβώς τα δεδομένα της κατάστασης του στοιχείου. Η μέθοδος `build()` ορίζεται εντός της κλάσης `State`, και περιγράφει τη δομή του widget συναρτήσει της κατάστασης της κλάσης.

Όταν η κατάσταση του widget αλλάζει έτσι ώστε να χρειάζεται η ανανέωση του UI, τότε το Flutter πρέπει να ειδοποιηθεί καταλλήλως ώστε να καλέσει τη μέθοδο `build()`. Αυτό ακριβώς επιτυγχάνεται μέσω της μεθόδου `setState()`, η οποία καλείται εντός του δεδομένου widget, σημειώνοντάς το ως “dirty” και σηματοδοτώντας στο Flutter την ανάγκη κλήσης της `build()` ώστε το UI να ανανεωθεί με τα νέα δεδομένα.

Το application (ή shared) state από την άλλη είναι η κατάσταση της εφαρμογής που δεν περιορίζεται σε ένα widget, αλλά την οποία μοιράζονται πολλαπλά διαφορετικά μέρη του UI. Το γεγονός ότι το app state δεν ανήκει

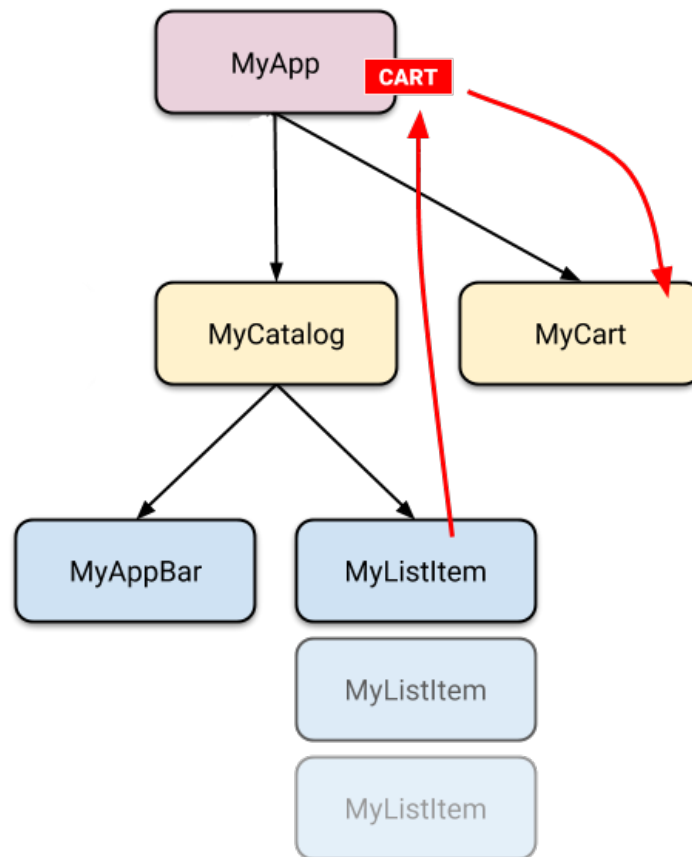
σε ένα μόνο widget παρουσιάζει το εξής πρόβλημα: πως μπορεί ένα widget να έχει πρόσβαση στο app state σε οποιοδήποτε μέρος του δέντρου και αν βρίσκεται, και πως μπορεί η πρόσβαση αυτή να γίνεται με τον πιο απλό, ασφαλή, αλλά και αποδοτικό τρόπο;

Η απάντηση σε αυτό το ερώτημα βασίζεται στην τακτική που ονομάζεται “*lifting state up*”, την οποία μπορούμε να μεταφράσουμε ως “αναγωγή κατάστασης”. Μέσω της αναγωγής κατάστασης, η κατάσταση που μοιράζονται πολλαπλά στοιχεία μεταφέρεται σε έναν κόμβο που βρίσκεται ανώτερα στο δέντρο από τα στοιχεία αυτά. Με άλλα λόγια, το shared state μεταφέρεται σε κόμβο που αποτελεί *κοινό πρόγονο* των στοιχείων στο δέντρο.

Το shared state, ως parent πια των εμπλεκόμενων widgets, μπορεί να ειδοποιεί όλα τα widget-παιδιά του για αλλαγές στην κατάσταση, ενώ το κάθε widget-παιδί λαμβάνει πρόσβαση στο shared state αφού το αναζητήσει διατρέχοντας τους απογόνους του του δέντρου.

Οι μηχανισμοί για τη παραπάνω λειτουργία προσφέρονται από την βιβλιοθήκη `provider`[10], η οποία χρησιμοποιείται κατά κόρον και από την ομάδα του Flutter. Η βιβλιοθήκη παρέχει τους λεγόμενους `Provider` οι οποίοι απλά περικλείουν την κλάση που αποτελεί το shared state, κάνοντάς την εύκολα διαθέσιμη στα υπόλοιπα μέρη της εφαρμογής. Ωστόσο, σε συνδυασμό με την κλάση `ChangeNotifier` η οποία είναι μέρος του Flutter framework, ένας `Provider` μπορεί επιπροσθέτως να ειδοποιεί τα widget που έχουν πρόσβαση στο shared state κάθε φορά που η κατάσταση μεταβάλλεται, πυροδοτώντας την ανανέωσή τους.

Τα στοιχεία που επιθυμούν να παρακολουθούν έναν `Provider` περικλείονται από ένα widget `Consumer` το οποίο αναλαμβάνει την ακρόαση του state που παρέχει ο συγκεκριμένος `Provider` και χειρίζεται αυτόματα το rebuild του εσωκλειόμενου widget. Με αυτόν τον τρόπο έχουμε πλήρως αυτόματο συγχρονισμό του shared state και ενημέρωση του UI από οποιοδήποτε μέρος της εφαρμογής σε οποιοδήποτε άλλο, υλοποιώντας πλήρως τη φιλοσοφία του reactive programming.



Σχήμα 2.4: Παράδειγμα αναγωγής κατάστασης. Η δεδομένη εφαρμογή περιέχει ένα μία λίστα από widgets `MyListItem` που αποτελούν τα προϊόντα ενός e-shop, και ένα widget `MyCart` που εμφανίζει τα προϊόντα που έχουν προστεθεί στο καλάθι. Τόσο τα `MyListItem` όσο και το `MyCart` πρέπει να έχουν πρόσβαση σε μία κλάση που αναπαριστά το καλάθι του χρήστη, αφού για παράδειγμα όταν ένα `MyListItem` πατηθεί το αντίστοιχο προϊόν πρέπει να προστίθεται στο καλάθι και ταυτόχρονα να εμφανίζεται στο `MyCart`. Έτσι δημιουργείται ένας provider `Cart` που περιέχει την κατάσταση του καλαθιού, και τοποθετείται στον κοινό πρόγονο του `MyListItem` και του `MyCart` δηλαδή στο `MyApp`. Η προσθήκη ενός προϊόντος από το `MyListItem` προσπελάζει τον provider `Cart`, ο οποίος ειδοποιεί αμέσως το `MyCart` για την αλλαγή. Πηγή: [9]

Κεφάλαιο 3

Node.js

3.1 JavaScript

Η JavaScript είναι γλώσσα προγραμματισμού η οποία κατέχει κεντρική θέση στον τομέα των διαδικτυακών εφαρμογών. Η JavaScript χρησιμοποιείται πρωτίστως στο client-side/frontend μέρος των εφαρμογών, όπου εκτελείται από τον browser του χρήστη με σκοπό να προσφέρει διαδραστικότητα σε ιστοσελίδες HTML/CSS, αλλά η χρήση της εντοπίζεται και στο backend κομμάτι, δηλαδή σε servers.

Δημιουργήθηκε από τον Brendan Eich το 1995 υπό την ηγεσία της Netscape ως πρόσθετο στον τότε δημοφιλή browser Netscape Navigator, προκειμένου να καταστήσει δυνατή την εκτέλεση δυναμικής συμπεριφοράς στις έως τότε στατικές ιστοσελίδες[11]. Σταδιακά και άλλες ανταγωνίστριες εταιρείες ενσωμάτωσαν την JavaScript στους browser τους, αναπτύσσοντας η καθεμία τον δικό της engine για την εκτέλεση κώδικα JavaScript (V8 από την Google, JavaScriptCore από την Apple, SpiderMonkey από τη Mozilla).

Η εξάπλωση της JavaScript οδήγησε στην ανάγκη τυποποίησης της γλώσσας για χρήση σε διαδικτυακές εφαρμογές, ώστε να διευκολυνθεί η συγγραφή κώδικα συμβατού με όλους τους browser της αγοράς. Έτσι το 2009, μετά από τη σύσταση επιτροπής αποτελούμενη από εκπροσώπους των κύριων εταιρειών (Google, Microsoft, Mozilla, και άλλες) δημιουργήθηκε το πρότυπο ECMAScript 5 του οποίου οι προδιαγραφές, μαζί με όλες τις αναθεωρήσεις τους, ακολουθούνται πλέον από όλους τους συμβατικούς browsers.

Η JavaScript είναι τυπικά μία interpreted γλώσσα, ωστόσο σε όλες τις

συμβατικές υλοποιήσεις της (συμπεριλαμβανομένων όλων των σύγχρονων browser) ο κώδικας γίνεται και just-in-time compiled (JIT). Με αυτόν τον τρόπο επιτυγχάνεται η γρήγορη ταχύτητα εκτέλεσης η οποία είναι απαραίτητη στις διαδικτυακές εφαρμογές, καθώς συνδυάζεται ο μικρός χρόνος εκκίνησης ενός interpreter μαζί με τις υψηλές επιδόσεις που προσφέρει η μεταγλώττιση κομβικών τμημάτων του κώδικα σε κώδικα μηχανής.

3.2 Χαρακτηριστικά¹

Όσον αφορά τα χαρακτηριστικά της, η JavaScript είναι γλώσσα που προσφέρεται τόσο για αντικειμενοστραφή όσο και για συναρτησιακό προγραμματισμό.

Οι αντικειμενοστραφείς λειτουργίες της γλώσσας υλοποιούνται μέσα από τα πρότυπα (prototypes). Κάθε αντικείμενο που δημιουργείται κατά την εκτέλεση του κώδικα μέσω ενός constructor διαθέτει ένα property που αναφέρεται στο πρότυπο στο οποίο ανήκει. Το πρότυπο είναι και αυτό με τη σειρά του ένα αντικείμενο το οποίο περιέχει ένα σύνολο από properties. Αυτό σημαίνει ότι κάθε αντικείμενο έχει αυτόματα πρόσβαση σε όλες τις ιδιότητες του προτύπου στο οποίο ανήκει, και επομένως όλα τα αντικείμενα του ίδιου προτύπου μοιράζονται όλα τα properties του. Κάθε φορά που προσπελάζεται μία ιδιότητα ενός αντικειμένου, αρχικά ελέγχεται η ύπαρξη property του αντικειμένου με το ίδιο όνομα (own property). Σε περίπτωση που η ιδιότητα δε βρεθεί ανάμεσα στα own properties του αντικειμένου, η ιδιότητα αναζητείται ανάμεσα στα properties του προτύπου στο οποίο ανήκει. Η διαδικασία αυτή συνεχίζεται αναζητώντας αναδρομικά στο πρότυπο του προτύπου κ.ο.κ, τερματίζοντας είτε μόλις η ιδιότητα βρεθεί, είτε μόλις η αλυσίδα των προτύπων φτάσει στο βασικό πρότυπο που ονομάζεται 'Object'. Συνεπώς, τα πρότυπα υλοποιούν ακριβώς την έννοια της *κληρονομικότητας*, η οποία αποτελεί τον βασικό πυλώνα του αντικειμενοστραφούς προγραμματισμού. Εξάλλου, τα πρότυπα προσφέρουν μεγάλα περιθώρια βελτιστοποίησης, καθώς ιδιότητες οι οποίες είναι ίδιες για μία οικογένεια αντικειμένων μπορούν να τοποθετηθούν σε ένα κοινό πρότυπο, με αποτέλεσμα τη μείωση της απαιτούμενης μνήμης και την αύξηση της επίδοσης.

Η έκδοση ECMAScript 5 πρόσθεσε κλάσεις στην JavaScript ως βασική πια μονάδα της γλώσσας. Η υλοποίηση των κλάσεων γίνεται στην πραγματι-

¹Πηγή για τις παρακάτω πληροφορίες αποτελεί το επίσημο ECMAScript specification[12] και τα MDN Web Docs[13]

κότητα και αυτή μέσα από τα πρότυπα, αλλά όλες οι ιδιότητες που αναμένονται από μία ολοκληρωμένη αντικειμενοστραφή γλώσσα (στατικές μέθοδοι, access modifiers, getters/setters, και φυσικά κληρονομικότητα) είναι πια άμεσα διαθέσιμες στον προγραμματιστή.

Από την άλλη, η JavaScript χαρακτηρίζεται και ως γλώσσα συναρτησιακού προγραμματισμού. Οι συναρτήσεις στην JavaScript αντιμετωπίζονται ως απλά αντικείμενα, γεγονός που σημαίνει ότι με απόλυτα φυσικό τρόπο μπορούν να χρησιμοποιηθούν ως μεταβλητές, να οριστούν ως literals (anonymous functions), και να αποτελέσουν παραμέτρους άλλων συναρτήσεων. Οι συναρτήσεις στην JavaScript είναι δηλαδή first-class citizens. Έτσι η γλώσσα προσφέρεται για προγραμματισμό που βασίζεται στη σύνθεση συναρτήσεων, ενώ η ύπαρξη περαιτέρω λειτουργιών όπως τα function closures επιτρέπουν πιο προχωρημένες μεθόδους συναρτησιακού προγραμματισμού.

Ένα ακόμη σημαντικό χαρακτηριστικό της JavaScript είναι το weak και dynamic typing. Το dynamic typing αντιδιαστέλλεται με το static typing, και αναφέρεται στο γεγονός ότι η γλώσσα δεν πραγματοποιεί στατικό έλεγχο των τύπων κατά τη μεταγλώττιση του κώδικα, αλλά αντιθέτως καθορίζει τους τύπους των μεταβλητών μόνο κατά τη στιγμή της εκτέλεσης, βάσει της τιμής τους. Αυτό σημαίνει ότι ο τύπος μίας μεταβλητής δεν καθορίζεται όταν αυτή δηλώνεται, και μπορεί να αλλάξει ανά πάσα στιγμή κατά την εκτέλεση του προγράμματος (σε αντίθεση για παράδειγμα με τη C/C++ όπου ο τύπος δηλώνεται πάντα όταν ορίζεται μια μεταβλητή και παραμένει αμετάβλητος καθ' όλη τη διάρκεια του προγράμματος).

Ο όρος weakly typed από την άλλη δεν είναι τόσο καθορισμένος. Σε γενικές γραμμές αναφέρεται στο γεγονός ότι η γλώσσα επιτρέπει την αυτόματη και υπόρρητη μετατροπή τιμών από ένα τύπο σε άλλον. Η JavaScript χαρακτηρίζεται ως weakly typed επειδή πραγματοποιεί implicit type conversions πρακτικά σε κάθε περίπτωση πράξης μεταξύ διαφορετικών τύπων, αντί να εγείρει εξαίρεση τύπου όπως θα έκανε μία strongly typed γλώσσα. Για παράδειγμα, στην JavaScript μία “πρόσθεση” ενός string με έναν αριθμό όπως `"example" + 2024` μετατρέπει τον αριθμό σε string και έπειτα πραγματοποιεί concatenation, με τελικό αποτέλεσμα το string `"example2024"`.

Το dynamic/weak typing της JavaScript μπορεί να θεωρηθεί πιο πρακτικό από το static/strong typing άλλων γλωσσών, καθώς απαλλάσσει τον προγραμ-

ματιστή από την ανάγκη ορισμού και διαχείρισης όλων των τύπων, ενώ μειώνει τον νοητικό φόρτο που οφείλεται στα casting και τις ρητές μετατροπές τύπων. Ωστόσο, αυτό το είδος “χαλαρού” συστήματος τύπων μπορεί να δημιουργήσει προβλήματα.

Οι δυναμικοί τύποι δεν προφυλάσσουν το πρόγραμμα από κάποια βασικά σφάλματα τα οποία ένα στατικό σύστημα τύπων θα εντόπιζε άμεσα κατά τη μεταγλώττιση του προγράμματος, ενώ οι “κρυφές” μετατροπές τύπων μπορούν πολύ εύκολα να έχουν απρόσμενες συνέπειες οι οποίες δεν είναι άμεσα εμφανείς στον προγραμματιστή. Παράλληλα, η χρήση τύπων μπορεί στην πραγματικότητα να μειώσει τον νοητικό φόρτο του προγραμματιστή· εφόσον ο τύπος κάθε μεταβλητής είναι γνωστός και αμετάβλητος, η λειτουργία του προγράμματος είναι πιο εύκολα ελέγξιμη και συντηρήσιμη.

3.3 TypeScript

Η TypeScript είναι γλώσσα που επεκτείνει την JavaScript, προσφέροντας στατικό έλεγχο τύπων κατά τη μεταγλώττιση του κώδικα προκειμένου να αντιμετωπίσει τις αδυναμίες του dynamic typing. Οποιοδήποτε πρόγραμμα JavaScript μπορεί να προαχθεί σε TypeScript προσθέτοντας προαιρετικούς τύπους στις μεταβλητές που χρησιμοποιεί.

Έτσι, σε αντίθεση με τη γνήσια JavaScript όπου οι τύποι όλων των μεταβλητών καθορίζονται μόνο τη στιγμή της εκτέλεσης, η TypeScript συνάγει τους τύπους των μεταβλητών, και ελέγχει για πιθανά σφάλματα τύπων *πριν* την εκτέλεση του κώδικα, αποφεύγοντας έτσι ένα σημαντικό μέρος των runtime errors και bugs[14]. Η TypeScript δίνει επίσης τη δυνατότητα στον προγραμματιστή να προσθέσει type annotations σε παραμέτρους συναρτήσεων, ενώ είναι επίσης δυνατός ο ορισμός interfaces για τον χειρισμό δομημένων δεδομένων (structured data), όπως αυτά που στέλνει και δέχεται ένα API. Με αυτόν τον τρόπο, εκτός από την αυξημένη προστασία έναντι σφαλμάτων, λανθασμένης χρήσης μεταβλητών και ανεπιθύμητων συμπεριφορών, ο ίδιος ο κώδικας γίνεται συγχρόνως πιο κατανοητός και εύκολα διαχειρίσιμος από τον προγραμματιστή.

Ένα αρχείο TypeScript μεταφράζεται πάντα σε καθαρή JavaScript (transpiled) πριν εκτελεστεί. Κατά τη διάρκεια της μετάφρασης αυτής όλες οι πληροφορίες που αφορούν τους τύπους αφαιρούνται από το πρόγραμμα εφόσον έχει επαληθευτεί η ορθότητά τους, με αποτέλεσμα ο τελικός κώδικας να είναι

εκτελέσιμος από οποιονδήποτε JavaScript engine χωρίς την ανάγκη ειδικής μεταχείρισης των χαρακτηριστικών της TypeScript.

3.4 Node.js

Το Node.js είναι ένα open-source runtime environment για την JavaScript. Δημιουργήθηκε το 2009 από τον Ryan Dahl με στόχο να προσφέρει τη δυνατότητα εκτέλεσης αποδοτικών προγραμμάτων υψηλού ταυτοχρονισμού (high concurrency) βασιζόμενη σε μία event-driven και non-blocking αρχιτεκτονική[15, 16]. Η κύρια χρήση του Node.js είναι ως περιβάλλον για την εκτέλεση back-end εφαρμογών, δηλαδή για τη λειτουργία ενός server.



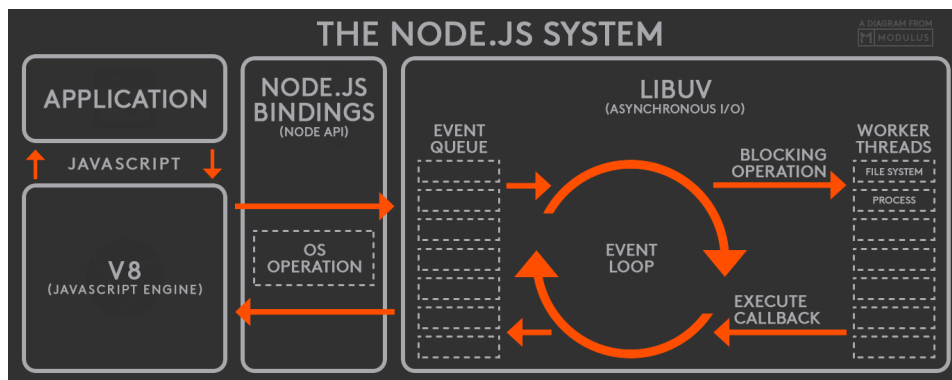
Σχήμα 3.1: Το λογότυπο του Node.js

Μέσω του Node.js ο Ryan Dahl θέλησε να δώσει λύση σε ένα πρόβλημα που ήταν μέχρι τότε διαδεδομένο στον τομέα των εφαρμογών server, αυτό της αδυναμίας διαχείρισης μεγάλου όγκου ταυτόχρονων συνδέσεων. Η δομή των περισσότερων εφαρμογών server βασιζόταν στην αυστηρά σειριακή εκτέλεση εντολών και στη χρήση πολλών ξεχωριστών threads για την εξυπηρέτηση όλων των συνδέσεων, επομένως η απαιτήσις πόρων και το latency ήταν ιδιαίτερα αυξημένα για εφαρμογές μεγάλης κλίμακας. Η χρήση διαφορετικών threads θεωρούταν απαραίτητη για την αποφυγή παύσης εκτέλεσης της εφαρμογής στην περίπτωση blocking κλήσεων I/O, όπως πρόσβαση στον δίσκο, σε μία βάση δεδομένων, ή στο διαδίκτυο.

Το Node.js κατάφερε να ξεπεράσει αυτό το εμπόδιο και αξιοποιώντας *non-blocking* κλήσεις I/O με το λεγόμενο *event loop* επιτρέπει την δημιουργία γρήγορων εφαρμογών διαδικτύου που λειτουργούν σε ένα μόνο thread.

Ο κεντρικός άξονας του Node.js είναι το event loop. Το event loop είναι ο βρόχος ο οποίος εκτελείται από το πρόγραμμα καθ' όλη τη διάρκειά ζωής του και είναι υπεύθυνος για την δρομολόγηση των ασύγχρονων λειτουργιών. Το event loop δέχεται όλα τα σχετικά events που εμφανίζονται (πχ. HTTP requests, callbacks), τα επεξεργάζεται είτε αναθέτοντάς τα στο λειτουργικό σύστημα είτε μέσω ενός thread pool, και έπειτα επιστρέφει στην κανονική ροή του προγράμματος[18].

Το γεγονός ότι το Node.js αναθέτει τις εργασίες αυτές στο kernel του συστήματος σημαίνει ότι το ίδιο είναι ελεύθερο να συνεχίσει να επεξεργάζεται τα εισερχόμενα events και να εκτελεί τις σύγχρονες διεργασίες, χωρίς να



Σχήμα 3.2: Η βασική αρχιτεκτονική του Node.js. Πηγή: [17].

χρειάζεται να περιμένει την ολοκλήρωση των blocking διεργασιών. Αυτό, σε συνδυασμό με τη single-threaded φύση του event loop, επιτρέπει στο Node.js να χειρίζεται πολύ μεγαλύτερο αριθμό ταυτόχρονων συνδέσεων από άλλες υλοποιήσεις (όπως έναν κλασσικό PHP server), με λιγότερες απαιτήσεις σε μνήμη. Ο μηχανισμός του event loop αλλά και όλες οι non-blocking λειτουργίες του Node.js, πχ. το network και file I/O, παρέχονται από τη βιβλιοθήκη libuv.

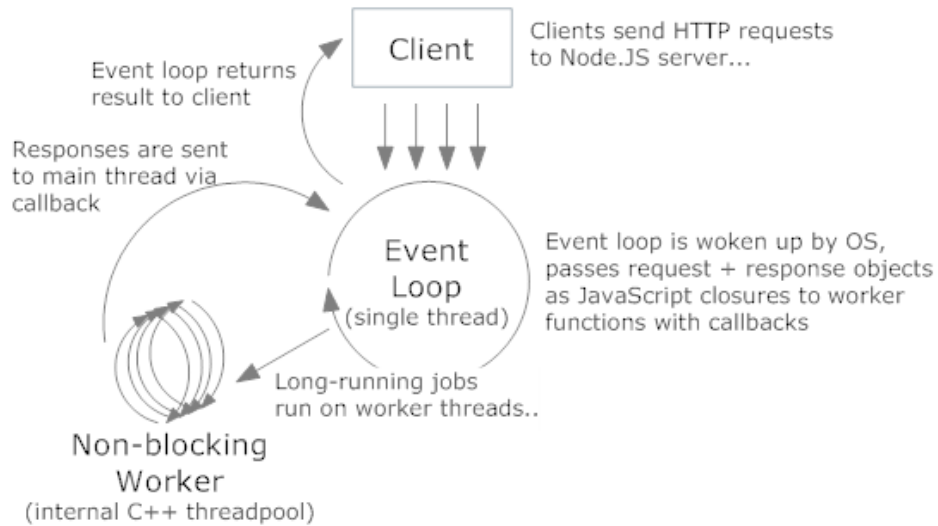
Παρόλο που ο κώδικας JavaScript της εφαρμογής είναι single-threaded, το Node.js αξιοποιεί πολλαπλά threads στο παρασκήνιο για τις παραπάνω λειτουργίες.

Προκειμένου να εκτελέσει τον κώδικα JavaScript εκτός του περιβάλλοντος ενός browser, το Node.js χρησιμοποιεί τον V8 engine της Google.

Ο κώδικας που εκτελεί το Node.js είναι κώδικας JavaScript. Η JavaScript επιλέχθηκε ως η γλώσσα εκτέλεσης όχι μόνο γιατί ήταν ήδη διαδεδομένη στον χώρο των διαδικτυακών εφαρμογών, αλλά κυρίως γιατί είναι κατάλληλα σχεδιασμένη για να υποστηρίξει τις απαιτήσεις τους Node.js. Από την απαρχή της η JavaScript υποστήριζε τον event-driven και ασύγχρονο προγραμματισμό, καθώς αυτός ήταν απαραίτητος για την ομαλή non-blocking λειτουργία των ιστοσελίδων, ενώ οι ανώνυμες συναρτήσεις και τα function closures ευνοούσαν περαιτέρω τη χρήση των callbacks. Η χρήση της JavaScript στο Node.js δίνει επίσης τη δυνατότητα για full-stack JavaScript development, δηλαδή ομοιόμορφη χρήση της JavaScript για την ανάπτυξη τόσο του frontend όσο και του backend μέρους μίας εφαρμογής.

Το Node.js άρχισε να γίνεται δημοφιλές σύντομα μετά από την παρουσίασή του, και πλέον η πλειοψηφία όλων των εταιριών παγκοσμίως το χρησιμοποιούν

Node.JS Processing Model



Σχήμα 3.3: Η λειτουργία του event loop στο Node.js. Πηγή: [19].

σε κάποιο τουλάχιστον μέρος του backend τους. Κάποιες εταιρίες που χρησιμοποιούν σε μεγάλο βαθμό το Node.js είναι η Netflix[20], η PayPal[21], η LinkedIn[22], η NASA[23], και η Bloomberg[24]. Στο παρελθόν η Uber επίσης χρησιμοποιούσε το Node.js για το μεγαλύτερο μέρος του backend της, και μάλιστα ήταν από τις πρώτες μεγάλες εταιρίες που το υιοθέτησαν[25].

Κεφάλαιο 4

WebSocket

Το WebSocket είναι ένα πρωτόκολλο επικοινωνίας το οποίο επιτρέπει την ανταλλαγή μηνυμάτων μεταξύ client και host και προς τις δύο κατευθύνσεις ταυτόχρονα (full duplex). Το WebSocket αποτελεί συγκεκριμένα ένα application-layer protocol σύμφωνα με την ορολογία του μοντέλου OSI, το οποίο χρησιμοποιεί το TCP ως transport layer για τη μεταφορά των frames που περιέχουν τα δεδομένα προς αποστολή.

Μία πρώιμη εκδοχή του WebSocket εμφανίστηκε για πρώτη φορά το 2008 σε ένα προσχέδιο των προδιαγραφών του HTML5 υπό το όνομα TCPConnection, ενώ η τελική έκδοση του πρωτοκόλλου δημοσιεύτηκε το 2011. Πριν από το WebSocket, οι επιλογές για duplex επικοινωνία client και server ήταν περιορισμένες, και βασιζόνταν συχνά στην κατάχρηση του πρωτοκόλλου HTTP. Για παράδειγμα, μία μέθοδος που χρησιμοποιούταν πριν τη διάδοση των WebSocket για αμφίδρομη επικοινωνία με έναν server ήταν το λεγόμενο *long polling*. Σύμφωνα με την τακτική του long polling ο χρήστης στέλνει κανονικά HTTP requests στον server, αλλά αντίθετα με τη συνηθισμένη περίπτωση όπου ο server στέλνει αμέσως το response στο χρήστη, ο server αναμένει, αφήνοντας τη σύνδεση ανοιχτή, και στέλνοντας το αντίστοιχο response μόνο μόλις εμφανιστούν νέα δεδομένα. Τη στιγμή που ο χρήστης λάβει το response αυτό η σύνδεση τερματίζεται, και ο χρήστης στέλνει αμέσως ένα νέο request, συνεχίζοντας την ίδια διαδικασία.

Παρόλο που με αυτόν τον τρόπο επιτυγχάνεται η συνεχής επικοινωνία του server με τον client, το πρωτόκολλο HTTP δεν είναι σχεδιασμένο για αυτόν τον σκοπό, και μία τέτοια υλοποίηση δεν είναι ιδανική. Το long polling στις

περισσότερες περιπτώσεις καταναλώνει πολύ περισσότερους πόρους από ότι θα χρειαζόταν μία απλή σύνδεση, ενώ εισάγονται μεγάλες καθυστερήσεις λόγω του overhead που απαιτεί το HTTP. Εξ'άλλου, το long polling προορίζεται περισσότερο για περιπτώσεις όπου ο μεγαλύτερος όγκος της επικοινωνίας συμβαίνει στην κατεύθυνση server προς client. Παρόμοιες λύσεις που βασίζονται στο HTTP όπως τα *Server Sent Events* λύνουν κάποια από τα προβλήματα του long polling, αλλά εξακολουθούν να διαθέτουν αυτήν την ασυμμετρία ανάμεσα στα δύο άκρα της σύνδεσης.

Το WebSocket σχεδιάστηκε εξ αρχής ως πρωτόκολλο προορισμένο για full duplex συνδέσεις, και επομένως υπερέχει των παραπάνω εναλλακτικών όσον αφορά την real-time αμφίδρομη επικοινωνία client και server. Είναι σημαντικό να αναφέρουμε ότι όλες οι παραπάνω μέθοδοι (συμπεριλαμβανομένων των WebSocket) σχεδιάστηκαν κυρίως για χρήση σε browsers, και επομένως οι περιορισμοί που αναφέραμε είναι απόρροια των περιορισμών που επιβάλλουν οι browsers.

Ένα εναλλακτικό πρωτόκολλο το οποίο είναι κατάλληλο για αμφίδρομη real-time επικοινωνία είναι το WebRTC. Το WebRTC είναι σχεδιασμένο για real-time *communication*, δηλαδή για την επικοινωνία χρηστών μέσω audio και video streaming, και για τον σκοπό αυτό χρησιμοποιεί το πρωτόκολλο UDP. Το UDP επιτυγχάνει το χαμηλό latency το οποίο είναι απαραίτητο για την ομαλή οπτικοακουστική επικοινωνία, κάνοντας όμως αρκετές υποχωρήσεις όσον αφορά την αξιοπιστία των πακέτων δεδομένων. Το UDP συγκεκριμένα δεν προσφέρει καμία εγγύηση για την ακεραιότητα των πακέτων, αλλά ούτε και για την σωστή σειρά λήψης τους από τον δέκτη.

Αυτό το γεγονός καθιστά το πρωτόκολλο ακατάλληλο για την περίπτωση μας, δεδομένου ότι επιθυμούμε να είμαστε σίγουροι ότι όλα τα μηνύματα φτάνουν από τους χρήστες στον server στη σωστή σειρά, χωρίς σφάλματα, επαναλήψεις, ή απώλειες. Το πρωτόκολλο το οποίο προσφέρει τις εγγυήσεις αυτές είναι το TCP, και πάνω σε αυτό το πρωτόκολλο βασίζεται το WebSocket, γεγονός που δικαιολογεί την επιλογή μας.

Για την επίτευξη αμφίδρομης επικοινωνίας μεταξύ δύο αυθαίρετων χρηστών, μία απλή σύνδεση TCP είναι σαφώς αρκετή, και μάλιστα είναι πιο αποδοτική σε σχέση με την προσθήκη του WebSocket. Πράγματι, καθώς η εφαρμογή μας είναι native και δεν κάνει χρήση του browser, θα μπορούσαμε να υλοποιήσουμε την επικοινωνία χρηστών και server μέσω απλών TCP sock-

ets, χωρίς τη χρήση των WebSocket. Ωστόσο, το TCP είναι ένα stream-based πρωτόκολλο, γεγονός που σημαίνει ότι τα δεδομένα φτάνουν στον δέκτη ως μία συνεχή ροή από bytes, χωρίς διαχωρισμό μεταξύ των μηνυμάτων που στέλνει ο αποστολέας. Με άλλα λόγια κάθε φορά που ο δέκτης διαβάζει τα δεδομένα από το socket, δεν λαμβάνει απαραίτητα ένα μήνυμα όπως αυτό στάλθηκε από τον πομπό, αλλά ενδεχομένως να λάβει μόνο μέρος ενός μηνύματος, ή ακόμα και πολλαπλά μηνύματα μαζί.

Έτσι, το πρωτόκολλο TCP χρειάζεται να επαυξηθεί με το κατάλληλο *framing* των δεδομένων, το οποίο θα παρουσιάζει στον δέκτη μεμονωμένα μηνύματα αντί για ένα byte stream. Το πρωτόκολλο WebSocket κάνει ακριβώς αυτό, προσθέτοντας απλά έναν μηχανισμό framing πάνω στο TCP με πολύ μικρό overhead[26]. Για χάρη ευκολίας λοιπόν επιλέξαμε να κάνουμε χρήση του WebSocket στην εφαρμογή μας, αντί να υλοποιήσουμε το δικό μας πρωτόκολλο framing.

Κεφάλαιο 5

Frontend

Η εφαρμογή κινητού δημιουργήθηκε χρησιμοποιώντας το framework του Flutter. Το Flutter επιλέχθηκε ως framework κυρίως λόγω της δυνατότητας που προσφέρει για εύκολη και γρήγορη ανάπτυξη εφαρμογών, και λόγω της απαίτησής μας για cross-platform λειτουργία. Η ανάπτυξη της εφαρμογής στη native γλώσσα κάθε πλατφόρμας θα απαιτούσε σημαντικά περισσότερο χρόνο, καθώς θα χρειαζόταν να δημιουργήσουμε δύο ανεξάρτητες εφαρμογές (μία για την πλατφόρμα Android, και μία για την πλατφόρμα iOS) γράφοντας πρακτικά τη διπλάσια ποσότητα κώδικα σε δύο διαφορετικές γλώσσες. Εκτός από τον διπλασιασμό του χρόνου συγγραφής του κώδικα, η ύπαρξη δύο codebase αυξάνει τον αριθμό των ενδεχόμενων σφαλμάτων που μπορεί να προκύψουν, καθιστά σε βάθος χρόνου πιο δύσκολη τη συντήρηση, και απαιτεί επιπλέον προσοχή για τη διασφάλιση της ισότητας των δύο εκδόσεων της εφαρμογής (feature parity). Αξιοποιώντας το Flutter γράψαμε έναν ενιαίο κώδικα ο οποίος λειτουργεί αμέσως και στις δύο πλατφόρμες, ενώ είμαστε σίγουροι ότι η συμπεριφορά της εφαρμογής θα είναι ίδια για όλους τους χρήστες ανεξαρτήτως της συσκευής που χρησιμοποιούν.

5.1 Λειτουργίες

Ως οδηγός, ο χρήστης έχει τη δυνατότητα να προσθέσει στο προφίλ του τα οχήματα που χρησιμοποιεί, προσδιορίζοντας τα χαρακτηριστικά που επιτρέπουν την αναγνώρισή τους από τους πεζούς. Συγκεκριμένα, για το κάθε όχημα ο χρήστης προσδιορίζει το μοντέλο (επιλέγοντας από μία περιεκτική λίστα) και τον αριθμό πινακίδας. Προκειμένου να διευκολυνθεί η άμεση αναγνώριση του

οχήματος από τους επιβάτες ο χρήστης μπορεί προαιρετικά να προσθέσει το χρώμα αλλά και μία φωτογραφία του.

Πέρα των στοιχείων του οχήματος, ο κάθε χρήστης (πεζός ή οδηγός) μπορεί επίσης προαιρετικά να ανεβάσει μία φωτογραφία προφίλ με την οποία θα παρουσιάζεται στους υπόλοιπους χρήστες. Τέλος, ο χρήστης έχει μία βαθμολογία η οποία βασίζεται σε όλες τις προηγούμενες αλληλεπιδράσεις τους με άλλους χρήστες της εφαρμογής, η οποία αναπαριστά την ποιότητα του ως οδηγός ή επιβάτης.

Κατά την έναρξη της εφαρμογής, ο χρήστης παραπέμπεται στην υπηρεσία ταυτοποίησης της σχολής προκειμένου να επαληθευτεί η ταυτότητά του ως φοιτητής. Κατά την επιτυχή ταυτοποίηση η εφαρμογή ανοίγει μία σύνδεση WebSocket με τον server, στέλνοντας ταυτόχρονα τον AM και το όνομα του φοιτητή. Σημαντικό είναι το γεγονός ότι η σύνδεση με τον server επιτυγχάνεται μόνο εφόσον έχει επαληθευτεί η ταυτότητα του χρήστη. Με αυτόν τον τρόπο όχι μόνο απαγορεύεται η χρήση της εφαρμογής από τρίτους, αλλά και ο server προστατεύεται από κακόβουλες ενέργειες. Ακόμα και αν κάποιος χρήστης επιχειρήσει να κάνει κατάχρηση της εφαρμογής θα πρέπει να έχει ήδη συνδεθεί με τα προσωπικά του στοιχεία, και σε αυτή την περίπτωση η ταυτότητά του μπορεί να προσδιοριστεί και να ληφθούν τα απαραίτητα μέτρα όπως ο αποκλεισμός του συγκεκριμένου λογαριασμού.

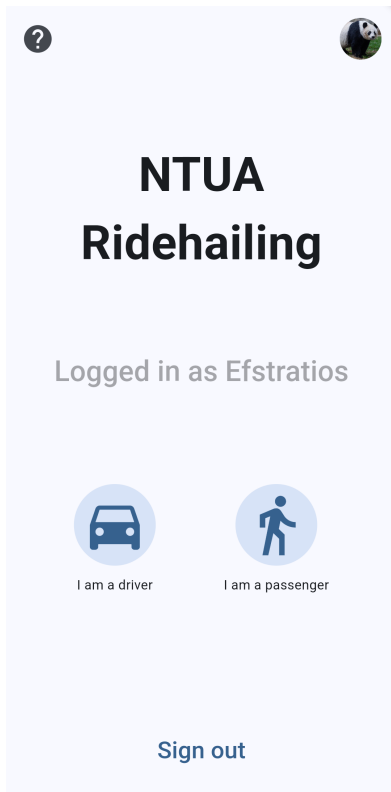
Για την ταυτοποίηση χρησιμοποιούμε το AppAuth, έναν OAuth/OpenID Connect client ο οποίος είναι σχεδιασμένος για την επαλήθευση ταυτότητας χρηστών μέσω κινητών εφαρμογών. Η ταυτοποίηση πραγματοποιείται με ασφαλή τρόπο στη συσκευή και το αποτέλεσμα της είναι ένα token (JWT) που περιέχει τα προσωπικά στοιχεία του χρήστη μαζί με μία “υπογραφή” που πιστοποιεί την αυθεντικότητά του. Το token αυτό στέλνεται στον server κατά την έναρξη της συνεδρίας πάντα μέσω HTTPS σύνδεσης, και εφόσον επαληθευτεί ως έγκυρο ο server γνωρίζει πια την ταυτότητα του χρήστη πίσω από τη συγκεκριμένη σύνδεση. Η διαδικασία γίνεται με τέτοιο τρόπο ώστε σε καμία περίπτωση να μη φανερώνεται ο κωδικός λογαριασμού του χρήστη στην εφαρμογή, ενώ η πιθανότητα κλοπής της ταυτότητάς του είναι κατεσταλμένη. Τα προσωπικά δεδομένα του χρήστη προστατεύονται περαιτέρω καθώς μόνο τα απολύτως απαραίτητα στοιχεία αιτούνται από την εφαρμογή (AM και όνομα), ο χρήστης πληροφορείται ρητά για την κοινοποίηση των δεδομένων αυτών, και απαιτείται η τελική συναίνεση του για την αποστολή τους στον server.

Μετά από την επιτυχή ταυτοποίησή του, ο χρήστης έχει τη δυνατότητα να επιλέξει το ρόλο του ως οδηγός ή ως επιβάτης.

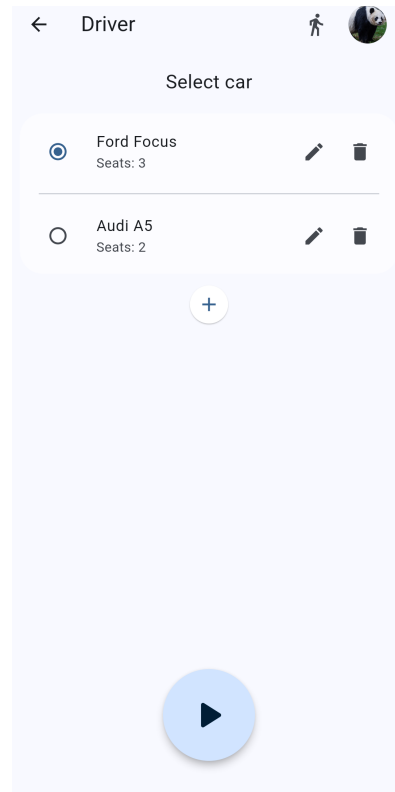
Η διαδικασία ξεκινάει όταν ο οδηγός πατήσει το κουμπί εκκίνησης, έχοντας προηγουμένως προσδιορίσει το όχημα με το οποίο μετακινείται. Η εφαρμογή αρχικά παραμένει ανενεργή, παρατηρώντας μόνο το στίγμα του οδηγού έως ότου αυτός πλησιάσει αρκετά στη στάση όπου περιμένουν οι πεζοί. Σε αυτό το σημείο η εφαρμογή επικοινωνεί με τον server, ρωτώντας για την ύπαρξη πεζών στη στάση, και στη περίπτωση που υπάρχουν όντως πεζοί η εφαρμογή ειδοποιεί τον οδηγό. Αν ο οδηγός αποδεχτεί, τότε κάποιος από τους πεζούς ορίζεται ως ο επιβάτης του.

Όσον αφορά τον πεζό, η εφαρμογή επίσης παραμένει αρχικά ανενεργή μέχρι κάποιος οδηγός να γίνει διαθέσιμος. Μόλις κάποιος οδηγός εμφανιστεί, ένας μικρός αριθμός από πεζούς (το πολύ 5) ειδοποιείται για την ύπαρξη του, και ο πρώτος από αυτούς που αποδεχτεί την πρόσκληση ορίζεται ως ο επιβάτης του οδηγού.

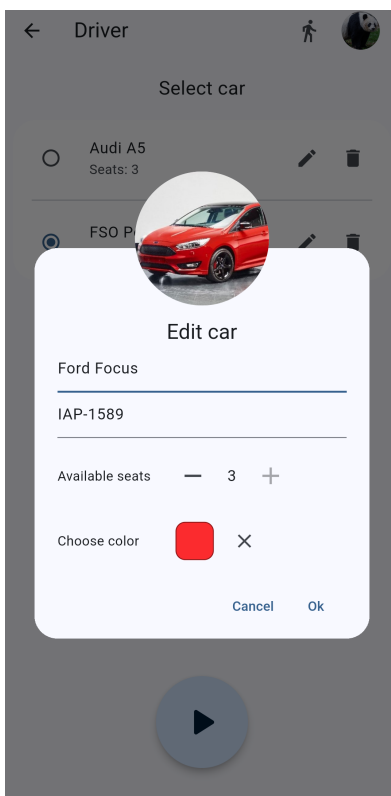
Από αυτό το σημείο και πέρα, οδηγός και επιβάτης παρουσιάζονται με τα στοιχεία του άλλου (όνομα, αριθμός πινακίδας, φωτογραφίες κτλ.) καθώς και με έναν χάρτη της περιοχής πάνω στον οποίο φαίνονται τα στίγματά τους. Ο οδηγός κατευθύνεται προς τον πεζό, και ο πεζός παρατηρεί στον χάρτη τη διαδρομή του οδηγού του. Μόλις ο οδηγός πλησιάσει στη στάση, και οι δύο τους ειδοποιούνται από την εφαρμογή ώστε ο πεζός να επιβιβαστεί στο όχημα. Σε περίπτωση που ο οδηγός αποχωρήσει μην έχοντας λάβει τον πεζό, η συνεδρία τους ακυρώνεται και η εφαρμογή του πεζού συνεχίζει να αναζητά άλλους οδηγούς. Η εφαρμογή παρακολουθεί την πορεία του οχήματος προς τη σχολή, και μόλις αυτό φτάσει στον προορισμό του, τερματίζει τη διαδρομή και δίνει την επιλογή σε οδηγό και πεζό να αξιολογήσουν ο ένας τον άλλο.



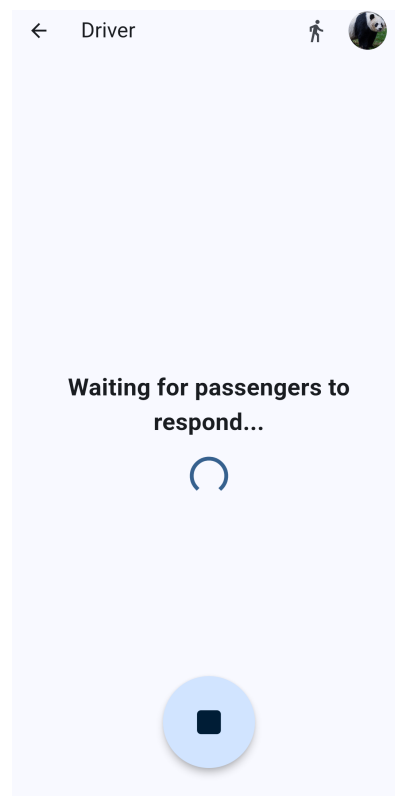
(α') Η αρχική οθόνη της εφαρμογής αφού ο χρήστης έχει συνδεθεί.



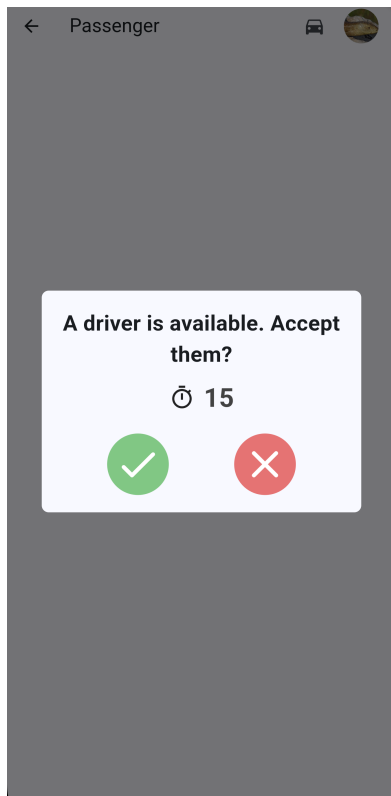
(β') Η οθόνη επιλογής οχήματος του οδηγού.



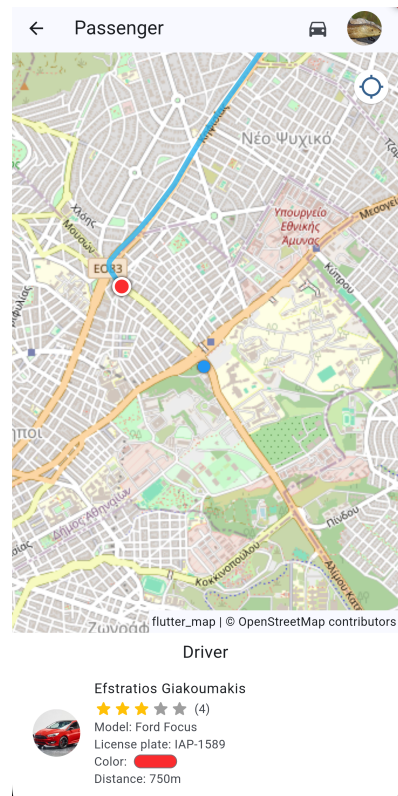
(γ') Η οθόνη τροποποίησης ενός οχήματος.



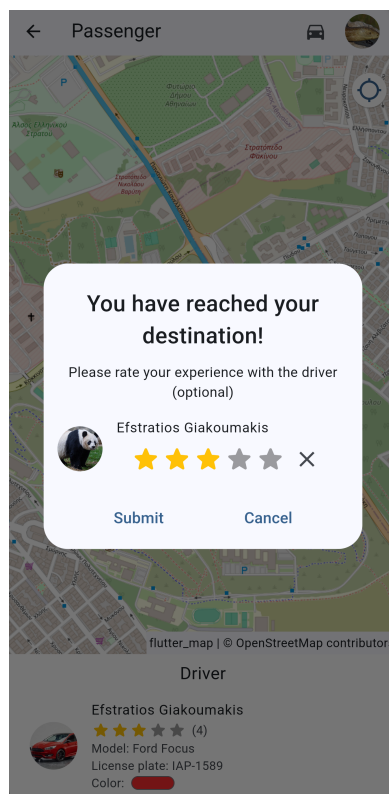
(δ') Η οθόνη όσο ο οδηγός αναμένει την εμφάνιση πεζού.



(ε') Η οθόνη του πεζού όταν ερωτάται για την αποδοχή του οδηγού.



(στ') Η οθόνη του χάρτη κατά τη διάρκεια της διαδρομής.



(ζ') Η τελική οθόνη αξιολόγησης του οδηγού.

Σχήμα 5.1: Μερικές από τις οθόνες της εφαρμογής

5.2 Παρουσίαση κώδικα

Σε αυτό το σημείο θα αναλύσουμε τον κώδικα της εφαρμογής. Ορισμένα σημεία παρουσιάζονται τροποποιημένα για λόγους απλούστευσης.

Το entry-point της εφαρμογής είναι η συνάρτηση `main`. Στη `main` αρχικοποιούμε ό,τι ρυθμίσεις χρειάζονται, και έπειτα εκκινούμε την εφαρμογή μέσω της συνάρτησης `runApp`.

```
main.dart

1  void main() {
2    runApp(const RidehailingApp());
3  }
4
5  class RidehailingApp extends StatelessWidget {
6    const RidehailingApp({super.key});
7
8    @override
9    Widget build(BuildContext context) {
10     return MultiProvider(
11       providers: [
12         ChangeNotifierProvider(create: (_) => User()),
13         ChangeNotifierProvider(create: (_) => SocketConnection()),
14         ChangeNotifierProvider(create: (_) => Authenticator())
15       ],
16       child: MaterialApp(
17         title: 'NTUA-Ridehailing',
18         theme: ThemeData(
19           colorSchemeSeed: Colors.blue,
20           useMaterial3: true,
21         ),
22         home: const WelcomePage(),
23       ),
24     );
25   }
26 }
```

Η `runApp` δέχεται ένα widget το οποίο θα αποτελέσει τη “ρίζα” του widget tree της εφαρμογής. Στην περίπτωση μας η ρίζα είναι ένα `MaterialApp`, το οποίο αποτελεί ένα βασικό wrapper widget που περιέχει έτοιμες τις κύριες λειτουργίες που χρειάζεται η εφαρμογή, όπως themes, animations και navigation, σύμφωνα με τις προδιαγραφές του Material Design της Google. Εναλλακτικά η `runApp` μπορεί να δεχτεί ένα `CupertinoApp` widget για τη δημιουργία εφαρμογής που ακολουθεί τα πρότυπα της Apple. Το `MaterialApp` με τη σειρά του περιέχει στην παράμετρο `home` την αρχική οθόνη της εφαρμογής που θα εμφανιστεί κατά την εκκίνηση, η οποία στην περίπτωση μας είναι

η `WelcomePage`.

Παρατηρούμε επίσης ότι το `MaterialApp` είναι εμφωλευμένο σε ένα widget `MultiProvider`. Αυτό το widget παρέχει τα στοιχεία του χρήστη που περιέχονται στην κλάση `User` σε όλα τα widgets της εφαρμογής που τα χρειάζονται. Χρειαζόμαστε αυτόν τον `Provider` καθώς πολλά διαφορετικά widget σε όλα τα routes της εφαρμογής εμφανίζουν κάποιο στοιχείο του χρήστη, και όλα αυτά τα widgets πρέπει να ενημερωθούν τη στιγμή που υπάρξει κάποια αλλαγή στα στοιχεία αυτά ώστε να μην παρουσιάζονται ποτέ stale δεδομένα. Για παράδειγμα, όλα τα routes της εφαρμογής εμφανίζουν στο app bar την εικόνα προφίλ του συνδεδεμένου χρήστη, και μόλις ο χρήστης αλλάξει την εικόνα προφίλ του όλα αυτά τα widget πρέπει αμέσως να ενημερωθούν με την καινούρια εικόνα. Τοποθετώντας τον `Provider` στη ρίζα του widget tree εξασφαλίζουμε ότι όλα τα widgets της εφαρμογής θα έχουν εύκολη πρόσβαση στην κλάση `User`.

Εκτός από την κλάση `User`, ο provider παρέχει επίσης τις κλάσεις `SocketConnection` και `Authenticator`, οι οποίες χειρίζονται τη σύνδεση `WebSocket` με τον server και την ταυτοποίηση του χρήστη. Ο κώδικας του `SocketConnection` βρίσκεται στο τέλος αυτού του κεφαλαίου ενώ η κλάση `Authenticator` αναλύεται στο Παράρτημα Β'.

5.2.1 Αρχική οθόνη

Η αρχική οθόνη της εφαρμογής είναι και αυτή ένα widget, και συγκεκριμένα ανήκει στην κλάση `StatefulWidget`. Όπως και οι περισσότερες οθόνες της εφαρμογής, το `WelcomePage` widget περιέχει στοιχεία UI που αλλάζουν την εμφάνισή τους ανάλογα με την κατάσταση του, επομένως πρέπει να χειρίζεται το δικό του ephemeral state. Η βάση του `WelcomePage` είναι ένα `Scaffold`, το οποίο αποτελεί το widget-υπόβαθρο για την υπόλοιπη διάταξη της οθόνης.

welcome.dart

```
1 class WelcomePage extends StatefulWidget {
2   const WelcomePage({super.key});
3
4   @override
5   State<WelcomePage> createState() => _WelcomePageState();
6 }
7
8 class _WelcomePageState extends State<WelcomePage> {
9
10   @override
```

```

11   Widget build(BuildContext context) {
12     return Scaffold(
13       body: SafeArea(
14         child: Column(
15           children: <Widget>[
16             ...
17           ],
18         ),
19       ),
20     );
21   }
22 }

```

To Scaffold περιέχει μία στήλη από τα παρακάτω widgets:

welcome.dart/children

```

1   IconButton(
2     icon: const Icon(Icons.help),
3     iconSize: 30,
4     onPressed: showHelpDialog,
5   ),
6
7   Consumer<SocketConnection>(
8     builder: (socket) => Visibility(
9       visible: socket.status == SocketStatus.connected,
10      child: const UserAvatarButton(),
11    ),
12  ),
13
14  const Text(
15    'NTUA Ridehailing',
16    style: TextStyle(fontSize: 50, fontWeight: FontWeight.w900),
17  ),
18
19  Consumer2<SocketConnection, User>(
20    builder: (socket, user) {
21      String displayText;
22      if (socket.status == SocketStatus.disconnected) {
23        displayText = 'Log in';
24      } else if (socket.status == SocketStatus.connecting ||
25        user.givenName.isEmpty) {
26        displayText = 'Connecting...';
27      } else {
28        displayText = 'Logged in as \${user.givenName}';
29      }
30      return TextButton(
31        onPressed: socket.status == SocketStatus.disconnected
32          ? _connectToServer
33          : null,

```

```

34     child: Text(
35       displayText,
36       style: const TextStyle(fontSize: 30),
37       textAlign: TextAlign.center,
38     ),
39   );
40 },
41 ),
42
43 Consumer<SocketConnection>(
44   builder: (socket) => Visibility(
45     visible: socket.status != SocketStatus.connected,
46     child: const Text('You must be logged in to use the app'),
47   ),
48 ),
49
50 Consumer<SocketConnection>(
51   builder: (context, socket, child) => Row(
52     mainAxisAlignment: MainAxisAlignment.spaceEvenly,
53     children: [
54       SubtitledButton(
55         icon: const Icon(Icons.directions_car),
56         subtitle: const Text('I am a driver'),
57         onPressed: socket.status == SocketStatus.connected
58           ? () => Navigator.push(
59             context,
60             MaterialPageRoute(
61               builder: (context) => const DriverPage(),
62             ),
63           )
64       : null,
65     ),
66       SubtitledButton(
67         icon: const Icon(Icons.directions_walk),
68         subtitle: const Text('I am a passenger'),
69         onPressed: socket.status == SocketStatus.connected
70           ? () => Navigator.push(
71             context,
72             MaterialPageRoute(
73               builder: (context) => const PassengerPage(),
74             ),
75           )
76       : null,
77     ),
78   ],
79 ),
80 ),
81
82 Consumer<SocketConnection>(
83   builder: (socket) => Visibility(
84     visible: socket.status == SocketStatus.connected,
85     child: TextButton(

```

```

86     onPressed: () async {
87       final reply = await signOutAlert(context: context);
88       if (reply) socket.setStatus(SocketStatus.disconnected);
89     },
90     child: const Text('Sign out', style: TextStyle(fontSize: 25)),
91   ),
92 ),
93 ),

```

Παρατηρούμε ότι τα περισσότερα από τα παραπάνω widgets βρίσκονται εντός ενός widget Consumer. Αυτά είναι εκείνα ακριβώς τα widgets της οθόνης τα οποία πρέπει να ενημερώνονται αμέσως όταν υπάρξουν μεταβολές στους providers SocketConnection ή User. Για παράδειγμα, το τελευταίο widget στη λίστα είναι ένα κουμπί που επιτρέπει στον χρήστη να αποσυνδεθεί από τον λογαριασμό του, και για ευνόητους λόγους επιθυμούμε να είναι εμφανές μόνο εφόσον ο χρήστης είναι ήδη συνδεδεμένος. Για αυτόν τον λόγο το widget περιλαμβάνεται από ένα Visibility το οποίο ελέγχει την ορατότητά του. Προκειμένου όμως η ορατότητα του widget να αλλάζει δυναμικά σύμφωνα με την κατάσταση της σύνδεσης, περιλαμβάνουμε επιπλέον το Visibility με έναν Consumer ο οποίος “ακούει” στο SocketConnection και ενημερώνει αναλόγως το Visibility.

Το κουμπί του login είναι ενεργοποιημένο εφόσον ο χρήστης δεν είναι ήδη συνδεδεμένος με τον server, και μόλις πατηθεί καλείται η συνάρτηση connectToServer η οποία πραγματοποιεί τη σύνδεση:

welcome.dart/connectToServer

```

1  void _connectToServer() async {
2    final socket = context.read<SocketConnection>();
3    final idToken = await Authenticator.authenticate();
4    if (idToken == null) {
5      ScaffoldMessenger
6        .of(context)
7        .showSnackBar(snackBarAuthenticationError);
8      return;
9    }
10   await socket.connect(idToken);
11   if (socket.status != SocketStatus.connected) {
12     ScaffoldMessenger
13       .of(context)
14       .showSnackBar(snackBarConnectionError);
15   }
16 }

```

Δεδομένου ότι η σύνδεση με τον server απαιτεί ένα ID Token που ταυτοποιεί τον χρήστη ως φοιτητή της σχολής, πραγματοποιούμε αρχικά το authentication με τη σχολή μέσω του AppAuth, και έπειτα δημιουργούμε μία νέα σύνδεση με τον server χρησιμοποιώντας το token που λάβαμε. Εάν αποτύχει είτε το authentication είτε η σύνδεση με τον server, ειδοποιούμε τον χρήστη με το αντίστοιχο μήνυμα σε μορφή snack bar.

Το κουμπί σύνδεσης με τον server και τα δύο κουμπιά για τη μετάβαση στη λειτουργία οδηγού/πεζού είναι τα σημαντικότερα μέρη της αρχικής οθόνης. Εφόσον η σύνδεση είναι επιτυχής, ο χρήστης μπορεί να προχωρήσει στο επόμενο μέρος της εφαρμογής πατώντας κάποιο από τα δύο κουμπιά οδηγού/πεζού, τα οποία θα τον μεταφέρουν σε μία νέα οθόνη τύπου `DriverPage` ή `PassengerPage` αντίστοιχα.

5.2.2 Λειτουργία πεζού

Θα εξετάσουμε αρχικά την οθόνη για τη λειτουργία πεζού.

```
passenger.dart

1 Widget build(BuildContext context) {
2   return Scaffold(
3     appBar: AppBar(
4       title: const Text('Passenger'),
5       actions: [
6         SwitchModeButton(),
7         UserAvatarButton(),
8       ],
9     ),
10    body: driver == null
11      ? PassengerStatusScreen()
12      : MapScreen(),
13  );
14 }
```

Στο άνω μέρος της οθόνης βρίσκεται το App Bar το οποίο διαθέτει κουμπιά για την εναλλαγή της λειτουργίας από πεζό σε οδηγό, και για την προβολή του προφίλ του χρήστη.

Αν ο πεζός αναζητάει οδηγό τότε το κύριο σώμα της οθόνης εμφανίζει την `PassengerStatusScreen` η οποία πληροφορεί τον χρήστη για την κατάσταση της αναζήτησης. Μόλις ο πεζός δεχτεί έναν οδηγό τότε το περιεχόμενο της οθόνης αλλάζει και εμφανίζει τη `MapScreen`, στην οποία παρουσιάζεται ο χάρτης με την τοποθεσία και τις πληροφορίες του οδηγού.

Ο πεζός επικοινωνεί συνεχώς με τον κεντρικό server, ανταλλάσσοντας μηνύματα σχετικά με την κατάστασή του. Επομένως, στην εφαρμογή έχει προσδιοριστεί μία συνάρτηση που δέχεται τα μηνύματα από τον server και εκτελεί τις απαραίτητες ενέργειες για την ενημέρωση της κατάστασης:

passenger.dart/socketHandler

```
1 void socketHandler(message) async {
2   final decoded = jsonDecode(message);
3   final type = decoded['type'];
4   final data = decoded['data'];
5   switch (type) {
6     case typePingPassengers:
7       HapticFeedback.heavyImpact();
8       _acceptDriverDialog();
9       break;
10    case typeUpdateDriver:
11      driver = data['driver'];
12      driverPositions.add(driver['coordinates']);
13      if (driverArrived) {
14        final driverDistance = Geolocator.distanceBetween(
15          driver['coordinates'],
16          myCoordinates,
17        );
18        if (driverDistance > maxSeperation) {
19          HapticFeedback.heavyImpact();
20          driver = null;
21          socketConnection
22            .channel
23            .add(jsonEncode({'type': typeOutOfRange, 'data': {}}));
24          break;
25        }
26      }
27      if (Geolocator.distanceBetween(driver['coordinates'], busStop)
28        ↪ <
29        arrivalRange && !driverArrived) {
30        HapticFeedback.heavyImpact();
31        driverArrived = true;
32        _showArrivedDialog();
33      }
34      break;
35    case typeArrivedDestination:
36      HapticFeedback.heavyImpact();
37      await _showFinishDialog(driver);
38      Navigator.popUntil(context, (route) => route.isFirst);
39      break;
40    }
41  }
```

Η συνάρτηση socketHandler δέχεται το μήνυμα το οποίο είναι κωδικοποιη-

μένο με τη μορφή JSON string, και αφού το αποκωδικοποιεί εξετάζει τον τύπο του και δρα αναλόγως:

- Το μήνυμα τύπου `pingPassengers` στέλνεται από τον server όταν ένας οδηγός ρωτά για τη διαθεσιμότητα του πεζού, επομένως μόλις η εφαρμογή το δεχτεί εμφανίζει στον χρήστη το παράθυρο αποδοχής του οδηγού.
- Όταν η εφαρμογή δέχεται μήνυμα `updateDriver` το οποίο περιέχει νέα δεδομένα για τον οδηγό (πχ. την τοποθεσία του), τότε ενημερώνει αρχικά την αντίστοιχη μεταβλητή του οδηγού. Αν ο οδηγός έχει μόλις φτάσει στη στάση όπου περιμένει ο πεζός τότε η εφαρμογή ειδοποιεί τον χρήστη μέσω του `showArrivedDialog`. Επιπλέον, η εφαρμογή ελέγχει συνεχώς την απόσταση του οδηγού προκειμένου να διαπιστώσει αν ο οδηγός εγκατέλειψε τον πεζό, και στην περίπτωση αυτή τερματίζει τη συνεδρία και ειδοποιεί τον server.
- Τέλος, η εφαρμογή μπορεί να δεχτεί μήνυμα `arrivedAtDestination` το οποίο σηματοδοτεί την άφιξη του πεζού στη σχολή. Σε αυτή την περίπτωση εμφανίζεται μέσω της `showFinishDialog` το αντίστοιχο παράθυρο το οποίο ειδοποιεί τον πεζό για την άφιξη στον προορισμό του, προσφέροντας παράλληλα τη δυνατότητα αξιολόγησης του οδηγού.

Ακολουθεί η οθόνη του χάρτη:

passenger.dart/MapScreen

```
1  class MapScreen extends StatelessWidget {
2    const MapScreen({super.key});
3
4    @override
5    Widget build(BuildContext context) {
6      return Column(
7        children: [
8          FlutterMap(
9            children: [
10             TileLayer(
11               urlTemplate: mapUri,
12               tileBounds: LatLngBounds(
13                 const LatLng(38.01304, 23.74121),
14                 const LatLng(37.97043, 23.80078),
15               ),
16             ),
17             PolylineLayer(
18               polylines: [
```

```

19         Polyline(points: driverPositions),
20     ],
21 ),
22     MarkerLayer(
23         markers: [
24             Marker(point: myCoordinates),
25             Marker(point: driver['coordinates']),
26         ],
27     ),
28     const SimpleAttributionWidget(
29         source: Text('OpenStreetMap contributors'),
30     ),
31 ],
32 ),
33     DriverInfoBox(driver: driver),
34 ]
35 );
36 }
37 }

```

Η MapScreen περιέχει τον χάρτη πάνω στον οποίο εμφανίζεται το στίγμα του πεζού και του οδηγού, καθώς και ένα πλαίσιο στο οποίο αναγράφονται οι σχετικές πληροφορίες του οδηγού και του οχήματός του. Στον χάρτη εμφανίζεται επίσης μία τεθλασμένη γραμμή (polyline) τα σημεία της οποίας αναπαριστούν τις προηγούμενες τοποθεσίες του οδηγού, έτσι ώστε ο χρήστης να μπορεί να παρακολουθεί την πορεία του οδηγού του να διαγράφεται σε πραγματικό χρόνο.

Το πλαίσιο πληροφοριών περιέχει τα σχετικά στοιχεία του οδηγού, συμπεριλαμβανομένης και της απόστασης του οδηγού από την στάση.

passenger.dart/DriverInfoBox

```

1  class DriverInfoBox extends StatelessWidget {
2    const DriverInfoBox({
3      super.key,
4      required this.driver,
5    });
6
7    final Map<String, dynamic> driver;
8
9    @override
10   Widget build(BuildContext context) {
11     return Container(
12       color: Colors.white,
13       child: Column(
14         children: Text(

```

```

15         'Driver',
16         style: TextStyle(fontSize: 20),
17         textAlign: TextAlign.center,
18     ),
19     ListTile(
20       leading: CachedNetworkImage(
21         imageUrl:
22           ↪ "$mediaHost/images/users/\${driver['picture']}",
23         placeholder: (context, url) => Image.asset(
24           'assets/images/blank_profile.png',
25         ),
26       ),
27       tileColor: Colors.lightBlue,
28       title: Text(driver['full_name']),
29       subtitle: Column(
30         crossAxisAlignment: CrossAxisAlignment.start,
31         children: [
32           RatingBarWithCount(driver['rating']),
33           LabelledWidget("Model:",
34             ↪ Text(driver['car']['model'])),
35           LabelledWidget("License plate:",
36             ↪ Text(driver['car']['license'])),
37           LabelledWidget(
38             "Color:",
39             ColorIndicator(color:
40               ↪ Color(driver['car']['color'])),
41           ),
42           Text("Distance: \${Geolocator.distanceBetween(
43             driver["coordinates"],
44             busStop)}m",
45         ],
46       ),
47     ),
48   );

```

Η μέθοδος `showFinishDialog` όπως αναφέραμε εμφανίζει σε οδηγό και επιβάτες ένα Dialog το οποίο τους πληροφορεί για το τέλος της διαδρομής, και τους δίνει την ευκαιρία να αξιολογήσουν ο ένας τον άλλον. Ο επιβάτης μπορεί να επιλέξει τον αριθμό αστεριών στο `RatingBar` που αντιστοιχεί στον οδηγό, και αφού πατήσει το κουμπί υποβολής όλες οι τιμές στέλνονται στον server για καταμέτρηση.

showFinishDialog

```

1 Future<void> _showFinishDialog({
2   required BuildContext context,
3   required List<User> users,
4 }) async {
5   final socket = context.read<SocketConnection>();
6   List<ValueNotifier<double>> ratings = List.generate(
7     users.length,
8     (index) => ValueNotifier(0),
9   );
10  List<Widget> ratingBars = List.generate(
11    users.length,
12    (index) => ValueListenableBuilder(
13      valueListenable: ratings[index],
14      builder: (context, value, child) {
15        return Row(
16          children: [
17            RatingBar.builder(
18              initialRating: value,
19              minRating: 1,
20              itemBuilder: (context, index) =>
21                const Icon(Icons.star_rounded, color:
22                  ⇨ Colors.amber),
23              onRatingUpdate: (newRating) {
24                ratings[index].value = newRating;
25              },
26            ),
27          ],
28        );
29      },
30    );
31
32  final reply = await showDialog<bool>(<
33    context: context,
34    builder: (context) => AlertDialog(
35      content: Column(
36        children: [
37          Text('You have reached your destination!'),
38          Text('Please rate your experience with the passengers'),
39          ListView.separated(
40            itemCount: users.length,
41            itemBuilder: (context, index) {
42              return ListTile(
43                title: Text(users[index]['full_name']),
44                leading: UserAvatar(url: users[index]['picture']),
45                subtitle: ratingBars[index],
46              );
47            },
48          ),
49        ],

```

```

50     ),
51     actions: [
52       TextButton(
53         onPressed: () => Navigator.pop(context, true),
54         child: const Text('Submit'),
55       ),
56       TextButton(
57         onPressed: () => Navigator.pop(context, false),
58         child: const Text('Cancel'),
59       ),
60     ]
61   ),
62 );
63
64 if(reply == false) return;
65
66 final ratingList = ratings.map((e) => e.value).toList();
67
68 socket.channel.add(
69   jsonEncode({
70     'type': typeSendRatings,
71     'data': ratingList,
72   }),
73 );
74 }

```

5.2.3 Λειτουργία οδηγού

Τώρα θα αναλύσουμε των κώδικα της οθόνης του οδηγού.

driver.dart

```

1  class DriverPage extends StatefulWidget {
2    const DriverPage({super.key});
3
4    @override
5    State<DriverPage> createState() => _DriverPageState();
6  }
7
8  class _DriverPageState extends State<DriverPage> {
9    Widget build(BuildContext context) {
10      Widget driverScreen;
11      if (!driving) {
12        driverScreen = CarListScreen(
13          carList: CarList(
14            selected: selectedCar,
15            onEditPressed: (id) => _editCar(id),
16          ),
17          onPressed: context.watch<User>().cars.length < 3
18            ? () => _createCar()

```

```

19         : null,
20     );
21     } else if (passengers.isEmpty) {
22         driverScreen = DriverStatusScreen();
23     } else {
24         driverScreen = MapScreen(
25             passengers: passengers,
26             coordinates: myCoordinates,
27         );
28     }
29
30     return Scaffold(
31         appBar: AppBar(
32             title: const Text('Driver'),
33             actions: [
34                 SwitchModeButton(),
35                 UserAvatarButton(),
36             ],
37         ),
38         body: driverScreen,
39         floatingActionButton: Visibility(
40             visible: selectedCar.value != null && passengers.isEmpty,
41             child: Align(
42                 alignment: Alignment.bottomCenter,
43                 child: LargeFAB(
44                     driving: driving,
45                     onPressed: _toggleDriving,
46                     tooltip: driving ? 'Stop driving' : 'Start driving',
47                 ),
48             ),
49         ),
50     );
51 }
52 }

```

Η οθόνη του οδηγού είναι όμοια με την οθόνη του πεζού, με τη διαφορά ότι προστίθεται η `CarListScreen`, μία επιπλέον όψη που επιτρέπει στον χρήστη να επιλέξει το όχημά του πριν την έναρξη της διαδρομής. Η `CarListScreen` συγκεκριμένα περιέχει το widget `CarList` το οποίο εμφανίζει τα οχήματα του χρήστη σε μία λίστα, επιτρέποντας την επιλογή, τροποποίηση και διαγραφή τους:

driver.dart/CarList

```

1     class CarList extends StatelessWidget {
2     const CarList({
3         super.key,
4         required this.selected,

```

```

5     required this.onEditPressed,
6   });
7
8   final ValueNotifier<String?> selected;
9   final Future<void> Function(String) onEditPressed;
10
11   @override
12   Widget build(BuildContext context) {
13     final cars = context.watch<User>().cars;
14     return ListView.separated(
15       shrinkWrap: true,
16       itemCount: cars.length,
17       itemBuilder: (context, index) {
18         return ListTile(
19           shape: RoundedRectangleBorder(
20             borderRadius: BorderRadius.circular(24),
21           ),
22           onTap: () => selected.value = index,
23           title: Text(cars[index].model),
24           subtitle: Align(
25             alignment: Alignment.centerLeft,
26             child: LabelledWidget("Seats:",
27               ↪ Text(cars[index].seats)),
28           ),
29           leading: Radio<String>(
30             value: index,
31             groupValue: selected.value,
32             onChanged: (value) => selected.value = value,
33           ),
34           trailing: Row(
35             mainAxisAlignment: MainAxisAlignment.min,
36             children: [
37               IconButton(
38                 icon: const Icon(Icons.edit),
39                 tooltip: 'Edit',
40                 onPressed: () => onEditPressed(index),
41               ),
42               IconButton(
43                 icon: const Icon(Icons.delete),
44                 tooltip: 'Delete',
45                 onPressed: () async {
46                   final reply = await showDeleteCarDialog(context);
47                   if (reply == true) {
48                     socket.channel.add(
49                       jsonEncode({
50                         'type': typeRemoveCar,
51                         'data': index,
52                       })),
53                     );
54                 }
55             ],
56           ),
57         );
58       },
59     );
60   }

```

```

56         ],
57     ),
58 );
59 },
60 );
61 }
62 }

```

Έπειτα παρουσιάζουμε την συνάρτηση που επιτρέπει στον χρήστη να προσθέσει οχήματα στον λογαριασμό του:

driver.dart/createCar

```

1 Future<void> _createCar() async {
2   ValueNotifier<String?> picture;
3   ValueNotifier<Color?> color = ValueNotifier(null);
4   ValueNotifier<int> seats = ValueNotifier<int>(2);
5
6   final dialogChildren = [
7     Text('Create a car'),
8     Autocomplete<String>(
9       fieldViewBuilder: (textEditingController) {
10        return TextFormField(
11          controller: textEditingController,
12          decoration: const InputDecoration(
13            hintText: 'Car model',
14          ),
15          validator: (value) => suggestions.contains(value)
16            ? null
17            : 'Please select a valid car model',
18          onEditingComplete: () =>
19            _modelNameController.text =
20              ⇨ textEditingController.text,
21        );
22      },
23      optionsBuilder: (textEditingValue) {
24        return suggestions.where(
25          (element) => element.contains(textEditingValue.text()),
26        );
27      },
28      onSelected: (option) => _modelNameController.text = option,
29    ),
30    TextFormField(
31      controller: _licensePlateController,
32      decoration: const InputDecoration(hintText: 'License plate'),
33      validator: (value) =>
34        value == null ||
35        ⇨ licensePlateRegex.hasMatch(value.toUpperCase())
36          ? null
37          : 'Please enter a valid license plate number',

```

```

36     ),
37     Row(
38       children: [
39         const Text('Available seats'),
40         ValueListenableBuilder(
41           valueListenable: seats,
42           builder: (context, value, child) {
43             return IconButton(
44               onPressed: value > 1 ? () => --seats.value : null,
45               icon: const Icon(Icons.remove),
46               iconSize: 30,
47             );
48           },
49         ),
50         ValueListenableBuilder(
51           valueListenable: seats,
52           builder: (context, value, child) => Text('\$value'),
53         ),
54         ValueListenableBuilder(
55           valueListenable: seats,
56           builder: (context, value, child) => IconButton(
57             onPressed: value < 3 ? () => ++seats.value : null,
58             icon: const Icon(Icons.add),
59             iconSize: 30,
60           ),
61         ),
62       ],
63     ),
64     Row(
65       children: [
66         const Text('Choose color'),
67         ColorPickerPopover(colorNotifier: color),
68       ],
69     ),
70     Row(
71       mainAxisAlignment: MainAxisAlignment.end,
72       children: [
73         TextButton(
74           child: const Text('Cancel'),
75           onPressed: () => Navigator.pop(context),
76         ),
77         TextButton(
78           child: const Text('Submit'),
79           onPressed: () async {
80             final modelName = _modelNameController.text;
81             final licensePlate = _licensePlateController.text;
82             Navigator.pop(
83               context,
84               ({
85                 'model': modelName,
86                 'license': licensePlate,
87                 'seats': seats,

```

```

88         'picture': picture.value,
89         'color': color,
90     }},
91 );
92 },
93 ),
94 ],
95 ),
96 ];
97
98 final car = await showDialog(
99     context: context,
100     barrierDismissible: false,
101     builder: (context) {
102         return Column(
103             children: [
104                 ValueListenableBuilder(
105                     valueListenable: picture,
106                     builder: (context, picture, child) {
107                         return ImageButton(
108                             imageUrl: picture,
109                             onTap: () async {
110                                 final imagePath = await pickImage();
111                                 if (imagePath == null) return;
112                                 picture.value = imagePath;
113                                 color.value = getAverageColor(File(imagePath));
114                             },
115                         );
116                     },
117                 ),
118                 ...dialogChildren
119             ]
120         );
121     },
122 );
123
124 if (car != null) {
125     socketConnection.send(
126         jsonEncode({'type': typeAddCar, 'data': car}),
127     );
128 }
129 }

```

Όταν η `createCar` καλείται, εμφανίζεται στην οθόνη ένα Dialog το οποίο περιέχει τη φόρμα στην οποία ο χρήστης μπορεί να εισάγει τα στοιχεία του οχήματος.

Το μοντέλο επιλέγεται ανάμεσα από μία λίστα οχημάτων που εμφανίζονται στον χρήστη μέσω του autocomplete. Αυτό γίνεται όχι μόνο για χάρη διευκόλυνσης του χρήστη, αλλά και για να εξασφαλίσουμε ότι το πεδίο θα περιέχει

έγκυρες τιμές. Το πεδίο του αριθμού πινακίδας ελέγχει επίσης την εγκυρότητα της τιμής που εισάγει ο χρήστης, συμβάλλοντας στην αποφυγή σφαλμάτων κατά την εισαγωγή. Ο χρήστης ενθαρρύνεται επίσης να ανεβάσει μία φωτογραφία του οχήματός του ώστε να διευκολύνει την αναγνώρισή του από τους πεζούς. Εναλλακτικά, αν δεν διατίθεται φωτογραφία, ο χρήστης έχει τη δυνατότητα να προσδιορίσει το χρώμα του οχήματος επιλέγοντας από μία παλέτα χρωμάτων. Αν κάποιο από τα πεδία της φόρμας δεν είναι έγκυρο, ο χρήστης ενημερώνεται ώστε να διορθώσει τυχών λάθη.

Μόλις ολοκληρωθεί η εισαγωγή των στοιχείων και πατηθεί το κουμπί υποβολής, όλες οι παραπάνω πληροφορίες στέλνονται στον server, και το όχημα προστίθεται στην λίστα οχημάτων του χρήστη.

Παρατηρούμε ότι στο παραπάνω κομμάτι κώδικα έχουμε κάνει χρήση της κλάσης `ValueNotifier` και του widget `ValueListenableBuilder`. Τα αντικείμενα αυτά εξυπηρετούν παρόμοιο σκοπό με τους `Provider` και `Consumer` για τους οποίους κάναμε λόγο προηγουμένως: τα widget τα οποία περιλαμβάνονται από έναν `ValueListenableBuilder` “ακούνε” έναν δεδομένο `ValueNotifier`, και ενημερώνουν αναλόγως την εμφάνισή τους μόλις παρουσιαστούν αλλαγές στην κατάσταση του.

Έτσι για παράδειγμα το widget που εμφανίζει την εικόνα του οχήματος θα αλλάξει αμέσως μόλις μία νέα εικόνα επιλεγεί από τον χρήστη, αφού ο `ValueListenableBuilder` ακούει στον `ValueNotifier picture`. Ο λόγος που δεν χρησιμοποιούμε `Providers` και σε αυτή την περίπτωση είναι ότι οι συγκεκριμένες τιμές που θέλουμε να ενημερώνονται (`picture`, `seats`, `color`) είναι περιορισμένες στο widget αυτό, επομένως η χρήση μεμονωμένων `ValueNotifier` όχι μόνο είναι αρκετή για τον σκοπό μας, αλλά απλουστεύει και τον κώδικα.

Ένα άλλο σημαντικό κομμάτι για τη λειτουργία της εφαρμογής είναι αυτό που παρακολουθεί το στίγμα του χρήστη και ενημερώνει τον server για της θέση του οχήματος:

driver.dart/onPositionChanged

```
1 void init(){
2   Geolocator.getPositionStream(
3     locationSettings: const LocationSettings(
4       accuracy: LocationAccuracy.high,
5       distanceFilter: 20,
6     ),
```

```

7      ).listen(_onPositionChanged);
8  }
9
10 void _onPositionChanged(Position? coordinates) async {
11     socketConnection.send(
12         jsonEncode({
13             'type': typeUpdateDriver,
14             'data': {
15                 'car': user.cars[selectedCar.value],
16                 'coordinates': coordinates,
17             },
18         }),
19     );
20     if(passengers.isEmpty) return;
21     if (!arrivedAtBusStop &&
22         Geolocator.distanceBetween(coordinates, busStop) < 100
23     ) {
24         arrivedAtBusStop = true;
25         _showArrivedDialog();
26         return;
27     }
28     if (!arrivedAtUniversity &&
29         Geolocator.distanceBetween(coordinates, university) < 300
30     ) {
31         arrivedAtUniversity = true;
32         _getPassengersStreamSubscription.cancel();
33         positionStream.cancel();
34         socketConnection.send(
35             jsonEncode({
36                 'type': typeArrivedDestination,
37                 'data': {}
38             })
39         );
40         await _showFinishDialog(
41             context: context,
42             users: passengers,
43         );
44         Navigator.popUntil(context, (route) => route.isFirst);
45         return;
46     }
47 }

```

Το plugin Geolocator μας δίνει πρόσβαση σε μία ροή/stream συντεταγμένων που προέρχονται από το GPS της συσκευής και αντιστοιχούν στο στίγμα του χρήστη. Κάθε φορά που ενημερώνεται το στίγμα της συσκευής καλείται η μέθοδος `onPositionChanged`.

Η `onPositionChanged` αρχικά στέλνει το νέο στίγμα στον server προκειμένου η πληροφορία αυτή να μεταφερθεί στους πεζούς που παρακολουθούν τον οδηγό. Όταν ο οδηγός φτάσει στην στάση λεωφορείου τότε εμφανίζεται

το μήνυμα `showArrivedDialog` που προτρέπει τον οδηγό να σταματήσει ώστε να επιβιβαστούν οι πεζοί. Όταν ο οδηγός φτάσει στην σχολή τότε η εφαρμογή ενημερώνει τον server για την άφιξη και εμφανίζει το παράθυρο αξιολόγησης των επιβατών, πριν επιστρέψει τελικά στην αρχική οθόνη.

5.2.4 Σύνδεση WebSocket

Τέλος, παραθέτουμε τον κώδικα της κλάσης `SocketConnection`.

Η κλάση περιέχει το αντικείμενο `WebSocket channel` το οποίο αναπαριστά τη σύνδεση με τον server. Η σύνδεση δημιουργείται μέσω της μεθόδου `create`, η οποία δέχεται ως παράμετρο το ID Token που έχει ληφθεί έπειτα από την επιτυχή ταυτοποίηση του χρήστη. Τα εισερχόμενα δεδομένα που φτάνουν στο `WebSocket` από τον server οδηγούνται μέσω ενός `Stream` στο υπόλοιπο πρόγραμμα όπου μπορούν να επεξεργαστούν. Η εφαρμογή μπορεί να στείλει μηνύματα προς τον server μέσω της μεθόδου `send`.

SocketConnection

```
1  class SocketConnection with ChangeNotifier {
2    late WebSocket channel;
3    final receivingController = StreamController<String>();
4    late final StreamSubscription<String> receivingSubscription;
5    SocketStatus status = SocketStatus.disconnected;
6
7    SocketConnection() {
8      receivingSubscription =
9        ↪ receivingController.stream.listen((event) {});
10   }
11
12   Future<void> connect(String token) async {
13     setStatus(SocketStatus.connecting);
14     WebSocket? connectionResult;
15     try {
16       connectionResult = await WebSocket.connect(
17         apiHost,
18         headers: {'authentication': token},
19       ).timeout(const Duration(seconds: 10));
20     } catch (error) {
21       setStatus(SocketStatus.disconnected);
22       return;
23     }
24     setStatus(SocketStatus.connected);
25     channel = connectionResult;
26     channel.listen(
27       (data) => receivingController.add(data),
28       onDone: () => _onDone(),
29     );
30   }
31 }
```

```
28     onError: (error) => _onError(error),
29   );
30 }
31
32 void send(dynamic data){
33   channel.add(data);
34 }
35
36 void _onDone() {
37   setStatus(SocketStatus.disconnected);
38 }
39
40 void _onError(error) {
41   setStatus(SocketStatus.disconnected);
42 }
43
44 void setStatus(SocketStatus newStatus) {
45   status = newStatus;
46   notifyListeners();
47 }
48 }
```


Κεφάλαιο 6

Backend

Η φύση της εφαρμογής επιβάλλει την ύπαρξη ενός κεντρικού server ο οποίος συντονίζει τις διάφορες λειτουργίες. Οι πεζοί και οι οδηγοί χρειάζεται να επικοινωνούν μεταξύ τους σε πραγματικό χρόνο, και στην επικοινωνία αυτή μεσολαβεί ο server. Ο κάθε χρήστης στέλνει όλα τα μηνύματα του στον server, ο server με τη σειρά του τα επεξεργάζεται αλλάζοντας αναλόγως την εσωτερική του κατάσταση, και έπειτα προωθεί περαιτέρω μηνύματα στους απαραίτητους χρήστες.

Για την εφαρμογή του server χρησιμοποιήσαμε το Node.js. Η εφαρμογή αποτελείται από δύο υπηρεσίες:

- την κύρια υπηρεσία API που χειρίζεται όλη την εσωτερική λογική και με την οποία οι χρήστες επικοινωνούν
- την υπηρεσία που χειρίζεται τα αρχεία πολυμέσων, και συγκεκριμένα τις εικόνες που ανεβάζουν οι χρήστες

Ο διαχωρισμός αυτός έγινε καθώς διαμερίζοντας τον server σε ειδικευμένα μέρη μπορούμε να πετύχουμε καλύτερη επίδοση. Αποθηκεύοντας ξεχωριστά όλα τα αρχεία πολυμέσων (αρχεία σχετικά μεγάλου μεγέθους) και μεταφέροντας την αρμοδιότητα διαχείρισής τους σε άλλο server, ελαφραίνουμε τον φόρτο από την κύρια υπηρεσία και πετυχαίνουμε μικρότερο latency. Ακόμα και στην περίπτωση που οι δύο υπηρεσίες βρίσκονται στον ίδιο server, έχουμε καλύτερη απόδοση καθώς οι δύο διεργασίες του Node.js είναι ανεξάρτητες, και επομένως μπορούν να εκτελούνται σε διαφορετικά thread/πυρήνες CPU, με την κάθε διεργασία να έχει το δικό της event loop.

Για τον web server χρησιμοποιήσαμε το Nginx, κυρίως ως reverse proxy. Το Nginx βρίσκεται “μπροστά” από το Node.js και δέχεται πρώτο όλα τα requests που φτάνουν στον server.

Μέσω του reverse proxy ανακατευθύνουμε τα requests στη θύρα της υπηρεσίας API ή της υπηρεσίας πολυμέσων ανάλογα με το URL, καταφέροντας με αυτόν τον τρόπο να έχουμε το ίδιο domain και για τους δύο server. Ο server Nginx είναι επίσης υπεύθυνος για την αποστολή των στατικών αρχείων στον χρήστη (πχ. τα αρχεία εικόνων), μία λειτουργία στην οποία υπερέχει του Node.js ως προς την ταχύτητα. Τέλος, μέσω του Nginx ενσωματώνουμε την κρυπτογράφηση SSL/TLS για την ασφαλή επικοινωνία των χρηστών με τον server μέσω HTTPS, με πιστοποιητικό που λαμβάνουμε από τη γνωστή αρχή πιστοποίησης *Let's Encrypt*.

Τα δεδομένα των χρηστών πρέπει να συγχρονίζονται μεταξύ των συσκευών στις οποίες συνδέονται, επομένως είναι απαραίτητη η αποθήκευσή τους στον server. Για τον σκοπό αυτόν χρησιμοποιούμε μία σχεσιακή βάση δεδομένων SQL, και συγκεκριμένα το λογισμικό MySQL. Η βάση αποτελείται από δύο tables, ένα για τις πληροφορίες των χρηστών και ένα για τα οχήματα. Συγκεκριμένα οι πληροφορίες που συγκεντρώνονται στη βάση για τον κάθε χρήστη αποτελούνται από τον αριθμό μητρώου του (ο οποίος λειτουργεί ως το στοιχείο ταυτοποίησης), τη βαθμολογία του, και προαιρετικά μία φωτογραφία. Όσον αφορά τα οχήματα, αποθηκεύεται ο AM του χρήστη στον οποίο αυτό ανήκει, το μοντέλο, και ο αριθμός πινακίδας του, ενώ προαιρετικά μπορεί να συμπεριληφθεί το χρώμα και μία φωτογραφία. Έχοντας αποθηκεύσει τα δεδομένα στη βάση, κάθε φορά που ένας χρήστης συνδέεται στην εφαρμογή τα υπάρχοντα στοιχεία του στέλνονται από τον server στη συσκευή του, ενώ ό,τι νέα στοιχεία προκύψουν κατά τη χρήση της εφαρμογής στέλνονται στον server για την ανανέωση της βάσης. Περισσότερες πληροφορίες για τη βάση δεδομένων βρίσκονται στο Παράρτημα Γ’

Η επικοινωνία client-server επιτυγχάνεται μέσα από την τεχνολογία WebSocket. Ο χρήστης της εφαρμογής, αφού ταυτοποιηθεί ως φοιτητής από την υπηρεσία της σχολής, ανοίγει σύνδεση WebSocket με τον server. Η σύνδεση αυτή διαρκεί για όλη τη διάρκεια της συνεδρίας, και τερματίζεται μόλις ο χρήστης αποσυνδεθεί. Το πρωτόκολλο WebSocket χρησιμοποιήθηκε καθώς αξιολογήθηκε ως το καταλληλότερο για τις απαιτήσεις της εφαρμογής ως προς τη real-time επικοινωνία, αφού παρέχει άμεση και αμφίδρομη επικοινωνία με

μικρές καθυστερήσεις και προσφέρει εγγυήσεις για την επιτυχή αποστολή των πακέτων χωρίς σφάλματα.

Όπως αναφέραμε προηγουμένως, το backend αποτελείται από δύο ανεξάρτητα μέρη, έναν κλασσικό HTTP server για τα αρχεία πολυμέσων, και έναν WebSocket server για το API.

Ο media server ανταποκρίνεται απλά σε GET/POST/DELETE requests με τον ευνόητο τρόπο, επιστρέφοντας/ανεβάζοντας/διαγράφοντας αρχεία εικόνων αντιστοίχως. Το μόνο στοιχείο που αξίζει να σημειωθεί είναι η ύπαρξη ελέγχων για την προστασία έναντι κακόβουλων ενεργειών, συγκεκριμένα η ύπαρξη ανώτατου ορίου των 2MB για τα αρχεία που ανεβάζονται¹, η απαγόρευση αρχείων που δεν έχουν τύπο εικόνας PNG/JPEG, και φυσικά η απαγόρευση ανεβάσματος αρχείων από χρήστες που δεν είναι πιστοποιημένοι.

Η λειτουργία του API server είναι πιο ενδιαφέρουσα. Για τη σύνδεση μέσω WebSocket απαιτείται πρώτα η αποστολή ενός HTTP upgrade request από τον client προς τον server, ακολουθούμενη από την αποδοχή του αιτήματος από τον server (opening handshake). Σε αυτή την εναρκτήρια αίτηση, ο χρήστης στέλνει με τη μορφή JWT (JSON Web Token) τα στοιχεία της ταυτότητάς του όπως αυτά παραχωρήθηκαν από τον Identity Provider της σχολής, και ο server ελέγχει την εγκυρότητά τους. Αν για οποιονδήποτε λόγο ο server κρίνει την ταυτότητα του χρήστη ως μη έγκυρη², τότε η σύνδεση τερματίζεται αμέσως. Αν η ταυτότητα του χρήστη είναι έγκυρη, τότε η σύνδεση επιτυγχάνεται και ο χρήστης έχει πλέον τη δυνατότητα να επικοινωνήσει με το API του server μέσω της εφαρμογής.

Κατά την επιτυχή σύνδεση του χρήστη, ο server αρχικά αναζητά τον χρήστη στη βάση δεδομένων. Αν ο χρήστης συνδέεται στην εφαρμογή για πρώτη φορά και επομένως δεν υπάρχει ο αντίστοιχος λογαριασμός στη βάση, τότε δημιουργείται ένας νέος άδειος λογαριασμός· διαφορετικά ο server διαβάζει τα αποθηκευμένα στοιχεία από τη βάση και τα στέλνει πίσω στον χρήστη.

Από το σημείο αυτό και πέρα, ο server αναμένει την αποστολή μηνυμάτων από την εφαρμογή, εκτελώντας τις ανάλογες λειτουργίες κατά τη λήψη τους,

¹ Στην πραγματικότητα το όριο των 2MB επιβάλλεται από το Nginx για όλα τα αιτήματα που φτάνουν στον server

² Η ταυτότητα ενός χρήστη κρίνεται ως μη έγκυρη αν το JWT απουσιάζει εντελώς ή έχει λήξει, αν δεν περιέχει όλα τα ταυτοποιητικά στοιχεία που είναι απαραίτητα, ή αν η υπογραφή του δεν αντιστοιχεί στον Identity Provider της σχολής

και στέλνοντας ενδεχομένως άλλα μηνύματα πίσω στον χρήστη. Το κάθε μήνυμα που ανταλλάσσεται μεταξύ εφαρμογής και server είναι ένα αντικείμενο JSON το οποίο περιέχει δύο πεδία: τον τύπο και τα δεδομένα. Ο τύπος του κάθε μηνύματος υποδεικνύει την ενέργεια που επιθυμεί να λάβει ο χρήστης, και ανήκει σε μία προκαθορισμένη λίστα που γνωρίζει ο server και η εφαρμογή. Στα δεδομένα του μηνύματος συμπεριλαμβάνονται ό,τι άλλες πληροφορίες είναι απαραίτητες για τον πλήρη προσδιορισμό της ενέργειας που πρέπει να εκτελεστεί.

Παρακάτω περιγράφουμε όλους τους τύπους των μηνυμάτων μαζί με τη λειτουργία τους.

Νέος πεζός/οδηγός: Στέλνεται από τον χρήστη μόλις αυτός συνδέεται στην εφαρμογή ως πεζός ή οδηγός. Μαζί με το μήνυμα στέλνονται οι συντεταγμένες του χρήστη, καθώς και το όχημα του αν πρόκειται για οδηγό. Ο server προσθέτει τον χρήστη στη λίστα των πεζών ή στη λίστα των οδηγών, και ειδοποιεί τον χρήστη για την επιτυχή προσθήκη του στην υπηρεσία.

Παύση πεζού/οδηγού: Στέλνεται από τον χρήστη όταν επιθυμεί να διακόψει την υπηρεσία. Ο πεζός ή οδηγός αφαιρείται από την αντίστοιχη λίστα. Σε περίπτωση που ο χρήστης βρίσκεται εν μέσω διαδρομής τότε αυτή ακυρώνεται, και ειδοποιούνται οι χρήστες που πρέπει: αν πρόκειται για παύση οδηγού τότε ενημερώνονται όλοι οι επιβάτες του, ενώ αν πρόκειται για παύση επιβάτη τότε ενημερώνεται ο οδηγός του.

Ενημέρωση πεζού/οδηγού: Στέλνεται κάθε φορά που ένας ενεργός χρήστης μετακινείται. Στο μήνυμα περιλαμβάνεται και το στίγμα του χρήστη. Ο server ενημερώνει τις συντεταγμένες που έχει αποθηκεύσει για τον χρήστη, και ειδοποιεί τους άλλους συμμετέχοντες για τη νέα θέση του.

Ερώτηση για ύπαρξη πεζών: Στέλνεται από έναν νέο οδηγό όταν αυτός επιθυμεί να μάθει αν υπάρχουν διαθέσιμοι πεζοί που αναμένουν στη στάση. Ο server απαντάει καταφατικά αν υπάρχουν όντως πεζοί.

Αναζήτηση επιβατών: Στέλνεται από έναν οδηγό όταν αυτός δηλώνει την επιθυμία του να δεχτεί επιβάτες για μεταφορά. Ο server επιλέγει έναν αριθμό από πεζούς που βρίσκονται στη στάση και τους στέλνει αίτημα για την αποδοχή του οδηγού.

Αποδοχή οδηγού: Στέλνεται από έναν πεζό ο οποίος αποδέχεται τον οδηγό του. Ο server επιβεβαιώνει την ανάθεση του οδηγού στον πεζό, και ενημερώνει τον οδηγό για τον νέο επιβάτη του.

Εγκατάλειψη πεζού: Στέλνεται από έναν πεζό στον οποίο έχει ανατεθεί οδηγός, αλλά του οποίου ο οδηγός προσπέρασε τη στάση χωρίς να τον παραλάβει. Ο server αφαιρεί τον πεζό από τους επιβάτες του οδηγού και αρχίζει αμέσως να αναζητεί νέο οδηγό για εκείνον.

Άφιξη στον προορισμό: Στέλνεται από τον οδηγό όταν φτάσει στη σχολή με τους επιβάτες του. Ο server τερματίζει τη διαδρομή αφαιρώντας τον οδηγό και τους επιβάτες από τις αντίστοιχες λίστες.

Αποστολή αξιολόγησης: Στέλνεται από χρήστη που επιθυμεί να βαθμολογήσει την εμπειρία του με έναν οδηγό ή πεζό. Τα δεδομένα περιλαμβάνουν το AM του υπό αξιολόγηση χρήστη μαζί με την τιμή της βαθμολογίας. Ο server ελέγχει την εγκυρότητα των στοιχείων³ και ενημερώνει την τιμή της βαθμολογίας του αντίστοιχου χρήστη στη βάση δεδομένων.

Προσθήκη/ενημέρωση/αφαίρεση οχήματος: Στέλνεται από χρήστη που επιθυμεί να αλλάξει ένα από τα διαθέσιμα οχήματά του. Τα δεδομένα αποτελούν όλα τα στοιχεία του εν λόγω οχήματος, τα οποία ο server ελέγχει πριν προσθέσει το όχημα στη βάση δεδομένων.

Προσθήκη/ενημέρωση/αφαίρεση φωτογραφίας: Στέλνεται από χρήστη που επιθυμεί να αλλάξει τη φωτογραφία στο προφίλ του. Ο server κατεβάζει τη φωτογραφία στο δίσκο και αφού ελέγξει την καταλληλότητά της μέσω ενός φίλτρου/neural network, τη θέτει ως τη φωτογραφία του χρήστη στη βάση δεδομένων.

Αποσύνδεση: Στέλνεται από τον χρήστη όταν αυτός κλείνει την εφαρμογή ή αποσυνδέεται από τον λογαριασμό του. Ο server διακόπτει την τρέχουσα διαδρομή του χρήστη και τερματίζει τη σύνδεσή του.

³Συγκεκριμένα ελέγχεται αν η τιμή της βαθμολογίας κυμαίνεται μεταξύ του 1 και του 5, και το αν έχει ως στόχο τον οδηγό/επιβάτη με τον οποίο ο χρήστης μόλις ολοκλήρωσε τη διαδρομή του.

6.1 Παρουσίαση κώδικα

Τώρα θα αναλύσουμε τον κώδικα του κεντρικού server που συντονίζει την επικοινωνία οδηγών και επιβατών.

Καθ'όλη τη λειτουργία του server θα χρειάζεται να αναφερόμαστε σε χρήστες και να προσπελάζουμε τα στοιχεία τους δεδομένου του αριθμού μητρώου τους. Για αυτόν τον λόγο δημιουργούμε τις παρακάτω associative arrays με κλειδιά τους AM των χρηστών.

api.ts

```
1 let socketArray = new Map<string, WebSocket>();
2 let driverArray = new Map<string, Driver>();
3 let passengerArray = new Map<string, Passenger>();
```

Η socketArray περιέχει τις συνδέσεις WebSocket των χρηστών, και είναι απαραίτητη για να μπορούμε να στέλνουμε μηνύματα σε αυθαίρετους χρήστες.

Οι driverArray και passengerArray περιέχουν τους τρέχοντες οδηγούς και πεζούς αντίστοιχα.

Το πρώτο βήμα είναι η δημιουργία του WebSocket server ο οποίος θα ανταποκρίνεται στα αιτήματα των χρηστών. Δεδομένου ότι επιθυμούμε η ταυτοποίηση του κάθε χρήστη να γίνεται πριν από την έναρξη της σύνδεσης, χρειάζεται να ορίσουμε έναν κλασσικό HTTP server ο οποίος θα δεχτεί το αρχικό αίτημα και θα ανοίξει τη σύνδεση WebSocket μόνο αν η ταυτοποίηση είναι επιτυχής.

api.ts

```
1 const httpServer = createServer();
2 const wss = new WebSocketServer({ noServer: true });
3
4 httpServer.on("upgrade", async (req, socket, head) => {
5   let credentials = await authenticate(req);
6   if (!credentials) {
7     socket.write("HTTP/1.1 401 Unauthorized\r\n\r\n");
8     socket.destroy();
9     loggerMain.warn("Error authenticating user");
10    return;
11  }
12
13  wss.handleUpgrade(req, socket, head, (ws) => {
14    wss.emit("connection", ws, req, credentials);
15  });
```

```

16  });
17
18  httpServer.listen(env.API_PORT);

```

Ο `httpServer` δέχεται requests για upgrade του πρωτοκόλλου σε WebSocket και εκτελεί την ταυτοποίηση μέσω της `authenticate`. Αν η ταυτοποίηση αποτύχει τότε η σύνδεση διακόπτεται αμέσως, διαφορετικά ο χρήστης συνδέεται με τον WebSocket server.

Μόλις ανοίξει μία σύνδεση με τον WebSocket server τότε ο server εκτελεί μερικές ακόμα προκαταρκτικές ενέργειες πριν αρχίσει να ανταποκρίνεται στα μηνύματα του χρήστη:

```

api.ts
1  wss.on(
2    "connection",
3    async (ws, req, credentials) => {
4      if (socketArray[credentials.id]) {
5        loggerMain.warn(
6          `User \${credentials.id} already connected`
7        );
8        ws.close(4001, "user already connected");
9        return;
10     }
11     let user = await getUser(credentials.id);
12     if (!user) {
13       loggerMain.info(
14         `User \${credentials.id} not found. Creating new user...`
15       );
16       user = await createUser(credentials.id);
17       if (!user) {
18         loggerMain.warn(`Cannot create new user
19           ↳ \${credentials.id}`);
20         ws.close(4002, "cannot create user");
21         return;
22       }
23       user.full_name = credentials.full_name;
24       socketArray[user.id] = ws;
25       ws.send(msgToJSON(typeOfMessage.login, user));
26       loggerMain.info(`User logged in: \${user}`);
27
28       ws.on("message", (message) => handleMessage(message, ws));
29     }
30   });

```

Αφού ελέγξουμε ότι ο χρήστης δεν είναι ήδη συνδεδεμένος, προσπαθούμε

να ανακτήσουμε τα στοιχεία του από τη βάση δεδομένων. Αν ο χρήστης δεν υπάρχει στη βάση (δηλαδή συνδέεται για πρώτη φορά) τότε δημιουργούμε νέο προφίλ. Αφού προσθέσουμε τον χρήστη στη λίστα των συνδέσεων για εύκολη πρόσβαση, του στέλνουμε τα στοιχεία του που ήταν αποθηκευμένα στη βάση, και θέτουμε τον message handler ώστε να ανταποκρίνεται πλέον στα μηνύματά του.

Μόλις λάβουμε κάποιο μήνυμα αρχικά πραγματοποιούμε validation για να εξασφαλίσουμε ότι είναι well-formed, και έπειτα εξετάζουμε τον τύπο του προκειμένου να αποφανθούμε για το πως θα το επεξεργαστούμε. Οι τύποι των μηνυμάτων είναι καθορισμένοι, και η επιλογή της ενέργειας γίνεται μέσω ενός switch statement:

handleMessage

```
1  async function handleMessage(message: string, ws: WebSocket) {
2    if(!validateMessage(message)) return;
3    const { type, data } = JSON.parse(message);
4    switch (type) {
5      case typeOfMessage.newDriver: {
6        driverArray[user.id] = {
7          ...user,
8          coordinates: data.coordinates,
9          car: data.car,
10         candidates: [],
11         passengers: [],
12       };
13       loggerMain.info(`New driver: ${driverArray[user.id]}`);
14       break;
15     }
16     case typeOfMessage.newPassenger: {
17       passengerArray[user.id] = {
18         ...user,
19         coordinates: data.coordinates,
20       };
21       loggerMain.info(`New passenger: ${passengerArray[user.id]}`);
22       break;
23     }
24   }
25   case typeOfMessage.updateDriver: {
26     driverArray[user.id].coordinates = data.coordinates;
27     for (const passenger of driverArray[user.id].passengers) {
28       socketArray[passenger].send(
29         msgToJSON(typeOfMessage.updateDriver, driverArray[user.id])
30       );
31     }
32     loggerMain.info(`Updated driver: ${driverArray[user.id]}`);
```

```

33     break;
34 }
35 case typeOfMessage.updatePassenger: {
36     passengerArray[user.id].coordinates = data.coordinates;
37     const driver = passengerArray[user.id].driver_id;
38     if (driver) {
39         socketArray[driver].send(
40             msgToJSON(
41                 typeOfMessage.updatePassenger,
42                 passengerArray[user.id],
43             ));
44     }
45     loggerMain.info(`Updated passenger:
46     ↪ \${passengerArray[user.id]}`);
47     break;
48 }

```

Στην περίπτωση που το μήνυμα αφορά την άφιξη ενός νέου οδηγού ή πεζού, τότε ο χρήστης απλά προστίθεται στην αντίστοιχη λίστα μαζί με τις συντεταγμένες του, όπως φαίνεται παραπάνω.

Όταν ο χρήστης στέλνει μήνυμα ανανέωσης της τοποθεσίας του, τότε ο server ενημερώνει τις συντεταγμένες του. Επιπλέον, αν πρόκειται για οδηγό, πληροφορούνται όλοι οι επιβάτες στο όχημα του χρήστη για τη νέα θέση του, ενώ αν πρόκειται για επιβάτη ενημερώνεται μόνο ο οδηγός του.

Ακολουθούν τα μηνύματα που αφορούν το ταίριασμα οδηγών με τους επιβάτες τους.

Το μήνυμα τύπου `pingPassengers` στέλνεται από τον οδηγό που επιθυμεί να παραλάβει πεζούς, ενώ το `pingDriver` στέλνεται από τους επιλεγμένους πεζούς ως απάντηση στο αίτημα του οδηγού.

handleMessage

```

1  case typeOfMessage.pingPassengers: {
2      driverArray[user.id].candidates = sampleSize(passengerArray, 5);
3      for (const id of driverArray[user.id].candidates) {
4          passengerArray[id].driver_id = user.id;
5          socketArray[id].send(
6              msgToJSON(typeOfMessage.pingPassengers, user.id)
7          );
8      }
9      loggerMain.info(
10         `Driver \${user.id} pinged passengers \${
11             driverArray[user.id].candidates

```

```

12     }`
13   );
14   break;
15 }
16 case typeOfMessage.pingDriver: {
17   const driver_id = passengerArray[user.id].driver_id;
18   remove(
19     driverArray[driver_id].candidates,
20     (passenger) => passenger == user.id
21   );
22   if (data.refused) {
23     delete passengerArray[user.id].driver_id;
24     loggerMain.info(`Passenger ${user.id} refused driver
25       ↪ ${driver_id}`);
26     break;
27   }
28   if (
29     driverArray[driver_id].passengers.length >=
30     driverArray[driver_id].car.seats
31   ) {
32     ws.send(msgToJSON(typeOfMessage.pingDriver, null));
33     delete passengerArray[user.id].driver_id;
34     break;
35   }
36   driverArray[driver_id].passengers.push(user.id);
37   ws.send(msgToJSON(typeOfMessage.pingDriver,
38     ↪ driverArray[driver_id]));
39   socketArray[driver_id].send(
40     msgToJSON(typeOfMessage.updatePassenger,
41       ↪ passengerArray[user.id])
42   );
43   loggerMain.info(`Passenger ${user.id} paired with driver
44     ↪ ${driver_id}`);
45   break;
46 }

```

Όταν δεχόμαστε μήνυμα `pingPassengers` από έναν οδηγό, επιλέγουμε τυχαία 5 χρήστες από τη λίστα των πεζών και τους αναθέτουμε προσωρινά ως υποψήφιους επιβάτες. Αφού ενημερώσουμε τους επιλεγμένους πεζούς για τον υποψήφιο οδηγό τους, αναμένουμε την απάντησή τους μέσα από μήνυμα τύπου `pingDriver`. Μόλις δεχτούμε την απάντηση ενός δεδομένου πεζού, τον αφαιρούμε από τη λίστα των υποψήφιων επιβατών και ελέγχουμε αν το αίτημα έγινε αποδεκτό ή απορρίφθηκε.

Αν ο πεζός αποδέχτηκε το αίτημα του οδηγού τότε ελέγχουμε αν υπάρχουν διαθέσιμες θέσεις στο όχημα, και αν όχι, ενημερώνουμε τον πεζό για την έλλειψη θέσεων. Διαφορετικά, ο πεζός προστίθεται στη λίστα των επιβατών του οδηγού, και πλέον επιβάτης και οδηγός θα ενημερώνονται ο ένας για την

τοποθεσία του άλλου.

Αν για οποιονδήποτε λόγο ο χρήστης αποφασίσει να διακόψει τη διαδρομή του, τότε στέλνεται το αντίστοιχο μήνυμα παύσης. Εκτός από την αφαίρεση του χρήστη από την αντίστοιχη λίστα `driverArray/passengerArray`, στην περίπτωση εξόδου του οδηγού ενημερώνουμε όλους τους επιβάτες του, ενώ στην περίπτωση εξόδου ενός επιβάτη ενημερώνουμε τον οδηγό του.

handleMessage

```
1  case typeOfMessage.stopDriver: {
2    for (const passenger of driverArray[id].passengers){
3      delete passengerArray[passenger].driver_id;
4      socketArray[passenger].send(
5        msgToJSON(typeOfMessage.updateDriver, null)
6      );
7    }
8    delete driverArray[id];
9    loggerMain.info(`Stopped driver ${user.id}`);
10   break;
11 }
12 case typeOfMessage.stopPassenger: {
13   if (passengerArray[id].driver_id) {
14     socketArray[passengerArray[id].driver_id].send(
15       msgToJSON(typeOfMessage.updatePassenger, { cancelled: id })
16     );
17     remove(
18       driverArray[passengerArray[id].driver_id].passengers,
19       (passenger) => passenger == id
20     );
21   }
22   delete passengerArray[id];
23   loggerMain.info(`Stopped passenger ${user.id}`);
24   break;
25 }
```

Το τελευταίο στάδιο της διαδρομής είναι η άφιξη στον προορισμό και η αποστολή των αξιολογήσεων:

handleMessage

```
1  case typeOfMessage.arrivedDestination: {
2    for (const passenger of driverArray[user.id].passengers) {
3      socketArray[passenger].send(
4        msgToJSON(typeOfMessage.arrivedDestination, null)
5      );
6      pendingRatings[passenger] = [user.id];
7    }
8  }
```

```

8   pendingRatings[user.id] = driverArray[user.id].passengers;
9   loggerMain.info(
10    `Driver \${user.id} arrived at destination with passengers \${
11      driverArray[user.id].passengers
12    }`
13  );
14  break;
15 }
16 case typeOfMessage.sendRatings: {
17   for (const {id, rating} of data) {
18     if(remove(pendingRatings[user.id], (e) => e == id).length == 0)
19       ↪ continue;
20     if (rating == 0) continue;
21     addUserRating(id, rating);
22     loggerMain.info(
23       `User \${user.id} rated user \${id} with \${rating} stars`
24     );
25   }
26   break;
27 }

```

Μόλις ο οδηγός φτάσει στον προορισμό τότε στέλνει το μήνυμα `arrived-Destination` στον server, ο οποίος με τη σειρά του ενημερώνει όλους τους επιβάτες για την άφιξη στη σχολή.

Ταυτόχρονα ορίζουμε μέσω της λίστας `pendingRatings` το ποιοι χρήστες μπορούν να αξιολογήσουν ποιους.⁴ Μόλις ληφθεί ένα μήνυμα που περιέχει τις αξιολογήσεις χρηστών, ελέγχεται πρώτα η εγκυρότητά του, και έπειτα ενημερώνονται οι αντίστοιχες βαθμολογίες στην βάση δεδομένων.

Τέλος, έχουμε δύο τύπους μηνυμάτων που αφορούν την δημιουργία οχήματος και την μεταφόρτωση εικόνας προφίλ:

handleMessage

```

1  case typeOfMessage.updateUserPicture: {
2    let newPicture = await updateUserPicture(user.id, data);
3    loggerMain.info(`Updated profile picture of \${user.id} to
4      ↪ \${newPicture}`);
5    if (user.picture) deletePicture(user.picture);
6    user.picture = newPicture;
7    ws.send(msgToJSON(typeOfMessage.updateUserPicture, newPicture));
8  }

```

⁴ Αυτό χρειάζεται για να εξασφαλίσουμε ότι οι αξιολογήσεις είναι γνήσιες και γίνονται μετά από την περάτωση μίας διαδρομής. Διαφορετικά ο κάθε χρήστης θα μπορούσε να στείλει αυθαίρετα αξιολογήσεις σε οποιονδήποτε άλλο χρήστη, χωρίς να έχει συμμετάσχει σε κάποια διαδρομή

```

8   const resultNSFW = await isNSFW(newPicture);
9   if (!resultNSFW) return;
10  deleteUserPicture(newPicture);
11  updateUserPicture(user.id, null);
12  socketArray[user.id].send(
13    msgToJSON(typeOfMessage.deleteUserPicture, picture)
14  );
15  break;
16 }
17 case typeOfMessage.addCar: {
18   const car = await createCar(user.id, data);
19   loggerMain.info(`Added car to \${user.id}: \${car}`);
20   user.cars[car.id] = car;
21   ws.send(msgToJSON(typeOfMessage.addCar, car));
22
23   if (!car.picture) break;
24   const resultNSFW = await isNSFW(car.picture);
25   if (!resultNSFW) return;
26   deleteCarPicture(car.picture);
27   updateCarPicture(user.id, null);
28   socketArray[user.id].send(
29     msgToJSON(
30       typeOfMessage.deleteCarPicture,
31       {id: car.id, picture: picture}
32     ));
33   break;
34 }

```

Για την αλλαγή εικόνας προφίλ, έχει ήδη προηγηθεί η μεταφόρτωση της εικόνας από τον χρήστη στον ειδικό server πολυμέσων. Μέσω του μήνυματος `updateUserPicture` ο χρήστης στέλνει μόνο το όνομα του νέου αρχείου, ώστε ο server να ανανεώσει την φωτογραφία στην βάση δεδομένων. Η νέα φωτογραφία ελέγχεται από το φίλτρο καταλληλότητας, και αν διαπιστωθεί ότι περιέχει ακατάλληλο περιεχόμενο τότε διαγράφεται αμέσως από τον server και ο χρήστης ενημερώνεται για τη διαγραφή. Το φίλτρο καταλληλότητας περιγράφεται περισσότερο στο Παράρτημα Α'.

Κεφάλαιο 7

Συμπεράσματα

Σε αυτή τη διπλωματική εργασία παρουσιάσαμε μία ολοκληρωμένη εφαρμογή συνεπιβατισμού η οποία στοχεύει να λύσει το πρόβλημα της συμφόρησης που αναπτύσσεται κοντά στο Πολυτεχνείο. Η εκπόνηση του παρόντος έργου περιέλαβε την έρευνα και βαθύτερη εξοικείωσή μας όχι μόνο με την ανάπτυξη κινητών εφαρμογών, αλλά και με θέματα που αφορούν την λειτουργία των server και των βάσεων δεδομένων, τη διαδικτυακή επικοινωνία, και την ταυτοποίηση. Παράλληλα εξοικειωθήκαμε και με συγκεκριμένες τεχνολογίες οι οποίες είτε είναι εδραιωμένες στον τομέα των διαδικτυακών εφαρμογών όπως το Node.js/Express.js και η MySQL, είτε είναι ανερχόμενες όπως το Flutter.

Η εφαρμογή εμφανίζει αρκετά σύνθετη λειτουργία, αφού είναι υπεύθυνη για τον ταυτόχρονο συντονισμό πολλαπλών διαφορετικών διαδρομών στις οποίες εμπλέκονται πολλαπλοί χρήστες, όλα σε πραγματικό χρόνο. Ωστόσο ευελπιστούμε ότι η εφαρμογή είναι σε θέση να χειριστεί αποτελεσματικά την πολυπλοκότητα αυτή, και να δώσει λύση στο προκείμενο πρόβλημα. Έπειτα από προκαταρκτικές δοκιμές για να εξασφαλίσουμε ότι η εφαρμογή λειτουργεί όσο το δυνατόν καλύτερα, το επόμενο στάδιο είναι η διετέλεση περισσότερων δοκιμών σε πραγματικές πλέον συνθήκες μέσω ενός beta test, για τον εντοπισμό σφαλμάτων και την περαιτέρω βελτίωση της εφαρμογής μέσα από το feedback των φοιτητών. Μετά το πέρας του δοκιμαστικού σταδίου, η εφαρμογή μπορεί να γίνει διαθέσιμη σε όλους πλέον τους φοιτητές της σχολής και να αξιολογηθεί η αποτελεσματικότητά της για τη μείωση του συνωστισμού. Σε περίπτωση που η εφαρμογή είναι επιτυχής, στο μέλλον θα μπορούσε ακόμα να επεκταθεί και για χρήση σε άλλα πανεπιστήμια τα οποία αντιμετωπίζουν παρόμοιο πρόβλημα.

Παράρτημα Α΄

Εντοπισμός ακατάλληλου περιεχομένου μέσω μηχανικής όρασης

Όπως έχουμε αναφέρει, ένα μέρος της εφαρμογής αφορά το ανέβασμα εικόνων από τους χρήστες για εμφάνιση στο δημόσιο προφίλ τους. Ήταν σημαντικό να προσφέρουμε τη δυνατότητα αυτή στους χρήστες, καθώς θεωρήσαμε ότι οι φωτογραφίες καθιστούν την εφαρμογή πιο εύχρηστη, επιτρέποντας στους χρήστες να αναγνωρίσουν εύκολα ο ένας τον άλλον, ενώ ενισχύουν ταυτόχρονα την κοινωνική διάσταση της εφαρμογής. Ωστόσο, αυτή η επιλογή που δίνεται στους χρήστες ενδέχεται να παρουσιάσει κινδύνους. Ο κάθε χρήστης μπορεί να ανεβάσει φωτογραφίες ακατάλληλου περιεχομένου, και οι φωτογραφίες αυτές μπορεί να εμφανιστούν στις συσκευές άλλων χρηστών, γεγονός που πρέπει σε κάθε περίπτωση να αποφευχθεί. Η επώνυμη φύση της εφαρμογής που επιβάλλεται από την απαίτηση ταυτοποίησης μειώνει σημαντικά την πιθανότητα τέτοιων περιστατικών, όμως πρέπει να είμαστε σε θέση να εντοπίσουμε άμεσα το υλικό αυτό αν τυχόν εμφανιστεί.

Για τον σκοπό αυτό χρησιμοποιήσαμε ένα μοντέλο μηχανικής μάθησης. Αφού δοκιμάσαμε αρκετά μοντέλα, αποφασίσαμε να επιλέξουμε για την περίπτωση μας το OpenNSFW2[27], το οποίο αποτελεί σύγχρονη υλοποίηση του δημοφιλούς μοντέλου OpenNSFW[28] της Yahoo. Το μοντέλο είναι τύπου Residual Network, και βασίζεται στο ResNet50 της Microsoft, με υλοποίηση στη βιβλιοθήκη Tensorflow. Επιλέξαμε το συγκεκριμένο μοντέλο καθώς έπει-

τα από δοκιμές συμπεράναμε ότι προσφέρει τη μεγαλύτερη ακρίβεια με μικρό κόστος τόσο όσον αφορά το μέγεθος όσο και τον υπολογιστικό φόρτο.

Για τη λειτουργία του μοντέλου δημιουργήσαμε μέσω του framework Flask έναν απλό τοπικό server Python¹ ο οποίος λειτουργεί παράλληλα με τον κύριο server. Ο server αυτός φορτώνει το μοντέλο μηχανικής όρασης και αναμένει αιτήματα HTTP για την αξιολόγηση του περιεχομένου ενός δεδομένου αρχείου. Μόλις λάβει ένα αίτημα, ο server δίνει το αρχείο εικόνας ως είσοδο στο μοντέλο και λαμβάνει ως έξοδο την πιθανότητα απεικόνισης ακατάλληλου περιεχομένου, την οποία στέλνει ως απάντηση πίσω στον κύριο server.

Είναι σημαντικό να σημειωθεί ότι εφόσον οι δύο servers ανήκουν σε διαφορετικές διεργασίες η λειτουργία τους είναι παράλληλη, και καθώς η επικοινωνία επιτυγχάνεται μέσω ασύγχρονων non-blocking HTTP calls ο κύριος server σε καμία περίπτωση δεν παύει να επεξεργάζεται αιτήματα περιμένοντας την απόκριση του server μηχανικής όρασης. Η υπολογιστικά ακριβή εργασία του classification γίνεται σε ξεχωριστό thread, και ακόμα και σε διαφορετικό πυρήνα CPU. Έτσι μπορούμε να έχουμε γρήγορη αναγνώριση εικόνων και επομένως άμεση εντόπιση πιθανών παραβάσεων, χωρίς όμως αυτό να είναι εις βάρος της ταχύτητας του server και επομένως της εμπειρίας του χρήστη.

¹Αρχικά επιχειρήσαμε να χρησιμοποιήσουμε το Node.js και για αυτόν τον σκοπό ώστε να έχουμε ομοιομορφία όσον αφορά τη γλώσσα, όμως κάποιες ασυνέπειες στην υλοποίηση των διεπαφών του Tensorflow έκαναν προβληματική τη φόρτωση του μοντέλου στη JavaScript

Παράρτημα Β΄

Authentication

Για την ταυτοποίηση των χρηστών είναι απαραίτητη η επικοινωνία της εφαρμογής με το σύστημα ταυτοποίησης του Πολυτεχνείου, το οποίο βασίζεται στο πρωτόκολλο SAML¹. Αντί να επικοινωνούμε απευθείας με τον Identity Provider της σχολής, εκτελούμε την ταυτοποίηση μέσω ενός server Keycloak² ο οποίος είναι ρυθμισμένος έτσι ώστε να λειτουργεί ως Identity Broker εκ μέρους του. Χρησιμοποιώντας το Keycloak η ταυτοποίηση μπορεί να γίνει μέσω του πρωτοκόλλου OpenID Connect το οποίο είναι πιο εύκολα υλοποιήσιμο από το SAML σε μοντέρνες εφαρμογές.

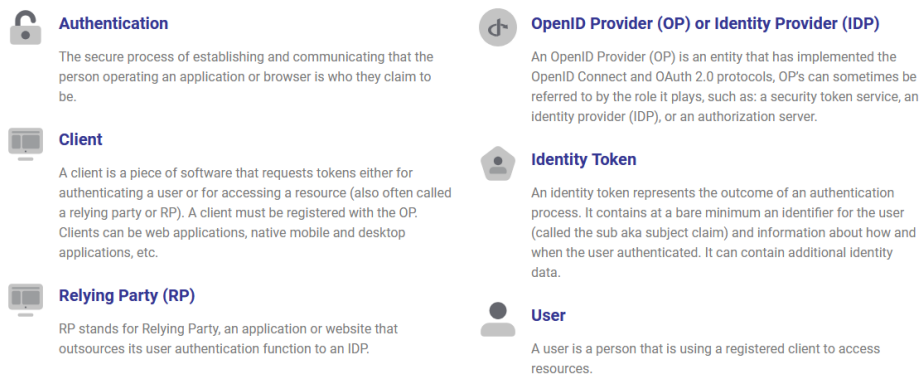
B΄.1 OAuth και OpenID Connect

Ο όρος authentication αφορά την ταυτοποίηση ενός χρήστη, ενώ το authorization αφορά μόνο την παροχή άδειας σε μία υπηρεσία για την πρόσβαση σε προστατευμένους πόρους εκ μέρους του χρήστη. Οι δύο έννοιες είναι στενά συνδεδεμένες, ωστόσο είναι σημαντικό να μη συγχέονται.

Οι προδιαγραφές του OAuth 2.0 δημοσιεύθηκαν τον Οκτώβριο του 2012 από τον οργανισμό IETF (Internet Engineering Task Force) ως ενημέρωση των προδιαγραφών του πρωτοκόλλου OAuth 1.0 που είχε δημοσιευθεί το 2007, και γρήγορα υιοθετήθηκαν από πολλές υπηρεσίες του διαδικτύου. Παρά το γεγονός ότι το OAuth αποτελεί αυστηρά πρωτόκολλο authorization και δεν είναι

¹Το SAML (Security Assertion Markup Language) είναι ένα σύστημα σχεδιασμένο για την ασφαλή ανταλλαγή ισχυρισμών (assertions) που αφορούν την ταυτότητα και τις εξουσιοδοτήσεις ενός χρήστη

²Ο server Keycloak δημιουργήθηκε από τους φοιτητές της σχολής Άγγελο Κολαίτη και Σταμάτη Κατσιαούνη-Μολυβά



Σχήμα Β'.1: Ορισμοί των βασικών εννοιών που χρησιμοποιούνται στο πρωτόκολλο OpenID Connect. Πηγή: [30].

σχεδιασμένο για authentication, από την απαρχή του το OAuth χρησιμοποιήθηκε καταχρηστικά για το authentication χρηστών από μεγάλο μέρος του διαδικτύου. Αυτή η λανθασμένη χρήση του OAuth για authentication παρουσιάζει κινδύνους για την ασφάλεια των δεδομένων του χρήστη[29].

Το OpenID Connect (OIDC) από την άλλη είναι ένα πρωτόκολλο σχεδιασμένο ειδικά για το ασφαλή authentication χρηστών. Δημοσιεύτηκε το 2014 ως βελτίωση του παλαιότερου πρωτοκόλλου OpenID 1.0/2.0, και βασίζεται πάνω στο authorization framework του OAuth 2.0. Το OIDC παρουσιάζει μία μέθοδο ταυτοποίησης η οποία είναι συμβατή με το ήδη εδραιωμένο πρωτόκολλο OAuth 2.0, επιτρέποντας την εύκολη υλοποίηση του authentication τόσο από πλευράς του Identity Provider όσο και από πλευράς του Client[30].

B'.2 Browser authentication

Τώρα θα περιγράψουμε συνοπτικά την συμβατική διαδικασία ταυτοποίησης που εκτελείται συνήθως στο browser χρησιμοποιώντας το πρωτόκολλο OIDC. Αυτή η μορφή ταυτοποίησης χρησιμοποιείται και από εμάς για την ειδική ιστοσελίδα της εφαρμογής η οποία περιγράφεται στο Παράρτημα Δ', μαζί με τις λεπτομέρειες υλοποίησης.

Η ροή ταυτοποίησης ξεκινάει όταν ο χρήστης ζητάει πρόσβαση στον πόρο που απαιτεί δικαιώματα πρόσβασης. Ο web server δέχεται το αίτημα και ελέγχει αν ο συγκεκριμένος χρήστης είναι ήδη συνδεδεμένος. Αν όχι, τότε ξεκινάει το

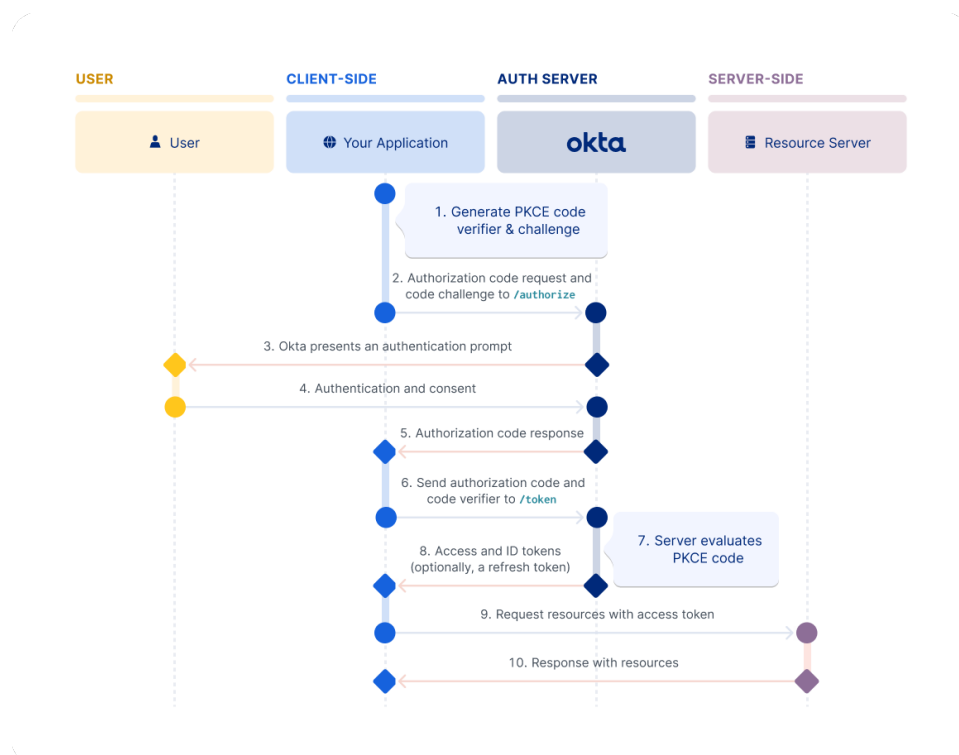
λεγόμενο authorization code flow που ορίζεται από το πρωτόκολλο OpenID Connect για την ταυτοποίηση του χρήστη. Ο server (Relying Party) ανακατευθύνει τον browser του χρήστη στην ιστοσελίδα του Identity Provider με τις κατάλληλες παραμέτρους που προσδιορίζουν το αίτημα (client ID, code challenge κ.ά.). Αφού ο χρήστης εισάγει τα στοιχεία του, το Keycloak ανακατευθύνει με τη σειρά του τον browser πίσω στην ιστοσελίδα του server, περιλαμβάνοντας στις παραμέτρους του συνδέσμου έναν κώδικα που δίνει πρόσβαση στα στοιχεία του χρήστη. Μόλις ο server λάβει την αίτηση αυτή από τον χρήστη, παρουσιάζει τον κώδικα στο Keycloak, το οποίο τελικά του προσφέρει τα στοιχεία του χρήστη με τη μορφή JWT. Σε αυτό το σημείο ο server επαληθεύει την εγκυρότητα του token, και αφού εξάγει τις πληροφορίες που χρειάζεται τις αποθηκεύει στο session του χρήστη. Από το σημείο αυτό και πέρα, κάθε αίτημα του χρήστη προς τον server συνοδεύεται από ένα session cookie το οποίο ο server μπορεί να χρησιμοποιήσει για να αντιστοιχίσει το αίτημα του χρήστη με το κατάλληλο session, και επομένως με την ταυτότητα του.

B'.3 Native app authentication

Το authentication του χρήστη στην εφαρμογή κινητού λειτουργεί με παρόμοιο τρόπο, με τη διαφορά ότι ο *client* είναι πια μία εφαρμογή κινητού και όχι browser, γεγονός που σημαίνει ότι απαιτείται ειδική μέριμνα για την προστασία των προσωπικών στοιχείων του χρήστη. Για τον σκοπό αυτό συστήνεται η χρήση του AppAuth, μίας βιβλιοθήκης που δημιουργήθηκε από την OpenID Foundation για authentication εντός κινητών εφαρμογών.

Το βασικό σημείο που χρήζει προσοχής είναι η επικινδυνότητα της χρήσης *embedded user-agent* και συγκεκριμένα των in-app webviews στη διαδικασία ταυτοποίησης. Τα webviews χρησιμοποιούνται εκτεταμένα σε κινητές εφαρμογές καθώς επιτρέπουν τη φόρτωση ιστοσελίδων εντός της εφαρμογής, παρέχοντας αυξημένο έλεγχο στην εμφάνιση και λειτουργία του “browser”. Αυτές όμως οι δυνατότητες που προσφέρουν τα webviews τα καθιστούν πλέον ακατάλληλα για authentication, καθώς η εφαρμογή έχει πλήρη πρόσβαση στα στοιχεία σύνδεσης που ο χρήστης εισάγει στη φόρμα ταυτοποίησης.[32, §8.12] Για αυτόν τον λόγο το AppAuth χρησιμοποιεί εξ ολοκλήρου τον αξιόπιστο default system browser του χρήστη³ για το authentication flow, επιστρέφο-

³Safari στις συσκευές iOS, και Google Chrome στις συσκευές Android



Σχήμα Β'.2: Το authorization code flow σε περισσότερη λεπτομέρεια. Στην περίπτωση μας ο *client* είναι ο web server, και ο *auth server* είναι το Keycloak. Η διαδικασία σταματάει στο βήμα 8 καθώς μας ενδιαφέρει μόνο το ID Token το οποίο ταυτοποιεί τον χρήστη, και δε χρειαζόμαστε το access token για πρόσβαση σε εξωτερικό API/*resource server*. Πηγή: [31].

ντας στο τέλος της διαδικασίας μόνο το authorization code στην εφαρμογή. Έτσι προστατευόμαστε από τους κινδύνους ασφαλείας που εγκυμονούν τα webviews, και διατηρούμε την εμπιστοσύνη του χρήστη στην ασφάλεια της εφαρμογής. Επιπλέον, ο εξωτερικός browser διατηρεί όλους τους αποθηκευμένους κωδικούς και τα sessions του χρήστη, διευκολύνοντας τη σύνδεση του στην εφαρμογή και επομένως βελτιώνοντας την εμπειρία του.

Στην εφαρμογή η κλάση Authenticator είναι υπεύθυνη για το authentication. Η κλάση χρησιμοποιεί το plugin FlutterAppAuth το οποίο υλοποιεί το πρωτόκολλο AppAuth για όλες τις πλατφόρμες. Στην κλάση περιέχεται το ID Token ως string, καθώς και οι μέθοδοι authenticate και endSession που εκτελούν τις προφανείς λειτουργίες.

authenticator.dart

```
1 class Authenticator {
2   const appAuth = FlutterAppAuth();
3   String? _idToken;
4   String? get idToken => _idToken;
5
6   Future<String?> authenticate() async {
7     final request = AuthorizationTokenRequest(
8       authClientID,
9       '\$appScheme:/auth',
10      issuer: authIssuer,
11      scopes: ['openid', 'profile', 'email'],
12      additionalParameters: {'kc_idp_hint': 'saml'},
13    );
14    final response = await
15      ↪ appAuth.authorizeAndExchangeCode(request);
16    _idToken = response?.idToken;
17    return _idToken;
18  }
19  Future<EndSessionResponse?> endSession() {
20    final request = EndSessionRequest(
21      issuer: authIssuer,
22      idTokenHint: _idToken,
23      postLogoutRedirectUrl: '\$appScheme:/',
24    );
25    final response = await appAuth.endSession(request);
26    if (response != null) _idToken = null;
27    return response;
28  }
```

Η μέθοδος authenticate στέλνει το authorization request με τις κατάλληλες παραμέτρους στον Identity Provider της σχολής, και λαμβάνει το ID

Token το οποίο τελικά θα σταλεί στον server για την έναρξη της σύνδεσης. Με παρόμοιο τρόπο η μέθοδος `endSession` στέλνει το `end session request` στον IDP και ακυρώνει το υπάρχον ID Token, λήγοντας τη σύνδεση με τον server.

Αξιοσημείωτη είναι η χρήση του `appScheme` στο πεδίο του `redirectUri`. Το `appScheme`⁴ αποτελεί το αναγνωριστικό της εφαρμογής, και η χρήση του ως `redirect URI` σημαίνει ότι μόλις ολοκληρωθεί η ταυτοποίηση στον IDP, ο χρήστης θα μεταβεί αυτόματα πίσω στην εφαρμογή όπως περιγράψαμε προηγουμένως.

Από την πλευρά του server, το JSON Web Token λαμβάνεται μέσα από τα headers του request, και αποκωδικοποιείται/επαληθεύεται. Αν το JWT είναι έγκυρο τότε επιστρέφεται ένα αντικείμενο με τα σχετικά στοιχεία του χρήστη και ο server μπορεί να προχωρήσει στη ρύθμιση της σύνδεσης.

api.ts/authenticate

```
1  async function authenticate(req: IncomingMessage) {
2    let idToken = req.headers["authentication"];
3    if (!idToken) return;
4    let decodedToken;
5    try {
6      decodedToken = (await jwtVerify(idToken, JWKS)).payload;
7    } catch (error) {
8      loggerMain.warn(error);
9      return;
10   }
11   if (!decodedToken || !decodedToken.name || !decodedToken.id) {
12     loggerMain.warn(
13       `Client tried to connect with invalid info: ` +
14       JSON.stringify(decodedToken, null, 2)
15     );
16     return;
17   }
18   return {
19     id: decodedToken.id as string,
20     full_name: decodedToken.name as string,
21   } as Credentials;
22 }
```

⁴Το app scheme της εφαρμογής μας είναι το “gr.ntua.ece.ridehailing” σύμφωνα με τη σύμβαση του reverse domain naming

Παράρτημα Γ'

Βάση Δεδομένων

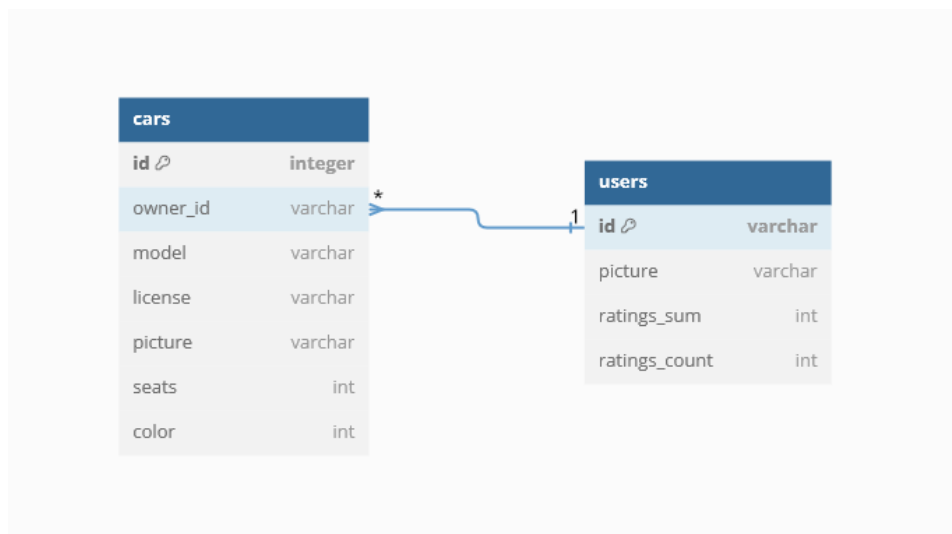
Μέσα στις απαιτήσεις της εφαρμογής μας συμπεριλαμβάνεται και η αποθήκευση των δεδομένων των χρηστών online. Για την αποθήκευση των δεδομένων αυτών στον server η ιδανική λύση είναι η χρήση μίας σχεσιακής βάσης δεδομένων τύπου SQL (στην περίπτωση μας της MySQL).

Όπως έχουμε αναφέρει, τα δεδομένα που επιθυμούμε να αποθηκεύσουμε στη βάση αφορούν τους χρήστες και τα οχήματά τους. Ο κάθε χρήστης έχει έναν αριθμό μητρώου, τα οχήματά του, μία βαθμολογία, και μία εικόνα προφίλ. Οι πληροφορίες του κάθε οχήματος ενός χρήστη περιλαμβάνουν το μοντέλο, τον αριθμό πινακίδας, τον αριθμό θέσεων, το χρώμα, και μία φωτογραφία.

Είναι σαφές ότι η βάση μας θα αποτελείται από δύο πίνακες/tables, ένα για τους χρήστες, και ένα για τα οχήματα. Τα απαραίτητα columns που θα διαθέτει ο κάθε πίνακας είναι προφανή. Εφόσον το κάθε όχημα ανήκει σε έναν ακριβώς χρήστη, και επειδή ο κάθε χρήστης επιτρέπεται να έχει πολλαπλά οχήματα, έχουμε μία σχέση one-to-many μεταξύ χρηστών και οχημάτων. Για να υλοποιήσουμε τη σχέση αυτή στη βάση, θα προσθέσουμε στις στήλες του πίνακα οχημάτων τον AM του αντίστοιχου χρήστη-ιδιοκτήτη ως foreign key.

Το primary key του πίνακα χρηστών είναι το AM τους, ενώ το primary key των οχημάτων είναι ένας απλός auto-incrementing ακέραιος. Εφόσον όλες οι εικόνες είναι αποθηκευμένες ως αρχεία στο δίσκο, η πληροφορία που περιέχεται στη βάση είναι το όνομα/UUID του αρχείου που αντιστοιχεί στην κάθε εικόνα. Το χρώμα του κάθε οχήματος αναπαριστάται ως ένας 32-bit unsigned integer που αντιστοιχεί στα τέσσερα 8-bit κανάλια ARGB¹. Η βαθμολογία του χρήστη

¹Ο λόγος που χρησιμοποιούμε ακέραιο τύπο για το χρώμα αντί για ένα string που αναπαριστά το hex color code είναι το γεγονός ότι το Flutter ορίζει χρώματα μόνο βάσει της ακέρειας 32-bit αναπαράστασής τους, επομένως έτσι αποφεύγουμε τις περιττές μετατροπές τιμών. Εξ άλλου, η



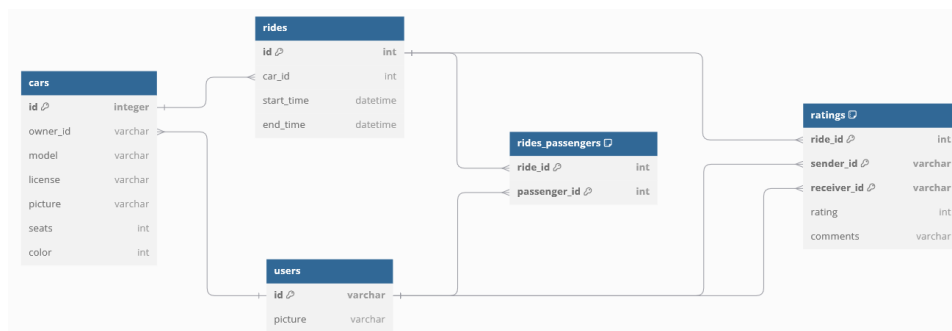
Σχήμα Γ'.1: Σχηματικό διάγραμμα που περιγράφει τη δομή της βάσης δεδομένων που χρησιμοποιούμε. (Δημιουργήθηκε μέσω της ιστοσελίδας dbdiagram.io)

αποθηκεύεται σε δύο μερή, το πλήθος και το άθροισμα των αξιολογήσεων, ώστε να μπορούμε να εξάγουμε και να ενημερώνουμε κατάλληλα τον μέσο όρο.

Η μορφή της βάσης ως έχει είναι αρκετά απλή καθώς δεν υπάρχει η ανάγκη διαχείρισης πιο πολύπλοκων δεδομένων. Ωστόσο εύκολα μπορεί να επεκταθεί αν οι απαιτήσεις της εφαρμογής γίνουν πιο σύνθετες. Για παράδειγμα, αν επιθυμούσαμε να καταγράφουμε κάθε διαδρομή στη βάση, θα προσθέταμε έναν επιπλέον πίνακα διαδρομών στον οποίο θα βρίσκονταν όλα τα σχετικά δεδομένα της διαδρομής που μας ενδιαφέρουν. Εκτός από το ID, για την κάθε διαδρομή θα ορίζαμε τον οδηγό και τους επιβάτες οι οποίοι συμμετείχαν σε αυτή, καθώς και το όχημα του οδηγού. Το ID του οδηγού και του οχήματος εύκολα προστίθενται ως στήλες του πίνακα διαδρομών καθώς και στις δύο περιπτώσεις έχουμε μία σχέση one-to-many.

Οι επιβάτες μίας διαδρομής ωστόσο ορίζουν μία σχέση many-to-many, καθώς πολλοί επιβάτες μπορούν να συμμετέχουν σε μία διαδρομή, και κάθε επιβάτης μπορεί να συμμετάσχει σε πολλές διαδρομές σε διαφορετικές χρονικές στιγμές. Για αυτό το λόγο απαιτείται η χρήση ενός επιπλέον συνδετικού πίνακα (associative table) ο οποίος θα αναπαριστά τη many-to-many relationship, με

αναπαράσταση με ακέραιο στη βάση είναι ελαφρώς πιο αποδοτική από την χρήση string, ωστόσο η διαφορά αυτή δεν είναι τόσο σημαντική.



Σχήμα Γ'.2: Σχηματικό διάγραμμα που περιγράφει μία εναλλακτική, πιο σύνθετη δομή για τη βάση δεδομένων που μπορούμε να χρησιμοποιήσουμε αν η εφαρμογή επεκταθεί στο μέλλον

ένα foreign key για το ID της διαδρομής και ένα foreign key για τον επιβάτη.

Τέλος, θα μπορούσαμε να εξάγουμε τις αξιολογήσεις των χρηστών σε έναν ξεχωριστό πίνακα, προκειμένου να ενσωματώσουμε όλες τις πληροφορίες της κάθε αξιολόγησης. Στον πίνακα αυτόν κάθε αξιολόγηση θα ορίζεται από τον συνδυασμό του ID της διαδρομής στην οποία συνέβει, και τα ID του αξιολογητή και αξιολογούμενου. Εκτός από τον αριθμό αστεριών, μπορούμε επίσης να προσθέσουμε μία στήλη για επιπλέον σχόλια του αξιολογητή.

Παράρτημα Δ΄

Ιστοσελίδα

Η εφαρμογή μας συνοδεύεται από μία ιστοσελίδα που περιέχει συμπληρωματικές πληροφορίες όπως τους όρους χρήσης και την πολιτική απορρήτου, και η οποία επιτρέπει στους χρήστες να δουν τις λεπτομέρειες του λογαριασμού τους αλλά και να διαγράψουν τα αποθηκευμένα στοιχεία τους. Επομένως, εκτός από τους servers που αναλύσαμε στο κύριο μέρος οι οποίοι είναι απαραίτητοι για τη λειτουργία της εφαρμογής, διαθέτουμε και έναν ακόμα ανεξάρτητο web server που είναι υπεύθυνος για την ιστοσελίδα αυτή.

Ο server χρησιμοποιεί το framework Express.js, το πιο ευρέως διαδεδομένο framework για τη δημιουργία web εφαρμογών μέσω του Node.js.

Οι στατικές σελίδες παρέχονται απευθείας από το Nginx και δεν παρουσιάζουν ιδιαίτερο ενδιαφέρον. Επομένως, θα αναλύσουμε το μέρος της ιστοσελίδας που είναι δυναμικό, δηλαδή τη σελίδα που εμφανίζει το προφίλ του χρήστη. Η χρήση του Express.js βασίζεται στο αντικείμενο που κατά σύμβαση ονομάζουμε `app` το οποίο περιέχει όλο το API του framework, και μέσω του οποίου ορίζουμε τη συμπεριφορά της εφαρμογής. Το παρακάτω μέρος του κώδικα ορίζει τη συμπεριφορά της εφαρμογής όταν ο χρήστης μεταβεί στη σελίδα του προφίλ του (GET request στο relative URL `/profile`). Εφόσον η σελίδα αυτή δεν είναι ίδια για τον κάθε χρήστη, ο server χρειάζεται να ξέρει την ταυτότητα του ατόμου προκειμένου να στείλει τη σωστή σελίδα με τα στοιχεία του. Έτσι ο server πραγματοποιεί πρώτα το authentication του χρήστη, και αφού διαπιστώσει την ταυτότητά του προσπελάζει τη βάση δεδομένων για να συμπληρώσει τη σελίδα με τα κατάλληλα στοιχεία¹. Όταν ο server ταυτοποι-

¹Η συμπλήρωση του HTML αρχείου με τα κατάλληλα στοιχεία γίνεται μέσω της EJS, μίας απλής γλώσσας templating για HTML

ήσει έναν χρήστη αποθηκεύει τον AM του στο αντίστοιχο session έτσι ώστε μελλοντικά αιτήματα στη σελίδα προφίλ να μην απαιτούν ξανά την ταυτοποίηση του χρήστη, τουλάχιστον μέχρι ο χρήστης να τερματίσει το session κλείνοντας τον browser του.

```
0
1  nst app = express();
2
3  app.get("/profile", async (req, res) => {
4    if (!req.session.userId) {
5      res.redirect("/auth");
6      return;
7    }
8    let user = await getUser(req.session.userId);
9    if (!user) {
10     res.status(500).redirect("/");
11     return;
12   }
13
14   res.render("profile", {
15     id: user.id,
16     name: user.full_name,
17     cars: user.cars,
18     ratings_count: user.ratings_count,
19     ratings_sum: user.ratings_sum,
20     picture: user.picture,
21   });
22 });
```

Έτσι λοιπόν βλέπουμε ότι όταν ο χρήστης μεταβεί στη σελίδα profile ελέγχουμε αρχικά αν υπάρχει ήδη session, και αν όχι τότε οδηγούμε τον χρήστη στη σελίδα ταυτοποίησης auth. Αν ο χρήστης είναι ταυτοποιημένος τότε μέσω της `getUser` διαβάζουμε όλα τα στοιχεία του από τη βάση, και έπειτα στέλνουμε τη συμπληρωμένη σελίδα προφίλ μέσω της συνάρτησης `render`.

Ακολουθεί ο χειρισμός του authentication:

```
website.ts
1  const myIssuer = await Issuer.discover(env.ISSUER);
2
3  const oidcClient = new myIssuer.Client({
4    client_id: env.WEB_CLIENT_ID,
5    client_secret: env.WEB_CLIENT_SECRET,
6    redirect_uris: [env.WEB_REDIRECT_URI],
7    response_types: ["code"],
8  });
```

```

9
10 app.get("/auth", (req, res) => {
11     req.session.codeVerifier = generators.codeVerifier();
12     const code_challenge =
13         ↪ generators.codeChallenge(req.session.codeVerifier);
14     const authUrl =
15         oidcClient.authorizationUrl({
16             scope: "openid profile email",
17             code_challenge: code_challenge,
18             code_challenge_method: "S256",
19         }) + "&kc_idp_hint=saml";
20     res.redirect(authUrl);
21 });
22
23 app.get("/auth/cb", async (req, res) => {
24     const params = oidcClient.callbackParams(req);
25     try {
26         const tokenSet = await oidcClient.callback(env.WEB_REDIRECT_URI,
27             ↪ params, {
28                 code_verifier: req.session.codeVerifier,
29             });
30         if (!tokenSet.claims().id) {
31             res.status(401).send();
32             return;
33         }
34         req.session.idToken = tokenSet.id_token;
35         req.session.userId = tokenSet.claims().id;
36         res.redirect("/profile");
37     } catch (error) {
38         res.status(401).send();
39     }
40 });

```

Αρχικά δημιουργούμε τον OpenID Client που θα είναι μέρος του server μας εισάγοντας τα στοιχεία που ορίσαμε κατά την εγγραφή στον OpenID Provider του Keycloak. Όπως είδαμε προηγουμένως, όταν απαιτείται η ταυτοποίηση ενός χρήστη μεταβαίνουμε στο route `/auth`. Από εκεί κάνουμε `redirect` τον χρήστη στο URL του provider όπου θα πραγματοποιηθεί η ταυτοποίηση, περνώντας το λεγόμενο `code challenge` που είναι κρίσιμο για τη λειτουργία του PKCE. Μόλις η ταυτοποίηση ολοκληρωθεί, ο OpenID Provider ανακατευθύνει τον χρήστη στο URL `/auth/cb` της ιστοσελίδας μας, συμπεριλαμβάνοντας μία σειρά από παραμέτρους στο URL, και ο server μπορεί χρησιμοποιώντας αυτές τις παραμέτρους να ζητήσει το ID token του χρήστη μέσω ενός POST request στον OpenID provider. Το ID token αποθηκεύεται στο session του χρήστη, και ο χρήστης μεταβαίνει τελικά στη σελίδα προφίλ του.

Τέλος, έχουμε τις δύο δράσεις της σελίδας προφίλ, την αποσύνδεση και τη

διαγραφή λογαριασμού:

```
website.ts
1  app.post("/profile/logout", (req, res) => {
2    const logoutUrl = oidcClient.endSessionUrl({
3      post_logout_redirect_uri: env.WEB_LOGOUT_REDIRECT_URI,
4      id_token_hint: req.session.idToken,
5    });
6    res.redirect(logoutUrl);
7    req.session.destroy((err) => {
8      if (err) loggerMain.error(err);
9    });
10  });
11
12  app.post("/profile/delete", async (req, res) => {
13    if (!req.session.userId) {
14      res.status(401).send();
15      return;
16    }
17    await removeUser(req.session.userId);
18    res.redirect("/profile/logout");
19  });
```

Κατά την αποσύνδεση, ο server απλά οδηγεί τον χρήστη στο κατάλληλο URL του OpenID Provider το οποίο θα τον αποσυνδέσει από τον λογαριασμό του, και έπειτα καταστρέφει το αντίστοιχο session. Αντίστοιχα όταν ο χρήστης πατάει του κουμπι διαγραφής τότε διαγράφουμε τον λογαριασμό του από τη βάση δεδομένων και προχωράμε στην αποσύνδεσή του.

Για την υλοποίηση των sessions χρησιμοποιούμε τη NoSQL βάση δεδομένων Redis. Το κάθε session αποθηκεύεται ως ένα key-value pair, με το session ID να χρησιμοποιείται ως key και το αντίστοιχο session cookie ως value.

Μία σημαντική λεπτομέρεια που πρέπει να τονίσουμε είναι το attribute SameSite που θέτουμε ως "Lax" στο session cookie. Το attribute αυτό πληροφορεί τον browser του χρήστη για το πως να χειριστεί την αποστολή του συγκεκριμένου cookie στην περίπτωση request που προέρχεται από διαφορετική ιστοσελίδα. Συγκεκριμένα, η ρύθμιση Lax σημαίνει ότι αν μία τυχαία ιστοσελίδα προσπαθήσει να στείλει ένα POST request στην ιστοσελίδα μας, τότε ο browser δεν θα μεταφέρει μαζί με το request το session cookie του χρήστη.

Το παραπάνω αποτελεί μέθοδο προστασίας έναντι επιθέσεων μέσω cross-site request forgery, και χρησιμοποιείται συγκεκριμένα για τη δράση του κουμπιού διαγραφής λογαριασμού. Το πάτημα το κουμπιού διαγραφής στέλνει απλά

ένα POST request στο κατάλληλο route της εφαρμογής το οποίο εκτελεί τη διαγραφή του χρήστη από τη βάση δεδομένων. Αν ο χρήστης επισκεφτεί μία ξένη κακόβουλη ιστοσελίδα (cross-site) τότε αυτή μπορεί πολύ εύκολα να στείλει το ίδιο ακριβώς request εκ μέρους του, και αν ο browser συμπεριλάβει το session cookie με αυτό το request τότε η ιστοσελίδα μπορεί να διαγράψει τον λογαριασμό του χρήστη εν αγνοία του. Το πρόβλημα λύνεται με τον ορισμό του cookie ως Lax, καθώς σε αυτή την περίπτωση ο browser του χρήστη δεν θα στείλει το αντίστοιχο session cookie με το POST request, και επομένως δε θα πραγματοποιήσει την καταστροφική πράξη.

website.ts

```
1 declare module "express-session" {
2   export interface SessionData {
3     codeVerifier: string;
4     idToken: string;
5     userId: string;
6   }
7 }
8
9 let redisClient = createClient();
10 redisClient.connect().catch(loggerMain.error);
11 let redisStore = new RedisStore({ client: redisClient, prefix:
    ↪ "ridehailing" });
12
13 app.use(
14   session({
15     store: redisStore,
16     secret: env.SESSION_SECRET,
17     resave: false,
18     saveUninitialized: false,
19     cookie: { secure: true, httpOnly: true, sameSite: "lax" },
20   })
21 );
```

← My Account



ID: el19012

Name: Efstratios Giakoumakis

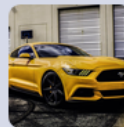
Rating: ★★☆☆☆ (5)

Vehicles

Model: Ford Focus
License plate: IAP-1589
Seats: 3
Color:



Model: Audi A5
License plate: III-1234
Seats: 2
Color:



Sign out

Delete account

Σχήμα Δ'.1: Η σελίδα προφίλ όπου εμφανίζονται όλα τα στοιχεία ενός χρήστη. Το URL της ιστοσελίδας είναι <https://ntua-ridehailing.dslab.ece.ntua.gr/profile>

Βιβλιογραφία

- [1] Susan A. Shaheen, Nelson D. Chan και Teresa Gaynor. «Casual car-pooling in the San Francisco Bay Area: Understanding user characteristics, behaviors, and motivations». Στο: *Transport Policy* 51 (2016). Transit Investment and Land Development. Edited by Xinyu (Jason) Cao and Qisheng Pan & Shared Use Mobility Innovations. Edited by Susan Shaheen, σσ. 165–173. ISSN: 0967-070X. DOI: <https://doi.org/10.1016/j.tranpol.2016.01.003>. URL: <https://www.sciencedirect.com/science/article/pii/S0967070X16300038>.
- [2] Efstratios Giakoumakis. *Code for NTUA Ridehailing mobile app*. <https://github.com/StratosGiak/NTUA-Ridehailing-App>.
- [3] Efstratios Giakoumakis. *Code for NTUA Ridehailing server*. <https://github.com/StratosGiak/NTUA-Ridehailing-Server>.
- [4] Tim Sneath & contributors. *Flutter architectural overview*. <https://docs.flutter.dev/resources/architectural-overview>.
- [5] Shams Zakhour. *Inside Flutter*. <https://docs.flutter.dev/resources/inside-flutter>.
- [6] Rémi Rousselet. *Sky: An Experiment Writing Dart for Mobile (Dart Developer Summit 2015)*. <https://www.youtube.com/watch?v=PnIWl33YMwA>.
- [7] Mozilla. *MVC - MDN Web Docs*. <https://developer.mozilla.org/en-US/docs/Glossary/MVC>.
- [8] Martin Fowler. *GUI Architectures*. <https://martinfowler.com/eaDev/uiArchs.html>. 2006.
- [9] Filip Hracek & contributors. *Simple app state management*. <https://docs.flutter.dev/data-and-backend/state-mgmt/simple>.

- [10] Rémi Rousselet. *provider*. <https://pub.dev/packages/provider>.
- [11] Allen Wirfs-Brock και Brendan Eich. «JavaScript: the first 20 years». Στο: *Proc. ACM Program. Lang.* 4.HOPL (Ιούν. 2020). DOI: 10.1145/3386327. URL: <https://doi.org/10.1145/3386327>.
- [12] ECMA International. *Standard ECMA-262 - ECMAScript® 2024 Language Specification*. https://ecma-international.org/wp-content/uploads/ECMA-262_15th_edition_june_2024.pdf. Ιούν. 2024.
- [13] Mozilla. *JavaScript - MDN Web Docs*. <https://developer.mozilla.org/en-US/docs/Web/JavaScript>.
- [14] Mozilla. *JavaScript - MDN Web Docs*. <https://developer.mozilla.org/en-US/docs/Glossary/TypeScript>.
- [15] Ryan Dahl. *Node.js presentation*. <https://www.youtube.com/watch?v=EeYvFl7li9E>.
- [16] Ryan Dahl. *Node.js presentation slides*. <https://s3.amazonaws.com/four.livejournal/20091117/jsconf.pdf>.
- [17] <https://x.com/RichOnTheWeb/status/494959181871316992>.
- [18] OpenJS Foundation. *Don't Block the Event Loop*. <https://nodejs.org/en/learn/asynchronous-work/dont-block-the-event-loop>.
- [19] <https://aaronstannard.com/intro-to-nodejs-for-net-developers/>.
- [20] Kristofer Baxter. *Making Netflix.com Faster*. <https://netflixtechblog.com/making-netflix-com-faster-f95d15f2e972>.
- [21] PayPal Tech Blog Team. *Node.js at PayPal*. <https://medium.com/paypal-tech/node-js-at-paypal-4e2d1d08ce4f>.
- [22] Octav Druta. *Building With Node.js At LinkedIn*. <https://medium.com/building-with-x/building-with-node-js-at-linkedin-ae4ea6af12f2>.
- [23] OpenJS Foundation. *Node.js Helps NASA Keep Astronauts Safe and Data Accessible*. https://web.archive.org/web/20231127154528/https://openjsf.org/wp-content/uploads/sites/84/2020/02/Case_Study-Node.js-NASA.pdf.

- [24] Daniel Ehrenberg. *Bloomberg Invests in Node.js. Shouldn't You?* <https://www.nearform.com/insights/bloomberg-invests-in-node-js-shouldnt-you/>.
- [25] OpenJS Foundation. *How Uber Uses Node.js to Scale Their Business*. <https://web.archive.org/web/20170225065423/https://nodejs.org/static/documents/casestudies/Nodejs-at-Uber.pdf>.
- [26] Alexey Melnikov και Ian Fette. *The WebSocket Protocol*. RFC 6455. Δεκ. 2011. DOI: 10.17487/RFC6455. URL: <https://www.rfc-editor.org/info/rfc6455>.
- [27] Bosco Yung. *Open-NSFW 2*. <https://github.com/bhky/opennsfw2>. Έκδ. 0.13.7. Δεκ. 2023.
- [28] Jay Mahadeokar και Gerry Pesavento. *Open Sourcing a Deep Learning Solution for Detecting NSFW Images — yahoo! Engineering*. <https://yahooeng.tumblr.com/post/151148689421/open-sourcing-a-deep-learning-solution-for>. Σεπτ. 2016.
- [29] Justin Richer. *User Authentication with OAuth 2.0*. <https://oauth.net/articles/authentication>.
- [30] OpenID Foundation. *How OpenID Connect Works*. <https://openid.net/developers/how-connect-works>.
- [31] Okta. *OAuth 2.0 and OpenID Connect overview*. <https://developer.okta.com/docs/concepts/oauth-openid/#authorization-code-flow-with-pkce-flow>.
- [32] William Denniss και John Bradley. *OAuth 2.0 for Native Apps*. RFC 8252. Οκτ. 2017. DOI: 10.17487/RFC8252. URL: <https://www.rfc-editor.org/info/rfc8252>.