

ΙΟΝΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ

Τμήμα Πληροφορικής



πτυχιακή εργασία

Web εφαρμογή για κηδεμόνες κατοικίδιων και κτηνιάτρους για τη
διαχείριση της υγείας και της θεραπείας κατοικίδιων ζώων

Μαρία Παπαναγιώτου
Επιβλέπων: Μιχαήλ Στεφανιδάκης

18 Ιουνίου 2024

Επιβλέπων

Μιχαήλ Στεφανιδάκης, Επίκουρος Καθηγητής

Τμήμα Πληροφορικής Ιόνιο Πανεπιστήμιο

Τριμελής Επιτροπή

Μιχαήλ Στεφανιδάκης, Αναπληρωτής Καθηγητής

Τμήμα Πληροφορικής Ιόνιο Πανεπιστήμιο

Θεόδωρος Ανδρόνικος, Καθηγητής

Τμήμα Πληροφορικής Ιόνιο Πανεπιστήμιο

Δημήτριος Ρίγγας, Εργαστηριακό Διδακτικό Προσωπικό

Τμήμα Πληροφορικής Ιόνιο Πανεπιστήμιο

Περίληψη

Η παρούσα διπλωματική εργασία επικεντρώνεται στη δημιουργία μιας διαδικτυακής εφαρμογής διαχείρισης ιατρικών δεδομένων κατοικίδιων. Η εφαρμογή αυτή παρέχει ένα ολοκληρωμένο περιβάλλον για κηδεμόνες κατοικίδιων και κτηνιάτρους, επιτρέποντας την αποθήκευση, διαχείριση και διαμοιρασμό ιατρικών δεδομένων. Με βασικές λειτουργίες όπως αποθήκευση ιστορικού και επερχόμενων γεγονότων, η εφαρμογή δημιουργήθηκε για να συμβάλει στην αποτελεσματική φροντίδα των κατοικίδιων. Για τους κηδεμόνες κατοικίδιων, η εφαρμογή θα προσφέρει μια ασφαλή και αξιόπιστη λύση για την αποθήκευση και διαχείριση των ιατρικών δεδομένων των κατοικίδιων τους. Για τους κτηνιάτρους, η εφαρμογή θα προσφέρει έναν αποτελεσματικό τρόπο επικοινωνίας με τους κηδεμόνες κατοικίδιων και την παρακολούθηση της υγείας των κατοικίδιων τους. Η ανάπτυξη θα γίνει με τις τεχνολογίες Golang, MongoDB, Typescript και Next.js, εξασφαλίζοντας πρόσβαση από οποιαδήποτε συσκευή με πρόσβαση στο διαδίκτυο. Η εφαρμογή αναμένεται να προσφέρει σημαντικά οφέλη τόσο στους κηδεμόνες κατοικίδιων όσο και στους κτηνιάτρους.

Λέξεις κλειδιά: ηλεκτρονικό βιβλιάριο, ιατρικά δεδομένα, κατοικίδια, διαχείριση, ψηφιακή υγεία

Abstract

The current thesis focuses on creating a web application for managing pet medical data. This application provides a comprehensive environment for both pet owners and veterinarians, allowing the storage, management, and sharing of medical data. With core features such as storing medical history and upcoming events, the application is created to contribute to the effective care of pets.

For pet owners, the application will offer a secure and reliable solution for storing and managing their pets' medical data. For veterinarians, it will provide an effective means of communication with pet owners and effective monitoring the health of their pets. The development will be carried out using Golang, MongoDB, Typescript, and Next.js technologies, ensuring accessibility from any device with internet access. The application is anticipated to deliver significant benefits to both pet owners and veterinarians.

Περιεχόμενα

Περίληψη.....	3
Abstract.....	4
Περιεχόμενα.....	5
Κεφάλαιο 1ο - Εισαγωγή.....	7
1.1 Περιγραφή του προβλήματος.....	7
1.2 Στόχοι.....	7
1.3 Καθορισμός απαιτήσεων.....	8
1.3.1 Λειτουργικές Απαιτήσεις.....	8
1.3.2 Μη-Λειτουργικές Απαιτήσεις.....	9
1.4 Διάρθρωση Εργασίας.....	9
Κεφάλαιο 2ο - Εργαλεία.....	10
2.1 Golang.....	10
2.1.1 Gin Gonic.....	10
2.2 Next.js.....	11
2.2.1 HTML.....	11
2.2.2 Tailwind css.....	11
2.2.3 Typescript.....	11
2.2.4 Heroicons.....	12
2.3 MongoDB.....	12
2.4 Open API.....	12
Κεφάλαιο 3ο - Υλοποίηση.....	13
3.1 Υλοποίηση εφαρμογής παρασκηνίου (backend application).....	13
3.1.1 Πλαίσιο της εφαρμογής (Domain).....	15
3.1.1.1 Οντότητα χρήστη.....	15
3.1.1.2 Οντότητα κατοικιδίου.....	16
3.1.1.3 Οντότητα Record.....	17
3.1.2 Εφαρμογή (Application).....	18
3.1.2.1 Υπηρεσίες.....	18
3.1.2.1.1 Υπηρεσίες χρήστη.....	18
3.1.2.1.2 Υπηρεσίες κατοικιδίου.....	24
3.1.2.1.3 Υπηρεσίες Εγγραφής.....	29
3.1.2.1.4 Υλοποίηση εφαρμογής.....	34
3.1.3 Επίπεδο Υποδομών (Infrastructure Layer).....	36
3.1.3.1 REST server.....	36
3.1.3.2 Αποθετήρια (Repositories).....	41
3.1.3.3 Main Module.....	45
3.2 Υλοποίηση εφαρμογής πελάτη.....	47
3.2.1 Οθόνη σύνδεσης χρήστη.....	48
3.2.2 Οθόνη εγγραφής χρήστη.....	50
3.2.3 Οθόνη κατοικιδίων (Κύρια οθόνη).....	52
3.2.4 Οθόνη δημιουργίας/επεξεργασίας προφίλ κατοικιδίου.....	55
3.2.5 Οθόνη κατοικιδίου.....	58

3.2.6 Οθόνη αιζέντας.....	67
3.2.7 Οθόνη προφίλ χρήστη.....	69
Κεφάλαιο 4ο - Deployment.....	72
4.1 Building process.....	72
4.2 Δημιουργία βάσης δεδομένων.....	74
Κεφάλαιο 5ο - Συμπεράσματα και Μελλοντικές βελτιώσεις.....	76
5.1 Συμπεράσματα.....	76
5.2 Μελλοντικές βελτιώσεις.....	76
Βιβλιογραφία και Ιστογραφία.....	77

Κεφάλαιο 1ο - Εισαγωγή

Η εξασφάλιση της υγείας και της ευημερίας των κατοικίδιων αποτελεί έναν από τους κύριους στόχους τόσο των κηδεμόνων όσο και των κτηνιάτρων τους. Προκειμένου να επιτευχθεί ο στόχος αυτός, η αποτελεσματική συνεργασία και η σαφής επικοινωνία μεταξύ των κηδεμόνων και των κτηνιάτρων αναδεικνύονται ως κρίσιμοι παράγοντες [1].

Παράλληλα, η ταχεία εξέλιξη της τεχνολογίας και η ψηφιοποίηση των δεδομένων έχουν επηρεάσει τον τρόπο διαχείρισης των υγειονομικών πληροφοριών [2].

Σε αυτό το πλαίσιο η παρούσα εργασία αποσκοπεί στον σχεδιασμό και την ανάπτυξη μιας εφαρμογής διαδικτύου για τη διαχείριση και τον διαμοιρασμό αυτών των δεδομένων. Ο τελικός στόχος είναι να διευκολύνει τη συνεργασία κηδεμόνων και κτηνιάτρων, προσφέροντας ταυτόχρονα βελτιωμένη φροντίδα για τα κατοικίδια.

1.1 Περιγραφή του προβλήματος

Ένα από τα συνηθέστερα προβλήματα που μπορεί να αντιμετωπίσει ένας κηδεμόνας κατοικίδιου είναι η έλλειψη του βιβλιαρίου υγείας κατά τη διάρκεια μιας επίσκεψης στον κτηνίατρο ή ακόμα και η πλήρης απώλεια αυτού. Σε αμφότερες τις περιπτώσεις, προκύπτει είτε μερική είτε ολική απώλεια δεδομένων, προκαλώντας πολλαπλά προβλήματα τόσο για τον κηδεμόνα όσο και για τον κτηνίατρο[3].

Κάποια από αυτά είναι:

1. **Καθυστερήσεις στην Παροχή Φροντίδας:** Για να προγραμματιστούν επόμενες εξετάσεις, εμβόλια και θεραπείες ο κτηνίατρος χρειάζεται τις πληροφορίες από το βιβλιάριο υγείας. Αν αυτές οι πληροφορίες λείπουν ή δεν είναι ολοκληρωμένες, υπάρχει πιθανότητα να υπάρξουν καθυστερήσεις στην παροχή φροντίδας.
2. **Ανεπαρκής Αγωγή:** Χωρίς όλα τα απαραίτητα ιστορικά δεδομένα, ο κτηνίατρος αδυνατεί να γνωρίζει ποια εμβόλια ή θεραπείες έχουν ήδη δοθεί στο κατοικίδιο. Αυτό μπορεί να οδηγήσει σε υπερβολική ή ανεπαρκή αγωγή.
3. **Δυσκολίες με την Αναγνώριση Ταυτότητας:** Σε πολλές χώρες το βιβλιάριο υγείας λειτουργεί ως αναγνώριση της ταυτότητας του κατοικίδιου. Η απώλεια αυτού του εγγράφου μπορεί να δημιουργήσει προβλήματα με την εισαγωγή σε καταλύματα και μέσα μεταφοράς.

1.2 Στόχοι

Ο στόχος της παρούσας διπλωματικής εργασίας επικεντρώνεται στην ανάπτυξη μιας διαδικτυακής εφαρμογής που θα λειτουργεί ως κεντρικό σύστημα αποθήκευσης, διαχείρισης και διαμοιρασμού ιατρικών δεδομένων για κατοικίδια. Η εφαρμογή αυτή θα προσφέρει ένα ολοκληρωμένο περιβάλλον, επιτρέποντας τόσο στους κηδεμόνες όσο και στους κτηνιάτρους να αποθηκεύουν, να διαχειρίζονται και να μοιράζονται τα ιατρικά δεδομένα των κατοικίδιων.

Παράλληλα, η εφαρμογή θα λειτουργεί ως μέσο ενημέρωσης, προσφέροντας πληροφορίες σχετικά με επερχόμενα γεγονότα, όπως εμβόλια, εξετάσεις και θεραπείες. Στόχος είναι η δημιουργία ενός ευέλικτου και χρήσιμου εργαλείου που θα συνεισφέρει στη βελτίωση της φροντίδας και της υγείας των κατοικίδιων, παρέχοντας ταυτόχρονα εύκολη πρόσβαση και αξιόπιστη ανταλλαγή πληροφοριών μεταξύ των εμπλεκόμενων φορέων.

Η σχεδίαση του λογισμικού βασίστηκε στην ιδέα ότι η αποθήκευση ιατρικών δεδομένων σε ψηφιακή μορφή παρέχει πολλαπλά πλεονεκτήματα σε σύγκριση με τη συμβατική, χειρόγραφη μορφή. Ανάμεσα σε αυτά περιλαμβάνονται η ευκολία στη δημιουργία, πρόσβαση, διαμοιρασμό, ενημέρωση, και ανάκτηση τέτοιων εγγράφων. Αυτά τα χαρακτηριστικά καθιστούν τις ηλεκτρονικές καταγραφές ιατρικών δεδομένων συνήθως πιο ολοκληρωμένες και αποτελεσματικές σε ό,τι αφορά την ταχύτητα και την αποτελεσματικότητα [3].

1.3 Καθορισμός απαιτήσεων

Για την αποτελεσματική υλοποίηση ενός πληροφοριακού συστήματος, είναι ζωτικής σημασίας να καθοριστούν εμπροθέσμως οι λειτουργικές και μη λειτουργικές απαιτήσεις που θα καθοδηγήσουν τη διαδικασία ανάπτυξης. Οι λειτουργικές απαιτήσεις καταγράφουν τις βασικές λειτουργίες που πρέπει να εκτελέσει το σύστημα, ενώ οι μη λειτουργικές απαιτήσεις περιγράφουν τα χαρακτηριστικά που επηρεάζουν τη συνολική του απόδοση.

Η σαφής οριοθέτηση αυτών των απαιτήσεων αποτελεί τη βάση για τη συνεχή επιτυχή ανάπτυξη, εξασφαλίζοντας ότι το τελικό προϊόν θα πληροί τις προσδοκίες των χρηστών και θα είναι συμβατό με τις αναγκαίες προδιαγραφές.

1.3.1 Λειτουργικές Απαιτήσεις

Οι λειτουργικές απαιτήσεις καθορίζουν τα χαρακτηριστικά ή τις λειτουργίες που πρέπει να εκτελεί ένα λογισμικό.

Οι απαιτήσεις αυτές αφορούν τις βασικές λειτουργίες της εφαρμογής όπως:

1. Διαχείριση Χρηστών:

- 1.1. Δημιουργία προσωπικού λογαριασμού για κηδεμόνες ή κτηνίατρο.
- 1.2. Επεξεργασία προσωπικών στοιχείων.
- 1.3. Διαγραφή προσωπικού λογαριασμού.

2. Διαχείριση Κατοικιδίων:

- 2.1. Δημιουργία σελίδας για κάθε κατοικίδιο.
- 2.2. Επεξεργασία στοιχείων του κατοικιδίου.
- 2.3. Διαγραφή σελίδας κατοικιδίου.

3. Κοινή Χρήση Δεδομένων: Δυνατότητα του κηδεμόνα να μοιραστεί τα δεδομένα του κατοικιδίου με κτηνίατρο.

4. Εγγραφές Δεδομένων Υγείας: Προσθήκη, επεξεργασία και διαγραφή εγγραφών υγείας του κατοικιδίου.

5. Εγγραφές Δεδομένων Βάρους: Προσθήκη, επεξεργασία και διαγραφή εγγραφών βάρους του κατοικιδίου.

6. Επιβεβαίωση Δεδομένων από Κτηνίατρο: Δυνατότητα να επιβεβαιώσει τις εγγραφές υγείας που προστέθηκαν από τον κηδεμόνα ένας κτηνίατρος.

7. Ορισμός και ενημέρωση επερχόμενων γεγονότων:

- 7.1. Ορισμός επερχόμενων γεγονότων από τον χρήστη.
- 7.2. Ενημέρωση για επερχόμενα γεγονότα.

1.3.2 Μη-Λειτουργικές Απαιτήσεις

Οι μη λειτουργικές απαιτήσεις καθορίζουν πώς πρέπει να συμπεριφέρεται ένα σύστημα λογισμικού και όχι τι πρέπει να κάνει. Αυτές οι απαιτήσεις είναι εξίσου σημαντικές και πρέπει να λαμβάνονται υπ' όψιν κατά την ανάπτυξη και τη συντήρηση ενός ιστότοπου. Ορισμένες από τις κύριες μη λειτουργικές απαιτήσεις περιλαμβάνουν:

1. **Απόδοση (Performance):** Ο ιστότοπος πρέπει να φορτώνει γρήγορα και να λειτουργεί ομαλά, ώστε οι χρήστες να μην αντιμετωπίζουν καθυστερήσεις ή προβλήματα.
2. **Κλιμακωσιμότητα (Scalability):** Ο ιστότοπος πρέπει να είναι σχεδιασμένος για να αντιμετωπίσει αυξημένα φορτία και χρήστες χωρίς να υποβαθμιστεί η απόδοσή του.
3. **Ασφάλεια (Security):** Ο ιστότοπος πρέπει να προστατεύεται από απειλές όπως κακόβουλο λογισμικό, επιθέσεις hacking και κλοπή δεδομένων.
4. **Διαθεσιμότητα (Availability):** Ο ιστότοπος πρέπει να είναι διαθέσιμος σχεδόν συνεχώς, χωρίς μακρές περιόδους αδιαθεσίας.
5. **Συμβατότητα (Compatibility):** Ο ιστότοπος πρέπει να λειτουργεί σωστά σε διάφορους τύπους περιληγμάτων και συσκευών, όπως κινητά τηλέφωνα, tablet και υπολογιστές.
6. **Διακείμενη εμπειρία χρήστη (User Experience - UX):** Ο ιστότοπος πρέπει να παρέχει μια ευχάριστη εμπειρία στους χρήστες, περιλαμβανομένου του σχεδιασμού, της πλοήγησης και της απεικόνισης.

Αυτές οι απαιτήσεις είναι σημαντικές για να εξασφαλιστεί ότι η εφαρμογή παρέχει υψηλής ποιότητας εμπειρία στους χρήστες και ότι λειτουργεί αποτελεσματικά και με ασφάλεια.

1.4 Διάρθρωση Εργασίας

Στο πρώτο κεφάλαιο περιγράφεται το πρόβλημα που λύνει η εργασία αυτή, καθώς και οι τελικοί στόχοι της. Τέλος παρουσιάζονται οι απαιτήσεις που πρέπει να πληροί η τελική εφαρμογή.

Στο δεύτερο κεφάλαιο θα περιγραφούν αναλυτικά τα εργαλεία που χρησιμοποιήθηκαν για την ανάπτυξη της εφαρμογής.

Συνεχίζοντας στο τρίτο κεφάλαιο αναπτύσσεται η υλοποίηση της εφαρμογής αναλυτικά. Παρουσιάζεται η εφαρμογή παρασκήνιου και η εφαρμογή πελάτη που είναι τα δύο βασικά μέρη της συνολικής εφαρμογής.

Το τέταρτο κεφάλαιο περιγράφει αναλυτικά τα βήματα ώστε να παραχθεί το εκτελέσιμο αρχείο της τελικής εφαρμογής.

Τέλος στο πέμπτο και τελευταίο κεφάλαιο ολοκληρώνεται η εργασία παρουσιάζοντας τα συμπεράσματα και πιθανές μελλοντικές βελτιώσεις.

Κεφαλαίο 2ο - Εργαλεία

Σε αυτή την ενότητα περιγράφονται αναλυτικά τα βασικά εργαλεία που χρησιμοποιήθηκαν στην ανάπτυξη της παρούσας εργασίας. Επιγραμματικά χρησιμοποιήθηκε η γλώσσα go για την υλοποίηση του διακομιστή και η mongodb για την αποθήκευση των δεδομένων, το nextjs σε συνδυασμό με τη TypeScript και το framework tailwind css για την ανάπτυξη της διεπαφής πελάτη.

Παράλληλα με αυτά τα εργαλεία χρησιμοποιήθηκαν βιβλιοθήκες ανοιχτού λογισμικού τόσο για την εφαρμογή διακομιστή (server application) όσο και τη διεπαφή πελάτη (user interface).

2.1 Golang

Η Go [4] επίσης γνωστή και ως Golang, είναι μία γλώσσα ανοιχτού κώδικα ανεπτυγμένη από την google. Χρησιμοποιείται ευρέως από προγραμματιστές για ανάπτυξη λογισμικού όπως εφαρμογές διαδικτύου (web applications), υπηρεσίες νέφους (cloud services), διεπαφές τερματικού (command-line interfaces) και πολλά άλλα είδη εφαρμογών.

Έχει ένα ανεπτυγμένο οικοσύστημα από συνεργάτες, κοινότητες και εργαλεία, όπου σε συνδυασμό με την ευκολία στην εκμάθηση της, την αποτελεσματική χρήση πολλαπλών νημάτων (concurrency), καθώς και τη γρήγορη εκτέλεση κώδικα, την καθιστά μια ιδανική επιλογή για την ανάπτυξη αυτής της εργασίας.

Κάποια από τα βασικά στοιχεία που προσφέρει η Go για ανάπτυξη εφαρμογών διαδικτύου (web applications) είναι:

- Ανεξάρτητο πλατφόρμας λογισμικό (cross-platform application)
- Πληθώρα παραδειγμάτων και υλικού για την εκμάθηση της γλώσσας
- Χρήση πολλαπλών νημάτων (threads)
- Πληθώρα βιβλιοθηκών ανοιχτού κώδικα
- Παραγωγή ενός εκτελέσιμου αρχείου για όλη την εφαρμογή
- Απόδοση

Σε συνδυασμό με τις βασικές λειτουργίες της γλώσσας για την ανάπτυξη του λογισμικού χρησιμοποιήθηκε και μια βιβλιοθήκη ανοιχτού κώδικα βασισμένη σε go, για την υποστήριξη της ανάπτυξης των υπηρεσιών ιστού, των πόρων καθώς και της διεπαφής προγραμματισμού εφαρμογών (API).

2.1.1 Gin Gonic

Η βιβλιοθήκη του gin gonic [5] έχει αναπτυχθεί σε go. Είναι ένα web framework όπου έχει φτιαχτεί για να προσφέρει διευκόλυνση στην ανάπτυξη εφαρμογών διαδικτύου.

Η βιβλιοθήκη προσφέρει [6]:

- **Υποστήριξη ενδιάμεσου λογισμικού (middleware support):** Ένα http αίτημα, πριν την κυρία λειτουργία του μπορεί να χρειαστεί να εκτελέσει μια σειρά από ενδιάμεσες λειτουργίες, όπως logging, εξουσιοδότηση, αυθεντικοποίηση και άλλα κομμάτια κοινού κώδικα.
- **Ομαδοποίηση μονοπατιών (routes grouping):** Πολλές φορές είναι αναγκαίο να ομαδοποιηθούν κάποια μονοπάτια (routes). Για παράδειγμα υπάρχουν μονοπάτια που χρειάζεται αυθεντικοποίηση (authentication) για να αποκτήσει κάποιος πρόσβαση, καθώς και μονοπάτια που είναι προσβάσιμα σε όλους.
- **Διαχείριση σφαλμάτων:** Ένας τρόπος να συλλέγονται όλα τα σφάλματα που συνέβησαν κατά τη διαδικασία διεκπεραίωσης ενός αιτήματος.
- **Ενσωματωμένο rendering**
- **Ταχύτητα διεκπεραίωσης αιτημάτων**
- **Διαχείριση σφαλμάτων για ομαλή λειτουργία του διακομιστή**

2.2 Next.js

Το Next.js [7] είναι ένα framework της react για την ανάπτυξη εφαρμογών. Αναπτύχθηκε από την εταιρεία vercel καθώς και από την κοινότητα της react [8]. Η βασική ιδέα πίσω από το Next.js είναι η απλή και αποτελεσματική ανάπτυξη react εφαρμογών, φιλικές στη βελτιστοποίηση ιστοσελίδων για μηχανές αναζήτησης (SEO-friendly).

Μερικά από τα βασικά χαρακτηριστικά [9] του Next.js είναι τα εξής:

- **Rendering μέσω server (server side rendering):** Περιλαμβάνει τον δικό του route handler που βοηθά στη γρήγορη πλοήγηση μεταξύ σελίδων.
- **Παραγωγή στατικών σελίδων (static site generation):** Μπορεί να δημιουργήσει στατικές σελίδες όπου μπορούν να επαναχρησιμοποιηθούν σε κάθε http αίτημα και να αποθηκευτούν σε cdn.
- **Αυτόματη δημιουργία μονοπατιών:** Για τη διευκόλυνση πλοήγησης μεταξύ των σελίδων της εφαρμογής.
- **Υποστήριξη κοινών χαρακτηριστικών μεταξύ σελίδων:** Υποστήριξη για κοινές λειτουργίες όπως αυθεντικοποίηση, χειρισμό σφαλμάτων και διεθνοποίηση (localization).

Για την παραγωγή των σελίδων της react εφαρμογής χρησιμοποιούνται επίσης τα βασικά δομικά στοιχεία μιας σελίδας διαδικτύου. Επιγραμματικά, η HTML για οργάνωση του περιεχομένου, το css μέσω του Tailwind framework για τη μορφοποίηση της εμφάνισης καθώς και η TypeScript για την προσθήκη δυναμικής συμπεριφοράς στη σελίδα.

2.2.1 HTML

Η HTML [10] είναι υπεύθυνη για την οργάνωση του περιεχομένου μιας διαδικτυακής σελίδας. Ορίζει τα στοιχεία που θα εμφανιστούν στη σελίδα, καθώς και τη θέση και την εμφάνιση αυτών των στοιχείων.

Τα βασικά στοιχεία [11] HTML περιλαμβάνουν:

- **Ετικέτες:** Οι ετικέτες είναι οι λέξεις κλειδιά που χρησιμοποιούνται για να ορίσουν τα στοιχεία HTML.
- **Χαρακτηριστικά:** Τα Χαρακτηριστικά είναι οι παράμετροι που μπορούν να χρησιμοποιηθούν για να προσαρμόσουν τη συμπεριφορά ενός στοιχείου HTML.
- **Περιεχόμενο:** Το περιεχόμενο είναι το κείμενο, οι εικόνες, τα βίντεο και άλλα στοιχεία που εμφανίζονται στη σελίδα.

2.2.2 Tailwind css

Το Tailwind css [12] είναι ένα css framework σχεδιασμένο για τη δημιουργία όμορφων και λειτουργικών εφαρμογών γρήγορα και εύκολα. Είναι μια καλή επιλογή για την αύξηση παραγωγικότητας, καθώς προσφέρει μια πληθώρα από παραδείγματα και έτοιμα στοιχεία [13].

Τα πλεονεκτήματα του Tailwind css, συμπεριλαμβανομένων:

- **Μείωση του κώδικα:** Το Tailwind χρησιμοποιεί κατηγορίες βοηθητικών κλάσεων για να ελέγξει την εμφάνιση των στοιχείων της HTML, γεγονός που μπορεί να βοηθήσει στη μείωση του κώδικα.
- **Ευκολία χρήσης:** Το Tailwind είναι εύκολο στην εκμάθηση και τη χρήση του, ακόμη και χωρίς να υπάρχει προηγούμενη εμπειρία με το css.
- **Βιβλιοθήκη με έτοιμα στοιχεία:** Το Tailwind περιέχει μια πληθώρα από έτοιμα στοιχεία που μπορούν να χρησιμοποιηθούν ως πρότυπα.

2.2.3 Typescript

Η TypeScript [14] είναι μια γλώσσα προγραμματισμού υψηλού επιπέδου που αναπτύχθηκε από τη Microsoft. Είναι μια επέκταση της JavaScript που προσθέτει ένα σύστημα τύπων (type system)[15]. Αυτό σημαίνει ότι ο προγραμματιστής πρέπει να δηλώσει τον τύπο των μεταβλητών, των συναρτήσεων και των άλλων στοιχείων του κώδικα.

Η χρήση σταθερών τύπων μεταβλητών, προσφέρει μια σειρά από πλεονεκτήματα, όπως:

- **Αυξημένη ασφάλεια:** Η ύπαρξη τύπων στις μεταβλητές μπορεί να βοηθήσει στην αποφυγή σφαλμάτων τα οποία μπορούν να οδηγήσουν σε ασταθείς ή μη προβλέψιμες εφαρμογές.
- **Ευκολία στη συντήρηση:** Βοηθά στην κατανόηση του κώδικα, γεγονός που μπορεί να διευκολύνει τη συντήρησή του.
- **Αυξημένη παραγωγικότητα:** Βοηθά τους προγραμματιστές να γράφουν πιο αποτελεσματικά κώδικα.

Η TypeScript είναι μια δημοφιλής γλώσσα προγραμματισμού που χρησιμοποιείται για την ανάπτυξη εφαρμογών ιστού.

2.2.4 Heroicons

Τα Heroicons [16] είναι ένα σύνολο εικονιδίων σχεδιασμένα για να χρησιμοποιούνται σε ιστοσελίδες και εφαρμογές. Δημιουργήθηκαν από την εταιρεία Heroicons και είναι διαθέσιμα δωρεάν για χρήση και προσαρμογή.

2.3 MongoDB

Η MongoDB [17] είναι μια μη σχεσιακή βάση δεδομένων που αποθηκεύει δεδομένα σε έγγραφα σε μία μορφή παρόμοια με JSON [18]. Είναι γνωστή για την ευελιξία, την κλιμακωσιμότητα και την απόδοσή που προσφέρει.

Κάποια από τα βασικά χαρακτηριστικά της MongoDB είναι:

- **Δομή βασισμένη σε έγγραφα:** Η MongoDB αποθηκεύει δεδομένα σε έγγραφα μορφής JSON. Αυτό καθιστά εύκολη την αποθήκευση και την αναζήτηση σύνθετων δομών δεδομένων.
- **Ευέλικτο σχήμα:** Τα έγγραφα δεν χρειάζεται να έχουν ένα σταθερό σχήμα, πράγμα που σημαίνει ότι μπορούν να προστεθούν ή να αφαιρεθούν πεδία ανάλογα με τις απαιτήσεις. Αυτό καθιστά εύκολη την προσαρμογή του σχήματος στις μεταβαλλόμενες απαιτήσεις των δεδομένων.
- **Κλιμακωσιμότητα:** Η MongoDB μπορεί να κλιμακωθεί οριζόντια προσθέτοντας περισσότερους διακομιστές. Αυτό την καθιστά ιδανική για εφαρμογές που πρέπει να χειριστούν μεγάλες ποσότητες επισκεψιμότητας.
- **Απόδοση:** Χρησιμοποιεί διάφορες τεχνικές για να βελτιστοποιήσει την πρόσβαση στα δεδομένα και την επεξεργασία ερωτημάτων.

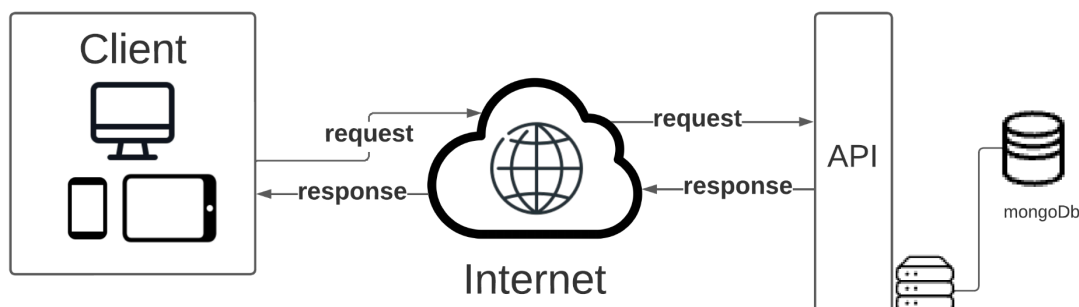
2.4 Open API

Το OpenAPI [19] είναι ένα εργαλείο που χρησιμοποιείται για τον σχεδιασμό, την τεκμηρίωση και την ανάπτυξη διασυνδέσεων με έμφαση στην αυτοματοποίηση και την επαναχρησιμοποίηση κώδικα. Το OpenAPI παρέχει ένα πρότυπο περιγραφής για τον σχεδιασμό του API της εφαρμογής, το οποίο περιγράφει τις διαθέσιμες ενέργειες, τους πόρους, τα μοντέλα δεδομένων και τις παραμέτρους που μπορούν να χρησιμοποιηθούν.

Κεφάλαιο 3ο - Υλοποίηση

Στο τρίτο κεφάλαιο θα εξετάσουμε λεπτομερώς την υλοποίηση της εφαρμογής. Σε αυτό το κεφάλαιο θα αναπτυχθούν η αρχιτεκτονική, καθώς και ο τρόπος που χρησιμοποιήθηκαν τα εργαλεία και οι τεχνολογίες που αναφέρθηκαν στο δεύτερο κεφάλαιο για να αναπτυχθεί το κάθε μέρος.

Η εφαρμογή αποτελείται από δύο βασικά μέρη, την εφαρμογή παρασκηνίου και τη διεπαφή πελάτη. Τα δύο μέρη επικοινωνούν μεταξύ τους μέσω ενός RESTful API [20].



Εικόνα 1 Επικοινωνία πελάτη - εξυπηρετητή

Για κάθε ένα από τα μέρη της εφαρμογής θα διερευνήσουμε τόσο τις διαδικασίες ανάπτυξης, όσο και τις τεχνικές που εφαρμόστηκαν ώστε να επιτευχθούν οι στόχοι της εφαρμογής. Συνεπώς κατά την ανάγνωση αυτής της ενότητας, θα κατανοήσετε πώς διαμορφώθηκε η εφαρμογή και πώς λειτουργεί.

3.1 Υλοποίηση εφαρμογής παρασκηνίου (backend application)

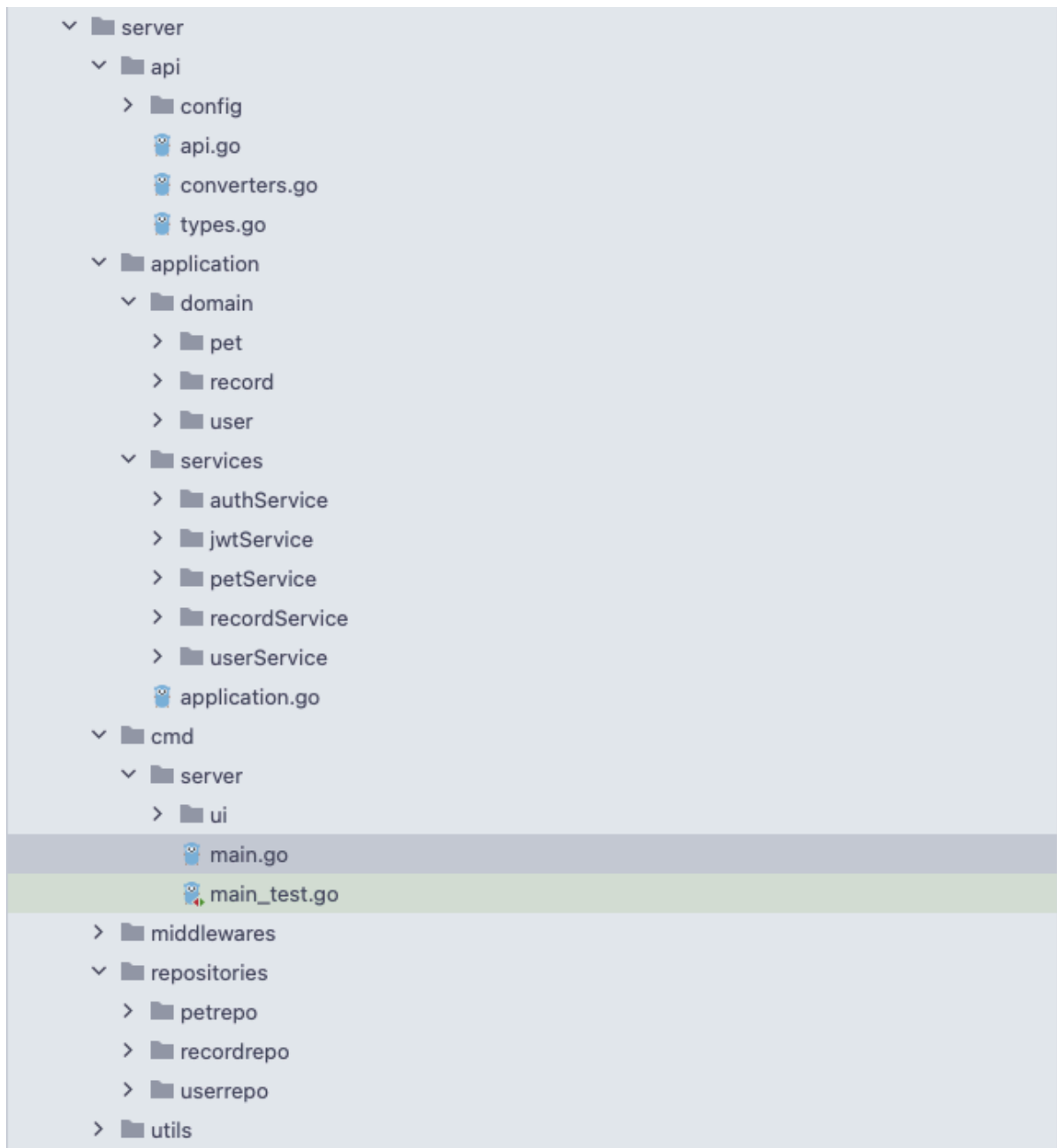
Η εφαρμογή παρασκηνίου (backend) αναπτύχθηκε χρησιμοποιώντας τη γλώσσα προγραμματισμού Golang, ακολουθώντας τις αρχές του Domain Driven Design (DDD) [21].

Το Domain Driven Design επιλέχθηκε προκειμένου να δημιουργηθεί μια ευέλικτη, εύκολη στην υλοποίηση και συντήρηση εφαρμογή.

Επιγραμματικά, το Domain Driven Design έχει τρία κύρια επίπεδα εστίασης. Ως πρώτο επίπεδο έχουμε το domain που αποτελεί τον πυρήνα της εφαρμογής. Το domain περικλείει τις οντότητες που δημιουργήθηκαν για να αναπαραστήσουν τις έννοιες και τις συμπεριφορές που σχετίζονται με το πρόβλημα.

Ένα επίπεδο πάνω από τον πυρήνα βρίσκεται το επίπεδο της εφαρμογής. Το επίπεδο εφαρμογής επικεντρώνεται κυρίως στον συντονισμό των ενεργειών μεταξύ της διεπαφής χρήστη και του domain επιπέδου. Για το σκοπό αυτό χρησιμοποιεί τις υπηρεσίες εφαρμογής που συνδέουν τις περιπτώσεις χρήσης της εφαρμογής με τις οντότητες του πυρήνα.

Τέλος, το εξωτερικό στρώμα είναι το στρώμα υποδομής (infrastructure), εκεί έχουμε όλες τις τεχνικές υπηρεσίες που απαιτούνται για την υποστήριξη των άλλων στρωμάτων. Στην περίπτωση μας, σε αυτό το επίπεδο βρίσκονται: Τα αποθετήρια, που διαχειρίζονται τον τρόπο με τον οποίο τα δεδομένα αποθηκεύονται και ανακτώνται από τη βάση δεδομένων, τον REST server, που διαχειρίζεται τα αιτήματα api και τα μετατρέπει σε λογική της εφαρμογής και τέλος τη διεπαφή χρήστη.



Εικόνα 2 Δομή του κώδικα παρασκηνίου

3.1.1 Πλαίσιο της εφαρμογής (Domain)

Σε αυτό το επίπεδο περιγράφονται οι βασικές οντότητες της εφαρμογής, όπως προκύπτουν από την ανάλυση των λειτουργιών απαιτήσεων.

Επιγραμματικά η υλοποίηση περιλαμβάνει τρεις βασικές οντότητες. Πρόκειται για τις οντότητες του χρήστη (User), του κατοικίδιου (Pet), καθώς και της εγγραφής (Record).

3.1.1.1 Οντότητα χρήστη

```
var types = map[Type]Type{
    Vet: Vet,
    Owner: Owner,
}

type User struct {
    Id          uuid.UUID
    CreatedAt   time.Time
    UpdatedAt   time.Time
    Deleted     bool
    UserType    Type
    Email       string
    PasswordHash string
    Name        string
    Surname     string
    Phone       string
    Address     string
    City        string
    State       string
    Country     string
    Zip         string
}
```

Κώδικας 1 Μοντέλο χρήστη

Το παραπάνω μοντέλο περιγράφει έναν χρήστη ως τύπο δεδομένων με όνομα "User". Το μοντέλο περιλαμβάνει ποικίλα πεδία για να μοντελοποιήσει έναν χρήστη ως εξής:

- **Id:** Μοναδικό αναγνωριστικό για κάθε χρήστη.
- **UserType:** Ένας χρήστης μπορεί να είναι είτε κηδεμόνας κάποιου κατοικίδιου είτε κτηνίατρος. Η κύρια διαφορά μεταξύ αυτών των δύο τύπων χρηστών είναι ότι ο κτηνίατρος έχει τη δυνατότητα να προσθέτει εγκεκριμένες έγγραφες, καθώς επίσης και να επικυρώνει τις εγγραφές που έχουν προστεθεί από άλλους χρήστες.
- **Στοιχεία σχετικά με την Αυθεντικοποίηση :**
 - **email:** Μοναδικό email ανα χρήστη.
 - **passwordHash :** Hash απο το επιλεγμένο συνθηματικό του χρήστη.
- **Στοιχεία προφίλ:**
 - **Όνομα:** Το όνομα του χρήστη.
 - **Επίθετο:** Το επίθετο του χρήστη.
 - **Στοιχεία επικοινωνίας:** Προαιρετικά στοιχεία επικοινωνίας για κάθε χρήστη. Η προσθήκη των στοιχείων αυτών προσθέτει αξία στην εφαρμογή, καθώς διευκολύνει την αμφίδρομη επικοινωνία μεταξύ χρηστών. Οι πληροφορίες αυτές μπορούν να χρησιμοποιηθούν για επικοινωνία από τον ιδιοκτήτη προς τον κτηνίατρο και αντίστροφα, δημιουργώντας έναν ολοκληρωμένο και αποτελεσματικό μηχανισμό επικοινωνίας.

3.1.1.2 Οντότητα κατοικιδίου

```
var types = map[Gender]Gender{
    M: M,
    F: F,
}
```

```
type Pet struct {
    Id          uuid.UUID
    CreatedAt   time.Time
    UpdatedAt   time.Time
    Deleted     bool
    Name        string
    Avatar      string
    DateOfBirth time.Time
    Gender      Gender
    BreedName   string
    Colors      []string
    Description string
    Pedigree    string
    Microchip   string
    OwnerId     uuid.UUID
    VetId       uuid.UUID
    Metas       map[string]string
}
```

Κώδικας 2 Μοντέλο κατοικιδίου

Το παραπάνω μοντέλο περιγράφει ένα κατοικίδιο ως τύπο δεδομένων με όνομα "Pet".

Αναλυτικά στο μοντέλο περιλαμβάνονται πληροφορίες για το κατοικίδιο:

- **Id:** Μοναδικό αναγνωριστικό του κατοικιδίου.
- **Στοιχεία σχετικά με το προφίλ :**
 - **DateOfBirth:** Η ημερομηνία καταγράφεται για την ακριβή γνώση της ηλικίας του κατοικιδίου.
 - **Gender:** Ορισμός του φύλου του κατοικιδίου, M για αρσενικό, F για θηλυκό.
 - **BreedName:** Η φυλή (ράτσα) του κατοικιδίου.
 - **Colors:** Πληροφορίες για τα χρώματα του τριχώματος.
 - **Description:** Μια σύντομη περιγραφή που μπορεί να περιλαμβάνει ιδιαιτερότητες και χαρακτηριστικά του κατοικιδίου, καθώς επίσης και σημαντικές πληροφορίες για την υγεία και τη φροντίδα του κατοικιδίου που θα έπρεπε να γνωρίζει ο κτηνίατρος.
 - **Pedigree:** Πληροφορίες σχετικά με την καταγωγή του κατοικιδίου εφόσον υπάρχουν.
 - **Microchip:** Ο αριθμός σήμανσης για την ταυτοποίηση του κατοικιδίου.
 - **Metas:** Επιπλέον πληροφορίες που μπορούν να αποθηκευτούν ως key-value στο πεδίο Metas, προσθέτοντας ευελιξία στην αποθήκευση προσαρμοσμένων πληροφοριών.
- **Avatar:** Μια εικόνα του κατοικιδίου για την ευκολότερη αναγνώριση.
- **Στοιχεία συσχέτισης:**
 - **OwnerId:** Αναπαριστά το μοναδικό αναγνωριστικό του χρήστη που δημιούργησε το προφίλ κατοικιδίου.
 - **VetId:** Αναπαριστά το μοναδικό αναγνωριστικό του κτηνιάτρου που αναλαμβάνει τη φροντίδα του κατοικιδίου.

Αυτό το μοντέλο προσφέρει μια πλήρη εικόνα του κατοικιδίου ως οντότητα, καθιστώντας δυνατή τη διεξαγωγή λεπτομερών και αποτελεσματικών διαδικασιών φροντίδας και διαχείρισης.

3.1.1.3 Οντότητα Record

```

type Record struct {
    Id          uuid.UUID
    CreatedAt   time.Time
    UpdatedAt   time.Time
    Deleted     bool
    PetId       uuid.UUID
    RecordType  Type
    Name        string
    Date        time.Time
    Lot         string
    Result      string
    Description string
    Notes       string
    AdministeredBy uuid.UUID
    VerifiedBy   uuid.UUID
    GroupId     uuid.UUID
}

var recordTypes = map[Type]Type{
    Vaccine:    Vaccine,
    Weight:     Weight,
    Temperature: Temperature,
    Nutering:   Nutering,
    Surgery:    Surgery,
    Medicine:   Medicine,
    Endoparasite: Endoparasite,
    Ectoparasite: Ectoparasite,
    Examination: Examination,
    Microchip:   Microchip,
    Diagnostic:  Diagnostic,
    Dental:      Dental,
    Other:       Other,
}

```

Κώδικας 3 Μοντέλο και είδη εγγραφής

Το παραπάνω μοντέλο περιγράφει μια εγγραφή ως τύπο δεδομένων με όνομα Record.

Αναλυτικά στο μοντέλο περιλαμβάνονται πληροφορίες για μια εγγραφή:

- **Id:** Μοναδικό αναγνωριστικό μια εγγραφής.
- **Στοιχεία σχετικά με μια εγγραφή:**
 - **RecordType:** Οι πιθανοί τύποι που μπορεί να έχει μια εγγραφή, όπως αυτοί καθορίζονται στο μοντέλο.
 - **Name:** Όνομα της εγγραφής.
 - **Date:** Η ημερομηνία που συνέβη ή έχει μελλοντικά προγραμματιστεί το συμβάν της εγγραφής.
 - **Lot:** Προαιρετικό πεδίο, συμπληρώνεται όταν η εγγραφή έχει αριθμό παρτίδας, όπως ένα εμβόλιο.
 - **Result:** Προαιρετικό πεδίο, Αποτέλεσμα της εγγραφής όπως αποτέλεσμα μιας διάγνωσης ή μιας εξέτασης. Υποχρεωτικό πεδίο όταν η εγγραφή είναι τύπου βάρους ή θερμοκρασίας.
 - **Description:** Προαιρετικό πεδίο, μια σύντομη περιγραφή.
 - **Notes:** Προαιρετικό πεδίο, σημειώσεις που αφορούν την εγγραφή.
- **Στοιχεία συσχέτισης:**
 - **PetId:** Κάθε εγγραφή συσχετίζεται πάντα με ένα κατοικίδιο μέσω του αναγνωριστικού (Id) του κατοικιδίου.
 - **AdministeredBy:** Σύνδεση με το αναγνωριστικό του χρήστη που τη δημιούργησε.
 - **VerifiedBy:** Το προαιρετικό αυτό πεδίο είναι η σύνδεση της εγγραφής με το αναγνωριστικό του κτηνιάτρου που την επικύρωσε. Μόνο οι κτηνίατροι έχουν την αρμοδιότητα να επικυρώσουν τις εγγραφές των κατοικιδίων.
 - **GroupId:** Μια εγγραφή μπορεί να σχετίζεται με ένα μελλοντικά προγραμματισμένο συμβάν.

3.1.2 Εφαρμογή (Application)

Το επόμενο επίπεδο που θα μελετήσουμε σε αυτό το κεφάλαιο είναι το επίπεδο της εφαρμογής.

Όπως αναφέρθηκε η εφαρμογή είναι ο συνδετικός κρίκος μεταξύ του domain και των εξωτερικών διεπαφών. Το επίπεδο αυτό είναι υπεύθυνο για τον συντονισμό των επιχειρησιακών διαδικασιών, τη διαχείριση της εκτέλεσης των λειτουργιών της εφαρμογής και την αλληλεπίδραση με τα υπόλοιπα επίπεδα.

Για τη διασύνδεσή του domain με τις εξωτερικές διεπαφές η εφαρμογή χρησιμοποιεί τα services που είναι βασισμένα στη διαμόρφωση κάθε μοντέλου.

```

/*
what the actor can do.
application talks with all the services
*/
type Application struct {
    petService    petService.Service
    userService   userService.Service
    recordService recordService.Service
}

```

Κώδικας 4 Μοντέλο της εφαρμογής

Για να μελετήσουμε σε βάθος τις λειτουργίες της εφαρμογής είναι απαραίτητο να αναπτυχθούν οι βασικές διεπαφές που την απαρτίζουν.

3.1.2.1 Υπηρεσίες

Οι υπηρεσίες συνδέουν το μοντέλο που υπάρχει στο domain με κάποια εξωτερική δομή αποθήκευσης, η οποία όμως υλοποιείται στο επόμενο επίπεδο των υποδομών (infrastructure layer). Ο τρόπος που επικοινωνούν μεταξύ τους γίνεται με dependency injection.

Το service εξαρτάται από ένα αποθετήριο, το οποίο είναι η δομή που διαχειρίζεται την αποθήκευση των πληροφοριών στη βάση δεδομένων.

Οι διεπαφές που υλοποιούνται είναι ένα συμβολικό συμβόλαιο μεταξύ της εφαρμογής και των υπηρεσιών, το οποίο ορίζει πώς αυτά τα συστατικά θα αλληλεπιδρούν μεταξύ τους.

3.1.2.1.1 Υπηρεσίες χρήστη

Οι υπηρεσίες χρήστη βασίζονται στο αποθετήριο του χρήστη το οποίο λαμβάνουν κατά τη δημιουργία τους.

```

type Service struct {
    repo user.Repository
}

func New(repo user.Repository) (Service, error) { return Service{repo: repo}, nil }

```

Κώδικας 5 Δημιουργία της διεπαφής χρήστη

- **Διεπαφή χρήστη**

Η διεπαφή χρήστη περιγράφει τις αλληλεπιδράσεις που μπορεί να έχει η υπηρεσία χρήστη.

```
type Service interface {
    User(id uuid.UUID) (user.User, error)
    Users(includeDel bool) ([]user.User, error)
    UsersByType(t user.Type, includeDel bool) ([]user.User, error)
    UserByType(id uuid.UUID, t user.Type, includeDel bool) (user.User, error)
    CreateUser(user services.UserCreateOptions) (user.User, string, error)
    UpdateUser(opts services.UserUpdateOptions, includeDel bool) (user.User, error)
    Authenticate(email string, password string) (user.User, string, error)
    DeleteUser(id uuid.UUID) error
    userByEmail(email string, includeDel bool) (user.User, bool)
    checkEmail(email string, id uuid.UUID, includeDel bool) error
}
```

Κώδικας 6 Διεπαφή υπηρεσιών χρήστη

- **Υλοποίησή υπηρεσιών χρήστη**

- **User:** Δέχεται ένα Id και επιστρέφει τον χρήστη που αντιστοιχεί, εάν υπάρχει. Αν ο χρήστης δεν βρεθεί, επιστρέφεται ένα σφάλμα.

```
func (s service) User(id uuid.UUID) (user.User, error) {
    return s.repo.User(id)
}
```

Κώδικας 7 Αναζήτηση χρήστη

- **Users:** Δέχεται σαν παράμετρο αν θα συμπεριλάβει τους διεγγραμμένους χρήστες στην λίστα, επιστρέφει όλους τους χρήστες που ταιριάζουν στα κριτήρια ή σφάλμα.

```
func (s *Service) Users(includeDel bool) ([]user.User, error) {
    return s.repo.Users(includeDel)
}
```

Κώδικας 8 Αναζήτηση χρηστών

- **UserByEmail:** Αυτή η μέθοδος είναι μια ιδιωτική συνάρτηση του service και δέχεται το email ενός χρήστη ως παράμετρο. Επιστρέφει τον χρήστη με αυτό το email και αν βρέθηκε.

```
func (s service) userByEmail(email string, includeDel bool) (user.User, bool) {
    var u user.User
    var found bool

    users, err := s.Users(includeDel)
    if err != nil { return user.Nil, false }
    for _, v := range users {
        if v.Email == email {
            u = v
            found = true
            break
        }
    }
    return u, found
}
```

Κώδικας 9 Αναζήτηση χρηστών με βάση το email

- **UsersByType:** Η μέθοδος δέχεται ως παράμετρο τον τύπο χρήστη. Στη συνέχεια, επιστρέφει όλους τους χρήστες που ταιριάζουν στον συγκεκριμένο τύπο χρήστη. Αν δεν βρεθούν χρήστες που να ταιριάζουν στα κριτήρια, επιστρέφεται ένα σφάλμα.

```
func (s *Service) UsersByType(t user.Type, includeDel bool) ([]user.User, error) {
    var users []user.User

    allUsers, err := s.Users(includeDel)
    if err != nil { return users, err }

    for _, u := range allUsers {
        if u.UserType == t {
            users = append(users, u)
        }
    }

    return users, nil
}
```

Κώδικας 10 Αναζήτηση χρηστών με βάση τον τύπο χρήστη

- **UserByType:** Η μέθοδος δέχεται ένα id και έναν τύπο χρήστη ως παραμέτρους. Στη συνέχεια, αναζητά στη λίστα των χρηστών που ταξινομούνται με βάση τον τύπο χρήστη. Επιστρέφει τον χρήστη που αντιστοιχεί στο συγκεκριμένο id εάν υπάρχει. Αν ο χρήστης δεν βρεθεί, επιστρέφεται ένα σφάλμα.

```
func (s *Service) UserByType(id uuid.UUID, t user.Type, includeDel bool) (user.User, error) {
    var u user.User

    users, err := s.UsersByType(t, includeDel)
    if err != nil { return user.Nil, err }
    err = user.ErrNotFound
    for _, v := range users {
        if v.Id == id {
            u = v
            err = nil
            break
        }
    }
    return u, err
}
```

Κώδικας 11 Αναζήτηση χρήστη με βάση τον τύπο χρήστη

- **Authenticate:** Η μέθοδος δέχεται σαν παράμετρο το email και το password. Αν τα στοιχεία είναι σωστά, αυθεντικοποιεί τον χρήστη που αντιστοιχούν σε αυτά. Στη συνέχεια, επιστρέφει τον αυθεντικοποιημένο χρήστη, ενώ εάν υπάρξει πρόβλημα κατά την αυθεντικοποίηση επιστρέφει ένα σφάλμα.

```
func (s *Service) Authenticate(email string, password string) (user.User, error) {
    var u, ok = s.UserByEmail(email, includeDel: true)
    if !ok { return user.User{}, user.ErrNotFound }

    if u.Deleted { return user.User{}, user.ErrUserDeleted }

    if !authService.CheckPasswordHash(password, u.PasswordHash) { return user.User{}, user.ErrAuthentication }

    return u, nil
}
```

Κώδικας 12 Αυθεντικοποίηση χρήστη

- **DeleteUser:** Η μέθοδος δέχεται το μοναδικό αναγνωριστικό (Id) ενός χρήστη ως παράμετρο και διαγράφει τον χρήστη αντίστοιχα. Επιστρέφει ένα σφάλμα εάν κάτι πάει λάθος κατά τη διαγραφή.

```
func (s *Service) DeleteUser(id uuid.UUID) error {
    return s.repo.DeleteUser(id)
}
```

Κώδικας 13 Διαγραφή χρήστη

- **CreateUser:** Η μέθοδος δέχεται τα options που παρέχουν τις πληροφορίες για τον νέο χρήστη που θα δημιουργηθεί. Έπειτα επιβεβαιώνει ότι είναι σωστά και δημιουργεί τον χρήστη. Τέλος, επιστρέφει τον νέο χρήστη ή ένα σφάλμα σε περίπτωση προβλήματος.

```
func (s service) CreateUser(opts services.UserCreateOptions) (user.User, string, error) {
    u := user.Nil

    typ, err := user.ParseType(opts.UserType)
    if err != nil { return u, "", user.ErrNoValidType }

    err = s.checkEmail(opts.Email, u.Id, includeDel: true)
    if err != nil { return u, "", err }

    err = passwordValidation(opts.Password)
    if err != nil { return u, "", err }

    hashed, err := authUtils.HashPassword(opts.Password)
    if err != nil { return u, "", err }

    if textUtils.TextIsEmpty(opts.Name) { return u, "", user.ErrNoValidName }

    if textUtils.TextIsEmpty(opts.Surname) { return u, "", user.ErrNoValidSurname }

    u.UserType = typ
    u.Email = opts.Email
    u.PasswordHash = hashed
    u.Name = opts.Name
    u.Surname = opts.Surname
    u.Phone = opts.Phone
    u.Address = opts.Address
    u.City = opts.City
    u.State = opts.State
    u.Country = opts.Country
    u.Zip = opts.Zip

    u, err = s.repo.CreateUser(u)
    if err != nil { return u, "", err }

    token, err := userToken(u)
    if err != nil { return u, token, err }

    return u, token, nil
}
```

Κώδικας 14 Δημιουργία χρήστη

- **UpdateUser:** Η μέθοδος δέχεται τα options που παρέχουν τις πληροφορίες για την ενημέρωση των στοιχείων του χρήστη. Επιβεβαιώνει ότι είναι σωστά και ενημερώνει τον χρήστη. Τέλος, επιστρέφει τον ενημερωμένο χρήστη ή ένα σφάλμα σε περίπτωση προβλήματος.

```
func (s service) UpdateUser(opts services.UserUpdateOptions, includeDel bool) (user.User, error) {
    u, err := s.User(opts.Id)
    if err != nil {
        return u, user.ErrNotFound
    }

    err = s.checkEmail(opts.Email, u.Id, includeDel)
    if err != nil {
        return u, err
    }

    if textUtils.TextIsEmpty(opts.Name) {
        return u, user.ErrNoValidName
    }

    if textUtils.TextIsEmpty(opts.Surname) {
        return u, user.ErrNoValidSurname
    }

    u.Email = opts.Email
    u.Name = opts.Name
    u.Surname = opts.Surname
    u.Phone = opts.Phone
    u.Address = opts.Address
    u.City = opts.City
    u.State = opts.State
    u.Country = opts.Country
    u.Zip = opts.Zip

    return s.repo.UpdateUser(u)
}
```

Κώδικας 15 Ανανέωση στοιχείων χρήστη

- **CheckEmail:** Αυτή η μέθοδος είναι μια ιδιωτική συνάρτηση του service. Δέχεται ένα email, το id ενός χρήστη και αν θα συμπεριλάβει τους διεγγραμμένους χρήστες. Επιστρέφει σφάλμα αν το email δεν είναι σωστό ή αν χρησιμοποιείται ήδη.

```
func (s service) checkEmail(email string, id uuid.UUID, includeDel bool) error {
    if !textUtils.IsEmailValid(email) {
        return user.ErrNoValidMail
    }

    u, ok := s.userByEmail(email, includeDel)

    if !ok { return nil }

    if u.Id == id { return nil }

    return user.ErrMailExists
}
```

Κώδικας 16 Επιβεβαίωση email

3.1.2.1.2 Υπηρεσίες κατοικιδίου

Οι υπηρεσίες των κατοικιδίων βασίζονται στο αποθετήριο του κατοικιδίου το οποίο λαμβάνουν κατά τη δημιουργία τους.

```
type Service struct {
    repo pet.Repository
}
```

```
func New(repo pet.Repository) (Service, error) { return Service{repo: repo}, nil }
```

Κώδικας 17 Δημιουργία των υπηρεσιών του κατοικιδίου

- **Διεπαφή κατοικιδίου**

Η διεπαφή του κατοικιδίου περιγράφει τις αλληλεπιδράσεις που μπορεί να έχει η υπηρεσία του.

```
type Service interface {
    Pet(id uuid.UUID) (pet.Pet, error)
    PetByUser(uid uuid.UUID, id uuid.UUID, includeDel bool) (pet.Pet, error)
    Pets(includeDel bool) ([]pet.Pet, error)
    PetsByUser(userId uuid.UUID, includeDel bool) (map[uuid.UUID]pet.Pet, error)
    CreatePet(opts services.PetCreateOptions) (pet.Pet, error)
    UpdatePet(opts services.PetUpdateOptions) (pet.Pet, error)
    DeletePet(uid uuid.UUID, id uuid.UUID) error
    removeVet(id uuid.UUID) error
    petsByOwner(userId uuid.UUID, includeDel bool) (map[uuid.UUID]pet.Pet, error)
    petByOwner(uid uuid.UUID, id uuid.UUID, includeDel bool) (pet.Pet, error)
}
```

Κώδικας 18 Διεπαφή κατοικιδίου

- **Υλοποίησή υπηρεσιών κατοικιδίου**

- ο **Pet**: Αυτή η μέθοδος δέχεται σαν παράμετρο το Id ενός κατοικιδίου και το επιστρέφει ή επιστρέφει ένα σφάλμα εάν δεν βρεθεί.

```
func (s service) Pet(id uuid.UUID) (pet.Pet, error) {
    return s.repo.Pet(id)
}
```

Κώδικας 19 Αναζήτηση κατοικιδίου

- ο **Pets**: Αυτή η μέθοδος δέχεται σαν παράμετρο εάν θα συμπεριλάβει τα διεγγραμμένα κατοικίδια στη λίστα και επιστρέφει τα κατοικίδια που ταιριάζουν στα κριτήρια ή ένα σφάλμα αν κάτι πάει λάθος.

```
func (s service) Pets(includeDel bool) ([]pet.Pet, error) {
    return s.repo.Pets(includeDel)
}
```

Κώδικας 20 Αναζήτηση κατοικιδίων

- **PetByUser:** Αυτή η μέθοδος δέχεται σαν παράμετρο το Id ενός χρήστη, το Id ενός κατοικίδιου και αν θα συμπεριλάβει διεγγραμμένα κατοικίδια. Επιστρέφει το κατοικίδιο αν ανήκει στον χρήστη, αλλιώς επιστρέφει ένα σφάλμα.

```
func (s *Service) PetByUser(uId uuid.UUID, id uuid.UUID, includeDel bool) (pet.Pet, error) {
    pets, err := s.PetsByUser(uId, includeDel)
    if err != nil { return pet.Nil, err }

    p, ok := pets[id]
    if !ok { return pet.Nil, pet.ErrNotFound }

    return p, nil
}
```

Κώδικας 21 Αναζήτηση κατοικίδιου με βάση ένα χρήστη

- **PetByOwner:** Αυτή η μέθοδος είναι μια ιδιωτική συνάρτηση του service. Δέχεται σαν παράμετρο το Id ενός χρήστη, το Id ενός κατοικίδιου και αν θα συμπεριλάβει διεγγραμμένα κατοικίδια. Επιστρέφει το κατοικίδιο αν ο χρήστης είναι ο κηδεμόνας, αλλιώς επιστρέφει ένα σφάλμα.

```
func (s service) petByOwner(uId uuid.UUID, id uuid.UUID, includeDel bool) (pet.Pet, error) {
    pets, err := s.petsByOwner(uId, includeDel)
    if err != nil { return pet.Nil, err }

    p, ok := pets[id]
    if !ok { return pet.Nil, pet.ErrNotFound }

    return p, nil
}
```

Κώδικας 22 Αναζήτηση κατοικίδιου με βάση τον κηδεμόνα τους

- **PetsByUser:** Δέχεται σαν παράμετρο το Id ενός χρήστη, το Id ενός κατοικίδιου και αν θα συμπεριλάβει διεγγραμμένα κατοικίδια. Επιστρέφει τα κατοικίδια που ταιριάζουν στα κριτήρια ή σφάλμα.

```
func (s service) PetsByUser(uId uuid.UUID, includeDel bool) (map[uuid.UUID]pet.Pet, error) {
    uPets := make(map[uuid.UUID]pet.Pet)

    pets, err := s.Pets(includeDel)
    if err != nil { return uPets, err }

    for _, p := range pets {
        if p.OwnerId == uId || p.VetId == uId {
            uPets[p.Id] = p
        }
    }

    return uPets, nil
}
```

Κώδικας 23 Αναζήτηση κατοικιδίων με βάση ένα χρήστη

- **PetsByOwner:** Αυτή η μέθοδος είναι μια ιδιωτική συνάρτηση του service. Δέχεται σαν παράμετρο το Id ενός χρήστη, το Id ενός κατοικίδιου και αν θα συμπεριλάβει διεγραμμένα κατοικίδια και επιστρέφει τα κατοικίδια που ο χρήστης είναι ο κηδεμόνας ή σφάλμα.

```
func (s service) petsByOwner(uId uuid.UUID, includeDel bool) (map[uuid.UUID]pet.Pet, error) {
    uPets := make(map[uuid.UUID]pet.Pet)

    pets, err := s.Pets(includeDel)
    if err != nil { return uPets, err }

    for _, p := range pets {
        if p.OwnerId == uId {
            uPets[p.Id] = p
        }
    }

    return uPets, nil
}
```

Κώδικας 24 Αναζήτηση κατοικιδίων με βάση τον κηδεμόνα τους

- **CreatePet:** Η μέθοδος δέχεται τα options που παρέχουν τις πληροφορίες για το νέο κατοικίδιο που θα δημιουργηθεί. Έπειτα επιβεβαιώνει ότι είναι σωστά και δημιουργεί το κατοικίδιο. Τέλος, επιστρέφει το νέο κατοικίδιο ή ένα σφάλμα σε περίπτωση προβλήματος.

```
func (s service) CreatePet(opts services.PetCreateOptions) (pet.Pet, error) {
    if textUtils.TextIsEmpty(opts.Name) { return pet.Nil, pet.ErrNoValidName }

    if opts.DateOfBirth.IsZero() { return pet.Nil, pet.ErrNoValidBirthDate }

    gender, err := pet.ParseGender(opts.Gender)
    if err != nil { return pet.Nil, err }

    if textUtils.TextIsEmpty(opts.BreedName) { return pet.Nil, pet.ErrNoValidBreedname }

    p := pet.Pet{}
    p.Name = opts.Name
    p.DateOfBirth = opts.DateOfBirth
    p.Gender = gender
    p.BreedName = opts.BreedName
    p.Colors = opts.Colors
    p.Description = opts.Description
    p.Pedigree = opts.Pedigree
    p.Microchip = opts.Microchip
    p.OwnerId = opts.OwnerId
    p.VetId = opts.VetId
    p.Metas = opts.Metas
    p.Avatar = opts.Avatar
    return s.repo.CreatePet(p)
}
```

Κώδικας 25 Δημιουργία κατοικιδίου

- **UpdatePet:** Η μέθοδος δέχεται τα options που παρέχουν τις πληροφορίες για την ενημέρωση των στοιχείων του κατοικίδιου. Επιβεβαιώνει ότι είναι σωστά και ενημερώνει το κατοικίδιο. Τέλος, επιστρέφει το ενημερωμένο κατοικίδιο ή ένα σφάλμα σε περίπτωση προβλήματος.

```
func (s service) UpdatePet(opts services.PetUpdateOptions) (pet.Pet, error) {
    p, err := s.PetByUser(opts.OwnerId, opts.Id, includeDel: false)
    if err != nil { return pet.Nil, err }

    if textUtils.TextIsEmpty(opts.Name) { return pet.Nil, pet.ErrNoValidName }

    if textUtils.TextIsEmpty(opts.BreedName) { return pet.Nil, pet.ErrNoValidBreedname }

    if opts.DateOfBirth.IsZero() { return pet.Nil, pet.ErrNoValidBirthDate }

    gender, err := pet.ParseGender(opts.Gender)
    if err != nil { return pet.Nil, err }

    p.Name = opts.Name
    p.DateOfBirth = opts.DateOfBirth
    p.Gender = gender
    p.BreedName = opts.BreedName
    p.Colors = opts.Colors
    p.Description = opts.Description
    p.Pedigree = opts.Pedigree
    p.Microchip = opts.Microchip
    p.VetId = opts.VetId
    p.Metas = opts.Metas
    p.Avatar = opts.Avatar

    return s.repo.UpdatePet(p)
}
```

Κώδικας 26 Ανανέωση κατοικίδιου

- **DeletePet:** Αυτή η μέθοδος δέχεται ως παραμέτρους το Id ενός κατοικίδιου και το Id ενός χρήστη. Διαγράφει το κατοικίδιο αν ο χρήστης είναι ο κηδεμόνας του ή αφαιρεί τον κτηνίατρο αν ο χρήστης είναι ο κτηνίατρος του. Επιστρέφει ένα μήνυμα σφάλματος σε περίπτωση προβλήματος.

```
func (s service) DeletePet(uId uuid.UUID, id uuid.UUID) error {
    _, err := s.petByOwner(uId, id, includeDel: true)

    if err != nil {
        // if it's not the owner then check if it's the pet vet
        // in this case we don't want to delete the pet but we
        // want to remove the vet
        if err == pet.ErrNotFound { return s.removeVet(id) }
        return err
    }

    return s.repo.DeletePet(id)
}
```

Κώδικας 27 Διαγραφή κατοικίδιου

- **RemoveVet:** Δέχεται σαν παράμετρο το μοναδικό αναγνωριστικό (Id) ενός κατοικίδιου και διαγράφει τον κτηνίατρο του. Επιστρέφει ένα μήνυμα σφάλματος σε περίπτωση προβλήματος.

```
func (s service) removeVet(id uuid.UUID) error {
    p, err := s.Pet(id)

    if err != nil { return err }
    p.VetId = uuid.Nil

    _, err = s.repo.UpdatePet(p)
    return err
}
```

Κώδικας 28 Διαγραφή του κτηνιάτρου του κατοικίδιου

3.1.2.1.3 Υπηρεσίες Εγγραφής

Οι υπηρεσίες των εγγραφών βασίζονται στο αποθετήριο των εγγραφών το οποίο λαμβάνουν κατά τη δημιουργία τους.

```
type Service struct {
    repo record.Repository
}

func New(repo record.Repository) (Service, error) {
    return Service{repo: repo}, nil
}
```

Κώδικας 29 Δημιουργία των υπηρεσιών των εγγραφών

- **Διεπαφή εγγραφής**

Η διεπαφή της εγγραφής ορίζει τις μεθόδους ή τις λειτουργίες που μπορεί να παρέχει για να επικοινωνήσει με άλλες δομές.

```
type Service interface {
    record(id uuid.UUID) (record.Record, error)
    PetsRecords(pIds []uuid.UUID, includeDel bool) (map[uuid.UUID]record.Record, error)
    PetRecords(pId uuid.UUID, includeDel bool) (map[uuid.UUID]record.Record, error)
    PetRecord(pId uuid.UUID, rId uuid.UUID, includeDel bool) (record.Record, error)
    CreateRecord(opts services.RecordCreateOptions) (record.Record, error)
    CreateRecords(opts services.RecordsCreateOptions) (map[uuid.UUID]record.Record, error)
    UpdateRecord(opts services.RecordUpdateOptions) (record.Record, error)
    DeleteRecord(id uuid.UUID) error
}
```

Κώδικας 30 Διεπαφή Εγγραφής

- **Υλοποίησή μεθόδων εγγραφής**

- **Record:** Αυτή η μέθοδος είναι μια ιδιωτική συνάρτηση του service. Δέχεται σαν παράμετρο το Id μίας εγγραφής και την επιστρέφει ή επιστρέφει σφάλμα σε περίπτωση προβλήματος.

```
func (s Service) Record(tId uuid.UUID) (record.Record, error) {
    return s.repo.Record(tId)
}
```

Κώδικας 31 Αναζήτηση εγγραφής

- **PetsRecords:** Η μέθοδος δέχεται σαν παράμετρο μια λίστα από Id κατοικίδιων και αν θα συμπεριλάβει διεγραμμένες εγγραφές. Επιστρέφει τις εγγραφές που ταιριάζουν στα κριτήρια ή σφάλμα.

```
func (s Service) PetsRecords(pIds []uuid.UUID, includeDel bool) (map[uuid.UUID]record.Record, error) {
    petRecords := make(map[uuid.UUID]record.Record)

    records, err := s.repo.Records(includeDel)
    if err != nil { return petRecords, err }

    for _, r := range records {
        lo.ForEach(pIds, func(pId uuid.UUID, _ int) {
            if r.PetId == pId {
                petRecords[r.Id] = r
            }
        })
    }

    return petRecords, nil
}
```

Κώδικας 32 Αναζήτηση εγγραφών κατοικίδιων

- **PetRecords:** Η μέθοδος δέχεται σαν παράμετρο το Id ενός κατοικίδιου και αν θα συμπεριλάβει διεγραμμένες εγγραφές. Επιστρέφει τις εγγραφές που ταιριάζουν στα κριτήρια ή σφάλμα.

```
func (s Service) PetRecords(pId uuid.UUID, includeDel bool) (map[uuid.UUID]record.Record, error) {
    petRecords := make(map[uuid.UUID]record.Record)

    records, err := s.repo.Records(includeDel)
    if err != nil { return petRecords, err }

    for _, r := range records {
        if r.PetId == pId {
            petRecords[r.Id] = r
        }
    }

    return petRecords, nil
}
```

Κώδικας 33 Αναζήτηση εγγραφών κατοικίδιου

- **PetRecord:** Η μέθοδος δέχεται σαν παράμετρο το μοναδικό αναγνωριστικό (Id) ενός κατοικίδιου, το μοναδικό αναγνωριστικό (Id) μιας εγγραφής και αν θα συμπεριλάβει διεγραμμένες εγγραφές. Επιστρέφει την εγγραφή του κατοικίδιου εάν βρεθεί ή σφάλμα.

```
func (s Service) PetRecord(pId uuid.UUID, rId uuid.UUID, includeDel bool) (record.Record, error) {
    petRecords, err := s.PetRecords(pId, includeDel)
    if err != nil { return record.Nil, err }
    petRecord, ok := petRecords[rId]
    if !ok { return record.Nil, record.ErrNotFound }

    return petRecord, nil
}
```

Κώδικας 34 Αναζήτηση εγγραφής κατοικίδιου

- **CreateRecord:** Η μέθοδος δέχεται τα options που παρέχουν τις πληροφορίες για τη νέα εγγραφή που θα δημιουργηθεί. Έπειτα επιβεβαιώνει ότι είναι σωστά και δημιουργεί την εγγραφή. Τέλος, επιστρέφει τη νέα εγγραφή ή ένα σφάλμα σε περίπτωση προβλήματος.

```
func (s service) CreateRecord(opts services.RecordCreateOptions) (record.Record, error) {
    r := record.Nil

    typ, err := record.ParseType(opts.RecordType)
    if err != nil { return r, record.ErrNotValidType }

    if typ == record.Weight || typ == record.Temperature {
        if textUtils.TextIsEmpty(opts.Result) { return record.Nil, record.ErrNotValidResult }
        if opts.Date.After(time.Now()) { return record.Nil, record.ErrNotValidDate }
    } else {
        if textUtils.TextIsEmpty(opts.Name) { return record.Nil, record.ErrNotValidName }
    }

    if opts.Date.IsZero() { return r, record.ErrNotValidDate }

    r.PetId = opts.PetId
    r.RecordType = typ
    r.Name = opts.Name
    r.Date = opts.Date
    r.Lot = opts.Lot
    r.Result = opts.Result
    r.Description = opts.Description
    r.Notes = opts.Notes

    if !opts.Date.After(time.Now()) {
        r.AdministeredBy = opts.AdministeredBy
        r.VerifiedBy = opts.VerifiedBy
    }

    return s.repo.CreateRecord(r)
}
```

Κώδικας 35 Δημιουργία μιας εγγραφής

- **CreateRecords:** Η μέθοδος δέχεται τα options που παρέχουν τις πληροφορίες για τις νέες εγγραφές που θα δημιουργηθούν. Έπειτα επιβεβαιώνει ότι είναι σωστά και δημιουργεί τις εγγραφές. Τέλος, επιστρέφει τις νέες εγγραφές ή ένα σφάλμα σε περίπτωση προβλήματος.

```
func (s service) CreateRecords(opts services.RecordsCreateOptions) (map[uuid.UUID]record.Record, error) {
    r := record.Nil

    typ, err := record.ParseType(opts.RecordType)
    if err != nil { return nil, record.ErrNotValidType }

    if typ == record.Weight || typ == record.Temperature {
        if textUtils.TextIsEmpty(opts.Result) { return nil, record.ErrNotValidResult }
        if opts.Date.After(time.Now()) { return nil, record.ErrNotValidDate }
    } else {
        if textUtils.TextIsEmpty(opts.Name) { return nil, record.ErrNotValidName }
    }

    if opts.Date.IsZero() { return nil, record.ErrNotValidDate }

    if opts.NextDate.IsZero() { return nil, record.ErrNotValidDate }

    recs := make([]record.Record, 2)
    r.PetId = opts.PetId
    r.RecordType = typ
    r.Name = opts.Name
    r.Date = opts.Date
    r.Lot = opts.Lot
    r.Result = opts.Result
    r.Description = opts.Description
    r.Notes = opts.Notes

    if !opts.Date.After(time.Now()) {
        r.AdministeredBy = opts.AdministeredBy
        r.VerifiedBy = opts.VerifiedBy
    }
    recs[0] = r

    nextRecord := record.Record{}
    nextRecord.PetId = opts.PetId
    nextRecord.RecordType = typ
    nextRecord.Name = opts.Name
    nextRecord.Date = opts.NextDate
    recs[1] = nextRecord

    recordsMap := make(map[uuid.UUID]record.Record)

    records, err := s.repo.CreateRecords(recs)
    if err != nil { return recordsMap, err }

    for _, rec := range records {
        recordsMap[rec.Id] = rec
    }
    return recordsMap, nil
}
```

Κώδικας 36 Δημιουργία μιας ομάδας εγγραφών

- **UpdateRecord:** Η μέθοδος δέχεται τα options που παρέχουν τις πληροφορίες για την ενημέρωση των στοιχείων της εγγραφής. Επιβεβαιώνει ότι είναι σωστά και ενημερώνει την εγγραφή. Τέλος, επιστρέφει την ενημερωμένη εγγραφή ή ένα σφάλμα σε περίπτωση προβλήματος.

```
func (s service) UpdateRecord(opts services.RecordUpdateOptions) (record.Record, error) {
    typ, err := record.ParseType(opts.RecordType)
    if err != nil { return record.Nil, record.ErrNotValidType }

    if typ == record.Weight || typ == record.Temperature {
        if textUtils.TextIsEmpty(opts.Result) { return record.Nil, record.ErrNotValidResult }
        if opts.Date.After(time.Now()) { return record.Nil, record.ErrNotValidDate }
    } else {
        if textUtils.TextIsEmpty(opts.Name) { return record.Nil, record.ErrNotValidName }
    }

    if opts.Date.IsZero() { return record.Nil, record.ErrNotValidDate }

    r, err := s.record(opts.Id)
    if err != nil { return record.Nil, err }

    r.RecordType = typ
    r.Name = opts.Name
    r.Date = opts.Date
    r.Lot = opts.Lot
    r.Result = opts.Result
    r.Description = opts.Description
    r.Notes = opts.Notes
    r.VerifiedBy = opts.VerifiedBy
    if r.AdministeredBy == uuid.Nil {
        r.AdministeredBy = opts.AdministeredBy
    }
    if opts.Date.After(time.Now()) {
        r.AdministeredBy = uuid.Nil
        r.VerifiedBy = uuid.Nil
    }

    return s.repo.UpdateRecord(r)
}
```

Κώδικας 37 Ανανέωση εγγραφής

- **DeleteRecord:** Η μέθοδος δέχεται σαν παράμετρο το Id μιας εγγραφής και τη διαγράφει. Στην περίπτωση που συμβεί σφάλμα το επιστρέφει.

```
func (s Service) DeleteRecord(id uuid.UUID) error {
    return s.repo.DeleteRecord(id)
}
```

Κώδικας 38 Διαγραφή εγγραφής

3.1.2.1.4 Υλοποίηση εφαρμογής

Αφού αναλύσαμε τις υπηρεσίες της εφαρμογής, ας δούμε τώρα πώς η εφαρμογή ενοποιεί όλα τα τμήματα της επιχειρηματικής λογικής.

Η εφαρμογή έχει την ευθύνη για την υλοποίηση όλων των σεναρίων που μπορεί να εκτελέσει ένας χρήστης της. Επιγραμματικά η εφαρμογή αφορά στον ορισμό και την υλοποίηση της παρακάτω διεπαφής:

```
type Application interface {
    CreateUser(opts services.UserCreateOptions) (user.User, string, error)
    UpdateUser(opts services.UserUpdateOptions, includeDel bool) (user.User, error)
    Users(includeDel bool) ([]user.User, error)
    UsersByType(t user.Type, includeDel bool) ([]user.User, error)
    User(id uuid.UUID) (user.User, error)
    UserByType(id uuid.UUID, t user.Type, includeDel bool) (user.User, error)
    DeleteUser(id uuid.UUID) error
    Authenticate(opts services.LoginOptions) (user.User, string, error)
    PetsByUser(uId uuid.UUID, includeDel bool) (map[uuid.UUID]pet.Pet, error)
    Pet(id uuid.UUID) (pet.Pet, error)
    PetByUser(uId uuid.UUID, id uuid.UUID, includeDel bool) (pet.Pet, error)
    DeletePet(uId uuid.UUID, id uuid.UUID) error
    CreatePet(opts services.PetCreateOptions) (pet.Pet, error)
    UpdatePet(opts services.PetUpdateOptions) (pet.Pet, error)
    CreateRecord(opts services.RecordCreateOptions) (record.Record, error)
    CreateRecords(opts services.RecordsCreateOptions) (map[uuid.UUID]record.Record, error)
    RecordsByUser(uId uuid.UUID, includeDel bool) (map[uuid.UUID]record.Record, error)
    RecordsByUserPet(uId uuid.UUID, pId uuid.UUID, includeDel bool) (map[uuid.UUID]record.Record, error)
    RecordByUserPet(uId uuid.UUID, pId uuid.UUID, tId uuid.UUID, includeDel bool) (record.Record, error)
    UpdateRecord(opts services.RecordUpdateOptions) (record.Record, error)
    DeleteRecordUserPet(uId uuid.UUID, pId uuid.UUID, id uuid.UUID) error
}
```

Κώδικας 39 Μέθοδοι που υλοποιεί η εφαρμογή

Όπως παρατηρούμε από τη διεπαφή, η εφαρμογή χρησιμοποιεί όλες τις υπηρεσίες για να πραγματοποιήσει τους απαραίτητους ελέγχους για τη συσχέτιση μεταξύ των μοντέλων. Επομένως, όταν δημιουργούμε την εφαρμογή, πρέπει να χρησιμοποιούμε dependency injection για να περάσουμε όλα τα αποθετήρια που θα χρειαστεί να δημιουργήσει τις υπηρεσίες της.

```
func New(opts Options) (Application, error) {
    //init services
    ps, err := petService.New(opts.PetRepo)
    if err != nil { return nil, err }
    us, err := userService.New(opts.UserRepo)
    if err != nil { return nil, err }
    rs, err := recordService.New(opts.RecordRepo)
    if err != nil { return nil, err }

    app := application{petService: ps, userService: us, recordService: rs}

    return &app, nil
}
```

Κώδικας 40 Δημιουργία της εφαρμογής

Μέθοδοι της εφαρμογής

Η υλοποίηση των μεθόδων που αφορούν τους χρήστες (CreateUser, UpdateUser, Users, UsersByType, User, UserByType, DeleteUser και Authenticate) είναι ακριβώς όπως περιγράφηκε στην υπηρεσία χρηστών (User Service). Έτσι, η εφαρμογή καλεί τις αντίστοιχες μεθόδους της υπηρεσίας χρηστών.

Αντίστοιχα, η υλοποίηση των μεθόδων που αφορούν τα κατοικίδια (PetsByUser, Pet, PetByUser, DeletePet και UpdatePet) είναι ακριβώς όπως στην υπηρεσία κατοικιδίων (Pet Service).

Η μόνη εξαίρεση αποτελεί η μέθοδος CreatePet. Στη μέθοδο CreatePet, η εφαρμογή ελέγχει ότι ο χρήστης που δημιουργεί το κατοικίδιο, καθώς και ο κτηνίατρος που έχει επιλεγεί, είναι έγκυροι χρήστες.

```
_, err := a.User(opts.OwnerId)
if err != nil { return pet.Nil, err }

if opts.VetId != uuid.Nil {
    _, err = a.UserByType(opts.VetId, user.Vet, includeDel: false)
    if err != nil { return pet.Nil, err }
}
```

Κώδικας 41 Έλεγχος πριν τη δημιουργία κατοικιδίου

Τέλος, για τη διαχείριση των εγγραφών (Records), η εφαρμογή χρειάζεται να πραγματοποιήσει κάποιους επιπλέον ελέγχους για να διασφαλίσει ότι τα στοιχεία του κατοικιδίου στο οποίο ανήκει είναι έγκυρα.

Για τις μεθόδους CreateRecord, CreateRecords, RecordsByUserPet και RecordByUserPet, η εφαρμογή ελέγχει ότι το κατοικίδιο ανήκει στον χρήστη που ζητάει τις εγγραφές.

```
_, err := a.PetByUser(uId, pId, includeDel: false)
if err != nil { return record.Nil, err }
```

Κώδικας 42 Έλεγχος ότι το κατοικίδιο ανήκει στον χρήστη που ζητάει τις εγγραφές

Για τις μεθόδους CreateRecord και CreateRecords, καθώς και για την UpdateRecord, γίνεται έλεγχος για την επικύρωση της εγγραφής, και ο επικυρωτής της πρέπει απαραίτητα να είναι ένας κτηνίατρος.

```
if opts.VerifiedBy != uuid.Nil {
    _, err := a.UserByType(opts.VerifiedBy, user.Vet, includeDel: true)
    if err != nil {
        switch err {
            case user.ErrNotFound: return nil, record.ErrNotValidVerifier
            default: return nil, err
        }
    }
}
```

Κώδικας 43 Έλεγχος για την επικύρωση της εγγραφής

Η μέθοδος `RecordsByUser` αρχικά βρίσκει όλα τα κατοικίδια του χρήστη και έπειτα επιστρέφει όλες τις εγγραφές τους.

```
}func (a *application) RecordsByUser(uid uuid.UUID, includeDel bool) (map[uuid.UUID]record.Record, error) {
    pets, err := a.PetsByUser(uid, includeDel)
}
    if err != nil { return nil, err }
    return a.recordService.PetsRecords(lo.Keys[uuid.UUID, pet.Pet](pets), includeDel)
}
```

Κώδικας 44 Εγγραφές όλων των κατοικιδίων του χρήστη

Τέλος, πριν τη διαγραφή μιας εγγραφής, η εφαρμογή ελέγχει ότι η εγγραφή ανήκει στο κατοικίδιο του χρήστη που ζητά τη διαγραφή της.

```
_, err := a.RecordByUserPet(uid, pId, id, includeDel: false)
if err != nil { return err }
```

Κώδικας 45 Εγγραφές όλων των κατοικιδίων του χρήστη

3.1.3 Επίπεδο Υποδομών (Infrastructure Layer)

Το επίπεδο υποδομών (Infrastructure Layer) αναφέρεται στην υποστήριξη των υπολοίπων επιπέδων της εφαρμογής. Αυτό το επίπεδο περιλαμβάνει όλα τα τεχνικά στοιχεία που απαιτούνται για τη λειτουργία της εφαρμογής, όπως η διαχείριση των βάσεων δεδομένων και η επικοινωνία με εξωτερικούς πόρους.

3.1.3.1 REST server

Στο κεφάλαιο αυτό θα αναπτυχθεί ο τρόπος δημιουργίας του API της εφαρμογής εξυπηρετητή. Αρχικά, θα περιγραφεί η διαδικασία σχεδιασμού του API, συμπεριλαμβανομένων των πόρων και των μεθόδων που θα προσφέρει. Έπειτα, θα αναλυθεί ο τρόπος υλοποίησης του API χρησιμοποιώντας το αρχιτεκτονικό στυλ REST και τις αρχές του HTTP [22]. Θα εξεταστεί η χρήση των διαφόρων μεθόδων HTTP για τη διαχείριση των πόρων και η μορφή των δεδομένων που θα μεταφέρονται.

Η σχεδίαση του API της εφαρμογής βασίστηκε στο OpenAPI, το οποίο χρησιμοποιήθηκε για τη δημιουργία των μονοπατιών, την περιγραφή των μεθόδων που χρησιμοποιούνται, καθώς επίσης και για τον καθορισμό των μοντέλων δεδομένων που θα επιστραφούν και των παραμέτρων που απαιτούνται, με βάση τις μεθόδους που υλοποιεί η εφαρμογή. Τα δεδομένα αυτά μοντελοποιήθηκαν με βάση το JSON μοντέλο.

Τα επιστρεφόμενα μοντέλα δημιουργήθηκαν με βάση τα μοντέλα του domain και επιστρέφονται σε μορφή JSON ώστε η διεπαφή χρήστη να μπορεί να τα μοντελοποιήσει αντίστοιχα.

Από τις απαιτήσεις της εφαρμογής προέκυψαν τα εξής μονοπάτια:

POST	/auth/register	▼
POST	/auth/login	▼
GET	/vets	▼
GET	/users	▼
GET	/user	▼
PATCH	/user	▼
DELETE	/user	▼
GET	/user/{id}	▼
POST	/pet	▼
GET	/pets	▼
GET	/records	▼
GET	/pet/{petId}	▼
PATCH	/pet/{petId}	▼
DELETE	/pet/{petId}	▼
POST	/pet/{petId}/record	▼
POST	/pet/{petId}/records	▼
GET	/pet/{petId}/records	▼
GET	/pet/{petId}/record/{recordId}	▼
PATCH	/pet/{petId}/record/{recordId}	▼
DELETE	/pet/{petId}/record/{recordId}	▼

Εικόνα 3 Μονοπάτια REST εφαρμογής

Αφού σχεδιάστηκε το μοντέλο του API, σειρά είχε η υλοποίησή του. Για την υλοποίησή του σε κώδικα Go, χρησιμοποιήθηκε το framework Gin Gonic.

Το framework επιτρέπει την περιγραφή μονοπατιών με τις μεθόδους που θα κληθούν σε κάθε μονοπάτι και θα αλληλεπιδράσουν με το επίπεδο της εφαρμογής.

Κάποια μονοπάτια είναι ανοιχτά για όλους τους χρήστες, όπως το μονοπάτι της εγγραφής και της σύνδεσης. Κάποια άλλα, όμως, είναι περιορισμένης πρόσβασης. Για παράδειγμα, η σελίδα κάποιου κατοικιδίου είναι προσβάσιμη μόνο από τον κηδεμόνα και τον κτηνίατρο που έχει ανατεθεί.

Με το Gin framework μπορούμε πολύ εύκολα να δημιουργήσουμε ομάδες με τα μονοπάτια που είναι περιορισμένης πρόσβασης και για κάθε ένα από αυτά τα μονοπάτια να τρέξουμε ενδιάμεσο κώδικα που ελέγχει ότι ο χρήστης που έστειλε το αίτημα είναι authenticated και έχει τα αντίστοιχα permissions.

```

api := &API{
    Engine: gin.Default(),
    app:    application,
}

```

Κώδικας 46 Δημιουργία API με Gin framework

```

api.NoRoute(middlewares.NoRouteMiddleware( urlPrefix: "/", ui, targetPath: "public"))

api.POST( relativePath: "/api/auth/register", api.register)
api.POST( relativePath: "/api/auth/login", api.login)
api.GET( relativePath: "/api/vets", api.vets)

userApi := api.Group( relativePath: "/").Use(middlewares.Auth())
userApi.GET("/api/users", api.users)
userApi.GET("/api/user", api.user)
userApi.GET("/api/user/:id", api.user)
userApi.PATCH("/api/user", api.updateUser)
userApi.DELETE("/api/user", api.deleteUser)

petApi := api.Group( relativePath: "/").Use(middlewares.Auth())
petApi.POST("/api/pet", api.createPet)
petApi.GET("/api/pets", api.pets)
petApi.GET("/api/pet/:petId", api.pet)
petApi.PATCH("/api/pet/:petId", api.updatePet)
petApi.DELETE("/api/pet/:petId", api.deletePet)

recordApi := api.Group( relativePath: "/").Use(middlewares.Auth())
recordApi.POST("/api/pet/:petId/record", api.createRecord)
recordApi.POST("/api/pet/:petId/records", api.createRecords)
recordApi.GET("/api/pet/:petId/records", api.recordsByPet)
recordApi.GET("/api/records", api.records)
recordApi.GET("/api/pet/:petId/record/:recordId", api.recordByPet)
recordApi.PATCH("/api/pet/:petId/record/:recordId", api.updateRecord)
recordApi.DELETE("/api/pet/:petId/record/:recordId", api.deleteRecord)

```

Κώδικας 47 Δημιουργία group μονοπατιών με ενδιάμεσο κώδικα

Για τα μονοπάτια περιορισμένης πρόσβασης καλείται πρώτα το authentication Middleware. Επιβεβαιώνει ότι ο χρήστης που ζητάει τα δεδομένα έχει πρόσβαση σε αυτά με βάση το authorization token που στέλνει ο χρήστης. Το authorization token είναι ένα Bearer token που ο χρήστης το λαμβάνει αφού συνδεθεί στην εφαρμογή και το στέλνει σε κάθε αίτημα προς τον διακομιστή με το header Authorization.

```

func Auth() gin.HandlerFunc {
    return func(c *gin.Context) {
        const BEARER_SCHEMA = "Bearer "
        authHeader := c.GetHeader("Authorization")
        if !strings.Contains(authHeader, BEARER_SCHEMA) {
            c.JSON(401, gin.H{"error": "request does not contain an access token"})
            c.Abort()
            return
        }
        tokenString := authHeader[len(BEARER_SCHEMA):]
        if tokenString == "" {
            c.JSON(401, gin.H{"error": "request does not contain an access token"})
            c.Abort()
            return
        }
        token, err := jwtUtils.ValidateToken(tokenString)
        if err != nil {
            c.JSON(http.StatusUnauthorized, gin.H{"error": "Unauthorized"})
            c.Abort()
            return
        }
        claims, ok := token.Claims.(jwt.MapClaims)
        if !ok || !token.Valid {
            c.JSON(http.StatusUnauthorized, gin.H{"error": "Unauthorized"})
            c.Abort()
            return
        }
        c.Set("UserId", claims["UserId"])
        c.Set("UserType", claims["UserType"])
        c.Next()
    }
}

```

Κώδικας 48 Authentication middleware

Όλες οι μέθοδοι υλοποιούνται με παρόμοιο τρόπο. Σκοπός είναι να λάβουν τα δεδομένα που στέλνει ο χρήστης είτε στο request body, είτε σαν παραμέτρους, να τα μετασχηματίσουν σε application options και να τα στείλουν στην εφαρμογή. Η εφαρμογή θα επεξεργαστεί τα αιτήματα και θα επιστρέψει είτε δεδομένα είτε σφάλμα. Αυτά τα δεδομένα θα μετασχηματιστούν σε μοντέλα JSON και θα σταλούν ως απάντηση.

```

func (api *API) register(c *gin.Context) {
    var requestBody UserCreateRequest
    err := c.ShouldBindJSON(&requestBody)
    if err != nil {
        c.JSON(http.StatusBadRequest, api.errorResponse(err))
        return
    }

    uOpts := UserCreateRequestToUserCreateOptions(requestBody)
    u, token, err := api.app.CreateUser(uOpts)
    if err != nil {
        switch err {
            case user.ErrUserDeleted,
                user.ErrNoValidType,
                user.ErrNoValidMail,
                user.ErrMailExists,
                user.ErrNoValidName,
                user.ErrNoValidSurname,
                user.ErrPasswordLength,
                user.ErrPasswordLowerCase,
                user.ErrPasswordUpperCase,
                user.ErrPasswordDigit,
                user.ErrPasswordSpecialChar: c.JSON(http.StatusBadRequest, api.errorResponse(err))
            default: c.JSON(http.StatusInternalServerError, api.errorResponse(err))
        }
        return
    }

    c.JSON(http.StatusCreated, gin.H{"user": UserToResponse(u), "token": token})
}

```

Κώδικας 49 Χειρισμός εγγραφής χρήστη

```

func UserToResponse(u user.User) *UserResponse {
    if u.Id == uuid.Nil { return nil }
    resp := UserResponse{}
    resp.Id = u.Id.String()
    resp.CreatedAt = u.CreatedAt.UnixMilli()
    resp.UpdatedAt = u.UpdatedAt.UnixMilli()
    resp.Deleted = u.Deleted
    resp.UserType = u.UserType
    resp.Email = u.Email
    resp.Name = u.Name
    resp.Surname = u.Surname
    resp.Phone = u.Phone
    resp.Address = u.Address
    resp.City = u.City
    resp.State = u.State
    resp.Country = u.Country
    resp.Zip = u.Zip
    return &resp
}

```

Κώδικας 50 Μετατροπή μοντέλου χρήστη σε μοντέλο απάντησης

```

type UserResponse struct {
    Id          string    `json:"id,omitempty"`
    CreatedAt   int64     `json:"createdAt,omitempty"`
    UpdatedAt   int64     `json:"updatedAt,omitempty"`
    Deleted     bool      `json:"deleted,omitempty"`
    UserType    user.Type `json:"userType,omitempty"`
    Email       string    `json:"email,omitempty"`
    Name        string    `json:"name,omitempty"`
    Surname     string    `json:"surname,omitempty"`
    Phone       string    `json:"phone,omitempty"`
    Address     string    `json:"address,omitempty"`
    City        string    `json:"city,omitempty"`
    State       string    `json:"state,omitempty"`
    Country     string    `json:"country,omitempty"`
    Zip         string    `json:"zip,omitempty"`
}

```

Κώδικας 51 Μοντέλο απάντησης χρήστη με χρήση json semantics

3.1.3.2 Αποθετήρια (Repositories)

Αποθετήριο (repository) [23] είναι η δομή που περιγράφει πώς αποθηκεύονται και πώς ανακτώνται τα δεδομένα από τη βάση δεδομένων. Το αποθετήριο μεταφράζει το domain model σε μοντέλο της βάσης και το αντίστροφο. Έτσι, με βάση τα μοντέλα μας, χρειαζόμαστε τρία διαφορετικά αποθετήρια: το αποθετήριο χρήστη, κατοικίδιου και εγγραφής.

Η βάση που έχει επιλεγεί για την ανάπτυξη της εφαρμογής είναι η MongoDB. Κάθε αποθετήριο βασίζεται σε μια συλλογή (collection) από εγγραφές (documents) και περιέχει μεθόδους μετασχηματισμού των μοντέλων σε μοντέλα της βάσης και αντίστροφα. Σε αυτό το κεφάλαιο θα αναπτυχθούν οι μέθοδοι που υπολογίζει κάθε αποθετήριο καθώς και οι μετασχηματισμοί που προκύπτουν. Όλα τα αποθετήρια έχουν ίδια δομή, οπότε θα αναπτυχθεί μόνο η δομή του αποθετηρίου χρήστη. Ωστόσο, ισχύει ακριβώς ο ίδιος τρόπος και για το αποθετήριο κατοικίδιου και εγγραφής.

- **Δομή αποθετηρίου**

Το αποθετήριο χρήστη υλοποιεί την εξής διεπαφή:

```

type Repository interface {
    CreateUser(user user.User) (user.User, error)
    User(id uuid.UUID) (user.User, error)
    Users(includeDel bool) ([]user.User, error)
    UpdateUser(u user.User) (user.User, error)
    DeleteUser(id uuid.UUID) error
}

```

Κώδικας 52 Διεπαφή αποθετηρίου χρήστη

```

type UserDBModel struct {
    Id          uuid.UUID `bson:"_id"`
    CreatedAt   time.Time `bson:"created_at"`
    UpdatedAt   time.Time `bson:"updated_at"`
    Deleted     bool      `bson:"deleted"`
    UserType    user.Type `bson:"user_type"`
    Email       string    `bson:"email"`
    PasswordHash string    `bson:"password_hash"`
    Name        string    `bson:"name"`
    Surname     string    `bson:"surname"`
    Phone       string    `bson:"phone,omitempty"`
    Address     string    `bson:"address,omitempty"`
    City        string    `bson:"city,omitempty"`
    State       string    `bson:"state,omitempty"`
    Country     string    `bson:"country,omitempty"`
    Zip         string    `bson:"zip,omitempty"`
}

```

Κώδικας 53 Μοντέλο χρήστη για αποθήκευση

- **CreateUser:** Δέχεται το domain μοντέλο ενός χρήστη, το μετατρέπει σε μοντέλο της βάσης και το αποθηκεύει. Επιστρέφει τον χρήστη ή σφάλμα.

```

func (r *repository) CreateUser(u user.User) (user.User, error) {
    r.mux.Lock()
    defer r.mux.Unlock()

    id, err := uuid.NewRandom()
    if err != nil { return user.Nil, err }
    u.Id = id

    now := time.Now()
    u.CreatedAt = now
    u.UpdatedAt = now

    u.Deleted = false

    dbUser, err := bson.Marshal(ConvertToUserDBModel(u))
    if err != nil { return user.Nil, err }

    _, err = r.users.InsertOne(context.Background(), dbUser)
    if err != nil { return user.Nil, err }

    return u, nil
}

```

Κώδικας 54 Αποθήκευση νέου χρήστη

- **User:** Ψάχνει στη βάση δεδομένων για τον χρήστη με το UUID. Αν τον βρει, τον μετατρέπει σε domain model και τον επιστρέφει, διαφορετικά επιστρέφει ένα σφάλμα.

```
func (r *repository) User(id uuid.UUID) (user.User, error) {
    r.mux.Lock()
    defer r.mux.Unlock()

    retrievedUser, err := r.userInternal(id)

    return ConvertToUserDomainModel(retrievedUser), err
}

func (r *repository) userInternal(id uuid.UUID) (UserDBModel, error) {
    var retrievedUser UserDBModel

    filter := bson.M{"_id": id}

    err := r.users.FindOne(context.Background(), filter).Decode(&retrievedUser)
    if err != nil { return UserDBModel{}, user.ErrNotFound }

    return retrievedUser, nil
}
```

Κώδικας 55 Ανάκτηση χρήστη από τη βάση

- **Users:** Επιστρέφει όλους τους χρήστες ή ένα σφάλμα.

```
func (r *repository) Users(includeDel bool) ([]user.User, error) {
    r.mux.Lock()
    defer r.mux.Unlock()

    var users []user.User
    var filter bson.M

    if includeDel {
        // Define an empty filter to retrieve all users
        filter = bson.M{}
    } else {
        filter = bson.M{"deleted": false}
    }

    ctx := context.Background()
    // Perform the find operation
    cursor, err := r.users.Find(ctx, filter)
    if err != nil { return users, err }
    defer cursor.Close()

    // Iterate over the cursor and decode the users
    for cursor.Next(ctx) {
        var u UserDBModel
        err = cursor.Decode(&u)
        if err != nil { return users, err }
        users = append(users, ConvertToUserDomainModel(u))
    }

    // Check for any errors during cursor iteration
    err = cursor.Err()
    if err != nil { return users, err }

    return users, nil
}
```

Κώδικας 56 Ανάκτηση χρηστών από τη βάση

- **UpdateUser:** Δέχεται το domain μοντέλο ενός χρήστη, το μετατρέπει σε μοντέλο της βάσης και το αποθηκεύει. Επιστρέφει τον χρήστη ή ένα σφάλμα.

```
func (r *repository) UpdateUser(u user.User) (user.User, error) {
    r.mux.Lock()
    defer r.mux.Unlock()

    updatedUser, err := r.updateUserInternal(ConvertToUserDBModel(u))
    if err != nil { return user.Nil, err }

    return ConvertToUserDomainModel(updatedUser), nil
}

func (r *repository) updateUserInternal(u UserDBModel) (UserDBModel, error) {
    // Define the filter to identify the document to update
    filter := bson.M{"_id": u.Id}

    u.UpdatedAt = time.Now()
    // Define the update document using the '$set' operator
    replacement, err := bson.Marshal(u)
    if err != nil { return UserDBModel{}, err }

    // Perform the update operation
    _, err = r.users.ReplaceOne(context.Background(), filter, replacement)
    if err != nil { return UserDBModel{}, err }

    return u, nil
}
```

Κώδικας 57 Ανανέωση στοιχείων χρήστη στη βάση

- **DeleteUser:** Ψάχνει στη βάση για τον χρήστη με το UUID. Εάν τον βρει, τότε τον διαγράφει (soft delete), αλλιώς επιστρέφει ένα σφάλμα.

```
func (r *repository) DeleteUser(id uuid.UUID) error {
    r.mux.Lock()
    defer r.mux.Unlock()

    retrievedUser, err := r.userInternal(id)
    if err != nil { return err }

    retrievedUser.Deleted = true

    _, err = r.updateUserInternal(retrievedUser)

    return err
}
```

Κώδικας 58 Διαγραφή χρήστη από τη βάση

3.1.3.3 Main Module

Ένα πρόγραμμα Go πρέπει να περιέχει τουλάχιστον ένα αρχείο με τη δήλωση της συνάρτησης `main()`. Η συνάρτηση `main()` είναι το σημείο εκκίνησης του προγράμματος και από εκεί αρχίζει η εκτέλεση.

Η `main` συνάρτηση διαβάζει τις αρχικές παραμέτρους που χρειάζονται για τη σύνδεση στη βάση. Εφόσον συνδεθεί, δημιουργεί τα αποθετήρια περνώντας τους το `collection` της MongoDB που αντιστοιχεί σε κάθε ένα.

```
uri := os.Getenv( key: "DB_URL")
if uri == "" {
    Log.Fatal( v...: "You must set your 'DB_URL' environment variable. See\n\t https://www.mongodb.com/docs/drivers/go/current/usage-examples/
}
//init db
ctx, cancel := context.WithTimeout(context.Background(), 10*time.Second)
defer cancel()

// Use the SetServerAPIOptions() method to set the Stable API version to 1
serverAPI := options.ServerAPI(options.ServerAPIVersion1)
client, err := mongo.Connect(ctx, options.Client().ApplyURI(uri).SetServerAPIOptions(serverAPI))
if err != nil { panic(err) }
defer func() {
    if err := client.Disconnect(context.TODO()); err != nil { panic(err) }
}()

// Send a ping to confirm a successful connection
var result bson.M
if err := client.Database( name: "admin").RunCommand(context.TODO(), bson.D{{ Key: "ping", Value: 1}}).Decode(&result); err != nil {
    panic(err)
}
fmt.Println( a...: "Pinged your deployment. You successfully connected to MongoDB!")

db := client.Database(os.Getenv( key: "DB_NAME"))
```

Κώδικας 59 Σύνδεση στη βάση

Έπειτα, δημιουργεί την εφαρμογή κάνοντας injection τα αποθετήρια που δημιούργησε. Αφού δημιουργηθεί η εφαρμογή, την περνάει με dependency injection στον REST client ώστε να μπορεί να καλέσει τις αντίστοιχες μεθόδους.

```
db := client.Database(os.Getenv( key: "DB_NAME"))

//init repos
petsCollection := db.Collection( name: "pets")
petRepo := petrepo.New(petsCollection)

usersCollection := db.Collection( name: "users")
userRepo := userrepo.New(usersCollection)

recordsCollection := db.Collection( name: "records")
recordRepo := recordrepo.New(recordsCollection)

//pass services to application
opts := application.Options{PetRepo: petRepo, UserRepo: userRepo, RecordRepo: recordRepo}
app, err := application.New(opts)
if err != nil { panic(err) }

restServer := api.New(app, ui)
```

Κώδικας 60 Δημιουργία εφαρμογής και REST client

Τέλος ξεκινάει να ακούει για αιτήματα στη διεύθυνση και πόρτα που έχει οριστεί από το configuration και να κάνει server το user interface.

```
go func() { // Start listening and serving requests
    err = restServer.Run(config.Host + ":" + config.Port)

    if err != nil { panic(err) }
}()
//ctrl + c to stop server
waitForInterrupt := make(chan os.Signal, 1)
signal.Notify(waitForInterrupt, os.Interrupt, os.Kill)

<-waitForInterrupt
```

Κώδικας 61 Εκκίνηση του server

3.2 Υλοποίηση εφαρμογής πελάτη

Για τη δημιουργία της εφαρμογής πελάτη (client side application) αναπτύχθηκε χρησιμοποιώντας το Next.js ένα framework βασισμένο στην React, τα αρχεία των σελίδων είναι σε μορφή tsx (typescript XML) με τη χρήση της γλώσσας typescript για τη λειτουργικότητα των σελίδων και τη χρήση HTML και CSS με το tailwind framework.

Τα εικονίδια που χρησιμοποιήθηκαν στις σελίδες είναι από τη βιβλιοθήκη ελεύθερου περιεχομένου “heroicons”.

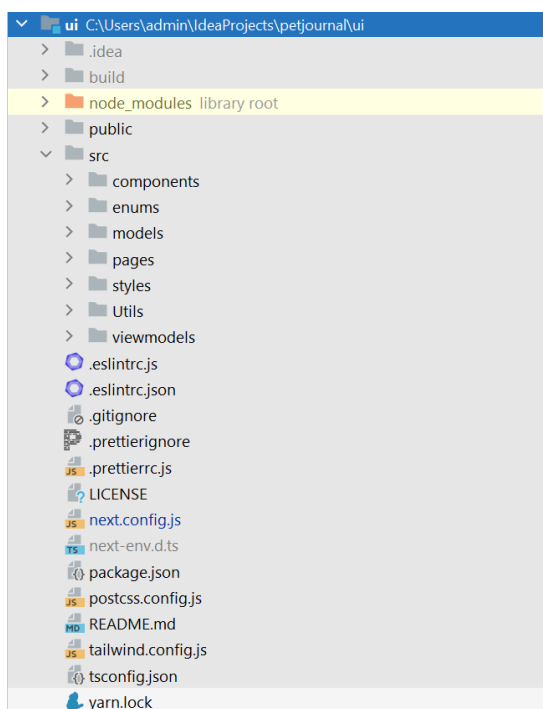
Τα διαγράμματα δημιουργήθηκαν με τη βιβλιοθήκη “recharts” [24].

Το ημερολόγιο που χρησιμοποιήθηκε τόσο στη σελίδα κατοικίδιου όσο και στη σελίδα συνολικής ατζέντας είναι από τη βιβλιοθήκη “react-big-calendar” [25].

Τέλος κατά τη δημιουργία κατοικίδιου για την αυτόματη εισαγωγή χρωμάτων από τη φωτογραφία χρησιμοποιήθηκε η βιβλιοθήκη “react-color-extractor” [26] και για την αφαίρεση χρώματος από τη λίστα η αλλαγή του χρώματος του κουμπιού ανάλογα με το χρώμα φόντου έγινε με τη χρήση της βιβλιοθήκης “tinycolor2” [27].

Το project είναι δομημένο με βάση το πρωτότυπο που έχει το Next.js [28]. Ο κώδικας είναι χωρισμένος σε:

- **components:** Δομικά στοιχεία που χρησιμοποιούνται σε σελίδες. Κάθε στοιχείο είναι ολοκληρωμένο και μπορεί να αλλάζει ανάλογα με τα δεδομένα που θα του περάσει κάποια σελίδα ή άλλο component.
- **enums:** Είναι ένας τύπος δεδομένων που χρησιμοποιείται σε πολλές γλώσσες προγραμματισμού για να ορίσει μια συλλογή από σταθερές τιμές.
- **models:** Όπως και στο domain της εφαρμογής παρασκηνίου έτσι και στην εφαρμογή πελάτη έχουμε τα μοντέλα της εφαρμογής.
- **pages:** Οι βασικές σελίδες της εφαρμογής.
- **styles:** Κεντρικά styles που εφαρμόζονται σε όλες τις σελίδες.
- **utils:** Εργαλεία με μεθόδους που επαναχρησιμοποιούνται.
- **viewmodels:** Το ViewModel περιέχει την κατάσταση και τη λογική της διεπαφής χρήστη και παρέχει τις ιδιότητες και τις εντολές που χρησιμοποιεί το View για να εμφανίσει τα δεδομένα και να χειριστεί τις ενέργειες του χρήστη.

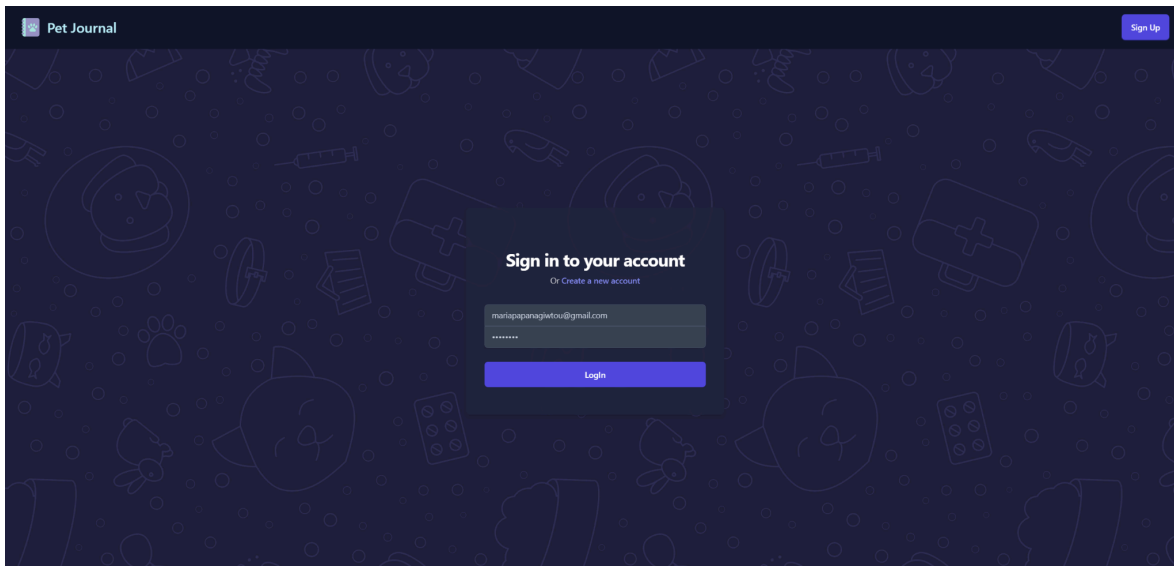


Κώδικας 62 Δομή εφαρμογής πελάτη

3.2.1 Οθόνη σύνδεσης χρήστη

Με την είσοδο του χρήστη στην εφαρμογή πλοηγείται στην αρχική σελίδα της εφαρμογής. Εάν δεν είναι συνδεδεμένος τότε αυτή είναι η σελίδα σύνδεσης.

Ο χρήστης καλείται να συμπληρώσει τη φόρμα με σωστά στοιχεία ώστε να συνδεθεί στην εφαρμογή.



Εικόνα 4 Σελίδα σύνδεσης.

Από την μεριά του κώδικα έχουμε δύο στοιχεία: ένα `viewmodel` που περιγράφει τα πεδία προς συμπλήρωση και μια μεταβλητή για μήνυμα λάθους μετά την επικοινωνία με τον server.

```
interface LoginState {
  vm: LoginViewModel
  serverError?: string
}

export default class LoginViewModel {
  public email: string = ""
  public password: string = ""
}
```

Κώδικας 63 Περιγραφή δομής `state` και `viewmodel` της σελίδας εγγραφής

Όταν ο χρήστης εισάγει τα στοιχεία του στα αντίστοιχα πεδία θα ενημερωθεί το `viewmodel` αντίστοιχα και το `view` θα λάβει τις ανανεωμένες τιμές των πεδίων.

```
private onEmailChange = (value: string) : void => {
  const vm : LoginViewModel = this.state.vm
  vm.email = value
  this.setState( state: {vm})
}

private onPasswordChange = (value: string) : void => {
  const vm : LoginViewModel = this.state.vm
  vm.password = value
  this.setState( state: {vm})
}
```

Κώδικας 64 Ενημέρωση τιμών των πεδίων στο `viewmodel` και στο `view`.

Μόλις ολοκληρωθεί η συμπλήρωση των στοιχείων και ο χρήστης πατήσει σύνδεση καλείται η μέθοδος `onSubmit`. Ασύγχρονα προωθείται ένα request με τα δεδομένα σύνδεσης στον server και αναμένεται η απάντηση. Αν η σύνδεση είναι επιτυχής καλείται η μέθοδος `onLoginSuccess` προωθώντας το jwt token του χρήστη. Εάν απέτυχε καλείται η `onFail` με το μήνυμα λάθος που έλαβε από τον server και το εμφανίζει στον χρήστη και διαγράφει οποιοδήποτε cookie σχετικό με token μπορεί να προϋπήρχε.

```
private onSubmit = async () : Promise<void> => {
  try {
    const resp : Response = await loginHandler(this.state.vm)
    const data: Auth = await resp.json()
    if (resp.ok) {
      this.onLoginSuccess(data.token!)
    } else {
      this.onFail((data as ErrorDto).error)
    }
  } catch (e: any) {
    this.onFail(e.message)
  }
}
```

Κώδικας 65 Μέθοδος υποβολής φόρμας

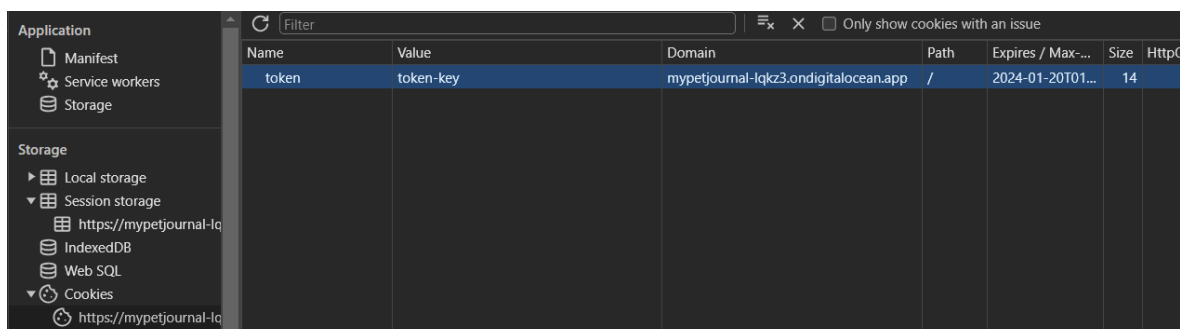
```
export function loginHandler(vm: LoginViewModel): Promise<Response> {
  return fetch( input: `${process.env.apiUrl}/auth/login`, init: {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify(vm),
  })
}
```

Κώδικας 66 Μέθοδος δημιουργίας αιτήματος σύνδεσης προς τον server

Εφόσον όλα πάνε καλά τότε η σύνδεση ήταν επιτυχής. Το jwt token του χρήστη σώζεται ως cookie στον περιηγητή και πλοηγείται στην αρχική σελίδα της εφαρμογής [29].

```
private onLoginSuccess = (token: string) : void => {
  const decoded: {exp: number} = jwtDecode(token)
  this.cookies.set( name: "token", token, options: {expires: decoded?.exp > Date.now() ? new Date(decoded?.exp) : undefined})
  this.props.router.replace( url: "/" )
}
```

Κώδικας 67 Μέθοδος επιτυχούς σύνδεσης.



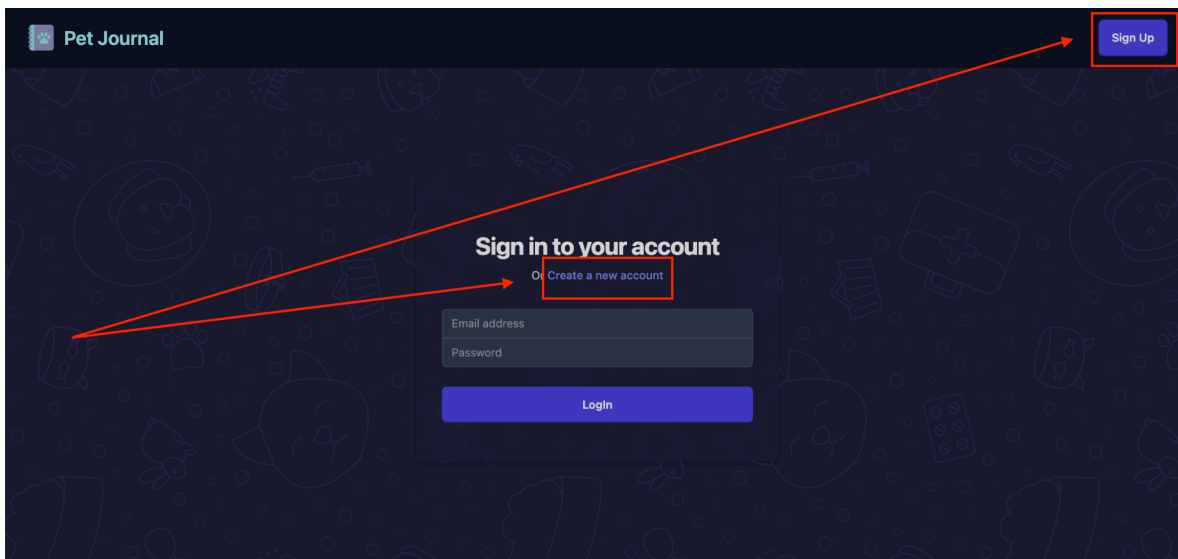
Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly
token	token-key	mypetjournal-lqkz3.ondigitalocean.app	/	2024-01-20T01:00:00.000Z	14	

Εικόνα 5 Αποθήκευση token ως browser cookie.

3.2.2 Οθόνη εγγραφής χρήστη

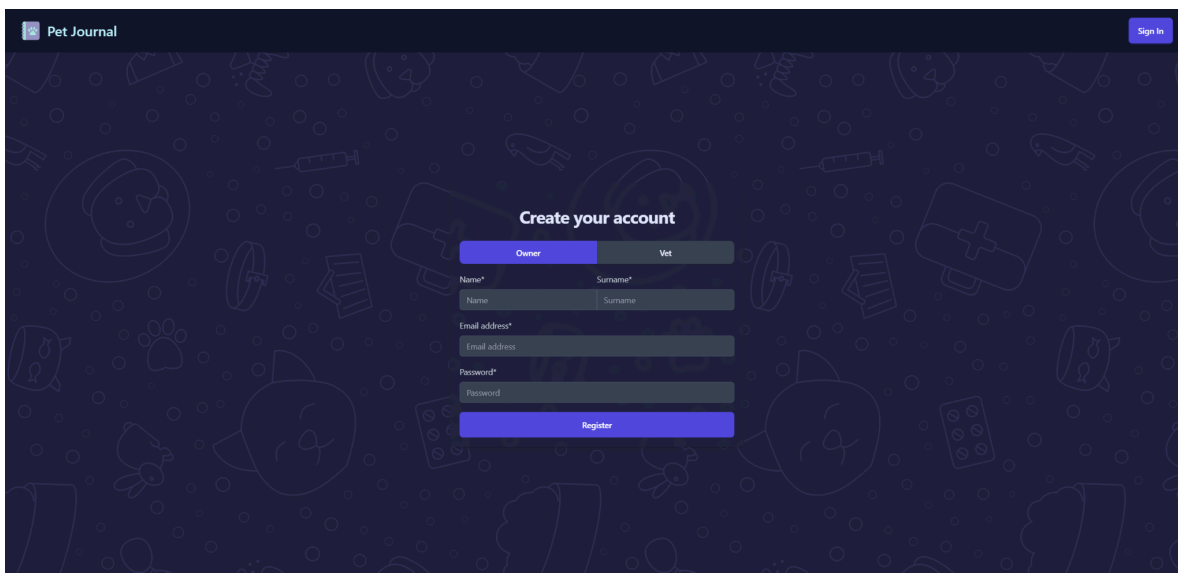
Εάν ο χρήστης δεν διαθέτει λογαριασμό μπορεί να διαλέξει να εγγραφεί στην εφαρμογή από δύο σημεία στην αρχική οθόνη σύνδεσης.

Ο χρήστης μπορεί να δει ένα κουμπί που γράφει εγγραφή στο πάνω δεξιό μέρος της οθόνης στην μπάρα πλοήγησης. Εναλλακτικά υπάρχει ένα κουμπί που προτρέπει τον χρήστη να εγγραφεί αν δεν έχει λογαριασμό στο πάνω μέρος της φόρμας σύνδεσης.



Εικόνα 6 Κουμπί ανακατεύθυνσης στη σελίδα εγγραφής.

Μόλις ο χρήστης επιλέξει να εγγραφεί στην εφαρμογή πλοηγείται στη σελίδα εγγραφής. Εκεί καλείται να συμπληρώσει τα στοιχεία του στην φόρμα εγγραφής.



Εικόνα 7 Σελίδα εγγραφής χρήστη

Όπως κατά τη σύνδεση έτσι και κατά την εγγραφή ενός χρήστη ενημερώνουμε τα στοιχεία με βάση ενός viewmodel που περιέχει όλα τα πεδία όπως φαίνονται στο γραφικό περιβάλλον.

Η φόρμα διαθέτει ελέγχους για την ορθότητα των στοιχείων που έχουν εισαχθεί. Εάν υπάρχει κάποιο λάθος ή κάποιο κενό ο χρήστης θα δει το αντίστοιχο μήνυμα διόρθωσης. Ο χρήστης δεν μπορεί να προχωρήσει σε εγγραφή μέχρι όλα τα στοιχεία να είναι σωστά. Οι έλεγχοι που γίνονται στη φόρμα είναι ίδιοι με αυτούς που γίνονται στην εφαρμογή παρασκηνίου για την εγγραφή ενός χρήστη.

Εικόνα 8 Έλεγχος φόρμας εγγραφής

Αφού συμπληρωθούν τα πεδία της φόρμας τότε ο χρήστης πατάει Εγγραφή και την υποβάλλει. Αρχικά ελέγχονται οι τιμές της για την ορθότητα τους. Έπειτα ασύγχρονα στέλνεται ένα request προς τον server για την εγγραφή του χρήστη. Εφόσον είναι επιτυχές τότε ο χρήστης συνδέεται και μεταφέρεται στην αρχική σελίδα. Αλλιώς εμφανίζεται μήνυμα λάθους όπως έρχεται από τον server.

```
private onSubmit = async () : Promise<void> => {
  this.state.vm.validate()
  if (!this.state.vm.isValid) {
    return
  }
  const response : Response = await signUpHandler(this.state.vm)

  const data: Auth = await response.json()

  if (response.ok) {
    // Login successful
    this.onLoginSuccess(data.token!)
  } else {
    // Login failed
    this.onFail((data as ErrorDto).error)
  }
}
```

Κώδικας 68 Υποβολή φόρμας εγγραφής

```

validate() :void {
  this.nameError = TextUtils.isEmpty(this._name) ? "" : "Name is required"
  this.passwordError = TextUtils.PasswordError(this._password)
  this.surnameError = TextUtils.isEmpty(this._surname) ? "" : "Surname is required"
  this.emailError = TextUtils.isValidEmail(this._email) ? "" : "Please provide a valid email"

  this._valid =
    TextUtils.isEmpty(this.nameError) &&
    TextUtils.isEmpty(this.surnameError) &&
    TextUtils.isEmpty(this.emailError) &&
    TextUtils.isEmpty(this.passwordError)
}

```

Κώδικας 69 Μέθοδος υποβολής φόρμας

```

export function signUpHandler(vm: RegistrationViewModel): Promise<Response> {
  return fetch( input: `${process.env.apiUrl}/auth/register`, init: {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify(vm.fields),
  })
}

```

Κώδικας 70 Μέθοδος δημιουργίας αιτήματος εγγραφής προς τον server

3.2.3 Οθόνη κατοικίδιων (Κύρια οθόνη)

Η οθόνη κατοικίδιων είναι η κύρια σελίδα της εφαρμογής. Για να πλοηγηθεί σε αυτή ένας χρήστης πρέπει πρώτα να έχει συνδεθεί.

Τη λειτουργία αυτή ελέγχει το component protected page [30]. Ψάχνει στα cookies το token του χρήστη και αν υπάρχει τότε καλεί την init μέθοδο της σελίδας. Αλλιώς αποσυνδέει τον χρήστη αφαιρώντας το token Cookie.

```

componentDidMount() :void {
  const t = this.cookies.get( name: "token")

  if (t !== undefined) {
    this.setState( state: {token: t})
    this.initInterval = setInterval( callback: () :void => {
      if (this.props.router.isReady) {
        this.props.init && this.props.init(t)
        clearInterval(this.initInterval)
      }
    }, ms: 1)
  } else {
    this.logout( callback: () => this.props.router.replace( url: "/auth/login"))
  }
}

```

Κώδικας 71 Μέθοδος εκκίνησης component protected page

Με την είσοδο του αυθεντικοποιημένου χρήστη στη σελίδα στέλνεται ένα αίτημα στον server να επιστρέψει όλα τα κατοικίδια που είναι συνδεδεμένα με τον χρήστη.

```

private initPage = (token?: string) : void => {
  this.setState( state: {token})

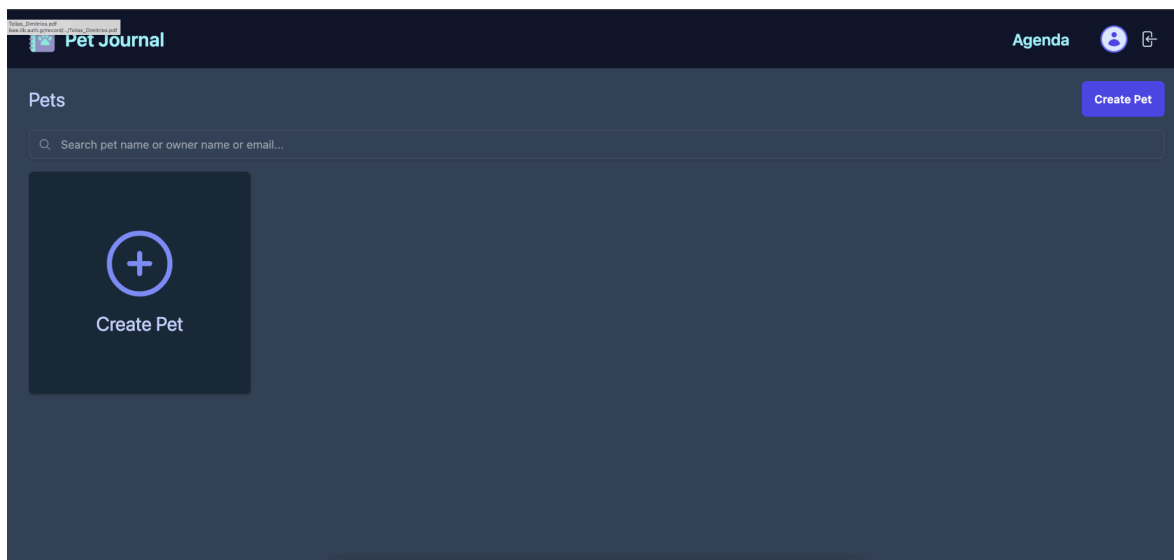
  this.getPets(token).catch((e) : void => {
    this.setState( state: {serverError: e.message})
    console.error("Could not get pets", e)
  })
}

private getPets = async (token?: string) : Promise<void> => {
  const resp : Response = await petsHandler(token)
  const data: Pet[] = await resp.json()
  if (resp.ok) {
    this.setState( state: {pets: data, loading: false})
  } else if (resp.status === 401) {
    this.logout( callback: () => this.props.router.replace( url: "/auth/login"))
  } else {
    throw Error((data as ErrorDto).error)
  }
}

```

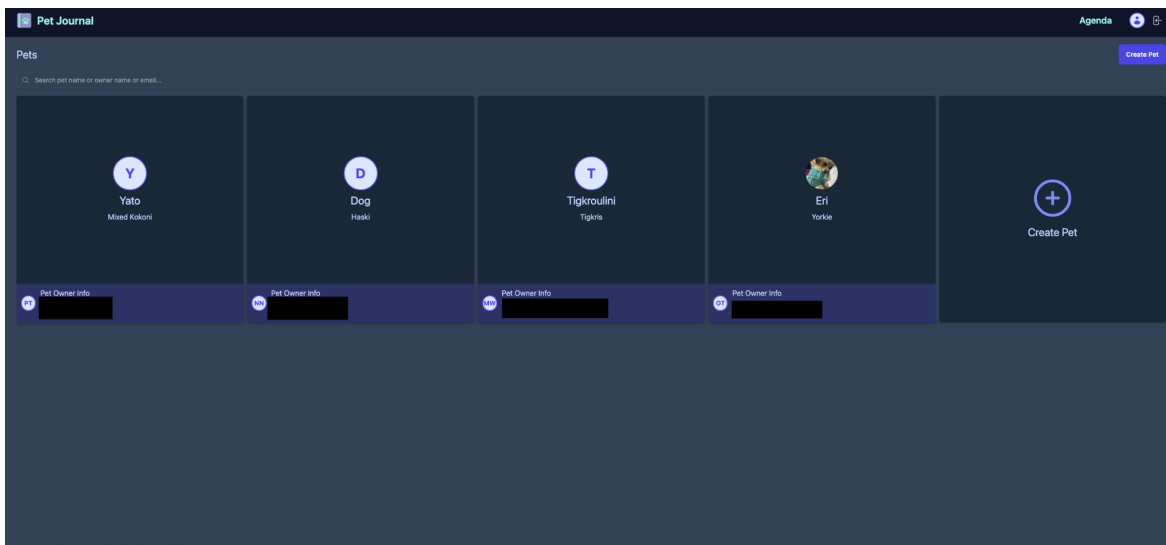
Κώδικας 72 Αρχικοποίηση σελίδας κατοικίδιων.

Ένας νέος χρήστης στην αρχική του σελίδα θα βλέπει ένα κενό tile για τη δημιουργία ενός νέου προφίλ κατοικίδιου.



Εικόνα 9 Αρχική σελίδα κατοικίδιων

Αφού δημιουργηθούν νέα προφίλ ή ένας κηδεμόνας κατοικίδιου προωθήσει το προφίλ του κατοικίδιου του στον κτηνίατρο του τότε τα κατοικίδια εμφανίζονται σε μορφής κάρτας και στο κάτω μέρος της περιέχουν τις βασικές πληροφορίες του κηδεμόνα του.



Εικόνα 10 Αρχική σελίδα κατοικίδιων χρήστη που έχει δημιουργήσει λογαριασμούς κατοικίδιων

Ο χρήστης μπορεί να αναζητήσει κάποιο κατοικίδιο με βάση τις πληροφορίες του ή του κηδεμόνα του.

```
private get filteredPets() : Pet[] {
    if (TextUtils.isEmpty(this.state.filter)) {
        return this.state.pets
    }

    return this.state.pets.filter((pet : Pet ) => {
        return (
            pet.name?.toUpperCase().includes(this.state.filter) ||
            pet.owner?.name?.toUpperCase().includes(this.state.filter) ||
            pet.owner?.surname?.toUpperCase().includes(this.state.filter) ||
            pet.owner?.email?.toUpperCase().includes(this.state.filter) ||
            pet.owner?.phone?.toUpperCase().includes(this.state.filter)
        )
    })
}
```

Κώδικας 73 Φίλτρα αναζήτησης κατοικίδιων

Ο χρήστης μπορεί να διαλέξει το κατοικίδιο που επιθυμεί και θα μεταφερθεί στη σελίδα του κατοικίδιου. Καθώς και από το πάνω μέρος της οθόνης στην μπάρα πλοήγησης μπορεί να πλοηγηθεί στη σελίδα της ατζέντας, στη σελίδα του λογαριασμού του ή να αποσυνδεθεί από την εφαρμογή.

3.2.4 Οθόνη δημιουργίας/επεξεργασίας προφίλ κατοικίδιου

Ο χρήστης θα πλοηγηθεί σε αυτή τη σελίδα όταν επιλέξει να δημιουργήσει ένα νέο προφίλ κατοικίδιου.

Εικόνα 11 Σελίδα δημιουργίας κατοικίδιου

Όπως και στη σελίδα δημιουργίας λογαριασμού χρήστη, έτσι και στη δημιουργία προφίλ κατοικίδιου ο χρήστης καλείται να συμπληρώσει τις απαραίτητες πληροφορίες για τη δημιουργία του προφίλ κατοικίδιου.

Εάν ο χρήστης διαλέξει να προσθέσει εικόνα του κατοικίδιου του τότε τα χρώματα ανανεώνονται αυτόματα με βάση τα κυριότερα χρώματα της παλέτας της εικόνας. Μπορεί να αφαιρέσει και να προσθέσει όσα χρώματα επιθυμεί.

Create a Pet profile

Name *

Date of Birth *

Gender *

Breed Name *

Colors

Description

Microchip

Pedigree

Min Weight

Share with vet

Εικόνα 12 Προσθήκη χρωμάτων κατοικίδιου

```
private getAvatarColors = (colors: string[]) : void => {
  const vm : PetCreationViewModel = this.state.vm

  vm.colors = vm.colors.filter((c : string) => !this.state.avatarColors.includes(c))
  vm.colors.push(...colors)
  this.setState( state: {vm, avatarColors: colors})
}
```

Κώδικας 74 Προσθήκη χρωμάτων κατοικίδιου στο viewModel

Όταν ο χρήστης συμπληρώσει τα απαραίτητα πεδία και πατήσει “δημιουργία”, αρχικά επιβεβαιώνονται τα στοιχεία της φόρμας. Έπειτα αποστέλλεται αίτημα στον server για τη δημιουργία του προφίλ κατοικίδιου.

Εάν όλα είναι σωστά τότε η εφαρμογή παρασκηνίου έχει δημιουργήσει το προφίλ του κατοικίδιου και τον επιστρέφει και ο χρήστης πλοηγείται αυτόματα στη σελίδα του κατοικίδιου. Σε περίπτωση που συμβεί κάποιο σφάλμα τότε εμφανίζεται το αντίστοιχο μήνυμα στην οθόνη.

```
private onSubmit = async () : Promise<void> => {
  this.state.vm.validate()
  if (!this.state.vm.isValid || !this.state.token) {
    this.forceUpdate()
    return
  }

  const response : Response = await petCreationHandler(this.state.vm, this.state.token)

  const data: Pet = await response.json()

  if (response.ok) {
    this.navigateToPetPage(data.id)
  } else if (response.status === 401) {
    this.logout( callback: () => this.props.router.replace( url: "/auth/login"))
  } else {
    this.setState( state: {serverError: (data as ErrorDto).error ?? ""})
  }
}
```

Κώδικας 75 Υποβολή φόρμας εγγραφής κατοικίδιου

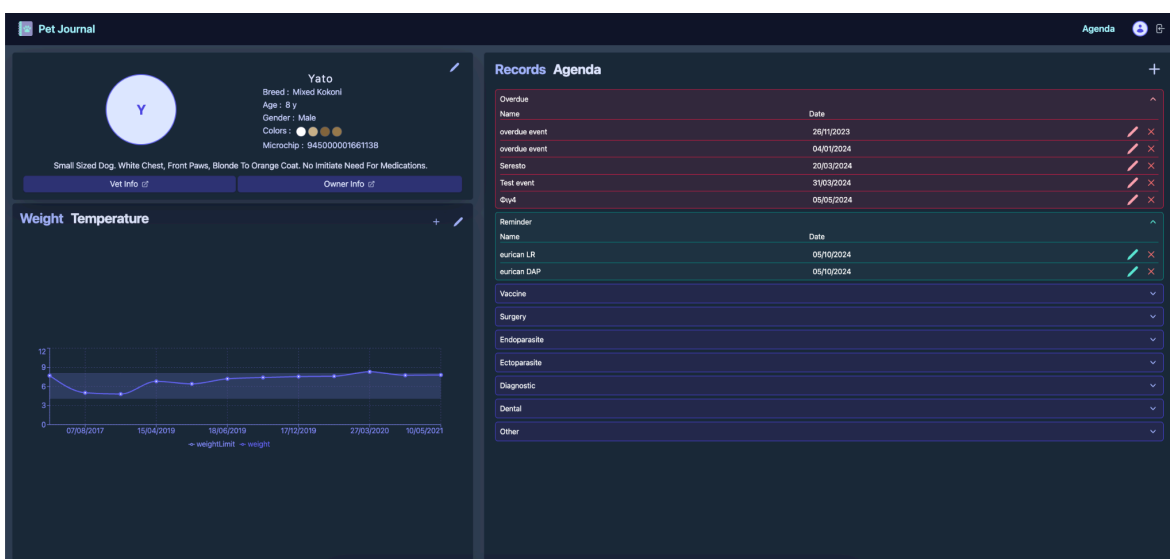
```
validate() : void {
  this.checkName()
  this.checkDateOfBirth()
  this.checkGender()
  this.checkBreedName()
  this._valid =
    TextUtils.isEmpty(this.nameError) &&
    TextUtils.isEmpty(this.dateOfBirthError) &&
    TextUtils.isEmpty(this.genderError) &&
    TextUtils.isEmpty(this.breedNameError)
}
```

Κώδικας 76 Επιβεβαίωση φόρμας δημιουργίας κατοικίδιου

3.2.5 Οθόνη κατοικίδιου

Η οθόνη κατοικίδιου είναι και η πλέον σημαντική σελίδα της εφαρμογής. Σε αυτή βρίσκουμε όλες τις πληροφορίες και τις εγγραφές που έχουμε για ένα κατοικίδιο. Χωρίζεται σε 4 βασικά μέρη. Στο πάνω αριστερό μέρος βρίσκονται οι βασικές πληροφορίες για το κατοικίδιο καθώς και οι πληροφορίες για τον κηδεμόνα και τον κτηνίατρο του. Κάτω αριστερά στην οθόνη έχουμε τα διαγράμματα των εγγραφών του βάρους και της θερμοκρασίας που απεικονίζονται με τη βοήθεια διαγραμμάτων. Στην περίπτωση των εγγραφών βάρους ο χρήστης μπορεί να δει εάν το βάρος του κατοικίδιου είναι στα όρια που έχει βάλει στις πληροφορίες προφίλ.

Στη δεξιά μεριά της σελίδας απεικονίζονται οι εγγραφές του κατοικίδιου ομαδοποιημένες με βάση τον τύπο τους. Εκεί ο χρήστης μπορεί να προσθέσει, να διαγράψει, να αντιγράψει καθώς και να επεξεργαστεί τις εγγραφές του κατοικίδιου. Εφόσον ο χρήστης το επιθυμεί τότε μπορεί να επιλέξει να βλέπει την ατζέντα του κατοικίδιου αντί να βλέπει αναλυτικά τις εγγραφές.



Εικόνα 13 Σελίδα κατοικίδιου

Κατά τη διαδικασία εκκίνησης της σελίδας στέλνουμε αίτημα στο server ζητώντας τα στοιχεία του κατοικίδιου και τις εγγραφές που το αφορούν.

```
private initPage = (token?: string) : void => {
  const petId : string = this.props.router.query.id as string
  this.getPet(petId, token).catch((e) => this.setState( state: {serverError: e.message}))
  this.getRecords(petId, token).catch((e) => this.setState( state: {serverError: e.message}))
}
```

Κώδικας 77 Εκκίνηση σελίδας κατοικίδιου

```

private getPet = async (id?: string, token?: string) : Promise<void> => {
  if (!id) {
    return
  }

  const resp : Response = await petHandler(id, token)
  const response: Pet = await resp.json()
  if (resp.ok) {
    this.setState( state: {pet: response, token})
  } else if (resp.status === 401) {
    this.logout( callback: () => this.props.router.replace( url: "/auth/login"))
  } else {
    throw Error((response as ErrorDto).error)
  }
}

private getRecords = async (id?: string, token?: string) : Promise<void> => {
  if (!id) {
    return
  }

  const resp : Response = await petRecordsHandler(id, token)
  const response: Record[] = await resp.json()
  if (resp.ok) {
    this.setState( state: {records: response, loading: false}, callback: () : void => {
      this.updateLineDatasets()
      this.updateEvents()
    })
  } else if (resp.status === 401) {
    this.logout( callback: () => this.props.router.replace( url: "/auth/login"))
  } else {
    throw Error((response as ErrorDto).error)
  }
}

```

Κώδικας 78 Μέθοδοι λήψης πληροφοριών κατοικιδίου

```

export function petHandler(id: string, token?: string): Promise<Response> {
  return fetch( input: `${process.env.apiUrl}/pet/${id}`, init: {
    method: "GET",
    headers: {
      Authorization: `Bearer ${token}`,
    },
  })
}

```

Κώδικας 79 Αίτημα λήψης πληροφοριών κατοικιδίου

```

export function petRecordsHandler(petId: string, token?: string): Promise<Response> {
  return fetch( input: `${process.env.apiUrl}/pet/${petId}/records`, init: {
    method: "GET",
    headers: {
      "Content-Type": "application/json",
      Authorization: `Bearer ${token}`,
    },
  },
)
}
}

```

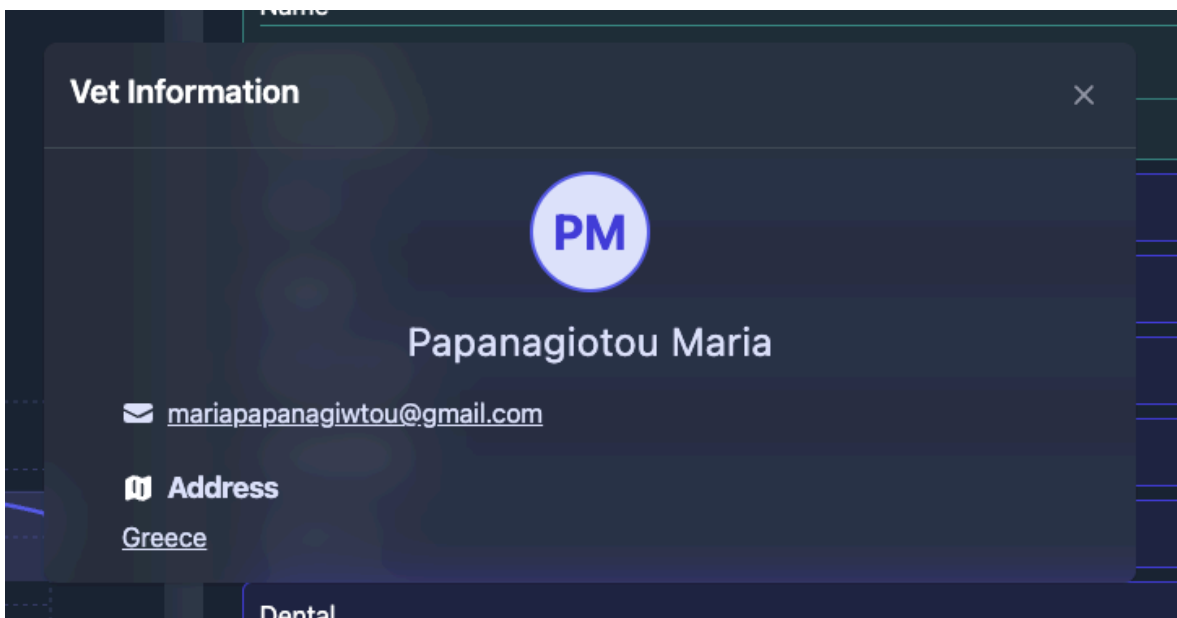
Κώδικας 80 Αίτημα λήψης εγγραφών κατοικίδιου

Στο πάνω αριστερό μέρος της σελίδας όπου περιέχει τις πληροφορίες του κατοικίδιου ο χρήστης μπορεί να διαλέξει να επεξεργαστεί τις πληροφορίες του πατώντας το κουμπί σε σχήμα μολυβιού οπότε ανακατευθύνεται στη σελίδα επεξεργασίας των στοιχείων του κατοικίδιου. Η σελίδα αυτή είναι όμοια με εκείνη της δημιουργίας του κατοικίδιου, με τη διαφορά ότι τα στοιχεία είναι προσυμπληρωμένα.

Εικόνα 14 Οθόνη επεξεργασίας κατοικίδιου

Εφόσον ο χρήστης το επιθυμεί μπορεί να ανανεώσει τα στοιχεία του κατοικίδιου του, να αλλάξει κτηνίατρο ή να διαγράψει το προφίλ πατώντας τον κάδο και επιβεβαιώνοντας την επιλογή του. Έπειτα επιστρέφει στη σελίδα του κατοικίδιου και μπορεί να δει τα αναμενόμενα στοιχεία.

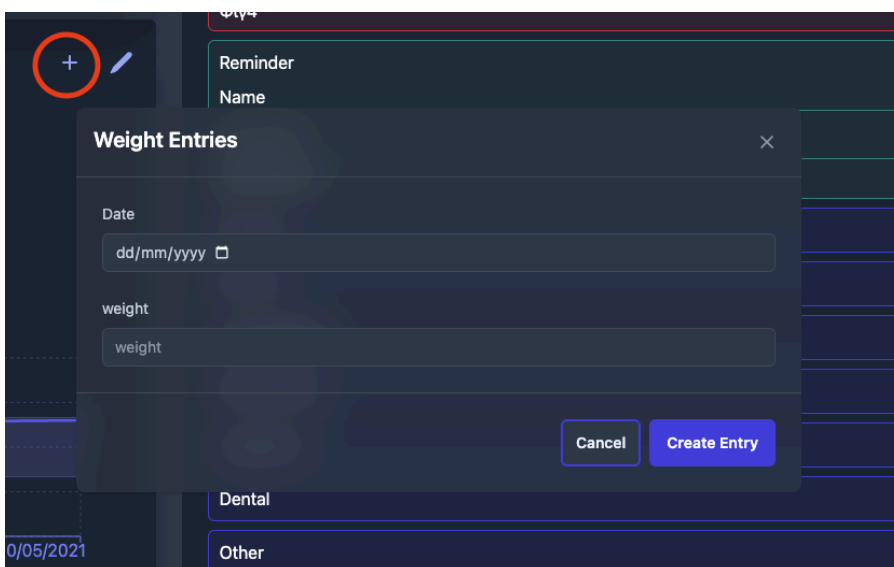
Στο κάτω μέρος των στοιχείων βρίσκονται δύο κουμπιά και όταν επιλεχθούν εμφανίζουν ένα dialog με τα στοιχεία του κτηνιάτρου ή του κηδεμόνα.



Εικόνα 15 Dialog στοιχείων κτηνιάτρου

Στην επιμέρους λειτουργικότητα της σελίδας αυτής ο χρήστης μπορεί να προσθέσει εγγραφές βάρους και θερμοκρασίας από το κάτω αριστερό μέρος πατώντας το κουμπί με το σύμβολο “+”.

Τότε θα εμφανιστεί ένα dialog για τη δημιουργία μιας εγγραφής βάρους. Αφού εισάγει τα απαραίτητα πεδία τότε μπορεί να προσθέσει την εγγραφή και θα εμφανιστεί το dialog επεξεργασίας των εγγράφων βάρους. Εναλλακτικά μπορεί πατώντας το κουμπί με το σύμβολο του μολυβιού να δει το dialog των εγγράφων και να προσθέσει μία νέα στην λίστα από το κουμπί “προσθήκη εγγραφής”.



Εικόνα 16 Dialog δημιουργίας εγγραφής βάρους.

```

private setCreateWeightEntry = () : void => {
  this.weightEntriesDialogRef?.setData(this.state.records)
  this.weightEntriesDialogRef?.setRecordType(this.state.selectedLineModel)
  this.weightEntriesDialogRef?.setCreate()
  this.weightEntriesDialogRef?.show()
}

```

Κώδικας 81 Αρχικοποίηση dialog δημιουργίας εγγραφής βάρους

```

private onCreateLineRecord = async (vm: RecordCreationViewModel) : Promise<void> => {
  await this.createRecord(vm).catch((err) => this.setState( state: {serverError: err.message}))
  this.weightEntriesDialogRef?.setData(this.state.records)
}

```

Κώδικας 82 Αίτημα δημιουργίας εγγραφής βάρους ή θερμοκρασίας

```

private createRecord = async (vm: RecordCreationViewModel) : Promise<void> => {
  const id : string = this.props.router.query.id as string
  if (!id) {
    throw Error( message: "Pet Id was not defined")
  }

  const resp : Response = await recordCreationHandler(vm, id, this.state.token)
  const response: Record = await resp.json()
  if (resp.ok) {
    this.state.records.push(response)
    this.setState( state: {records: this.state.records}, callback: () : void => {
      this.updateLineDatasets()
      this.updateEvents()
    })
  } else if (resp.status === 401) {
    this.logout( callback: () => this.props.router.replace( url: "/auth/login"))
  } else {
    throw Error((response as ErrorDto).error)
  }
}

```

Κώδικας 83 Αίτημα δημιουργίας μιας εγγραφής

Για τη δημιουργία, την επεξεργασία και τη διαγραφή μια εγγραφής δεν έχει σημασία αν η εγγραφή παρουσιάζεται με τη μορφή διαγράμματος ή όχι. Για την εφαρμογή backend όλες οι εγγραφές είναι ίδιες απλά είναι διαφορετικά τα πεδία που πρέπει να συμπληρωθούν ανάλογα τον τύπο εγγραφής.

Στη διεπαφή πελάτη διαχωρίζονται για να παρουσιαστούν με πιο ουσιαστική μορφή για τον τελικό χρήστη.



Εικόνα 17 Dialog μαζικής δημιουργίας/επεξεργασίας/διαγραφής εγγράφων βάρους ή θερμοκρασίας

Εφόσον ο χρήστης το επιθυμεί από το dialog μαζικής επεξεργασίας μπορεί να επιλέξει το κουμπί με το μολύβι και να επεξεργαστεί την εγγραφή. Αλλιώς μπορεί να τη διαγράψει πατώντας το εικονίδιο “X” και επιβεβαιώνοντας την επιλογή του.

```
private updateRecord = async (vm: UpdateRecordViewModel, id?: string) : Promise<void> => {
  if (!id) {
    throw Error( message: "no record to update")
  }

  const petId : string = this.props.router.query.id as string
  if (!petId) {
    throw Error( message: "Pet Id was not defined")
  }

  const resp : Response = await recordUpdateHandler(vm, id, petId, this.state.token)
  const response: Record = await resp.json()
  if (resp.ok) {
    const records : Record[] = this.state.records
    const index : number = records.findIndex((it : Record) : boolean => it.id === id)
    records[index] = response
    this.setState( state: {records}, callback: () : void => {
      this.updateLineDatasets()
      this.updateEvents()
    })
  } else if (resp.status === 401) {
    this.logout( callback: () => this.props.router.replace( url: "/auth/login"))
  } else {
    throw Error((response as ErrorDto).error)
  }
}
```

Κώδικας 84 Αίτημα ανανέωσης εγγραφής

```

private deleteRecord = async () : Promise<void> => {
  this.deleteDialogRef?.hide()

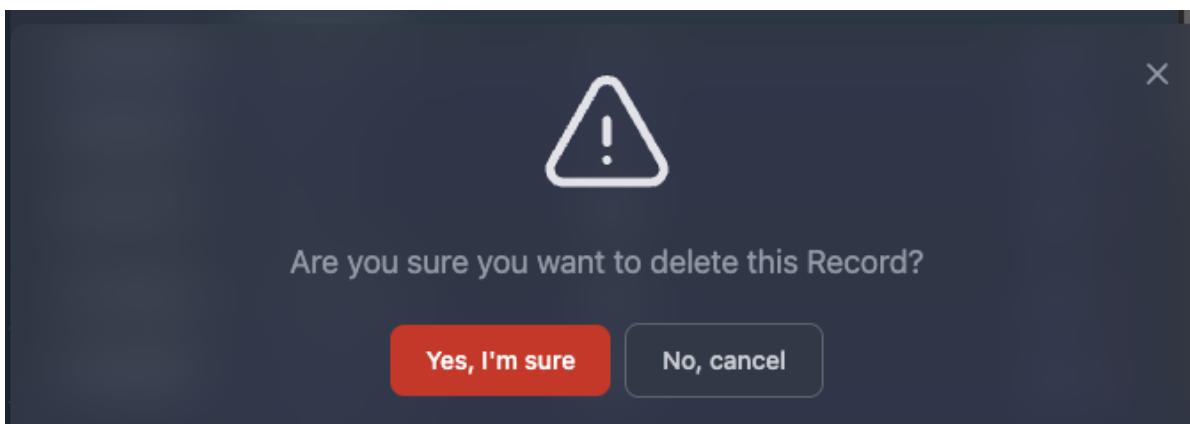
  const id : string | undefined = this.state.selectedRecordId
  if (!id) {
    throw Error( message: "no record to delete")
  }

  const petId : string = this.props.router.query.id as string
  if (!petId) {
    throw Error( message: "Pet Id was not defined")
  }

  const resp : Response = await recordDeletionHandler(id, petId, this.state.token)
  const response: Record = await resp.json()
  if (resp.ok) {
    const newRecords : Record[] = this.state.records.filter((r : Record) : boolean => r.id !== id)
    this.setState( state: {records: newRecords}, callback: () : void => {
      this.updateLineDatasets()
      this.updateEvents()
      this.weightEntriesDialogRef?.setData(newRecords)
    })
  } else if (resp.status === 401) {
    this.logout( callback: () => this.props.router.replace( url: "/auth/login"))
  } else {
    throw Error((response as ErrorDto).error)
  }
}

```

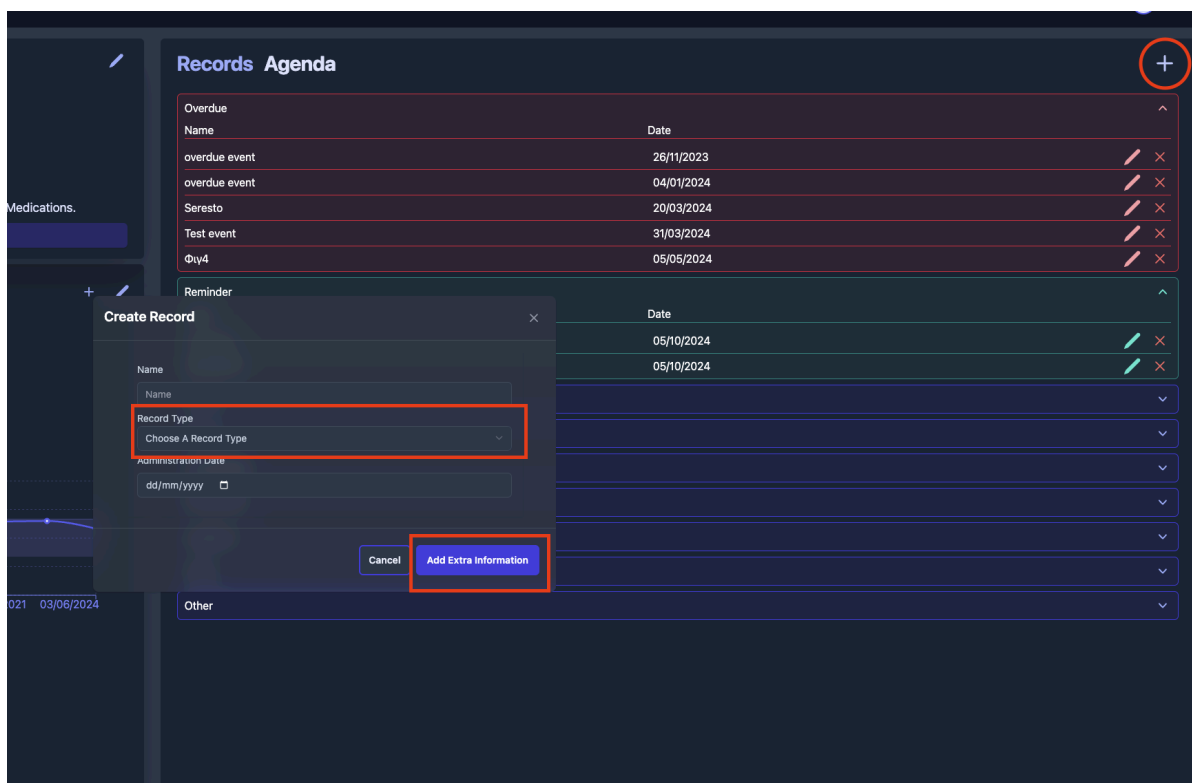
Κώδικας 85 Αίτημα διαγραφής εγγραφής



Εικόνα 18 Dialog επιβεβαίωσης διαγραφής εγγραφής

Από το δεξί μέρος ο χρήστης μπορεί να προσθέσει εγγραφές σχετικά με αλλά σημαντικά γεγονότα του κατοικίδιου όπως εμβολιασμοί, χειρουργεία κα.

Εμφανίζεται ένα dialog αντίστοιχο με αυτό της δημιουργίας εγγραφής βάρους μόνο που περιέχει κάποια επιπλέον σημαντικά πεδία όπως ο τύπος της εγγραφής καθώς και επιπλέον πληροφορίες που ίσως χρειάζονται ανάλογα τον τύπο.



Εικόνα 19 Dialog προσθήκης εγγραφής

Πολύ σημαντική επίσης είναι ότι ο χρήστης μπορεί να δημιουργήσει εγγραφές σε μελλοντική ημερομηνία, καθώς και να προγραμματίσει την επόμενη εγγραφή στην περίπτωση που είναι επαναλαμβανόμενη.

Create Record

Lot Number

Result

Description

Notes

☐ Is recurring

Please fill in this field.

Back Create Record

Εικόνα 20 Επαναλαμβανόμενη εγγραφή

Create Record

Schedule Next Occurance

dd/mm/yyyy

Back

Create Record

Εικόνα 21 εισαγωγή ημερομηνίας επανάληψης εγγραφής

Οι εγγραφές που είναι σε μελλοντική ημερομηνία εμφανίζονται ως reminder στη λίστα με πράσινο χρώμα. Εάν η ημερομηνία έχει περάσει αλλά δεν έχει γίνει κάποια ενέργεια τότε θεωρούνται εκπρόθεσμες και εμφανίζονται πρώτες στην λίστα με κόκκινο χρώμα.

Τέλος η λίστα με τις εγγραφές μπορεί να λάβει τη μορφή της ατζέντας του κατοικίδιου επιλέγοντας το κουμπί “ατζέντα”

Records Agenda

+

Today Back Next

March 2024

Month

Week

Day

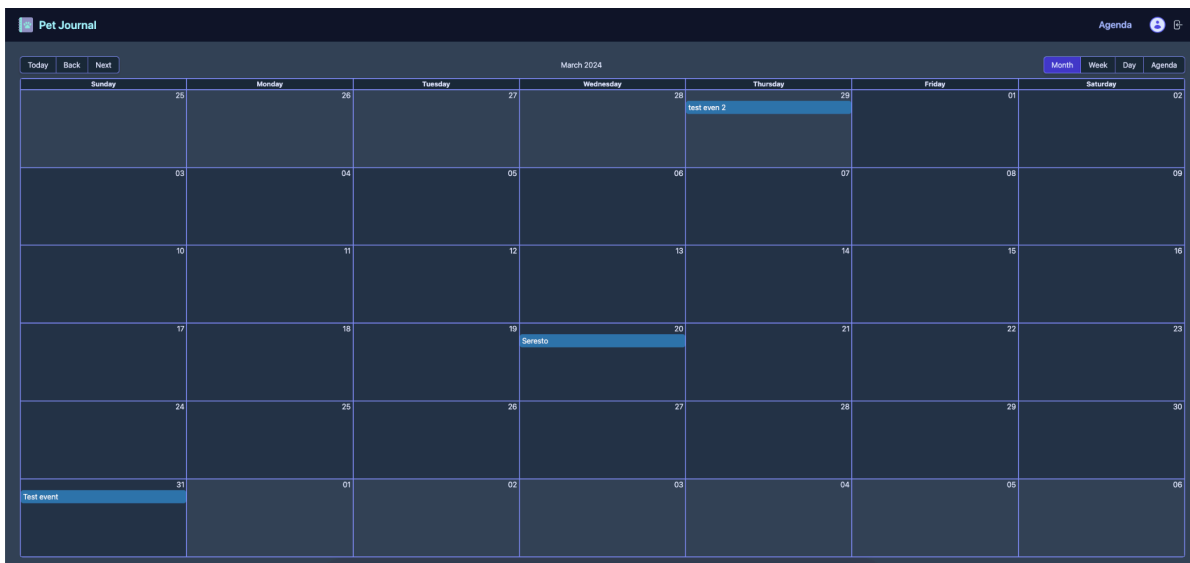
Agenda

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
25	26	27	28	29	01	02
03	04	05	06	07	08	09
10	11	12	13	14	15	16
17	18	19	20 Seresto	21	22	23
24	25	26	27	28	29	30
31 Test event	01	02	03	04	05	06

Εικόνα 23 Προβολή ατζέντας στο προφίλ κατοικίδιου

3.2.6 Οθόνη ατζέντας

Ο χρήστης μπορεί να πλοηγηθεί στην οθόνη ατζέντας από τη μπάρα πλοήγησης. Σε αυτή τη σελίδα θα βρει ένα ημερολόγιο που περιέχει τη συνολική εικόνα από τις εγγραφές όλων των κατοικίδιων.



Εικόνα 24 Οθόνη συνολικής ατζέντας

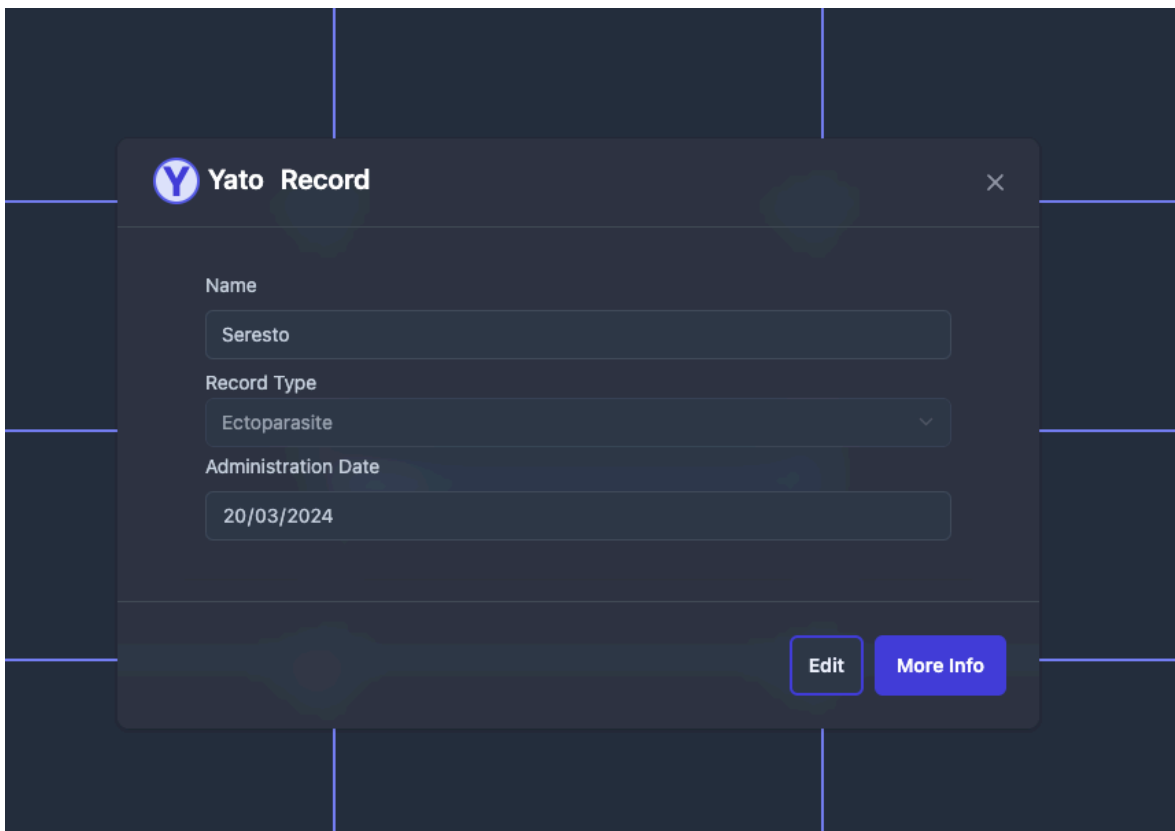
Όταν ένας αυθεντικοποιημένος χρήστης εισέρχεται στη σελίδα στέλνεται αίτημα στον server να φέρει όλα τα records από όλα τα κατοικίδια που έχει πρόσβαση ο χρήστης.

```
private initPage = (token?: string) : void => {
  this.getAllRecords(token).catch((e) : void => {
    console.error("Could not get User", e)
    this.setState( state: {serverError: e.message})
  })
  this.setState( state: {token, loading: true})
}

private updateEvents = () : void => {
  const events : CalendarEvent[] = this.state.records
  .filter((it : Record) => it.recordType !== RecordType.WEIGHT && it.recordType !== RecordType.TEMPERATURE)
  .flatMap( callback: (r : Record) => {
    const events: CalendarEvent[] = []
    const d : Date = new Date(r.date!)
    events.push(new CalendarEvent( allDay: true, r.name!, d, d, r, isEvent: true))
    return events
  })
  this.setState( state: {events, loading: false})
}
```

Κώδικας 86 Εκκίνηση οθόνης ατζέντας

Κάθε record εμφανίζεται στο ημερολόγιο σαν ένα event. Για να δει ο χρήστης περισσότερες πληροφορίες για το event αυτό μπορεί να το επιλέξει και θα εμφανιστεί ένα dialog με τις πληροφορίες της εγγραφής καθώς και ποιο κατοικίδιο αφορά.



Εικόνα 25 Προβολή επιλεγμένου record

Εφόσον χρειάζεται να γίνει κάποια αλλαγή στα στοιχεία της εγγραφής τότε ο χρήστης μπορεί να επιλέξει το κουμπί “επεξεργασία” και να ενημερώσει τα στοιχεία της εγγραφής.

```
private updateRecord = async (vm: UpdateRecordViewModel, r?: Record) : Promise<void> => {
  const id : string | undefined = r?.id
  if (!id) {
    throw Error( message: "no record to update")
  }

  const petId : string | undefined = r?.pet?.id
  if (!petId) {
    throw Error( message: "Pet Id was not defined")
  }

  const resp : Response = await recordUpdateHandler(vm, id, petId, this.state.token)
  const response: Record = await resp.json()
  if (resp.ok) {
    const records : Record[] = this.state.records
    const index : number = records.findIndex((it : Record) : boolean => it.id === id)
    records[index] = response
    this.setState( state: {records: records}, callback: () => this.updateEvents())
  } else if (resp.status === 401) {
    this.logout( callback: () => this.props.router.replace( url: "/auth/login"))
  } else {
    throw Error((response as ErrorDto).error)
  }
}
```

Κώδικας 87 Ενημέρωση των στοιχείων της εγγραφής

3.2.7 Οθόνη προφίλ χρήστη

Τέλος ο χρήστης από τη μπάρα πλοήγησης μπορεί να βρει το εικονίδιο που τον οδηγεί στη σελίδα του προφίλ του. Εκεί μπορεί να δει όλα τα στοιχεία που έχει δώσει κατά την εγγραφή καθώς και να τα επεξεργαστεί οποιοδήποτε στιγμή επιθυμεί.

Με την είσοδο του χρήστη στη σελίδα στέλνουμε ένα αίτημα στον server για να λάβουμε τα δεδομένα προφίλ που έχουμε στη βάση δεδομένων και στη συνέχεια τα φορτώνουμε στο viewmodel του λογαριασμού του χρήστη.

```
private initVm = () : void => {
    const u : User | undefined = this.state.user
    const vm : UpdateUserViewModel = this.state.vm
    vm.email = TextUtils.valueOrEmpty(u?.email, defaultV: "")
    vm.name = TextUtils.valueOrEmpty(u?.name, defaultV: "")
    vm.surname = TextUtils.valueOrEmpty(u?.surname, defaultV: "")
    vm.phone = TextUtils.valueOrEmpty(u?.phone, defaultV: "")
    vm.address = TextUtils.valueOrEmpty(u?.address, defaultV: "")
    vm.city = TextUtils.valueOrEmpty(u?.city, defaultV: "")
    vm.zip = TextUtils.valueOrEmpty(u?.zip, defaultV: "")
    vm.state = TextUtils.valueOrEmpty(u?.state, defaultV: "")
    vm.country = TextUtils.valueOrEmpty(u?.country, defaultV: "")

    this.setState( state: {vm})
}

private getAccount = async (token?: string) : Promise<void> => {
    const resp : Response = await userHandler(token)
    const data : User = await resp.json()
    if (resp.ok) {
        this.setState( state: {user: data}, callback: () => this.initVm())
    } else if (resp.status === 401) {
        this.logout( callback: () => this.props.router.replace( url: "/auth/login"))
    } else {
        throw new Error((data as ErrorDto).error)
    }
}

private initPage = (token?: string) : void => {
    this.setState( state: {token})

    this.getAccount(token).catch((e) : void => {
        console.error("Could not get User", e)
        this.setState( state: {serverError: e.message})
    })

    countries.registerLocale(require("i18n-iso-countries/langs/en.json"))
    const names : countries.LocalizedCountryName... = countries.getNames( lang: "en", options: {select: "official"})
    this.setState( state: {countriesKeys: Object.keys(names)})
}
```

Κώδικας 88 Εκκίνηση σελίδας προφίλ χρήστη

Εικόνα 26 Σελίδα προφίλ χρήστη

Σε περίπτωση που ο χρήστης θέλει επεξεργαστεί τα δεδομένα του τότε μπορεί να πατήσει το κουμπί “επεξεργασία”.

Όταν τελειώσει με τις αλλαγές και πατήσει αποθήκευση τότε στέλνουμε το αντίστοιχο αίτημα στον server για να ενημερωθούν τα στοιχεία. Σε περίπτωση κάποιου σφάλματος το εμφανίζουμε στην οθόνη, αλλιώς αναμένουμε τα δεδομένα όπως τα λαμβάνουμε από τον server και βγαίνουμε από το mode επεξεργασίας.

```
private onSubmit = async () : Promise<void> => {
  const vm : UpdateUserViewModel = this.state.vm
  const token : string | undefined = this.state.token

  vm.validate()
  if (!vm.isValid) {
    return
  }

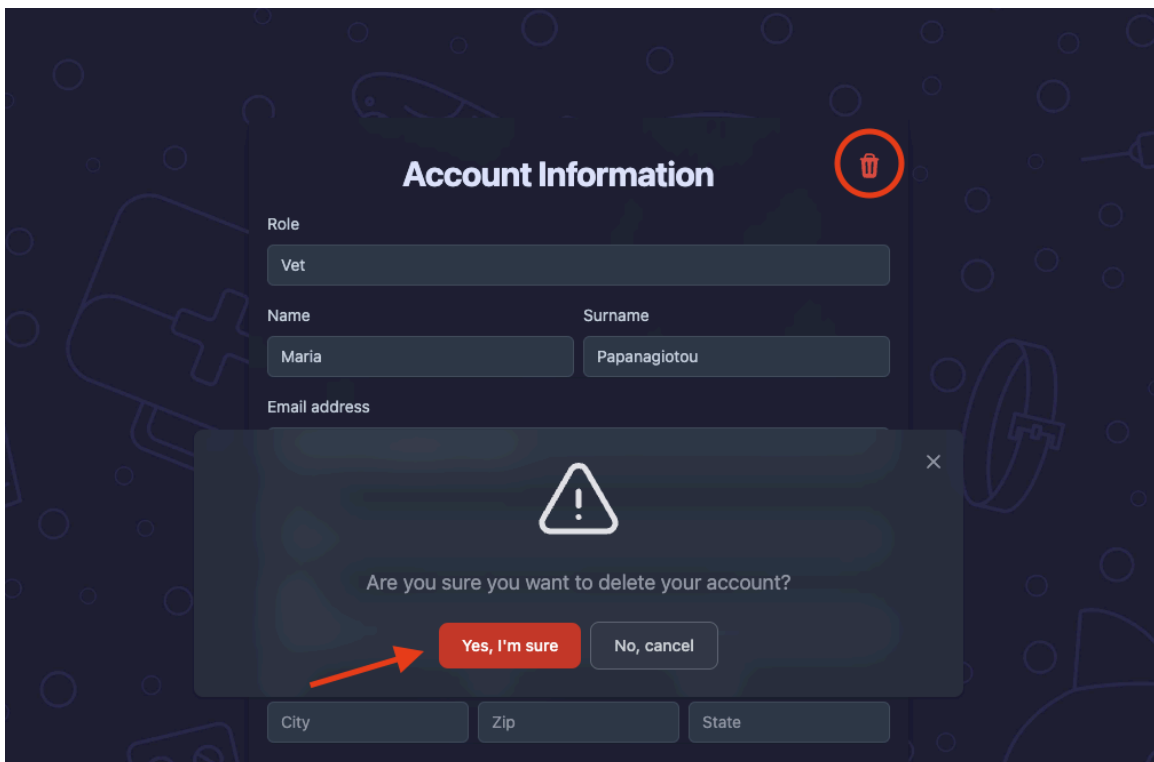
  const response : Response = await userUpdateHandler(vm, token)

  const data : User = await response.json()

  if (response.ok) {
    this.setState( state: {serverError: "", viewOnly: true, user: data})
  } else if (response.status === 401) {
    this.logout( callback: () => this.props.router.replace( url: "/auth/login"))
  } else {
    // update failed
    console.error("update failed", data)
    this.setState( state: {serverError: (data as ErrorDto).error ?? ""})
  }
}
```

Κώδικας 89 Ενημέρωση των στοιχείων χρήστη

Εάν ο χρήστης επιθυμεί να διαγράψει τον λογαριασμό του μπορεί να πατήσει το εικονίδιο του κάδου πάνω δεξιά και να επιβεβαιώσει την απόφαση του στο dialog επιβεβαίωσης διαγραφής λογαριασμού.



Εικόνα 27 Επιβεβαίωση διαγραφής λογαριασμού χρήστη

```
private deletePressed = async () : Promise<void> => {
  this.deleteDialogRef?.hide()

  const resp : Response = await userDeleteHandler(this.state.token)
  const data: {message: string; error?: string} = await resp.json()
  if (resp.ok) {
    this.logout( callback: () => this.props.router.replace( url: "/auth/login"))
  } else {
    this.setState( state: {serverError: data.error})
  }
}
```

Κώδικας 90 Διαγραφή λογαριασμού χρήστη

```
export function userDeleteHandler(token?: string): Promise<Response>
| return fetch( input: `${process.env.apiUrl}/user`, init: {
|   method: "DELETE",
|   headers: {
|     Authorization: `Bearer ${token}`,
|   },
| })
| }
```

Κώδικας 91 Αίτημα Διαγραφής λογαριασμού χρήστη

Κεφάλαιο 4ο - Deployment

Η παραγωγή και εκτέλεση ενός εκτελέσιμου αρχείου της εφαρμογής αποτελούν ουσιαστικό βήμα στη διαδικασία ανάπτυξης λογισμικού. Για τον σκοπό αυτό, επιλέχθηκε η χρήση του Docker [31].

Το Docker παρέχει ένα πλήρες σύστημα για τη δημιουργία εικονικών εφαρμογών, τα οποία στη συνέχεια μπορούν να εκτελεστούν ανεξάρτητα από το περιβάλλον όπου εκτελούνται. Με τη χρήση του Docker, επιτυγχάνεται η δημιουργία ενός εκτελέσιμου αρχείου της εφαρμογής που περιλαμβάνει όλα τα απαραίτητα dependencies και ρυθμίσεις, εξασφαλίζοντας τη συνοχή και αξιοπιστία της εκτέλεσης [32].

Με αυτή την προσέγγιση, η διαδικασία παραγωγής και εκτέλεσης γίνεται απλούστερη και πιο ευέλικτη, ενώ η εφαρμογή είναι πλέον φορητή και μπορεί να εκτελεστεί σε διάφορα περιβάλλοντα χωρίς την ανάγκη για σύνθετες ρυθμίσεις.

4.1 Building process

Το Next.js προσφέρει τη δυνατότητα παραγωγής στατικών σελίδων μέσω της διαδικασίας build, δημιουργώντας έναν φάκελο που περιλαμβάνει όλες τις απαραίτητες στατικές σελίδες. Κατά την ανταπόκριση σε αιτήματα χρηστών, ο server διανέμει αυτές τις σελίδες, έχοντας σχετικό dependency στον φάκελο με τις στατικές σελίδες. Για τον σκοπό αυτό, απαιτείται η δημιουργία μιας διαδικασίας με το Docker που θα αναλάβει το build της εφαρμογής πελάτη, μετατρέποντας τη σε στατικές σελίδες.

```
FROM node:18 as frontend

WORKDIR /app

# Copy the entire application code
COPY ui .

RUN yarn install

# Build the Next.js application and output to the "build" directory
RUN yarn build
```

Κώδικας 92 Διαδικασία Docker Build για τη διεπαφή χρήστη

Η Διεργασία ξεκινάει αντιγράφοντας την εικόνα για το Node v18 από το Docker Hub. Έπειτα μεταφερόμαστε στον φάκελο app όπου αντιγράφεται όλος ο κώδικας που υπάρχει στο φάκελο ui. Το επόμενο βήμα είναι να εγκαταστήσουμε όλες τις απαραίτητες βιβλιοθήκες που χρησιμοποιεί η εφαρμογή. Τέλος μπορούμε να χτίσουμε την εφαρμογή όπως έχουμε δηλώσει στο configuration από το NextJs.

```
const nextConfig : NextConfig = {
  output: "export",
  env: {
    apiUrl: "/api",
  },
  images: {
    unoptimized: true
  },
  distDir: 'build'
}
```

```
module.exports = nextConfig
```

Κώδικας 93 NextJs Build configuration

Αφού δημιουργηθεί ο φάκελος που περιέχει τις σελίδες της διεπαφής χρήστη χρειάζεται να τον μεταφέρουμε στον φάκελο public που υπάρχει μέσα στον server. Έπειτα έχουμε ότι χρειάζεται για να μπορέσουμε να παράξουμε τον πηγαίο κώδικα της εφαρμογής.

```
FROM golang:1.21.3 as builder

# Set the working directory to /go/src/app
WORKDIR /go/src/app

# Copy the local package files to the container's workspace
COPY server .

# Copy the "build" directory from the frontend build
COPY --from=frontend /app/build ./cmd/server/public

# Set environment variables
ENV CGO_ENABLED=0
ENV GOOS=linux
ENV GOARCH=amd64

# Build the Go application
RUN go build -o petJournal ./cmd/server
```

Κώδικας 94 Server build process

Η Διεργασία ξεκινάει αντιγράφοντας την εικόνα για τη golang v1.21.3 από το Docker Hub. Έπειτα μεταφερόμαστε στο φάκελο go>source>app όπου αντιγράφεται όλος ο κώδικας που υπάρχει στον φάκελο server. Αντιγράφεται ο φάκελος που δημιουργήθηκε στην προηγούμενη διεργασία στο φάκελο server>public. Στη συνέχεια δηλώνουμε τις απαραίτητες μεταβλητές για το περιβάλλον ανάπτυξης της εφαρμογής. Τέλος χτίζουμε την εφαρμογή.

```
# Start a new stage to reduce the final image size
FROM alpine:latest

# Set the working directory to /app
WORKDIR /app

# Copy the binary from the builder stage to the final stage
COPY --from=builder /go/src/app/petJournal .

# Set environment variable for your application
ENV SECRET_KEY=$SECRET_KEY
ENV DB_URL=$DB_URL
ENV DB_NAME=$DB_NAME

# Expose port 8080 for the application
EXPOSE 8080

# Define the command to run the application
ENTRYPOINT [ "./petJournal" ]
```

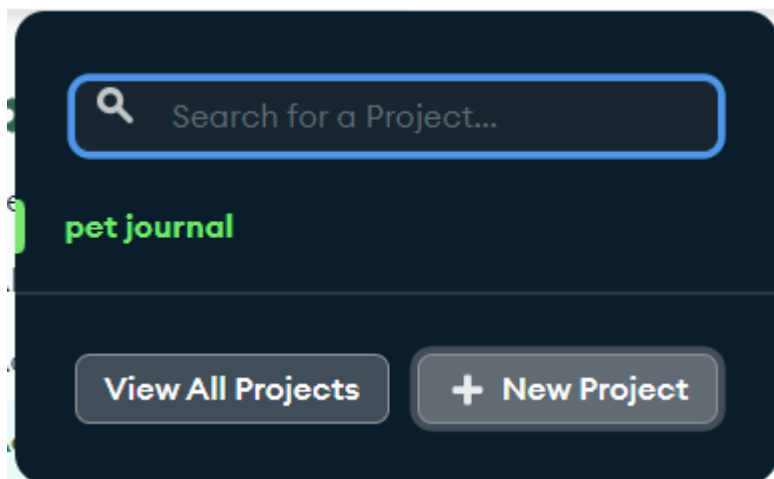
Κώδικας 95 Application start-up process

Τελευταίο βήμα είναι η εκτέλεση της εφαρμογής σε ένα docker container. Ορίζουμε ως λογισμικό το τελευταίο alpine distribution. Μεταφερόμαστε στο φάκελο app. Αντιγράφουμε στον φάκελο μόνο το εκτελέσιμο αρχείο όπως παράχθηκε στο προηγούμενο βήμα. Θέτουμε τις μεταβλητές που χρειάζεται ο server για να ξεκινήσει. Οι μεταβλητές αυτές είναι κρυφές για τους άλλους χρήστες και ορίζονται στο σύστημα του κάθε χρήστη. Έπειτα δηλώνουμε την πόρτα που μπορούμε να κάνουμε ερωτήματα στην εφαρμογή. Τέλος τρέχουμε την εφαρμογή, δημιουργείται ένα container και πλέον μπορούμε να έχουμε πρόσβαση στην εφαρμογή από το localhost:8080.

4.2 Δημιουργία βάσης δεδομένων

Για τη δημιουργία μιας βάσης δεδομένων χρησιμοποιήθηκε το atlas [33] που προσφέρει υπηρεσίες για την mongoDb.

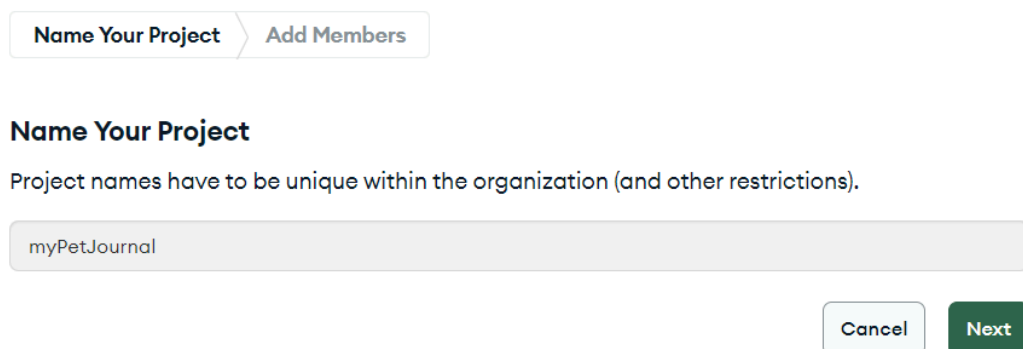
1. Δημιουργούμε ένα νέο project στην πλατφόρμα



Εικόνα 28 Δημιουργία νέου project

2. Δημιουργούμε τη βάση δεδομένων.

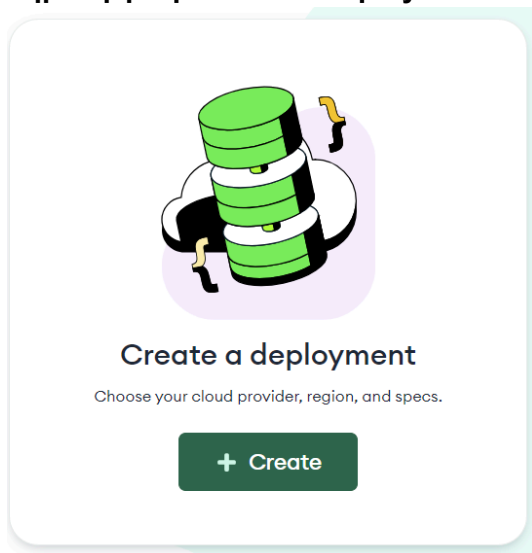
Το όνομα αυτού του πεδίου ορίζεται στο server σαν μεταβλητή περιβάλλοντος. (BD_NAME)



The screenshot shows a step titled "Name Your Project" with a subtitle "Project names have to be unique within the organization (and other restrictions)." Below this is a text input field containing "myPetJournal". At the bottom right are "Cancel" and "Next" buttons.

Εικόνα 29 Δημιουργούμε τη βάση δεδομένων.

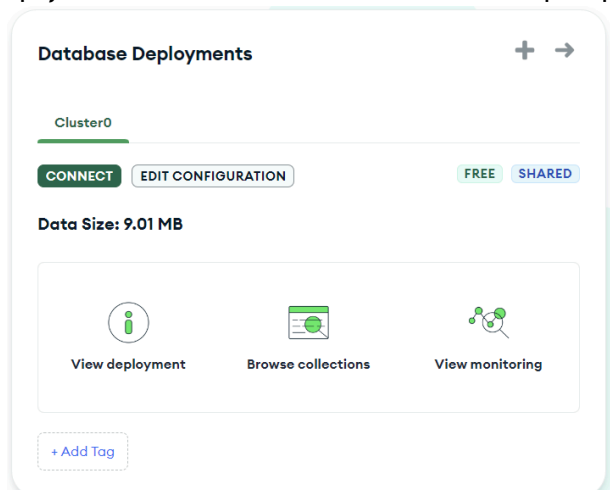
3. Δημιουργούμε ένα νέο deployment



Εικόνα 30 Δημιουργούμε νέο deployment

4. Συνδέουμε τη βάση στο server.

Πατώντας connect λαμβάνουμε το Url σύνδεσης στη βάση. Το όνομα αυτού του πεδίου ορίζεται στον server σαν μεταβλητή περιβάλλοντος. (BD_URL)



Εικόνα 31 Σύνδεση στη βάση

Κεφάλαιο 5ο - Συμπεράσματα και Μελλοντικές βελτιώσεις

5.1 Συμπεράσματα

Ο βασικός στόχος της παρούσας εργασίας ήταν η σχεδίαση και η ανάπτυξη μιας διαδικτυακής εφαρμογής για την ευημερία και ευζωία των κατοικίδιων, καθώς και την αποθήκευση και διαχείριση των ιατρικών τους δεδομένων από κτηνιάτρους και κηδεμόνες.

Κατά τον σχεδιασμό της εφαρμογής προέκυψαν οι απαιτήσεις της εφαρμογής για τη λειτουργικότητα της εφαρμογής ώστε να είναι χρήσιμη αλλά και εύχρηστη, τόσο για τους κτηνιάτρους όσο και για τους κηδεμόνες των κατοικίδιων. Από τον αρχικό σχεδιασμό προέκυψαν οι βασικές απαιτήσεις της εφαρμογής.

Στο στάδιο της ανάπτυξης μοντελοποιήθηκαν οι βασικές οντότητες και συσχετίσεις μεταξύ των κύριων δομικών μοντέλων. Πάνω στα μοντέλα αυτά βασίστηκε όλη η λειτουργικότητα και η αλληλεπίδραση της εφαρμογής.

Η τελική εφαρμογή που αναπτύχθηκε προσφέρει έναν αποτελεσματικό τρόπο αποθήκευσης των ιατρικών δεδομένων των κατοικίδιων από τους κηδεμόνες τους και από τον κτηνίατρο τους. Επιπρόσθετα δίνεται η δυνατότητα διαμοιρασμού των δεδομένων αυτών μεταξύ κηδεμόνων και κτηνιάτρων για πιο αποτελεσματική επικοινωνία και γνώση της κλινικής εικόνας των κατοικίδιων ανά πάσα στιγμή. Τέλος δίνεται η δυνατότητα για προγραμματισμό μελλοντικών επισκέψεων που μπορεί να χρησιμοποιηθεί ως υπενθύμιση για τους κηδεμόνες, αλλά και ως σύστημα διαχείρισης επόμενων επισκέψεων για τους κτηνιάτρους.

5.2 Μελλοντικές βελτιώσεις

Η εφαρμογή που αναπτύχθηκε επικεντρώθηκε κυρίως στην ψηφιοποίηση και των διαμοιρασμό των ιατρικών δεδομένων μεταξύ κτηνιάτρων και κηδεμόνων. Σε μελλοντική επέκταση της εφαρμογής θα μπορούσαν να προστεθούν κάποιες βελτιώσεις. Κάποιες από αυτές είναι οι εξής:

- Η υποστήριξη διαμοιρασμού των δεδομένων μεταξύ άλλων κηδεμόνων και όχι απαραίτητα μεταξύ κηδεμόνα/κτηνιάτρου.
- Η παροχή υπηρεσίας ηλεκτρονικών ραντεβού και διαχείριση αυτών από τους κτηνιάτρους.
- Η προσθήκη chat μεταξύ κηδεμόνων και κτηνιάτρου για πιο γρήγορη επικοινωνία με χρήση εικόνων, μηνυμάτων και βίντεο για μια πιο ολοκληρωμένη απομακρυσμένη βοήθεια.
- Η μετάφραση της εφαρμογής σε άλλες γλώσσες πέραν της αγγλικής καθώς και ο καλύτερος σχεδιασμός για κινητές συσκευές.
- Τέλος σημαντική θα ήταν και η λειτουργία προγραμματισμού επαναλαμβανόμενων γεγονότων και η υπενθύμιση αυτών μέσω ειδοποιήσεων.

Βιβλιογραφία και Ιστογραφία

- [1] Janke, N., Coe, J. B., Bernardo, T. M., Dewey, C. E., & Stone, E. A. (2021). Pet owners' and veterinarians' perceptions of information exchange and clinical decision-making in companion animal practice. *PloS one*, 16(2), e0245632. <https://doi.org/10.1371/journal.pone.0245632>
- [2] Paynter, A. N., Dunbar, M. D., Creevy, K. E., & Ruple, A. (2021). Veterinary Big Data: When Data Goes to the Dogs. *Animals : an open access journal from MDPI*, 11(7), 1872. <https://doi.org/10.3390/ani11071872>
- [3] Jack Tsai, Gary Bond, A comparison of electronic records to paper records in mental health centers, *International Journal for Quality in Health Care*, Volume 20, Issue 2, April 2008, Pages 136–143, <https://doi.org/10.1093/intqhc/mzm064>
- [4] go.dev (2024), Build simple, secure, scalable systems with Go, <https://go.dev/>
- [5] gin-gonic.com (2024), Gin Web Framework, <https://gin-gonic.com/>
- [6] gin-gonic.com (2024), Gin Introduction, <https://gin-gonic.com/docs/introduction/>
- [7] nextjs.org (2024), The React Framework for the Web, <https://nextjs.org/>
- [8] react.dev (2024), React: The library for web and native user interfaces, <https://react.dev/>
- [9] nextjs.org (2024), Next js Introduction, <https://nextjs.org/docs/>
- [10] html.com (2024), HTML For Beginners The Easy Way: Start Learning HTML & CSS Today, <https://html.com/>
- [11] html.com (2024), What is HTML?, https://html.com/#What_is_HTML
- [12] tailwindui.com (2024), Build your next idea even faster, <https://tailwindui.com/>
- [13] tailwindui.com (2024), Beautiful UI components, crafted with Tailwind CSS, <https://tailwindui.com/components>
- [14] typescriptlang.org (2024), TypeScript is JavaScript with syntax for types, <https://www.typescriptlang.org/>
- [15] typescriptlang.org (2024), TypeScript for JavaScript Programmers, <https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html>
- [16] heroicons.com (2024), Beautiful hand-crafted SVG icons, by the makers of Tailwind CSS, <https://heroicons.com/>
- [17] mongodb.com (2024), MongoDB Community Edition, <https://www.mongodb.com/products/self-managed/community-edition>
- [18] json.org (2024), Introducing JSON, <https://www.json.org/json-en.html>
- [19] openapis.org (2024), OpenAPI Specification, <https://spec.openapis.org/oas/latest.html#openapi-specification>
- [20] aws.amazon.com (2024), What is a RESTful API?, <https://aws.amazon.com/what-is/restful-api/>
- [21] Evans, E. (2004). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.
- [22] developer.mozilla.org (2024), Hypertext Transfer Protocol (HTTP),

<https://developer.mozilla.org/en-US/docs/Web/HTTP>

[23] medium.com (2024), Repository Pattern in Golang, <https://medium.com/@pererikbergman/repository-design-pattern-e28c0f3e4a30>

[24] recharts.org (2024), A composable charting library built on React components, <https://recharts.org/en-US/>

[25] jquense.github.io (2024), Big Calendar, <https://jquense.github.io/react-big-calendar/examples/index.html?path=/story/about-big-calendar--page>

[26] github.com (2024), react-color-extractor, <https://github.com/nitin42/react-color-extractor>

[27] bgrins.github.io (2024), TinyColor: Fast, small color manipulation and conversion for JavaScript, <https://bgrins.github.io/TinyColor/>

[28] nextjs.org (2024), Next.js Project Structure, <https://nextjs.org/docs/getting-started/project-structure>

[29] blog.logrocket.com (2024), JWT authentication: Best practices and when to use it, <https://blog.logrocket.com/jwt-authentication-best-practices/>

[30] dev.to (2024), How to implement protected routes in Next.js, <https://dev.to/seven/how-to-implement-protected-routes-in-nextjs-1m50>

[31] docker.com (2024), Accelerate how you build, share, and run applications, <https://www.docker.com/>

[32] docker.com (2024), Multi-stage builds, <https://docs.docker.com/build/building/multi-stage/>

[33] mongodb.com (2024), MongoDB Atlas: The multi-cloud developer data platform, <https://www.mongodb.com/atlas>