ΠΑΝΕΜΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ

ΤΜΗΜΑ ΒΙΟΜΗΧΑΝΙΚΗΣ ΣΧΕΔΙΑΣΗΣ & ΠΑΡΑΓΩΓΗΣ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Δημιουργία κυβερνοφυσικού συστήματος για την γεωργία.

Αθανασίου Παναγιώτης 71446946

ΑΘΗΝΑ, ΙΟΥΛΙΟΣ 2021

Η παρούσα διπλωματική εργασία εγκρίθηκε ομόφωνα από την τριμελή εξεταστική επιτροπή, η οποία ορίστικε από τη Γ.Σ. του Τμήματος Μηχανικών Βιομηχανικής Σχεδίασης και Παραγωγής του Πανεπιστήμιου Δυτικής Αττικής, σύμφωνα με το νόμο και τον εγκεκριμένο Οδηγό Σπουδών του τμήματος.

Επιβλέπων: Δρ. Πυρομάλης Δημήτριος
Επίκουρος Καθηγητής

Επιτροπή Αξιολόγησής:

Dimitrios Pyromalis
Digitally signed by Dimitrios Pyromalis
Date: 2021.07.21 17:12:50 +03'00'

Δρ. Πυρομάλης Δημήτριος
Επίκουρος Καθηγητής

Avraam Chatzopoulos
Digitally signed by Avraam Chatzopoulos
Date: 2021.07.23 20:07:51 +03'00'

Χατζόπουλος Αβραάμ
Λέκτορας Εφαρμογών

Christos Drosos
Digitally signed by Christos Drosos
Date: 2021.07.24 00:06:12 +03'00'

Δρόσος Χρήστος
ΕΔΙΠ

ΔΗΛΩΣΗ ΣΥΓΓΡΑΦΕΑ ΠΤΥΧΙΑΚΗΣ ΕΡΓΑΣΙΑΣ

Ο/η κάτωθι υπογεγραμμένος/η Αθανασίου Παναγιώτης του Παναγιώτη, με αριθμό μητρώου 71446946 φοιτητής/τρια του Πανεπιστημίου Δυτικής Αττικής της Σχολής Μηχανικών του Τμήματος Βιομηχανικής Σχεδίασης και Παραγωγής δηλώνω υπεύθυνα ότι:

«Είμαι συγγραφέας αυτής της πτυχιακής/διπλωματικής εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία.
Επίσης, οι όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών ή λέξεων, είτε ακριβώς είτε παραφρασμένες, αναφέρονται στο σύνολό τους, με πλήρη αναφορά στους συγγραφείς, τον εκδοτικό οίκο ή το περιοδικό, συμπεριλαμβανομένων και των πηγών που ενδεχομένως χρησιμοποιήθηκαν από το διαδίκτυο. Επίσης, βεβαιώνω ότι αυτή η εργασία έχει συγγραφεί από μένα αποκλειστικά και αποτελεί προϊόν πνευματικής ιδιοκτησίας τόσο δικής μου, όσο και του Ιδρύματος.

Παράβαση της ανωτέρω ακαδημαϊκής μου ευθύνης αποτελεί ουσιώδη λόγο για την ανάκληση του πτυχίου μου».

Ο/Η Δηλών/ούσα

……..

Αθανασίου Παναγιώτης

Ευχαριστίες

Θα ήθελα να ευχαριστήσω παρα πολύ τον επιβλέποντα καθηγητή μου Δρ. Πυρομάλη Δημήτριο για την καθοδήγηση του και την βοήθεια του για την επιτυχή ολοκλήρωση της εργασίας.

Πίνακας Περιεχομένων

1: ΕΙΣΑΓΩΓΗ

Η εργασία περιγράφει την δημιουργία μιας κυβερνοφυσιής εφαρμογής για την Γεωργία. Στο τέλος της εργασίας θα δούμε ένα ολοκληρωμένο σύστημα για τον αυτόματο έλεγχο και την διαχείριση συστήματος ποτίσου. Η διαχείριση θα γίνετε μέσο ιστοσελίδας ενώ μας δίνετε η δυνατότητα εισόδου και ελέγχου μέσω χρηστών.

Κάθε μέρα χρησιμοποιούνται τόνοι νερό για την καλλιέργεια διάφορων φυτών, αυτά τα φυτά συνηθώς ποτίζονται με ένα σταθερό πρόγραμμα ανάλογα την εποχή και το σημείο της ζωής τους. Παρόλα αυτά γνωρίζουμε είδη πως ένα φυτό χεριάζετε το χώμα του να διατηρείτε σε συγκεκριμένο ποσοστό υγρασίας, αυτό επηρεάζετε ανάλογα την εποχή και το πρόγραμμα δεν μπορεί να καλύψει πλήρως αυτήν την ανάγκη. Επίσης όσο δεν είμαστε εκεί να επιβλέπουμε την ανάπτυξη ενός φυτού δεν μπορούμε να γνωρίζουμε την κατάσταση του.

Εμείς θα σχεδιάσουμε μια ολοκληρωμένη λύση η οποία με την χρήση μικρό υπηρεσιών, θα μπορεί να ποτίζει αυτόματα για εμάς όταν χεριάζετε. Έτσι μειώνουμε το κόστος του νερού ενώ το χρησιμοποιούμε πιο σωστά επιτυγχάνοντας και συνεχή παρακολούθηση.

Η λύση μας θα χρειαστεί σύνολο πέντε διαδικτυακές εφαρμογές και μια βάση δεδομένων. Μπορεί να φαίνονται πολλές αλλά η κάθε μια από μόνη της είναι πολύ «ελαφριά», δηλαδή καταναλώνει πολύ λίγους πόρους μιας και η αρχιτεκτονική μας είναι αυτή των μικρό υπηρεσιών. Κάθε μια από τις εφαρμογές μας είναι σχεδιασμένη για να κάνει μόνο ένα πράγμα, ενώ δεν κρατάει κάποιες σημαντικές πληροφορίες όσο είναι σε λειτουργία, αυτό μας δίνει την δυνατότητα αν χρειαστεί να τρέξουμε παραπάνω φορές κάποια εφαρμογή ταυτόχρονα, δηλαδή δυο διακομιστές/υπολογιστές που μπορούν να εξυπηρετήσουν πελάτες ανεξάρτητος κατάστασης.

Οι εφαρμογές μας είναι οι εξής:

- Εφαρμογή ταυτοποίησης
- Εφαρμογή RestAPI [1] για την διαχείριση πόρων
- Εφαρμογή SPA [2] η οποία επικοινωνεί με τη Rest και δίνει ένα περιβάλλον διαχειρίσεις στον περιηγητή (browser) μας.
- Εφαρμογή MQTT [3] στην οποία επικοινωνούν οι κόμβοι για την λήψη ρυθμίσεων και αποστολή μετρήσεων.
- Εφαρμογή «υπηρεσίας» [4] η οποία εκτελείτε ανά δεκαπέντε λεπτά για την υλοποίηση του αυτομάτου ποτίσου.

Η εφαρμογή ταυτοποίησης είναι ένας διακομιστής RestAPI μαζί με ιστοσελίδα ταυτοποίησης και διαχείρισης SPA. Αυτή η εφαρμογή αναλαμβάνει την είσοδο ενός χρήστη στο σύστημα ενώ έχει περιορισμένες δυνατότητες όπως την δημιουργία άλλων χρηστών, επίσης δημιουργεί μερικά αντικείμενα «token» τα οποία λαμβάνει η εφαρμογή που ζήτησε να τακτοποιηθεί ένας χρήστης, αυτό το «token» το στέλνει σε οποιαδήποτε άλλη εφαρμογή για να της επιτραπεί η σύνδεση.

Η εφαρμογή RestAPI θέλουμε να δίνει ένα ευρέος φάσμα εντολών και αναζητήσεων. Στην δική μας περίπτωση δίνει την δυνατότητα σύνδεσης μέσω RestAPI ή SignalR [5] η οποία είναι μια τεχνολογία RPC [6] σε αντίθεση με την RestAPI μας δίνει την δυνατότητα μιας συνεχής σύνδεσης με το διακομιστή, έτσι μπορεί ο διακομιστής να καλέσει μεθόδους απομακρυσμένα στον περιηγητή. Εμείς κάνουμε χρήση διάφορων ειδοποιήσεων στην μορφή ενημέρωσης αφαίρεσης η και δημιουργίας αντικειμένων. Σημαντικό είναι ότι αν κάποιος χρησιμοποιεί την RestAPI επαφή του διακομιστή ένας άλλος χρήστης που χρησιμοποιεί την SignalR επαφή θα λάβει και πάλι ειδοποιήσεις για τις αλλαγές. Γενικά θα δημιουργήσουμε το σύστημα έτσι ώστε οι αλλαγές της βάσης δεδομένων μας θα μπορούν να σταλθούν και στον περιηγητή οπότε δεν θα έχει σημασία αν τρέχουμε παραπάνω διακομιστές της εφαρμογής RestAPI.

Παρακάτω βλέπουμε τις διάφορες διεπαφές της εφαρμογής RestAPI

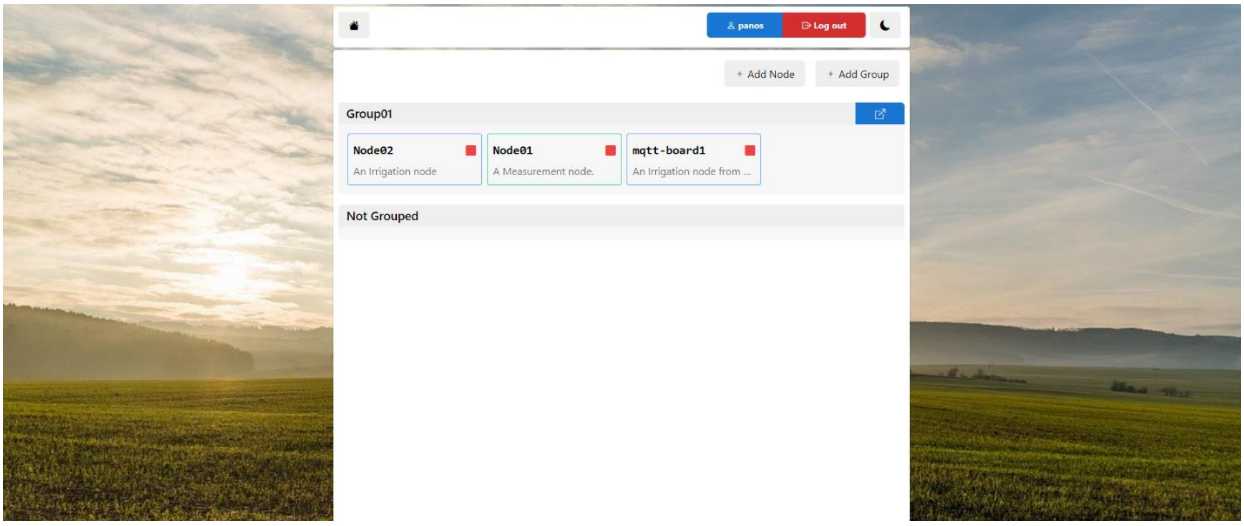
Παράμετρος		Περιγραφή
RestAPI	SignalR	
GET /v1/node	node:search	Αναζήτηση κόμβων
GET /v1/node/{id}	node:get	Ανάκτηση συγκεκριμένου κόμβου
POST /v1/node	node:create	Δημιουργία κόμβου
PATCH /v1/node	node:update	Ενημέρωση κόμβου
DELETE /v1/node/{id}	node:delete	Διαγραφή κόμβου
PATCH /v1/node/add	node:group-add	Προσθήκη κόμβου σε γκρουπ
PATCH /v1/node/remove	node:group-remove	Διαγραφή κόμβου από γκρουπ
PATCH /v1/node/refresh	node:refresh	Ανανέωση του “token” ενός κόμβου
	node:added	Ειδοποίηση ότι ένας κόμβος δημιουργήθηκε.
	node:updated	Ειδοποίηση ότι ένας κόμβος ενημερώθηκε
	node:removed	Ειδοποίηση διαγραφής ενός κόμβου
	node:connection	Ειδοποίηση αλλαγής σε κατάσταση σύνδεσης κόμβου
GET /v1/group	group:search	Αναζήτηση γκρουπ
GET /v1/group/{id}	group:get	Ανάκτηση συγκεκριμένου γκρουπ

POST /v1/group	group:create	Δημιουργία γκρουπ
PATCH /v1/group/	group:update	Ενημέρωση γκρουπ
DELETE /v1/group/{id}	group:delete	Διαγραφή γκρουπ
	group:added	Ενημέρωση δημιουργίας γκρουπ
	group:updated	Ενημέρωση αλλαγής γκρουπ
	group:removed	Ενημέρωση διαγραφής γκρουπ
GET /v1/irrigation	irrigation:search	Αναζήτηση εντολών ποτίσματος
GET /v1/irrigation/many	irrigation:search-many	Πολλαπλές αναζητήσεις εντολών
GET /v1/measurement	measurement:search	Αναζήτηση μετρήσεων
GET /v1/measurement/many	measurement:search-many	Πολλαπλές αναζητήσεις μετρήσεων

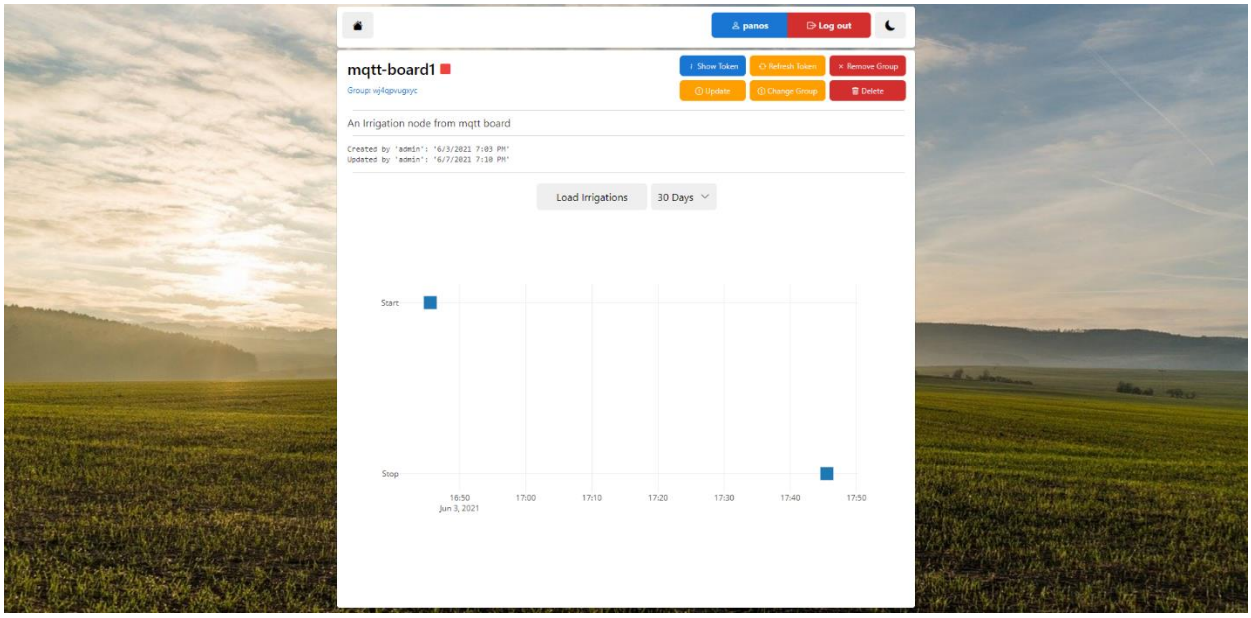
Οι παραπάνω διεπαφές του πίνακα περιέχουν και αντικείμενα αποστολής και λήψης τα οποία δεν φαίνονται στον πίνακα, αυτά τα αντικείμενα θα τα αποφασίσουμε αργότερα όταν έρθει η ώρα της υλοποίησης της εφαρμογής, για την ώρα ξέρουμε ότι θέλουμε να υλοποιήσουμε αυτά τα σημεία, έτσι μπορούμε να αρχίσουμε να σχεδιάζουμε και τις επόμενες εφαρμογές βασισμένες σε αυτήν.

Η εφαρμογή SPA η οποία αναλαμβάνει την παρουσίαση και την διαχείριση κάνοντας χρήση της εφαρμογής RestAPI θα σχεδιαστεί χρησιμοποιώντας τον πίνακα πιο πάνω.

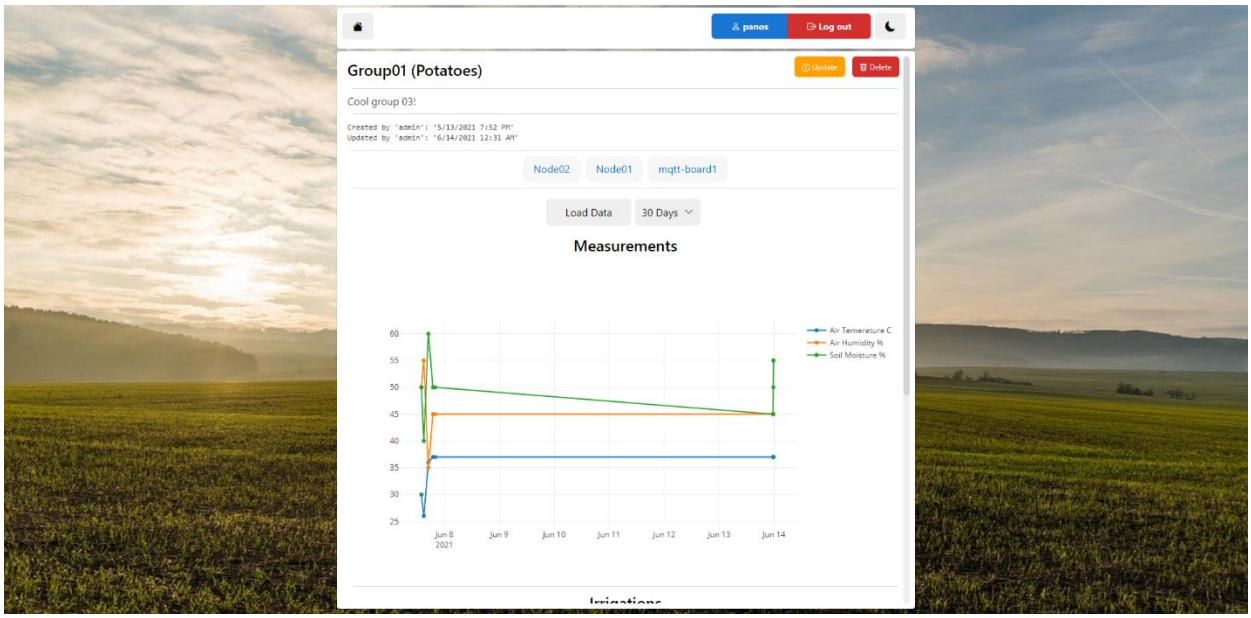
Παρακάτω βλέπουμε μερικές φωτογραφίες από το πως θα καταλήξει η εφαρμογή.



Εικόνα 1: Αρχική σελίδα SPA



Εικόνα 2: Σελίδα γκρουπ SPA



Εικόνα 3: Σελίδα κόμβου SPA

Η εφαρμογή MQTT αναλαμβάνει την πλήρη επικοινωνία και διαχείριση των κόμβων. Το API [7] που θα δημιουργήσουμε φαίνεται στον παρακάτω πίνακα. Όπου βλέπουμε “nodeId” είναι η μοναδική ταυτότητα ενός κόμβου που δημιουργείτε μαζί με τον κόμβο και δεν αλλάζει ποτέ.

MQTT Topic	Περιγραφή
node/connection/{nodeId}	Σημείο δημοσίευσης κατάστασης σύνδεσης κόμβων.

node/settings/{nodeId}	Σημείο δημοσίευσης ρυθμίσεων κόμβων.
node/irrigation/{nodeId}	Σημείο δημοσίευσης εντολών ποτίσματος.
node/measurement/{nodeId}	Σημείο δημοσίευσης μετρήσεων.

Για το παραπάνω API θέλουμε μόνο ο διακομιστής να μπορεί να δημοσιεύσει σε όλα τα "Topic" (Σημεία) εκτός από αυτό των μετρήσεων.

Επίσης θέλουμε ο κάθε κόμβος να τακτοποιείτε με ένα μοναδικό «Token» η αλλιώς έναν κωδικό για τον συγκεκριμένο κόμβο. Αν δεν είναι σωστός ο κωδικός τότε ο κόμβος δεν συνδέεται με τον διακομιστή και δεν μπορεί να λάβει ειδοποιήσεις η να στείλει νέα μηνύματα.

Τέλος θέλουμε αν η κατάσταση του «Token» ενός κόμβου αλλάξει, δηλαδή αλλάξει ο κωδικός του τότε να αποσυνδέσουμε τον συγκεκριμένο συνδεδεμένο κόμβο, αν δεν είναι συνδεδεμένος δεν μας ενδιαφέρει.

Η εφαρμογή υπηρεσίας έχει μόνο έναν σκοπό, να δημιουργεί καινούργιες εντολές για την εκκίνηση και την παύση ποτίσματος και να είναι ανεξάρτητη από το υπόλοιπο σύστημα. Στην δική μας περίπτωση θα χρησιμοποιείτε σαν εργαλείο που επικοινωνεί απευθείας με την βάση και θα εκτελεί όλες τις απαραίτητες ενέργειες για να δημιουργήσει εντολές για κάθε γκρουπ κόμβων.

Τέλος οι κόμβοι θα χρειαστούν την δική τους εφαρμογή. Εμείς επιλέξαμε να δημιουργήσουμε μια εφαρμογή .NET Framework την οποία εκτελούμε σε ένα Raspberry Pi Zero W κάνοντας χρήση του Mono [8]. Θέλουμε να δημιουργήσουμε τη εφαρμογή με τέτοιο τρόπο ώστε να ρυθμίζονται αρκετά πράγματα αυτό για να μπορούμε να χρησιμοποιούσε για παράδειγμα ένα Pi για δυο εφαρμογές η και μια εντελώς διαφορετική πλακέτα. Όσο η πλακέτα που επιλέξαμε χρησιμοποιεί λογισμικό Linux θα δούμε ότι θα μπορεί να εκτελέσει την εφαρμογή μας, αλλιώς κάνοντας χρήση του παραπάνω MQTT API μπορούμε να δημιουργήσουμε τέλειος διαφορετικούς κόμβους από άποψη τόσο πλακέτας όσο και γλώσσας προγραμματισμού. Εμείς επιλέξαμε την ιδιά γλώσσα και framework λόγο διευκόλυνσης του διαμοιρασμού κώδικα όπως θα δούμε στα επόμενα κεφάλαια.

Τέλος η διασύνδεση των κόμβων θα μπορούσε να γίνει με οποιονδήποτε τρόπο ανάλογα την τοπογραφία του κάθε μέρους, για παράδειγμα αν υπάρχει είδη κάλυψη WI-FI μπορούμε να συνδεθούμε απευθείας εκεί, αλλιώς μπορούμε να δοκιμάσουμε κάποιο MESH [9] δίκτυο η κάποιο δίκτυο μεγάλης εμβέλειας όπως το LoRa [10]. Η επιλογή δικτύωσης είναι καθαρό πρόβλημα κάθε εγκατάστασης και όσο προσφέρει την δυνατότητα σύνδεσης στο διαδίκτυο δεν μας επηρεάζει μάλιστα η τεχνολογία MQTT μπορεί να ανταπεξέλθει σε πολύ αδύναμα δίκτυα.

Στα παρακάτω κεφάλαια θα δούμε την επεξήγηση των επίλογων μας και την υλοποίησης του προτζεκτ, ενδεικτικά θα αναφέρονται οι λέξεις «πελάτης» η οποία υποδεικνύει μια εφαρμογή/συσκευή η οποία συνδέετε σε έναν διακομιστή.

2: ΑΡΧΙΤΕΚΤΟΝΙΚΗ

Πριν διαλέξουμε την αρχιτεκτονική μας θα πρέπει να καταλάβουμε και τι εννοούμε με τον όρο αυτό. Αρχιτεκτονική είναι ο τρόπος με τον οποίο η υπηρεσία μας λειτουργεί, πιο συγκεκριμένα καθορίζει πολλά πράγματα για το πως θα κάνουμε host μια υπηρεσία για το πως θα την αναπτύξουμε και τέλος για το πως θα «μεγαλώσουμε» ανάλογα με τις ανάγκες που έχουμε αυτό περιλαμβάνει και τις γλώσσες προγραμματισμού αλλά και το framework το οποίο θα χρησιμοποιήσουμε.

Η αρχιτεκτονική σε όλες τις διαδικτυακές εφαρμογές μπορεί πια να χωριστεί σε δύο μεγάλες κατηγορίες μονολιθικές (monoliths) [11] και σε μικρό υπηρεσίες (micro services) [12].

2.1: Γλώσσα Προγραμματισμού – Framework

Για γλώσσα προγραμματισμού επέλεξα να χρησιμοποιήσω την C# [13] της Microsoft και το .NET Core [14] framework πιο συγκεκριμένα τις τεχνολογίες ASP.NET Core [15], SignalR [5], MQTTnet [16] Blazor [17] Mono [8] και Openiddict [18].

Παρόλο που θα χρησιμοποιήσουμε μια μόνο γλωσσά για τον προγραμματισμό και τον διαμοιρασμό κώδικα θα δούμε ότι κάθε framework έχει τον δικό του σκοπό και συμπεριφορά.

Ο μεγαλύτερος λόγος είναι η καλή γνώση γλώσσας και framework, ενώ και με τα δύο να είναι πια open source [19] και να έχουν πολλές σπουδαίες εφαρμογές και υποστήριξη θα δούμε πιο κάτω ότι ήταν μια εύκολη επιλογή.

2.2: Μονολιθικές Εφαρμογές – Μικρό Υπηρεσίες

Όπως δηλώνει και το όνομα μονολιθικές εφαρμογές είναι αυτές που αναπτύσσονται σαν ένας μεγάλος βράχος, όλες οι λειτουργίες και οι δυνατότητες τρέχουν σε μια μονάδα επεξεργασίας (process) και οι και αναπτύσσονται ταυτόχρονα, μάλιστα αν θέλουμε να κάνουμε ενημέρωση σε μόνο μια λειτουργία και πάλι θα χρειαστεί να ενημερώσουμε ολόκληρη την εφαρμογή.

Για παράδειγμα ας σκεπτούμε ότι έχουμε ένα e-shop το οποίο πρέπει να χειρίζεται τον κατάλογο του καταστήματος αλλά ταυτόχρονα το καλάθι αγορών και τις ίδιες τις πληρωμές. Με μια μονολιθική εφαρμογή τρέχουμε και τις 3 αυτές υπηρεσίες ταυτόχρονα. Αυτό σημαίνει παρόλο που οι πιο πόλοι πελάτες βλέπουν τον κατάλογο και λίγοι κάνουν μια παραγγελία όλοι χρησιμοποιούν τον ίδιο διακομιστή και στην ουσία τον ίδιο υπολογιστή. Αυτό μπορεί να έχει διαφορά αποτελέσματα, ένα είναι η λειτουργία για το καλάθι να είναι πιο αργή όταν πολύς κόσμος κοιτά τον κατάλογο μια άλλη είναι ότι η λειτουργία των πληρωμών μπορεί να αλλάζει συχνά αλλά εμείς θα πρέπει να κάνουμε ενημέρωση σε ολόκληρη την εφαρμογή.

Οι μικρό υπηρεσίες είναι πιο περίπλοκες διότι όπως φαίνεται από το όνομα είναι πολλές μικρές υπηρεσίες. Δεν χρειαζόμαστε όλες οι λειτουργίες να είναι σε μια μεγάλη εφαρμογή, αντίθετος διαλέγουμε κάθε λειτουργία και την σπάμε στην δική της μικρή εφαρμογή, μάλιστα είναι αρκετά σύνηθες η κάθε μικρή εφαρμογή να είναι τελείως απομονωμένη και να έχει και την δική της βάση δεδομένων. Αυτό προσφέρει το πολύ μεγάλο πλεονέκτημα ότι κάθε λειτουργία αναπτύσσεται μόνη της και βέβαια γίνετε host μόνη της. Το μειονέκτημα εδώ ότι ο τρόπος που κάνουμε host μια εφαρμογή χτισμένη σε μικρό υπηρεσίες γίνεται λίγο πιο περιπλοκή.

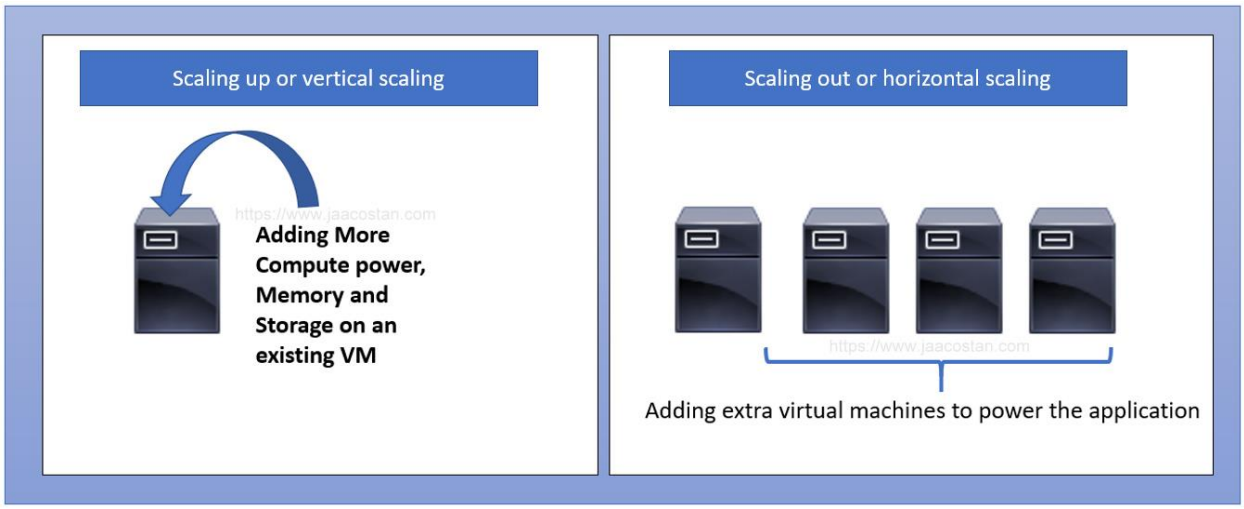
Για παράδειγμα το e-shop που αναφέραμε προηγούμενος ότι χεριάζετε τρεις μικρό υπηρεσίες, μια για τον κατάλογο του καταστήματος, μια για το καλάθι και μια για τις πληρωμές. Αυτό κάνει πολύ πιο απλή την ανάπτυξη των λειτουργιών αφού κάθε λειτουργία είναι μια υπηρεσία μόνη της. Όταν μια υπηρεσία έχει μεγάλο φόρτο δεν επηρεάζει τις άλλες δυο αφού συνήθως δεν τις τρέχουν ούτε στον ίδιο υπολογιστή και βέβαια όταν έρχεται η ώρα να κάνουμε ενημέρωση μια από τις υπηρεσίες οι άλλες δεν επηρεάζονται.

Άλλη μια μεγάλη διαφορά των μικρό υπηρεσιών και των μονολιθικών εφαρμογών είναι στον τρόπο που τις «μεγαλώνουμε» δηλαδή τι κάνουμε όταν χεριάζετε να τρέξουμε την εφαρμογή μας και σε δεύτερο υπολογιστή για να μπορέσει να εξυπηρέτηση την ζήτηση.

Χρησιμοποιώντας το παράδειγμα από θα υποθέσουμε ότι υπάρχει μεγάλος φόρτος στην λειτουργία του καλαθιού. Με την μονολιθική μας εφαρμογή θα πρέπει να τρέξουμε σε έναν δεύτερο υπολογιστή ολόκληρη την εφαρμογή δηλαδή θα λειτουργεί και ο κατάλογος και οι πληρωμές παρόλο που δεν τις χρειαζόμαστε ή το πιο σύνηθες θα πρέπει να δώσουμε παραπάνω πόρους στον υπολογιστή το οποίο ανάλογα και άλλους παράγοντες μπορεί να είναι δύσκολο. Αυτό μπορεί να κάνει ολόκληρη την εφαρμογή πιο αργή ενώ σιγουρά καταναλώνει και πιο πόλους πόρους από όσους χρειαζόμαστε.

Με την εφαρμογή μικρό υπηρεσιών όταν η λειτουργία του κάλαθου έχει μεγάλο φόρτο στον δεύτερο υπολογιστή θα τρέξουμε μόνο την λειτουργία του καλαθιού αυτό έχει ως αποτέλεσμα να χρησιμοποιούμε λιγότερους πόρους αφού τρέχουμε μόνο την λειτουργία που χρειαζόμαστε.

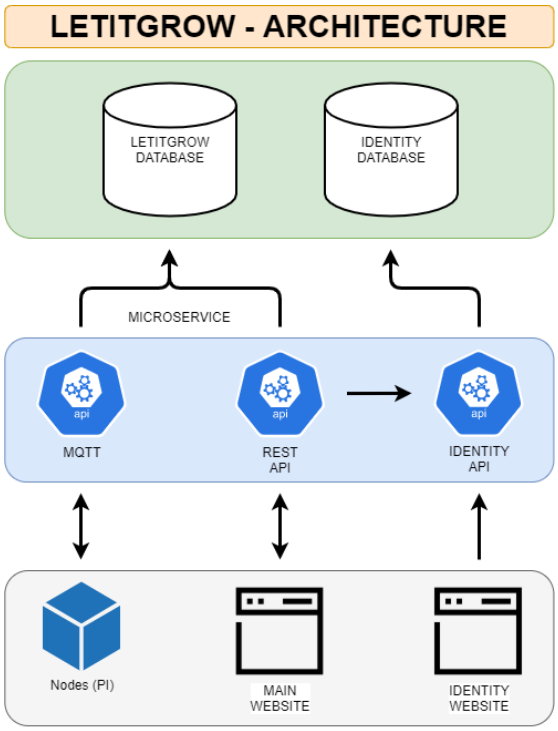
Ο τρόπος που λειτουργεί μια μονολιθική εφαρμογή και πώς την «μεγαλώνουμε» συνήθως ονομάζεται vertical scaling [20] που σημαίνει ότι δίνουμε και άλλους πόρους στην ιδιά μηχανή πριν αποφασίσουμε να κάνουμε horizontal scaling [21] που σημαίνει ότι χρησιμοποιούμε πιο πολλές μηχανές για τους λογούς που είπαμε πιο πριν Με τις μικρό υπηρεσίες κάνουμε πάντα horizontal scaling [21] αφού είναι πιο φτηνό και εύκολο.



Εικόνα 4: Vertical vs Horizontal scaling

2.3: Υπηρεσίες

Θα δημιουργήσουμε μια εφαρμογή μικρό υπηρεσιών. Οι υπηρεσίες που θα δημιουργήσουμε εμείς φαίνονται στο παρακάτω σχήμα. Αποφασίσαμε να χρησιμοποιήσουμε μικρό υπηρεσίες διότι από νωρίς γνωρίζουμε ότι ο φόρτος δεν θα είναι ομοιόμορφος και ότι μπορεί στο μέλλον να χρειαστεί να κάνουμε horizontal scaling [21].



Εικόνα 5: Αρχιτεκτονική

Όπως βλέπουμε οι μικρό υπηρεσίες μας περιλαμβάνουν την βάση δεδομένων την οποία θα δούμε πιο μετά, μια λειτουργία που χρησιμοποιεί την τεχνολογία MQTT [3] μια λειτουργία HTTP REST [22] και μια υπηρεσία ταυτοποίησης.

2.3.1: MQTT

Mqtt [3] είναι μια τεχνολογία/πρωτόκολλο pub/sub (publish subscribe) η οποία επιτρέπει σε οποιονδήποτε να δημοσιεύει μηνύματα σε διάφορα θέματα αλλά και να κάνει εγγραφή σε αυτά ώστε να λαμβάνει τα μηνύματα τα οποία δημοσιευτήκαν σε ένα θέμα.

Χρησιμοποιούνται διάφορες τεχνολογίες όπως Web Sockets [23] ή TCP/IP [24] για να γίνει η σύνδεση τους με έναν Broker (διακομιστή Mqtt) και έχει την δυνατότητα να μην χάνει την σύνδεση του ακόμα και αν το δίκτυο είναι αναξιόπιστο ενώ δίνει παρα πολλές επιλογές για τον διαμοιρασμό μηνυμάτων από έναν κόμβο στο διακομιστή και το αντίθετο.

Για την διεκπεραίωση τόσο του διακομιστή αλλά και του κόμβου θα χρησιμοποιήσουμε την βιβλιοθήκη κώδικα MQTTnet [16] η οποία είναι βιβλιοθήκη ανοιχτού κώδικα [19].

2.3.2: HTTP REST

HTTP ή REST [22] είναι μια αρκετά συχνή τεχνολογία που χρησιμοποιείτε για την επικοινωνία διαδικτυακών εφαρμογών. Το μοντέλο είναι χτισμένο στην αποστολή ενός μηνύματος η μίας αίτησης καλύτερα η οποία μπορεί να κατηγοριοποιηθεί σε «GET, POST, PUT, DELETE και άλλα» και επιστρέφει μια απάντηση. Η απάντηση έχει διάφορους κωδικούς πριν δούμε το μήνυμα ώστε να γνωρίζουμε αν η αποστολή έγινε με επιτυχία η αν υπήρξε κάποιο σφάλμα.

Τα δεδομένα στέλνονται και λαμβάνονται στην μορφή JSON [25] αλλά αυτό είναι στην επιλογή του προγραμματιστή.

2.3.3: SignalR

SignalR είναι μια βιβλιοθήκη για προγραμματιστές ASP.NET Core, για την προσθήκη λειτουργιών ιστού σε πραγματικό χρόνο. Η λειτουργικότητα διαδικτύου σε πραγματικό χρόνο είναι η δυνατότητα προώθησης περιεχομένου κώδικα από διακομιστή στους συνδεδεμένους πελάτες, όπως συμβαίνει, σε πραγματικό χρόνο.

Εκμεταλλεύεται πολλές τεχνολογίας μεταφοράς μηνυμάτων, επιλέγοντας αυτόματα τη καλύτερη διαθέσιμη δεδομένων των δυνατοτήτων του πελάτη και του διακομιστή.

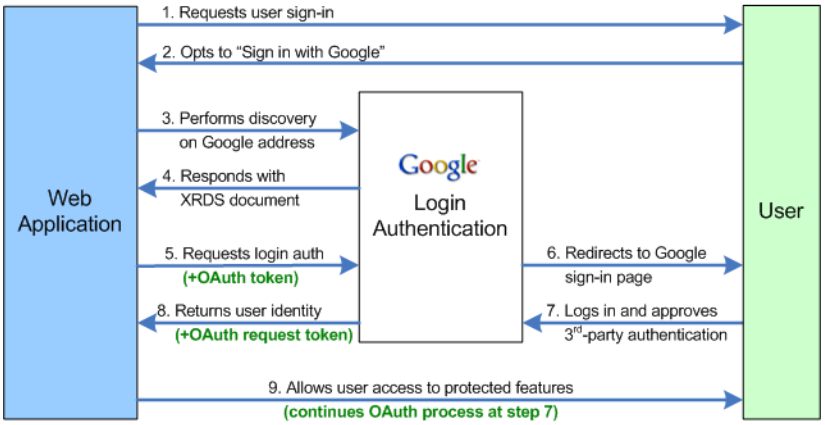
Εκμεταλλεύεται την τεχνολογία WebSocket [23] και HTML5 API [26] που επιτρέπει αμφίδρομη επικοινωνία μεταξύ του προγράμματος περιήγησης και του διακομιστή. Όταν δεν είναι διαθέσιμη η τεχνολογία WebSocket τότε θα χρησιμοποιεί άλλες τεχνικές και τεχνολογίες όπως Server Sent Events, γνωστό και ως EventSource ή Ajax long polling [27], ενώ ο κωδικός εφαρμογής παραμένει ο ίδιος.

Στην ουσία παρέχει ένα απλό, υψηλού επιπέδου API για την εκτέλεση RPC [6] διακομιστή-προς-πελάτη και το αντίθετο (καλέστε λειτουργίες JavaScript στο πρόγραμμα περιήγησης από την πλευρά διακομιστή) σε μια εφαρμογή ASP.NET Core, καθώς και προσθέτοντας χρήσιμα άγκιστρα για τη

διαχείριση σύνδεσης , όπως συμβάντα σύνδεσης / αποσύνδεσης, συνδέσεις ομαδοποίησης, εξουσιοδότηση.

2.3.4: Ταυτοποίηση

Για την ταυτοποίηση ενός χρήστη στο διαδίκτυο γίνεται χρήση διαφόρων πρωτοκόλλων τα οποία συνεργάζονται μεταξύ τους για να γίνει η επιτυχής σύνδεση ενός χρήστη. Αυτά τα πρωτόκολλα είναι τα OAuth [28] και OpenID [29]. Στην παρακάτω εικόνα φαίνεται ένα σχεδιάγραμμά ταυτοποίησης με OpenID.



Εικόνα 6: OpenID Workflow

Εμείς χρησιμοποιώντας την βιβλιοθήκη Openiddict [18] θα δημιουργήσουμε τον δικό μας διακομιστή ταυτοποίησης χρηστών ενώ ταυτόχρονα θα δημιουργήσουμε και τον διακομιστή ο οποίος θα αποθηκεύει και θα διαχειρίζεται τους χρήστες μας. Παρόλα αυτά επειδή η εφαρμογή μας και οι διακομιστές μας κάνουν χρήση του πρωτοκόλλου OpenID έχουν την δυνατότητα να χρησιμοποιήσουν και διαφορετικές εφαρμογές ταυτοποίησης με λίγες αλλαγές.

2.4: Ανάπτυξη Διαδικτυακών Εφαρμογών

Ο όρος διαδικτυακές εφαρμογές είναι αρκετά ευρείς και υπονοεί όλες τις εφαρμογές οι οποίες τρέχουν σε κάποιον διακομιστή και εξυπηρετούν τους πελάτες.

Παρόλα αυτά υπάρχουν διάφορες τεχνολογίες για να δημιουργήσεις μια τέτοια εφαρμογή. Στην δική μας περίπτωση χρησιμοποιώντας την τεχνολογία ASP.NET Core και κάποιες πρόσθετες βιβλιοθήκες θα δημιουργήσομε όλες τις απαραίτητες εφαρμογές μας.

Οι εφαρμογές χτισμένες με ASP.NET Core κάνουν χρήση μιας τεχνικής Dependency Injection [30] αυτό σημαίνει ότι το ίδιο το framework θα μας δίνει αντικείμενα που χρειαζόμαστε όταν εκτελούμε κώδικα, αυτό παρατηρείτε πιο κάτω σε ορισμένους κώδικες.

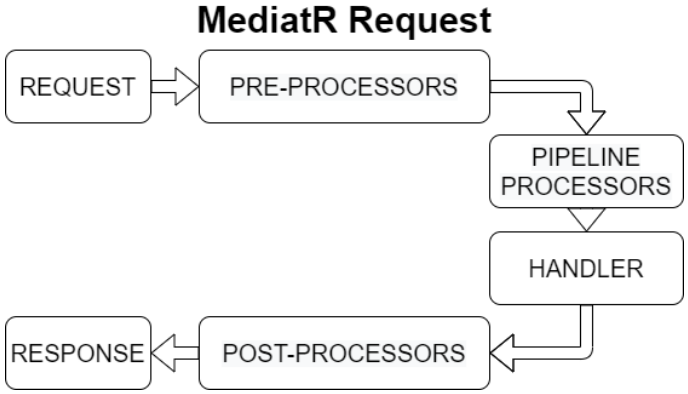
2.4.1: CQRS

Πιο συγκεκριμένα θα χρησιμοποιήσουμε την τεχνική CQRS [31] μέσω της βιβλιοθήκης MediatR [32]. CQRS είναι ακρωνύμιο για «Command and Query Responsibility Segregation» το οποίο στα ελληνικά σημαίνει «Διαχωρισμός ευθύνης εντολών και ερωτήματος». Εντολές είναι ο κώδικας ο οποίος θα έχει επιπτώσεις όπως για παράδειγμα να αλλάξουμε το όνομα ενός προϊόντος ενώ ερωτήματα είναι ο κώδικας ο οποίος απλά επιστρέφει ένα αποτέλεσα όπως βρες μου όλα τα προϊόντα κάτω τον 20€.

Αυτός ο διαχωρισμός είναι ευθηνή του προγραμματιστή να γίνει και δεν επηρεάζει κάποιον άλλον στο τέλος το αποτέλεσα για όλου τους πελάτες της εφαρμογής είναι ότι εκτελέσανε μία εντολή http ή rpc ή Mqtt ή οποιαδήποτε άλλο μέσο χρησιμοποιούμε για την επικοινωνία αυτών των εντολών στον διακομιστή. Για εμάς όμως θα μας δώσει την ευκολία να διαχωρίσουμε των κώδικα μας και να δούμε πώς μπορούμε να τον εμπλουτίσουμε μέσο αυτής της τεχνικής.

Οι αιτήσεις θα για να εκτελεστούν θα πρέπει να έχουν και μια αντίστοιχη μέθοδο που θα την εκτέλεσης εκεί είναι που η βιβλιοθήκη MediatR έρχεται να μας διευκολύνει. Μέσω της βιβλιοθήκης MediatR θα δημιουργήσουμε διάφορες εντολές οι οποίες θα πρέπει να εφαρμόζουν μια διεπαφή [33] IRequest<T>. Μία διεπαφή στην γλώσσα προγραμματισμού C# μας επιτρέπει να δηλώσουμε τον σκοπό που θα θέλαμε να εκτελεί ο κώδικας μας, ένα συμβόλαιο χωρίς όμως να γνωρίζουμε πιο αντικείμενο στα αλήθεια εκτελεί αυτές τις εντολές ενώ «T» [34] είναι ένας γενικός όρος στην προκειμένη περίπτωση δηλώνει τι αντικείμενο θα επιστρέψουμε. Έτσι στον κώδικα μας έχουμε για παράδειγμα το αντικείμενο «GetPerson : IRequest<Person>» αυτό δηλώνει στην βιβλιοθήκη ότι όταν θα έρχεται μία αίτηση «GetPerson» θα λαμβάνουμε μια απάντηση «Person» στις ενότητες των εφαρμογών θα δούμε πως αυτό γίνεται στον κώδικα. Να σημειωθεί ότι μας επιτρέπετε μόνο ένας κώδικας διαχείρισης για κάθε μία αίτηση.

Μαζί με το CQRS μοντέλο η βιβλιοθήκη μας δίνει διάφορους τρόπους επεξεργασίας μιας αίτησης, μας δίνει τρόπο να εκτελέσουμε κώδικα πριν εκτελεστεί ο κανονικός κώδικας της αίτησής, μας δίνει την δυνατότητα να τρέξουμε κώδικα σαν μια αλληλουχία μεθόδων με το ίδιο αποτέλεσμα και μας δίνει την δυνατότητα να τρέξουμε και κώδικα μετά το πέρας της εκτέλεσης πριν επιστραφεί η απάντηση μας όπως φέρνετε στο παρακάτω σχεδιάγραμμά.



Εικόνα 7: Εκτέλεση αίτησης μέσω της βιβλιοθήκης MediatR

Σε όλες μας τι εφαρμογές χρησιμοποιήσαμε τους παρακάτω κώδικες έτσι ώστε να πέτυχουμε πιο ομαλή και εύκολη εκτέλεση μίας αίτησης.

Ο παρακάτω κώδικας στην είναι υπεύθυνος για την καταγραφή κάθε μιας αίτησής, χρησιμοποιώντας την γλώσσα C# αναλυτικά έχουμε ένα αντικείμενο το οποίο όταν δημιουργείτε του δίνεται ένα άλλο αντικείμενο το οποίο έχει την δυνατότητα να κάνει καταγραφές. Στην συνέχεια στην εντολή Process η οποία θα εκτελείτε για κάθε αίτηση, θα κάνουμε μια απλή καταγραφή της κάθε αίτησής και επειδή χρησιμοποιούμε την τελευταία έκδοση της γλωσσάς (9.0) όταν καλείτε η εντολή LogInformtion αυτόματα καταγραφεί όλες τους της μεταβλητές σε μια μορφή που μπορεί να διαβάσει ένας άνθρωπος.

```
1 reference
public class RequestLogger<TRequest> : IRequestPreProcessor<TRequest> where TRequest : notnull
{
    private readonly ILogger<TRequest> logger;

    0 references
    public RequestLogger(ILogger<TRequest> logger)
    {
        this.logger = logger;
    }

    0 references
    public Task Process(TRequest request, CancellationToken cancellationToken)
    {
        logger.LogInformation("Request {Request}", request);

        return Task.CompletedTask;
    }
}
```

Εικόνα 8: Καταγραφή της στιγμής που ήρθε μια αίτηση και του περιεχόμενου της.

```
2021-06-16 12:14:29.640 +03:00 [INF] Request SearchGroups { }
```

Εικόνα 9: Παράδειγμα καταγραφής αίτησης με όνομα SearchGroups.

Στην παρακάτω εικόνα βλέπουμε κώδικα υπεύθυνο για έλεγχο κάθε μιας αίτησης με την βιβλιοθήκη FluentValidation [35] πριν καν εκτελέσουμε τον κώδικα μας. Αυτή την φόρα ζητάμε από την βιβλιοθήκη να μας δώσει μια λίστα από αντικείμενα τα οποία μπορούν να κάνουν τον έλεγχο μιας αίτησής. Τα αντικείμενα αυτά εφαρμόζουν την διεπαφή IValidator<TRequest> where IRequest<TResponse> οπότε μπορεί να μην ξέρουμε το ακριβές αντικείμενο αλλά ξέρουμε ότι θα έχει μια μέθοδο για να κάνει τον έλεγχο της αίτησής.

Αφού μας δοθεί αυτή η λίστα ελέγχουμε αν έχει οποιοδήποτε αντικείμενο και αν έχει χρησιμοποιώντας το χαρακτηριστικό LINQ [36] της γλώσσας C# εκτελούμε κάθε μέθοδο ελέγχου, μαζεύουμε όλα τα αποτελέσματα τα οποία υπάρχουν και τα μετατρέπουμε σε μια λίστα.

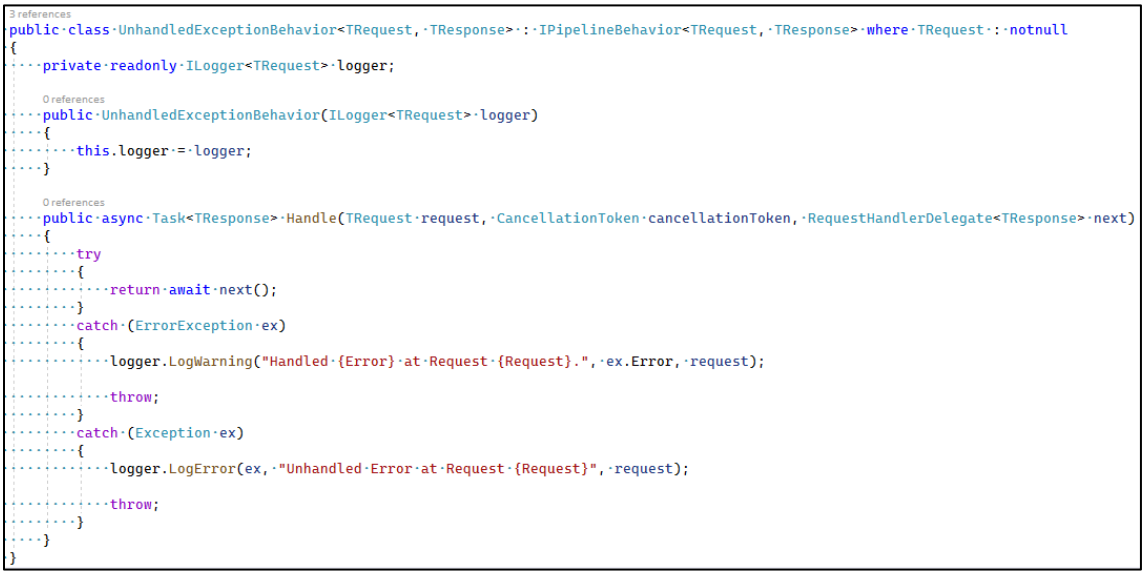
Στην συνέχεια κοιτάμε αν υπάρχουν σφάλματα σε αυτήν τη λίστα και αν ναι δημιουργούμε ένα συνολικό σφάλμα ελέγχου.

```
3 references
public class RequestValidationBehavior<TRequest, TResponse> : IPipelineBehavior<TRequest, TResponse>
    where TRequest : IRequest<TResponse>
{
    private readonly IEnumerable<IValidator<TRequest>> validators;

    0 references
    public RequestValidationBehavior(IEnumerable<IValidator<TRequest>> validators)
    {
        this.validators = validators;
    }

    0 references
    public Task<TResponse> Handle(TRequest request, CancellationToken cancellationToken, RequestHandlerDelegate<TResponse> next)
    {
        if (validators.Any())
        {
            var context = new ValidationContext<TRequest>(request);

            var failures = validators
                .Select(v => v.Validate(context))
                .SelectMany(result => result.Errors)
                .Where(f => f != null)
                .ToList();

            if (failures.Count != 0)
            {
                throw new ValidationException(failures);
            }
        }

        return next();
    }
}
```

Εικόνα 10: Έλεγχος κάθε αίτησης.

Ο παρακάτω και τελευταίος κώδικας είναι υπεύθυνος για την εκτέλεση της βασικής εντολής μέσα σε ένα δίχτυ ασφαλείας.

Εδώ πάλη ζητάμε ένα αντικείμενο καταγραφής ώστε σε περίπτωση σφάλματος να το καταγράψουμε, εκτελούμε την επόμενη εντολή και αν δεν υπάρχει σφάλμα επιστρέφουμε το αποτέλεσμα αν όμως υπάρξει σφάλμα κοιτάμε τον τύπο του, ανάλογα τον τύπο καταγράφουμε ένα διαφορετικό μήνυμα διότι υπάρχουν και σφάλματα τα οποία είναι αναμενόμενα.



```
3 references
public class UnhandledExceptionBehavior<TRequest, TResponse> : IPipelineBehavior<TRequest, TResponse> where TRequest : notnull
{
    private readonly ILogger<TRequest> logger;

    0 references
    public UnhandledExceptionBehavior(ILogger<TRequest> logger)
    {
        this.logger = logger;
    }

    0 references
    public async Task<TResponse> Handle(TRequest request, CancellationToken cancellationToken, RequestHandlerDelegate<TResponse> next)
    {
        try
        {
            return await next();
        }
        catch (ErrorException ex)
        {
            logger.LogWarning("Handled {Error} at Request {Request}.", ex.Error, request);

            throw;
        }
        catch (Exception ex)
        {
            logger.LogError(ex, "Unhandled Error at Request {Request}", request);

            throw;
        }
    }
}
```

Εικόνα 11: Διαχείριση σφαλμάτων αιτήσεων.

2.4.2:Notifications

Η βιβλιοθήκη MediatR μας δίνει και άλλον έναν τρόπο να εκτελέσουμε κώδικα ο οποίος διαφέρει από αυτό που είδαμε πιο πάνω.

Μας δίνει την δυνατότητα να εκτελέσουμε ειδοποιήσεις, αυτό σημαίνει ότι ο κώδικας μας εσωτερικά μπορεί να εκτελεί διάφορες μεθόδους σαν να ήρθε μια ειδοποίηση. Όλες οι ειδοποιήσεις πρέπει να εφαρμόζουν την διεπαφή INotification.

Μπορεί να ακούγετε απλό αλλά αυτό δίνει μεγάλη ευελιξία στο πρόγραμμα μας να εκτελεί κώδικα ασύγχρονα και μας διευκολύνει στις διάφορε ανάγκες που έχουν τα προγράμματα μας.

Μία ειδοποίηση δεν επιστρέφει κάτι και στα αλήθεια δεν μας ενδιαφέρει να επιστρέψει κάτι μας ενδιαφέρει στην ουσία με κάποιον τρόπο να ενημερώσουμε ένα άλλο σημείο του προγράμματος ότι κάτι συνέβη και αυτό θα δράσει αντίστοιχα δεν χεριάζετε να γνωρίζουμε αν έγινε με επιτυχία και τα σχετικά.

Οι ειδοποιήσεις μπορούν να έχουν πολλαπλά αντικείμενα τα οποία χειρίζονται τον ερχομό μιας ειδοποίησης για παράδειγμα μια ειδοποίηση «PersonAdded» μπορεί να έχει έναν κώδικα καταγραφής και έναν κώδικα της κανονικής εκτελέσεις, και όσους άλλους θέλουμε.

Ο παρακάτω κώδικας είναι ο κοινός κώδικας ο οποίος χειρίζεται όλες τις ειδοποιήσεις και απλά τις καταγράφει. Όπως και πριν δέχεται ένα αντικείμενο καταγραφής, καταγράφει την ειδοποίηση και τερματίζει.

```
1 reference
public class NotificationLogger<TNotification> : INotificationHandler<TNotification> where TNotification : INotification
{
    private readonly ILogger<TNotification> logger;

    0 references
    public NotificationLogger(ILogger<TNotification> logger)
    {
        this.logger = logger;
    }

    0 references
    public Task Handle(TNotification notification, CancellationToken cancellationToken)
    {
        logger.LogInformation("Notification {notification}", notification);

        return Task.CompletedTask;
    }
}
```

Εικόνα 12: Κώδικας καταγραφής ειδοποιήσεων.

2.4.3:Vertical Slicing

Vertical Slice Architecture [37], είναι ένα είδος αρχιτεκτονικής που μπορεί να εφαρμοστεί σε διάφορες γλώσσες προγραμματισμού. Το πρόβλημα όταν έχουμε πολλαπλά προτζεκτ η ακόμα και πολύ μεγάλα προτζεκτ είναι ότι όσο δημιουργούμε τις λύσεις μας θα χρειαστεί να αλλάξουμε πολλά διαφορετικά μέρη στον κώδικα μας.

Συνήθως προγράμματα και εφαρμογές δημιουργούνται από πολλαπλά προτζεκτ τα οποία γίνονται σαν μια στοίβα η διάφορα επίπεδα. Παρόλο που θα δούμε ότι ακόμα έχουμε μερικά επίπεδα αυτά πιο

πολύ δημιουργούνται για να εξυπηρετήσουν τον σκοπό της επαναχρησιμοποίησης κώδικα η της εύκολης αντικαταστατή.

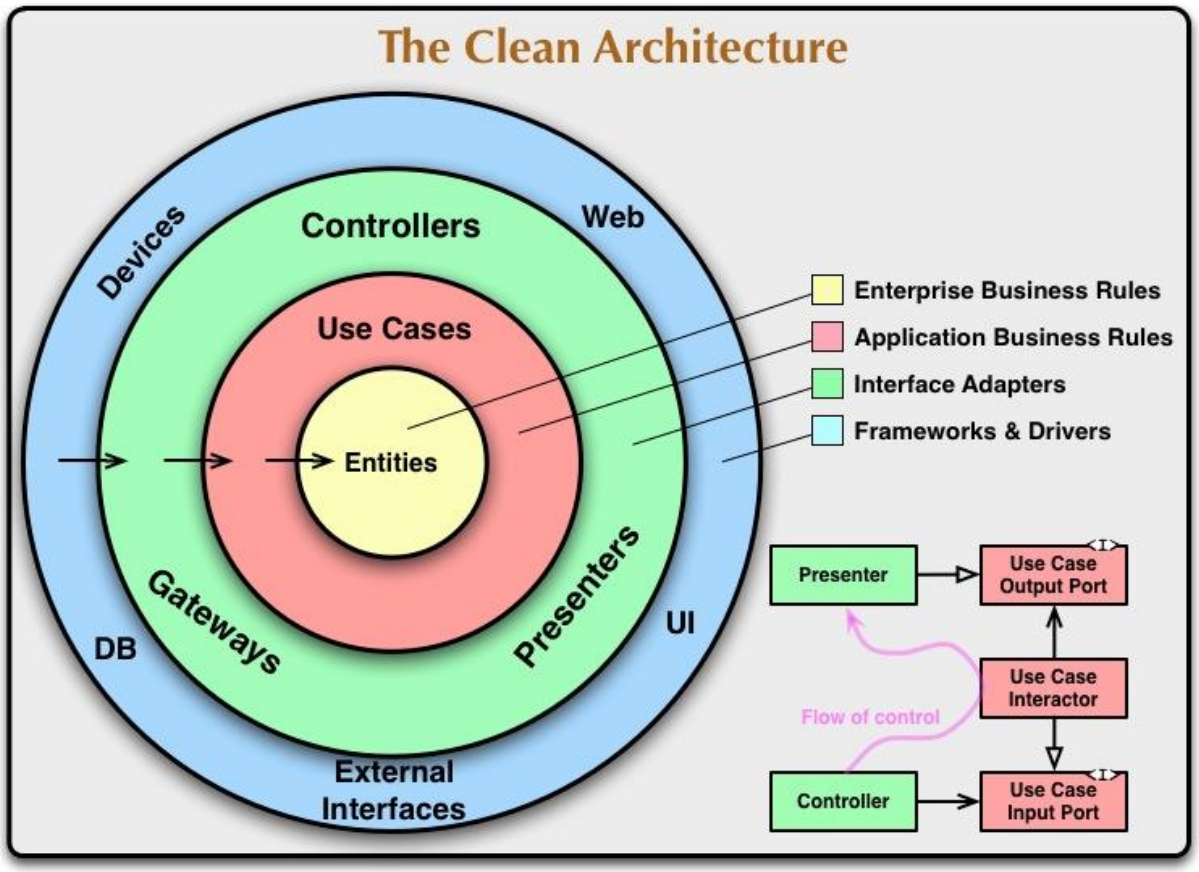


Photo from: https://jimmybogard.com/vertical-slice-architecture/

Εικόνα 13: A traditional layered/onion/clean architecture is monolithic in its approach.

Πριν δούμε τι είναι λοιπόν το Vertical Slice Architecture πρέπει να καταλάβουμε ότι δουλεύει μαζί με την αρχιτεκτονική CQRS. Αντί να σκεφτόμαστε το τον κώδικα και κάθε λύση μας σαν ένα κομμάτι του συστήματος θα δημιουργούμε κάθε λύση και κώδικα σαν ένα σύστημα μόνο του. Χρησιμοποιώντας την δυνατότητα της C# να χωρίζει κώδικας σε διάφορα επίπεδα τα οποία ονομάζονται namespaces [38] θα τοποθετούμε τον κώδικα σε προκαθορισμένα σημεία. Έτσι όταν για παράδειγμα θα δημιουργούμε την ιστοσελίδα μας και θα χρειαζόμαστε τις αίτησης, τα μοντέλα και τις διεπαφές που έχουν να κάνουνε με τους κόμβους, θα γνωρίζουμε ότι όλα όσα χρειαζόμαστε είναι σε ένα κοινό namespace η σε κάποιο εσωτερικό αλλά ποτέ έξω από αυτό.

Αυτό έχει ως αποτέλεσμα την πιο γρήγορη δημιουργία νέων δυνατών αλλά και την εύρεση σφαλμάτων, ενώ για οτιδήποτε χρειαζόμαστε σχετικά με ένα χαρακτηριστικό της εφαρμογής γνωρίζουμε που θα βρίσκεται και δεν χεριάζετε να το ψάχνουμε για ώρες ειδικά σε προτζεκτ όπως το δικό μας το οποίο έχει έναν πολύ μεγάλο όγκο τόσο κώδικα όσο και από προτζεκτ.

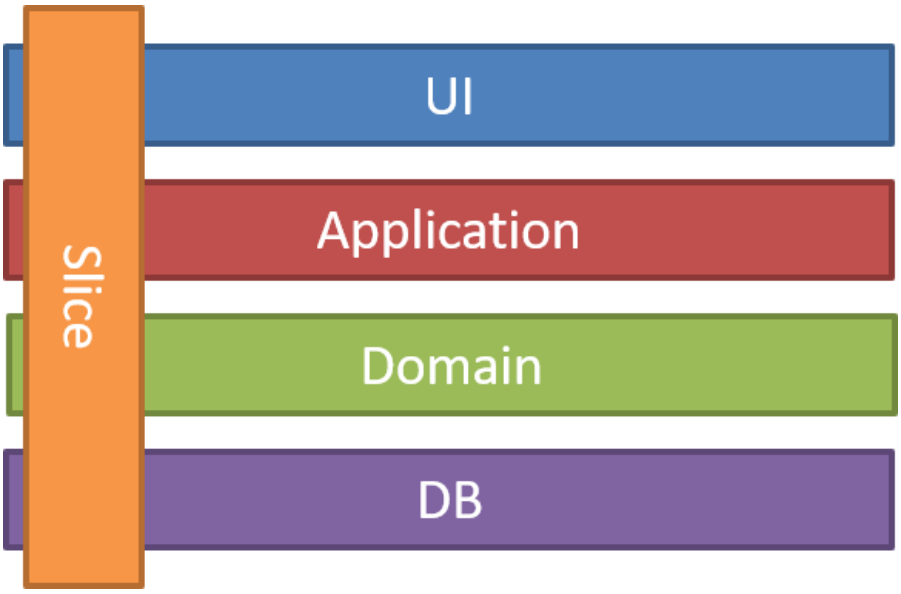


Photo from: https://jimmybogard.com/vertical-slice-architecture/

Εικόνα 14: Vertical Slice Architecture

2.5: Βιβλιοθήκες Κώδικα

Θα χρειαστεί να δημιουργήσουμε και κάποιες δίκες μας βιβλιοθήκες κοινού κώδικα όπως δηλαδή κώδικας που θα χρειαστούμε σε όλα τα προτζεκτ ανεξάρτητα προτζεκτ και πιο συγκεκριμένα σε όλους του διακομιστές μας. Όπως δημιουργήσαμε αρκετά κοινή λογική χρησιμοποιώντας την βιβλιοθήκη MediatR θα δημιουργήσουμε και μικρές δίκες μας βιβλιοθήκες για να μην επαναλαμβανόμαστε.

2.5.1: Abstractions

Οι πρώτοι κοινοί κώδικες βρίσκονται στην βιβλιοθήκη Abstractions, η οποία περιέχει κώδικα που μπορεί να χρησιμοποιηθεί και χρησιμοποιείτε σε μεγάλο βαθμό από όλα μας τα προτζεκτ.

Επειδή πιο κάτω θα δούμε ότι οι διακομιστές μας μπορούν να ρυθμιστούν χρησιμοποιώντας αρχεία YAML (Yet another markup language) [39] δημιουργήσαμε μερικές μεθόδους οι οποίες θα μας βοηθήσουν να ανακτήσουμε πληροφορίες από τα αρχεία αυτά πιο συγκεκριμένα από την διεπαφή IConfiguration η οποία έχει στην διάθεση της κάθε τιμή την οποία δημιουργήσαμε στα YAML αρχεία μας και όχι μόνο. Διαθέτει τιμές που μπορεί να βρίσκονται στο PATH μία μεταβλητή με πολλές τιμές του λειτουργικού συστήματος αλλά και όσες τιμές μπορεί να δόθηκαν σαν παράμετροι την ώρα της εκτέλεσης του προγράμματος μας.

Ο παρακάτω κώδικας μας επιστέφει την τιμή για το κλειδί «swagger:enabled» και σιγουρεύει ότι μπορεί να μετατραπεί σε Boolean δηλαδή μια τιμή που είναι αληθής η ψευδής αν δεν γίνετε επιστέφει ότι δεν χεριάζετε να ενεργοποιηθεί αυτή η ρύθμιση.

```
/// <summary>
/// Get whether swagger should be enabled or not.
/// </summary>
/// <param name="configuration">The configuration to search for the value.</param>
/// <returns>True to enable swagger otherwise false.</returns>
1 reference
public static bool GetSwagger(this IConfiguration configuration)
{
    if (bool.TryParse(configuration["swagger:enabled"], out var enabled))
    {
        return enabled;
    }

    return false;
}
```

Εικόνα 15: GetSwagger.

Ο κώδικας δίνουμε μια δική μας και λαμβάνουμε ότι τιμή υπάρχει στο κλειδί service + την δική μας τιμή ως γραμματοσειρά.

```
/// <summary>
/// Get a service uri.
/// </summary>
/// <param name="configuration">The configuration to search for values.</param>
/// <param name="key">The key of the service to search for.</param>
/// <returns>The complete uri for the service.</returns>
6 references
public static string GetService(this IConfiguration configuration, string key)
{
    return configuration.ReadValue($"services:{key}");
}
```

Εικόνα 16: GetService.

Εδώ θέλουμε σε μορφή γραμματοσειράς μια τιμή στην ενότητα secrets + την τιμή που επιζητούμε. Secrets είναι η ενότητα στην οποία βάζουμε κλειδιά που θέλουμε να μείνουν μυστικά δηλαδή να υπάρχουν στον διακομιστή μόνο όταν αυτός δουλεύει. Αυτό μπορεί να γίνει όταν εκτελούμε τον διακομιστή αλλά στην δική μας περίπτωση θα το κάνουμε μέσω τον yaml αρχείων.

```
/// <summary>
/// Get a value from the Secrets section.
/// </summary>
/// <param name="configuration">The configuration to search for values.</param>
/// <param name="key">The secret to search for.</param>
/// <returns>The secret.</returns>
3 references
public static string GetSecret(this IConfiguration configuration, string key)
{
    return configuration.ReadValue($"secrets:{key}");
}
```

Εικόνα 17: GetSecret ως string.

Ίδιος όπως ο παραπάνω κώδικας απλά κάνουμε ένα παραπάνω βήμα να πάρουμε μια σειρά από bytes αντί για γράμματα, αυτό διότι κάποιες φορές οι βιβλιοθήκες ταυτοποίησης θέλουν το συγκεκριμένο τύπο δεδομένων.

```
/// <summary>
/// Get a value from the Secrets section as byte[] using the provided encoding.
/// If no encoding is provided <see cref="Encoding.UTF8" /> will be used.
/// </summary>
/// <param name="configuration">The configuration to search for values.</param>
/// <param name="key">The secret to search for.</param>
/// <param name="encoding">The encoding to use, pass null or default to use UTF8.</param>
/// <returns>The secret in byte[] form.</returns>
0 references
public static byte[] GetSecret(this IConfiguration configuration, string key, Encoding? encoding = null)
{
    var secret = configuration.GetSecret(key);

    return encoding == null
            ? Encoding.UTF8.GetBytes(secret)
            : encoding.GetBytes(secret);
}
```

Εικόνα 18: GetSecret ως byte[].

Κώδικας που διαβάζει οποιαδήποτε τιμή θέλουμε εμείς σε μορφή γραμματοσειράς. Κάνει έλεγχο αν η τιμή που ψάχνουμε υπάρχει και αν δεν είναι άδεια. Αν κάποιο από τα προηγούμενα ισχύει θα δημιουργήσει ένα σφάλμα ότι δεν μπορεί να βρει την τιμή. Επίσης θα ελέγξει αν η τιμή μας ξεκινά με τον χαρακτήρα «<» αυτό διότι οι προκαθορισμένες ρυθμίσεις δεν είναι συμπληρωμένες απλά περιγράφουν την κάθε ρυθμίσει ξεκινώντας με «» μπροστά και «» στο τέλος για να τα αναγνωρίζει το πρόγραμμα ότι δεν έχουν δοθεί. Αν δεν έχουν δοθεί τότε θα δημιουργηθεί ανάλογο σφάλμα με το όνομα της τιμής που λείπει.

```
6 references
public static string ReadValue(this IConfiguration configuration, string key)
{
    var value = configuration[key];

    if (value is null or { Length: 0 })
    {
        throw new ArgumentNullException(nameof(key), $"Could not find value for: '{key}'");
    }
    else if (value.StartsWith("<"))
    {
        throw new ArgumentException($"No value has been provided for '{key}'", nameof(key));
    }

    return value;
}
```

Εικόνα 19: ReadValue.

Διεπαφή που προσδιορίζει μία μέθοδο η οποία επιστέφει την ώρα του διακομιστή σε μορφή UTC (Coordinated Universal Time) και περιέχει την διαφορά της ώρας από την ώρα UTC για ενδεχόμενο θα την χρειαστούμε. Δηλαδή στην περίπτωση αυτή +00:00 ενώ για παράδειγμα σε ώρα Ελλάδας θα ήτα +02:00 ή +03:00 ανάλογα την εποχή.

```
/// <summary>
/// A service that returns the current <see cref="DateTimeOffset" /> in utc.
/// </summary>
9 references
public interface IClockService
{
    /// <summary>
    /// Returns the time now in utc
    /// </summary>
    /// <returns></returns>
        7 references
    DateTimeOffset GetNow();
}
```

Εικόνα 20: IClockService.

Μία διεπαφή στην οποία ο σκοπός της είναι να δημιουργεί κλειδιά για την ταυτοποίηση κάθε αντικειμένου στην βάση δεδομένων ένα ID.

```
/// <summary>
/// A service to generate 11 character primary keys.
/// </summary>
14 references
public interface IPrimaryKeyService
{
    /// <summary>
    /// Generates an 11 character string suitable for use as a primary key.
    /// </summary>
    /// <returns>A 11 character string for use as a primary key.</returns>
        9 references
    string Create();
}
```

Εικόνα 21: IPrimaryKeyService.

Διεπαφή με σκοπό δημιουργίας κλειδιών για χρήση σαν κωδικούς ασφαλείας ή αλλιώς Tokens.

```
/// <summary>
/// A service to generate tokens.
/// </summary>

public interface ITokenService
{
    /// <summary>
    /// Generate a 45 character string suitable for use as a token.
    /// </summary>
    /// <returns>A string for use as a token.</returns>

    string Create();
}
```

Εικόνα 22: ITokenService.

Διεπαφή με σκοπό να επιστρέψει το όνομα χρήστη ο οποίος εκτέλεσε μια συγκεκριμένη ενέργεια για παράδειγμα την εντολή δημιουργίας ενός προϊόντος σε μια ιστοσελίδα.

```
/// <summary>
/// A service to grab the current user in the request pipeline.
/// </summary>
3 references
public interface IUserService
{
    /// <summary>
    /// Get the identity of the user or null if not found id is found.
    /// </summary>
    /// <returns>The user identity or null.</returns>
    4 references
    string? GetId();
}
```

Εικόνα 23: IUserService.

2.5.2: CoreHost

Βιβλιοθήκη την οποία τη χρησιμοποιούν όλες οι εφαρμογές οι οποίες θα χρειαστεί να τρέξουν σε κάποιον διακομιστή.

Κώδικας για που εφαρμόζει την IClockService διεπαφή επιστρέφοντας τιμή του συστήματος σε μορφή UTC.

```
/// <summary>
/// Implementation of <see cref="IClockService" />
/// </summary>
2 references
public class ClockService : IClockService
{
    /// <inheritdoc />
    7 references
    public DateTimeOffset GetNow()
    {
        return DateTimeOffset.UtcNow;
    }
}
```

Εικόνα 24: IClockService Implementation.

Κώδικας που εφαρμόζει την IPrimaryKeyService διεπαφή. Χρησιμοποιώντας την εσωτερική βιβλιοθήκη της C# Path χρησιμοποιούμε την μέθοδο GetRandomFileName, αυτή η μέθοδος μας εγγυάται ότι θα δημιουργήσει μια τυχαία αλληλουχία γραμμάτων με μια τελεία για 8ο ψηφίο και σύνολο 12 γράμματα. Εμείς αφού λάβουμε αυτήν την γραμματοσειρά αφαιρούμε την τελεία και επιστρέφουμε το αποτέλεσμα.

```
/// <summary>
/// Implementation of <see cref="IPrimaryKeyService" />
/// </summary>
2 references
public class PrimaryKeyService : IPrimaryKeyService
{
    /// <inheritdoc />
    9 references
    public string Create() =>
        // Removes "." from the string befor it returns.
        Path.GetRandomFileName().Remove(8, 1);
}
```

Εικόνα 25: IPrimaryKeyService Implementation.

Κώδικας που εφαρμόζει την διεπαφή ITokenService. Για να εφαρμόσουμε την διεπαφή δημιουργήσαμε μια μέθοδο η οποία χρησιμοποιεί συγκεκριμένους χαρακτήρες. Οι χαρακτήρες αυτοί είναι αποθηκευμένοι στο TokenChars. Επαναλαμβάνοντας τα βήματα length που μας δίνονται διαλέγουμε τυχαία έναν χαρακτήρα και το τον τοποθετούμε στην tokenPool διάταξη. Η «RandomNumberGenerator.GetInt32 [40] » μέθοδος μας εγγυάται αληθινά τυχαίους αριθμούς οι οποίοι δεν γίνετε να προβλεφθούν με κανέναν τρόπο έτσι είμαστε σίγουροι ότι διαλέξαμε αληθινά τυχαία έναν χαρακτήρα. Στο τέλος δημιουργούμε το token μας διαγράφουμε την αλληλουχία και επιστρέφουμε τη τιμή. Ο κώδικας που εφαρμόζει την διεπαφή καλεί την μέθοδο δίνοντας τον αριθμό 45, είναι ένας αρκετά μεγάλος αριθμός για τα σημερινά δεδομένα και καθιστά αδύνατο κάποιος να δημιουργήσει το ίδιο token στην τύχη σε συνδυασμό με την «RandomNumberGenerator.GetInt32» μέθοδο.

```
/// <summary>
/// Implementation of <see cref="ITokenService" />
/// </summary>
2 references
public class TokenService : ITokenService
{
    /// <inheritdoc />
    2 references
    public string Create() => Create(45);

    #region Token Implementations

    public const string TokenChars = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";

    /// <summary>
    /// Generate any length character string suitable for use as a token.
    /// </summary>
    /// <returns>A string for use as a token.</returns>
    1 reference
    public static string Create(int length)
    {
        var tokenPool = ArrayPool<char>.Shared.Rent(length);
        for (int i = 0; i < length; i++)
        {
            var charIndex = RandomNumberGenerator.GetInt32(0, TokenChars.Length);
            tokenPool[i] = TokenChars[charIndex];
        }

        var token = new string(tokenPool[..length]);
        ArrayPool<char>.Shared.Return(tokenPool);
        return token;
    }

    #endregion
}
```

Εικόνα 26: ITokenService Implementation.

Κώδικας που εφαρμόζει την διεπαφή IUserService, χρησιμοποιώντας το αντικείμενο IHttpContextAccessor το οποίο διαθέτετε από την πλατφόρμα ASP.NET Core προσάγουμε να διαβάσουμε το όνομα του χρήστη, και αν δεν γίνετε επιστρέφουμε «null».

```
/// <summary>
/// Implementation of <see cref="IUserService"/>
/// </summary>
3 references
public class UserService : IUserService
{
    private readonly IHttpContextAccessor httpContextAccessor;

    0 references
    public UserService(IHttpContextAccessor httpContextAccessor)
    {
        this.httpContextAccessor = httpContextAccessor;
    }

    /// <inheritdoc/>
    4 references
    public string? GetId() =>
        httpContextAccessor.HttpContext
            ?.User
            ?.Identity
            ?.Name;
}
```

Εικόνα 27: IUserService Implementation.

Κώδικας που επεκτείνει την λειτουργία της διεπαφή IConfiguration και μας βοηθά να ρυθμίσουμε την βάση δεδομένων μας CouchDB [41] την οποία θα δούμε στην ενότητα «Βάση Δεδομένων».

```
/// <summary>
/// Configure couchdb from the configuration.<br/>
/// Configures the http endoint, the name and the password.
/// </summary>
/// <param name="builder">The couchdb builder to configure.</param>
/// <param name="configuration">The configuration to use.</param>
/// <returns>The builder for more customization.</returns>
2 references
public static CouchOptionsBuilder Configure(this CouchOptionsBuilder builder, IConfiguration configuration)
{
    var values = configuration.GetService("couchdb").Split(';');

    if (values.Length < 3) throw new ArgumentException(
        $"Invalid configuration value. The connection string needs to be" +
        $"at least 3 values seperated by ';' you provided {values.Length}");

    var username = values[0];
    var password = values[1];
    var endpoint = new Url(values[2]);

    builder.UseBasicAuthentication(username, password)
           .UseEndpoint(endpoint);

    return builder;
}
```

Εικόνα 28: Configure μέθοδος για την βάση δεδομένων.

Κώδικας που μας επιστρέφει το όνομα από τις ρυθμίσεις για την βάση δεδομένων CouchDB.

```
/// <summary>
/// Get the couchdb database to use from the configuration.<br/>
/// </summary>
/// <param name="configuration">The configuration to use.</param>
/// <returns>The builder for more customization.</returns>
4 references
public static string GetCouchDatabase(this IConfiguration configuration)
{
    var values = configuration.GetService("couchdb").Split(';');

    if (values.Length < 4) throw new ArgumentException(
        $"Invalid configuration value. The connection string needs to be at least 4 values seperated by ';' " +
        $"where the fourht value is the database name. You provided {values.Length}");

    return values[3];
}
```

Εικόνα 29: GetCouchDatabase.

Κώδικας που διαβάζει τις ρυθμίσεις και ανάλογα θα ρυθμίσει την υπηρεσία Swagger [42]. Swagger είναι μια υπηρεσία που διαβάζει τον τελικό μας κώδικα και δημιουργεί μια δική του ιστοσελίδα που μας δείχνει τα σημεία επικοινωνίας του διακομιστή μας. Αυτό δουλεύει μόνο για την HTTP Rest υπηρεσία.

```
private static bool _use = false;
private static string _name = string.Empty;
private static string _version = string.Empty;

2 references
public static IServiceCollection TryAddSwagger(this IServiceCollection services, string name, string version, IConfiguration configuration)
{
    if (!configuration.GetSwagger()) return services;

    _use = true;
    _name = name;
    _version = version;

    var info = new OpenApiInfo
    {
        Version = version,
        Title = name,
    };

    services.AddSwaggerGen(options =>
    {
        options.SwaggerDoc($"v{version}", info);
    });

    return services;
}
```

Εικόνα 30: TryAddSwagger.

Όπως ο πιο πάνω κώδικας θα διαβάσει τις ρυθμίσεις και αν έχουν ενεργοποιηθεί θα ενεργοποίηση τα σημεία για την πρόσβαση και περιγραφή του HTTP Rest API μας.

```
2 references
public static IApplicationBuilder TryUseSwagger(this IApplicationBuilder app)
{
    if (!_use) return app;

    app.UseSwagger();
    app.UseSwaggerUI(options =>
    {
        options.SwaggerEndpoint(
            url: $"v{_version}/swagger.json",
            name: $"{_name} - {_version}");
    });

    return app;
}
```

Εικόνα 31: TryUseSwagger.

Κώδικας για την παραμετροποίηση του διακομιστή ο οποίος ρυθμίζει την αρχική τοποθεσία που χρησιμοποιεί η εφαρμογή στον φάκελο από την οποία εκτελείται. Ρυθμίζει τον διακομιστεί να διαβάζει ρυθμίσεις από το PATH των windows άμα ξεκινάνε με την λέξη «DOTNET_» και ότι άλλες ρυθμίσεις δώσαμε στην ιδιά την εφαρμογή στο ξεκίνημα. Στην συνέχει ρυθμίζετε η ίδια η εφαρμογή η οποία θα ψάξει σύνολο για τέσσερα διαφορετικά αρχεία που περιέχουν ρυθμίσεις ενώ πάλι χρησιμοποιούμε ρυθμίσεις από το PATH και από τις ρυθμίσεις που περαστήκαν στο ξεκίνημα. Τέλος ρυθμίζουμε την εφαρμογή καταγραφής η οποία θα ρυθμίζεται με περεταίρω ρυθμίσεις από το αντικείμενο IConfiguration.

```
5 references
public static IHostBuilder ConfigureCoreHost(this IHostBuilder builder, string[] args) => builder
    // Set default directory for app.
    .UseContentRoot(Environment.CurrentDirectory)
    // Configure the Core Host.
    .ConfigureHostConfiguration(options => options
        .AddEnvironmentVariables("DOTNET_")
        .AddCommandLine(args))
    // Configure the Core App.
    .ConfigureAppConfiguration((context, options) => options
        // Settings from where app assembly exists.
        .AddYaml($"{CurrentDir}/appsettings.yaml")
        .AddYaml($"{CurrentDir}/appsettings.{context.CurrentEnv()}.yaml")
        // Settings from where app executes.
        .AddYaml($"appsettings.yaml")
        .AddYaml($"appsettings.{context.CurrentEnv()}.yaml")
        .AddEnvironmentVariables()
        .AddCommandLine(args))
    // Set the default logging provider.
    .UseSerilog((context, options) => options
        .ReadFrom.Configuration(context.Configuration, "Serilog"));
```

Εικόνα 32: ConfigureCoreHost.

Κώδικας που εκτελεί τον διακομιστή μας, καταγραφεί το όνομα που θα του δώσουμε και ξεκινά. Σε περίπτωση σφάλματος φροντίζει να το καταγράψει πριν κλίσει τον διακομιστή.

```
4 references
public static async Task<int> RunCoreHostAsync(this IHost host, string name)
{
    try
    {
        Log.Information("Application '{name}' is starting.", name);
        await host.RunAsync();
        Log.Information("Application '{name}' shut down.", name);
        return 0;
    }
    catch (Exception ex)
    {
        Log.Fatal(ex, "Application '{name}' terminated unexpectedly.", name);
        return 1;
    }
    finally
    {
        Log.CloseAndFlush();
    }
}
```

Εικόνα 33: RunCoreHost.

Κώδικας που καλείτε από κάθε προτζεκτ στο οποίο θέλουμε να έχουμε διαθέσιμες τις διεπαφές που είδαμε πιο πριν ενώ δηλώνουμε ότι θα χρειαστούμε και την διεπαφή IHttpContextAccessor.

```
5 references
public static IServiceCollection AddCoreHostServices(this IServiceCollection services)
{
    // Service that helps to access the user id.
    services.AddHttpContextAccessor();

    services.AddSingleton<IClockService, ClockService>();
    services.AddSingleton<IPrimaryKeyService, PrimaryKeyService>();
    services.AddSingleton<ITokenService, TokenService>();
    services.AddScoped<IUserService, UserService>();

    return services;
}
```

Εικόνα 34: AddCoreHostServices.

2.5.3: CoreWeb

Βιβλιοθήκη την οποία τη χρησιμοποιούν όλες οι εφαρμογές οι οποίες θα τρέχουν σαν ιστοσελίδα. Πιο συγκεκριμένα όλες μας οι ιστοσελίδες θα τρέχουν σαν SPA (Singe Page Application) [2] οι οποίες είναι ιστοσελίδες που όταν φορτώσουν συμπεριφέρονται σαν μια εφαρμογή υπολογιστή.

Η βιβλιοθήκη είναι βασισμένη στην τεχνολογία Blazor και θα μας προσφέρει αρκετά αντικείμενα για να χτίσουμε τις ιστοσελίδες μας.

Στην βιβλιοθήκη αυτή επίσης θα έχουμε μερικά αρχεία CSS και JavaScript για να μην επαναλαμβάνουμε κώδικα. Πιο συγκεκριμένα η CSS που θα χρησιμοποιήσουμε θα δημιουργηθεί με την βοήθεια της βιβλιοθήκης Tailwind CSS [43] ενώ η JavaScript απλά θα βοηθά να αλλάζει η ιστοσελίδα μας σκούρο η φωτεινό θέμα.

Ο παρακάτω κώδικας αποθηκεύει στον περιηγητή του χρήστη το ενεργό θέμα, ενώ δημιουργεί μεθόδους οι οποίες ελέγχουν την αλλαγή διαλέγοντας το σωστό ισότοπο για το Water CSS το οποίο αντικαθιστάτε στην βασική html σελίδα όπως θα δούμε στις διαδικτυακές εφαρμογές πιο μετά.

Ο παρακάτω κώδικας εκτελεί απευθείας κώδικα JavaScript που πιο κάτω θα δούμε ότι ελέγχει το θέμα της ιστοσελίδας. Αυτός ο κώδικας απλά τον τυλίγει σε κώδικα C#.

```
10 references
public class ThemeJs
{
    public const string Light = "light";

    public const string Dark = "dark";

    private readonly IJSRuntime js;

    0 references
    public ThemeJs(IJSRuntime jsRuntime) => js = jsRuntime;

    2 references
    public ValueTask<string> GetTheme() => js.InvokeAsync<string>("window.getTheme");

    2 references
    public ValueTask<string> Toggle() => js.InvokeAsync<string>("window.toggleTheme");
}
```

Εικόνα 350: CoreWeb ThemeJs 1.

Ο παρακάτω JavaScript κώδικας χρησιμοποιείτε για να κάνουμε την εναλλαγή από σκούρο θέμα σε φωτεινό και το αντίθετο. Για να έχουμε σκούρο και φωτεινό θέμα χρησιμοποιούμε μια βιβλιοθήκη που λέγετε Water CSS [44] και μας δίνει μαζί με ωραία θέματα για τα κουμπιά τα χρώματα και την δυνατότητα του σκούρου η λευκού θέματος.

```
'use strict'
const themeKey = 'theme-stylesheet';
const urlBase = 'https://cdn.jsdelivr.net/npm/water.css@2/out/';
const stylesheet = document.getElementById(themeKey);
const root = document.querySelector(':root').style;

const getTheme = () => window.localStorage.getItem(themeKey);

const setTheme = (theme) => window.localStorage.setItem(themeKey, theme);

const updateTheme = () => {
    const theme = getTheme();
    stylesheet.href = `${urlBase}${theme}.min.css`;

    if (theme === 'light') {
        root.setProperty('--blue', '#1976d2');
        root.setProperty('--blue-hover', '#0d47a1');
        root.setProperty('--red', '#d32f2f');
        root.setProperty('--red-hover', '#b71c1c');
        root.setProperty('--amber', '#ffa000');
        root.setProperty('--amber-hover', '#ff6f00');
    }
    else {
        root.setProperty('--blue', '#004ba0');
        root.setProperty('--blue-hover', '#0069c0');
        root.setProperty('--red', '#9a0007');
        root.setProperty('--red-hover', '#ba000d');
        root.setProperty('--amber', '#c67100');
        root.setProperty('--amber-hover', '#c79100');
    }
};

const toggleTheme = () => {
    const newTheme = getTheme() === 'light' ? 'dark' : 'light';
    setTheme(newTheme);
    updateTheme();
    return newTheme;
};

if (!getTheme()) {
    const isLight = window.matchMedia('(prefers-color-scheme: light)').matches;
    setTheme(isLight ? 'light' : 'dark');
}

window.themeKey = themeKey;
window.getTheme = getTheme;
window.setTheme = setTheme;
window.updateTheme = updateTheme;
window.toggleTheme = toggleTheme;

updateTheme();
```

Εικόνα 361: CoreWeb ThemeJs 2.

Κώδικας που έχει αποθηκευμένες διάφορες μεταβλητές μεγέθους της ιστοσελίδας. Αυτές οι τιμές χρησιμοποιούνται από τις ιστοσελίδες για να δουν τι μέγεθος έχει ο περιηγητής και ανάλογα το μέγεθος μπορούν να εκτελέσουν κώδικα η και να αλλάξουν τα γραφικά τους.

```
1 reference
public static class BreakPointsCss
{
    /// <summary>
    /// @media (min-width: 640px);
    /// </summary>
    public const string sm = "(min-width: 640px)";

    /// <summary>
    /// @media (min-width: 768px);
    /// </summary>
    public const string md = "(min-width: 768px)";

    /// <summary>
    /// @media (min-width: 1024px);
    /// </summary>
    public const string lg = "(min-width: 1024px)";

    /// <summary>
    /// @media (min-width: 1280px);
    /// </summary>
    public const string xl = "(min-width: 1280px)";

    /// <summary>
    /// @media (min-width: 1536px);
    /// </summary>
    public const string xl2 = "(min-width: 1536px)";
}
```

Εικόνα 37: CoreWeb BreakPointsCss.

Διάφοροι τύποι κουμπιών html οι οποίοι χρησιμοποιούνται από διάφορα αντικείμενα που θα δούμε πιο κάτω.

```
/// <summary>
/// The type of a button default is submit.
/// </summary>
3 references
public enum ButtonType
{
    submit,
    button,
    reset
}
```

Εικόνα 38: CoreWeb ButtonType.

Βασικό αντικείμενο από το οποίο όλα μας τα αντικείμενα κληρονομούν διάφορη λειτουργικότητα η οποία έχει να κάνει με την CSS κάθε αντικειμένου html ώστε να κάνουμε πιο εύκολη την χρήση του.

```
28 references
public class BaseComponent : ComponentBase
{
    /// <summary>
    /// Class for other components to provide to yours.
    /// </summary>
    [Parameter]
    16 references
    public string? Class { get; set; }

    /// <summary>
    /// Css builder to use instead of using your own.
    /// </summary>
    protected CssBuilder css;

    /// <summary>
    /// Get a Css builder with Class prepended to it or emtpy.
    /// </summary>
    /// <returns>A Css builder for the current Class.</returns>
    0 references
    protected CssBuilder Css() => Class is null
        ? CssBuilder.Empty() : CreateCss(Class);

    /// <summary>
    /// Get a Css builder with Class appended.
    /// </summary>
    /// <returns>A Css builder for the current Class.</returns>
    1 reference
    protected CssBuilder Css(string css) => CssBuilder.Default(css).AddClass(Class);

    /// <summary>
    /// Get a new Css builder to build your css.
    /// </summary>
    /// <param name="css"></param>
    /// <returns>An empty css builder.</returns>
    8 references
    protected static CssBuilder CreateCss(string css) =>
        CssBuilder.Default(css);
}
```

Εικόνα 39: CoreWeb BaseComponent.

Βασικό αντικείμενο φόρτωσης το οποίο το χρησιμοποιούμε για να δείξουμε ότι μια διαδικασία έχει ξεκινήσει.

```
@inherits BaseComponent

<div class="@Class">
    <div class="spinner">
        <div class="rect1"></div>
        <div class="rect2"></div>
        <div class="rect3"></div>
        <div class="rect4"></div>
        <div class="rect5"></div>
    </div>
</div>
```

Εικόνα 40: CoreWeb Spinner.

Αντικείμενο το οποίο απλά δείχνει στον χρήστη ότι έχει περιηγηθεί σε ένα μέρος τις ιστοσελίδας μας το οποίο δεν υπάρχει.

```
<p>Sorry, there's nothing at this address.</p>
```

Εικόνα 41: CoreWeb NotFound.

Αντικείμενο το οποίο και αυτό δείχνει ότι μια διαδικασία έχει ξεκινήσει απλά πιο στοχευμένο για διαδικασίες που εκτελούνται σε κουμπιά και όταν ο χρήστης τα ξεκινάει αντί να συμβαίνουν λόγο της φόρτωσης της ιστοσελίδας.

```
@inherits BaseComponent

<div class="@Css("loader")"></div>
```

Εικόνα 42: CoreWeb Loader.

Το αντικείμενο που βλέπουμε στην συνέχεια είναι ένα κουμπί με την ιδιότητα της αντίδρασης. Πιο συγκεκριμένα για αυτήν τη αντίδραση χρησιμοποιούμε την βιβλιοθήκη ReactiveUI [45]. Η λογική αυτή βασίζετε ότι ο κώδικας μας είναι ασύγχρονος και ότι απλά αντιδρά σε διάφορα συμβάντα.

Χρησιμοποιώντας την παραπάνω λογική βλέπουμε ένα κουμπί το οποίο δέχεται διάφορες μεταβλητές.

- Command: Ο κώδικας που θα εκτελέσουμε
- CanExecute: Μια μεταβλητή που δείχνει ότι ο κώδικας μπορεί να εκτελεστεί, αν μάλιστα είναι ψευδής τότε το κουμπί αλλάζει και γίνετε απενεργοποιημένο φέροντας την δυνατότητα να πατηθεί και να εκτελέσει το Command.
- IsExecuting: Δείχνει ότι το κουμπί εκτελεί κώδικα.
- ChildContent: Είναι ο html κώδικας που θα εμφανιστεί μέσα στο κουμπί. Μάλιστα το συγκεκριμένο λειτουργεί σαν μέθοδος δίνοντας την δυνατότητα σε όποιον το χρησιμοποιεί να παραμετροποίηση τι δείχνει το κουμπί ανάλογα με τον εαυτό του.

- Type: Είναι ο τύπος του html κουμπιού μας.
- OnClick: Μέθοδος διαθέσιμη μόνο σε αυτό το αντικείμενο η οποία προσπαθεί να εκτελέσει το Command.
- OnParameterSet: Μέθοδος που εκτελείτε από τη πλατφόρμα Blazor αμέσως αφού όλες οι μεταβλητές έχουν οριστεί. Εκεί εμείς βλέποντας αν το Command έχει οριστεί χρησιμοποιούμε Reactive κώδικα για να αντιδράσουμε στις διάφορες εντολές του. Κάθε φορά που η μέθοδος του CanExecute εκτελείτε εμείς θέτουμε την δική μας μεταβλητή, το ίδιο και για την IsExecuting.

```
@inherits BaseComponent
@typeparam TResult

<button class="@Class" type="@Type" @onclick="OnClick" disabled=@(!CanExecute)>@ChildContent?.Invoke(this)</button>

@code{
    /// <summary>
    /// The underlying reactive command for this button. The button state
    /// will change depending on the command.
    /// </summary>
    [Parameter]
    public ReactiveCommand<Unit, TResult> Command { get; set; } = null!;

    /// <summary>
    /// Gets a property whose value indicates whether the command can currently execute.
    /// </summary>
    public bool CanExecute { get; private set; } = true;

    /// <summary>
    /// Gets a property whose value indicates whether the command is currently executing.
    /// </summary>
    public bool IsExecuting { get; private set; } = false;

    /// <summary>
    /// The button type. Default is submit.
    /// </summary>
    [Parameter]
    public ButtonType Type { get; set; }

    [Parameter]
    public RenderFragment<ReactiveButton<TResult>>? ChildContent { get; set; }

    protected void OnClick() => Command?.Invoke();

    protected override void OnParametersSet()
    {
        base.OnParametersSet();

        if (Command is null) return;

        Command.CanExecute.Subscribe(x =>
        {
            CanExecute = x;
            StateHasChanged();
        });
        Command.IsExecuting.Subscribe(x =>
        {
            IsExecuting = x;
            StateHasChanged();
        });
    }
}
```

Εικόνα 43: CoreWeb ReactiveButton.

Χρησιμοποιώντας το ReactiveButton δημιουργήσαμε ένα ακόμα πιο εξειδικευμένο κουμπί το SpinnerButton. Το κουμπί αυτό αντιδρώντας στο ReactiveButton αποφασίζει αν θα δήξει το αντικείμενο Loader η αν θα δήξει τον δικό μας κώδικα html.

```
@inherits BaseComponent
@typeparam TResult

<ReactiveButton Class="@Class" TResult="TResult" Type="Type" Command="Command" Context="button">
    @if (button.IsExecuting)
    {
        <div class="loader"></div>
    }
    else
    {
        @ChildContent
    }
</ReactiveButton>

@code{
    /// <summary>
    /// The underlying reactive command for this button. The button state
    /// will change depending on the command.
    /// </summary>
    [Parameter]
    public ReactiveCommand<Unit, TResult> Command { get; set; } = null!;

    /// <summary>
    /// The button type. Default is submit.
    /// </summary>
    [Parameter]
    public ButtonType Type { get; set; }

    [Parameter]
    public RenderFragment? ChildContent { get; set; }
}
```

Εικόνα 44: CoreWeb SpinnerButton.

Το επόμενο αντικείμενο λέγεται Dialog και όπως δηλώνει και το όνομα του απλά δείχνει έναν «διάλογο» πάνω από όλα τα αντικείμενα της σελίδας μας. Μάλιστα δείχνει μόνο ότι του έχουμε δώσει όταν πρέπει να τα δήξει.

```
@using System.Reactive.Subjects
@inherits BaseComponent
@implements IAsyncDisposable
@inject IJSRuntime js

<dialog id="@_id" class="@Class">
    @if (Title is not null)
    {
        <header>@Title</header>
    }
    @if (ChildContent is not null)
    {
        @ChildContent
    }
</dialog>
```

Εικόνα 45: CoreWeb Dialog 1.

Ο κώδικας με την σειρά του δίνει διάφορες μεθόδους για να εμφανίσουμε η να κρύψουμε αυτόν τον «διάλογο» ενώ δίνει και αντικείμενα στα οποία μπορούμε να αντιδράσουμε. Το ποιο σημαντικό κώματι όμως είναι ότι δίνουμε σε κάθε html dialog ένα μοναδικό υποκοριστικό (id). Αυτό το κάνουμε διότι θα χρειαστούμε JavaScript για να εμφανίσουμε τον διάλογο και στην JavaScript θα δώσουμε το υποκοριστικό αυτού του διαλόγου.

Έτσι προσθέτουμε και ένα μοναδικό τυχαίο συνδυασμό γράμματών για να είμαστε σίγουροι ότι όσοι διάλογοι και αν είναι σε μια σελίδα html εμείς θα δείξουμε τον σωστό και αυτό επιβεβαιώνετε από τις ιστοσελίδες πιο μετρά οπού κάνουμε βαριά χρήση των διαλόγων.

```
@code{
    private readonly string _id = $"dialog-{Guid.NewGuid():N}";
    private readonly Subject<bool> _whenVisible = new();

    [Parameter]
    public RenderFragment? ChildContent { get; set; }

    [Parameter]
    public string? Title { get; set; }

    public bool Visible { get; private set; }

    public IObservable<bool> WhenVisible => _whenVisible;

    private readonly Lazy<Task<IJSObjectReference>> dialog;

    public Dialog()
    {
        dialog = new(() => js.InvokeAsync<IJSObjectReference>("document.getElementById", _id).AsTask());
    }

    public async ValueTask ShowAsync()
    {
        await (await dialog.Value).InvokeVoidAsync("showModal");

        Visible = true;
        _whenVisible.OnNext(true);
    }

    public async ValueTask HideAsync()
    {
        await (await dialog.Value).InvokeVoidAsync("close");

        Visible = false;
        _whenVisible.OnNext(false);
    }

    public async ValueTask<bool> ToggleAsync()
    {
        if (!Visible)
            await ShowAsync();
        else
            await HideAsync();

        return Visible;
    }

    public async void Show() => await ShowAsync();

    public async void Hide() => await HideAsync();

    public async void Toggle() => await ToggleAsync();

    public async ValueTask DisposeAsync()
    {
        if (dialog.IsValueCreated)
        {
            await (await dialog.Value).DisposeAsync();
        }
    }
}
```

Εικόνα 46: CoreWeb Dialog 2.

Κώδικας που καλείτε από τις εφαρμογές για να δηλώσουν βασικές λειτουργίες. Στην προκειμένη περίπτωση δηλώνουν την λειτουργία «ThemeJs» που όπως είδαμε βοηθά να ελέγχουμε το θέμα της ιστοσελίδας από την C#.

```
0 references
public static class DependencyInjection
{
    2 references
    public static IServiceCollection AddCoreWeb(this IServiceCollection services)
    {
        services.AddSingleton<ThemeJs>();

        return services;
    }
}
```

Εικόνα 47: CoreWeb DependencyInjection

2.6: Ανάπτυξη Εφαρμογών Γραφικού Περιβάλλοντος Web

Όπως αναφαίρετά και σε προηγούμενο κεφάλαιο οι εφαρμογές μας θα είναι δημιουργημένες με την πλατφόρμα Blazor. Το Blazor είναι μια SPA (Singe Page Application) πλατφόρμα η οποία μας δίνει την δυνατότητα να τρέχουμε κώδικα C# στον περιηγητή μας αντί για JavaScript ενώ αν χρειαστεί έχουμε την δυνατότητα να καλέσουμε και JavaScript.

Χρησιμοποιώντας λοιπόν την δυνατότητα αυτή θα δούμε ότι έχοντας κοινές βιβλιοθήκες πετυχαίνουμε πλήρη επαναχρησιμοποίηση κώδικα με τους διακομιστές μας, αυτό κάνει πολύ πιο εύκολη την δουλειά μας αφού γνωρίζουμε απευθείας τα δεδομένα που θα πρέπει να δείξουμε στον χρήστη ενώ σε πιθανή αλλαγή του βασικού πακέτου θα έχουμε απειθείς τις αλλαγές και στην εφαρμογή μας όχι μόνο στον διακομιστή.

Θα χρειαστούμε επίσης την βιβλιοθήκη ReactiveUI η οποία θα μας επιτρέψει να εφαρμόσουμε την τεχνική MVVM (Model, View, ViewModels) [46] η οποία είναι μια τεχνική δημιουργίας εφαρμογών γραφικού περιεχομένου.

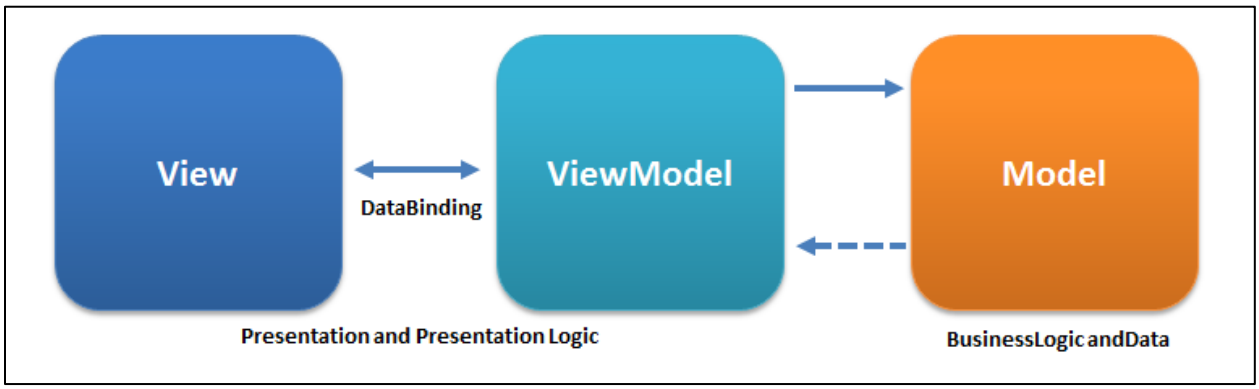


Photo from: https://en.wikipedia.org/wiki/Model-view-viewmodel

Εικόνα 48: MVVM

Στόχος της τεχνικής αυτής είναι ο πλήρης διαχωρισμός του κώδικα μιας σελίδας που θα αλληλοεπιδρά ο χρήστης. Πιο συγκεκριμένα δημιουργούμε μια κλάση η οποία θα έχει όλα όσα χρειαζόμαστε για να δει ο χρήστης ενώ ακόμα θα έχουμε και τις διάφορες εντολές που μπορεί να εκτελέσει αυτά τα ονομάζουμε ViewModels.

Ο βασικός κανόνας είναι ότι αυτό το αντικείμενο δεν γνωρίζει τίποτα για την όψη του τελικού περιβάλλοντος που θα δει ο χρήστης και απλά δίνει τα καταλληλά εργαλεία. Αυτό επιτρέπει σε αυτά τα αντικείμενα να μπορούν να χρησιμοποιηθούν σε οποιαδήποτε πλατφόρμα όπως Web, Window, Android Mac και άλλα.

Models θα είναι τα αντικείμενα τα οποία διαθέτουν την αληθινοί πληροφορία την οποία θα μας δώσει ένα ViewModels. Μάλιστα η δουλειά του ViewModels είναι να τα μαζέψει όλα και να τα μετατρέψει έτσι ώστε να μπούμε να τα αναπαραστήσουμε αλλά να μας δίνει και την δυνατότητα να τα αλλάξουμε. Τα Models δεν πρέπει να γνωρίζουν ότι βρίσκονται σε ένα ViewModel πρέπει απλά να είναι δεδομένα.

Σε σχέση με τα δυο προηγούμενα το αντικείμενο View είναι αυτό που θα υλοποιήσουμε και θα δει ο χρήστης. Είναι δεμένο με την πλατφόρμα του, για παράδειγμα σε εμάς είναι ένα αντικείμενο html ενώ στο Android θα μπορούσε να είναι ένα μενού. Το View είναι το πιο ελεύθερο από όλα και μπορεί να γνωρίζει τα πάντα τόσο για το ViewModel αλλά και για τα Models του μάλιστα πολλές φορές ενθαρρύνετε αυτή η συμπεριφορά.

Το μεγαλύτερο πλεονέκτημα είναι ότι η λογική του κώδικα μας το ViewModel είναι απομονωμένο και μπορεί να τρέξει σε οποιαδήποτε πλατφόρμα αυτό μας δίνει την δυνατότητα να το δοκιμάσουμε και χωρίς γραφικό περιβάλλον.

2.7: Ανάπτυξη Εφαρμογών Κονσόλας

Η ανάπτυξη εφαρμογών κονσόλας είναι αρκετά απλή. Θα χρησιμοποιήσουμε τις Reactive βιβλιοθήκες μαζί με τα Background Services που μας δίνει η πλατφόρμα .NET Core από τις βιβλιοθήκες Microsoft.Extensions.Hosting.

Αυτά μας επιτρέπουν να δημιουργήσουμε εφαρμογές που θα φορτώνουν ρυθμίσεις όπως οι διακομιστές μας αλλά θα εκτελούν διάφορους κώδικες σαν δουλείες στο υπόβαθρο.

Αυτές οι εφαρμογές είναι αρκετά συνήθεις και λέγονται κονσόλας διότι δεν έχουν κάποιο γραφικό περιχεόμενο απλά καταγράφουν την συμπεριφορά τους στην κονσόλα.

3: ΒΑΣΗ ΔΕΔΟΜΕΝΩΝ

Βάση δεδομένων, το πιο σημαντικό κομμάτι μιας εφαρμογής. Η βάση δεδομένων θα είναι το μέρος στο οποίο θα αποθηκεύονται όλα μας τα δεδομένα.

Υπάρχουν πολλές βάσεις δεδομένων οι οποίες χωρίζονται σε δυο μεγάλες κατηγορίες SQL (Structured Query Language) [47] και NoSQL (Not Only SQL) [48].

3.1: Βάσεις Δεδομένων SQL

Οι βάσεις SQL είναι Relational Databases (Σχεσιακές Βάσεις Δεδομένων) [49] αυτό σημαίνει ότι τα δεδομένα έχουν κάποια σχέση μεταξύ τους η και με το που αποθηκεύονται. Αυτό το καταλαβαίνουμε διότι τα δεδομένα μας αποθηκεύονται σε Tables (Τραπέζια) τα οποία έχουν κολώνες οπού η κάθε κολώνα έχει ένα όνομα και τύπο δεδομένου. Κάθε σειρά που εισχωρείτε στο τέλος του τραπεζιού είναι ένα ολοκληρωμένο αντικείμενο δεδομένων. Σχεσιακές δηλαδή σημαίνει ότι τα δεδομένα μας αποθηκεύονται με μια προκαθορισμένη δομή αλλιώς δεν μπορούμε να τα αποθηκεύσουμε.

Οι σχεσιακές βάσεις δεδομένων υπάρχον πολλά χρόνια και έχουν αρκετά πλεονεκτήματα, ένα από αυτά είναι η SQL ο οποία δίνει πολύ μεγάλη δύναμη για την αναζήτηση δεδομένων ενώ οι κολώνες προσφέρουν πόλους τύπους δεδομένων και έτσι βελτιστοποιούν τις διαδικασίες ανάγνωσης και αποθήκευσης. Ένα μεγάλο μειονέκτημα όμως είναι ότι πρέπει να γνωρίζουμε τι δεδομένα θα αποθηκεύσομε ενώ όταν θέλουμε να αλλάξουμε τις κολόνες ενός τραπεζιού θα πρέπει να εκτελέσουμε μια μετάβαση που συνήθως είναι μια πολύ μεγάλη και χρονοβόρα διαδικασία διότι πρέπει να είμαστε σίγουροί ότι δεν θα χάσουμε δεδομένα.

3.2: Βάσεις Δεδομένων NoSQL

Οι NoSQL βάσεις δεδομένων από την άλλη είναι πολύ γνωστές διότι είναι Schemaless (Χωρίς σχήμα) βάσεις δεδομένων. Αυτό σημαίνει ότι δεν χεριάζετε να δημιουργήσουμε κάποιο τραπέζι για να αποθηκεύσουμε τα δεδομένα μας με συγκεκριμένες κολώνες η δηλαδή ένα σχήμα, σε αυτές τις βάσεις απλά αποθηκεύουμε ότι αρχείο θέλουμε σε αρχεία τα οποία συνήθως είναι JSON [25] παρόλα αυτά μερικές βάσεις δημιουργούν κάποια επέκταση στα αρχεία JSON για να προσθέσουν δική τους λειτουργία.

Το πλεονέκτημα σε αυτήν την περίπτωση είναι ότι δεν χεριάζετε να ξέρουμε τι θα αποθηκεύσουμε για να το αποθηκεύσουμε ενώ όταν θέλουμε να αλλάξουμε το σχήμα της βάσης μας είναι αρκετά εύκολο, απλά κάνουμε μια ενημέρωση στην εφαρμογή που τα αποθηκεύει, από την άλλη ένα μειονέκτημα που έχουν αυτές οι βάσεις είναι ότι δεν υποστηρίζουν πάντα SQL και από βάση σε βάση ο τρόπος εύρεσης δεδομένων είναι αρκετά διαφορετικός με διαφορετικούς στόχους και σκοπούς.

3.3: Επιλογή Βάσης Δεδομένων

Η επιλογή μια βάσης δεδομένων δεν είναι πάντα εύκολη και ξεκάθαρη και στην προκείμενη περίπτωση για τις εφαρμογές μας κάναμε μια μεγάλη αλλαγή στην βάση μετά από σύντομο χρονικό διάστημα.

Για αρχή μερικές από τις πιο διάσημες βάσεις δεδομένων SQL είναι οι MySQL, Microsoft SQL Server, PostgreSQL ενώ από τις πιο διάσημες NoSQL βάσεις είναι οι MongoDB, Cassandra, CouchDB.

Εμείς στην αρχή επιλέξαμε την βάση δεδομένων PostgreSQL διότι είναι από τις πιο ανεπτυγμένες βάσεις με γρήγορες εγγραφές, γρήγορές αναζητήσεις και προσφέρει αρκετά τρίτα εργαλεία διαχείρισης. Παρόλα αυτά λόγο των γρήγορων αλλαγών της εφαρμογής μας μέχρι να φτάσουμε στο τελικό σχέδιο μας δημιούργησε πρόβλημα διότι περνάγαμε πιο πολύ χρόνο στην δημιουργία τραπέζιων και την ανάκτηση δεδομένων παρα στην ανάπτυξη της εφαρμογής μας ενώ κάποια ακόμα χαρακτηριστικά από την τελική επιλογή μας ώθησαν να αλλάξουμε.

Τελικά επιλέξαμε την βάση δεδομένων NoSQL CouchDB, μια από τις πιο εύχρηστες γρήγορες βάσεις δεδομένων NoSQL η οποία μας προσφέρει πολλά χρήσιμα χαρακτηριστικά τα οποία δεν περιμέναμε ότι θα θέλαμε από μια βάση δεδομένων.

3.4: CouchDB

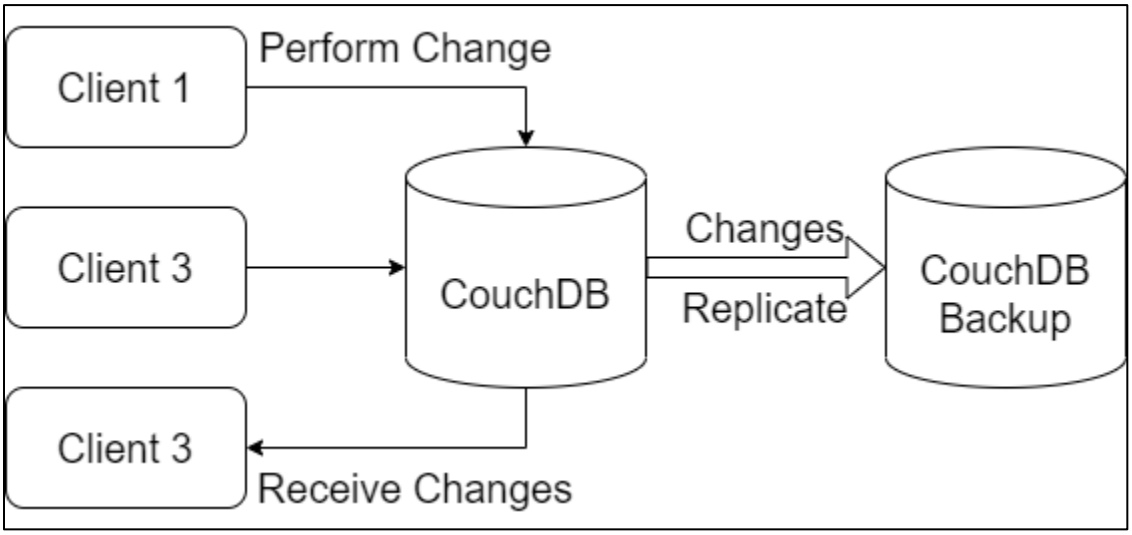
Η βάση δεδομένων CouchDB την οποία επιλέξαμε μας προσφέρει αρκετά χρήσιμα χαρακτηριστικά τόσο για την ανάπτυξη αλλά τόσο και την διαχείριση.

Ο πρώτος λόγος επιλογής ήταν ότι είναι μια NoSQL βιβλιοθήκη η οποία χρησιμοποιεί JSON αρχεία για να αποθηκεύει τα δεδομένα μας, αυτό μας δίνει την δυνατότητα αλλαγής τον δεδομένων μας στο μέλλον χωρίς να χεριάζετε να πειράξουμε την βάση μας, απλά φροντίζουμε στον κώδικα πως θα χειριζόμαστε τις τιμές οι οποίες δεν υπάρχον και φροντίζουμε να έχουν προεπιλεγμένες τιμές.

Ο δεύτερος λόγος είναι ότι η CouchDB στα αλήθεια είναι ένα ολόκληρο σύστημα DBMS (Database Management System), το οποίο έρχεται με ένα εύκολο και όμορφο γραφικό περιβάλλον που κατά την διαχείριση του πολύ εύκολη.

Τρίτος λόγος είναι ότι όλες οι αλλαγές είναι μοναδικές και από μονές τους σημειώνεται στην βάση δεδομένων χρησιμοποιώντας έναν μοναδικό αριθμό. Αυτός ο αριθμός μάλιστα ξεκινά με το πόσες φορές έχει γίνει αναβάθμιση και έχει την μορφή «10- …» αν για παράδειγμα η τιμή μας είχε αλλάξει 10 φορές. Επίσης καμία αλλαγή στα αλήθεια δεν είναι αλλαγή αφού μπορούμε να επιλέξουμε και να δούμε πιο παλιές εκδόσεις του αντικειμένου μας. Αυτό σημαίνει ότι και η διαγραφή ενός αντικειμένου στα αλήθεια απλά αφαιρεί το αντικείμενο από εμάς τους χρήστες ενώ στα αλήθεια δημιουργεί έναν νέο αριθμό και το μαρκάρει ως διαγραμμένο, αυτό το κάνει διότι η CouchDB δίνει την δυνατότητα συγχρονισμού της με οποιαδήποτε άλλη βάση CouchDB στον κόσμο ανεξάρτητος δικτύου.

Τέταρτος και τελευταίος λόγος έχει να κάνει με την ικανότητα της βάσης να συγχρονίζεται οπουδήποτε. Αυτό μάλιστα μας διευκόλυνε ώστε να χρησιμοποιήσουμε έναν λιγότερο διακομιστή. Η «changes» δυνατότητα μας επιτρέπει να παρακολουθήσουμε όλες τις αλλαγές που γίνανε στην βάση δεδομένων από κάποιο σημείο στον χρόνο για παράδειγμα πριν δέκα μέρες ως σήμερα μετά συνεχίζει να μας στέλνει τις αλλαγές όπως γίνονται σε πραγματικό χρόνο. Αυτό μας δίνει την δυνατότητα να χτίσουμε εφαρμογές οι οποίες δεν χρειάζεται να επικοινωνούν μεταξύ τους ακόμα και για αλλαγές που γίνονται στην βάση δεδομένων αφού η ίδια απλά θα ενημερώσει τον διακομιστή μέσα από το «changes» χαρακτηριστικό.



Εικόνα 49: CouchDB Changes

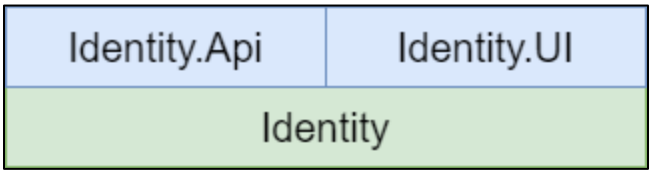
Επίσης η CouchDB μας δίνει έναν ωραίο τρόπο να προκαθορίσουμε διάφορα ευρετήρια για τα δεδομένα μας όταν αποφασίσουμε ποια είναι αυτά. Αυτό είναι το μόνο μειονέκτημα μέχρι στιγμής ότι δεν μπορούμε να κάνουμε αναζήτηση οπότε θέλουμε για κάποιο αντικείμενο. Το πλεονέκτημα είναι ότι επειδή δημιουργούμε αυτούς τους κανόνες μοναδικά για κάθε περίπτωση η βάση κρατά ένα καλά ενημερωμένο ευρετήριο το οποίο είναι και σε σειρά, για παράδειγμα το 2 θα είναι πάντα μετρά το 1 και ου το καθεξής, αυτό δίνει την δυνατότητα αιεύρεσης δεδομένων σε παρα πολύ μικρούς χρόνους.

Τέλος όλο το API της βάσης εκτελείτε μέσω HTTP/REST APIs οπότε δεν χρειαζόμαστε κάποιο εργαλείο μπορούμε να την χρησιμοποιούσε σαν να επισκεπτόμασταν κάποια ιστοσελίδα αν και αυτό δεν θα μας σύμφερε κάνει τον προγραμματισμό πολύ πιο εύκολο. Έτσι ο τρόπος διαχείρισης της γίνετε στέλνοντας διάφορα JSON αρχεία σε συγκεκριμένες διευθύνσεις και λαμβάνοντας πίσω JSON αρχεία σαν απάντηση με τα αποτελέσματα μας.

4: ΤΑΥΤΟΠΟΙΗΣΗ

Η εφαρμογή της ταυτοποιήσεις αποτελείτε σύνολο από τρία προτζεκτ. Το πρώτο ονομάζεται Identity και περιέχει μέσα βασικό κώδικα για την χρήση στον διακομιστή και στον περιηγητή.

Εφαρμόζει τις τεχνικές που είδαμε πιο πριν CQRS με την βιβλιοθήκη MediatR και Vertical Slice Architecture ενώ για την επίτευξη της ταυτοποίησης της αποθήκευση χρηστών και τον διαχειρισμό για τα Token χρησιμοποιούμε τις βιβλιοθήκες «`Microsoft.Extensions.Identity.Stores`» και «`OpenIddict.Core`» μάλιστα κατά την δημιουργία του προτζεκτ διαπιστώθηκε ότι δεν υπήρχαν βιβλιοθήκες για να αποθηκεύουν σε CouchDB έτσι χάρη σε αυτήν την εργασία δημιουργήσαμε και δυο βιβλιοθήκες ανοιχτού κώδικα την «`AspNetCore.Identity.CouchDB`» και «`P41.OpenIddict.CouchDB`» για να λύσουν το πρόβλημα αυτό ενώ οποιοσδήποτε θα μπορεί να τις χρησιμοποίησή αντί να χρειάζεται να δημιουργήσει δική του λύση.



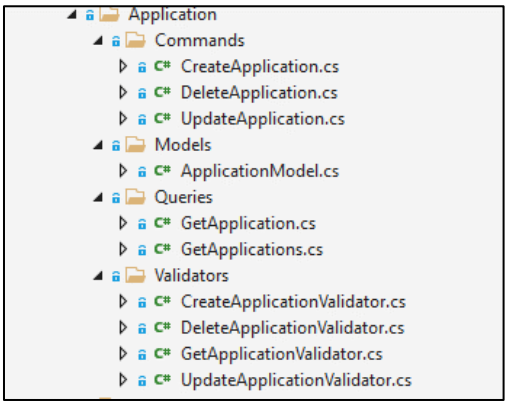
Εικόνα 50: Identity προτζεκτ.

4.1: Προτζεκτ Identity

Η βασική βιβλιοθήκη διαθέτει τέσσερίς φακέλους όπως φαίνεται και στην επόμενη εικόνα. Ο κάθε φάκελος αντιπροσωπεύει μια δυνατότητα της εφαρμογής,.

4.1.1: Φάκελος Application

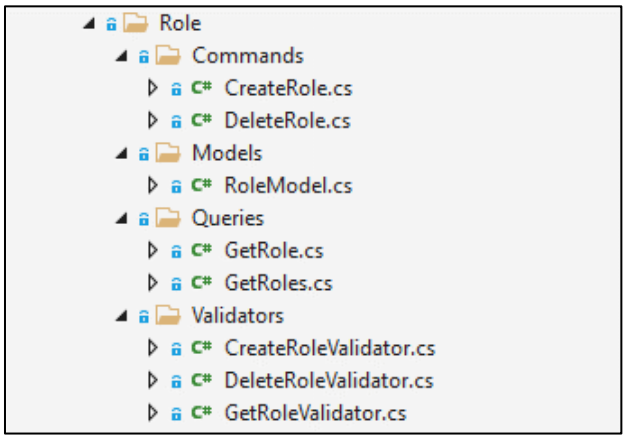
Αντιπροσωπεύει τις εφαρμογές μας οι οποίες μπορούν να τακτοποιήσουν κάποιο χρήστη. Στον φάκελο Commands έχουμε εντολές δημιουργίας ενημέρωσης και διαγράφης, στον φάκελο Models έχουμε το αντικείμενο το οποίο θα χρησιμοποιούμε για την μεταφορά, στο φάκελος Queries έχουμε αντικείμενα αναζήτησης και στον φάκελο Validators έχουμε αντικείμενα που κάνουν έλεγχο στα αντικείμενα στους φακέλους Commands και Queries.



Εικόνα 51: Identity προτζεκτ, Application

4.1.2: Φάκελος Role

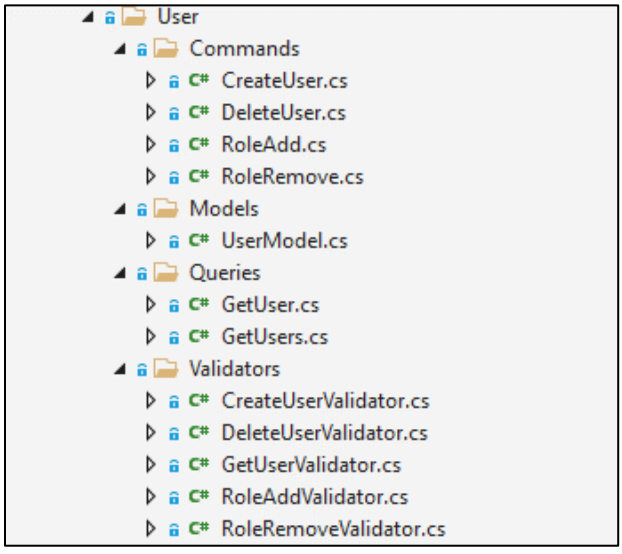
Αντιπροσωπεύει τους ρόλους που μπορεί να έχει ένας χρήστης, για παράδειγμα θα μπορούσε να είναι διαχειριστής. Στον φάκελο Commands έχουμε εντολές δημιουργίας και διαγράφης, στον φάκελο Models έχουμε αντικείμενα κατάλληλα για μεταφορά, στον φάκελο Queries έχουμε αντικείμενα αναζήτησης και στον φάκελο Validators έχουμε αντικείμενα ελέγχου των φακέλων Commands και Queries.



Εικόνα 52: Identity προτζεκτ, Role

4.1.3: Φάκελος User

Αντιπροσωπεύει τους χρήστες. Ο φάκελος Commands περιχέει εντολές δημιουργίας, διαγράφης, προσθήκης ενός ρολού και διαγράφης του στον χρήστη. Ο φάκελος Models περιέχει ένα αντικείμενο μεταφοράς στοιχειών ενός χρήστη, ο φάκελος Queries περιέχει αντικείμενα αναζήτησης και ο φάκελος Validators περιέχει αντικείμενα ελέγχου για τους φακέλους Commands και Queries.

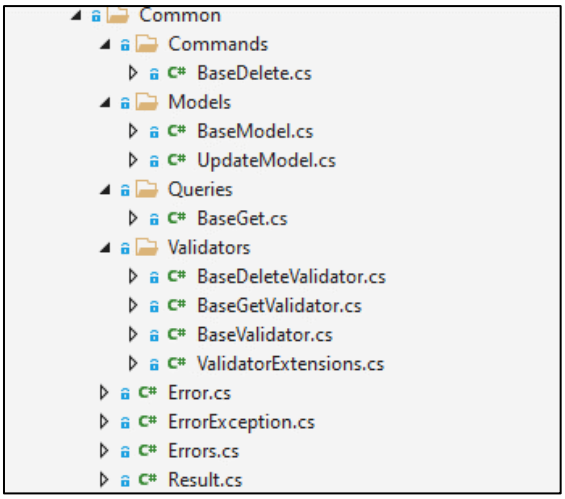


Εικόνα 53: Identity προτζεκτ, User

4.1.4: Φάκελος Common

Διαθέτει κοινό κώδικα που μπορεί να χρειαστούμε στην εφαρμογή μας. Ο φάκελος Commands περιχέει αντικείμενο σαν βάση για αντικείμενα διαγράφης, ο φάκελος Models περιέχει αντικείμενα για βασική μεταφορά και δεδομένων ενημέρωσης κάποιου αντικειμένου, ο φάκελος Queries περιχέει βασικό αντικείμενο για την δημιουργία αντικειμένων αναζήτησης και ο φάκελος Validators περιέχει βασικές εντολές και αντικείμενα για την εύκολη δημιουργία αντικειμένων ελέγχου.

Επίσης στον φάκελο Common περιέχονται βασικά αντικείμενά όπως το Error το οποίο περιγραφή ένα σφάλμα, για παράδειγμα τον τίτλο και τον λόγο που συνέβη, ένα αντικείμενο σφάλματος της C# που ονομάζετε Exception έτσι και το δικό μας λέγετε ErrorException διότι διαθέτει ένα αντικείμενο Error. Ένα αντικείμενο Errors που διαθέτη έτοιμα κοινά σφάλματα όπως «Αυτό το αντικείμενο δεν βρέθηκε.» και τέλος ένα αντικείμενο Result το οποίο η θα μας δώσει το επιθυμητό αποτέλεσμα για παράδειγμα έναν χρήστη UserModel η κάποιο Error το οποίο περιγράφει τι πήγε λάθος.



Εικόνα 54: Identity προτζεκτ, Common

4.2: Προτζεκτ Identity API

Αυτό το προτζεκτ θα είναι ο διακομιστής υπεύθυνος για την ταυτοποίηση ενώ θα χειρίζεται τους χρήστες τους ρόλους και της εφαρμογές.

Επίσης θα έχει μια μοναδική σελίδα στην οποία οι χρήστες θα μπορούν να κάνουν είσοδο.

Θα δημιουργήσουμε χειριστές για κάθε διαθέσιμη αίτηση του βασικού προτζεκτ Identity για τις εντολές και τις αναζητήσεις.

Παρακάτω βλέπουμε την αρχή της εφαρμογής Identity API. Η εφαρμογή δηλώνει διάφορες υπηρεσίες που θα χρειαστεί όπως, κώδικες ελέγχουμε FluentValidation, κώδικες χειρισμού αιτήσεων MediatR και κώδικα που εκτελείτε πριν και μετά από κάθε τις αίτηση.

Περεταίρω ρυθμίζει τον «πελάτη» που χρησιμοποιούμε για την επικοινωνία με την βάση δεδομένων CouchDB και μερικές ρυθμίσεις για την ταυτοποίηση όπως να χρησιμοποιεί σαν βάση δεδομένων την CouchDB.

```
21      await new HostBuilder()
22      .ConfigureCoreHost(args)
23      .ConfigureWebHostDefaults(options => options
24      .ConfigureServices((c, services) =>
25      {
26          var configuration = c.Configuration;
27
28          services.AddCoreHostServices();
29          services.AddValidatorsFromAssembly(Assembly.GetAssembly(typeof(ValidatorExtensions)));
30          services.AddMediatR(c => c.AsSingleton(), typeof(Worker).Assembly);
31          services.AddTransient(typeof(IPipelineBehavior<,>), typeof(RequestValidationBehavior<,>));
32          services.AddTransient(typeof(IPipelineBehavior<,>), typeof(UnhandledExceptionBehavior<,>));
33
34          // Configure CouchDb client.
35          services.AddSingleton<ICouchClient>(new CouchClient(o => o
36              .Configure(configuration)));
37
38          // Configure Identity stores and services.
39          services.AddIdentity<CouchDbUser, CouchDbRole>()
40              .AddCouchDbStores()
41              .SetDatabaseName(configuration.GetCouchDatabase());
42
43          // Configure Identity to use the same JWT claims as OpenIddict instead
44          // of the legacy WS-Federation claims it uses by default (ClaimTypes),
45          // which saves you from doing the mapping in your authorization controller.
46          services.Configure<IdentityOptions>(options =>
47          {
48              options.ClaimsIdentity.UserNameClaimType = Claims.Name;
49              options.ClaimsIdentity.UserIdClaimType = Claims.Subject;
50              options.ClaimsIdentity.RoleClaimType = Claims.Role;
51          });
```

Εικόνα 55: Identity API προτζεκτ, Program.cs 1

Στην εικόνα παρακάτω βλέπουμε την πλήρη ρυθμίσει την βιβλιοθήκης Openiddict η οποία χειρίζεται την ταυτοποίηση των χρηστών σε εφαρμογές μας μαζί με την βιβλιοθήκη “AspNet.Core.Identity”.

```
53      services.AddOpenIddict()
54          // Register the OpenIddict core components and to use the CouchDb stores and models.
55          .AddCore(options => options
56              .UseCouchDb()
57              .SetDatabaseName(configuration.GetCouchDatabase()))
58          // Register the OpenIddict server components.
59          .AddServer(options =>
60          {
61              // Enable the authorization, logout, token and userinfo endpoints.
62              options.SetAuthorizationEndpointUris("/connect/authorize")
63                     .SetLogoutEndpointUris("/connect/logout")
64                     .SetTokenEndpointUris("/connect/token")
65                     .SetUserinfoEndpointUris("/connect/userinfo");
66
67              // Mark the "email", "profile" and "roles" scopes as supported scopes.
68              options.RegisterScopes(Scopes.Email, Scopes.Profile, Scopes.Roles);
69
70              // Set token lifetime.
71              options.SetAccessTokenLifetime(TimeSpan.FromHours(1))
72                     .SetIdentityTokenLifetime(TimeSpan.FromHours(24))
73                     .SetRefreshTokenLifetime(TimeSpan.FromDays(90));
74
75              // Note: the sample uses the code and refresh token flows but you can enable
76              // the other flows if you need to support implicit, password or client credentials.
77              options.AllowAuthorizationCodeFlow()
78                     .AllowRefreshTokenFlow();
79
80              options.DisableAccessTokenEncryption();
81
82              // Register the signing and encryption credentials.
83              options.AddDevelopmentSigningCertificate()
84                     .AddEncryptionKey(new SymmetricSecurityKey(Convert.FromBase64String(configuration.GetSecret("jwt"))));
85
86              // Register the ASP.NET Core host and configure the ASP.NET Core-specific options.
87              options.UseAspNetCore()
88                     .EnableAuthorizationEndpointPassthrough()
89                     .EnableLogoutEndpointPassthrough()
90                     .EnableStatusCodePagesIntegration()
91                     .EnableTokenEndpointPassthrough();
92          })
93          // Register the OpenIddict validation components.
94          .AddValidation(options =>
95          {
96              options.UseLocalServer(); // Import the configuration from the local OpenIddict server instance.
97              options.UseAspNetCore(); // Register the ASP.NET Core host.
98          });
```

Εικόνα 56: Identity API προτζεκτ, Program.cs 2

Τέλος θέτουμε μερικές υπηρεσίες που εκτελούνται στο υπόβαθρο ενώ η εντολή AddControllersWithView ρυθμίζει τόσο το RestAPI μας όσο και την δημιουργία σελίδων όπως αυτή της εισόδου ενός χρήστη.

```
100     services.AddControllersWithViews();
101     services.AddRazorPages();
102
103     services.AddWorkers();
104
105     // Register swagger services only if swagger is enabled.
106     services.TryAddSwagger("Identity", "1", configuration);
107 })
```

Εικόνα 57: Identity API προτζεκτ, Program.cs 3

Ο παρακάτω κώδικας εκτελείτε όταν η εφαρμογή είναι έτοιμη να ξεκινήσει. Θέτει μερικές ρυθμίσεις όπως όταν είμαστε σε «Development» να μας δείχνει σελίδες με αναλυτικά σφάλματα, να ανακατευθύνει τους χρήστες σε HTTPS, να μπορεί να κάνει «Host» σελίδες SPA και τέλος να αντιστοιχήσει τα σημεία RestAPI της εφαρμογής μας.

```
.Configure((c, app) =>
{
    var configuration = c.Configuration;
    var enviroment = c.HostingEnvironment;

    if (enviroment.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseWebAssemblyDebugging();
    }
    else
    {
        app.UseExceptionHandler("/error");
        app.UseHsts();
    }

    // All the calls after this will always try to redirct to https.
    // For better security modify the Urls in appsettings to use only https endpoints.
    app.UseHttpsRedirection();

    // Enable static files.
    app.UseBlazorFrameworkFiles();
    app.UseStaticFiles();

    // Enable swagger generation and endpoints only when enabled in settings.
    app.TryUseSwagger();

    app.UseSerilogRequestLogging();

    app.UseRouting();
    app.UseCors(builder =>
    {
        builder.AllowAnyOrigin();
        builder.AllowAnyMethod();
        builder.AllowAnyHeader();
    });
    app.UseAuthentication();
    app.UseAuthorization();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
        endpoints.MapRazorPages();
        endpoints.MapFallbackToFile("index.html");
    });
}))
 .Build()
 .RunCoreHostAsync("Identity");
```

Εικόνα 58: Identity API προτζεκτ, Program.cs 4

Οι παρακάτω ρυθμίσεις χρησιμοποιούνται από το εργαλείο YARN, αφού έχουμε τρέξει την εντολή «yarn» μπορούμε να τρέξουμε την εντολή «yarn run cssbuild» για να δημιουργήσουμε τον απαραίτητο κώδικα CSS για τις διαδικτυακές μας εφαρμογές Identity.

```
{
  "name": "identity",
  "description": "",
  "license": "",
  "repository": "",
  "devDependencies": {
    "autoprefixer": "^10.0.2",
    "cssnano": "^5.0.1",
    "postcss": "^8.2.1",
    "postcss-cli": "^8.3.0",
    "postcss-import": "^14.0.1",
    "tailwindcss": "^2.1.1"
  },
  "postcss": {
    "plugins": {
      "postcss-import": {},
      "tailwindcss": {},
      "autoprefixer": {},
      "cssnano": {}
    }
  },
  "scripts": {
    "csswatch": "postcss ./wwwroot/app.css -o ./wwwroot/app.min.css --watch",
    "cssbuild": "postcss ./wwwroot/app.css -o ./wwwroot/app.min.css",
    "csspublish": "set NODE_ENV=production&& postcss ./wwwroot/app.css -o ./wwwroot/app.min.css"
  },
  "version": "0.1.0"
}
```

Εικόνα 59: Identity API προτζεκτ, package.json

Οι παρακάτω ρυθμίσεις καθοδηγούν το εργαλείο TailwindCSS για το ποιες σελίδες κώδικα περιέχουν CSS στοιχεία όπως “class” για να μην τα αφαιρέσει από το τελικό αρχείο CSS.

```
module.exports = {
    purge: [
        './**/*.html',
        '.**/*.cshtml',
        '../Identity.UI/**/*.html',
        '../Identity.UI/**/*.cshtml',
    ],
    darkMode: false, // or 'media' or 'class'
    plugins: [],
}
```

Εικόνα 60: Identity API προτζεκτ, tailwind.config.js

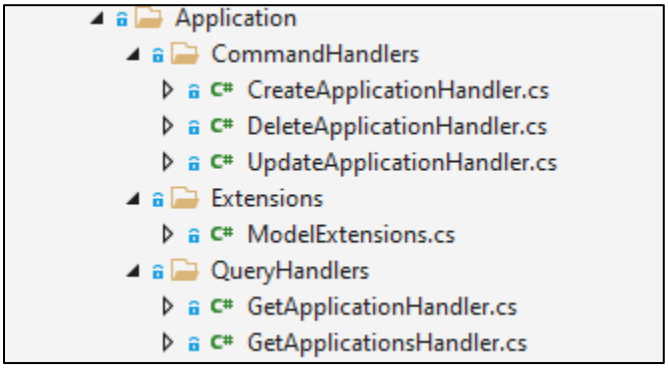
Οι παρακάτω ρυθμίσεις χρησιμοποιούνται από την εφαρμογή για να της δώσουμε τα σημεία της βάσης δεδομένων, της αρχικής εφαρμογής διαχείρισης με όνομα Blazor και το JWT «κωδικό» από τον οποίο θα δημιουργούνται τα διαφορά «Token» ταυτοποίησης.

```
Urls: https://*:5021

Swagger:
  Enabled: false

Services:
  Blazor: <The full Blazor application url>
  CouchDb: <The couchdb username;password;url;db example => admin;admin;http://localhost:5984;my_database (db is optional)>

Secrets:
  JWT: <JWT Secret here>

Serilog:
  MinimumLevel:
    Default: Information
    Override:
      System: Warning
      Microsoft: Warning
      Microsoft.Hosting.Lifetime: Information
  WriteTo:
    - {Name: Console}
    - {Name: File, Args: {Path: "logs/logs.txt", RollingInterval: Day}}

AllowedHosts: "*"
```

Εικόνα 61: Identity API προτζεκτ, appsettings.yaml

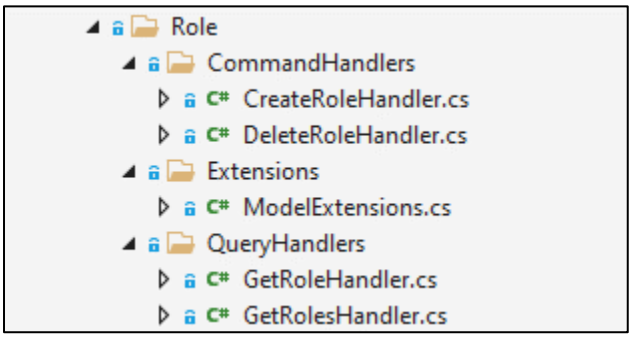
4.2.1: Περιγραφή Φακέλων

Ο φάκελος Application περιέχει όλες τις εντολές δημιουργίας ενημέρωσης διαγράφης και αναζήτησης για τις διαθέσιμες εφαρμογές ταυτοποίησης μας οι οποίες εδώ είναι μια, μαζί περιέχει και μερικές μεθόδους μετατροπής στο αρχείο ModelExtensions.cs.



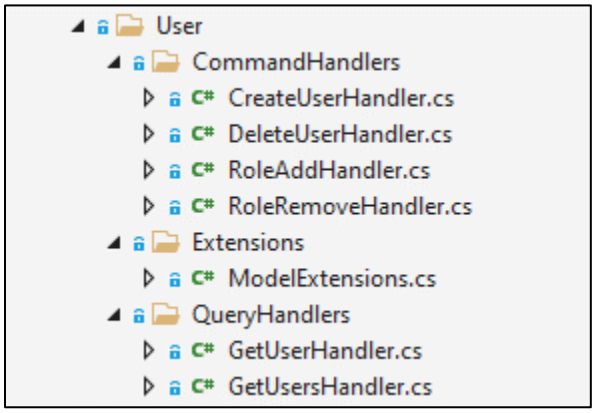
Εικόνα 62: Identity API προτζεκτ, Application

Ο φάκελος Role περιέχει όλες τις εντολές δημιουργίας ενημέρωσης διαγράφης και αναζήτησης για τις διαθέσιμων ρολών στους οποίους ένας χρήστης μπορεί να προστεθεί. Εμάς η εφαρμογή διαθέτει μόνο δυο τους «Admin» και «User», μαζί περιέχει και μερικές μεθόδους μετατροπής στο αρχείο ModelExtensions.cs.



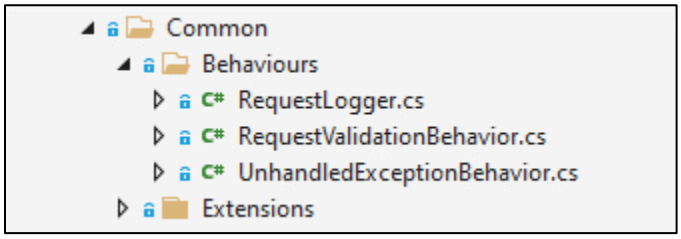
Εικόνα 63: Identity API προτζεκτ, Role

Ο φάκελος User περιέχει όλες τις εντολές δημιουργίας ενημέρωσης διαγράφης και αναζήτησης για τους χρήστες, μαζί περιέχει και μερικές μεθόδους μετατροπής στο αρχείο ModelExtensions.cs.



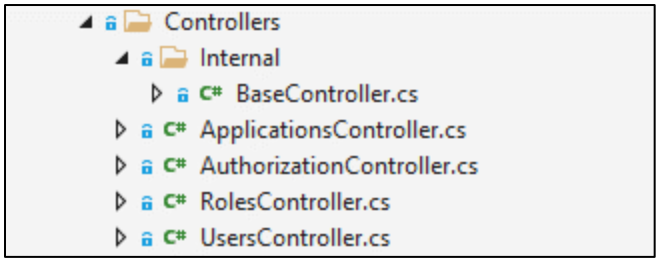
Εικόνα 64: Identity API προτζεκτ, User

Ο φάκελος Common περιέχει κοινό κώδικα τον οποίο χρησιμοποιούμε σε όλο το προτζεκτ.



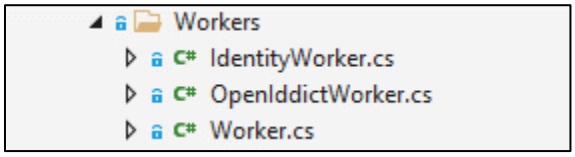
Εικόνα 65: Identity API προτζεκτ, Common

Ο φάκελος Controller περιέχει κώδικες οι οποίοι με λίγες γραμμές δημιουργούν όλο το RestAPI της εφαρμογής μας και προωθούν τις εντολές που λαμβάνουν στην βιβλιοθήκη MediatR.



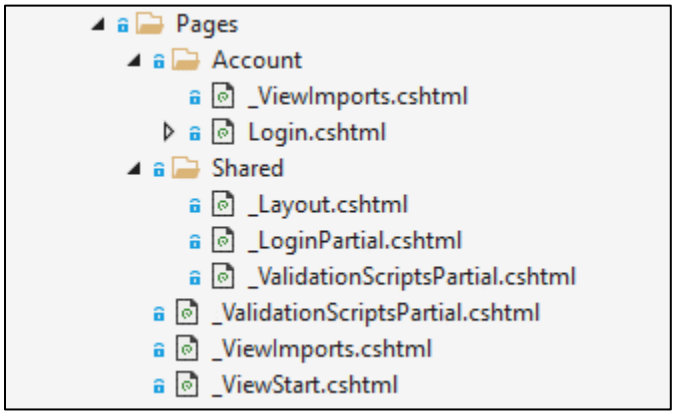
Εικόνα 66: Identity API προτζεκτ, Controllers

Στον φάκελο Workers βλέπουμε κώδικα υπηρεσιών ο οποίος εκτελείτε μόνο όταν η εφαρμογής μας εκτελείτε για πρώτη φορά μετά δεν ξανά εκτελείτε. Το μόνο που κάνει είναι να προ τοποθετήσει μερικά αντικείμενα στην βάση δεδομένων μας όπως χρήστες και ρόλους.



Εικόνα 67: Identity API προτζεκτ, Workers

Ο φάκελος Pages δημιουργημένος από το εργαλείο ανάπτυξης κώδικα Visual Studio περιέχει έναν πρότυπο κώδικα στον οποίο κάναμε πολύ μικρές μετατροπές για να εμφανίσουμε την σελίδα εισόδου του χρήστη μας η οποία ορίζετε στο αρχείο Login.cshtml. Όλα τα αλλά αρχεία τα αφήσαμε όπως δημιουργήθηκαν.



Εικόνα 68: Identity API προτζεκτ, Pages

4.3: Προτζεκτ Identity UI

Αυτό το προτζεκτ είναι μια εφαρμογή SPA Blazor. Εδώ δημιουργήσαμε μια απλή εφαρμογή χρησιμοποιώντας όλες τις βασικέ βιβλιοθήκες Web που είδαμε σε προηγούμενα κεφαλιά.

Έτσι επιτυγχάνουμε να έχουμε ένα όμορφο περιβάλλον χρήστη. Πέρα από τους βασικούς φακέλους θα δούμε και μερικά αρχεία τα οποία είναι απαραίτητα για να λειτουργήσει η εφαρμογή μας.

Το _imports.razor, αυτό το αρχείο χρησιμοποιείτε για να πούμε στην εφαρμογή ότι θα έχουμε κάποια κοινά namespaces, δηλαδή σε κάθε φάκελο θα χρησιμοποιούμε κώδικα από αυτά τα namespaces χωρίς να χρειαστεί να τα δηλώσουμε σε κάθε αρχείο κώδικα ξεχωριστά.

Το αρχείο App.razor είναι το αντικείμενο της εφαρμογής μας, όταν η εφαρμογή ξεκινάει αντικαθιστά στον κώδικα html του περιηγητή αυτό τα αρχείο και από εκεί και πέρα λειτουργεί ο δικός μας κώδικας.

Στην αρχή χρησιμοποιούμε το αντικείμενο CascadingAuthenticationState αυτό θα μας κάνει διαθέσιμο στην εφαρμογή το αν ο χρήστης έχει κάνει είσοδο η όχι. Μετά τα αντικείμενα Router, Found και NotFound φροντίζουν να δείξουν ανάλογο περιχεόμενο με το αν ο χρήστης κάνει περιήγηση σε έναν σύνδεσμο ο οποίος δεν υπάρχει.

Τέλος με το AuthorizeRouteView ελέγχουμε αν ο χρήστης είναι συνδεδεμένος, αν όχι τον κατευθύνουμε να κάνει είσοδο αλλιώς του εμφανίζουμε το μήνυμα ότι δεν έχει το δικαίωμα να δει την συγκεκριμένη σελίδα.

```
1   @using LetItGrow.Identity.Common.Layouts
2
3   <CascadingAuthenticationState>
4       <Router AppAssembly="typeof(MainLayout).Assembly">
5           <Found Context="routeData">
6               <AuthorizeRouteView RouteData="routeData" DefaultLayout="typeof(MainLayout)">
7                   <NotAuthorized>
8                       @if (context.User.Identity is { IsAuthenticated: false })
9                       {
10                          <RedirectToLogin />
11                      }
12                      else
13                      {
14                          <LayoutView Layout="typeof(IdentityLayout)">
15                              <p>You are not authorized to access this resource.</p>
16                          </LayoutView>
17                      }
18                  </NotAuthorized>
19              </AuthorizeRouteView>
20          </Found>
21          <NotFound>
22              <p>Sorry, there's nothing at this address.</p>
23          </NotFound>
24      </Router>
25  </CascadingAuthenticationState>
```

Εικόνα 69: Identity UI προτζεκτ, App.razor

Το αρχείο Program.cs είναι το πρώτο αρχείο που εκτελείτε, εδώ ρυθμίζουμε την εφαρμογής μας. Ξεκινάμε δημιουργώντας ένα αντικείμενο με όνομα builder το οποίο θα «κτίσει» την εφαρμογή μας. Στην αρχή βλέπουμε ότι ρυθμίζουμε το αντικείμενο «app» στην html να αντικατασταθεί με το αντικείμενο App.razor.

Στην συνέχεια ρυθμίζουμε βασικές λειτουργίες της βασικής βιβλιοθήκης ενώ δηλώνουμε κάποια βοηθητικά αντικείμενα τα οποία θα μας βοηθήσουμε στην ανάκτηση δεδομένων από τον διακομιστή.

Με την εντολή AddOidcAuthentication ρυθμίζουμε την συμπεριφορά ταυτοποίησης του χρήστη από τις ρυθμίσεις appsettings.json.

Τέλος καλούμε της εντολές Build() και RunAsync() ώστε να ξεκινήσει η εφαρμογή μας.

```
public static Task Main(string[] args)
{
    var builder = WebAssemblyHostBuilder.CreateDefault(args);
    builder.RootComponents.Add<App>("app");

    var services = builder.Services;
    var configuration = builder.Configuration;

    services.AddCoreWeb();
    services.AddSingleton<IFlurlClient>(sp => new FlurlClient(builder.HostEnvironment.BaseAddress));
    services.AddSingleton<ApplicationService>();
    services.AddSingleton<UserService>();
    services.AddSingleton<RoleService>();
    services.AddSingleton<IMediaQueryService, MediaQueryService>();

    FlurlHttp.Configure(config =>
    {
        config.JsonSerializer = new SystemJsonSerializer();
    });

    services.AddOidcAuthentication(options =>
    {
        options.UserOptions.RoleClaim = "role";

        // Note: response_mode=fragment is the best option for a SPA. Unfortunately, the Blazor WASM
        // authentication stack is impacted by a bug that prevents it from correctly extracting
        // authorization error responses (e.g error=access_denied responses) from the URL fragment.
        // For more information about this bug, visit https://github.com/dotnet/aspnetcore/issues/28344.
        configuration.Bind("OpenId", options.ProviderOptions);
    });

    return builder.Build().RunAsync();
}
```

Εικόνα 70: Identity UI προτζεκτ, Program.cs

4.3.1: Φάκελος Wwwroot

Το αρχείο html που φορτώνει στη αρχή ο περιηγητής. Παρατηρούμε το αντικείμενο app το οποίο στην αρχή δείχνει ένα αντικείμενο φόρτωσης και στο τέλος τα δυο script αντικείμενα τα οποία δίνουν τόσο την λειτουργικότητα της εφαρμογής όσο και της ταυτοποίησης.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0, user-scalable=no" />
    <title>Identity</title>
    <base href="/" />
    <link rel="preload" as="style" href="https://cdn.jsdelivr.net/npm/water.css@2/out/light.min.css">
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/water.css@2/out/dark.min.css" id="theme-stylesheet">
    <link rel="stylesheet" href="app.min.css" />
    <script src="_content/LetItGrow.CoreWeb/theme.js"></script>
</head>

<body>
    <app>
        <div class="fixed inset-0 flex items-center justify-center">
            <div class="spinner">
                <div class="rect1"></div>
                <div class="rect2"></div>
                <div class="rect3"></div>
                <div class="rect4"></div>
                <div class="rect5"></div>
            </div>
        </div>
    </app>

    <div id="blazor-error-ui">
        An unhandled error has occurred.
        <a href="" class="reload">Reload</a>
        <a class="dismiss">✕</a>
    </div>

    <script src="_content/Microsoft.AspNetCore.Components.WebAssembly.Authentication/AuthenticationService.js"></script>
    <script src="_framework/blazor.webassembly.js"></script>
</body>
</html>
```

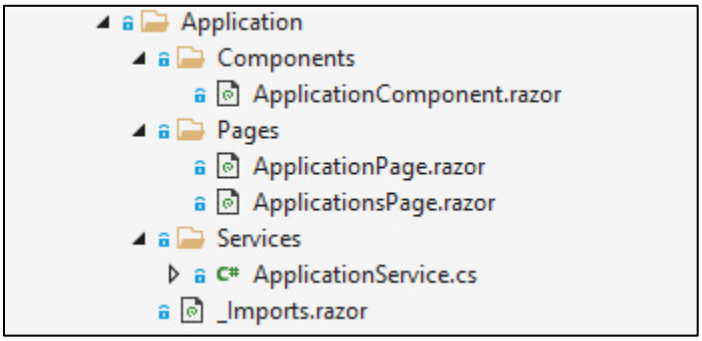
Εικόνα 71: Identity UI προτζεκτ, wwwroot, index.html

Αρχείο ρυθμίσεων της εφαρμογής, από εδώ μπορούμε να αλλάξουμε κάποιες βασικές ρυθμίσεις στην εφαρμογή μας χωρίς να χρειάζεται να γράψουμε η να αλλάξουμε κώδικα, οι συγκεκριμένες ρυθμίσεις έχουν να κάνουν με την ταυτοποίηση του χρήστη.

```
{
  "OpenId": {
    "ClientId": "identity",
    "Authority": "<The full url of the identity service>",
    "PostLogoutRedirectUri": "<The url of this app>",
    "ResponseType": "code",
    "ResponseMode": "query",
    "DefaultScopes": [ "roles" ]
  }
}
```

Εικόνα 72: Identity UI προτζεκτ, wwwroot, appsettings.json

4.3.2: Φάκελος Application



Εικόνα 73: Identity UI προτζεκτ, Application

Κοινά namespaces τα οποία θα χρειαστούν οι εσωτερικοί κώδικες στον φάκελο Application.

```
@using LetItGrow.Identity.Application.Components
@using LetItGrow.Identity.Application.Models
@using LetItGrow.Identity.Application.Services
@using LetItGrow.Identity.Application.Validators
```

Εικόνα 74: Identity UI προτζεκτ, Application, _imports.razor

Βασικό αντικείμενο το οποίο φροντίζει να δήξει στο χρήστη ένα αντικείμενο Application.

```
@inherits BaseComponent

<div class="@css">
    <h2 class="p-1 my-1">@Model.ClientId</h2>
    <p class="p-1 my-1">@Model.DisplayName</p>
    @foreach (var uri in Model.RedirectUris)
    {
        <p class="my-1 px-4 py-1 rounded-xl border bg-blue-500 text-gray-100">@uri</p>
    }
    @foreach (var uri in Model.PostLogoutRedirectUris)
    {
        <p class="my-1 px-4 py-1 rounded-xl border bg-blue-500 text-gray-100">@uri</p>
    }
</div>

@code{
    [Parameter]
    public ApplicationModel Model { get; set; } = null!;

    protected override void OnParametersSet()
    {
        if (Model is null) throw new ArgumentNullException(nameof(Model));

        css = CreateCss("border shadow p-2")
            .AddClass(Class);
    }
}
```

Εικόνα 75: Identity UI προτζεκτ, Application, ApplicationComponent.razor

Σελίδα στην οποία οδηγείτε ο χρήστης για ένα συγκεκριμένο αντικείμενο Application, το οποίο βρίσκετε σε σύνδεσμο με την μορφή «/app/id». Η σελίδα φροντίζει να είναι διαθέσιμη μόνο σε χρήστες με δικαιώματα διαχειριστή και ανάλογα με το αν κατάφερε να βρει το αντικείμενο Application από το ID θα δήξει ένα ανάλογο μήνυμα.

```
@page "/app/{Id}"
@attribute [Authorize(Roles = "Admin")]
@inject ApplicationService service

@if (Id is null)
{
    <NotFound />
}
else if (error is not null)
{
    <ErrorComponent Error="error" />
}
else if (model is not null)
{
    <ApplicationComponent Model="model" />
}
else
{
    <Loading />
}

@code{
    [Parameter]
    public string? Id { get; set; }

    private Error? error;
    public ApplicationModel? model;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (!firstRender || Id is null) return;

        (await service.GetAsync(Id))
        .Switch(
            r => model = r,
            e => error = e
        );
    }
}
```

Εικόνα 76: Identity UI προτζεκτ, Application, ApplicationPage.razor

Σελίδα στην οποία ο χρήστης βλέπει όλα τα διαθέσιμα αντικείμενα Application η οποία είναι διαθέσιμη στην διεύθυνση «/app». Η σελίδα φροντίζει να δήξει οποιοδήποτε σφάλμα αν υπάρχει αλλιώς για κάθε εφαρμογή που βρήκε θα την δήξει χρησιμοποιώντας αντικείμενα ApplicationComponent. Η σελίδα αυτή είναι διαθέσιμη μόνο για διαχειριστές.

```
@page "/app"
@attribute [Authorize(Roles = "Admin")]
@inject ApplicationService service

<div class="flex justify-between">
    <p>These are the available applications.</p>
    @*<button>Add</button>*@
</div>

@if (error is not null)
{
    <ErrorComponent Error="error" />
}
else if (models is not null)
{
    @* todo: Add delete button *@
    <Virtualize Items="models" Context="model">
        <ApplicationComponent Class="m-2" Model="model" />
    </Virtualize>
}
else
{
    <div class="flex items-center justify-center">
        <Loading />
    </div>
}

@code{
    private Error? error;
    private List<ApplicationModel>? models;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (!firstRender) return;

        (await service.GetAllAsync())
        .Switch(
            r => models = new(r),
            e => error = e
        );

        StateHasChanged();
    }
}
```

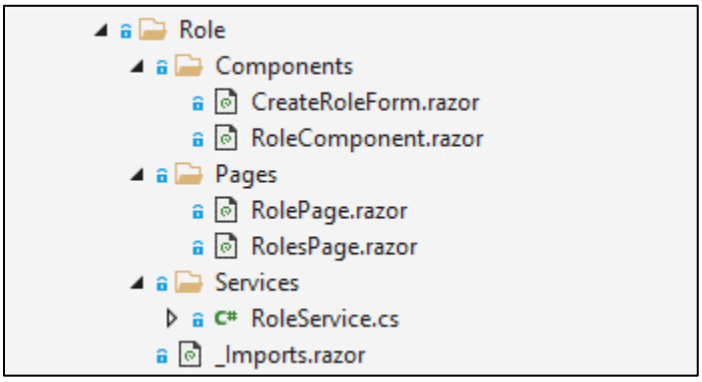
Εικόνα 77: Identity UI προτζεκτ, Application, ApplicationsPage.razor

Κώδικας τον οποίο χρησιμοποιούν οι σελίδες και είναι υπεύθυνος για την φόρτωση αλλά και εκτέλεση εντολών στον διακομιστή. Μόλις ολοκληρωθεί κάποια εντολή επιστρέφει ένα αντικείμενο Result το οποίο η θα έχει τα αποτελέσματα τα οποία περιμέναμε η θα έχει ένα σφάλμα.

```
0 references
public async Task<Result<ApplicationModel>> CreateAsync(CreateApplication request)
{
    return await Request
        .PostJsonAsync(request)
        .ReceiveJson<ApplicationModel>()
        .SendRequestAsync()
        .ConfigureAwait(false);
}

0 references
public async Task<Result<UpdateModel>> UpdateAsync(UpdateApplication request)
{
    return await Request
        .PutJsonAsync(request)
        .ReceiveJson<UpdateModel>()
        .SendRequestAsync()
        .ConfigureAwait(false);
}

0 references
public async Task<Result<Unit>> DeleteAsync(DeleteApplication request)
{
    await Request
        .SendJsonAsync(HttpMethod.Delete, request)
        .SendRequestAsync()
        .ConfigureAwait(false);

    return Unit.Default;
}

1 reference
public async Task<Result<ApplicationModel>> GetAsync(string id)
{
    return await Request
        .AppendPathSegment(id)
        .GetJsonAsync<ApplicationModel>()
        .SendRequestAsync()
        .ConfigureAwait(false);
}

1 reference
public async Task<Result<ApplicationModel[]>> GetAllAsync()
{
    return await Request
        .GetJsonAsync<ApplicationModel[]>()
        .SendRequestAsync()
        .ConfigureAwait(false);
}
```

Εικόνα 78: Identity UI προτζεκτ, Application, ApplicationService.cs

4.3.3: Φάκελος Role



Εικόνα 79: Identity UI προτζεκτ, Role

Κοινά namespaces τα οποία θα χρειαστούν όλα τα αντικείμενα στον φάκελο Role.

```
1    @using LetItGrow.Identity.Role.Components
2    @using LetItGrow.Identity.Role.Models
3    @using LetItGrow.Identity.Role.Services
4    @using LetItGrow.Identity.Role.Validators
```

Εικόνα 80: Identity UI προτζεκτ, Role, _imports.razor

Αντικείμενο για την σωστή προβολή των αντικειμένων Role.

```
1    @inherits BaseComponent
2
3    <div class="@css">
4        <h2 class="my-1">@Model.Name</h2>
5    </div>
6
7    @code{
8        [Parameter]
9        public RoleModel Model { get; set; } = null!;
10
11       protected override void OnParametersSet()
12       {
13           if (Model is null) throw new ArgumentNullException(nameof(Model));
14
15           css = CreateCss("border shadow p-2")
16               .AddClass(Class);
17       }
18   }
```

Εικόνα 81: Identity UI προτζεκτ, Role, RoleComponent.razor

Σελίδα στην οποία μπορεί να περιηγηθεί ο χρήστης στον σύνδεσμο «/role/id» η οποία προσπαθεί να φορτώσει τον συγκεκριμένο ρόλο μέσω του ID. Αν δεν βρεθεί ο ρόλος θα δήξει ανάλογο μήνυμα, και αυτή η σελίδα είναι διαθέσιμη μόνο σε χρήστες με τον ρόλο του διαχειριστή.

```
@page "/role/{Id}"
@attribute [Authorize(Roles = "Admin")]
@inject RoleService service

@if (Id is null)
{
    <NotFound />
}
else if (error is not null)
{
    <ErrorComponent Error="error" />
}
else if (model is not null)
{
    <RoleComponent Model="model" />
}
else
{
    <Loading />
}

@code{
    [Parameter]
    public string? Id { get; set; }

    private Error? error;
    private RoleModel? model;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (!firstRender || Id is null) return;

        (await service.GetAsync(Id))
        .Switch(
            r => model = r,
            e => error = e
        );

        StateHasChanged();
    }
}
```

Εικόνα 82: Identity UI προτζεκτ, Role, RolePage.razor

Σελίδα στην οποία ένας χρήστης με δικαιώματα διαχειριστή μπορεί να δει όλους τους διαθέσιμους ρόλους. Η σελίδα αναλαμβάνει να δήξει οποιοδήποτε σφάλμα στον χρήστη ενώ παρουσιάζει όλους τους διαθέσιμους ρόλους σε αντικείμενα RoleComponent.

```
@page "/role"
@attribute [Authorize(Roles = "Admin")]
@inject RoleService service

<div class="flex justify-between">
    <p>These are the available roles.</p>
    @*<button>Add</button>*@
</div>

@if (error is not null)
{
    <ErrorComponent Error="error" />
}
else if (models is not null)
{
    @* todo: Add delete button *@
    <Virtualize Items="models" Context="model">
        <RoleComponent Class="m-2" Model="model" />
    </Virtualize>
}
else
{
    <div class="flex items-center justify-center">
        <Loading />
    </div>
}

@code{
    private Error? error;
    private List<RoleModel>? models;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (!firstRender) return;

        (await service.GetAllAsync())
        .Switch(
            r => models = new(r),
            e => error = e
        );

        StateHasChanged();
    }
}
```

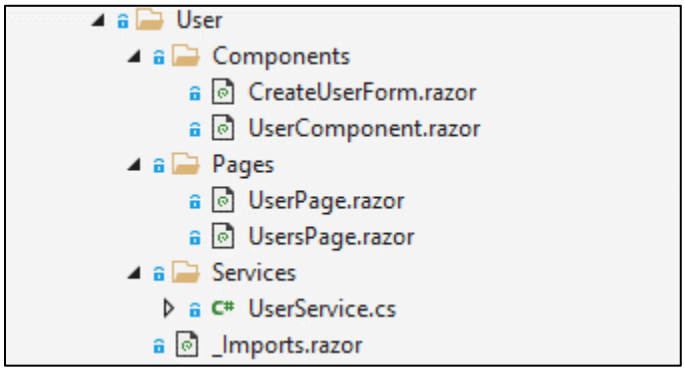
Εικόνα 83: Identity UI προτζεκτ, Role, RolesPage.razor

Κώδικας υπεύθυνος για την επικοινωνία με τον διακομιστή. Αναλαμβάνει να εκτελέσει όλες τις εντολές η αναζητήσεις στον διακομιστή και επιστρέφει ένα αντικείμενο Result το οποίο η θα έχει το επιθυμητό αποτέλεσμα η θα περιέχει ένα αντικείμενο Error δηλαδή ένα σφάλμα.

```
0 references
public async Task<Result<RoleModel>> CreateAsync(CreateRole request)
{
    return await Request
        .PostJsonAsync(request)
        .ReceiveJson<RoleModel>()
        .SendRequestAsync()
        .ConfigureAwait(false);
}

0 references
public async Task<Result<Unit>> DeleteAsync(DeleteRole request)
{
    await Request
        .SendJsonAsync(HttpMethod.Delete, request)
        .SendRequestAsync()
        .ConfigureAwait(false);

    return Unit.Default;
}

1 reference
public async Task<Result<RoleModel>> GetAsync(string id)
{
    return await Request
        .AppendPathSegment(id)
        .GetJsonAsync<RoleModel>()
        .SendRequestAsync()
        .ConfigureAwait(false);
}

1 reference
public async Task<Result<RoleModel[]>> GetAllAsync()
{
    return await Request
        .GetJsonAsync<RoleModel[]>()
        .SendRequestAsync()
        .ConfigureAwait(false);
}
```

Εικόνα 84: Identity UI προτζεκτ, Role, RoleService.cs

4.3.4: Φάκελος User



Εικόνα 85: Identity UI προτζεκτ, User

Κοινά namespaces για όλον τον φάκελο User.

```
@using LetItGrow.Identity.User.Components
@using LetItGrow.Identity.User.Models
@using LetItGrow.Identity.User.Services
@using LetItGrow.Identity.User.Validators

```

Εικόνα 86: Identity UI προτζεκτ, User, _imports.razor

Βασικό αντικείμενο προβολής ενός αντικειμένου User.

```
@inherits BaseComponent

<div class="@css">
    @* todo: Show roles as pills *@
    <h2 class="my-1">@Model.UserName</h2>
    @foreach(var role in Model.Roles)
    {
        <span class="my-1 px-4 py-1 rounded-xl border bg-blue-500 text-gray-100">@role.Name</span>
    }
</div>

@code{
    [Parameter]
    public UserModel Model { get; set; } = null!;

    protected override void OnParametersSet()
    {
        if (Model is null) throw new ArgumentNullException(nameof(Model));

        css = CreateCss("border shadow p-2")
            .AddClass(Class);
    }
}
```

Εικόνα 87: Identity UI προτζεκτ, User, UserComponent.razor

Βασικό αντικείμενο υπεύθυνο για την δημιουργία ενός νέου χρήστη μέσω μιας html φόρμας.

```
@using LetItGrow.Identity.User.Commands
@inherits BaseComponent
@inject UserService service

<EditForm class="@Class" EditContext="form">
    <FluentValidationValidator Validator="form.Validator" />

    <div>
        <label>UserName</label>
        <InputText placeholder="Username" @bind-Value="form.Request.UserName" />
        <ValidationMessage For="() => form.Request.UserName" />
    </div>

    <div>
        <label>Password</label>
        <InputText type="password" placeholder="Password" @bind-Value="form.Request.Password" />
        <ValidationMessage For="() => form.Request.Password" />
    </div>

    <ErrorComponent Error="error" />

    <div class="mt-4 flex justify-end space-x-2">
        <SpinnerButton Class="btn text-gray-100" TResult="Unit" Command="form">Create</SpinnerButton>
        <button type="button" class="btn-cancel text-gray-100" @onclick="Canceled!">Cancel</button>
    </div>
</EditForm>
```

Εικόνα 88: Identity UI προτζεκτ, User, CreateUserForm.razor 1

Ο κώδικας της φόρμας σε σχέση με άλλα αντικείμενα δίνει την δυνατότητα να εκτελέσουμε κώδικα όταν πατηθεί το κουμπί Cancel η έχει τελείωση η εκτέλεση από το κουμπί Create. Επίσης κάνει χρήση του κοινού αντικειμένου FormContext το οποίο διαχειρίζεται την εκτέλεση και τον έλεγχο της φόρμας. Τέλος χρησιμοποιώντας ένα ReactiveCommand εκτελούμε κώδικα μετά από οποιαδήποτε εκτέλεσης για να ενημερώσουμε την σελίδα ότι κάποια αντικείμενα έχουν αλλάξει.

```
@code{
    [Parameter]
    public Action<UserModel>? Created { get; set; }

    [Parameter]
    public Action? Canceled { get; set; }

    private FormContext<CreateUser, CreateUserValidator, Result<UserModel>> form;

    private Error? error { get; set; }

    public CreateUserForm()
    {
        form = new(
            async request => await service.CreateAsync(request),
            result => result.Switch(Success, Failure));

        form.Command.Subscribe(x => StateHasChanged());
    }

    private void Success(UserModel model)
    {
        Created?.Invoke(model);
        form.Reset();
        error = null;
    }

    private void Failure(Error error)
    {
        this.error = error;
    }
}
```

Εικόνα 89: Identity UI προτζεκτ, User, CreateUserForm.razor 2

Σελίδα στην οποία ένας χρήστης με δικαίωμα διαχειριστή μπορεί να δει πληροφορίες για έναν συγκεκριμένο χρήστη στον σύνδεσμο «/user/id». Η σελίδα φροντίζει να δήξει ανάλογο μήνυμα με την επιτυχία φόρτωσης του χρήστη από τον διακομιστή.

```
@page "/user/{Id}"
@attribute [Authorize(Roles = "Admin")]
@inject UserService service

@if (error == Errors.NotFound)
{
    <NotFound />
}
else if (error is not null)
{
    <ErrorComponent Error="error" />
}
else if (model is not null)
{
    <UserComponent Model="model" />
}
else
{
    <Loading />
}

@code{
    [Parameter]
    public string? Id { get; set; }

    private Error? error;
    private UserModel? model;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (!firstRender || Id is null) return;

        (await service.GetAsync(Id))
        .Switch(
            r => model = r,
            e => error = e
        );

        StateHasChanged();
    }
}
```

Εικόνα 90: Identity UI προτζεκτ, User, UserPage.razor

Σελίδα διαθέσιμη για χρήστες με δικαιώματα διαχειριστή στο σύνδεσμο «/user». Η σελίδα φροντίζει να φορτώσει όλους του χρήστες από τον διακομιστή και να τους προβάλει χρησιμοποιώντας αντικείμενα UserComponent η να δήξει στον χρήστη το σφάλμα σε περίπτωση αποτυχίας φόρτωσης.

```
@page "/user"
@attribute [Authorize(Roles = "Admin")]
@inject UserService service

<div class="flex justify-between">
    <p>These are the app users.</p>
    <button @onclick="async () => await dialog.ShowAsync()">Add</button>
</div>

@if (error is not null)
{
    <ErrorComponent Error="error" />
}
else if (models is not null)
{
    @* todo: Add delete button *@
    <div class="w-auto">
        <Virtualize Items="models" Context="model">
            <UserComponent Class="m-2" Model="model" />
        </Virtualize>
    </div>
}
else
{
    <div class="flex items-center justify-center">
        <Loading />
    </div>
}

<Dialog @ref="dialog" Class="w-1/3" Title="Create User">
    <CreateUserForm Created="user => models?.Add(user)" Canceled="async ()=> await dialog.HideAsync()" />
</Dialog>

@code{
    private Error? error;
    private List<UserModel>? models;

    private Dialog dialog = null!;

    protected override async Task OnAfterRenderAsync(bool firstRender)
    {
        if (!firstRender) return;

        (await service.GetAllAsync())
        .Switch(
            r => models = new(r),
            e => error = e
        );

        StateHasChanged();
    }
}
```

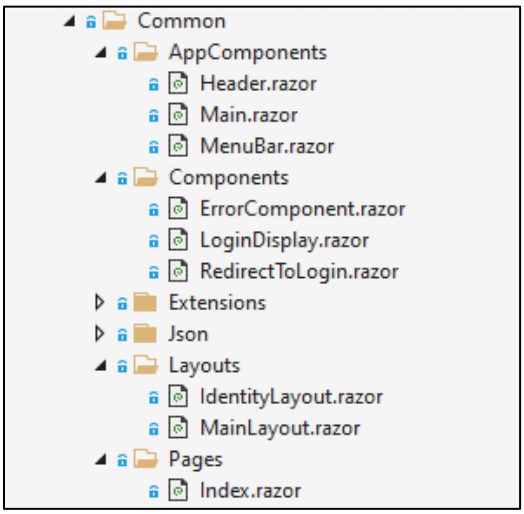
Εικόνα 91: Identity UI προτζεκτ, User, UsersPage.razor

Κοινώς κώδικας που χρησιμοποιείτε από τις σελίδες και κάποια αντικείμενα για την διαχείριση εντολές και αναζητήσεων προς τον διακομιστή.

```
public async Task<Result<UserModel>> CreateAsync(CreateUser request)
{
    return await Request
        .PostJsonAsync(request)
        .ReceiveJson<UserModel>()
        .SendRequestAsync()
        .ConfigureAwait(false);
}

0 references
public async Task<Result<Unit>> DeleteAsync(DeleteUser request)
{
    await Request
        .SendJsonAsync(HttpMethod.Delete, request)
        .SendRequestAsync()
        .ConfigureAwait(false);

    return Unit.Default;
}

1 reference
public async Task<Result<UserModel>> GetAsync(string id)
{
    return await Request
        .AppendPathSegment(id)
        .GetJsonAsync<UserModel>()
        .SendRequestAsync()
        .ConfigureAwait(false);
}

1 reference
public async Task<Result<UserModel[]>> GetAllAsync()
{
    return await Request
        .GetJsonAsync<UserModel[]>()
        .SendRequestAsync()
        .ConfigureAwait(false);
}

0 references
public async Task<Result<UpdateModel>> AddRole(RoleAdd request)
{
    return await Request
        .AppendPathSegment("role")
        .PutJsonAsync(request)
        .ReceiveJson<UpdateModel>()
        .SendRequestAsync()
        .ConfigureAwait(false);
}

0 references
public async Task<Result<UpdateModel>> RemoveRole(RoleRemove request)
{
    return await Request
        .AppendPathSegment("role")
        .SendJsonAsync(HttpMethod.Delete, request)
        .ReceiveJson<UpdateModel>()
        .SendRequestAsync()
        .ConfigureAwait(false);
}
```

Εικόνα 92: Identity UI προτζεκτ, User, UserService.cs

4.3.5: Φάκελος Common



Εικόνα 93: Identity UI προτζεκτ, Common

Το αντικείμενο Header χρησιμοποιείτε στο αρχικό σχέδιο της είναι η μεγάλη μπάρα στην κορυφή της σελίδας η οποία περιέχει το αντικείμενο MenuBar, το κουμπί αλλαγής θέματος και το αντικείμενο LoginDisplay.

```
@inject ThemeJs theme

<header class="content header flex justify-between">
    <MenuBar />

    <LoginDisplay Class="ml-auto" />

    <button class="px-4" @onclick="Toggle">
        <i class="bi @(IsLight ? "bi-moon-fill" : "bi-sun")"></i>
    </button>
</header>

@code{
    private bool IsLight = false;

    private async void Toggle()
    {
        IsLight = await theme.Toggle() == ThemeJs.Light;
        StateHasChanged();
    }

    protected async override Task OnAfterRenderAsync(bool firstRender)
    {
        if (!firstRender) return;

        IsLight = await theme.GetTheme() == ThemeJs.Light;
        StateHasChanged();
    }
}
```

Εικόνα 94: Identity UI προτζεκτ, Common, Header.razor

Το αντικείμενο MenuBar το οποίο χρησιμοποιείτε στο Header είναι υπεύθυνο για την προβολή των κουμπιών περιήγησης. Χρησιμοποιώντας τα αντικείμενα AuthorizeView με ρόλο Admin επιλέγει να δείξει όλα τα κουμπιά η μόνο το κουμπί της αρχικής. Το αντικείμενο MediaQuery φροντίζει ότι αν η διαθέσιμη οθόνη της εφαρμογής είναι αρκετά μικρή θα αφαιρέσει τα κουμπιά για ένα κουμπί μενού το οποίο με τη σειρά του θα εμφανίζει στο χρήστη όλα τα διαθέσιμα κουμπιά περιήγησης, αυτό γίνετε μόνο όταν ο χρήστης που έχει επισκεφτεί την ιστοσελίδα έχει δικαιώματα διαχειριστή.

```
@inject NavigationManager nav

<div>
<AuthorizeView Roles="Admin">
<Authorized>
<MediaQuery Media="@BreakPointsCss.sm">
<Matched>
    <button class="@btn" @onclick="() => nav.NavigateTo(Home)">
        <i class="bi bi-house-fill"></i>
    </button>
    <button class="@btn" @onclick="() => nav.NavigateTo(Role)">Roles</button>
    <button class="@btn" @onclick="() => nav.NavigateTo(User)">Users</button>
    <button class="@btn" @onclick="() => nav.NavigateTo(Apps)">Apps</button>
</Matched>
<Unmatched>
    <button class="@btn" @onclick="() => dialog.Toggle()">
        <i class="bi bi-three-dots"></i>
    </button>

    <Dialog @ref="dialog" Class="w-1/2 bg-transparent">
    <div class="flex flex-col items-center justify-center">
        <button class="@btn shadow my-2 w-full" @onclick="() => nav.NavigateTo(Home)">Home</button>
        <button class="@btn shadow my-2 w-full" @onclick="() => nav.NavigateTo(Role)">Roles</button>
        <button class="@btn shadow my-2 w-full" @onclick="() => nav.NavigateTo(User)">Users</button>
        <button class="@btn shadow my-2 w-full" @onclick="() => nav.NavigateTo(Apps)">Apps</button>
        <hr />
        <button class="@btn shadow mt-2 w-full btn-cancel" @onclick="() => dialog.Hide()">Close</button>
    </div>
    </Dialog>
</Unmatched>
</MediaQuery>
</Authorized>

<NotAuthorized>
    <button class="@btn" @onclick="() => nav.NavigateTo(Home)">
        <i class="bi bi-house-fill"></i>
    </button>
</NotAuthorized>
</AuthorizeView>
</div>

@code{
    private Dialog dialog = null!;
    private const string btn = "ml-1 py-2 px-3";
    private const string Home = "/";
    private const string Role = "role";
    private const string User = "user";
    private const string Apps = "app";
}
```

Εικόνα 95: Identity UI προτζεκτ, Common, MenuBar.razor

Το αντικείμενο Main φροντίζει την προβολή όλων των σελίδων, όλες οι σελίδες μας χρησιμοποιούν ένα κοινό αντικείμενο προβολής χωρίς να χεριάζετε η κάθε σελίδα να προβάλει όλα τα κοινά αντικείμενα τα οποία έχει στην διάθεση του ο χρήστης, αντιθέτως δείχνει μόνο ότι χεριάζετε.

```
<div class="content main">
    @ChildeContent
</div>

@code{
    [Parameter]
    public RenderFragment? ChildeContent { get; set; }
}
```

Εικόνα 96: Identity UI προτζεκτ, Common, Main.razor

Κοινό αντικείμενο που μπορεί να χρησιμοποιηθεί από όλες τις σελίδες για να προβάλει κάποιο σφάλμα η ποιο συγκεκριμένα ένα αντικείμενο Error.

Φροντίζουμε το αντικείμενο να ξεχωρίζει από τα άλλα με κόκκινο χρώμα, ενώ ο τίτλος και κωδικός του σφάλματος είναι με μεγάλα κεφάλαια γράμματα.

```
@inherits BaseComponent

@if (Error is not null)
{
    <div class="@css">
        <h2 class="my-1">@Error.Title <span class="font-mono">(@Error.Status)</span></h2>
        <p class="my-1">@Error.Detail</p>
        @foreach(var (_, detail) in Error.Metadata)
        {
            <p class="my-1">@detail</p>
        }
    </div>
}

@code{
    [Parameter]
    public Error? Error { get; set; }

    protected override void OnParametersSet() =>
        css = CreateCss("error border rounded p-2 text-gray-100 bg-red-500").AddClass(Class);
}
```

Εικόνα 97: Identity UI προτζεκτ, Common, ErrorComponent.razor

Αντικείμενο το οποίο περιέχει δυο κουμπιά, ένα με το όνομα του ενεργού χρήστη και ένα με την εντολή αποσύνδεσης. Τα δυο κουμπιά είναι ενωμένα και φαίνονται σαν ένα αντικείμενο. Τα κουμπιά αυτά μπορούν να εμφανιστούν σε μια σελίδα μόνο όταν ο χρήστης ο οποίος επισκέπτεται την ιστοσελίδα έχει κάνει είσοδο.

```
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@using Microsoft.Extensions.Configuration
@inject NavigationManager Navigation
@inject SignOutSessionStateManager SignOutManager
@inject IConfiguration conf

<AuthorizeView>
    <Authorized>
        <div class="@css">
            <button class="font-bold text-gray-100 btn mr-0 rounded-r-none" @onclick="GoToUserProfile">
                <i class="bi bi-person"></i> @context.User.Identity!.Name
            </button>
            <button class="font-bold text-gray-100 btn-cancel ml-0 rounded-l-none" @onclick="BeginSignOut">
                <i class="bi bi-box-arrow-right"></i> Log out
            </button>
        </div>
    </Authorized>
</AuthorizeView>

@code{
    private CssBuilder css;

    private void GoToUserProfile()
    {
        Navigation.NavigateTo("authentication/profile");
    }

    private async Task BeginSignOut(MouseEventArgs args)
    {
        await SignOutManager.SetSignOutState();

        var url = conf.GetValue<string>("openid:Authority", "https://localhost:5021").TrimEnd('/');
        Navigation.NavigateTo($"{url}/account/login", true);
    }

    [Parameter]
    public string Class { get; set; } = string.Empty;

    protected override void OnParametersSet()
    {
        base.OnParametersSet();
        css = CssBuilder.Default("flex text-sm").AddClass(Class);
    }
}
```

Εικόνα 98: Identity UI προτζεκτ, Common, LoginDisplay.razor

Αντικείμενο το οποίο όταν προβληθεί σε μια σελίδα θα κάνει αυτόματα ανακατεύθυνση στον διακομιστή ταυτοποίησης, έτσι ώστε ο χρήστης να κάνει είσοδο.

```
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@using Microsoft.Extensions.Configuration
@inject NavigationManager nav
@inject IConfiguration conf

@code{
    protected override void OnInitialized()
    {
        var url = conf.GetValue<string>("openid:Authority", "https://localhost:5021").TrimEnd('/');
        nav.NavigateTo($"{url}/account/login", true);
    }
}
```

Εικόνα 99: Identity UI προτζεκτ, Common, RedirectToLogin.razor

Αντικείμενο Layout το οποίο είναι υπεύθυνο να εμφανίζει σελίδες, το συγκεκριμένο χρησιμοποιείτε μόνο όταν ο χρήστης οδηγείτε για ταυτοποίηση, έτσι όλα τα μηνύματα ταυτοποίησης εμφανίζονται μέσα σε αυτό αντί στο συνηθισμένο με την μπάρα περιήγησης.

```
1   @inherits LayoutComponentBase
2
3   <div class="flex justify-center p-4 rounded-xl shadow bg-alt">
4       @Body
5   </div>
6
7   @code{
8
9   }
10
```

Εικόνα 100: Identity UI προτζεκτ, Common, IdentityLayout.razor

Αντικείμενο Layout το οποίο είναι υπεύθυνο για την εμφάνιση της σελίδας. Στο συγκεκριμένο βλέπουμε ότι αν ο χρήστης δεν είναι συνδεδεμένος τον οδηγούμε στη σελίδα ταυτοποίησης ενώ αν είναι συνδεδεμένος χρησιμοποιούμε τα αντικείμενα Header και Main για να ζωγραφίσουμε το αρχικό σχήμα της σελίδας μας. Επίσης έχουμε τυλίξει όλα τα αντικείμενα της σελίδας με το αντικείμενο MediaQueryList αυτό δίνει την δυνατότητα σε αντικείμενα τα οποία είναι εσωτερικά σε αυτό να μπορούν να δουν το μέγεθος της ιστοσελίδας και να εκτελέσουν κώδικα ανάλογα με τις αλλαγές στο μέγεθος, ένα παράδειγμα είναι η χρήση που κάναμε στο αντικείμενο MenuBar.

```
1   @inherits LayoutComponentBase
2   @using LetItGrow.Identity.Common.AppComponents
3   @using LetItGrow.Identity.Common.Components
4
5   <AuthorizeView>
6   <NotAuthorized>
7       <RedirectToLogin />
8   </NotAuthorized>
9
10  <Authorized>
11      <MediaQueryList>
12          <Header />
13          <Main ChildeContent="Body!" />
14      </MediaQueryList>
15  </Authorized>
16  </AuthorizeView>
17
```

Εικόνα 101: Identity UI προτζεκτ, Common, MainLayout.razor

Η αρχική σελίδα της εφαρμογής η οποία δείχνει το μήνυμα ότι «Είμαστε συνδεδεμένοι».

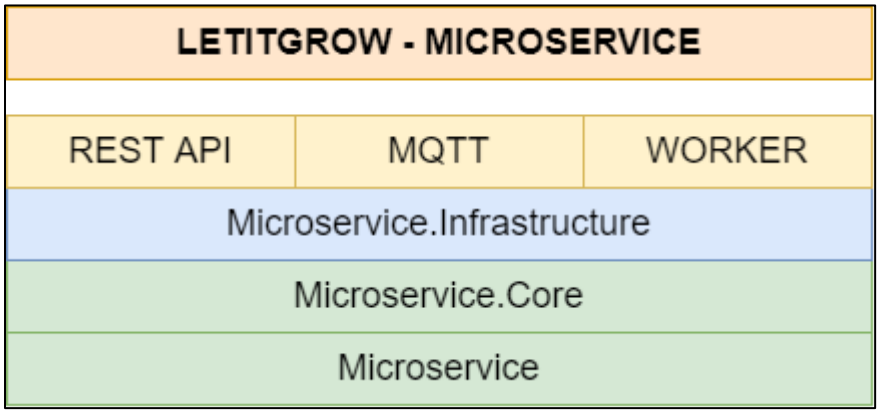
```
1   @page "/"
2
3   <div class="flex justify-center">
4       <h1>You are logged in!</h1>
5   </div>
6
```

Εικόνα 102: Identity UI προτζεκτ, Common, Index.razor

5: ΜΙΚΡΟΥΠΗΡΕΣΙΑ

Μικρουπηρεσία ονομάσαμε την αρχική εφαρμογή διότι είναι βασισμένη στην αρχιτεκτονική μικρό υπηρεσιών. Η δομή της συνολικής εφαρμογής περιέχει τρία βασικά προτζεκτ στα οποία βασίζονται οι τρεις εφαρμογές μας, δυο διακομιστές και μια εφαρμογή worker «εργάτης» [4].

Στην παρακάτω εικόνα φαίνεται η δομή της συνολικής εφαρμογής.



Εικόνα 103: Microservice Architecture

Αυτή η δομή μας επιτρέπει να δημιουργήσουμε κάθε διακομιστή ξεχωριστά οπότε και σε περίπτωση μεγάλου φόρτου μπορούμε να δημιουργήσουμε όποιο το κομμάτι το οποίο χεριάζετε. Για παράδειγμα η εφαρμογή REST API το ποιο είναι σίγουρο ότι θα έχει μικρή κίνηση από τους χρήστες μας, η εφαρμογή Worker γνωρίζουμε είδη ότι χεριάζετε να τρέχει μόνο μια φορά ανά μερικά λεπτά ενώ η εφαρμογή MQTT θα είναι αυτή με την πιο πολλή κίνηση από διάφορους κόμβους.

5.1: Βιβλιοθήκη Microservice

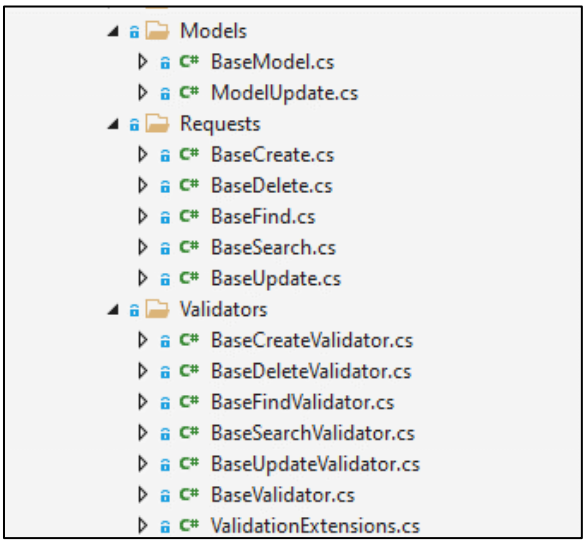
Η βιβλιοθήκη microservice είναι η βάση όλων των εφαρμογών μας. Προσφέρει όλες τις αιτήσεις που μπορεί να στείλει κάποιος στον διακομιστή μας ανεξάρτητος υποδομής μαζί με αρκετούς βοηθητικούς κώδικες οι οποίοι βρίσκονται στον φάκελος Common, μάλιστα το μεγαλύτερο μέρος των αιτήσεων απλά κληρωνόμουν την λειτουργία από βασικά αντικείμενα που θα δούμε. Ακολουθώντας την αρχιτεκτονική Vertical Slice Architecture κάθε φάκελος της εφαρμογής μας είναι αφοσιωμένος στο κομμάτι του. Παρόλα αυτά όλοι οι φάκελοι έχουν κάποιους κοινούς εσωτερικούς φάκελους, αυτοί είναι:

- Models: Κώδικές οι οποίοι έχουν να κάνουν με την παρουσίαση των αντικειμένων μας.
- Requests: Κώδικες οι οποίοι έχουν να κάνουν με την εκτέλεση εντολών η αναζητήσεων σε κάποιον από τους διακομιστές μας.
- Validators: Κώδικες οι οποίοι ελέγχουν την ορθότητα των δεδομένων τα οποία προέρχονται από μια αίτηση. Για παράδειγμα μια αίτηση εύρεσης οποιουδήποτε αντικειμένου στους διακομιστές μας θα πρέπει να προωθήσει το ID του αντικειμένου, αυτό το ID είναι πάντα

έντεκα χαρακτήρες οπότε ελέγχοντας και επιβεβαιώνοντας ότι αυτό το πεδίο είναι λάθος μπορούμε να στείλουμε το ανάλογο σφάλμα.

- Notifications: Κώδικες οι οποίοι επισημάνουν έναν γεγονός η μια ειδοποίηση. Η διαχείριση τους γίνετε ασύγχρονα και χρησιμοποιείτε για απλή ενημερώσει ενός συμβάντος όπως για παράδειγμα την δημιουργία η ενημέρωση ενός αντικειμένου.

Η βιβλιοθήκη περιχέει βασικά αντικείμενα στον φάκελο Common για όλες τις πιθανές αιτήσεις ενώ αντίστοιχα περιέχει βασικά Models και Validators όπως φαίνονται στην επόμενη εικόνα. Όλοι οι υπόλοιποι κώδικες στους αντιστοιχούν φακέλους κληρονομούν τα βασικά χαρακτηριστικά και την λειτουργία με αποτέλεσμα να μην χεριάζετε να ξαναγράψουμε πολύ κώδικα.



Εικόνα 104: Microservice προτζεκτ, Common Classes

Για παράδειγμα ανεξάρτητα της αναζήτησης μας ξέρουμε ότι μια εντολή Find θα δέχεται ένα μοναδικό ID για την εύρεση του αντικειμένου μας ενώ θα υποδεικνύου τι αντικείμενο θα επιστραφεί.

```
1   using MediatR;
2   using System.Text.Json.Serialization;
3
4   namespace LetItGrow.Microservice.Common.Requests
5   {
        3 references
6       public abstract record BaseFind<TResponse> : IRequest<TResponse>
7       {
8           /// <summary>
9           /// The id for the entity to find.
10          /// </summary>
11          [JsonPropertyName("id")]
                11 references
12          public string Id { get; set; } = string.Empty;
13      }
14  }
```

Εικόνα 105: Microservice προτζεκτ, BaseFind

Έτσι ο παραπάνω κώδικας επιτρέπει στη δημιουργία απλών αντικειμένων όπως της επόμενης εικόνας. Βλέπουμε ότι δεν χεριάζετε να ξαναδημιουργήσουμε ότι έχει να κάνει με το υποκοριστικό και την επιστροφή ενός αντικειμένου ενώ μας επιτρέπει να επεκτείνουμε τις δυνατότητες του αντικειμένου μας αν αυτό χεριάζετε.

```
1   using LetItGrow.Microservice.Common.Requests;
2   using LetItGrow.Microservice.Node.Models;
3
4   namespace LetItGrow.Microservice.Node.Requests
5   {
6       /// <summary>
7       /// A request to get a single node using its id.
8       /// </summary>
            10 references
9       public record FindNode : BaseFind<NodeModel>
10      {
                1 reference
11          public FindNode()
12          {
13          }
14
                1 reference
15          public FindNode(string id)
16          {
17              Id = id;
18          }
19      }
20  }
```

Εικόνα 106: Microservice προτζεκτ, FindNode

5.2: Βιβλιοθήκη Microservice.Core

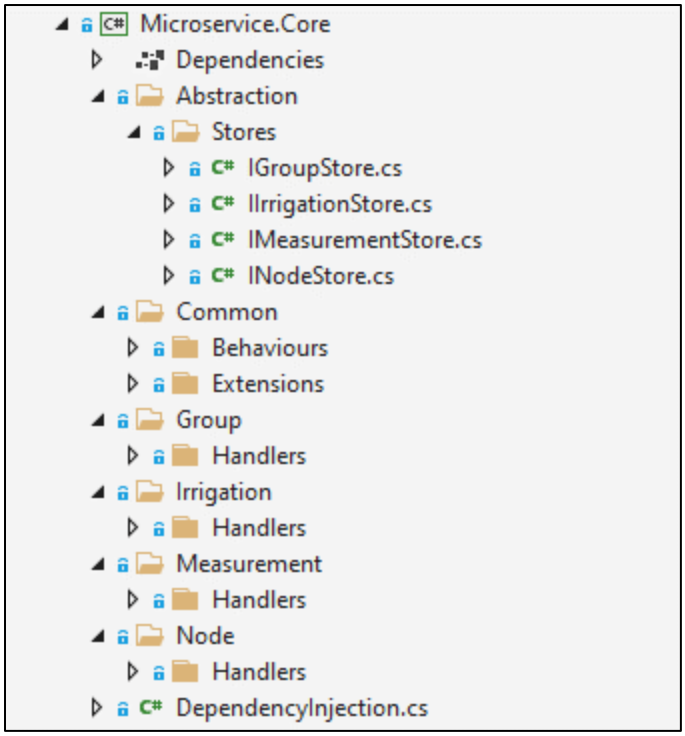
Η βιβλιοθήκη Core δημιουργήθηκε μόνο για εφαρμογές και είναι η βάση αυτών. Η βιβλιοθήκη έχει σκοπό στην δημιουργία κώδικα για την διαχείριση όλων των αιτήσεων της βασικής βιβλιοθήκης Microservce.

Στον φάκελο Common δημιουργήσαμε τον κώδικα που εκτελεί η βιβλιοθήκη MediatR σε κάθε αίτηση όπως αναφέραμε στο κεφάλαιο 2.5.2: CoreHost και μερικές μεθόδους οι οικείες επεκτείνουν την λειτουργικότητα όλης της εφαρμογής χωρίς να κάνουμε συνέχεια αντιγραφή επικόλληση.

Στους φακέλους Group, Irrigation, Measurement και Node βλέπουμε μόνο τον φάκελος Handlers. Αυτοί οι φάκελοι περιέχουν των κώδικα ο οποίος χειρίζεται κάθε αίτηση.

Είναι αρκετά απλός κώδικας το μόνο που κάνει είναι να μεταφέρει κάθε αίτηση στην ανάλογη διεπαφή δηλαδή οι Handlers του Group μεταφέρουν τις ανάλογες εντολές στην διεπαφή IGroupStore στον φάκελο Abstractions/Stores ενώ μετά την ολοκλήρωση της εντολής στέλνουν και μια ειδοποίηση μέσω της βιβλιοθήκης MediatR.

Δεν χεριάζετε όλες οι εφαρμογές να διαχειρίζονται όλες την ειδοποιήσεις, έχουμε δημιουργήσει κώδικα για την διαχείριση αυτών μόνο σε όσες χεριάζετε στις υπόλοιπες τις αγνοούμε.



Εικόνα 107: Microservice.Core προτζεκτ

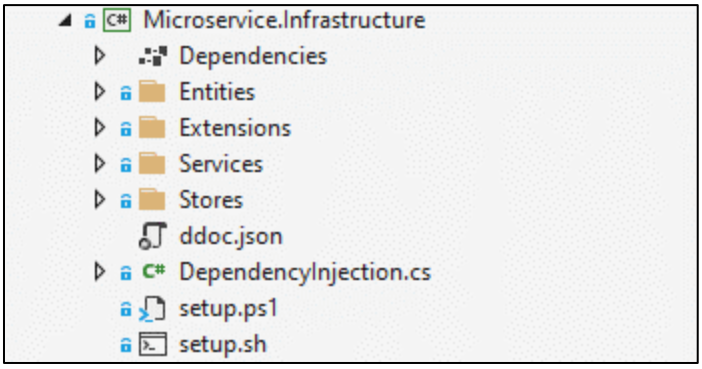
Ο λόγος για τον οποίο επιλέξαμε να δημιουργήσουμε άλλον έναν διαχωρισμό χρησιμοποιώντας τις διεπαφές στον φάκελο Abstraction/Stores έχει να κάνει και με την μετάβαση που κάναμε από PostgreSQL σε CouchDB.

Μας επιτρέπει να δημιουργήσουμε μια βιβλιοθήκη όπως η Microservice.Infrastructure η οποία θα διαθέτει κώδικα ο οποίος επεκτείνει αυτές τις διεπαφές. Με αυτόν τον τρόπο άμα ποτέ θελήσουμε να αλλάξουμε βάση δεδομένων το μόνο που θα πρέπει να κάνουμε είναι να δημιουργήσουμε ξανά την βιβλιοθήκη Microservice.Architecture αλλά αυτήν την φορά με την καινούργια βάση.

Οπότε μας επιτρέπει να δημιουργήσουμε ένα επίπεδο μετάφρασης από την εκτέλεση εντολών στην αληθινή δημιουργία και διαχείριση των δεδομένων.

5.3: Βιβλιοθήκη Microservice.Infrastructure

Η βιβλιοθήκη περιέχει μερικά αρχεία script για την προ τοποθέτηση ρυθμίσεων στην βάση δεδομένων CouchDB, φακέλους για την διαχείριση τον δεδομένων αυτοί είναι οι Entities στον οποίο έχουμε τα δεδομένα της βάσης και κώδικα μετατροπής σε δεδομένα Model, Extensions κώδικες που επεκτείνουν την λειτουργία κάποιον αντικειμένων, Services που περιέχει ένα αντικείμενο το ποιο τυλίγει βασικές λειτουργίες της βιβλιοθήκης 2.5.2: CoreHost και ο φάκελος Stores ο οποίος περιέχει τα αντικείμενα που εφαρμόζουν τις διεπαφές της βιβλιοθήκης Microservice.Core.



Εικόνα 108: Microservice.Infrastructure προτζεκτ

Ο παρακάτω κώδικας είναι ένα παράδειγμα του πως γίνετε η δημιουργία ενός αντικειμένου Node στην βάση δεδομένων.

```
///<inheritdoc/>
3 references
public async ValueTask<NodeModel?> Find(string id, CancellationToken token)
{
    try
    {
        return (await GetDatabase()
            .FindAsync(id, cancellationToken: token)
            .ConfigureAwait(false))
            ?.ToModel();
    }
    catch (Exception ex)
    {
        throw new ErrorException(Errors.ServiceUnavailable, ex);
    }
}
```

Χρησιμοποιώντας ένα αντικείμενο “ICouchClient” από την βιβλιοθήκη CouchDB.NET [50] θα εκτελούμε την εντολή FindAsync η οποία αν βρει το αντικείμενο μας το επιστρέψει σαν Model αλλιώς επιστρέφει την τιμή «null». Επίσης αν κάποιο σφάλμα δημιουργηθεί όπως η μη δυνατότητα σύνδεσης θα δημιουργήσουμε ένα δικό μας σφάλμα που θα λέει ότι η υπηρεσία δεν είναι διαθέσιμη.

5.5: Προτζεκτ Microservice.Mqtt

Στο προτζεκτ αυτό θα δημιουργήσουμε τα διάφορα σημεία με τα οποία επικοινωνούν οι κόμβοι μας, και όλο τον κώδικα για να δημιουργεί ένας κόμβος μετρήσεις, να ενημερώνετε για νέες ρυθμίσεις και εντολές ποτίσματος όπως δημιουργούνται στην βάση δεδομένων, τέλος σε περίπτωση αλλαγής του Token (Κωδικού) ενός κόμβου να γίνετε άμεσα η αποσύνδεση του από τον διακομιστή.

Η τεχνολογία Mqtt είναι μια τεχνολογία pub/sub αυτό σημαίνει ότι σε όλα τα σημεία μπορούμε να δημοσιεύσουμε ένα νέο μήνυμα η και να περιμένουμε να έρθει ένα νέο μήνυμα. Ο διακομιστής αναλαμβάνει να κάνει την διανομή μηνυμάτων.

Ο διακομιστής δέχεται τις παρακάτω ρυθμίσεις μέσο του αρχείου appsettings.yaml. Οι ρυθμίσεις θέτουν σε ποιον σύνδεσμο θα δέχεται συνδέσεις και επίσης θέτουν τις ρυθμίσεις για την βάση δεδομένων. Οι ρυθμίσεις αυτές περιλαμβάνουν όνομα χρήστη, κωδικό, διεύθυνση και προαιρετικά το όνομα της βάσης δεδομένων όλα χωρισμένα με το σύμβολο «;».

```
Urls: https://*:5031

Services:
  CouchDb: <The couchdb username;password;url example => admin;admin;http://localhost:5984>

Serilog:
  MinimumLevel:
    Default: Information
    Override:
      System: Warning
      Microsoft: Warning
      Microsoft.Hosting.Lifetime: Information
      Microsoft.AspNetCore.SignalR: Information
      Microsoft.Extensions.Diagnostics.HealthChecks: Warning
  WriteTo:
    - { Name: Console }
    - { Name: File, Args: { Path: "logs/logs.txt", RollingInterval: Day } }
```

Εικόνα 109: Microservice.Mqtt προτζεκτ, appsettings.yaml

5.5.1: Σημεία Επικοινωνίας

Σε κάθε σημείο επικοινωνίας σιγουρευόμαστε ότι μήνυμα προέρχεται από τον διακομιστή και αυτό διότι ο διακομιστής δημοσιεύει αυτά τα μηνύματα ανάλογα την κατάσταση. Το connection δημοσιεύετε όταν αλλάζει η κατάσταση σύνδεσης, το settings όταν ένας κόμβος έχει ενημερωθεί και έχουν αλλάξει οι ρυθμίσεις του, όταν έρθει νέα εντολή αφού ο διακομιστής παρακολουθεί τις αλλαγές στην βάση δεδομένων. Το μόνο σημείο το οποίο δημοσιεύει ένας κόμβος είναι το Measurement για να δημοσιεύσει νέες μετρήσεις πρέπει το Id που την δημοσιεύει να είναι ίδιο με τον Id της σύνδεσης. Όταν μια μέτρηση δημοσιεύετε με την βιβλιοθήκη MediatR εκτελούμε μια αίτηση δημιουργίας μέτρησης, ο κώδικας που εκτελείτε βρίσκετε στην βιβλιοθήκη Microservice.Core.

```
[MqttRoute("node/connection/{nodeId}")]
0 references
public Task Connection(string nodeId)
{
    GetLogger<NodeMqttController>().LogDebug("Publish Connection");
    return ClientId is null ? Ok() : BadMessage();
}

[MqttRoute("node/settings/{nodeId}")]
0 references
public Task Settings(string nodeId)
{
    GetLogger<NodeMqttController>().LogDebug("Publish Settings");
    return ClientId is null ? Ok() : BadMessage();
}

[MqttRoute("node/irrigation/{nodeId}")]
0 references
public Task Irrigate(string nodeId)
{
    GetLogger<NodeMqttController>().LogDebug("Publish Irrigation");
    return ClientId is null ? Ok() : BadMessage();
}

[MqttRoute("node/measurement/{nodeId}")]
0 references
public Task Measure(string nodeId) =>
    VerifyClientIdEqualsNodeId(nodeId)
        ? Send<CreateMeasurement, Unit>()
        : BadMessage();
```

Εικόνα 110: Microservice.Mqtt προτζεκτ, Mqtt Routes

5.5.2: Λήψη Ενημερώσεων από την CouchDB

Στον παρακάτω κώδικα βλέπουμε τις ενημερώσεις για πότισμα, δηλώνουμε ενδιαφέρον για ενημερώσεις σε όλα τα αντικείμενα της βάσης τα οποία περνάνε στην κατηγορία Views.Irrigation από την στιγμή «τώρα» αυτό διότι η CouchDB έχει την δυνατότητα να μας στείλει ενημερώσεις για όλα τα αντικείμενα που έχει από την ώρα της δημιουργίας της αλλά εμάς μας ενδιαφέρουν οι ενημερώσεις από όταν συνδεθήκαμε και μετά.

```
var changes = db.GetContinuousChangesAsync(
    options: new()
    {
        IncludeDocs = true,
        Since = "now",
        Heartbeat = 60_001
    },
    filter: ChangesFeedFilter.View($"{Views.Irrigations.Design}/{Views.Irrigations.Value}"),
    stoppingToken);
```

Εικόνα 111: Microservice.Mqtt προτζεκτ, Irrigation Changes 1

Για κάθε αλλαγή που έρχεται κοιτάμε αν ήταν διαγραφή η αν ήταν ενημέρωση βεβαιώνοντας ότι η μεταβλητή Rev δεν ξεκινά με τον αριθμό «1». Αν αυτό ισχύει το «continue» στέλνει τον κώδικα στην αρχή του «foreach». Αν λοιπόν η εντολή που ήρθε μόλις δημιουργήθηκε για πρώτη φορά δημιουργούμε ένα κλειδί μοναδικό με την τιμή Id, κοιτάμε αν αυτή η τιμή είναι αποθηκευμένη στην προσωρινή μνήμη, αυτό διότι κάποιες φορές η CouchDB στέλνει δυο φορές μια ενημέρωση. Στην περίπτωση που βρούμε στην κοινή μνήμη το αντικείμενο εκτελούμε την εντολή «continue» αλλιώς δημιουργούμε μια νέα τιμή στην προσωρινή μνήμη με το κλειδί μας για τουλάχιστον μισή ώρα η και μέχρι να την διαβάσουμε ξανά αλλιώς διαγράφετε.

Τέλος εκτελούμε την μέθοδο PublishAsync η οποία θα δημοσιεύσει στο σημείο MQTT την νέα μέτρηση σε μορφή Model.

```
await foreach (var change in changes)
{
    if (change.Deleted || !RevIsOne(change.Document.Rev)) continue;

    var key = $"irrigation-hash:{change.Document.Id}";

    if (cache.TryGetValue(key, out _)) continue;

    cache.Set(key, byte.MinValue, new MemoryCacheEntryOptions
    {
        SlidingExpiration = TimeSpan.FromMinutes(10)
    });

    await PublishAsync(change.Document.ToModel());
}
```

Εικόνα 112: Microservice.Mqtt προτζεκτ, Irrigation Changes 2

Στον παρακάτω κώδικα βλέπουμε τις ενημερώσεις για κόμβους, δηλώνουμε ενδιαφέρον για ενημερώσεις σε όλα τα αντικείμενα τα οποία περνάνε στην κατηγορία Views.Node.

```
var changes = db.GetContinuousChangesAsync(
    options: new()
    {
        IncludeDocs = true,
        Since = "now",
        Heartbeat = 60_001
    },
    filter: ChangesFeedFilter.View($"{Views.Node.Design}/{Views.Node.Value}"),
    stoppingToken);
```

Εικόνα 113: Microservice.Mqtt προτζεκτ, Node Changes 1

Για κάθε αλλαγή που έρχεται θα δημοσιεύσουμε και ανάλογες ειδοποιήσεις MediatR, αυτό το κάνουμε έτσι ώστε η κάθε ειδοποίηση να διαχειριστή με τον δικό της τρόπο.

Πρώτα θα εκλέξουμε αν το έγγραφο έχει διαγραφεί, σε αυτήν τη περίπτωση θα δημοσιεύσουμε την ειδοποίηση NodeDeleted με το Id και το Rev και θα εκτελέσουμε την εντολή «continue». Δεν μας ενδιαφέρουν αλλά δεδομένα πέρα από το μοναδικό Id του κόμβου και το Rev το οποίο είναι ο τελευταίος κωδικός αλλαγής για αυτό και τα χρησιμοποιούμε στην ειδοποίηση.

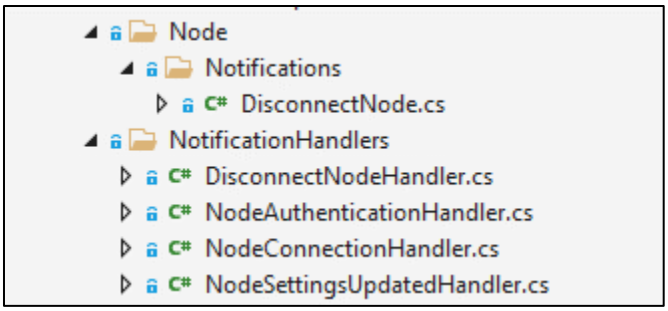
Σε περίπτωση που δεν έχει διαγραφτεί ο κόμβος θα δημοσιεύσουμε την ειδοποίηση NodeTokenChanged και NodeSettingsUpdated, η ειδοποίηση NodeTokenChanged ενημερώνει για αλλαγή του κωδικού ενώ η ειδοποίηση NodeSettingsUpdated ενημερώνει ότι αλλάξαν οι ρυθμίσεις.

```
await foreach (var change in changes)
{
    var node = change.Document;

    if (change.Deleted)
    {
        publisher.PublishAndForget(new NodeDeleted(node.Id, node.Rev));
        continue;
    }

    publisher.PublishAndForget(new NodeTokenChanged(node.Id, node.Rev, node.Token));
    publisher.PublishAndForget(new NodeSettingsUpdated(node.Id, node.Type, node.Settings.ToJsonDocument()));
}
```

Εικόνα 114: Microservice.Mqtt προτζεκτ, Node Changes 2

5.5.3: Κώδικας Διαχείρισης Ενημερώσεων

Μαζί με τις τρεις ειδοποιήσεις που χειριζόμαστε, δημιουργήσαμε και μια ακόμα μόνο για αυτό το προτζεκτ η οποία θα ειδοποιεί ότι ένας κόμβος πρέπει να αποσυνδεθεί από τον διακομιστή. Αυτό γίνετε σε περίπτωση διαγράφης ενός κόμβου η λόγο αλλαγής του κωδικού του (Token).



Εικόνα 115: Microservice.Mqtt προτζεκτ, Notification/Notification Handlers

Ο παρακάτω κώδικας εκτελείτε για κάθε μια δημοσίευση ενημέρωσης DiscconectNode, χρησιμοποιώντας το Id του κόμβου και τον IMqttServer ψάχνουμε για έναν κόμβο με το ίδιο Id, φιλτράρουμε το αποτέλεσμα έτσι ώστε αν ο κόμβος δεν υπάρχει να μην εκτελέσουμε κώδικα αλλιώς διαλέγουμε την ενεργή σύνδεση και καλούμε την μέθοδο DisconnectAsync η οποία θα διακόψει την συγκεκριμένη σύνδεση.

```
0 references
public Task Handle(DisconnectNode notification, CancellationToken cancellationToken)
{
    DisconnectNode(notification.NodeId);

    return Task.CompletedTask;
}

1 reference
private void DisconnectNode(string nodeId) => Observable
    .FromAsync(mqtt.GetClientStatusAsync)
    .Select(x => x.FirstOrDefault(status => status.ClientId == nodeId))
    .Where(x => x is not null)
    .Subscribe(status => Observable.FromAsync(status!.DisconnectAsync).Subscribe());
```

Εικόνα 116: Microservice.Mqtt προτζεκτ, DisconnectNodeHandler.cs

Ο κώδικας που διαχειρίζεται τις ειδοποιήσεις NodeConnection χρησιμοποιώντας την διεπαφή ICouchDatabase ανακτά τον κόμβο με βάση το Id του και του αλλάζει την σύνδεση ανάλογα με το τι έστειλε η ειδοποίηση.

```
public NodeConnectionHandler(ICouchClient couch, IConfiguration configuration)
{
    db = couch.GetDatabase<Node>(configuration.GetCouchDbName() ?? "letitgrow", Discriminators.Node);
}

public async Task Handle(NodeConnection notification, CancellationToken cancellationToken)
{
    var node = await db.FindAsync(notification.NodeId, cancellationToken: cancellationToken)
        ?? throw new ErrorException(Errors.NotFound);

    node.Connected = notification.Status;

    await db.AddOrUpdateAsync(node, cancellationToken: cancellationToken);
}
```

Εικόνα 117: Microservice.Mqtt προτζεκτ, NodeConnectionHandler.cs

Ο κώδικας σε κάθε ειδοποίηση NodeSettingsUpdated δημιουργεί ένα ζεύγος κλειδιού τιμής οπού η τιμή είναι το hash (αριθμός ίδιος για κάθε παρόμοια γραμματοσειρά) των ρυθμίσεων. Αν στην προσωρινή μνήμη υπάρχει το κλειδί συγκρίνουμε την τιμή του με την καινούργια και αν είναι ιδίες σταματάμε την εκτέλεση εκεί. Αν δεν είναι προσθέτουμε την καινούργια τιμή στην προσωρινή μνήμη και δημοσιεύουμε ένα νέο μήνυμα MQTT για τον κόμβο.

```
public async Task Handle(NodeSettingsUpdated notification, CancellationToken cancellationToken)
{
    var key = $"node-settings-hash:{notification.Id}";
    var hashToCompare = notification.Settings.GetJsonHashCode();

    if (cache.TryGetValue(key, out int hash)
        && hashToCompare == hash)
        return;

    cache.Set(key, hashToCompare, new MemoryCacheEntryOptions
    {
        SlidingExpiration = TimeSpan.FromDays(14)
    });

    await PublishAsync(notification.Id, notification.Type, notification.Settings);
}

1 reference
private Task PublishAsync(string clientId, NodeType type, JsonDocument settings) => mqtt.PublishAsync(b => b
    .WithAtLeastOnceQoS()
    .WithTopic($"node/settings/{clientId}")
    .WithContentType(ContentTypes.Proto)
    .WithProtoPayload(type switch
    {
        NodeType.Irrigation => ProtoSerializer.Serialize<IrrigationSettings>(settings),
        NodeType.Measurement => ProtoSerializer.Serialize<MeasurementSettings>(settings),
        _ => throw new ArgumentException("Unknown node type at NodeSettingsUpdatedHander", nameof(type))
    })
    .WithRetainFlag(true));
```

Εικόνα 118: Microservice.Mqtt προτζεκτ, NodeSettingsUpdatedHandler

Ο κώδικας διαχείρισης ειδοποιήσεων NodeTokenChanged εφαρμόζετε από το ίδιο αντικείμενο μαζί με τον κώδικα για την διαχείριση ειδοποιήσεων NodeDeleted, επίσης εφαρμόζει την διεπαφή INodeTokenAuthenticator η οποία έχει μια μέθοδο για να τακτοποιεί τους κόμβους με τον κωδικό τους. Αυτό το κάναμε ώστε ένα αντικείμενο να είναι υπεύθυνο για ότι έχει να κάνει με την διαχείριση των συνδέσεων και να μειώσουμε τον επαναλαμβανόμενο κώδικα.

Όταν έρχεται ειδοποίηση ότι ένας κόμβος διαγράφτηκε κοιτάμε αν τον έχουμε στην προσωρινή μνήμη. Αν τον έχουμε τον αφαιρούμε. Τέλος δημοσιεύουμε μια ειδοποίηση αποσύνδεσης για τον συγκεκριμένο κόμβο.

```
public Task Handle(NodeDeleted notification, CancellationToken cancellationToken)
{
    var token = FindToken(notification.NodeId);

    if (token is { })
    {
        RemoveToken(notification.NodeId);
    }

    publisher.PublishAndForget(new DisconnectNode(notification.NodeId));

    return Task.CompletedTask;
}
```

Εικόνα 119: Microservice.Mqtt προτζεκτ, NodeAuthenticationHandler.cs 1

Όταν έρχεται ειδοποίηση αλλαγής Token για έναν κόμβο προσπαθούμε να ανακτήσουμε πληροφορίες από την προσωρινή μνήμη, αν δεν υπάρχει τότε το αποθηκεύουμε, αν υπάρχει τότε συγκρίνουμε τα δυο Token και αν είναι διαφορετικά ενημερώνουμε το αποθηκευμένο Token και στενόφυλλέ μια ειδοποίηση αποσύνδεσης.

```
0 references
public Task Handle(NodeTokenChanged notification, CancellationToken cancellationToken)
{
    var nodeId = notification.NodeId;
    var token = FindToken(nodeId);

    if (token is null)
    {
        SetToken(nodeId, notification.Token);
    }
    else if (token != notification.Token)
    {
        SetToken(nodeId, notification.Token);
        publisher.PublishAndForget(new DisconnectNode(nodeId));
    }

    return Task.CompletedTask;
}
```

Εικόνα 120: Microservice.Mqtt προτζεκτ, NodeAuthenticationHandler.cs 2

Ο κώδικας υλοποίησης της διεπαφή INodeAuthenticator, όπως και πριν προσπαθούμε να ανακτήσουμε από την προσωρινή μνήμη το Token για τον συγκεκριμένο κόμβο, αν υπάρχει συγκρίνουμε τα δυο token και επιστρέφουμε το αποτέλεσμα. Αν δεν είναι διαθέσιμο κάποιο token ανακτάμε από την βάση δεδομένων τον συγκεκριμένο κόμβο, αν τον ανακτήσουμε με επιτυχία αποθηκεύουμε στην προσωρινή μνήμη το Token και επιστρέφουμε πάλι την σύγκριση τους. Αν κανένα από τα παραπάνω δεν συνέβη επιστρέφουμε ότι το συγκεκριμένο token δεν είναι έγκυρο.

```
3 references
public async ValueTask<bool> Authenticate(string nodeId, string token)
{
    if (FindToken(nodeId) is { } storedToken)
    {
        return storedToken == token;
    }

    var node = await store.Find(nodeId, default);
    if (node is not null)
    {
        SetToken(nodeId, token);

        return node.Token == token;
    }
    return false;
}
```

Εικόνα 121: Microservice.Mqtt προτζεκτ, NodeAuthenticationHandler.cs 3

5.5.4: Κώδικας MQTT

Για να λειτουργήσει σωστά ο διακομιστής MQTT θα χρειαστεί να δημιουργήσουμε και μερικά αντικείμενα για την διαχείριση ορισμένων συμβάντων όπως για παράδειγμα την σύνδεση ενός κόμβου. Συνολικά δημιουργήσαμε κώδικες για την αποθήκευση μηνυμάτων MQTT τα οποία πρέπει να διατηρούνται, την σύνδεση, την αποσύνδεση και την ταυτοποίηση ενός κόμβου.

Ο παρακάτω κώδικας δημιουργεί ένα αρχείο «RetainedMessages.json» το οποίο θα αποθηκεύει μηνύματα που πρέπει να διατηρεί ο διακομιστής. Όταν φορτώνει τα μηνύματα που το αρχείο δεν υπάρχει μια κενή λίστα χωρίς μηνύματα αλλιώς κάνει μετατροπή του αρχείου σε μηνύματα.

```
private static readonly string Filename = Path.Combine(Environment.CurrentDirectory, "RetainedMessages.json");

0 references
public async Task SaveRetainedMessagesAsync(IList<MqttApplicationMessage> messages)
{
    await File.WriteAllTextAsync(Filename, JsonConvert.SerializeObject(messages));
}

0 references
public async Task<IList<MqttApplicationMessage>> LoadRetainedMessagesAsync()
{
    return File.Exists(Filename)
        ? JsonConvert.DeserializeObject<List<MqttApplicationMessage>>(await File.ReadAllTextAsync(Filename))!
        : new List<MqttApplicationMessage>();
}
```

Εικόνα 122: Microservice.Mqtt προτζεκτ, MqttServerRetainedMessageStorage.cs

Ο παρακάτω κώδικας χρησιμοποιώντας την διεπαφή INodeAuthenticator προσπαθεί να δει αν ένας κόμβος έχει το δικαίωμα σύνδεσης στον διακομιστή. Αν ναι το καταγραφή και επιστρέφει, αν όχι καταγραφεί την αποτυχία σύνδεσης και αποσυνδέει τον συγκεκριμένο κόμβο στέλνοντας του το σφάλμα ότι δεν έχει δικαίωμα σύνδεσης.

```
private readonly INodeTokenAuthenticator authenticator;
private readonly ILogger<MqttServerConnectionValidator> logger;

0 references
public MqttServerConnectionValidator(INodeTokenAuthenticator authenticator, ILogger<MqttServerConnectionValidator> logger)
{
    this.authenticator = authenticator;
    this.logger = logger;
}

0 references
public async Task ValidateConnectionAsync(MqttConnectionValidatorContext context)
{
    if (await authenticator.Authenticate(context.Username, context.Password))
    {
        logger.LogInformation("Client '{id}' authorized using '{username}'", context.ClientId, context.Username);
        return;
    }

    logger.LogWarning("Client '{id}' not authorized using '{username}'", context.ClientId, context.Username);
    context.ReasonCode = MQTTnet.Protocol.MqttConnectReasonCode.NotAuthorized;
}
```

Εικόνα 123: Microservice.Mqtt προτζεκτ, MqttServerConnectionValidator.cs

Ο παρακάτω κώδικας διαχειρίζεται την σύνδεση ενός κόμβου, δηλαδή όταν ένας κόμβος τακτοποιηθεί και συνδεθεί αυτός ο κώδικας θα εκτελεστεί.

Το πρώτο πράγμα που κάνουμε είναι να ανακτήσουμε τον κόμβο από την βάση δεδομένων αν αυτό είναι εφικτό αλλιώς δημιουργούμε ένα σφάλμα. Στη συνέχεια στέλνουμε τις ειδοποιήσεις σύνδεσης με τιμή “true” και ρυθμίσεων έτσι ώστε ο κόμβος να έχει τις τελευταίες ρυθμίσεις.

Τέλος καταγράφουμε την σύνδεση στέλνουμε το MQTT μήνυμα και προσπαθούμε να προσθέσουμε τον κόμβο σε μια λίστα η οποία κρατά όλα τα id των συνδεδεμένων κόμβων.

```
0 references
public async Task HandleClientConnectedAsync(MqttServerClientConnectedEventArgs eventArgs)
{
    var id = eventArgs.ClientId;
    var node = await db.FindAsync(id) ?? throw new ErrorException(Errors.NotFound);

    publisher.PublishAndForget(new NodeConnection(node.Id, true, clock.GetNow()));
    publisher.PublishAndForget(new NodeSettingsUpdated(node.Id, node.Type, node.Settings.ToJsonDocument()));

    logger.LogInformation("Client '{id}' connected", id);
    await PublishAsync(id, true);
    ConnectedNodes.TryAdd(id, null);
}
```

Εικόνα 124: Microservice.Mqtt προτζεκτ, MqttServerConnectionHandler.cs 1

Ο παρακάτω κώδικας διαχειρίζεται την αποσύνδεση ενός κόμβου, δηλαδή ο κώδικας που θα εκτελεστεί μετά την αποσύνδεση ενός κόμβου. Πρώτα προσπαθούμε να ανακτήσουμε τον κόμβο από την βάση αλλιώς δημιουργούμε ένα σφάλμα. Αφού ανακτήσουμε τον κόμβο δημοσιεύουμε την ειδοποίηση σύνδεσης με την τιμή “false” καταγράφουμε την αποσύνδεση, δημοσιεύουμε στην ροή MQTT και διαγράφουμε τον κόμβο από την λίστα μας με τους συνδεδεμένους κόμβους.

```
0 references
public async Task HandleClientDisconnectedAsync(MqttServerClientDisconnectedEventArgs eventArgs)
{
    var id = eventArgs.ClientId;
    var node = await db.FindAsync(id) ?? throw new ErrorException(Errors.NotFound);

    publisher.PublishAndForget(new NodeConnection(node.Id, false, clock.GetNow()));

    logger.LogInformation("Client '{id}' disconnected", id);
    await PublishAsync(id, false);
    ConnectedNodes.TryRemove(id, out _);
}
```

Εικόνα 125: Microservice.Mqtt προτζεκτ, MqttServerConnectionHandler.cs 2

5.6: Προτζεκτ Microservice.RestApi

Το προτζεκτ Microservice.RestApi είναι ο διακομιστής με τον οποίο οποιαδήποτε εφαρμογή επικοινωνεί. Μάλιστα μπορούμε να δημιουργήσουμε πολλές εφαρμογές οι οποίες επικοινωνούν με αυτόν τον διακομιστή, εμείς επιλέξαμε να κάνουμε μια ιστοσελίδα αλλά τίποτα δεν μας αποτρέπει από την δημιουργία μιας IOS ή Android εφαρμογής. Η εφαρμογή μας διαθέτει σημεία σύνδεσης SignalR η HTTP Rest και χάρη στην χρήση της βιβλιοθήκης Microservice.Core και MediatR θα δούμε ότι ο κώδικας είναι απλά εκεί για την δημιουργία των σημείων στον διακομιστή μας και για τίποτα παραπάνω.

Παρακάτω βλέπουμε τις ρυθμίσεις του διακομιστή, με τις οποίες θα γνωρίζει που να κάνει επιβεβαίωση των πελατών και που βρίσκετε η βάση δεδομένων. Secret είναι η τιμή με την οποία θα αποκωδικοποιούμε τα στοιχεία ταυτοποίησης του χρήστη τα οποία στέλνονται αυτόματα μετά την σύνδεση ενός χρήστη στην σελίδα ταυτοποίησης.

```
1   AllowedHosts: "*"
2   Urls: https://*:5011
3
4   Swagger:
5     Enabled: false
6
7   Services:
8     Identity: <The full Identity application url>
9     CouchDb: <The couchdb username;password;url example => admin;admin;http://localhost:5984>
10
11  Secrets:
12    JWT: <JWT Secret here>
```

Εικόνα 126: Microservice.RestApi προτζεκτ, appsettings.yaml

Ο παρακάτω κώδικας διαχειρίζεται τις αλλαγές των κόμβων στην βάση δεδομένων, όπως είδαμε στο προτζεκτ MQTT λαμβάνουμε ειδοποιήσεις μόνο για τα αντικείμενα τα οποία θα ταιριάζουνε στην μέθοδο “Views.Node”.

Ο κώδικας σε περίπτωση διαγραφής του αντικειμένου θα δημοσιεύσει την ειδοποίηση NodeDeleted.

Στην περίπτωση που η τιμή Rev ξεκινά με τον αριθμό «1» θα δημοσιεύσει την ειδοποίηση NodeCreated.

Τέλος σε οποιαδήποτε άλλη περίπτωση θα δημοσιεύσει την ειδοποίηση NodeUpdated.

```
0 references
protected override async Task ExecuteAsync(CancellationToken stoppingToken)
{
    await Task.Yield();
    await Task.Delay(1000, stoppingToken).Ignore();

    if (stoppingToken.IsCancellationRequested) return;

    var changes = db.GetContinuousChangesAsync(
        options: new()
        {
            IncludeDocs = true,
            Since = "now",
            Heartbeat = 60_001
        },
        filter: ChangesFeedFilter.View($"{Views.Node.Design}/{Views.Node.Value}"),
        stoppingToken);

    await foreach (var change in changes)
    {
        var node = change.Document;

        if (change.Deleted)
        {
            publisher.PublishAndForget(new NodeDeleted(node.Id, node.Rev));
        }
        else if (GetRev(node) == 1)
        {
            publisher.PublishAndForget(new NodeCreated(node.ToModel()));
        }
        else
        {
            publisher.PublishAndForget(new NodeUpdated(node.ToModel()));
        }
    }
}
```

Εικόνα 127: Microservice.RestApi προτζεκτ, CouchDbNodeChangesWorker.cs

Ο παρακάτω κώδικας διαχειρίζεται τις αλλαγές των γκρουπ στην βάση δεδομένων, όπως είδαμε στο παραπάνω λαμβάνουμε ειδοποιήσεις μόνο για τα αντικείμενα τα οποία θα ταιριάζουνε στην μέθοδο “Views.Group”.

Ο κώδικας σε περίπτωση διαγραφής του αντικειμένου θα δημοσιεύσει την ειδοποίηση GroupDeleted.

Στην περίπτωση που η τιμή Rev ξεκινά με τον αριθμό «1» θα δημοσιεύσει την ειδοποίηση GroupCreated.

Τέλος σε οποιαδήποτε άλλη περίπτωση θα δημοσιεύσει την ειδοποίηση GroupUpdated.

```
0 references
protected override async Task ExecuteAsync(CancellationToken stoppingToken)
{
    await Task.Yield();
    await Task.Delay(1000, stoppingToken).Ignore();

    if (stoppingToken.IsCancellationRequested) return;

    var changes = db.GetContinuousChangesAsync(
        options: new()
        {
            IncludeDocs = true,
            Since = "now",
            Heartbeat = 60_001
        },
        filter: ChangesFeedFilter.View($"{Views.Group.Design}/{Views.Group.Value}"),
        stoppingToken);

    await foreach (var change in changes)
    {
        var group = change.Document;

        if (change.Deleted)
        {
            publisher.PublishAndForget(new GroupDeleted(group.Id, group.Rev));
        }
        else if (GetRev(group) == 1)
        {
            publisher.PublishAndForget(new GroupCreated(group.ToModel()));
        }
        else
        {
            publisher.PublishAndForget(new GroupUpdated(group.ToModel()));
        }
    }
}
```

Εικόνα 128: Microservice.RestApi προτζεκτ, CouchDbGroupChangesWorker.cs

Παρακάτω θα δούμε πως δημιουργούμε σημεία εκτέλεσης για τα αντικείμενα ομάδας που έχουμε «Group». Με τον ίδιο τρόπο δημιουργούμε σημεία για τις μετρήσεις για το πότισμα και για τους κόμβους.

Παρακάτω βλέπουμε των κώδικα δημιουργίας σημείων εκτέλεσης για την βιβλιοθήκη SignalR κάθε σημείο έχει το όνομα εκτέλεσης, δέχεται την ανάλογη αίτηση την οποία θα στείλει στην βιβλιοθήκη MediatR και επιστρέφει το ανάλογο αποτέλεσμα για κάθε αίτηση.

```
// NodeGroup part of the hub.
10 references
public partial class ApiHub
{
    [HubMethodName("group:get")]
    0 references
    public Task<GroupModel> NodeGroup_Get(FindGroup request) =>
        SendRequest(request, Context.ConnectionAborted);

    [HubMethodName("group:search")]
    0 references
    public Task<GroupModel[]> NodeGroup_Get(SearchGroups request) =>
        SendRequest(request, Context.ConnectionAborted);

    [HubMethodName("group:create")]
    0 references
    public Task<GroupModel> NodeGroup_Create(CreateGroup request) =>
        SendRequest(request, Context.ConnectionAborted);

    [HubMethodName("group:update")]
    0 references
    public Task<ModelUpdate> NodeGroup_Update(UpdateGroup request) =>
        SendRequest(request, Context.ConnectionAborted);

    [HubMethodName("group:delete")]
    0 references
    public Task<Unit> NodeGroup_Delete(DeleteGroup request) =>
        SendRequest(request, Context.ConnectionAborted);
}
```

Εικόνα 129: Microservice.RestApi προτζεκτ, GroupHub.cs

Παρακάτω βλέπουμε των κώδικα που δημιουργεί σημεία εκτέλεσης για την τεχνολογία HTTP Rest, βλέπουμε διάφορα σημεία στον κώδικα τα οποία περιγράφουν καλύτερα την κάθε μέθοδο όπως “ProducesResponseType” αυτά χρησιμοποιούνται στην βιβλιοθήκη Swagger.

Επίσης βλέπουμε διάφορα αντικείμενα τα οποία στον συγκεκριμένο κώδικα θα διαχωρίσουν την κάθε μέθοδο εκτέλεσης.

Η κλάση GroupController κάνει διαθέσιμα όλα τα παρακάτω σημεία στην διεύθυνση “v1/group”. Χρησιμοποιώντας την ιδιότητα της τεχνολογίας HTTP να έχουμε διαφορετικούς τύπους αίτησης χωρίζουμε την κάθε μέθοδο σε μια ανάλογη αίτηση για παράδειγμα HTTPGET γίνετε η μέθοδος που στέλνει το αντικείμενο SearchGroups για να αναζήτηση σε όλα τα Group ενώ HTTPGET(ID) γίνετε η μέθοδος η οποία αναζητεί μόνο ένα συγκεκριμένο Group επίσης HTTPDELETE χρησιμοποιείται για την διαγραφή ενός Group και ακολουθούμε την ίδια λογική για όλες τις μεθόδους για τις ομάδες, τις μετρήσεις, τα ποτίσματα και τους κόμβους.

```
[ProducesResponseType(StatusCodes.Status401Unauthorized)]
[ProducesResponseType(StatusCodes.Status400BadRequest, Type = typeof(Error))]

public class GroupController : ApiController
{
    [HttpGet]
    [ProducesResponseType(StatusCodes.Status200OK, Type = typeof(GroupModel[]))]

    public Task<IActionResult> Get([FromQuery] SearchGroups request, CancellationToken token) =>
        SendRequest(request, token);

    [HttpGet("{id}")]
    [ProducesResponseType(StatusCodes.Status200OK, Type = typeof(GroupModel))]
    [ProducesResponseType(StatusCodes.Status404NotFound, Type = typeof(Error))]

    public Task<IActionResult> Get(string id, CancellationToken token) =>
        SendRequest(new FindGroup { Id = id }, token);

    [HttpPost]
    [ProducesResponseType(StatusCodes.Status201Created, Type = typeof(GroupModel))]

    public Task<IActionResult> Create([FromBody] CreateGroup request, CancellationToken token) =>
        SendCreateRequest(nameof(Get), r => new { id = r.Id }, request, token);

    [HttpPatch]
    [ProducesResponseType(StatusCodes.Status200OK, Type = typeof(ModelUpdate))]
    [ProducesResponseType(StatusCodes.Status409Conflict, Type = typeof(Error))]

    public Task<IActionResult> Update([FromBody] UpdateGroup request, CancellationToken token) =>
        SendRequest(request, token);

    [HttpDelete]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType(StatusCodes.Status404NotFound, Type = typeof(Error))]

    public Task<IActionResult> Delete([FromBody] DeleteGroup request, CancellationToken token) =>
        SendRequest(request, token);
}
```

Εικόνα 130: Microservice.RestApi προτζεκτ, GroupController.cs

Όπως και πριν υπάρχει αρκετά παρόμοιος κώδικας και για τους κόμβους και αυτό χάρη στην κοινή χρήση κώδικα από την βιβλιοθήκη μας Microservice.Core, το μόνο που μας ενδιαφέρει είναι η τελική προσαρμογή κώδικα που θα εφαρμόσει την επικοινωνία μεταξύ πελάτη και διακομιστή.

Ο παρακάτω κώδικας χειρίζεται ειδοποιήσεις GroupCreated από την βιβλιοθήκη MediatR. Χρησιμοποιώντας την βοηθητική μέθοδο ShouldSend γνωρίζουμε αν έχουμε ξαναστείλει την ίδια ειδοποίηση αν όχι ειδοποιούμε όλους τους συνδεδεμένους πελάτες στο διακομιστή ότι ένα Group δημιουργήθηκε.

```
0 references
public Task Handle(GroupCreated notification, CancellationToken cancellationToken)
{
    return ShouldSend($"group:create:{notification.Group.Id}", notification.Group.ConcurrencyStamp)
        ? hub.Clients.All.SendAsync("group:added", notification, cancellationToken)
        : Task.CompletedTask;
}
```

Εικόνα 131: Microservice.RestApi προτζεκτ, GroupNotificationHandler.cs 1

Ο παρακάτω κώδικας χειρίζεται ειδοποιήσεις GroupUpdated από την βιβλιοθήκη MediatR. Αν δεν έχουμε ξαναστείλει αυτήν την ειδοποίηση ειδοποιούμε όλους τους συνδεδεμένους πελάτες στον διακομιστή ότι ένα Group έχει ενημερωθεί.

```
0 references
public Task Handle(GroupUpdated notification, CancellationToken cancellationToken)
{
    return ShouldSend($"group:update:{notification.Group.Id}", notification.Group.ConcurrencyStamp)
        ? hub.Clients.All.SendAsync("group:updated", notification, cancellationToken)
        : Task.CompletedTask;
}
```

Εικόνα 132: Microservice.RestApi προτζεκτ, GroupNotificationHandler.cs 2

Ο παρακάτω κώδικας χειρίζεται ειδοποιήσεις GroupDeleted από την βιβλιοθήκη MediatR. Αν δεν έχουμε στείλει αυτήν την ειδοποίηση ειδοποιούμε όλους τους συνδεδεμένους πελάτες στον διακομιστή ότι ένα Group έχει διαγραφθεί.

```
0 references
public Task Handle(GroupDeleted notification, CancellationToken cancellationToken)
{
    return ShouldSend($"group:delete:{notification.GroupId}", notification.Rev)
        ? hub.Clients.All.SendAsync("group:removed", notification, cancellationToken)
        : Task.CompletedTask;
}
```

Εικόνα 133: Microservice.RestApi προτζεκτ, GroupNotificationHandler.cs 3

5.7: Προτζεκτ Microservice.Worker

Παρακάτω βλέπουμε τις ρυθμίσεις τις οποίες χρειάζεται το προτζεκτ Microservice.Worker. Βλέπουμε ότι έχει ανάγκη μόνο για την σύνδεση του στην βάση δεδομένων CouchDB.

```
Services:
  CouchDb: <The couchdb username;password;url example => admin;admin;http://localhost:5984>
```

Εικόνα 134: Microservice.Worker προτζεκτ, appsettings.yaml

Το συγκεκριμένο προτζεκτ μπορεί να εκτελείτε και «On Demand» δηλαδή όποτε θέλουμε εμείς να υπολογίσει τα διάφορα ποτίσματα για κάθε Group.

Το συγκεκριμένο προτζεκτ είναι μια εφαρμογή κονσόλας πιο συγκεκριμένα μια υπηρεσία η οποία ξεκινά και κάθε δεκαπέντε λεπτά θα εκτελεί κώδικα ο οποίος θα υπολογίζει αν ένα Group πρέπει να ξεκινήσει η να σταματήσει να ποτίζει.

Παρακάτω βλέπουμε τον κώδικα ο οποίος εκτελείτε στην αρχή της υπηρεσίας, βλέπουμε ότι χρησιμοποιούμε όλες τις βασικές βιβλιοθήκες όπως Microservice.Core, Microservice.Infrastructure και CoreHost. Δηλώνοντας και τον κώδικα ο οποίος θα εκτελεστεί σαν υπηρεσία με την γραμμή AddHostedService<Worker>(). Τέλος εκτελούμε την εντολή RunCoreHostAsync της βιβλιοθήκης CoreHost έτσι ξεκινάμε την εκτέλεση και καταγραφή της εφαρμογής.

```
await new HostBuilder()
.ConfigureCoreHost(args)
.ConfigureServices((context, services) =>
{
    var conf = context.Configuration;

    services.AddCoreHostServices();
    services.AddMicroserviceCore(conf);
    services.AddMicroserviceInfrastructure(conf);

    services.AddMemoryCache();

    services.AddSingleton<IIrrigator>(s => s.GetRequiredService<Worker>());
    services.AddHostedService<Worker>();
})
.Build()
.RunCoreHostAsync("Worker");
```

Εικόνα 135: Microservice.Worker προτζεκτ, Program.cs

Η βασική μέθοδος που εκτελείτε κάθε δεκαπέντε λεπτά είναι η παρακάτω. Στην αρχή κοιτάμε πάντα αν πρέπει να ακυρώσουμε την εκτέλεση μας μέσο του αντικειμένου «stoppingToken» αν ναι τότε δεν εκτελούμε κώδικα.

Κάθε φορά στην αρχή καταγράφουμε ότι ξεκινήσαμε τον υπολογισμό και δημιουργία ποτίσματος και στο τέλος ότι αυτή η διαδικασία τελείωσε.

Ενδιάμεσα χρησιμοποιώντας την διεπαφή IGroupStore ανακτάμε όλα τα Group τα οποία δεν έχουν τον τύπο “None” δηλαδή έχουν ρυθμιστεί για κάποιο φυτό.

Στην συνέχεια σημειώνουμε τις ημερομηνίες στις οποίες θα αναζητήσουμε μετρήσεις. Το διάστημα το οποίο δημιουργούμε είναι από την επόμενη ημέρα έως μια ώρα πιο πριν από την στιγμή της δημιουργίας της. Αυτό το κάνουμε για να είμαστε σίγουροι ότι θα συμπεριλάβουμε και μετρήσεις που δημιουργήθηκαν την ώρα που εμείς καταγράφαμε την ώρα.

Μετά εκτελώντας την μέθοδο Calculate για κάθε Group δημιουργούμε μια λίστα κόμβων ποτίσματος και το αν πρέπει να ποτίσουν όπως θα δούμε πιο μετά.

Τέλος για κάθε εντολή που υπολογίσαμε εκτελούμε την εντολή Update που θα δούμε πως δημιουργεί καινούριες εντολές ποτίσματος είτε για την εκκίνηση είτε για το σταμάτημα.

```
3 references
public async Task Irrigate(CancellationToken stoppingToken)
{
    if (stoppingToken.IsCancellationRequested) return;

    logger.LogInformation("Irrigation calculation started");

    var groups = (await groupStore.Search(new(), stoppingToken))
        .Where(x => x.Type is not GroupType.None);

    var start = DateTimeOffset.UtcNow.AddDays(1);
    var end = start.AddHours(-1).AddDays(-1);

    var actions = await groups
        .Select(group => Calculate(group, start, end, stoppingToken));

    await actions
        .Select(action => Update(action, stoppingToken));

    logger.LogInformation("Irrigation calculation ended");
}
```

Εικόνα 136: Microservice.Worker προτζεκτ, Worker.cs 1

Η μέθοδος Calculate υπολογίζει για κάθε γκρουπ αν πρέπει να ξεκινήσει η να σταματήσει το πότισμα και επιστρέφει μια λίστα με τα Id των κόμβων οι οποίοι θα λάβουν την εντολή και την εντολή ίδια την εντολή.

Ξεκινώντας αναζητάμε κάθε κόμβο για το συγκεκριμένο Group, μετά αναζητάμε όλες τις μετρήσεις αυτού του Group χρησιμοποιώντας τα Id των κόμβων οι οποίοι είναι κομβόι που μετρήσεων και τις ημερομηνίες που μας έδωσε ο κώδικας που καλεί την εντολή αυτή.

Σε περίπτωση που δεν βρούμε μετρήσεις επιστρέφουμε μια άδεια λίστα με την τιμή false. Αλλιώς στην συνέχεια υπολογίζουμε τον μέσο όρο των τελευταίων μετρήσεων και ανάλογα το τύπο του Group αποφασίζουμε αν πρέπει να ποτίσουμε η όχι. Τέλος επιστρέφουμε την λίστα των κόμβων που ποτίζουμε με την εντολή την οποία υπολογίσαμε.

```
1 reference
private async Task<(string[] Ids, bool Status)> Calculate(
    GroupModel group, DateTimeOffset start, DateTimeOffset end, CancellationToken token)
{
    var nodes = await nodeStore.Search(new() { GroupId = group.Id }, token);

    var measurements = await measurementStore.SearchMany(
        cancellationToken: token,
        request: nodes
            .Where(n => n.Type == NodeType.Measurement)
            .Select(n => new SearchMeasurements
            {
                NodeId = n.Id,
                StartDate = start,
                EndDate = end
            }).ToArray());

    if (measurements.Length == 0) return (Array.Empty<string>(), false);

    var moisture = measurements.Average(m => m.SoilMoisture);

    var status = group.Type switch
    {
        GroupType.Potatoes => moisture <= 50,
        _ => false
    };

    var ids = nodes
        .Where(n => n.Type == NodeType.Irrigation)
        .Select(x => x.Id)
        .ToArray();

    return (ids, status);
}
```

Εικόνα 137: Microservice.Worker προτζεκτ, Worker.cs 2

Η μέθοδος Update δέχεται μια λίστα από Id και την κατάσταση τους δηλαδή αν πρέπει η δεν πρέπει να ξεκινήσουν να ποτίζουν. Στην αρχή ελέγχουμε αν όντως έχουμε μια λίστα από Id αν όχι δεν εκτελούμε κώδικα.

Για να αποτρέψουμε την δημιουργία συνεχομένων εντολών θα δούμε ότι και στην αρχή δημιουργούμε στην προσωρινή μνήμη τις τελευταίες εντολές που στείλαμε για κάθε έναν κόμβο, η πρώτη εντολή που εκτελούμε δημιουργεί την μεταβλητή “tocache” η οποία περιέχει όλα τα Id τα οποία δεν έχουμε υπολογίσει τι πρέπει να κάνουν. Στην συνέχεια για κάθε κόμβο ο οποίος δεν είναι στην λίστα “tocache” του οποίου η τιμή “status” είναι διαφορετική από αυτήν που υπολογίσαμε δημιουργούμε μια νέα εντολή ποτίσματος, αν η τιμή “status” είναι “true” τότε η εντολή θα είναι “Start” αλλιώς θα είναι “Stop” αυτήν την εντολή την αποθηκεύουμε σε μια λίστα.

Για κάθε Id που είναι στην λίστα “tocache” δημιουργούμε πάλι μια ανάλογη εντολή για το “status”. ΜΕΤΚΑ αποθηκεύουμε στην προσωρινή μνήμη για κάθε Id την τιμή “status” και επιστρέφουμε πίσω ένα αντικείμενο Task το οποίο ολοκληρώνετε όταν δημιουργηθούν όλες οι εντολές που δημιουργήσαμε.

```
1 reference
private async Task Update((string[] Ids, bool status) action, CancellationToken token)
{
    var (ids, status) = action;

    if (ids.Length == 0) return;

    var tocache = ids.Where(id => cache.TryGetValue(id, out _) is false).ToHashSet();

    var cachedTask = Task.Run(async () => await ids
        .Where(id => tocache.Contains(id) is false)
        .Where(id => cache.Get<bool>(id) != status)
        .Select(id => irrigationStore.Create(
        cancellationToken: token,
        request: new()
        {
            IssuedAt = DateTimeOffset.UtcNow,
            NodeId = id,
            Type = status ? IrrigationType.Start : IrrigationType.Stop
        })));

    var notCachedTask = Task.Run(async () => await tocache
        .Select(id => irrigationStore.Create(
        cancellationToken: token,
        request: new()
        {
            IssuedAt = DateTimeOffset.UtcNow,
            NodeId = id,
            Type = status ? IrrigationType.Start : IrrigationType.Stop
        })));

    foreach (var id in ids)
    {
        Cache(id, status);
    }

    await Task.WhenAll(cachedTask, notCachedTask);
}
```

Εικόνα 138: Microservice.Worker προτζεκτ, Worker.cs 3

Τέλος η εντολή ExecuteAsync είναι η εντολή που εκτελείτε όταν η εφαρμογή μας ξεκινά. Στην αρχή ανακτάμε όλους τους κόμβους που έχουμε δημιουργήσει και για κάθε κόμβο ανακτάμε τις τελευταίες εντολές ποτίσματος που είχε για τις επτά προηγούμενες μέρες.

Μετά φιλτράρουμε κάθε εντολή δημιουργώντας “Groups” από κάθε εντολή χρησιμοποιώντας το Id του κόμβου για τον οποίο δημιουργήθηκε, ξέροντας ότι είναι κατανεμημένες χρονολογικά γνωρίζουμε οπτή η πρώτη εντολή είναι η πιο πρόσφατη για αυτό και επιλέγουμε την πρώτη, τέλος μετατρέπουμε τις εντολές στο τελευταίο τους “status” δηλαδή “true” αν η εντολή ήταν “Start” αλλιώς “false”.

Για κάθε “status” που υπολογίσαμε το αποθηκεύουμε στην προσωρινή μνήμη.

Τέλος εκτελούμε μια εντολή ποτίσματος και ξεκινάμε ένα αντικείμενο “timer” το οποίο κάθε δεκαπέντε λεπτά θα εκτελεί την μέθοδο Irrigate. Για να μην κλείσει η εφαρμογή μας το επιστρέφουμε ένα αντικείμενο Task το οποίο θα ολοκληρωθεί και θα σημάνει το τέλος της εκτέλεσης μας όταν το “stoppingToken” σταματήσει το αντικείμενο “timer” που χρησιμοποιούμε.

```
0 references
protected override async Task ExecuteAsync(CancellationToken stoppingToken)
{
    var nodes = await nodeStore.Search(new(), stoppingToken);

    var irrigations = await irrigationStore.SearchMany(
        cancellationToken: stoppingToken,
        request: nodes
            .Where(n => n.Type == NodeType.Irrigation)
            .Select(n => new SearchIrrigations
            {
                NodeId = n.Id,
                StartDate = DateTimeOffset.UtcNow.AddDays(1),
                EndDate = DateTimeOffset.UtcNow.AddDays(-8)
            }).ToArray());

    var statuses = irrigations
        .GroupBy(i => i.NodeId)
        .Select(i => i.First())
        .Select(i => (i.NodeId, i.Type == IrrigationType.Start));

    foreach (var (id, status) in statuses)
    {
        Cache(id, status);
    }

    Observable.FromAsync(Irrigate).Subscribe(stoppingToken);
    interval.Subscribe(
        token: stoppingToken,
        onNext: t => Observable.FromAsync(Irrigate).Subscribe(stoppingToken));

    await interval.ToTask(stoppingToken);
}
```

Εικόνα 139: Microservice.Worker προτζεκτ, Worker.cs 4

Παρακάτω βλέπουμε τον κώδικα που εκτελείτε πριν την εντολή ExecuteAsync, και η οποία δημιουργεί το αντικείμενο “Timer”. Η περίοδος την οποία χρησιμοποιεί αυτό το αντικείμενο μπορεί να ρυθμιστεί από τις ρυθμίσεις appsettings.yaml σε περίπτωση που δεν έχει ρυθμιστεί την προ τοποθετούμε στα δεκαπέντε λεπτά ενώ καταγράφουμε και την περίοδο την οποία θα χρησιμοποιεί η εφαρμογή μας.

```
0 references
public Worker(ILogger<Worker> logger,
    INodeStore nodeStore,
    IGroupStore groupStore,
    IIrrigationStore irrigationStore,
    IMeasurementStore measurementStore,
    IMemoryCache cache,
    IConfiguration configuration)
{
    this.logger = logger;
    this.nodeStore = nodeStore;
    this.groupStore = groupStore;
    this.irrigationStore = irrigationStore;
    this.measurementStore = measurementStore;
    this.cache = cache;
    var period = configuration.GetValue("period", 15);
    interval = Observable.Interval(TimeSpan.FromMinutes(period));
    logger.LogInformation("Period set to '{period}'", period);
}
```

Εικόνα 140: Microservice.Worker προτζεκτ, Worker.cs 5

Χρησιμοποιώντας όλα τα παραπάνω καταφέραμε να δημιουργήσουμε την λειτουργία η οποία οδηγεί το αυτόματο πότισμα.

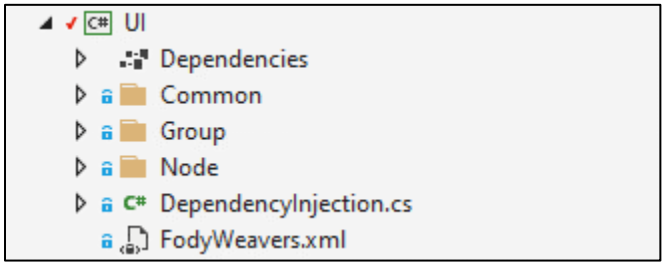
Το πιο σημαντικό από όλα είναι ότι χάρη στην χρήση της βάσης δεδομένων CouchDB η εφαρμογή μας δεν χεριάζετε να καλέσει κάποια άλλη υπηρεσία η πολλές αν αναγκαστούμε να τρέχουμε παραπάνω από έναν διακομιστή για τους κόμβους, αντίθετος απλά δημιουργεί νέα αντικείμενα στην βάση δεδομένων και αυτή με τη σειρά της ενημερώνει οποίο διακομιστεί έχει ζητήσει να ενημερώνετέ για αυτά τα νέα αντικείμενα όπως είδαμε ότι κάνουμε σε όλα τα παραπάνω προτζεκτ.

6: ΙΣΤΟΣΕΛΙΔΑ ΔΙΑΧΕΙΡΗΣΗΣ

Για την ιστοσελίδα διαχείρισης χρησιμοποιούμε σαν βάση την βασική βιβλιοθήκη Microservice και δημιουργούμε μια βιβλιοθήκη βάση και την ιδιά την ιστοσελίδα.

6.1: Βιβλιοθήκη UI

Και εδώ συνεχίζουμε με κάθε φάκελο για κάθε σημαντικό σημείο της εφαρμογής μας. Εσωτερικά κάθε φάκελος περιέχει φακέλους Services και ViewModels.

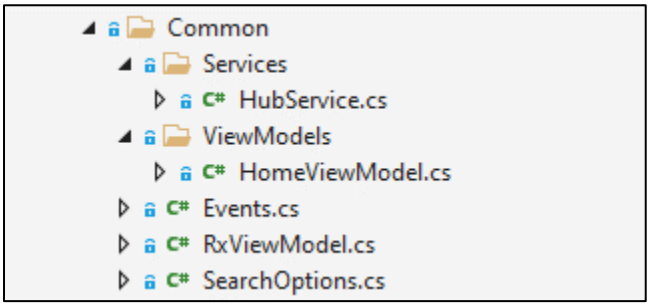


Εικόνα 141: UI προτζεκτ

Χρησιμοποιώντας τα παρακάτω κοινά στοιχεία του φακέλου Common θα δούμε πως χτίσαμε μια βάση στην οποία γράφουμε μόνο κώδικα για την χρήση μας χρησιμοποιώντας μερικά κοινά χαρακτηριστικά.

Πιο σημαντικό όμως είναι πως δημιουργήσαμε κώδικα ο οποίος μπορεί να χρησιμοποιηθεί για να την επίτευξη δημιουργίας οποιαδήποτε εφαρμογής.

Αυτές οι εφαρμογές θα μπορούσανε να είναι για κονσόλα, μια ιστοσελίδα όπως θα δούμε πιο μετά, για Windows, για Android ή και IOS. Όπου χρησιμοποιούμε κώδικα C# και dotnet για την δημιουργία μιας εφαρμογής μπορούμε να κάνουμε χρήση της συγκεκριμένης βιβλιοθήκης ακολουθώντας τις αρχές δημιουργίας εφαρμογής MVVM [46].



Εικόνα 142: UI προτζεκτ, Φάκελος Common

Η πιο βασική μας κλάση είναι η HubService, και αυτό διότι αυτή διαχειρίζεται την σύνδεση με τον διακομιστή μας μέσω της βιβλιοθήκης SignalR. Πέρα από την διαχειριστή της σύνδεσης είναι υπεύθυνη και για την αποστολή όλων των αιτήσεων στον διακομιστή.

Η παρακάτω εικόνα δείχνει το αντικείμενο σύνδεσης το οποίο η δική μας κλάση τυλίγει για να επιτύχουμε τις παραπάνω απαιτήσεις, δίνει έναν τρόπο για να παρακολουθούμε την κατάσταση της συνδέσεις ενώ διαθέτουμε και αντικείμενα στα οποία μπορούμε να κάνουμε «Subscribe» για να εκτελείτε συγκεκριμένος κώδικας για το κάθε συμβάν.

```
15 references
private HubConnection Hub { get; }

0 references
public HubConnectionState State => Hub.State;

2 references
public IObservable<Exception> WhenDisconnected { get; }

2 references
public IObservable<Exception> WhenReconnecting { get; }

2 references
public IObservable<string> WhenReconnected { get; }
```

Εικόνα 143: UI προτζεκτ, HubService.cs 1

Παρακάτω βλέπουμε τον κώδικα που ξεκινά την σύνδεση μας στο διακομιστή. Σε περίπτωση που η σύνδεση δεν είναι στη κατάσταση «Αποσυνδεδεμένη» δεν χεριάζετε να εκτελέσουμε κώδικα αφού η θα είμαστε συνδεδεμένοι η θα προσπαθούμε να επανασυνδεθούμε.

Αν είμαστε αποσυνδεδεμένη δημοσιεύουμε ένα συμβάν ότι ξεκινήσαμε την σύνδεση, εκτελούμε τον κώδικα που θα δημιουργήσει τη σύνδεση της εφαρμογής μας με τον διακομιστή και τέλος δημοσιεύουμε ένα συμβάν ότι είμαστε συνδεδεμένοι στον διακομιστή.

```
1 reference
public async Task ConnectAsync(CancellationToken token = default)
{
    if (Hub.State != HubConnectionState.Disconnected) return;

    Events.Publish(HubConnectionState.Connecting);

    await Hub.StartAsync(token);

    Events.Publish(HubConnectionState.Connected);
}
```

Εικόνα 144: UI προτζεκτ, HubService.cs 2

Όταν δημιουργούμε την κλάση αυτή χρειαζόμαστε την σύνδεση που θα τυλίξουμε για να δώσει την λειτουργικότητα που επιθυμούμε. Αφού γίνει αυτό δηλώνουμε κώδικα ο οποίος θα εκτελείτε όταν η σύνδεση έχει κλίσει, αυτός ο κώδικας περιμένει πέντε δευτερόλεπτα και μετρά ξεκινά την σύνδεση από την αρχή. Μετά μετατρέπουμε όλα τα συμβάντα στα αντίστοιχα αντικείμενα IObservable που είδαμε πιο πριν.

Τέλος δηλώνουμε κώδικα ο οποίος θα εκτελείτε για κάθε συμβάν, αυτοί οι κώδικες απλά δημοσιεύουν σε ολόκληρη την εφαρμογή την κατάσταση της σύνδεσης.

```
0 references
public HubService(HubConnection hub)
{
    Hub = hub;
    Hub.Closed += async (arg) =>
    {
        await Task.Delay(5000);
        await ConnectAsync();
    };

    WhenDisconnected = Observable.FromEvent<Func<Exception, Task>, Exception>(
        converter => args => { converter(args); return Task.CompletedTask; },
        handler => Hub.Closed += handler,
        handler => Hub.Closed -= handler);

    WhenReconnecting = Observable.FromEvent<Func<Exception, Task>, Exception>(
        converter => args => { converter(args); return Task.CompletedTask; },
        handler => Hub.Reconnecting += handler,
        handler => Hub.Reconnecting -= handler);

    WhenReconnected = Observable.FromEvent<Func<string, Task>, string>(
        converter => args => { converter(args); return Task.CompletedTask; },
        handler => Hub.Reconnected += handler,
        handler => Hub.Reconnected -= handler);

    WhenReconnecting.Subscribe(ex => Events.Publish(HubConnectionState.Reconnecting));
    WhenReconnected.Subscribe(ex => Events.Publish(HubConnectionState.Connected));
    WhenDisconnected.Subscribe(ex => Events.Publish(HubConnectionState.Disconnected));
}
```

Εικόνα 1452:UI προτζεκτ, HubService.cs 3

Ο παρακάτω κώδικας επιτρέπει την δήλωση μεθόδων οι οποίες θα εκτελεστούν από τον διακομιστή στον πελάτη. Δέχεται μεθόδους που λαμβάνουν μόνο ένα αντικείμενο σαν είσοδο, παρόλο που η σύνδεση υποστηρίζει παραπάνω αντικείμενα έχουμε δομήσει την εφαρμογή με τέτοιο τρόπο που μόνο με ένα μας αρκεί λόγο της χρήσης αιτήσεων και ειδοποιήσεων της βιβλιοθήκης MediatR.

```
6 references
public void On<T1>(string methodName, Action<T1> handler) =>
    Hub.On(methodName, handler);
```

Εικόνα 1463: UI προτζεκτ, HubService.cs 4

Η παρακάτω μέθοδος είναι η μέθοδος με την οποία στέλνουμε αιτήσεις στον διακομιστή. Δέχεται μια αίτηση και δίνει πίσω ένα αντικείμενο Result το οποίο η θα περιέχει την απάντηση η θα περιέχει ένα σφάλμα.

Πριν γίνει η αποστολή για κάθε αίτηση γίνετε ο έλεγχος της σύνδεσης σε έναν while βρόγχο. Αν η σύνδεση δεν είναι σε κατάσταση «Συνδεδεμένη» τότε περιμένουμε μισό δευτερόλεπτο, αλλιώς ο κώδικας εκτελείτε.

Πριν εκτελεστεί ο κώδικας στη σύνδεση βλέπουμε μια καθυστέρηση ενός δευτερολέπτου αυτό έγινε για την δοκιμή διάφορων στοιχειών στο γραφικό περιβάλλον και θα διαγραφτεί στην κανονική έκδοση.

Τέλος ο κώδικας εκτελείτε σε ένα μπλοκ Try Catch όπου αν υπάρξει οποιοδήποτε σφάλμα θα το χειριστεί η μέθοδος ExceptionHandler η οποία θα επιστρέψει το ανάλογο αντικείμενο Error.

```
17 references
public async Task<Result<TResponse>> SendAsync<TResponse>(
    string methodName, IRequest<TResponse> request, CancellationToken token = default)
{
    while (Hub.State != HubConnectionState.Connected) await Task.Delay(500, token);

    try
    {
        await Task.Delay(1000, token);

        return await Hub.InvokeAsync<TResponse>(methodName, request, token);
    }
    catch (Exception ex)
    {
        return ExceptionHandler(ex);
    }
}
```

Εικόνα 147: UI προτζεκτ, HubService.cs 5

Η συγκεκριμένη μέθοδος ελέγχει αν το C# σφάλμα προήλθε από τον διακομιστή, αν ναι ψάχνει να δει αν έχουμε τοποθέτηση ένα αντικείμενο Error μέσα στο μήνυμα το οποίο το τοποθετούμε μετά τον χαρακτήρα «~». Αν ναι επιστρέφει αυτό το αντικείμενο αλλιώς δημιουργεί ένα καινούργιο με το συγκεκριμένο μήνυμα με τίτλο «Unknown Error».

```
1 reference
protected static Error ExceptionHandler(Exception exception) => exception switch
{
    HubException { Message: var msg } => msg.Contains('~')
            ? JsonSerializer.Deserialize<Error>(msg.Split('~')[1])!
            : new Error(title: "Dashboard Hub error", detail: msg, status: exception.HResult),

    _ => new Error(title: "Unknown Error", detail: exception.Message, status: exception.HResult)
};
```

Εικόνα 148: UI προτζεκτ, HubService.cs 6

Ένα παράδειγμα χρήσης του κώδικα HubService είναι στην κλάση GroupService η οποία την χρησιμοποιεί για να στέλνει τις δικές τις εντολές στον διακομιστή. Το ίδιο κάναμε και για το NodeService.

Στην εικόνα βλέπουμε την κλάση GroupService η οποία απλά χεριάζετε το αντικείμενο HubService για να στείλει συγκεκριμένες αιτήσεις και να διαχειριστεί ειδοποιήσεις σχετικές με τα αντικείμενα Group.

Χρησιμοποιώντας την βιβλιοθήκη DynamicData [51] δημιουργούμε ένα αντικείμενο το οποίο θα αποθηκεύει προσωρινά όλα τα Group. Όλα τα Group θα ενημερώνονται ανάλογα με τις ειδοποιήσεις που λαμβάνουμε από τον διακομιστή ενώ μας δίνει την δυνατότητα να εκφράζουμε αυτές τις αλλαγές στο γραφικό μας περιβάλλον αφού η βιβλιοθήκη είναι μέρος της ReactiveUI και βοηθά στην ομαλή σύνδεση της τεχνικής MVVM και των δεδομένων από οποιαδήποτε πηγή.

Δηλώνουμε καλώντας την μέθοδο «On» μια μέθοδο για κάθε πιθανή ειδοποίηση. Στις μεθόδους που δηλώσαμε απλά εκτελούμε την ανάλογη εντολή στην πηγή την οποία κρατάμε στην μνήμη και αυτή με την σειρά της θα μεταφέρει τις ανάλογες αλλαγές.

```
private bool tryLoad = true;

private readonly SourceCache<GroupModel, string> _groups = new(group => group.Id);
private readonly HubService hub;

0 references
public GroupService(HubService hub)
{
    hub.On<GroupCreated>("group:added", GroupAdded);
    hub.On<GroupUpdated>("group:updated", GroupUpdated);
    hub.On<GroupDeleted>("group:removed", GroupRemoved);
    this.hub = hub;
}

/// <inheritdoc/>
2 references
public IEnumerable<GroupModel> Cache...

1 reference
private void GroupAdded(GroupCreated notification) =>
    _groups.AddOrUpdate(notification.Group);

1 reference
private void GroupUpdated(GroupUpdated notification) =>
    _groups.AddOrUpdate(notification.Group);

1 reference
private void GroupRemoved(GroupDeleted notification) =>
    _groups.RemoveKey(notification.GroupId);
```

Εικόνα 149: UI προτζεκτ, GroupService.cs 1

Η μέθοδος Connect χρησιμοποιείται για την σύνδεση μας με την λίστα την οποία κρατάμε στην μνήμη. Η λίστα μας στέλνει αλλαγές, έτσι κάνοντας χρήση αυτής της μεθόδου μπορούμε να λάβουμε όλα τα αντικείμενα Group και να συνεχίσουμε να λαμβάνουμε ενημερώσεις σχετικά με αυτά ενώ λαμβάνουμε ενημερώσεις σε περίπτωση που γίνει μια προσθήκη η και αφαίρεση από την λίστα.

Χρησιμοποιούμε την μεταβλητή “tryLoad” η οποία αν είναι “true” θα προσπαθήσει να φορτώσει για πρώτη φορά όλα τα αντικείμενα Group έτσι ώστε να τα έχουμε στην λίστα μας. Σε περίπτωση οποιαδήποτε αποτυχίας θέτουμε την μεταβλητή σε κατάσταση “false” για να ξανά προσπαθήσουμε την επόμενη φορά.

```
/// <inheritdoc/>
3 references
public IObservable<IChangeSet<GroupModel, string>> Connect(Func<GroupModel, bool>? predicate = null)
{
    if (tryLoad)
    {
        tryLoad = false;
        _ = Task.Run(async () =>
        {
            var result = await Search(new());
            result.Switch(
                s => _groups.EditDiff(s, static (l, r) => l.Id == r.Id),
                e =>
                {
                    Console.WriteLine(e);
                    tryLoad = true;
                });
        });
    }

    return _groups.Connect(predicate);
}
```

Εικόνα 150: UI προτζεκτ, GroupService.cs 2

Η μέθοδος Watch κάνει την ιδιά λειτουργία με την μέθοδο Connect μόνο που αντί να λαμβάνουμε ειδοποιήσεις αλλαγής για ολόκληρη την λίστα λαμβάνουμε ειδοποιήσεις μόνο για ένα συγκεκριμένο αντικείμενο μέσα στην λίστα.

```
/// <inheritdoc/>
2 references
public IObservable<GroupModel> Watch(string id) =>
    _groups.WatchValue(id);
```

Εικόνα 151: UI προτζεκτ, GroupService.cs 3

Η μέθοδος Get φροντίζει να φορτώσει το οποιοδήποτε Group από την λίστα μας, αν αυτή δεν το έχει τότε προσπαθεί να το ανακτήσει από τον διακομιστή, πριν επιστρέψει το αποτέλεσμα Result η ιδιά η μέθοδος δοκιμάζει να δει αν είχαμε ένα επιτυχές αποτέλεσμα, στην περίπτωση αυτή θα κάνει εισχώρηση του Group στην λίστα μας αλλιώς θα καταχωρήσει το σφάλμα και μετά θα επιστρέψει το αποτέλεσμα Result.

```
/// <inheritdoc/>
3 references
public async ValueTask<Result<GroupModel>> Get(string id)
{
    var lookup = _groups.Lookup(id);

    if (lookup.HasValue) return lookup.Value;

    var result = await hub.SendAsync("group:get", new FindGroup(id));
    result.Switch(
        s => _groups.AddOrUpdate(s),
        e => Console.WriteLine(e));

    return result;
}
```

Εικόνα 152: UI προτζεκτ, GroupService.cs 4

Οι επόμενες μέθοδοι εκτελούνται απευθείας στον διακομιστή, μιας και η αναζήτηση πρέπει να περιλαβαίνει όλα τα γκρουπ, ενώ αν δημιουργηθεί, ενημερωθεί η και διαγραφθεί με επιτυχία ένα Group ο διακομιστής θα μας ενημερώσει και οι μέθοδοι που δηλώσαμε στην αρχή θα εκτελεστούν και θα διαχειριστούν το ανάλογο συμβάν.

```
/// <inheritdoc/>
2 references
public Task<Result<GroupModel[]>> Search(SearchGroups request) =>
    hub.SendAsync("group:search", request);

/// <inheritdoc/>
2 references
public Task<Result<GroupModel>> Create(CreateGroup request) =>
    hub.SendAsync("group:create", request);

/// <inheritdoc/>
2 references
public Task<Result<ModelUpdate>> Update(UpdateGroup request) =>
    hub.SendAsync("group:update", request);

/// <inheritdoc/>
2 references
public Task<Result<Unit>> Delete(DeleteGroup request) =>
    hub.SendAsync("group:delete", request);
```

Εικόνα 153: UI προτζεκτ, GroupService.cs 5

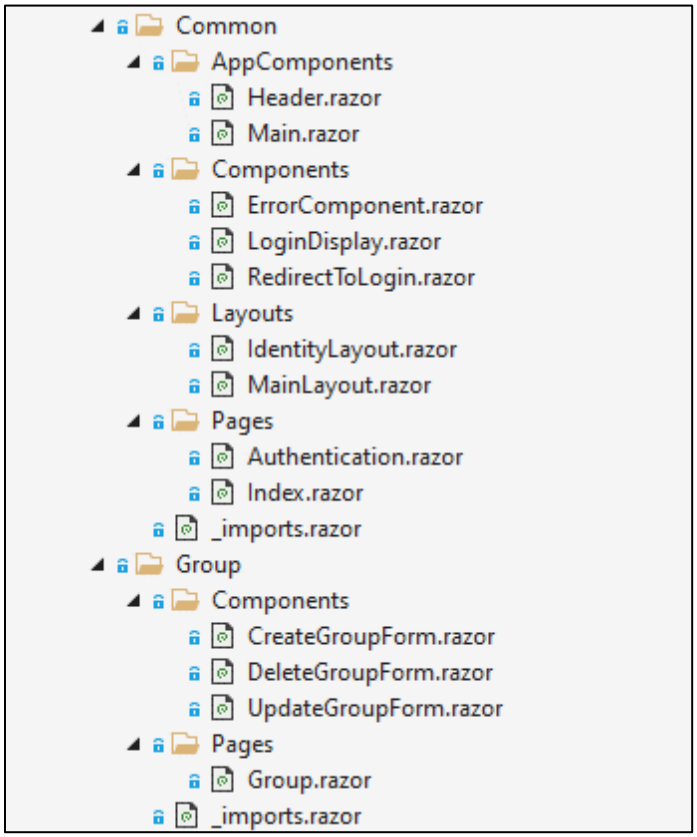
Τέλος οποιαδήποτε εφαρμογή για να εκμεταλλευτεί την βασική βιβλιοθήκη πρέπει μόνο να καλέσει την μέθοδο AddUI, αυτή θα φροντίσει να δηλώσει τα απαραίτητα Services και ViewModels.

```
1 reference
public static IServiceCollection AddUI(this IServiceCollection services, IConfiguration configuration)
{
    // Add services
    services.AddSingleton<HubService>();
    services.AddSingleton<IGroupService, GroupService>();
    services.AddSingleton<INodeService, NodeService>();

    // Add ViewModels
    services.AddScoped<HomeViewModel>();

    return services;
}
```

Εικόνα 154: UI προτζεκτ, DependencyInjection.cs

6.1: Προτζεκτ UI.Web

Στη παρακάτω εικόνα βλέπουμε ένα μέρος του προτζεκτ UI.Web, παρόμοιο με όλα τα αλλά ο φάκελος Common διαθέτει ότι χρειαζόμαστε για να δημιουργήσουμε την εφαρμογή με φακέλους όπως του Group να διαθέτουν μόνο τα απαραίτητα εργαλεία για το συγκεκριμένο αντικείμενο.



Εικόνα 155: UI.Web προτζεκτ

Παρακάτω βλέπουμε τις ρυθμίσεις της διαδικτυακής μας εφαρμογής. Αυτές οι ρυθμίσεις φορτώνονται μαζί με την εφαρμογή μας αυτόματα και περιλαμβάνουν:

- Όλες τις ρυθμίσεις ταυτοποιήσεις «OpenID»
- Την διεύθυνση του διακομιστή RestAPI.

```
{
  "OpenId": {
    "ClientId": "blazor",
    "Authority": "<The full url of the identity service>",
    "PostLogoutRedirectUri": "<The url of this app>",
    "ResponseType": "code",
    "ResponseMode": "query",
    "DefaultScopes": [ "roles" ]
  },
  "Services": {
    "Api": "<The full url of the rest api service>"
  }
}
```

Εικόνα 156: UI.Web προτζεκτ, appsettings.json

Παρακάτω βλέπουμε πως και στην εφαρμογή ταυτοποίησης βασικά αντικείμενα Blazor. Το αντικείμενο CascadingAuthenticationState κάνει διαθέσιμη την κατάσταση ταυτοποίησης του χρήστη, το αντικείμενο Router φροντίζει να φορτώνει ανάλογες σελίδες όταν περιηγούμαστε σε αυτές. Όταν η σελίδα βρεθεί θα δείξει ότι έχει μέσα το αντικείμενο Found στο οποίο κοιτάμε τα στοιχεία ταυτοποιήσεις του χρήστη και η τον κατευθύνουμε στην σελίδα ταυτοποιήσεις η του δείχνουμε ένα μήνυμα ότι δεν έχει δικαίωμα να δει αυτήν τη σελίδα. Αν η σελίδα δεν βρεθεί δείχνει το αντικείμενο NotFound το οποίο δείχνει ένα μήνυμα ότι η σελίδα που αναζητάμε δεν βρέθηκε.

```
<CascadingAuthenticationState>
    <Router AppAssembly="typeof(MainLayout).Assembly">
        <Found Context="routeData">
            <AuthorizeRouteView RouteData="routeData" DefaultLayout="typeof(MainLayout)">
                <NotAuthorized>
                    @if (context.User.Identity is { IsAuthenticated: false })
                    {
                        <RedirectToLogin />
                    }
                    else
                    {
                        <LayoutView Layout="typeof(IdentityLayout)">
                            <p>You are not authorized to access this resource.</p>
                        </LayoutView>
                    }
                </NotAuthorized>
            </AuthorizeRouteView>
        </Found>
        <NotFound>
            <p>Sorry, there's nothing at this address.</p>
        </NotFound>
    </Router>
</CascadingAuthenticationState>
```

Εικόνα 157: UI.Web προτζεκτ, App.razor

Ο παρακάτω κώδικας είναι ο πρώτος που εκτελεί η εφαρμογή μας πριν δείξει κάποιο γραφικό στοιχείο. Εδώ δηλώνουμε «κλάσεις» που θα χρησιμοποιήσουμε μετά ενώ κατευθύνουμε την εφαρμογή να αντικαταστήσει ένα αντικείμενο “app” της HTML με τον κώδικα που είδαμε παραπάνω.

Στο τέλος της εικόνας βλέπουμε ότι ρυθμίζουμε την σύνδεση SignalR με την ρύθμιση “API” ενώ τα ταυτόχρονα ρυθμίζουμε να κάνει αυτόματα επανασύνδεση και να στέλνει σε κάθε μήνυμα το “token” ταυτοποίησης του χρήστη.

```
var builder = WebAssemblyHostBuilder.CreateDefault(args);
builder.RootComponents.Add<App>("app");

var services = builder.Services;
var configuration = builder.Configuration;

services.AddCoreWeb();
services.AddUI(configuration);

services
    .AddSingleton<HubConnection>(sp => new HubConnectionBuilder()
    .WithUrl(new Uri(Path.Combine(configuration.GetService("api"), "v1/dashboard")), options =>
    {
        options.AccessTokenProvider = async () =>
        {
            var provider = sp.CreateScope().ServiceProvider.GetRequiredService<IAccessTokenProvider>();
            var result = await provider.RequestAccessToken();
            AccessToken token;

            while (!result.TryGetToken(out token))
            {
                Console.WriteLine("Getting token");
                await Task.Delay(2000);
                result = await provider.RequestAccessToken();
            }
            return token.Value;
        };
    })
    .WithAutomaticReconnect()
    .Build());
```

Εικόνα 158: UI.Web προτζεκτ, Program.cs 1

Παρακάτω με την εντολή αυτή φορτώνουμε της ρυθμίσεις “OpenID” στην βιβλιοθήκη ταυτοποιήσεις και τις θέτουμε ότι ο ρόλος ενός χρήστη βρίσκεται στο κλειδί “role”. Εστί η βιβλιοθήκη ταυτοποιήσεις έχει ρυθμιστεί και ξέρει πως να διαχειριστεί την ταυτοποίηση του χρήστη μας.

```
services.AddOidcAuthentication(options =>
{
    options.UserOptions.RoleClaim = "role";

    // Note: response_mode=fragment is the best option for a SPA. Unfortunately, the Blazor WASM
    // authentication stack is impacted by a bug that prevents it from correctly extracting
    // authorization error responses (e.g error=access_denied responses) from the URL fragment.
    // For more information about this bug, visit https://github.com/dotnet/aspnetcore/issues/28344.
    configuration.Bind("OpenId", options.ProviderOptions);
});

return builder.Build().RunAsync();
```

Εικόνα 159: UI.Web προτζεκτ, Program.cs 2

Παρακάτω βλέπουμε τον κώδικα για την μπάρα της εφαρμογής που διαθέτει ένα κουμπί για επιστροφή στην αρχική σελίδα, το αντικείμενο LoginDisplay το οποίο δείχνει τα στοιχεία του χρήστη και ένα κουμπί αποσύνδεσης και τέλος το κουμπί αλλαγής θέματος από σκούρο σε φωτεινό η και το αντίθετο.

Ο κώδικας είναι ανάλογος για το κάθε κουμπί, ενώ εκτελούμε κώδικα για την σωστή τοποθέτηση του θέματος στο κουμπί την πρώτη φορά που η σελίδα μας φορτώνει. Τέλος όταν η σελίδα δημιουργείτε και οι παράμετροι έχουν τοποθετηθεί στην μέθοδο OnParametersSet προσθέτουμε τον δικό μας κώδικα CSS στο αντικείμενο CSS το οποίο χρησιμοποιείται στο header.

```
<header class="@css">
    <button class="px-4" @onclick="HomeClick">
        <i class="bi bi-house-fill"></i>
    </button>

    <LoginDisplay Class="ml-auto" />

    <button class="px-4" @onclick="Toggle">
        <i class="bi @(IsLight ? "bi-moon-fill" : "bi-sun")"></i>
    </button>
</header>

@code{
    private bool IsLight = false;

    private void HomeClick() => nav.NavigateTo("/");

    private async void Toggle()
    {
        IsLight = await theme.Toggle() == ThemeJs.Light;
        StateHasChanged();
    }

    protected async override Task OnAfterRenderAsync(bool firstRender)
    {
        if (!firstRender) return;

        IsLight = await theme.GetTheme() == ThemeJs.Light;
        StateHasChanged();
    }

    protected override void OnParametersSet()
    {
        base.OnParametersSet();
        css = CreateCss("content header flex items-center").AddClass(Class);
    }
}
```

Εικόνα 160: UI.Web προτζεκτ, Header.razor

Το αντικείμενο main απλά θέτει τον κώδικα CSS και δείχνει ότι αντικείμενο του δοθεί με την μεταβλητή ChildContent. Το αντικείμενο αυτό είναι εκεί που θα εμφανίζονται όλες οι σελίδες έτσι δεν χεριάζετε σε κάθε σελίδα να χρησιμοποιούμε το αντικείμενο header.

```
<section class="@css">
    @ChildContent
</section>

@code{
    [Parameter]
    public RenderFragment? ChildContent { get; set; }

    protected override void OnParametersSet()
    {
        base.OnParametersSet();

        css = CreateCss("content main overflow-hidden").AddClass(Class);
    }
}
```

Εικόνα 161: UI.Web προτζεκτ, Main.razor

Το MainLayout είναι ο αρχικός σχεδιασμός της σελίδας. Εδώ αν ο χρήστης δεν είναι συνδεδεμένος τον προωθούμε στην σελίδα εισόδου αλλιώς εμφανίζουμε τα αντικείμενα Header και το Main, στο οποίο Main του δίνουμε το «σώμα» τις εφαρμογής, τις σελίδες μας τις οποίες με την σειρά του η Main θα δείξει.

```
<AuthorizeView>
<NotAuthorized>
    <RedirectToLogin />
</NotAuthorized>

@*Run the whole app only when the user is authorized.*@
<Authorized>
    <Header />
    <Main ChildContent="Body!" />

    @code{
        protected override async Task OnInitializedAsync()
        {
            Events.WhenConnection().Subscribe(c => Console.WriteLine(c.ToString()));
            await hub.ConnectAsync(default);
        }
    }
</Authorized>
</AuthorizeView>
```

Εικόνα 162: UI.Web προτζεκτ, MainLayout.razor

Το αρχείο σχεδιασμού σελίδας IdentityLayout χρησιμοποιείτε για να δείχνει πληροφορίες σχετικά με την σελίδα ταυτοποιήσεις. Για αυτό και είναι τόσο απλό διότι χρειαζόμαστε μόνο τα μηνύματα να φαίνονται στον χρήστη και τίποτε άλλο.

```
<div class="flex justify-center p-4 rounded-xl shadow bg-alt">
    @Body
</div>

@code{

}
```

Εικόνα 163: UI.Web προτζεκτ, IdentityLayout.razor

Το αντικείμενο LoginDisplay δημιουργεί δυο κουμπιά, ένα με το όνομα του χρήστη και ένα για την αποσύνδεση του, αυτό μόνο αν ο χρήστης έχει τακτοποιηθεί. Ο κώδικα εκτελεί την ανάλογη ανακατεύθυνση του χρήστη ανάλογα με το ποιο κουμπί πάτησε.

```
<AuthorizeView>
    <Authorized>
        <div class="@css">
            <button class="font-bold text-gray-100 btn mr-0 rounded-r-none" @onclick="GoToUserProfile">
                <i class="bi bi-person"></i> @context.User.Identity!.Name
            </button>
            <button class="font-bold text-gray-100 btn-cancel ml-0 rounded-l-none" @onclick="BeginSignOut">
                <i class="bi bi-box-arrow-right"></i> Log out
            </button>
        </div>
    </Authorized>
</AuthorizeView>

@code{
    private CssBuilder css;

    private void GoToUserProfile()
    {
        Navigation.NavigateTo("authentication/profile");
    }

    private async Task BeginSignOut(MouseEventArgs args)
    {
        await SignOutManager.SetSignOutState();
        Navigation.NavigateTo("authentication/logout");
    }

    [Parameter]
    public string Class { get; set; } = string.Empty;

    protected override void OnParametersSet()
    {
        base.OnParametersSet();
        css = CssBuilder.Default("flex text-sm").AddClass(Class);
    }
}
```

Εικόνα 164: UI.Web προτζεκτ, LoginDisplay.razor

Το αντικείμενο RedircetToLogin, αν εμφανιστεί σε μια σελίδα, δηλαδή γίνει διαθέσιμο και «ζωγραφιστεί» τότε με το που εκτελεστεί ο κώδικας προ τοποθέτησης του θα κάνει ανακατεύθυνση στην σελίδα εισόδου.

```
@inject NavigationManager Navigation
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@code{
    protected override void OnInitialized()
    {
        Navigation.NavigateTo($"authentication/login?returnUrl={Uri.EscapeDataString(Navigation.Uri)}");
    }
}
```

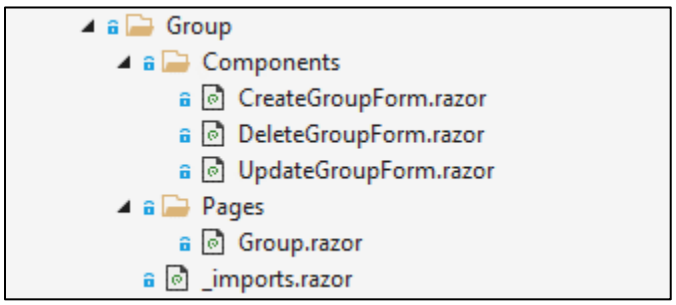
Εικόνα 165: UI.Web προτζεκτ, RedirectToLogin.razor

Παρακάτω βλέπουμε την δομή ενός φακέλου προορισμένος για οτιδήποτε έχει να κάνει με τα γκρουπ. Παρόμοιος φάκελος υπάρχει και για τους κόμβους μας.

Αυτοί οι φάκελοι πέρα από το αρχείο _imports.razor το οποίο μας βοηθά να μην εισάγουμε συνέχεια τα ίδια namespace στην αρχή κάθε αρχείου έχουν φακέλους και αντικείμενα μόνο σχετικά με τον αρχικό φάκελο.

Ο φάκελος Components περιέχει αντικείμενα τα οποία μπορούμε να εμφανίσουμε οπουδήποτε, σε άλλα αντικείμενα η και σελίδες.

Ο φάκελος Pages περιέχει αντικείμενα τα οποία προορίζονται να γίνουν σελίδες, δηλαδή αντικείμενα στα οποία μπορούμε να περιηγηθούμε μέσω της διεύθυνσης του περιηγητή μας.



Εικόνα 166: UI.Web προτζεκτ, Φάκελος Group

Παρακάτω βλέπουμε την φόρμα δημιουργίας ενός γκρουπ. Χρησιμοποιώντας τα βασικά αντικείμενα της βιβλιοθήκης Microservice και CoreWeb το μόνο που μένει να κάνουμε είναι να δημιουργήσουμε τα διάφορα πεδία που πρέπει να συμπληρωθούν και το κουμπί εκτέλεσης μαζί με τον κώδικα του.

```
<EditForm EditContext="form">
    <FluentValidationValidator Validator="form.Validator" />

    <fieldset>
        <legend>Creating Group</legend>

        <div>
            <label>Name</label>
            <InputText @bind-Value="form.Request.Name" />
            <ValidationMessage class="block text-red-400" For="() => form.Request.Name" />
        </div>

        <div>
            <label>Description</label>
            <InputTextArea rows="2" @bind-Value="form.Request.Description" />
            <ValidationMessage For="() => form.Request.Description" />
        </div>

        <div>
            <label>Type</label>
            <InputSelect @bind-Value="form.Request.Type">
                @foreach (var type in Enum.GetValues(typeof(GroupType)))
                {
                    <option value="@type">@type</option>
                }
            </InputSelect>
            <ValidationMessage class="w-full" For="() => form.Request.Type" />
        </div>
    </fieldset>
</EditForm>

<div class="flex justify-end space-x-2">
    <SpinnerButton Class="btn text-gray-100" Command="form.Command">Create</SpinnerButton>
    <button type="button" class="btn-cancel text-gray-100" @onclick="OnCancelClick">Cancel</button>
</div>
```

Εικόνα 167: UI.Web προτζεκτ, CreateGroupForm.razor 1

Παρακάτω βλέπουμε την δημιουργία του αντικειμένου FormContext, το οποίο θα κάνει την επιβεβαίωση των στοιχείων πριν την εκτέλεση της εντολής GroupService.Create και όταν τελειώσει η εκτέλεση θα εκτελεστεί ο κώδικας result.Switch ο οποίος εκτελεί την μέθοδο Success η σε περίπτωση αποτυχίας εκτελεί την μέθοδο Failure.

Η μέθοδος Success δεν χεριάζετε να κάνει κάτι αφού στην περίπτωση δημιουργίας ο διακομιστής θα φροντίσει να μας στείλει ειδοποίηση δημιουργίας μαζί με το καινούργιο αντικείμενο.

Η μέθοδος Failure απλά καταγραφεί το σφάλμα στην κονσόλα του περιηγητή.

```
@code{
    [Parameter]
    public EventCallback<MouseEventArgs> OnCancelClick { get; set; }

    private FormContext<CreateGroup, CreateGroupValidator, Result<GroupModel>> form;

    public CreateGroupForm()
    {
        form = new(
            request => groupService.Create(request),
            result => result.Switch(Success, Failure));
    }

    protected void Success(GroupModel result)
    {

    }

    protected void Failure(Error error)
    {
        Console.WriteLine(error);
    }
}
```

Εικόνα 168: UI.Web προτζεκτ, CreateGroupForm.razor 2

Το αρχείο DeleteGroupForm.razor είναι μια φόρμα η οποία επιβεβαιώνει την διαγραφή ενός γκρουπ. Διαθέτει δυο κουμπιά το κουμπί διαγραφής που θα εκτελέσει την εντολή και το κουμπί ακυρώσεις το οποίο εκτελεί ότι κώδικα του δοθεί.

```
<div>
    <h1 class="text-center">Delete this node?</h1>
    
    <hr />

    <h2 class="text-center">It won't be possible to undo this action!</h2>

    <div class="mt-5 flex justify-center space-x-2">
        <SpinnerButton Class="btn text-gray-100" Command="form.Command">Delete</SpinnerButton>
        <button type="button" class="btn-cancel text-gray-100" @onclick="OnCancelClick">Cancel</button>
    </div>
</div>
```

Εικόνα 169: UI.Web προτζεκτ, DeleteGroupForm.razor 1

Ο κώδικας παρακάτω διαθέτη το FormContext το οποίο οργανώνει την εκτέλεση της εντολής, του γκρουπ προς διαγραφή ένα προαιρετικό μήνυμα και την εντολής που εκτελείτε με το πάτημα από το κουμπί ακύρωσης. Όταν δημιουργείτε το αντικείμενο δημιουργούμε το FormContext με τις εντολές Εκτέλεσης (Execute) και Αποτελέσματος(Result). Η εντολής Execute διαγραφεί οποιοδήποτε μήνυμα και στέλνει στον διακομιστή την εντολής διαγραφής μέσο του IGroupService. Η εντολή Result εκτελεί κώδικας ανάλογα με το αποτέλεσμα, αν η εκτέλεση έγινε με επιτυχία οδηγούμε τον χρήστη στην αρχική σελίδα, αν όμως υπήρξε κάποιο σφάλμα τότε θέτουμε την παράμετρο Message στο ανάλογο σφάλμα.

```
@code{
    private FormContext<DeleteGroup, DeleteGroupValidator, Result<Unit>> form;
    private GroupModel group { get; set; } = new();
    private string? message;

    [Parameter]
    public EventCallback OnCancelClick { get; set; }

    [Parameter]
    public GroupModel Group
    {
        get => group;
        set
        {
            group = value;
            form.Reset(new DeleteGroup(group));
            StateHasChanged();
        }
    }

    public DeleteGroupForm() => form = new(Execute, Result);

    private Task<Result<Unit>> Execute(DeleteGroup request)
    {
        message = null;
        return groupService.Delete(request);
    }

    private void Result(Result<Unit> r) => r.Switch(
        result => nav.NavigateTo("/"),
        error => message = error.Title);
}
```

Εικόνα 170: UI.Web προτζεκτ, DeleteGroupForm.razor 2

Το αρχείο UpdateGroupForm.razor.razor.razor περιέχει το αντικείμενο το οποίο διαθέτει μια φόρμα για την ενημέρωση κάποιον στοιχείων ενός γκρουπ. Πιο συγκεκριμένα διαθέτει την αλλαγή του ονόματος, της περιγραφής και το τύπο ενός γκρουπ. Παρακάτω βρίσκονται τα κουμπιά ενημερώσεις και ακύρωσης.

```
<EditForm EditContext="form">
    <FluentValidationValidator Validator="form.Validator" />

    <fieldset>
        <legend>Update Group</legend>

        @if (messages is not null)
        {
            <p><u><b>@messages</b></u></p>
        }

        <div>
            <label>Name</label>
            <InputText @bind-Value="form.Request.Name" />
            <ValidationMessage class="block text-red-400" For="() => form.Request.Name" />
        </div>

        <div>
            <label>Description</label>
            <InputTextArea @bind-Value="form.Request.Description" />
            <ValidationMessage For="() => form.Request.Description" />
        </div>

        <div>
            <label>Type</label>
            <InputSelect @bind-Value="form.Request.Type">
                @foreach (var type in Enum.GetValues(typeof(GroupType)))
                {
                    <option value="@type">@type</option>
                }
            </InputSelect>
            <ValidationMessage class="w-full" For="() => form.Request.Type" />
        </div>
    </fieldset>

    <div class="flex justify-end space-x-2">
        <SpinnerButton Class="btn text-gray-100" Command="form.Command">Update</SpinnerButton>
        <button type="button" class="btn-cancel text-gray-100" @onclick="OnCancelClick">Cancel</button>
    </div>
</EditForm>
```

Εικόνα 171: UI.Web προτζεκτ, UpdateGroupForm.razor 1

Το αντικείμενο μας διαθέτει το FormContext, το γκρουπ προς ενημέρωση και ένα προαιρετικό μήνυμα. Όταν το αντικείμενο μας δημιουργείτε δημιουργούμε και το αντικείμενο FormContext με τις εντολές Εκτέλεσης (Execute), Αποτελέσματος (Result) και το αν πρέπει να σταλθεί η εντολή (ShouldSentRequest). Η εντολή SetGroup φροντίζει να ενημερώνει το γκρουπ το οποίο η φόρμα διαθέτει εσωτερικά. Η εντολή Execute στέλνει την εντολή ενημέρωσης στον διακομιστή χρησιμοποιώντας την μέθοδο "GenerateRequest()". Η εντολή Result αν δεν έχουμε σφάλμα θέτει το μήνυμα σε "Group updated" και ενημερώνει την σελίδα ότι άλλαξε η κατάσταση του γκρουπ, αν έχουμε σφάλμα τότε το μήνυμα γίνετε ο τίτλος του σφάλματος και το καταγράφουμε στην κονσόλα του περιηγητή.

```
@code{
    private FormContext<UpdateGroup, UpdateGroupValidator, Result<ModelUpdate>> form;
    private GroupModel group = null!;
    private string? messages;

    [Parameter]
    public EventCallback<MouseEventArgs> OnCancelClick { get; set; }

    [Parameter]
    public GroupModel Group
    {
        get => group;
        set => SetGroup(value);
    }


    public UpdateGroupForm()
    {
        form = new(Execute, Result, ShouldSentRequest);
    }

    public void SetGroup(GroupModel group)
    {
        this.group = group;
        messages = null;
        form.Reset(GenerateForm());

        StateHasChanged();
    }

    private Task<Result<ModelUpdate>> Execute(UpdateGroup request) =>
        groupService.Update(GenerateRequest());

    private void Result(Result<ModelUpdate> r) => r.Switch(
        result =>
        {
            messages = "Group updated!";
            StateHasChanged();
        },
        error =>
        {
            messages = error.Title;
            Console.WriteLine(error);
        });
```

Εικόνα 172: UI.Web προτζεκτ, UpdateGroupForm.razor 2

Η εντολή GenerateForm δημιουργεί το αντικείμενο που θα δείχνει η φόρμα μας στο χρήστη. Η εντολή ShouldSentRequest κάνει τον έλεγχο αν η φόρμα μας είναι διαφορετική από το γκρουπ μας, αν όχι τότε δεν υπάρχει λόγος αποστολής. Τέλος η εντολή GenerateRequest δημιουργεί ένα αντικείμενο ενημερώσεις που θα διαθέτει μόνο τις αλλαγμένες μεταβλητός του γκρουπ.

```
private UpdateGroup GenerateForm() => new()
{
    Id = Group.Id,
    Type = Group.Type,
    ConcurrencyStamp = Group.ConcurrencyStamp,
    Name = Group.Name,
    Description = Group.Description,
};

private bool ShouldSentRequest()
{
    var form = this.form.Request;
    var group = Group;

    var send =
        form.Name is { Length: > 0 } &&
        form.Name != group.Name ||
        form.Type != group.Type ||
        form.Description != group.Description;


    messages = send ? null : "Change something to send!";
    return send;
}

private UpdateGroup GenerateRequest()
{
    var form = this.form.Request;
    var group = Group;

    var request = new UpdateGroup(group);
    if (form.Name != group.Name)
    {
        request.Name = form.Name;
    }
    if (form.Type != group.Type)
    {
        request.Type = form.Type;
    }
    if (form.Description != group.Description)
    {
        request.Description = form.Description;
    }
    return request;
}
```

Εικόνα 173: UI.Web προτζεκτ, UpdateGroupForm.razor 3

Το GroupViewModel.cs είναι η κλάση η οποία φροντίζει να "δείξει" στον χρήστη όλες τις πληροφορίες για ένα γκρουπ. Διαθέτει τις "Reactive" μεταβλητές, (μεταβλητές η οποίες μπορεί να αλλάξουν) Group, Nodes, NodeData και SearchOptions.

Το Group είναι το γκρουπ προς προβολή στο χρήστη, και στην αρχή όταν το φορτώνουμε έχει την τιμή null ενώ μετά μόλις φορτώσει θα είναι το γκρουπ που ψάχναμε.

Το Nodes είναι οι διάφοροι κόμβοι οι οποίοι ανήκουν σε αυτό το γκρουπ, δεν υπάρχει μέχρι να φορτώσει το γκρουπ.

Το NodeData διαθέτει τα δεδομένα μετρήσεων και τις εντολές ποτίσματος όταν αποφασίσουμε να τις φορτώσουμε από τον διακομιστή.

Το SearchOptions διαθέτει ρυθμίσεις εύρεσης οι οποίες καθορίζουν πόσο πίσω θέλουμε να φορτώσουμε δεδομένα στο NodeData.

Τέλος διαθέτουμε και δυο εντολές, η μια εντολή εκτελείτε για να φορτώσει τους κόμβους και η άλλη για να φορτώσει τις μετρήσεις.

```
[Reactive]
public GroupModel Group { get; set; }

[Reactive]
5 references
public NodeModel[]? Nodes { get; set; }

[Reactive]
3 references
public (IrrigationModel[] irrigations, MeasurementModel[] measurements)? NodeData { get; set; }

[Reactive]
6 references
public SearchOptions SearchOptions { get; set; }

2 references
public ReactiveCommand<Unit, Unit> LoadNodes { get; }

2 references
public ReactiveCommand<Unit, Unit> LoadNodeData { get; }
```

Εικόνα 174: UI.Web προτζεκτ, GroupViewModel.cs 1

Παρακάτω βλέπουμε τον κώδικα που εκτελείτε όταν δημιουργείτε ένα αντικείμενο ο GroupViewModel, είναι αρκετά συνηθισμένο όλη η λογική της εφαρμογής μας να δημιουργείτε εδώ, διότι δεν εκτελείτε μόνο όταν προκύπτει ένα συμβάν.

Πριν δημιουργήσουμε την λογική μας προ τοποθετούμε μερικές μεταβλητές. Την μεταβλητή Loading η οποία διαθέτουν όλα τα ViewModel, την κάνουμε true, την μεταβλητή Group την κάνουμε null και την μεταβλητή SearchOptions την βάζουμε στις τρεις ημέρες.

Δημιουργούμε την εντολή LoadNodes, αυτή όταν εκτελείτε καλεί την εντολή Search του INodeService και θέτει το γκρουπ στο Id του γκρουπ μας. Όταν τελειώσει η εκτέλεση αν είχαμε αποτέλεσμα θέτει την μεταβλητή Nodes από αυτό αλλιώς καταγραφεί το σφάλμα.

Δημιουργούμε τη εντολή LoadNodeData, αυτή θα εκτελείτε μόνο όταν η τιμή των κόμβων δεν είναι null και έχει ένα η παραπάνω αντικείμενα. Στην εκτέλεση της δημιουργεί μερικές προσωρινές μεταβλητές όπως, ενδιάμεσα σε ποιες ημερομηνίες θα κάνει αναζήτηση, προσωρινό αντικείμενο για την αποθήκευση των αποτελεσμάτων και όλα τα id των κόμβων.

Στην συνέχεια δημιουργεί δυο Task τα οποία ασύγχρονα θα ανακτήσουν τις τιμές προς αναζήτηση. Εμεί περιμένουμε μέχρι να τελειώσουν και οι δυο με τη εντολή "Task.WhenAll", όταν τελειώσουν θέτουμε τα αποτελέσματα στην μεταβλητή data που είχαμε δημιουργήσει και την ίδια την θέτουμε στην μεταβλητή NodeData.

```
1 reference
public GroupViewModel(string groupId, IGroupService groupService, INodeService nodeService)
{
    Loading = true;
    Group = null!;
    SearchOptions = SearchOptions.ThreeDays;

    LoadNodes = ReactiveCommand.CreateFromTask(async () =>
        await nodeService.Search(new() { GroupId = Group.Id }).SwitchAsync(
            s => Nodes = s,
            e => Console.WriteLine(e)));

    LoadNodeData = ReactiveCommand.CreateFromTask(
        canExecute: this.WhenAnyValue(x => x.Nodes, n => n is { Length: > 0 }),
        execute: async () =>
        {
            var data = (irrigations: Array.Empty<IrrigationModel>(), measurements: Array.Empty<MeasurementModel>());
            var startDate = DateTimeOffset.UtcNow.AddDays(1);
            var endDate = startDate.AddDays(-1 * SearchOptions - 1);
            var ids = Nodes!.Select(x => x.Id).ToHashSet();

            Console.WriteLine(SearchOptions.Days);

            var irrigations = nodeService.GetIrrigations(new SearchManyIrrigations(ids, startDate, endDate));
            var measurements = nodeService.GetMeasurements(new SearchManyMeasurements(ids, startDate, endDate));

            await Task.WhenAll(irrigations, measurements);

            irrigations.Result.Switch(s => data.irrigations = s, e => Console.WriteLine(e));
            measurements.Result.Switch(s => data.measurements = s, e => Console.WriteLine(e));

            NodeData = data;
        });
```

Εικόνα 175: UI.Web προτζεκτ, GroupViewModel.cs 2

Με την εντολή "this.WhenActivated" δηλώνουμε κώδικα ο οποίος εκτελείτε όταν ένα αντικείμενο "ενεργοποιείτε" από το view του, αυτό σημαίνει ότι το αντικείμενο μας έχει εμφανιστεί στον χρήστη. Οπότε όταν εκτελείτε αυτή η εντολή εμείς φορτώνουμε το γκρουπ το οποίο πρέπει να δείξουμε. Όταν η εντολή εκτελεστεί επιτυχώς θέτουμε το γκρουπ στην μεταβλητή Group αλλιώς καταγράφουμε το σφάλμα. Τέλος εκτελείτε ο κώδικας "Subscribe" ο οποίος αν δει ότι το Group είναι ακόμα null τότε θέτει την μεταβλητή NotFound σε true, ενώ θέτει το Loading σε false, αν έχουμε βρει τελικά το γκρουπ μας εκτελούμε και την εντολή FindNodes.

```
this.WhenActivated((CompositeDisposable disposables) =>
{
    Observable
        .FromAsync(async () => await groupService.Get(groupId).SwitchAsync(
            s => groupService.Watch(s.Id).BindTo(this, vm => vm.Group),
            e => Console.WriteLine(e)))
        .Subscribe(_ =>
        {
            NotFound = Group is null;
            Loading = false;

            if (NotFound is false)
                LoadNodes.Execute().Subscribe();
        });
});
```

Εικόνα 176: UI.Web προτζεκτ, GroupViewModel.cs 3

Το αρχείο GroupPage.razor περιέχει το View ενός γκρουπ και συνδέεται με το GroupViewModel. Στην παρακάτω εικόνα βλέπουμε ότι αν η μεταβλητή Loading είναι true τότε θέτουμε τον τίτλο της σελίδας στο "LetItGrow Group - Id" και δείχνουμε το αντικείμενο φορτώσεις.

```
@if (VM.Loading)
{
    <Title Value="@($"LetItGrow Group - {Id}")"></Title>
    <div class="w-full h-full flex items-center justify-center">
        <Loading />
    </div>
}
```

Εικόνα 177: UI.Web προτζεκτ, GroupPage.razor 1

Παρακάτω βλέπουμε τι θα δει ο χρήστης όταν η μεταβλητή Loading είναι false. Στην αρχή θέτουμε τον τίτλο τις σελίδας σε ""LetItGrow Group - Name", ενώ μετά δημιουργούμε την μπάρα με τις πληροφορίες και τα κουμπιά για τις διάφορε ενέργειες που μπορούμε να εκτελέσουμε σε ένα γκρουπ.

```
else
{
    <Title Value="@($"LetItGrow Group - {VM.Group.Name}")"></Title>

    <div class="pl-2 h-full overflow-y-auto overflow-x-hidden flex flex-col">
        <div class="flex justify-between">
            @{ const string btnCss = "text-gray-100 text-xs px-3 py-2 w-30"; }

            <h2 class="my-1">@VM.Group.Name (@VM.Group.Type)</h2>

            <div>
                <button class="btn-update @btnCss" @onclick="() => updateDialog.Show()">
                    <i class="bi bi-exclamation-circle"></i> Update
                </button>
                <button class="btn-cancel @btnCss" @onclick="() => deleteDialog.Show()">
                    <i class="bi bi-trash"></i> Delete
                </button>
            </div>
        </div>

        <hr class="m-2" />

        <p class="my-0">@(VM.Group.Description ?? "No Description")</p>

        <hr class="m-2" />

        <p class="text-xs font-mono mt-1 mb-0">Created by '@VM.Group.CreatedBy': '@VM.Group.CreatedAt.LocalDateTime.ToString("g")'</p>
        <p class="text-xs font-mono mt-0 mb-1">Updated by '@VM.Group.UpdatedBy': '@VM.Group.UpdatedAt.LocalDateTime.ToString("g")'</p>

        <hr class="m-2" />
```

Εικόνα 178: UI.Web προτζεκτ, GroupPage.razor 2

Παρακάτω βλέπουμε τους κόμβους που διαθέτει το γκρουπ μας, αν η μεταβλητή είναι null τότε δείχνουμε το αντικείμενο Loading αλλιώς δείχνουμε κάθε έναν κόμβο μέσα σε ένα στρογγυλό πλαίσιο. Προς το τέλος βλέπομε το κουμπί φόρτωσης δεδομένων των κόμβων του γκρουπ μαζί με ένα κουτί επίλογων για το πόσες μέρες πιο πριν θέλουμε να κάνουμε αναζήτηση.

```
        <div class="h-10 flex gap-2 justify-center overflow-x-auto" style="min-height: 40px">
        @if (VM.Nodes is null)
        {
            <Loading Class="h-10 flex justify-center" />
        }
        else
        {
            <Virtualize Context="node" Items="VM.Nodes">
                <a class="px-4 py-2 border bg-alt rounded-xl" href="/node/@node.Id">@node.Name</a>
            </Virtualize>
        }
        </div>

        <hr class="m-2" />

        <div class="p-2 flex items-center justify-center">
            <SpinnerButton Command="VM.LoadNodeData">
                Load Data
            </SpinnerButton>

            <select class="w-auto" @bind="VM.SearchOptions.Days">
                @foreach(var o in SearchOptions.All)
                {
                    <option value="@o.Days">@o</option>
                }
            </select>
        </div>
```

Εικόνα 179: UI.Web προτζεκτ, GroupPage.razor 3

Προς το τέλος της σελίδας αν τα δεδομένα NodeData δεν είναι null θα δείξουμε δυο γραφήματα, ένα με τις μετρήσεις και ένα με τις εντολές ποτίσματος. Στο τέλος τις σελίδας βρίσκονται οι "διάλογοι" που περιέχουν τις φόρμες ενημερώσεις και διαγραφής.

```
        @if (VM.NodeData is not null)
        {
            <h2 class="my-0 text-center">Measurements</h2>
            <PlotlyChart style="min-height: 500px; width: 100%" @ref="measureChart"
                         Config="new() { Responsive = true }"
                         Layout="new() { HoverMode = HoverModeEnum.Closest }" />

            <hr class="m-2" />

            <h2 class="my-0 text-center">Irrigations</h2>
            <PlotlyChart style="min-height: 500px; width: 100%" @ref="irrigateChart"
                         Config="new() { Responsive = true }"
                         Layout="new() { HoverMode = HoverModeEnum.Closest }" />
        }
    </div>

    <Dialog @ref="updateDialog" Class="w-11/12 lg:w-1/3">
        <UpdateGroupForm Group="VM.Group" OnCancelClick="() => updateDialog.Hide()" />
    </Dialog>

    <Dialog @ref="deleteDialog" Class="w-11/12 lg:w-1/3">
        <DeleteGroupForm Group="VM.Group" OnCancelClick="() => deleteDialog.Hide()" />
    </Dialog>
}
```

Εικόνα 180: UI.Web προτζεκτ, GroupPage.razor 4

Ο κώδικας της επόμενης εικόνας είναι αρκετά απλός, διαθέτει μερικά αντικείμενα της ιδίας σελίδας όπως τους "διάλογους" και τα διαγράμματα για να μπορεί να εκτελέσει τις εντολές τους και το GroupViewModel αυτής της σελίδας. Τέλος όταν οι παράμετροι της σελίδας έχουν τοποθετηθεί στις τιμές τους τότε δημιουργούμε το ViewModel μας, εκτελούμε την εντολή "Activate" και γράφουμε έναν κώδικα ο οποίος θα ενώσει τα NodeData με τα γραφήματα μας. Κάθε φορά που αλλάζει η μεταβλητή NodeData και δεν είναι null, μετατρέπουμε τις λίστες με τα δεδομένα σε σημεία γραφήματος και σε κάθε γράφημα τα τοποθετούμε και εκτελούμε την εντολή τους React η οποία θα ενημερώσει όλα τα σημεία του γραφήματος.

```
@code{
    private Dialog updateDialog = null!;
    private Dialog deleteDialog = null!;
    private PlotlyChart irrigateChart = null!;
    private PlotlyChart measureChart = null!;

    [Parameter]
    public string Id { get; set; } = string.Empty;

    public GroupViewModel VM { get => ViewModel!; set => ViewModel = value; }

    protected override void OnParametersSet()
    {
        base.OnParametersSet();
        VM = new(Id, groupService, nodeService);
        VM.Activator.Activate();

        VM.WhenAnyValue(x => x.NodeData)
            .WhereNotNull()
            .Subscribe(d =>
            {
                irrigateChart.Data = d!.Value.irrigations.ToTraceList();
                measureChart.Data = d!.Value.measurements.ToTraceList();

                Observable.FromAsync(measureChart.React).Subscribe(_ => StateHasChanged());
                Observable.FromAsync(irrigateChart.React).Subscribe(_ => StateHasChanged());
            });
    }
}
```

Εικόνα 181: UI.Web προτζεκτ, GroupPage.razor 5

7: IOT ΚΟΜΒΟΣ

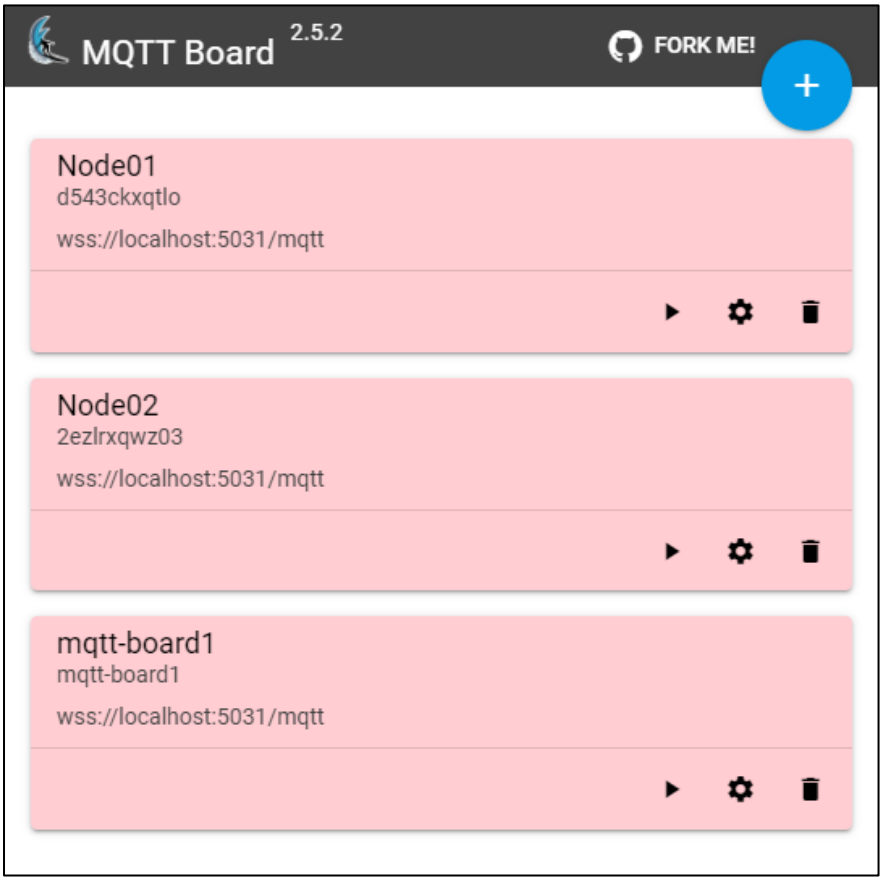
Η εφαρμογή IOT που θα τρέχει στους κόμβους μας χρησιμοποιώντας σαν βάση την βιβλιοθήκη Microservice θα έχει την δυνατότητα να λειτουργεί σε μικροϋπολογιστές όπως είναι το Raspberry PI με την βοήθεια του Mono Framework [8].

Η εφαρμογή έχει την δυνατότητα να χρησιμοποιηθεί σε οποιονδήποτε μικροϋπολογιστή Linux και δίνει δυνατότητα παραμετροποιήσεις των pins όλα αυτά θα τα δούμε στα επόμενα κεφάλαια.

Παρόλα αυτά όσο ο κόμβος καλεί τον διακομιστή MQTT σωστά, μπορούμε να αντικαταστήσουμε τελείως την εφαρμογή με κάποια νέα, μάλιστα κατά την διάρκεια ανάπτυξης του διακομιστή κάναμε ακριβώς αυτό όπως θα δούμε στο κεφάλαιο 7.1 Ιστοσελίδα MQTT Board.

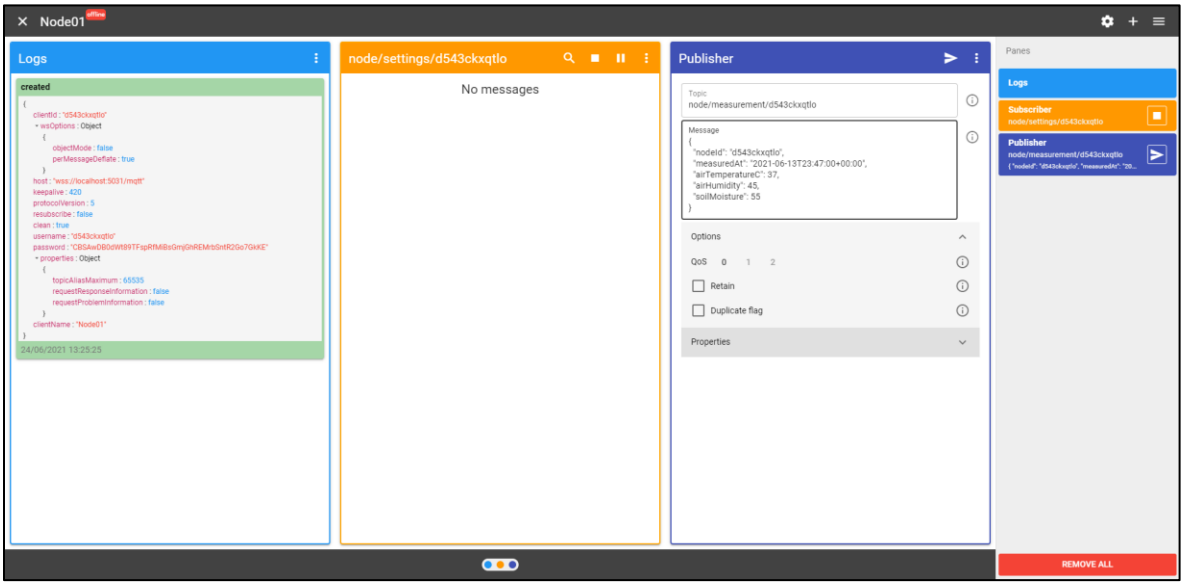
7.1 Ιστοσελίδα MQTT Board

Η ιστοσελίδα MQTT Board μας επιτρέπει να δημιουργούμε πελάτες MQTT, όπως φαίνεται στην παρακάτω εικόνα. Ο κάθε πελάτης έχει την δυνατότητα να ρυθμιστεί ξεχωριστά από τους άλλους ενώ μπορεί να δημοσιεύσει μηνύματα η και να κάνει εγγραφή για να λαμβάνει μηνύματα.



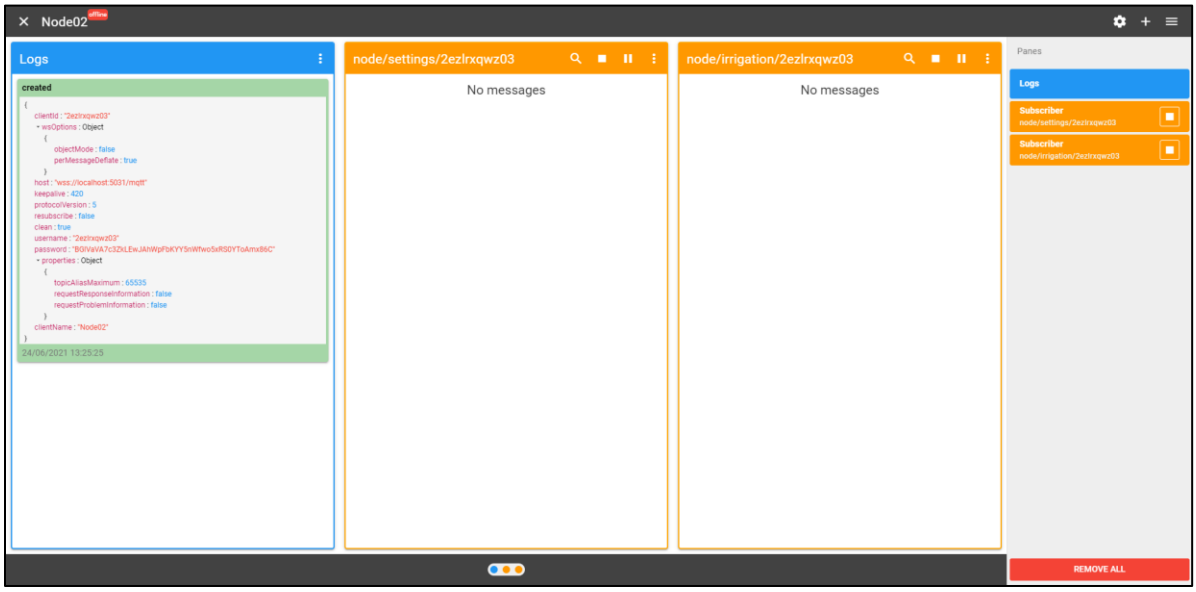
Εικόνα 182: MQTT Board Ιστοσελίδα

Παρακάτω βλέπουμε έναν κόμβο μετρήσεων. Αυτός ο κόμβος έχει ρυθμιστεί με στοιχεία από κόμβο που δημιουργήσαμε στην αρχική ιστοσελίδα του προτζεκτ. Χρησιμοποιώντας αυτά τα στοιχεία για να συνδεθεί στέλνει μετρήσεις μέσω του «Publisher» πάνελ και λαμβάνει ειδοποιήσεις για ρυθμίσεις μέσω του «node/settings/id» πάνελ.



Εικόνα 183: MQTT Board Ιστοσελίδα, Node01

Στην παρακάτω εικόνα βλέπουμε τον κόμβο Node02 ο οποίος λαμβάνει ειδοποιήσεις για ρυθμίσεις μέσο του «node/settings/id» πάνελ και ειδοποιήσεις στο «node/irrigation/id» πάνελ για την εκκίνηση η τον σταματημό του ποτιστικού μηχανισμού.



Εικόνα 184: MQTT Board Ιστοσελίδα, Node02

7.2: Βιβλιοθήκη NodeIot.Core

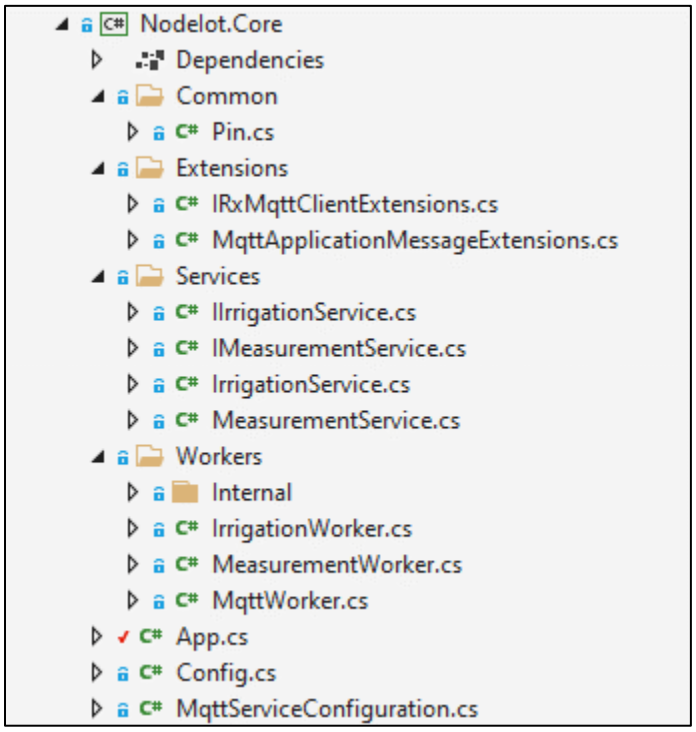
Παρακάτω βλέπουμε την δομή του προτζεκτ για τους IOT κόμβους, εδώ η δομή διαφέρει σε σχέση με όλα τα προηγούμαι προτζεκτ και είναι πιο συμβατική.

Διαθέτουμε το φάκελο Common οπού χρησιμοποιούμε κοινά class, τον φάκελο Extensions ο οποίος διαθέτει κλάσης με μεθόδους επέκτασης, τον φάκελο Services που διαθέτει κώδικα με υπηρεσίες σχετικές με τον κόμβο όπως για την μέτρηση και τον φάκελο Workers με κώδικα ο οποίος εκτελείτε στο υπόβαθρο όταν ξεκινά η εφαρμογή, μάλιστα ανάλογα τον τύπο της εφαρμογής θα εκτελεστεί και η ανάλογη υπηρεσία.

Στο φάκελο του προτζεκτ έχουμε τρία αρχεία-κλήσεις, το App οπού είναι η εφαρμογή μας την οποία εκτελεί το προτζεκτ Runner.

Το αρχείο Config το οποίο φορτώνει και περιέχει όλες τις ρυθμίσεις που μπορεί να ρύθμιση η εφαρμογή Runner η οποιαδήποτε εφαρμογή εκτελέσει το App.

MqttServiceConfiguration έχει τον κώδικα ο οποίος ρυθμίζει τον Mqtt Worker.



Εικόνα 185: NodeIot.Core προτζεκτ

Παρακάτω βλέπουμε τον κώδικα δημιουργίας της εφαρμογής. Η εφαρμογή όταν δημιουργείτε μπορεί να δεχθεί μεθόδους για περεταίρω τροποποίηση από τη πλατφόρμα εκτέλεσης. Στην αρχή δημιουργούμε το περιβάλλον της εφαρμογής “Host” στο οποίο ρυθμίζουμε την καταγραφή και την φορτώσει ρυθμίσεων από αρχεία yaml.

```
1 reference
public static App Create(string[] args,
    Action<HostBuilderContext, LoggerConfiguration>? configureLogging = null,
    Action<HostBuilderContext, IConfigurationBuilder>? configureAppConfig = null,
    Action<HostBuilderContext, IServiceCollection>? configureServices = null)
{
    var host = new HostBuilder()
        .UseSerilog((c, b) =>
        {
            b.MinimumLevel.Information();
            b.WriteTo.Console();
            b.WriteTo.File(path: $"{c.RootPath()}/logs/log-.txt", rollingInterval: RollingInterval.Day);
            configureLogging?.Invoke(c, b);
        })
        .ConfigureAppConfiguration((c, b) =>
        {
            b.AddYamlFile($"appsettings.yaml", optional: false, reloadOnChange: false);
            b.AddYamlFile($"{c.CurrentPath()}/appsettings.yaml", optional: true, reloadOnChange: false);
#if DEBUG
            b.AddYamlFile($"appsettings.dev.yaml", optional: true, reloadOnChange: false);
#else
            b.AddYamlFile($"appsettings.prod.yaml", optional: true, reloadOnChange: false);
#endif
            configureAppConfig?.Invoke(c, b);

            b.AddEnvironmentVariables();
            b.AddCommandLine(args);
        })
```

Εικόνα 186: Προτζεκτ NodeIot.Core, App.cs 1

Στην συνέχεια φορτώνουμε τις ρυθμίσεις στην κλάση Config και ρυθμίζουμε διάφορες υπηρεσίες που θα χρειαστεί η εφαρμογή μας όπως MQTT και κώδικα εκτέλεσης εντολών. Επίσης αν η εφαρμογή μας είναι σε ρυθμίσει “Test” αλλάζουμε τις υπηρεσίες εντολών και μετρήσεων.

```
        .ConfigureServices((c, services) =>
        {
            // Make configuration available to the app through static class.
            Config.Configuration = c.Configuration;

            services.AddSingleton<GpioController>();
            services.AddMqtt();

            switch (Config.Type)
            {
                case "irrigation":
                    if (Config.Test)
                        services.AddSingleton<IIrrigationService, TestIrrigationService>();
                    else
                        services.AddSingleton<IIrrigationService, GpioIrrigationService>();
                    services.AddHostedService<IrrigationWorker>();
                    break;

                case "measurement":
                    if (Config.Test)
                        services.AddSingleton<IMeasurementService, TestMeasurementService>();
                    else
                        services.AddSingleton<IMeasurementService, GpioMeasurementService>();
                    services.AddHostedService<MeasurementWorker>();
                    break;
            }

            configureServices?.Invoke(c, services);
        })
        .Build();

    return new App(host);
}
```

Εικόνα 187: Προτζεκτ NodeIot.Core, App.cs 2

Τέλος η εντολή RunAsync θα εκτελεστεί από οποιαδήποτε πλατφόρμα και θα ξεκινήσει την εφαρμογή μας. Αν η εφαρμογή δεν είναι σε “Test” τότε θα ενεργοποιήσουμε το “Power” Led, στην συνέχεια εκτελέσουμε τον “Host” και η εφαρμογή μας θα έχει ξεκινήσει. Αν κάποιο σφάλμα εμφανιστεί τότε θα το καταγράψουμε πριν κλείσουμε την εφαρμογή.

```
1 reference
112     public async Task<int> RunAsync(CancellationToken token = default)
113     {
114         if (!Config.Test)
115         {
116             var power = new Pin(Config.Led.Power, host.GetRequiredService<GpioController>());
117             power.Open(PinMode.Output);
118             power.Value = PinValue.High;
119         }
120         try
121         {
122             Log.Information("Node '{type}' with Id '{id}' is starting.", Config.Type, Config.ClientId);
123             await host.RunAsync(token);
124             return 0;
125         }
126         catch (Exception ex)
127         {
128             Log.Fatal(ex, "Node '{type}' with Id '{id}' terminated unexpectedly.", Config.Type, Config.ClientId);
129             return 1;
130         }
131         finally
132         {
133             Log.CloseAndFlush();
134         }
135     }
136 }
137 }
```

Εικόνα 188: Προτζεκτ NodeIot.Core, App.cs 3

Παρακάτω βλέπουμε των κώδικα του MqttWorker, ο οποίος χρησιμοποιώντας την βιβλιοθήκη «MQTTnet.Extensions.External.RxMQTT.Client» κάνει χρήση Reactive κώδικα. Παρακάτω βλέπουμε την αρχή σύνδεσης με τον διακομιστή MQTT και την αποσύνδεση του.

```
0 references
29      public async Task StartAsync(CancellationToken cancellationToken)
30      {
31          _client.Connected
32              .ObserveOn(ThreadPoolScheduler.Instance)
33              .Distinct()
34              .Throttle(TimeSpan.FromMilliseconds(500))
35              .Subscribe(state =>
36              {
37                  Log.Information("'{Id}' connection state '{state}'", Id, state);
38              });
39          await _client.StartAsync(_options);
40      }
41
            0 references
42      public async Task StopAsync(CancellationToken cancellationToken)
43      {
44          await _client.StopAsync();
45      }
46  }
47 }
```

Εικόνα 189: Προτζεκτ NodeIot.Core, MqttWorker.cs

Παρακάτω βλέπουμε τον κώδικα μετρήσεων. Ο κώδικας μετρήσεων τρέχει στο υπόβαθρο σαν υπηρεσία, λαμβάνει μια σύνδεση MQTT από την οποία παρακολουθεί για αλλαγές στις ρυθμίσεις του συγκεκριμένου κόμβου, όταν έρχονται αλλαγές τότε τις μετατρέπει σε κλάση και τις θέτει στις ρυθμίσεις της οποίες κρατά τοπικά αποθηκευμένες μόνο αν αυτές είναι σωστές.

```
2 references
public override Task Initialize(CancellationToken stop)
{
    _client
    .Connect($"{Config.TopicPrefix}/settings/{Config.ClientId}")
    .ObserveOn(ThreadPoolScheduler.Instance)
    .Select(args => args.ApplicationMessage.GetPayload<MeasurementSettings>())
    .SubscribeOn(ThreadPoolScheduler.Instance)
    .Subscribe(
    token: stop,
    onNext: settings =>
    {
        if (settings is null ||
            settings == Settings) return;

        if (settings is
            {
                PollInterval: > 3600 or < 60
            })
        {
            Log.Warning("Invalid settings `{settings}` received", JsonSerializer.Serialize(settings));
            return;
        }

        Settings = settings;
    });

    return Task.CompletedTask;
}
```

Εικόνα 190: Προτζεκτ NodeIot.Core, MeasurementWorker.cs 1

Κάθε φορά που εκτελείτε ο κώδικα WorkAsync εμείς υπολογίζουμε την ώρα στην οποία θα πρέπει να κάνουμε μια μέτρηση, και εκτελούμε ένα Delay μέχρι να έρθει αυτή η στιγμή, στο οποίο δίνουμε και ένα “CancellationToken” το οποίο σημάνει τον τερματισμό της εφαρμογής. Αν η εφαρμογή δεν τερματίστηκε εκτελούμε τον κώδικα Measure.

```
2 references
protected override async Task WorkAsync(CancellationToken stop)
{
    DateTimeOffset start = DateTimeOffset.Now;
    DateTimeOffset finish = start.AddSeconds(Settings.PollInterval);

    await Task.Delay(finish.Subtract(start), stop).Ignore().ConfigureAwait(false);

    if (stop.IsCancellationRequested) return;

    Measure();
}
```

Εικόνα 191: Προτζεκτ NodeIot.Core, MeasurementWorker.cs 2

Ο κώδικα Measure εκτελεί την διαδικασία μιας μέτρησης, μέσω της διεπαφή IMeasurementService λαμβάνει τις τρεις τιμές που πρέπει να στείλουμε στον διακομιστή. Στην συνέχεια τις κάνει Publish στο σημείο μετρήσεων με το δικό του ID.

```
72      protected void Measure()
73      {
74          var (tempC, humidity, soilMoisture) = _measurements.Get();
75          var topic = $"{Config.TopicPrefix}/measurement/{Config.ClientId}";
76          _client.Publish(topic, new CreateMeasurement
77          {
78              NodeId = Config.ClientId,
79              MeasuredAt = DateTimeOffset.UtcNow,
80              AirTemperatureC = tempC,
81              AirHumidity = humidity,
82              SoilMoisture = soilMoisture
83          });
84      }
```

Εικόνα 192: Προτζεκτ NodeIot.Core, MeasurementWorker.cs 3

Παρακάτω βλέπουμε τον κώδικα εκτέλεσης εντολών ποτίσματος. Στην αρχή όπως και στον κώδικα μετρήσεων παρακολουθούμε για αλλαγές στις ρυθμίσεις του κόμβου ενώ και εδώ ο κώδικα εκτελείτε στο υπόβαθρο σαν υπηρεσία.

```
36      public override Task Initialize(CancellationToken stop)
37      {
38          _client
39          .Connect($"{Config.TopicPrefix}/settings/{Config.ClientId}")
40          .ObserveOn(ThreadPoolScheduler.Instance)
41          .Select(args => args.ApplicationMessage.GetPayload<IrrigationSettings>())
42          .SubscribeOn(ThreadPoolScheduler.Instance)
43          .Subscribe(
44          token: stop,
45          onNext: settings =>
46          {
47              if (settings is null ||
48                  settings == Settings) return;
49
50              if (settings is
51                  {
52                      PollInterval: > 3600 or < 60
53                  })
54                  {
55                      Log.Warning("Invalid settings `{settings}` received", JsonSerializer.Serialize(settings));
56                      return;
57                  }
58
59              Settings = settings;
60          });
```

Εικόνα 193: Προτζεκτ NodeIot.Core, IrrigationWorker.cs 1

Παρακάτω βλέπουμε ότι πριν ξεκινήσει η εφαρμογή θα παρακολουθούμε και για τις εντολές εκτέλεσης ποτίσματος. Έτσι όταν έρχεται μια εντολή μέσο της διεπαφή IIrrigationService θέτουμε σε “true” η “false” την εκκίνηση του ποτίσματος.

```
62              _client
63                  .Connect($"{Config.TopicPrefix}/irrigations/{Config.ClientId}")
64                  .ObserveOn(ThreadPoolScheduler.Instance)
65                  .Select(args => args.ApplicationMessage.GetPayload<CreateIrrigation>())
66                  .SubscribeOn(ThreadPoolScheduler.Instance)
67                  .Subscribe(
68                  token: stop,
69                  onNext: irrigation =>
70                  {
71                      if (irrigation is { Type: IrrigationType.Start })
72                      {
73                          _irrigations.Set(true);
74                          return;
75                      }
76
77                      _irrigations.Set(false);
78                  });
79
80              return Task.CompletedTask;
81          }
82
```

Εικόνα 194: Προτζεκτ NodeIot.Core, IrrigationWorker.cs 2

Ακόμη μια διαφορά που παρατηρείται παραπάνω σε σχέση με της μετρήσεις είναι ότι εδώ δεν έχουμε ανάγκη για την μέθοδο WorkAsync και αυτό διότι ο κώδικα μας θα εκτελείτε ασύγχρονα μόνο όταν λαμβάνει νέες εντολές.

7.3: Βιβλιοθήκη NodeIot.Devices

Η βιβλιοθήκη NodeIot.Devices επαναχρησιμοποιεί κώδικα από το προτζεκτ .NetCore IOT, το δημιουργήσαμε για να κάνουμε Compile τον συγκεκριμένο κώδικα σε .Net Standard διότι το προτζεκτ ανοικτού κώδικα δεν ο υποστηρίζει.

Ο φάκελος Dhtxx περιέχει κώδικα σχετικό με αισθητήρες Dht.

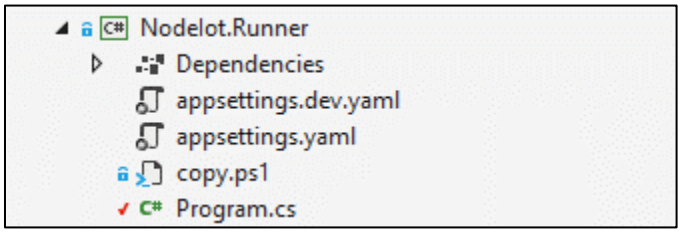
Ο φάκελος Mcp3xxx περιέχει κώδικα σχετικό με ολοκληρωμένα μετατροπής αναλογικού σήματος σε ψηφιακό.

Το αρχείο DelayHelper χρησιμοποιείται από τους παραπάνω φακέλους.

Εικόνα 195: NodeIot.Devices προτζεκτ

7.4: Προτζεκτ NodeIot.Runner

Το προτζεκτ NodeIot.Runner υπάρχει μόνο για να εκτελεί τον κώδικα σε οποία πλατφόρμα μας ενδιαφέρει. Περιέχει τις ρυθμίσεις που χεριάζετε ο κόμβος και ένα αρχείο script για την αντιγραφή του σε έναν κόμβο.



Εικόνα 196: Nodelot.Runner προτζεκτ

Το αρχείο Program.cs περιέχει τον κώδικα που εκτελείτε στον κόμβο ο οποίος απλά δημιουργεί ένα νέο App και το εκτελεί.

```
using LetItGrow.NodeIot;

await App.Create(args).RunAsync();
```

Εικόνα 197: Nodelot.Runner προτζεκτ, Program.cs

Παρακάτω βλέπουμε τις ρυθμίσεις που μπορεί να δεχθεί ένας κόμβος, κάποιες είναι προαιρετικές και προ τοποθετημένες και κάποιες είναι αναγκαστικές. Αναγκαστικές είναι ο τύπος ενός κόμβου, το Id του και ο κωδικός του. Επίσης χεριάζετε το URL του διακομιστή Mqtt. Το Certificate υπάρχει για μελλοντική χρήση και δεν είναι αναγκαστικό.

Προαιρετικές είναι οι ρυθμίσεις για τα Led τα οποία είναι το power που απεικονίζει ότι η εφαρμογή τρέχει και το Connection που αποκομίζει ότι έχει χαθεί η σύνδεση με τον διακομιστή.

Dht ρυθμίζει το Pin του αισθητήρα και Led το Pin στο οποίο θα δείχνει ότι λειτουργεί, version είναι η έκδοση του αισθητήρα μας.

Soil ρυθμίζει σε πια Pin θα διαβάσει τα δεδομένα του αισθητήρα υγρασίας χώματος μέσο του μετατροπέα και πιο Led θα ανάβει όταν συμβαίνει αυτή η διαδικασία.

Pump ρυθμίζει σε πιο Pin ενεργοποιείτε το ρελέ και σε πιο Pin ανάβει το λαμπάκι λειτουργίας.

Οι παραπάνω ρυθμίσεις είναι βασισμένες σε Rasperry Pi Zero W, και μπορούν να ρυθμιστούν ανάλογα αν έχουμε αλλάξει κόμβο, η βιβλιοθήκη μας κάνει χρήση της "libgpiod " μέσω της " .NET Core IoT Libraries " και μπορεί να ρυθμιστεί ανάλογα για κάθε linux μικροϋπολογιστή.

```yaml
# Node Mandatory Settings.

Type: <The type of the node. "Measurement" or "Irrigation">
ClientId: <The id of the node>
Token: <Token that will be send to the server>
Server: <Url to the server that this node should connect>
Certificate: <Path to certificate>

# Most connections include an `Led` paramter.
# This is the led to indicate the device is being used

# --General Connections
Led : { Power: 05, Connection: 06 } # Working Led

# --Measurement Connections
Dht : { Pin: 10, Led: 13, Version: 11 } # Dht sensor (11, 12, 21, 22)
Soil: { Pin: 10, Led: 19 } # Soil sensor

# --Irrigation Connections
Pump: { Pin: 10, Led: 26 } # Pump actuator

```

Εικόνα 198: NodeIot.Runner προτζεκτ, appsettings.yaml

Παρακάτω βλέπουμε το προαιρετικό script το οποίο είναι γραμμένο σε PowerShell. Το script αυτό πηγαίνει στον φάκελο οπού δημιουργείτε η εφαρμογή μας και λέγετε artifacts. από εκεί διαγραφεί τον φάκελο logs σε περίπτωση που υπάρχει και καταγράφει την διαγραφή, στη συνέχεια διαγράφει τον κώδικα από τον κόμβο μας και το καταγράφει. Τέλος κάνει αντιγραφή το artifacts στον κόμβο μας και καταγράφει το τέλος της εκτέλεσης του.

```powershell
[CmdletBinding()]
param (
    [Parameter(Mandatory)]
    [string] $node
)

$artifacts = "../../artifacts/bin/Node.Runner/Debug/net48"

Remove-Item $artifacts/logs/ -r *>$null
Write-Host "Removed local logs"

ssh $node "rm node -r" *>$null
Write-Host "Removed remote files"

Write-Host "Copying files to remote"
scp -r $artifacts "$node:node" *>$null

Write-Host "Finished"
```

Εικόνα 199: NodeIot.Runner προτζεκτ, copy.ps1

8: Εκτέλεση

Για την εκτέλεση όλων τον διακομιστών και της ρυθμίσεις τους πέρα από το κάθε αρχείο appsettings.yaml χρησιμοποιήσαμε και την εφαρμογή tye.

Η εφαρμογή αυτή μας επιτρέπει να διαχειριστούμε όλους τους διακομιστές ταυτόχρονα και να τους προσθέσουμε περεταίρω ρυθμίσεις.

Παρακάτω βλέπουμε το αρχείο tye.yaml το οποίο είναι οι ρυθμίσεις του εργαλείου αυτού. Με αυτό το άρχει οδηγούμε το εργαλείο tye έτσι ώστε να ξεκινήσει τον κάθε διακομιστή. Το αρχείο αυτό είναι στον φάκελο src, δηλαδή ένα επίπεδο πιο πάνω από κάθε φάκελο προτζεκτ.

Για να τρέξουμε όλες τις εφαρμογές μας λοιπόν αφού έχουμε εγκαταστήσει το dotnet sdk και την εφαρμογή tye με την εντολή "tye run".

Για την εκτέλεση κάνουμε περιήγηση με την εφαρμογή κονσόλας και εκτελούμε την εντολή "tye run".

Εναλλακτικά αφού έχουμε κάνει εγκατάσταση τα ίδια πράγματα εκτελούμε το αρχείο run.ps1 το οποίο θα εκτελέσει τη παραπάνω εντολή.

```yaml
name: LetItGrow
services:
- name: blazor
  project: UI.Web/UI.Web.csproj
  tags: [ui]
  replicas: 1
  bindings:
  - port: 5001
    protocol: https

- name: api
  project: Microservice.RestApi/Microservice.RestApi.csproj
  tags: [api, dev]
  replicas: 1
  bindings:
  - port: 5011
    protocol: https

- name: identity
  project: Identity.Api/Identity.Api.csproj
  tags: [id, dev]
  replicas: 1
  bindings:
  - port: 5021
    protocol: https

- name: mqtt
  project: Microservice.Mqtt/Microservice.Mqtt.csproj
  tags: [mqtt, dev]
  replicas: 1
  bindings:
  - port: 5031
    protocol: https

- name: worker
  project: Microservice.Worker/Microservice.Worker.csproj
  tags: [worker, dev]
  replicas: 1
```

Εικόνα 200: LetItGrow, tye.yaml

Βιβλιογραφικές Αναφορές

[1] «RestAPI,» [Ηλεκτρονικό]. Available: https://www.redhat.com/en/topics/api/what-is-a-rest-api.

[2] «Single Page Application,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Single-page_application.

[3] «MQTT,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/MQTT.

[4] «Worker Services,» [Ηλεκτρονικό]. Available: https://www.stevejgordon.co.uk/what-are-dotnet-worker-services.

[5] «SignalR,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/SignalR.

[6] «RPC,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Remote_procedure_call.

[7] «API,» Wikipedia, [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Application_programming_interface.

[8] «Mono,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Mono_%28software%29.

[9] «MESH Network,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Mesh_networking.

[10] «LoRa Network,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/LoRa#LoRaWAN.

[11] «Monolithic,» [Ηλεκτρονικό]. Available: https://docs.microsoft.com/en-us/dotnet/architecture/containerized-lifecycle/design-develop-containerized-apps/monolithic-applications.

[12] «Microservices,» [Ηλεκτρονικό]. Available: https://aws.amazon.com/microservices/.

[13] «CSharp,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/C_Sharp_%28programming_language%29.

[14] «.NET Core,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/.NET_Core.

[15] «ASP.NET Core,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/ASP.NET_Core.

[16] «MQTTNet,» [Ηλεκτρονικό]. Available: https://github.com/chkr1011/MQTTnet.

[17] «Blazor,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Blazor.

[18] «Openiddict,» [Ηλεκτρονικό]. Available: https://github.com/openiddict/openiddict-core.

[19] «Open Source,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Open_source.

[20] «Vertical Scaling,» [Ηλεκτρονικό]. Available: https://www.techopedia.com/definition/9912/vertical-scaling.

[21] «Horizontal Scaling,» [Ηλεκτρονικό]. Available: https://www.techopedia.com/definition/7594/horizontal-scaling.

[22] «HTTP,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol.

[23] «WebSockets,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/WebSocket.

[24] «TCP IP,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Internet_protocol_suite.

[25] «JSON,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/JSON.

[26] «HTML,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/HTML5.

[27] «Long Polling vs WebSockets vs EventSource,» [Ηλεκτρονικό]. Available: https://medium.com/system-design-blog/long-polling-vs-websockets-vs-server-sent-events-c43ba96df7c1.

[28] «OAuth,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/OAuth.

[29] «OpenID,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/OpenID.

[30] «Dependency Injection,» [Ηλεκτρονικό]. Available: https://docs.microsoft.com/en-us/dotnet/core/extensions/dependency-injection.

[31] «CQRS,» [Ηλεκτρονικό]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs.

[32] «MediatR,» [Ηλεκτρονικό]. Available: https://github.com/jbogard/MediatR/.

[33] «CSharp Interface,» [Ηλεκτρονικό]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/interface.

[34] «CSharp Generics,» [Ηλεκτρονικό]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/generics.

[35] «FluentValidation,» [Ηλεκτρονικό]. Available: https://fluentvalidation.net/.

[36] «LINQ,» [Ηλεκτρονικό]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/.

[37] «Vertical Slice Architecture,» [Ηλεκτρονικό]. Available: https://jimmybogard.com/vertical-slice-architecture/.

[38] «Namespaces,» [Ηλεκτρονικό]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/namespaces/using-namespaces.

[39] «YAML,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/YAML.

[40] «RandomNUmberGenerator GetInt32,» [Ηλεκτρονικό]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.randomnumbergenerator.getint32?view=net-5.0.

[41] «CouchDB,» [Ηλεκτρονικό]. Available: https://couchdb.apache.org/.

[42] «Swagger,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Swagger_(software).

[43] «TailwindCSS,» [Ηλεκτρονικό]. Available: https://tailwindcss.com/.

[44] «WaterCss,» [Ηλεκτρονικό]. Available: https://watercss.kognise.dev/.

[45] «ReactiveUI,» [Ηλεκτρονικό]. Available: https://www.reactiveui.net/.

[46] «MVVM,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel.

[47] «SQL,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/SQL.

[48] «NoSQL,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/NoSQL.

[49] «Relational Databases,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Relational_database.

[50] «CouchDB.NET,» [Ηλεκτρονικό]. Available: https://github.com/matteobortolazzo/couchdb-net.

[51] «DynamicData,» [Ηλεκτρονικό]. Available: https://github.com/reactivemarbles/DynamicData/.