ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΛΟΠΟΝΝΗΣΟΥ
ΣΧΟΛΗ ΟΙΚΟΝΟΜΙΑΣ, ΔΙΟΙΚΗΣΗΣ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ
Π.Μ.Σ. ΣΤΗΝ ΕΠΙΣΤΗΜΗ ΚΑΙ ΤΕΧΝΟΛΟΓΙΑ ΥΠΟΛΟΓΙΣΤΩΝ

Μεταπτυχιακή Εργασία

Δημιουργία Μετεωρολογικού Σταθμού με Raspberry PI και χρήση τεχνολογιών C# , Entity Framework και Azure

Επιβλέπων καθηγητής: ***Γρηγόρης Δημητρουλάκος***

Φοιτητές: ***Σπαής Γεώργιος*** **(ΑΜ 237025388490)**

Δημόπουλος Θεόδωρος **(ΑΜ 2022201802006)**

ΤΡΙΠΟΛΗ 2020

Θέλουμε να επισημάνουμε ότι αυτά τα δύο χρόνια παρακολούθησης του μεταπτυχιακού προγράμματος ήταν πάρα πολύ δύσκολα, με δεδομένη την χιλιομετρική απόσταση και τον ελάχιστο χρόνο που είχαμε, λόγω επαγγελματικών και οικογενειακών υποχρεώσεων. Δοκιμάστηκαν οι δυνάμεις και οι αντοχές μας και ευτυχώς που είχαμε ο ένας τον άλλον.

Νιώθουμε την ανάγκη να ευχαριστήσουμε :

Το Πανεπιστήμιο της Τρίπολης, που μας δέχθηκε και μας έδωσε την ευκαιρία να παρακολουθήσουμε και μπορέσαμε να εμπλουτίσουμε τις γνώσεις μας.

Ιδιαίτερα, τον επιβλέποντα καθηγητή Γρηγόρη Δημητρουλάκο για την καθοδήγηση του, τις γνώσεις του, την ανθρωπιά του και την αυστηρότητα του όπου χρειάστηκε. Ένα σημαντικό μερίδιο του αποτελέσματος της προσπάθεια μας του ανήκει.

Τον συνάδελφο Σπύρο Πουλή, που μας παρότρυνε και ξεκινήσαμε αυτό το μεγάλο, αλλά και όμορφο ταξίδι.

Τους Διευθυντές των εργασιών μας, για τις διευκολύνσεις που μας έκαναν, προκειμένου να μπορέσουμε να παραβρισκόμαστε και να παρακολουθούμε τα μαθήματα μας στην Τρίπολη.

Τις συζύγους μας για την υπομονή και την κατανόηση που έδειξαν.

Περιεχόμενα

Ευρετήριο Εικόνων

ΚΕΦΑΛΑΙΟ 1 – ΕΚΤΕΤΑΜΕΝΗ ΠΕΡΙΛΗΨΗ - EXTENDED ABSTRACT

Εκτεταμένη Περίληψη

Η παρατήρηση των καιρικών φαινομένων άρχισε από τη εποχή της ακμής των Βαβυλωνίων και συνεχίστηκε σε όλη την εξέλιξη της ιστορίας. Αρχικά στα πρώτα μοντέλα παρατήρησης και πρόγνωσης του καιρού η μεθοδολογία που ακολουθούνταν απείχε σε μεγάλο βαθμό από την σημερινή, μια που χρησιμοποιούνταν τις περισσότερες φορές ανορθόδοξες και αντιεπιστημονικές τεχνικές.

Όλα αυτή η αντιμετώπιση άλλαξε την δεκαετία του 1950 όποτε και αρχίζει να διαμορφώνεται η επιστημονική παρατήρηση των καιρικών φαινομένων και η οργάνωση τους με την χρήση υπολογιστικών συστημάτων. Τα μαθηματικά μοντέλα που αναπτύχθηκαν απαιτούσαν πληθώρα επαναλαμβανόμενων πράξεων, καθώς και μετεωρολογικές μετρήσεις από όσο το δυνατό περισσότερα γεωγραφικά σημεία. Η συλλογή των δεδομένων και ο υπολογισμοί, δεν θα μπορούσαν να γίνουν έγκαιρα και με ακρίβεια εάν δεν χρησιμοποιούνταν τεχνολογίες Πληροφορικής και Επικοινωνιών. Είναι χαρακτηριστικό ότι η ακρίβεια στην πρόγνωση είναι πλέον σχεδόν απόλυτη, χάρις τα σύγχρονα και μεγάλης επεξεργαστικής ισχύς υπολογιστικά συστήματα, αλλά και με ένα ενοποιημένο με την χρήση του διαδικτύου και εξαιρετικά εκτεταμένο δικτύου μετεωρολογικών σταθμών.

Το ερέθισμα που οδήγησε στην δημιουργία του Meteo Project είναι η δημιουργία ενός σχετικά οικονομικού και απόλυτα παραμετροποιήσιμου μετεωρολογικού σταθμού, που θα συλλέγει σε μια κεντρική βάση δεδομένων μετρήσεις σε πραγματικό χρόνο. Για την υλοποίηση του έργου χρησιμοποιήθηκαν τεχνολογίες C#, MS Asure, Entity Framework, καθώς και του δημοφιλούς και αξιόπιστου μικροελεγκτή Raspberry Pi.

Στο σύγγραμμα αναλύεται η μεθοδολογία που ακολουθήθηκε για την υλοποίηση του συστήματος. Μέσα από τις σελίδες του ο αναγνώστης ακολουθώντας απλά βήματα μπορεί να υλοποιήσει το σύνολο της κατασκευής και των προγραμμάτων που την υποστηρίζουν. Το τελικό αποτέλεσμα είναι η δημιουργία μιας πλατφόρμας που συλλέγει μετεωρολογικά δεδομένα σε περιοδικό χρόνο που αποφασίζει ο χρήστης και τα οποία φιλοξενούνται σε ένα κεντρικό cloud ιστορικό αρχείο. Η μετεξέλιξη του έργου θα μπορούσε να είναι η χρήση των δεδομένων αυτών και η υλοποίηση ενός μοντέλου πρόβλεψης του καιρού. Αυτό θα μπορούσε να γίνει στην ήδη υπάρχουσα υποδομή του Meteo Project.

Extended Abstract

The observation of the weather began in the heyday of the Babylonians and continued throughout history. Initially, in the first models of weather observation and forecasting, the methodology followed was largely different from the current one, since most of the time unorthodox and unscientific techniques were used.

All this treatment changed in the 1950s when the scientific observation of weather phenomena and their organization began to take shape with the use of computer systems. The mathematical models developed required a plethora of repetitive actions, as well as meteorological measurements from as many geographical locations as possible. Data collection and calculations could not be done in a timely and accurate manner without the use of Information and Communication technologies. It is characteristic that the accuracy of the forecast is now almost perfect, thanks to the modern and highly processing computing systems, but also with a unified with the use of the internet and an extremely extensive network of meteorological stations.

The stimulus that led to the creation of the Meteo Project is the creation of a relatively economical and fully customizable meteorological station, which will collect real-time measurements at a central database. C #, MS Asure, Entity Framework technologies, as well as the popular and reliable Raspberry Pi microcontroller were used to implement the project.

The dissertation analyzes the methodology followed for the implementation of the system. Through its pages, the reader, following simple steps, can implement all the construction and the programs that support it. The end result is the creation of a platform that collects meteorological data in a periodical decided by the user and hosted on a central cloud historical file. The development of the project could be the use of this data and the implementation of a weather forecast model. This could be done in the existing infrastructure of the Meteo Project.

ΚΕΦΑΛΑΙΟ 2 - ΑΡΧΙΤΕΚΤΟΝΙΚΗ ΤΟΥ PROJECT

Το project αυτό είναι ένας μετεωρολογικός σταθμός με δυνατότητα αποστολής και ανάγνωσης των μετρήσεων από το cloud. Η cloud platform που χρησιμοποιήθηκε είναι το Azure της Microsoft. Η γλώσσα προγραμματισμού ανάπτυξης είναι η C# με χρήση .NET Core 2.1, .NET framework 4.8 και UWP. Το περιβάλλον ανάπτυξης είναι το Visual Studio 2019 Enterprise edition. Για τη δημιουργία database χρησιμοποιήθηκε το Entity Framework, και επιλέχτηκε η τεχνική Code First.

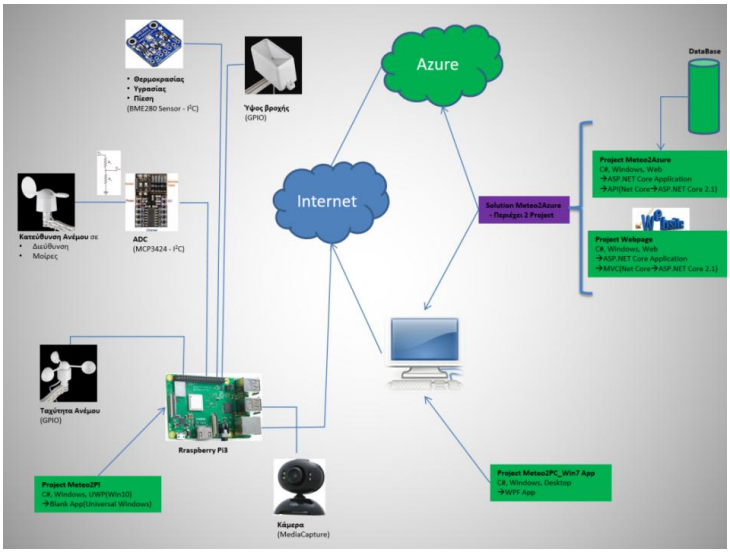
Επιπρόσθετα, χρησιμοποιήθηκαν και δύο βιβλιοθήκες τρίτων, όπως οι **BuildAzure.IoT.Adafruit.BME280**, για τον αισθητήρα θερμοκρασίας, υγρασίας, πίεσης και η **Newtonsoft.Json**, για την υποστήριξη JSON.

Εικόνα 1 MeteoProject Υλοποίηση

2.1 Αρχιτεκτονική του project

Για την κατασκευή του μετεωρολογικού σταθμού (Εικόνα 1), χρησιμοποιήθηκαν αισθητήρες θερμοκρασίας, υγρασίας, ατμοσφαιρικής πίεσης, ταχύτητας ανέμου, κατεύθυνσης ανέμου και ύψους βροχής. Επίσης χρησιμοποιήθηκε και μία web camera, έτσι ώστε να λαμβάνεται μία live εικόνα από το σημείου που βρίσκεται εγκατεστημένος ο μετεωρολογικός σταθμός. Όλοι αισθητήρες και η camera, συνδέονται με κατάλληλα πρόσθετα ηλεκτρονικά κυκλώματα και συνδέσεις, με έναν μικροελεγκτή Raspberry PI, ο οποίος συνδέεται στο internet και στέλνει τις μετρήσεις στον server που υπάρχει στο Azure. Στο Azure εκτελούνται δύο services και ο SQL Server που εξυπηρετεί τη βάση δεδομένων με τις μετρήσεις. Το ένα app service είναι ένα REST API service το οποίο λαμβάνει τα δεδομένα από το Raspberry και τα αποθηκεύει στη database. Το δεύτερο app service είναι ένα Web service το οποίο διαβάζει τις μετρήσεις από τη database και εμφανίζει σε μία ιστοσελίδα τις τρέχουσες τιμές και την live εικόνα. Επίσης εμφανίζει σε διαγράμματα τις μετρήσεις των τελευταίων έξι ωρών.

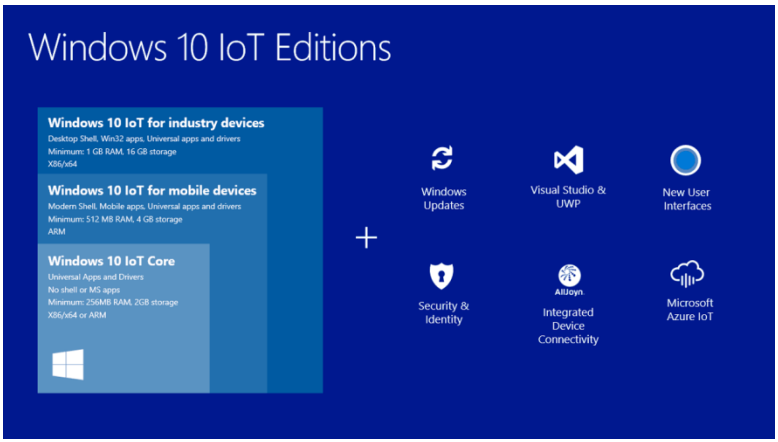
Επίσης για τις ανάγκες της ανάπτυξης και του debugging αναπτύχθηκε και μία desktop εφαρμογή (για Windows 7 και 10). Αυτή η εφαρμογή είχε τον ρόλο του API consumer. Έστελνε δηλαδή HTTP εντολές προς το service (είτε αυτό έτρεχε τοπικά, είτε στο Azure), εξομοιώνοντας έτσι την λειτουργία του Raspberry. Στο παρακάτω διάγραμμα (Εικόνα 2), φαίνεται σχηματικά η αρχιτεκτονική του project.



Εικόνα 2 Αρχιτεκτονική Project

2.2 Υποστηριζόμενο λογισμικό

Για το λειτουργικό σύστημα του μικροελεγκτή (Raspberry PI) επιλέχτηκαν τα Windows 10 IoT Core (Εικόνα 3). Είναι μία ειδική έκδοση των Windows 10 για μικροελεγκτές, η οποία παρέχει ασφαλή συνδεσιμότητα στο Internet, πρόσβαση σε Windows Updates και πλήρη λειτουργικότητα για να ανάπτυξη εφαρμογών UWP με το Visual Studio.



Εικόνα 3 Windows 10 IoT Editions

ΚΕΦΑΛΑΙΟ 3 -Hardware

3.1 Project part list

- Raspberry PI 3 Model B+
- DFROBOT I2C Environmental Sensor (Bosch BME280 combined temperature / humidity / pressure sensor)
- DFROBOT MCP3424 18-Bit ADC-4 Channel with Programmable Gain Amplifier
- SPARKFUN Weather Station kit (anemometer, wind vane, rain gauge)

3.2 Raspberry PI

Ο μικροελεγκτής που χρησιμοποιήθηκε για τις ανάγκες του project είναι ένα Raspberry PI 3 Model B+ (Εικόνα 4) με λειτουργικό Microsoft Windows 10 IoT. Θα είχε προτιμηθεί η παλαιότερη έκδοση PI 3 Model B λόγω καλύτερης συμβατότητας με τα Windows 10 IoT, όμως επειδή ήταν δυσεύρετο στην αγορά, καταλήξαμε στο Model B+ και έγινε μία προσπάθεια να ξεπεραστούν κάποια issues ασυμβατότητας, τα οποία θα αναλυθούν στη συνέχεια. Τελικά τα όποια issues λύθηκαν και το σύστημα λειτούργησε χωρίς προβλήματα.

Για την εξωτερική μνήμη του Raspberry επιλέχθηκε το micro SD **Sandisk Ultra microSDHC 16GB**, το οποίο είναι πιστοποιημένο ότι λειτουργεί σωστά με τα Windows 10 IOT.

Εικόνα 4 Raspberry Pi 3 Model B+

3.3 Temperature, Humidity Pressure sensor

Για τη μέτρηση θερμοκρασίας, υγρασίας και ατμοσφαιρικής πίεσης, οι επιλογές είναι πολλές. Υπάρχουν διάφοροι αισθητήρες στην αγορά, με διαφορετικές δυνατότητες, ευαισθησία, ακρίβεια, digital interfaces κλπ.

Μετά από έρευνα επιλέχτηκε ο αισθητήρας BME280 της Bosch, ο οποίος είναι 3-in-1. Είναι δηλαδή αισθητήρας θερμοκρασίας, υγρασίας και πίεσης σε ένα κύκλωμα, το οποίο ήταν και το πιο σημαντικό κριτήριο επιλογής. Δύο άλλα σημαντικά κριτήρια ήταν το ότι:

1) διαθέτει διπλό digital interface και μπορεί να δουλέψει και σε SPI και σε I^2C.
2) είναι συμβατός με 3.3V και 5V μικροελεκτές.

Τη στιγμή της έρευνας για την επιλογή αισθητήρα, δεν ήταν ξεκάθαρες οι ανάγκες που αφορούν τα δύο παραπάνω κριτήρια, λόγω της πιθανής ανάγκης επιπλέον συσκευών, επομένως το διπλό interface και το μεγάλο εύρος τάσης λειτουργίας, ήταν ιδανικές λύσεις καθώς προσέφεραν μεγάλη ευελιξία.

Σαν ολοκληρωμένη λύση επιλέξαμε το DFROBOT I2C Environmental Sensor (Εικόνα 5).

Εικόνα 5 DFROBOT I2C Environmental Sensor (BME280 Bosh)

Οι προδιαγραφές του αισθητήρα είναι:

- Temperature Measuring Range: -40℃~+85℃, resolution of 0.1℃, deviation of ±0.5℃
- Humidity Measuring Range: 0~100%RH, resolution of 0.1%RH, deviation of ±2%RH
- Pressure Measuring Range: 300~1100hPa

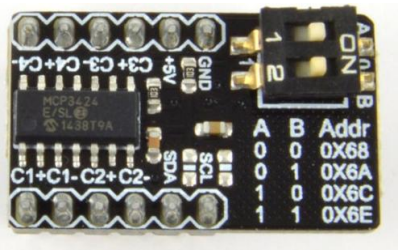
Στο τελικό setup προτιμήθηκε το I^2C digital interface, για ευκολία στη κατασκευή (το SPI είναι 4-wire ενώ το I^2C είναι 3-wire), για ομοιομορφία με άλλες συσκευές (να είναι όλες I^2C) και για ταχύτερη ανάπτυξη του κώδικα της εφαρμογής.

3.4 Analog to Digital Converter

Το σύστημα μέτρησης της κατεύθυνσης του αέρα (όπως θα αναλυθεί παρακάτω) είναι αναλογικό. Επομένως χρειαζόμαστε έναν Analog to Digital Converter (ADC), για να μετατρέψει την τάση εξόδου του συστήματος σε ψηφιακή τιμή, έτσι ώστε να γίνει αντιστοίχηση με την κατεύθυνση του ανέμου.

Όπως είναι γνωστό υπάρχουν αμέτρητοι ADCs στην αγορά, με πάρα πολλούς συνδυασμούς χαρακτηριστικών και προδιαγραφών. Για παράδειγμα, υπάρχουν ADCs με sample frequency από μερικά samples per second, μέχρι και αρκετά Giga samples per second. Επίσης, με χαμηλή ανάλυση δειγμάτων μέχρι και πάρα πολύ υψηλή, άρα και με πάρα πολύ υψηλή ακρίβεια τάσης μέτρησης. Το πλήθος των εισόδων και ταυτόχρονων μετατροπών είναι ένα βασικό χαρακτηριστικό επίσης, καθώς και ο τύπος εισόδου (single ended ή differential ή υποστήριξη και των δύο).

Μετά από έρευνα καταλήξαμε σε ένα 18bit Multi-Channel I^2C ADC, και συγκεκριμένα στο MCP3424 της εταιρίας Microchip (Εικόνα 6). Η επιλογή έγινε μετά από ανάλυση και αξιολόγηση όλων των παραπάνω κριτηρίων σε συνδυασμό με τις ανάγκες και τις απαιτήσεις του project.



Εικόνα 6 ADC MCP3424 της εταιρίας Microchip

Οι προδιαγραφές του ADC είναι:

- Operating Voltage: 2.7 - 5.5V
- Programmable Resolution: 12, 14, 16, 18bits
- On-Board Programmable Gain Amplifier (PGA): x1,x2,x4,x8
- Programmable Data Rate: 240, 60, 15, 3.75 SPS
- Input Interface: 4 differential channels
- Output Interface: I^2C

Όπως φαίνεται από τις παραπάνω προδιαγραφές, το digital interface του είναι I^2C, όπως άλλωστε και του αισθητήρα θερμοκρασίας/υγρασίας/πίεσης.

Σαν ολοκληρωμένο σύστημα επιλέχθηκε το DFROBOT MCP3424 18-Bit ADC-4 Channel with Programmable Gain Amplifier.

Ένα αρνητικό στοιχείο, για τις ανάγκες του project, στις προδιαγραφές του MCP3424 είναι το On-Board Voltage Reference το οποίο είναι 2.048V ± 0.05%, χωρίς δυνατότητα προγραμματισμού. Επομένως, η μέγιστη τάση single-ended εισόδου που μπορεί να μετρήσει το συγκεκριμένο ADC είναι 2.048 V. Όπως θα αναλυθεί αργότερα, αυτό τελικά αποδείχθηκε ότι είναι πρόβλημα, καθώς προσέθεσε μία παραπάνω παράμετρο πολυπλοκότητας στην υλοποίηση.

3.5 Weather Station

Επιλέχθηκε ο μετεωρολογικός σταθμός της SparkFun (Εικόνα 7). Το κιτ αποτελείται από έναν μετρητή ταχύτητας ανέμου, έναν μετρητή διεύθυνσης ανέμου, έναν μετρητή ύψους βροχής και όλα τα παρελκόμενα εξαρτήματα και καλώδια για τη συναρμολόγηση του σταθμού.

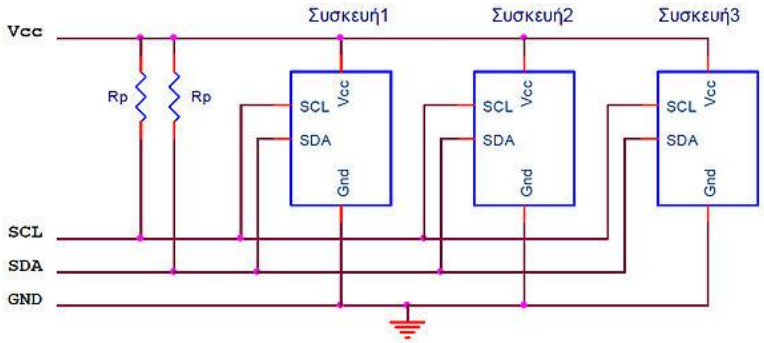
Ο κατασκευαστής του κιτ είναι η εταιρία Argent Data Systems.

Εικόνα 7 Μετεωρολογικός σταθμός της SparkFun.

3.6 Δίαυλος I²C

Ο δίαυλος I²C είναι ένας σειριακός δίαυλος που δημιουργήθηκε από τη Philips (τώρα NXP) και χρησιμοποιείται για την σύνδεση περιφερειακών μικρής ταχύτητας σε μητρικές πλακέτες (motherboards), ενσωματωμένα συστήματα (embedded systems), κινητά τηλέφωνα ή άλλες ηλεκτρονικές συσκευές. Ο δίαυλος I2C δεν χρησιμοποιείται μόνο για την επικοινωνία συσκευών που βρίσκονται πάνω σε ένα τυπωμένο κύκλωμα, αλλά και για την επικοινωνία συσκευών που συνδέονται με καλώδια.

Στη παρακάτω εικόνα (Εικόνα 8) φαίνεται ένα παράδειγμα διαύλου I2C. Όπως φαίνεται για τη μεταφορά των δεδομένων (0 ή 1) χρησιμοποιεί μόνο δύο καλώδια (τα οποία είναι ημιαμφίδρομης κατεύθυνσης): Τα SCL και SDA. Η γραμμή SCL είναι η γραμμή ρολογιού, ενώ η SDA είναι η γραμμή δεδομένων. Οι γραμμές αυτές συνδέονται σε όλες τις συσκευές, που υπάρχουν πάνω στο δίαυλο I²C. Προφανώς εκτός από τα παραπάνω καλώδια που μεταφέρουν δεδομένα, απαιτείται και ένα τρίτο καλώδιο το οποίο είναι η γείωση (GND) ή 0 V.



Εικόνα 8 Παράδειγμα διαύλου I2C

Επίσης μπορεί να υπάρχει (προαιρετικά) και ένα τέταρτο καλώδιο το οποίο είναι η γραμμή τροφοδοσίας (VCC ή VDD), με την οποία τροφοδοτούνται με ισχύ οι διάφορες συσκευές που συνδέονται στο δίαυλο. Τυπικές τάσεις που χρησιμοποιούνται στο δίαυλο είναι τα +5V ή +3,3V, αν και επιτρέπονται συστήματα με διαφορετικές τάσεις (συνήθως στην περιοχή από +1,2V έως +5,5V).

Ο μέγιστος αριθμός κόμβων (συσκευών), που μπορούν να συνδεθούν στον δίαυλο, περιορίζεται από τον αριθμό των διαθέσιμων διευθύνσεων (θα επεξηγηθεί παρακάτω). Οι συσκευές στον δίαυλο I²C είναι είτε Κύριοι (Masters) είτε Υποτελείς (Slave). Η Master συσκευή είναι αυτή που ελέγχει και οδηγεί τη γραμμή ρολογιού SCL (παράγει τους παλμούς ρολογιού). Οι Slave συσκευές είναι αυτές που ανταποκρίνονται στις συσκευές Master. Μία συσκευή Slave δεν μπορεί να ξεκινήσει μία μεταφορά πάνω στο

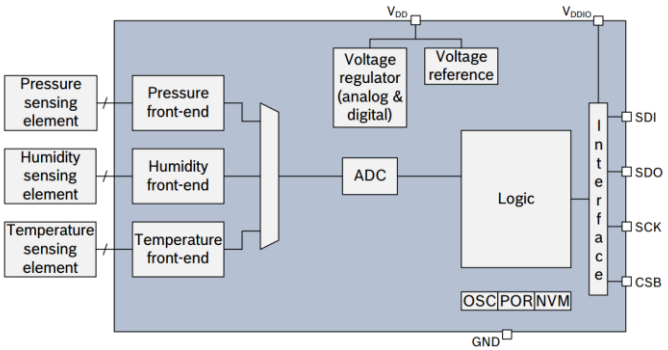
δίαυλο, μόνο μία συσκευή Master μπορεί. Σε έναν δίαυλο μπορεί να είναι συνδεμένες πολλές Master και πολλές Slave συσκευές. Και οι Master και οι Slave συσκευές μπορούν να μεταφέρουν δεδομένα στον δίαυλο, αλλά μόνο οι Master συσκευές ελέγχουν την μεταφορά.

Σε όλες τις slave συσκευές που συνδέονται στον δίαυλο, έχει αποδοθεί ένας αριθμός σαν διεύθυνση. Οι master συσκευές δεν είναι απαραίτητο να έχουν διεύθυνση, εκτός εάν υπάρχουν πολλές master συσκευές στον δίαυλο (περιβάλλον Multi-master). Οι master συσκευές μπορούν να διαλέξουν αυθαίρετα μία από τις συνδεμένες slave συσκευές για επικοινωνία, χρησιμοποιώντας τη διεύθυνσή της. Οι διευθύνσεις των συσκευών του I²C διαύλου είναι είτε 7 bit (θεωρητικά έως 128 συσκευές στο δίαυλο), είτε 10 bit (θεωρητικά έως 1024 συσκευές στο δίαυλο) ή ακόμη και 16 bit (θεωρητικά 65536 συσκευές στο δίαυλο).

3.7 Περιγραφή αισθητήρα θερμοκρασίας, υγρασίας, πίεσης (BME280)

Ο BME280 είναι ένας συνδυαστικός αισθητήρας θερμοκρασίας, υγρασίας και ατμοσφαιρικής πίεσης, υψηλής ακρίβειας, ευαισθησίας και ανάλυσης με χαμηλά επίπεδα θορύβου.

Ο αισθητήρας παρέχει SPI και I^2C interface και η τάση λειτουργίας του είναι 1.71 έως 3.6V. Οι μετρήσεις μπορούν να παίρνονται κατ' εντολή από τον host, ή να γίνονται σε τακτά χρονικά διαστήματα. Το block διάγραμμα του αισθητήρα είναι το παρακάτω(Εικόνα 9):



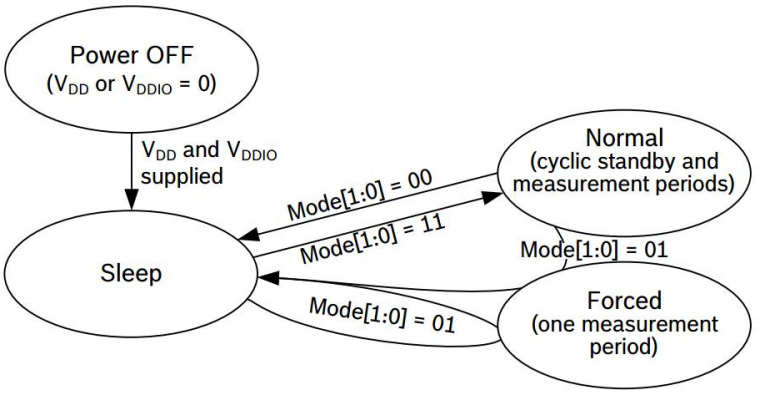
Εικόνα 9 Block διάγραμμα του αισθητήρα (BME280)

Ο αισθητήρας έχει τρεις καταστάσεις λειτουργίας:

- Sleep mode: δεν παίρνει μετρήσεις, πλήρης πρόσβαση στους καταχωρητές, default κατάσταση μετά την εκκίνηση του αισθητήρα

- Forced mode: παίρνει μία μέτρηση, αποθηκεύει το αποτέλεσμα και επιστρέφει σε sleep mode
- Normal mode: διαρκής κύκλος μετρήσεων και περίοδοι αδράνειας.

Παρακάτω φαίνεται το transition diagram του αισθητήρα(Εικόνα 10):



Εικόνα 10 Transition diagram BME280

Η ανάγνωση των δεδομένων από τον συγκεκριμένο αισθητήρα, και κυρίως η δημιουργία των πραγματικών τιμών από τις raw τιμές που παίρνουμε από τους καταχωρητές, είναι μία πολύπλοκη διαδικασία. Για το λόγο αυτό η Bosch συνιστά τη χρήση του BME280 API που έχει φτιάξει για τον σκοπό αυτό.

3.8 Περιγραφή του Analog to Digital Converter (MCP3424)

Το MCP3424 είναι ένα 18-bit delta-sigma analog-to-digital converter (ΔΣ A/D), χαμηλού θορύβου και υψηλής ακρίβειας της εταιρίας Microchip. Μπορεί να μετατρέψει αναλογικές εισόδους σε ψηφιακούς κώδικες με ανάλυση έως 18bit. Η τάση λειτουργίας είναι από 2.7V έως 5.5V.

Ο ρυθμός που μπορεί να μετατρέψουν τις αναλογικές τιμές σε ψηφιακές είναι προγραμματιζόμενος με configuration bit settings σε 3.75, 15, 60 ή 240 samples per second. Ο προγραμματισμός αυτός, αλλά και η ανάγνωση των τιμών, γίνεται με I^2C interface.

Προγραμματιζόμενο είναι επίσης και το gain του ADC, το οποίο μπορεί να πάρει τιμές x1, x2, x4 ή x8 πριν τη μετατροπή. Αυτό μας επιτρέπει να μετατρέψουμε ένα πολύ αδύναμο σήμα εισόδου, με υψηλή ανάλυση.

Το MCP3424 μπορεί να λειτουργήσει σε δύο modes:

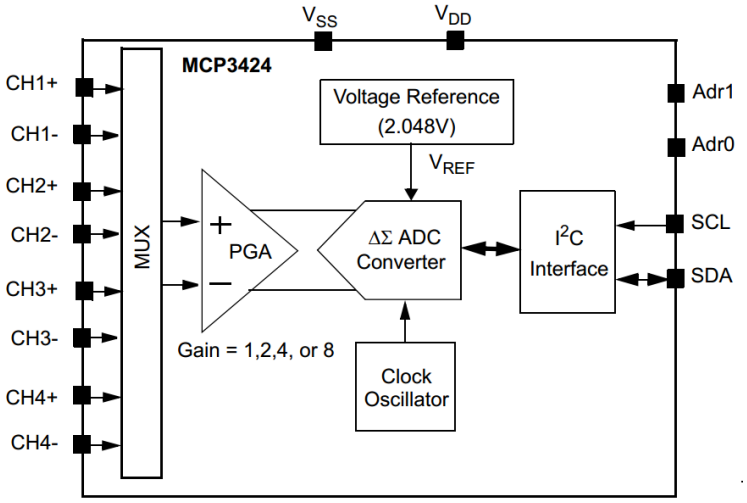
1. One-shot conversion mode: Κάνει μία μετατροπή και αυτόματα γυρίζει σε stand-by mode μέχρι να πάρει πάλι εντολή για νέα μετατροπή.
2. Continuous mode: Κάνει συνεχώς μετατροπές, με μία ταχύτητα που έχουμε θέσει εμείς.

Η συσκευή ενημερώνει τον output buffer με την πιο πρόσφατη ψηφιακή τιμή μετατροπής.

Το MCP3424 υποστηρίζει από το εργοστάσιο 8 I^2C διευθύνσεις. Η I^2C διεύθυνση είναι επιλεγόμενη από δύο pins. Μπορούμε να θέσουμε τα pins αυτά σε HIGH, LOW ή unconnected, και έτσι να δηλώσουμε ποια από τις 8 διαθέσιμες διευθύνσεις επιθυμούμε.

Το MCP3424 διαθέτει 4 differential εισόδους CH1+ CH1-, CH2+ CH2-, CH3+ CH3-, CH4+ CH4-. Αν συνδέσουμε τις CH1- CH2- CH3- CH4- στη γείωση, μπορούμε να έχουμε 4 εισόδους single ended. Στην εφαρμογή αυτή χρειαζόμαστε single-ended εισόδους.

Παρακάτω φαίνεται το block διάγραμμα του MCP3424(Εικόνα 11):



Εικόνα 11 Block διάγραμμα του MCP3424

Διακρίνονται οι τέσσερεις αναλογικοί είσοδοι, τα δύο pins επιλογής I^2C διεύθυνσης, και τα clock και data του I^2C interface (SDA, SCL). Τα V_{DD} και V_{SS} είναι η τροφοδοσία και η γείωση του ADC αντίστοιχα.

Όπως διακρίνεται στο παραπάνω block διάγραμμα, η τάση αναφοράς (voltage reference) είναι 2.048V. Σε αντίθεση με άλλα ADC, στο MCP3424 δεν υπάρχει δυνατότητα

προγραμματιζόμενης τάσης αναφοράς. Αυτό στη πορεία της υλοποίησης αποδείχθηκε ότι είναι πρόβλημα, καθώς τάσεις εισόδου μεγαλύτερες από 2.048V δεν μπορούν να μετρηθούν. Όπως θα δούμε όμως στη συνέχεια, μία standard συνδεσμολογία του συστήματος μέτρησης κατεύθυνσης ανέμου παράγει τάσεις μεγαλύτερες από την τάση αναφοράς του ADC. Η λύσεις ήταν δύο: να αγοραστεί κάποιο άλλο ακριβότερο ADC που διέθετε προγραμματιζόμενη τάση αναφοράς, ή να φτιαχτεί ένα άλλο κύκλωμα για τον σύστημα κατεύθυνσης ανέμου. Επιλέχτηκε η δεύτερη λύση, η οποία θα αναλυθεί αργότερα.

Η συσκευή πραγματοποιεί μετατροπές χρησιμοποιώντας την εσωτερική τάση αναφοράς (V_{REF}=2,048V). Συνεπώς η απόλυτη τιμή της διαφορικής τάσης (V_{IN}), συμπεριλαμβανομένου του PGA, πρέπει να είναι μικρότερη της εσωτερικής τάσης αναφοράς. Η συσκευή θα δώσει χαρακτηριστικές τιμές κορεσμού για την ψηφιακή τιμή που υπολογίστηκε (όλα ‘1’ ή όλα ‘0’ εκτός από το sign bit), αν η απόλυτη τιμή της διαφορικής τάσης (V_{IN}), συμπεριλαμβανομένου του PGA, είναι μεγαλύτερη από την εσωτερική τάση αναφοράς (V_{REF}=2,048V). Το εύρος των τάσεων δίνεται από την παρακάτω σχέση:

$$-V_{REF} \le (V_{IN} \bullet PGA) \le (V_{REF} - 1LSB)$$

Where:

V_{IN} = CHn+ - CHn-

V_{REF} = 2.048V

Αν η τάση εισόδου είναι μεγαλύτερη από το παραπάνω όριο, μπορούμε να χρησιμοποιήσουμε έναν διαιρέτη τάσης, έτσι ώστε να φέρουμε την τάση εισόδου εντός των παραπάνω ορίων. Αυτό ακριβώς κάναμε για αυτό project, όπως άλλωστε θα αναλυθεί στη συνέχεια, στη παράγραφο που αφορά το σύστημα μέτρησης κατεύθυνσης του ανέμου.

Η ψηφιακή τιμή εξόδου είναι σχετική με την τάση εισόδου και της τιμής του PGA. Το output data format είναι σε two’s complement. Συνεπώς το MSB μπορεί να θεωρηθεί σαν sign bit. Όταν το MSB είναι ‘0’ η είσοδος είναι θετική. Όταν το MSB είναι ‘1’ η είσοδος είναι αρνητική. Για παράδειγμα:

a) Για ελάχιστη αρνητική τιμή εισόδου: 100…000
   π.χ. (CH_n + -CH_{n-}) * PGA = -2,048V

b) Για μηδενική διαφορική τιμή εισόδου: 000…000
π.χ. (CH_n + -CH_{n-}) = 0

c) Για μέγιστη θετική τιμή εισόδου: 011…111
π.χ. (CH_n + -CH_{n-}) * PGA = 2,048V

Το MSB πάντα αποστέλλεται πρώτο μέσω του I^2C. Η ψηφιακές τιμές εξόδου δεν θα κάνουν roll-over ακόμα και αν η τάση εισόδου ξεπεράσει το μέγιστο όριο. Σε αυτή τη περίπτωση η τιμή εξόδου θα είναι κλειδωμένη στη τιμή 0111…11 για όλες τις τάσεις εισόδου που είναι μεγαλύτερες από (V_{REF} – 1 LSB) / PGA. Ο παρακάτω πίνακας δείχνει τις τιμές εξόδου για διάφορες τιμές εισόδου για ανάλυση 18 bit.

Input Voltage: [CHn+ - CHn-] • PGA	Digital Output Code
≥ V_{REF}	011111111111111111
V_{REF} - 1 LSB	011111111111111111
2 LSB	000000000000000010
1 LSB	000000000000000001
0	000000000000000000
-1 LSB	111111111111111111
-2 LSB	111111111111111110
- V_{REF}	100000000000000000
< -V_{REF}	100000000000000000

Note 1: MSB is a sign indicator:
0: Positive input (CHn+ > CHn-)
1: Negative input (CHn+ < CHn-)
2: Output data format is binary two's complement.

Στον παρακάτω πίνακα φαίνονται οι ελάχιστες και οι μέγιστες τιμές των ψηφιακών τιμών εξόδου:

Resolution Setting	Data Rate	Minimum Code	Maximum Code
12	240 SPS	-2048	2047
14	60 SPS	-8192	8191
16	15 SPS	-32768	32767
18	3.75 SPS	-131072	131071

Note: Maximum n-bit code = 2^{N-1} - 1
Minimum n-bit code = -1 x 2^{N-1}

Όπως είδαμε το MCP3424 υποστηρίζει ανάλυση 12, 14, 16 ,18 bits. Η αντιστοιχία της ρύθμισης ανάλυσης σε bits και της ανάλυσης σε Volt που μπορούμε να έχουμε φαίνεται στον παρακάτω πίνακα:

Resolution Setting	LSB
12 bits	1 mV
14 bits	250 µV
16 bits	62.5 µV
18 bits	15.625 µV

Ο μαθηματικός τύπος για αυτόν τον υπολογισμό του παραπάνω πίνακα είναι:

$$LSB = \frac{2 \times V_{REF}}{2^N} = \frac{2 \times 2.048V}{2^N}$$

Where:

N = Resolution, which is programmed in the Configuration Register.

Για την εφαρμογή μας, μία ανάλυση της τάξης 1mV είναι αρκετή, οπότε 12bits είναι η κατάλληλη ρύθμιση.

Όταν πάρουμε την ψηφιακή τιμή εξόδου από το ADC, το επόμενο βήμα είναι μετατρέψουμε την τιμή αυτή στην πραγματική τάση εισόδου που μετρήθηκε σε Volt. Ο τύπος για αυτή τη μετατροπή είναι:

If MSB = 0 (Positive Output Code):

$$Input\ Voltage = (Output\ Code) \bullet \frac{LSB}{PGA}$$

If MSB = 1 (Negative Output Code):

$$Input\ Voltage = (2\text{'s complement of Output Code}) \bullet \frac{LSB}{PGA}$$

Where:

LSB = See the table above

2's complement = 1's complement + 1

Από τον παρακάτω τύπο προκύπτει ότι για την περίπτωσή μας όπου θα έχουμε μόνο θετικές τιμές εισόδου (MSB '0'), η τάση αυτή θα προκύπτει πολλαπλασιάζοντας την ψηφιακή τιμή εξόδου με το LSB (βάση του αντίστοιχου πίνακα παραπάνω) και διαιρώντας με το PGA.

Η συσκευή έχει έναν 8-bit configuration register για να επιλέξουμε: input channels, conversion mode, και PGA gain. Αυτός ο καταχωρητής μας επιτρέπει να αλλάξουμε τις συνθήκες λειτουργίας της συσκευής και να ελέγξουμε τη κατάσταση λειτουργίας της. Μπορούμε να τροποποιήσουμε την τιμή του καταχωρητή όσες φορές θέλουμε κατά τη

διάρκεια της λειτουργίας της συσκευής. Παρακάτω φαίνεται η δομή του configuration register:

R/W-1	R/W-0	R/W-0	R/W-1	R/W-0	R/W-0	R/W-0	R/W-0
$\overline{RDY}$	C1	C0	$\overline{O}$/C	S1	S0	G1	G0
1 *	0 *	0 *	1 *	0 *	0 *	0 *	0 *
bit 7							bit 0

* Default Configuration after Power-On Reset

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7 **$\overline{RDY}$:** Ready Bit

Αυτό είναι το data ready flag. Σε read mode, αυτό το bit δείχνει αν ο output register έχει ενημερωθεί με την τελευταία τιμή μετατροπής. Σε One-Shot mode, γράφοντας '1' σε αυτό το bit ξεκινάει μία νέα μετατροπή.

Διαβάζοντας το $\overline{RDY}$ bit με μία read command:

1 = Ο output register δεν έχει ενημερωθεί

0 = Ο output register έχει ενημερωθεί με την τελευταία τιμή μετατροπής

Γράφοντας το $\overline{RDY}$ bit με μία write command:

Continuous Conversion Mode: Καμία επίδραση

One-Shot Conversion Mode:

1 = Ξεκινάει μία νέα μετατροπή

0 = Καμία επίδραση

bit 6-5 **C1-C0:** Channel Selection Bits

00 = Select Channel 1 **(Default)**

01 = Select Channel 2

10 = Select Channel 3

11 = Select Channel 4

bit 4 **$\overline{O}/C$:** Conversion Mode Bit

1 = Continuous Conversion Mode (**Default).** Η συσκευή κάνει μετατροπές συνεχώς

0 = One-Shot Conversion Mode. Η συσκευή κάνει μόνο μία μετατροπή και μεταβαίνει σε low power standby mode, μέχρι να λάβει κάποια νέα read ή write εντολή.

bit 3-2 **S1-S0:** Sample Rate Selection Bit

00 = 240 SPS (12 bits) **(Default)**

01 = 60 SPS (14 bits)

10 = 15 SPS (16 bits)

11 = 3.75 SPS (18 bits)

bit 1-0 **G1-G0:** PGA Gain Selection Bits
00 = x1 **(Default)**
01 = x2
10 = x4
11 = x8

Όταν στέλνουμε μία write command στο I^2C bus, η συσκευή περιμένει δύο byte: το πρώτο byte είναι η διεύθυνση της συσκευής και το δεύτερο byte είναι το configuration byte που θα γραφτεί στον configuration register. Όποιο άλλο byte τυχών ακολουθεί το δεύτερο, αγνοείται.

Όταν στέλνουμε μία read command στο I^2C bus, η συσκευή στέλνει και τα conversion data και το configuration byte.

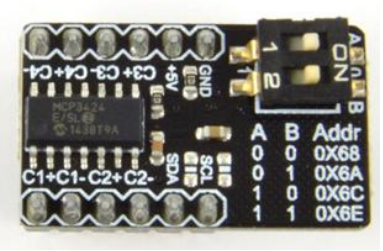
Όταν η συσκευή είναι ρυθμισμένη για 18-bit μετατροπή, στέλνει τρία data bytes και το configuration byte. Τα πρώτα 6 data bits του πρώτου data byte είναι επαναλαμβανόμενο το MSB (=sign bit) των conversion data. Μπορούμε να αγνοήσουμε τα πρώτα 6 data bits, και να πάρουμε το $7^{ο}$ bit σαν το MSB των conversion data. Το LSB του $3^{ου}$ data byte είναι το LSB των conversion data.

Όταν η συσκευή είναι ρυθμισμένη για 12, 14 ή 16-bit μετατροπή, στέλνει δύο data bytes και το configuration byte. Σε ρύθμιση για 12-bit μετατροπή, όπως στη περίπτωσή μας, τα τέσσερα πρώτα bits είναι επαναλαμβανόμενο το MSB (=sign bit) και μπορούν να αγνοηθούν. Το πέμπτο bit (D11) είναι το MSB (=sign bit) των conversion data. Το LSB του $2^{ου}$ data byte είναι το LSB των conversion data.

Conversion Option	Digital Output Codes
18-bits	MMMMMMD17D16 (1st data byte) - D15 ~ D8 (2nd data byte) - D7 ~ D0 (3rd data byte) - Configuration byte. **(Note 1)**
16-bits	D15 ~ D8 (1st data byte) - D7 ~ D0 (2nd data byte) - Configuration byte. **(Note 2)**
14-bits	MMD13D ~ D8 (1st data byte) - D7 ~ D0 (2nd data byte) - Configuration byte. **(Note 3)**
12-bits	MMMMD11 ~ D8 (1st data byte) - D7 ~ D0 (2nd data byte) - Configuration byte. **(Note 4)**

Note 1: D17 is MSB (= sign bit), M is repeated MSB of the data byte.
2: D15 is MSB (= sign bit).
3: D13 is MSB (= sign bit), M is repeated MSB of the data byte.
4: D11 is MSB (= sign bit), M is repeated MSB of the data byte.

Όπως είδαμε νωρίτερα, για τις ανάγκες του project χρησιμοποιήθηκε το ολοκληρωμένο σύστημα από την DFROBOT (Εικόνα 12).



Εικόνα 12 MCP3424 DFROBOT

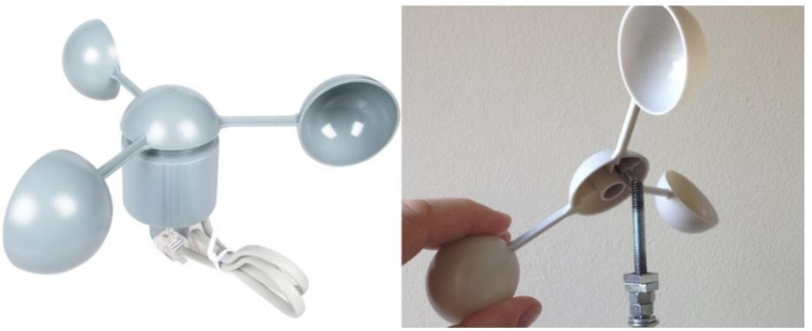
Όπως μπορούμε να διακρίνουμε, έχει ένα dip switch δύο επαφών για την επιλογή της επιθυμητής I^2C address. Ακριβώς κάτω από αυτό και πάνω στη πλακέτα, αναγράφονται οι συνδυασμοί των pin του dip switch και η αντίστοιχη διεύθυνση. Η ρύθμιση που επιλέξαμε για το project είναι A=0, B=0 επομένως I^2C address 0x68.

Είναι σημαντικό να αναφερθεί εδώ, πως παρόλο που το ADC λειτουργεί σύμφωνα με τον κατασκευαστή με τάση τροφοδοσίας 2,7V – 5,5V, όταν δώσαμε τάση 3,3V από το Raspberry, το ADC δεν λειτουργούσε σωστά. Μετά από πολύ έρευνα και πολλές δοκιμές, δοκιμάσαμε να αλλάξουμε την τροφοδοσία, και δώσαμε 5V πάλι από το Raspberry. Όλα τα προβλήματα σταμάτησαν να υπάρχουν και η συσκευή λειτουργούσε πλέον κανονικά.

3.9 Αισθητήρας μέτρησης ταχύτητας ανέμου

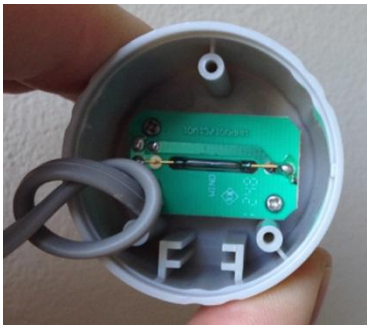
Ο αισθητήρας μέτρησης ταχύτητας ανέμου αποτελεί μέρους του μετεωρολογικού σταθμού της SparkFun όπως αναφέρθηκε νωρίτερα.

Έχει τρεις βραχίονες με «κουτάλες» στις άκρες οι οποίες πιάνουν τον αέρα και περιστρέφουν τους βραχίονες (Εικόνα 13). Αν αποσυναρμολογήσουμε το ανεμόμετρο, θα βρούμε ένα μικρό μαγνήτη ενσωματωμένο στο κάτω μέρος.



Εικόνα 13 Αισθητήρας μέτρησης ταχύτητας ανέμου SparkFun

Επίσης υπάρχει και ένα reed switch πάνω σε μία πλακέτα. Αυτός λειτουργεί ως διακόπτης που ανοίγει και κλείνει, καθώς ο μαγνήτης περνάει από πάνω του (Εικόνα 14 και 15).



Εικόνα 14 Αισθητήρας μέτρησης ταχύτητας ανέμου SparkFun

Εικόνα 15 Μαγνήτης αισθητήρα μέτρησης ταχύτητας ανέμου

Επομένως, ο αισθητήρας ταχύτητας ανέμου είναι ένας μαγνήτης και ένα μαγνητικό reed switch. Καθώς ο αισθητήρας περιστρέφεται ο μαγνήτης περνάει πάνω από το switch, αναγκάζοντάς το κα κλείσει. Μία ταχύτητα ανέμου 2,4 km/h θα έχει ως αποτέλεσμα ο διακόπτης να κλείσει 1 φορά το δευτερόλεπτο. Επομένως πρέπει απλώς να μετρήσουμε πόσες κλείνει ο διακόπτης σε κάθε δευτερόλεπτο για να υπολογίσουμε τη ταχύτητα του ανέμου.

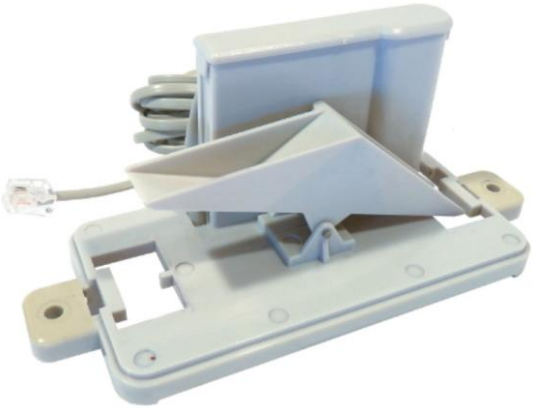
Στο Raspberry αυτό γίνεται εύκολα, παρατηρώντας τις αλλαγές στη κατάσταση ενός GPIO pin που συνδέεται με το switch και καταγράφοντας αυτές τις αλλαγές σε μία μονάδα του χρόνου.

Το πρόβλημα όμως είναι ότι στο datasheet δεν αναφέρεται με ακρίβεια η λειτουργία του αισθητήρα. Συγκεκριμένα, μετρώντας τα άκρα των καλωδίων του αισθητήρα παρατηρήσαμε ότι ο αισθητήρας μένει σε HIGH (switch open) για ~90^{o} περιστροφής, και μετά μένει σε LOW (switch close) για 90^{o} περιστροφής. Επομένως, μπορούμε να μετρήσουμε όχι τα LOW ανά δευτερόλεπτο που αντιστοιχούν σε ταχύτητα αέρα 2.4 km/h, αλλά και τα LOW και τα HIGH ανά δευτερόλεπτο που αντιστοιχούν σε ταχύτητα αέρα 1,2 km/h. Μετρώντας δηλαδή όλες τις αλλαγές (και H>L και L->H) μπορούμε να διπλασιάσουμε την ανάλυση του αισθητήρα.

3.10 Αισθητήρας μέτρησης ύψους βροχής

Παρόμοια με τον αισθητήρα ταχύτητας ανέμου, ο αισθητήρας ύψους βροχής (Εικόνα 16 και 17) λειτουργεί με έναν μαγνήτη και ένα reed switch. Μέσα στον αισθητήρα υπάρχει μία συσκευή ανατροπής (κάτι σαν τραμπάλα), η οποία ταλαντώνεται από τη μία και από την άλλη μεριά με το βάρος του νερού που πέφτει σε αυτήν, κάνοντας μία κίνηση για κάθε 0,2794 mm βροχής που πέφτει.

Εικόνα 16 Αισθητήρας Βροχής



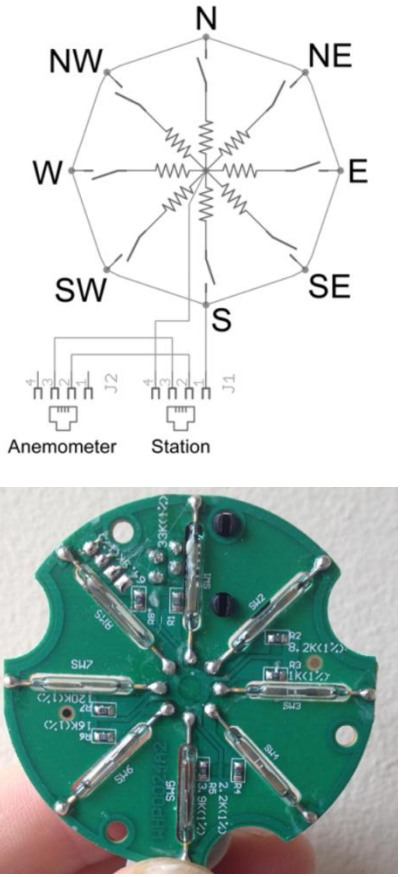
Εικόνα 17 Αισθητήρας Βροχής

Όταν ένα bucket γεμίσει με νερό και γύρει προς τη μεριά του, ο μαγνήτης περνάει μπροστά από το reed switch προκαλώντας το στιγμιαίο κλείσιμό του. Επομένως, αν

συνδέσουμε το σύστημα σε ένα GPIO pin του Raspberry, μπορούμε να μετράμε πόσες φορές έκλεισε το reed switch. Κάτι παρόμοιο δηλαδή με το ανεμόμετρο. Η διαφορά εδώ είναι ότι μας ενδιαφέρει -και πρέπει να μετράμε- μόνο τα κλεισίματα του reed switch, δηλαδή πρακτικά θα πρέπει να μετράμε στο Raspberry πόσες φορές το GPIO pin πέρασε από HIGH σε LOW.

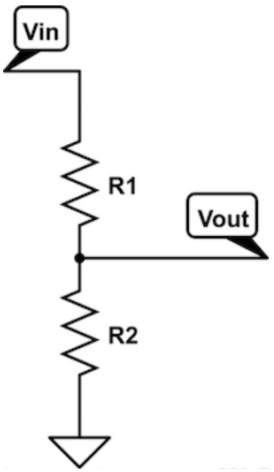
3.11 Αισθητήρας κατεύθυνσης ανέμου

Ο αισθητήρας κατεύθυνσης ανέμου είναι ο πιο πολύπλοκος από τους τρεις. Έχει οκτώ διακόπτες όπου καθένας είναι συνδεδεμένος σε μία διαφορετική αντίσταση. Ο μαγνήτης του αισθητήρα είναι πιθανό να κλείσει δύο διακόπτες ταυτόχρονα, επιτρέποντας έτσι μέχρι και 16 διαφορετικές θέσεις (Εικόνα 18).



Εικόνα 18 Αισθητήρας κατεύθυνσης ανέμου

Με τη χρήση μιας εξωτερικής αντίστασης και σε συνδυασμό με τις αντιστάσεις του αισθητήρα μπορούμε να φτιάξουμε έναν διαιρέτη τάσης (voltage divider). Η τάση εξόδου αυτού του διαιρέτη μπορεί να μετρηθεί με ένα αναλογικό σε ψηφιακό μετατροπέα (A/D converter).

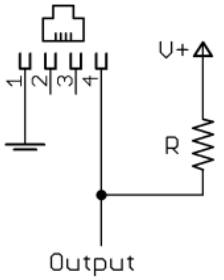


Ο διαιρέτης τάσης είναι ένα βασικό κύκλωμα στην ηλεκτρονική, που χρησιμοποιείται όταν θέλουμε να μειώσουμε μία μεγάλη τάση σε μία μικρότερη, και λειτουργεί ως εξής: δίνουμε μία επιθυμητή τάση V_{in} και επιλέγουμε μία αντίσταση R_1. Αυτό το κύκλωμα, σε συνδυασμό με την αντίσταση εντός του αισθητήρα (R_2) που ενεργοποιείται ανάλογα με τον διακόπτη που κλείνει, αποτελεί έναν διαιρέτη τάσης. Η τάση εξόδου V_{out} προκύπτει από τον παρακάτω μαθηματικό τύπο:

$$V_{out} = \frac{R_2}{R_1 + R_2} * V_{in}$$

όπου:

- R_1 είναι η αντίσταση R της επιλογής μας
- R_2 είναι η αντίσταση του αισθητήρα ανάλογα με τον διακόπτη που έχει κλείσει
- V_{in} η τάση επιλογής μας για την υλοποίηση του διαιρέτη
- V_{out} η τάση εξόδου που θα μετρήσουμε με ένα ADC



Στο παραπάνω σχήμα βλέπουμε τον RJ-11 connector του αισθητήρα, όπου το pin 1 συνδέεται στη γη και το pin 4 στην αντίσταση του διαιρέτη τάσης.

Στον παρακάτω πίνακα δίνονται σαν παράδειγμα από το datasheet του κατασκευαστή, η αντίσταση του αισθητήρα ανάλογα με την κατεύθυνση του ανέμου σε μοίρες, καθώς και η τάση εξόδου V_{Output} που θα προκύψει αν βάλουμε για V^+ = 5V και R_1=10kΩ.

Direction (Degrees)	Resistance (Ohms)	Voltage (V=5v, R=10k)
0	33k	3.84v
22.5	6.57k	1.98v
45	8.2k	2.25v
67.5	891	0.41v
90	1k	0.45v
112.5	688	0.32v
135	2.2k	0.90v
157.5	1.41k	0.62v
180	3.9k	1.40v
202.5	3.14k	1.19v
225	16k	3.08v
247.5	14.12k	2.93v
270	120k	4.62v
292.5	42.12k	4.04v
315	64.9k	4.78v
337.5	21.88k	3.43v

Το πρόβλημα στη περίπτωσή μας με τον παραπάνω πίνακα, είναι ότι υπάρχουν τάσεις εξόδου οι οποίες είναι άνω των 2,048V που είναι η τάση αναφοράς του ADC που έχουμε επιλέξει, επομένως όπως αναλύθηκε νωρίτερα αυτές οι τάσεις δεν μπορούν να

μετρηθούν. Η λύση ήταν να επιλέξουμε δικές μας τιμές για V^+ και για R_1 έτσι ώστε να προκύπτουν τάσεις εξόδου V_{Output} που να μπορούν να μετρηθούν από το ADC.

Έπειτα από αρκετούς υπολογισμούς και μετρήσεις με τα πολύμετρο, επιλέξαμε:

V^+ = 3,3V

R_1 = 100 kΩ

Ο πίνακας που προκύπτει είναι:

Direction (Degrees)	Resistance (Ω)	Voltage (V= 3,3V , R = 100 kΩ
0	33k	0,819 V
22,5	6,57k	0,203 V
45	8,2k	0,250 V
67,5	891	0,029 V
90	1k	0,032 V
112,5	688	0,022 V
135	2,2k	0,071 V
157,5	1,41k	0,046 V
180	3,9k	0,124 V
202,5	3,14k	0,100 V
225	16k	0,455 V
247,5	14,12k	0,408 V
270	120k	1,8 V
292,5	42,12k	0,978 V
315	64,9k	1,299 V
337,5	21,88k	0,592 V

Στον παραπάνω πίνακα παρατηρούμε ότι η μέγιστη τάση εξόδου είναι 1,8V η οποία είναι εντός των προβλεπόμενων ορίων. Επίσης μερικές από τις τάσεις εξόδου είναι ύψους μερικών δεκάδων mV, πράγμα το οποίο δεν αποτελεί πρόβλημα, καθώς στα 12 bit ανάλυση που έχουμε επιλέξει να ρυθμίσουμε το ADC, έχουμε ακρίβεια 1mV, η οποία είναι ικανοποιητική. Σε αντίθετη περίπτωση, θα έπρεπε να επιλέξουμε υψηλότερη ανάλυση στο ADC (π.χ. 14, 16 ή 18 bit).

Κεφάλαιο 4 - Windows 10 IOT Core

4.1 Windows 10 IoT Core

Τα Windows 10 IoT Core είναι μία ειδική δωρεάν έκδοση των Windows 10, η οποία είναι optimized για μικρές συσκευές με ή χωρίς οθόνη, η οποία τρέχει σε x86/x64 και ARM συσκευές, όπως το Raspberry PI.

Διαφορές μεταξύ Windows 10 Desktop και Windows 10 IoT Core

Διαφορετικά features διαθέσιμα στα Desktop και στα IoT Core:

- Τα Windows 10 IoT Core θα κάνουν boot σε μία default app αντί για το κλασσικό desktop ενός desktop-like PC. Η default app μπορεί να είναι μία custom εφαρμογή ή η default εφαρμογή των Windows 10 IoT core. Αυτή η default app των windows παρέχει ένα στοιχειώδες περιβάλλον για πρόσβαση στο shell και σε στοιχειώδη πίνακα ελέγχου.
- Το FileOpenPicker API δεν υποστηρίζεται στα Windows 10 IoT Core. Για να προσπελάσουμε local drives ή removable storage, πρέπει να φτιάξουμε δική μας εφαρμογή και να υλοποιήσουμε αυτή τη λειτουργία.
- Τα Windows 10 Desktop υποστηρίζουν περισσότερους drivers από ότι τα Windows 10 IoT Core. Για να λειτουργήσει η ίδια(ες) συσκευή(ές) στα Windows 10 IoT Core όπως στα Desktop, ίσως χρειαστεί να φτιάξουμε δικούς μας drivers ή να κάνουμε κάποιο workaround, ιδίως σε ARM συσκευές.

4.2 Υποστηριζόμενη έκδοση Windows 10 IoT και γνωστά issues

Όπως είπαμε και πιο πριν, το Raspberry PI 3 Model B+ έχει περιορισμένη συμβατότητα με τα Windows 10 IoT core. Η μόνη έκδοση Windows 10 IoT Core που υποστηρίζει το Raspberry PI 3 Model B+ είναι η technical insider preview έκδοση Build 17661. Καμία από τις νεότερες insider preview καθώς και καμία commercial έκδοση δεν υποστηρίζεται. Και αυτό είναι πολύ σημαντικό, καθώς αν δοκιμάσει κάποιος να εγκαταστήσει νεότερη insider preview ή commercial έκδοση, δημιουργούνται μεγάλα και περίεργα προβλήματα στη εγκατάσταση και στη λειτουργία του συστήματος. Επίσης η insider preview Build 17661 λειτουργεί μόνο σε PI Model B+, και δεν θα κάνει boot π.χ. σε PI 2.

Σύμφωνα με τη Microsoft μερικά από τα γνωστά issues σε αυτή την έκδοση είναι:

- Δεν λειτουργεί το On-Board WiFi
- Δεν λειτουργεί το On-Board Bluetooth
- Δεν λειτουργεί ο driver για την touch screen
- Δεν λειτουργεί το activity LED της SD card
- Οι επιδόσεις αναπαραγωγής video είναι περιορισμένες
- Η PiCAM που συνδέεται στο on-board camera bus δεν υποστηρίζεται
- Η υποστήριξη συσκευών κάμερας USB είναι περιορισμένη
- Ορισμένα USB πληκτρολόγια και ποντίκια δεν υποστηρίζονται

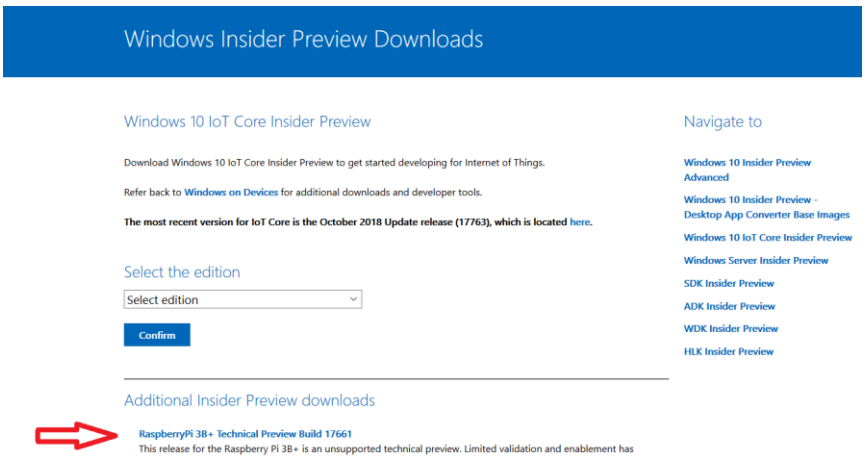
Η Microsoft έχει δημιουργήσει μία λίστα με τις πιστοποιημένες συσκευές που υποστηρίζονται.

4.3 Εγκατάσταση των Windows 10 IoT Core στο Raspberry PI

Για τα board που υποστηρίζονται από τα Windows IoT, η Microsoft παρέχει ένα ready-made Full Flash Update (FFU) image, κάνοντας έτσι πάρα πολύ εύκολη τη διαδικασία εγκατάστασης των Windows 10 IoT στη συσκευή μας.

Επειδή όπως είπαμε παραπάνω, η μόνη έκδοση Windows 10 IoT Core που υποστηρίζει το Raspberry PI 3 Model B+ είναι insider preview, δημιουργήσαμε ένα insider account στη Microsoft. Αυτό έγινε γιατί, για να κατεβάσει κάποιος μία insider preview έκδοση πρέπει να γραφτεί ως insider στη Microsoft.

Στην Insider Preview download page της Microsoft(Εικόνα 19) προσέχουμε να κατεβάζουμε την σωστή έκδοση για Raspberry PI 3 B+ από το σημείο που δείχνει το κόκκινο βέλος.



Εικόνα 19 Page Insider Preview Download

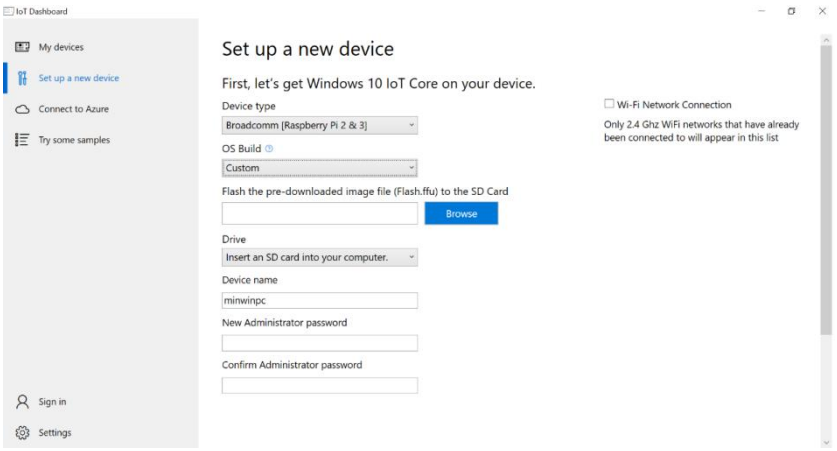
Το image κατεβαίνει σε μορφή .iso:
(Windows10_InsiderPreview_IoTCore_RPi3B_en-us_17661.iso).

Ανοίγουμε το αρχείο .iso και μέσα περιέχει ένα .msi αρχείο εγκατάστασης. Το εκτελούμε και ακολουθούμε τις οδηγίες. Στο σημείο εγκατάστασης υπάρχει το .ffu αρχείο, το οποίο είναι το image των Windows 10 IoT Core με το οποίο θα κάνουμε flash το Raspberry.

Τοποθετούμε την micro SD μνήμη όπου θα εγκατασταθούν τα Windows 10 IoT στο PC μας, μέσω του κατάλληλου adaptor (usb, SD, κλπ).

Για να εγκαταστήσουμε το .ffu αρχείο, είναι απαραίτητη η εφαρμογή **Windows 10 IoT Core Dashboard** η οποία είναι διαθέσιμη δωρεάν από τη Microsoft.

Αφού την κατεβάσουμε και την εκτελέσουμε, πατάμε στα αριστερά την επιλογή "**Setup a new device"**, όπως φαίνεται παρακάτω (Εικόνα 20):



Εικόνα 20 Windows 10 IoT Core Dashboard

Στην κύρια οθόνη στο κέντρο επιλέγουμε:

Device type: *Raspberry PI 2 & 3*

OS Build: *Custom*

Browse: *επιλέγουμε το .ffu αρχείο που κατεβάσαμε νωρίτερα*

Drive: *Εδώ θα πρέπει να φαίνεται η micro SD που τοποθετήσαμε νωρίτερα στο PC*

Device name: *Δίνουμε ένα όνομα στη συσκευή μας*

Administrator password: *Ορίζουμε το administrator password των Windows 10 IoT*

Τέλος, καλό είναι στα δεξιά να απενεργοποιήσουμε το wifi. Είναι προτιμότερο να συνδεόμαστε με το Raspberry με Ethernet.

Ξεκινάει η διαδικασία εγκατάστασης. Μόλις ολοκληρωθεί, κλείνουμε το dashboard, βγάζουμε από το PC την SD κάρτα και την τοποθετούμε στο Raspberry, το οποίο είναι έτοιμο πλέον για boot με τα Windows 10 IoT Core.

4.4 Πρώτη εκκίνηση του Raspberry PI με Windows 10 IoT Core

Με την εγκατάσταση των Windows 10 IoT Core, είμαστε έτοιμοι να ξεκινήσουμε το Raspberry για πρώτη φορά.

Αφού το συνδέσουμε με **Ethernet** και όχι με Wifi (το οποίο ούτως ή άλλως δεν λειτουργεί όπως έχουμε πει), βάζουμε και την SD μνήμη στο Raspberry.

Συνδέουμε το Raspberry με μία οθόνη στο HDMI interface, συνδέουμε ένα πληκτρολόγιο και ένα ποντίκι και τέλος το τροφοδοτικό. Ξεκινάει η εκκίνηση των Windows 10 IoT (Εικόνα 21), η οποία ενδέχεται να διαρκέσει αρκετή ώρα, καθώς πιθανότατα θα γίνουν και κάποια updates.

Εικόνα 21 Installation Page

Μόλις ολοκληρωθεί η εκκίνηση, βλέπουμε την παρακάτω αρχική οθόνη, που μας δίνει κάποιες πληροφορίες για το Raspberry (IP address, USB devices, κλπ). Στο σημείο

αυτό το Raspberry λειτουργεί και αυτή η οθόνη αποτελεί την «επιφάνεια εργασίας» των Windows IoT Core (Εικόνα 22).

Η διαχείριση όμως του συστήματος δεν από εδώ, όπως θα δούμε παρακάτω.

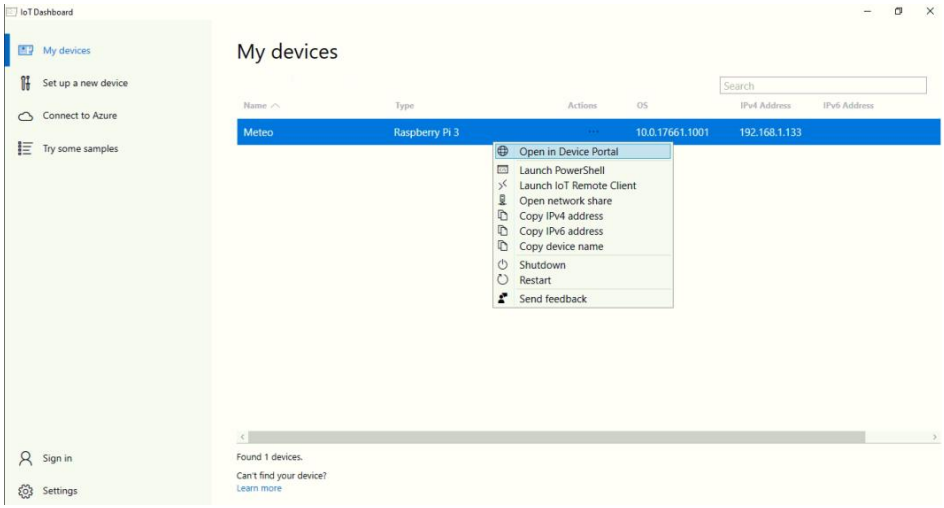
Εικόνα 22 Raspberry Page "Επιφάνεια Εργασίας"

4.5 Διαχείριση του συστήματος

Η διαχείριση του συστήματος Raspberry με Windows 10 IoT, γίνεται πολύ εύκολα από ένα web portal.

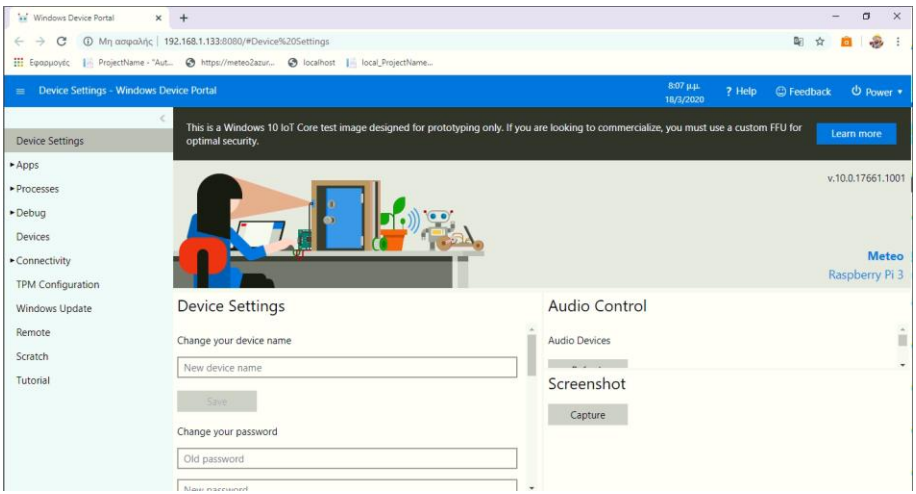
Για να συνδεθούμε, εκτελούμε πρώτα στο PC το Windows 10 IoT dashboard, και επιλέγουμε αριστερά **“My devices”** όπως φαίνεται στην παρακάτω εικόνα. Στην οθόνη θα εμφανιστούν όσα Raspberry είναι στο LAN μας. Ίσως χρειαστούν λίγα λεπτά για να εμφανιστούν οι συσκευές.

Μόλις εμφανιστεί το Raspberry, κάνουμε δεξί κλικ πάνω του και επιλέγουμε **“Open device portal”**. Στη συνέχεια δίνουμε όνομα χρήστη **Administrator** και το password που δηλώσαμε κατά την εγκατάσταση των Windows IoT Core (Εικόνα 23).

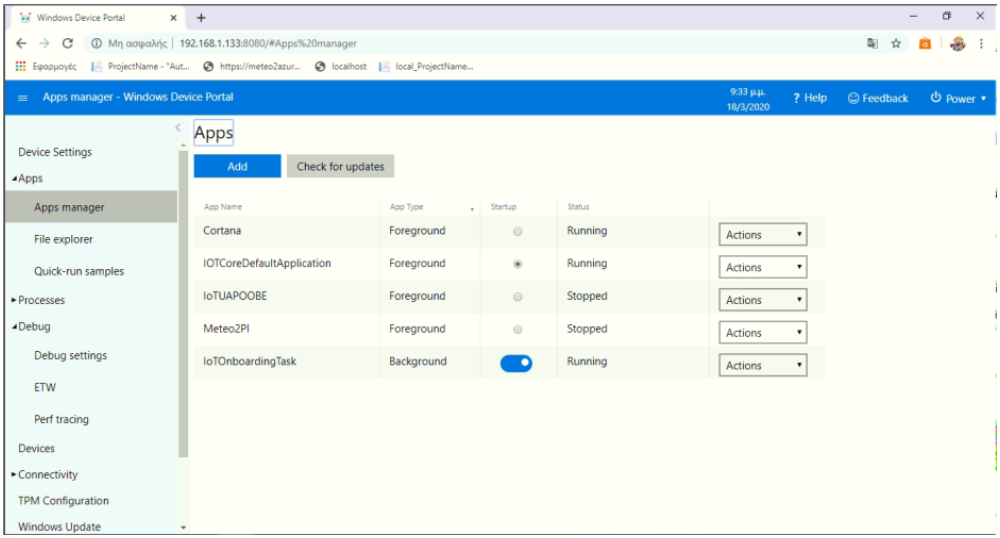


Εικόνα 23 Device Portal

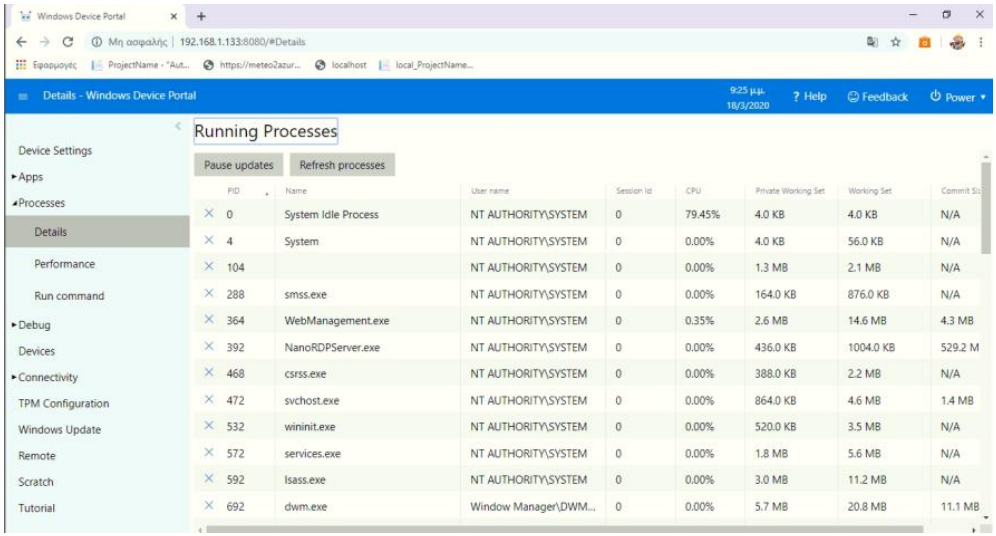
Μόλις συνδεθούμε επιτυχώς θα δούμε την παρακάτω οθόνη του Windows Device Portal (Εικόνα 24). Από εδώ γίνεται όλη η διαχείριση των Windows, από την αλλαγή του administrator password και το shutdown της συσκευής, μέχρι διαχείριση διεργασιών, performance monitoring, device management, κλπ (Εικόνες 24-25-26-27-28).



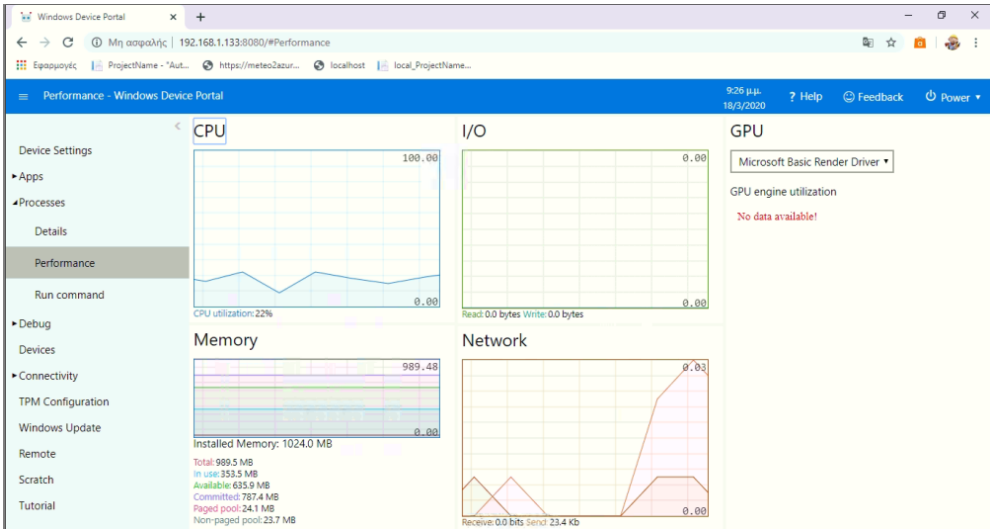
Εικόνα 24 Device Settings



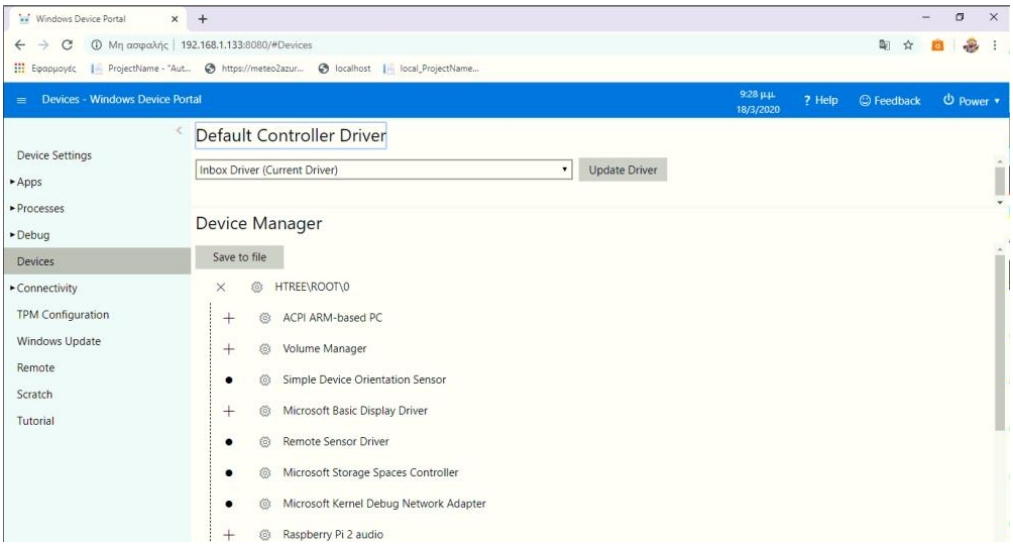
Εικόνα 25 Apps Manager



Εικόνα 26 Running Processes



Εικόνα 27 Performance



Εικόνα 28 Device Panel

ΚΕΦΑΛΑΙΟ 5 - Azure

5.1 Azure account

Το Microsoft Azure είναι η cloud πλατφόρμα της Microsoft. Υποστηρίζει πάρα πολλές υπηρεσίες, με πάρα πολλές δυνατότητες σε διάφορα abstraction levels. Μπορεί δηλαδή κάποιος να δημιουργήσει ένα απλό Windows ή Linux Virtual Machine ή ακόμα και ένα πανίσχυρο VM με πολύ μνήμη πολλές CPU, και πολύ μεγάλο storage. Υπάρχουν όμως και η δυνατότητα να τρέξει κάποιος π.χ. ένα WEB Application χωρίς να χρειαστεί να δημιουργήσει VM, να εγκαταστήσει Web Server κλπ και επιπλέον να έχει και την ευθύνη για τη διαχείριση, το scaling, το performance κλπ. Σημαντικό είναι επίσης ότι οι χρεώσεις γίνονται ανάλογα με τη χρήση των resources.

Προκειμένου να δοκιμάσει κανείς τις υπηρεσίες του Azure, η Microsoft δίνει τη δυνατότητα δημιουργίας δωρεάν λογαριασμού για ένα χρόνο. Με το δωρεάν λογαριασμό παρέχονται τα παρακάτω:

- 12 μήνες δωρεάν χρήση σε δημοφιλείς υπηρεσίες.
- Δωρεάν χρήση (ακόμα και μετά το πέρας των 12 μηνών) σε 25+ υπηρεσίες.
- $200 για ένα μήνα για χρήση όλων υπηρεσιών του Azure.

Για το project χρειαζόμαστε τις υπηρεσίες **App Service** και **SQL Database**. Συγκεκριμένα, το project απαιτεί ένα API App service για την εφαρμογή που λαμβάνει τα δεδομένα από το απομακρυσμένο Raspberry PI (μετρήσεις από αισθητήρες), ένα WEB App service το οποίο θα έχει την ιστοσελίδα με την παρουσίαση των μετρήσεων και μία SQL database.

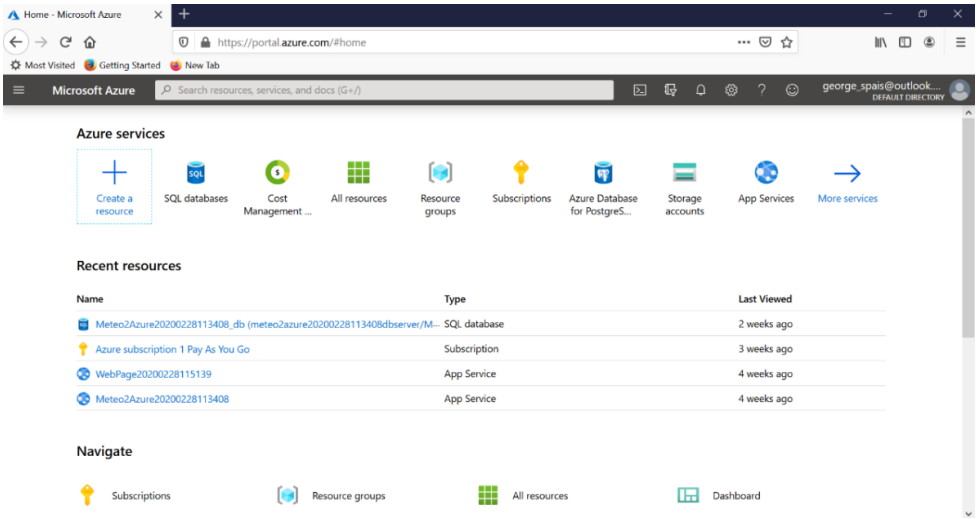
Όλες οι απαιτούμενες για το project Azure υπηρεσίες, καλύπτονται από το δωρεάν πλάνο, καθώς αυτό επιτρέπει μέχρι και 10 web, mobile, API App services (ακόμα και μετά τη λήξη του δωρεάν 12μηνου), και 250 GB SQL Database για ένα χρόνο.

Για τη δημιουργία του Azure free account, πατάμε την επιλογή *Start Free* από την ιστοσελίδα **https://azure.microsoft.com/en-us/**

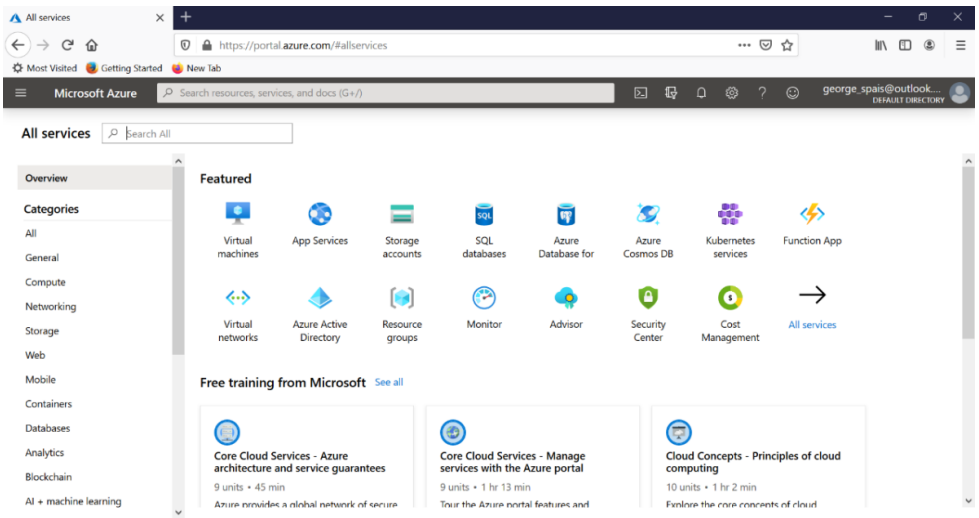
και ακολουθούμε τις οδηγίες. Απαιτείται η δήλωση πιστωτικής κάρτας κατά την εγγραφή, για την περίπτωση που χρησιμοποιήσουμε υπηρεσίες εκτός του free plan.

5.2 Είσοδος στο Azure Portal

Μόλις ολοκληρώσουμε την εγγραφή, πατάμε στην προηγούμενη ιστοσελίδα *Sign in* και αφού συνδεθούμε, πατάμε πάνω δεξιά στο *Portal*. Τώρα βρισκόμαστε στην αρχική σελίδα του Azure Portal (Εικόνα 29), και πατάμε στο βέλος *More Services* αν θέλουμε να δούμε τις υπηρεσίες που διαθέτει το Azure (Εικόνα 30):



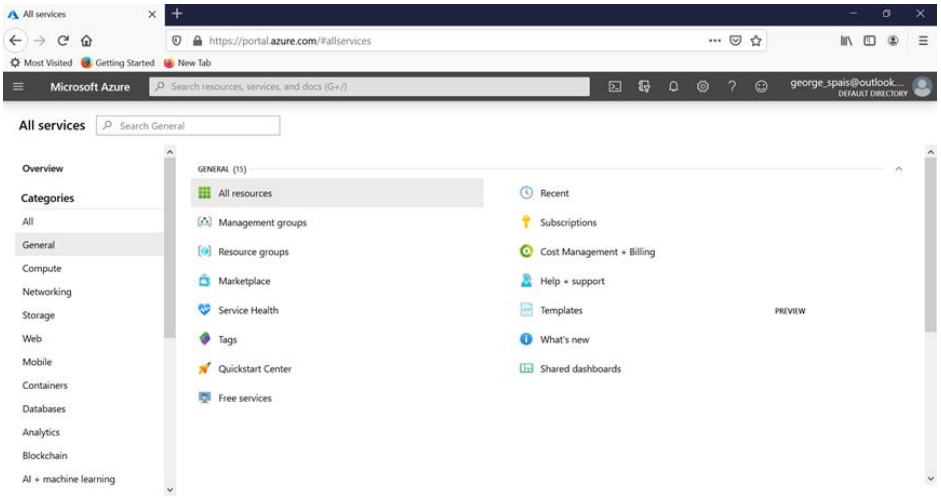
Εικόνα 29 Asure Portal First Page



Εικόνα 30 Azure Portal Services

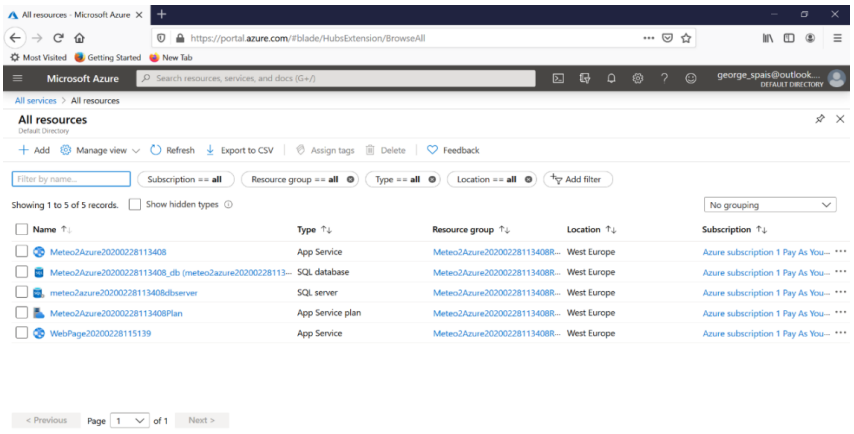
Στην παραπάνω οθόνη βλέπουμε αριστερά τις κατηγορίες των υπηρεσιών του Azure και στην κύρια οθόνη στο κέντρο αναλυτικά τις υπηρεσίες την κατηγορίας που επιλέξαμε.

Για παράδειγμα αν επιλέξουμε την κατηγορία **General** (Εικόνα 31), θα δούμε κάποιες επιλογές που αφορούν το account μας.



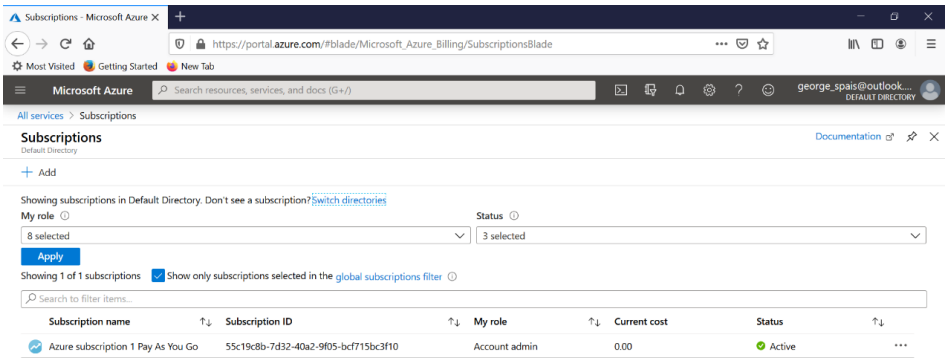
Εικόνα 31 Azure Portal Account Services

Για παράδειγμα, στην Επιλογή **All resources** (Εικόνα 32) μπορούμε να δούμε τα resources (App services, databases, storage κλπ) που έχουμε ενεργοποιήσει και χρησιμοποιούμε.



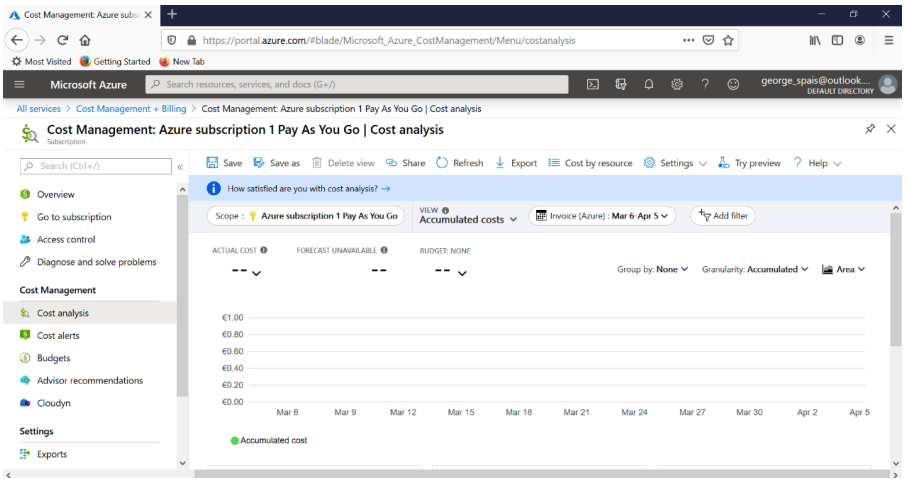
Εικόνα 32 Azure Portal All Resources

Στην επιλογή **Subscriptions** (Εικόνα 33) βλέπουμε πληροφορίες για το subscription που έχουμε στο Azure. Τον πρώτο μήνα του free account, το Azure subscription είναι τύπου *Free*. Μόλις τελειώσει ο πρώτος μήνας αλλάζει αυτόματα σε τύπο *Pay As You Go*. Επίσης εδώ βλέπουμε με μία ματιά την τρέχουσα χρέωσή μας στο Azure.

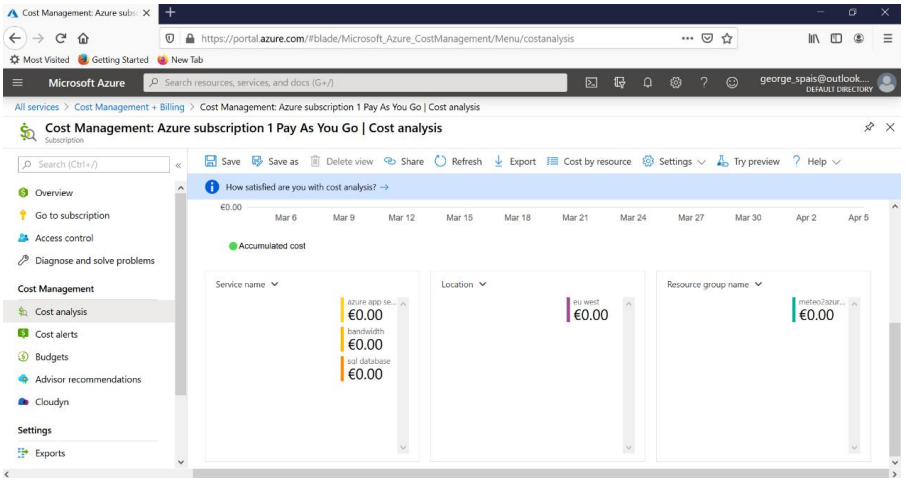


Εικόνα 33 Azure Portal Subscriptions

Μία άλλη πολύ χρήσιμη επιλογή της κατηγορίας General, είναι το **Cost Management** (Εικόνες 34 και 35). Εδώ μπορούμε να δούμε αναλυτικά τις χρεώσεις, ανά μήνα, ανά περίοδο, ανά υπηρεσία κλπ. Επίσης μας ενημερώνει για τις τρέχουσες χρεώσεις, καθώς και για εκτιμώμενη χρέωση μέχρι το τέλος του μήνα. Επίσης η δυνατότητα alert όταν οι χρεώσεις ξεπεράσουν κάποιο όριο που θα θέσουμε εμείς.



Εικόνα 34 Azure Portal Cost Management



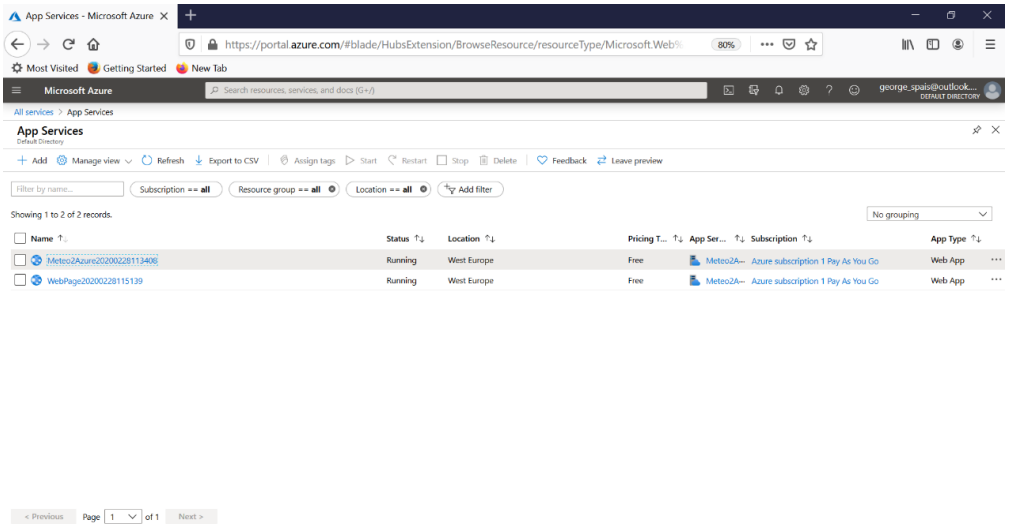
Εικόνα 35 Azure Portal Cost Analysis

5.3 App service management

Στην κατηγορία **Web** υπάρχει η επιλογή **App Services** (Εικόνα 36). Από την επιλογή αυτή μπορούμε να δούμε τα services που έχουμε εγκαταστήσει και να τα διαχειριστούμε. Παρέχονται πάρα πολλές δυνατότητες που αφορούν τα services, όπως μελέτη performance, access control, security, activity log κλπ. Μπορούμε επίσης να σταματήσουμε την εκτέλεση ενός service, να το κάνουμε reset, ή να το διαγράψουμε οριστικά.

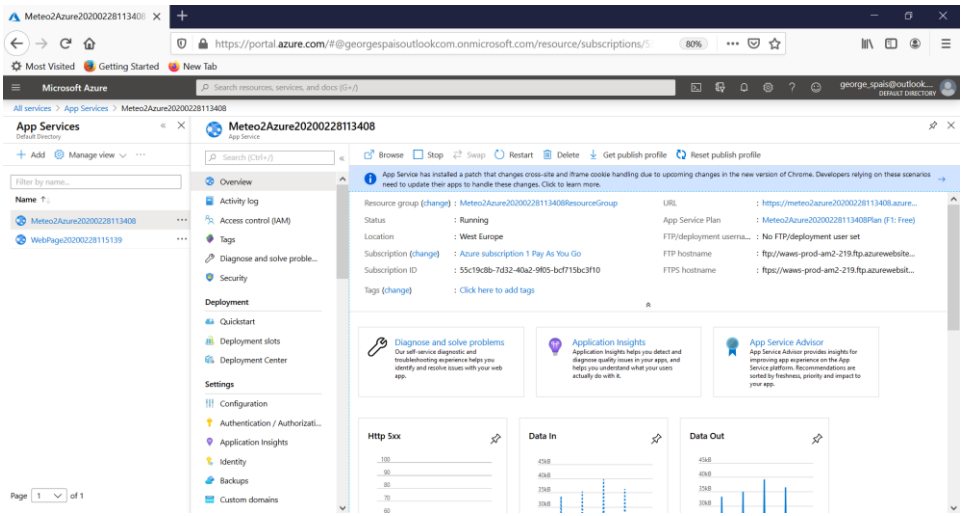
Επιλέγοντας λοιπόν την επιλογή App Services (Εικόνα 36), βλέπουμε τα services που έχουμε εγκαταστήσει και εκτελούνται στο Azure. Αυτή τη στιγμή έχουμε δύο App Services:

- Το Meteo2Azure20200228113408 που είναι το API service το οποίο δέχεται τα δεδομένα (μετρήσεις) από το Raspberry και τα αποθηκεύει σε μία βάση δεδομένων, και
- Το WebPage20200228115139 που είναι το WEB service το οποίο διαβάζει τα δεδομένα από τη βάση, δημιουργεί την ιστοσελίδα και τη στέλνει στον browser

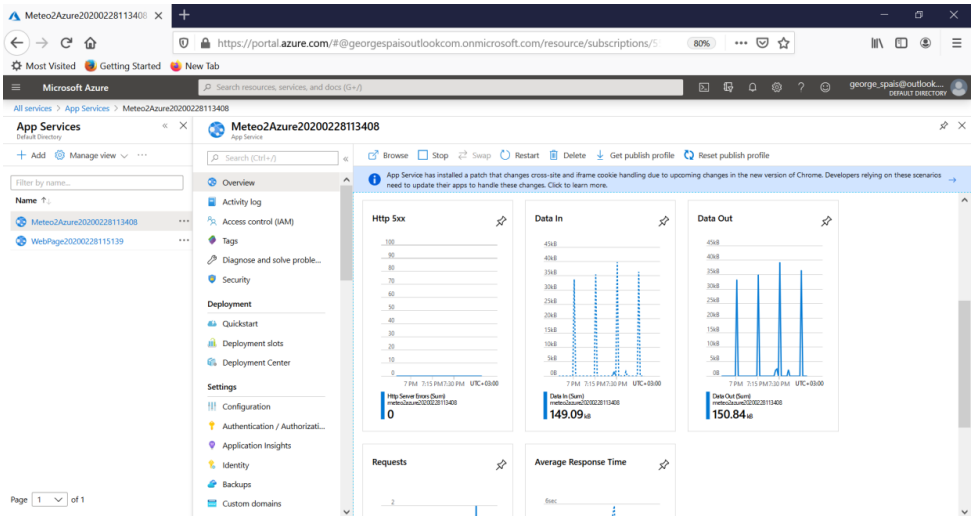


Εικόνα 36 Azure Portal App Services

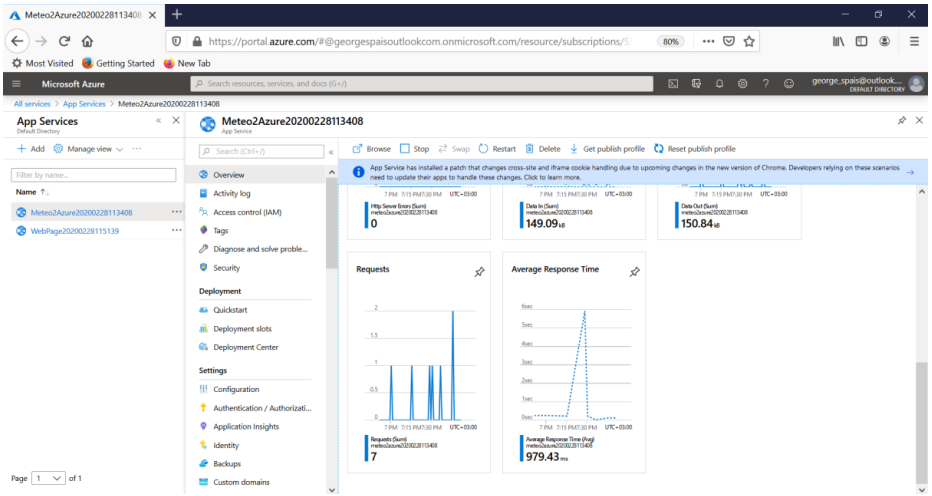
Επιλέγουμε τον API service *(Meteo2Azure20200228113408)* και στην επόμενη οθόνη βλέπουμε διάφορες πληροφορίες (Εικόνα 37), όπως το Status του service, το Location, το subscription plan που ανήκει, το URL και όπως θα δούμε στις επόμενες εικόνες και μερικά performance γραφήματα, όπως Data In/Out, HTTP errors (Εικόνα 38), number of requests και Average Response Time (Εικόνα 39). Υπάρχουν επίσης και οι επιλογές Stop, Restart, Delete κλπ.



Εικόνα 37 Azure Portal Meteo2Azure App Services Overview 1



Εικόνα 38 Azure Portal Meteo2Azure App Services Overview 2



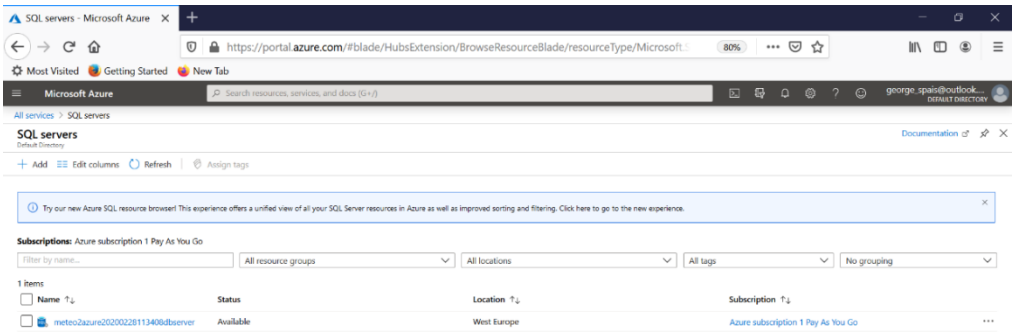
Εικόνα 39 Azure Portal Meteo2Azure App Services Overview 3

Τις ίδιες πληροφορίες θα πάρουμε αν επιλέξουμε να δούμε και το WEB App.

5.4 SQL Server management

Από την κατηγορία **Databases** μπορούμε να δούμε και διαχειριστούμε τις database που έχουμε ενεργοποιήσει στο Azure.

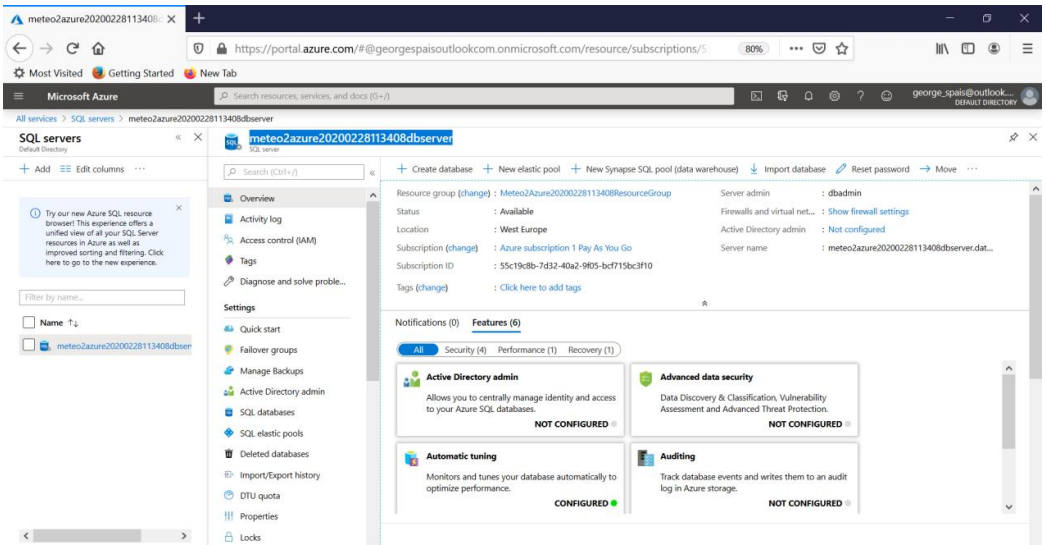
Αρχικά επιλέγουμε, κατηγορία Databases και μετά **SQL Servers** (Εικόνα 40) και βλέπουμε τον SQL Server *(meteo2azure20200228113408dbserver)* που έχουμε εγκαταστήσει για τις ανάγκες του project.



Εικόνα 40 Azure Portal SQL Servers 1

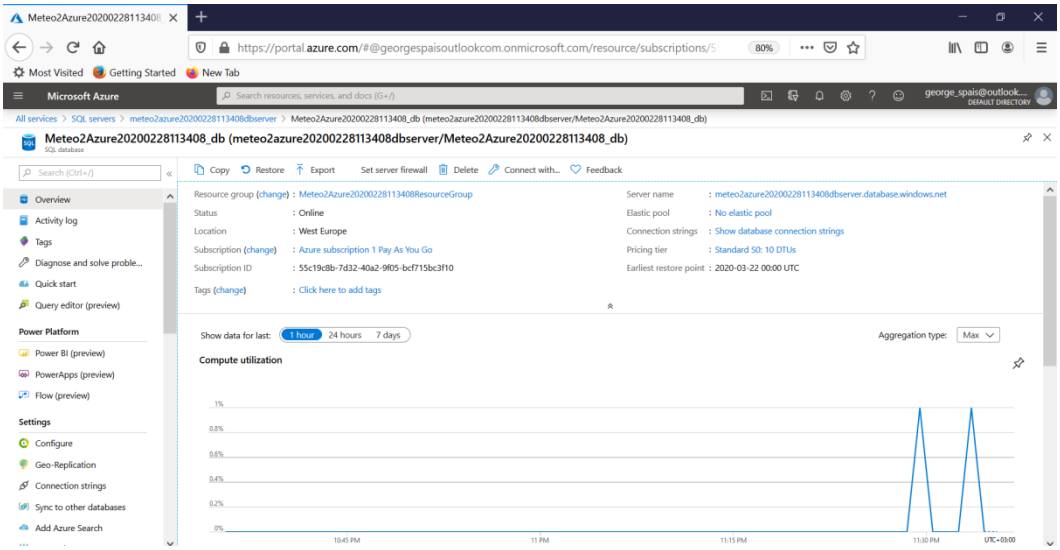
Την Εικόνα 41 βλέπουμε όταν επιλέξουμε το:

meteo2azure20200228113408dbserver

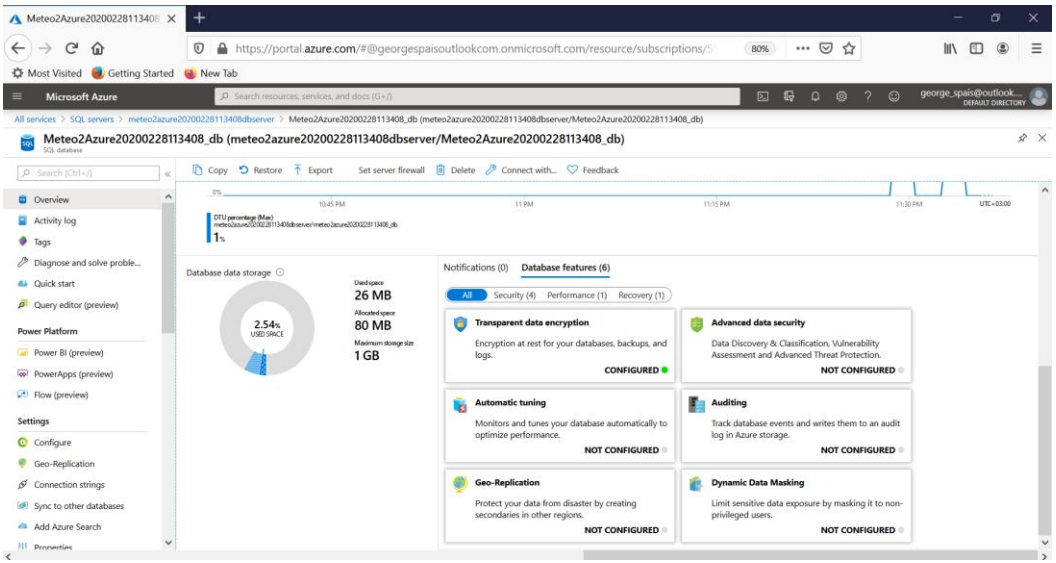


Εικόνα 41 Azure Portal SQL Servers 1

Τις Εικόνες 42 και 43 βλέπουμε όταν επιλέξουμε την **SQL database** *meteo2azure20200228113408dbserver*



Εικόνα 42 Azure Portal SQL Server Overview 1

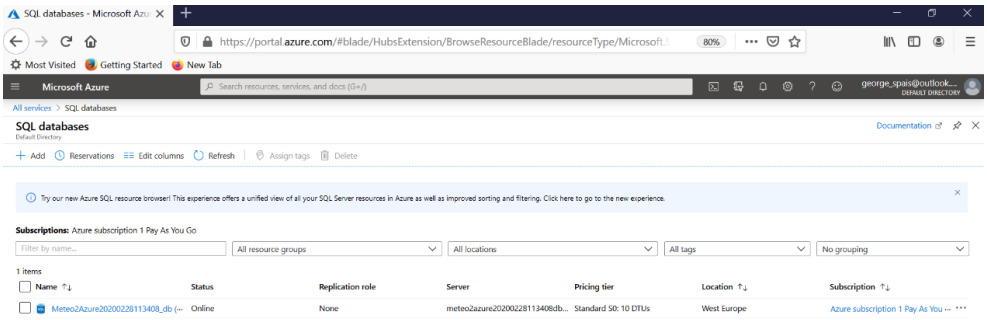


Εικόνα 43 Azure Portal SQL Server Overview 2

Μπορούμε να δούμε το utilization της βάσης, το storage της βάσης, διάφορες πληροφορίες του SQL Server και κάνουμε διάφορες ενέργειες διαχείρισης.

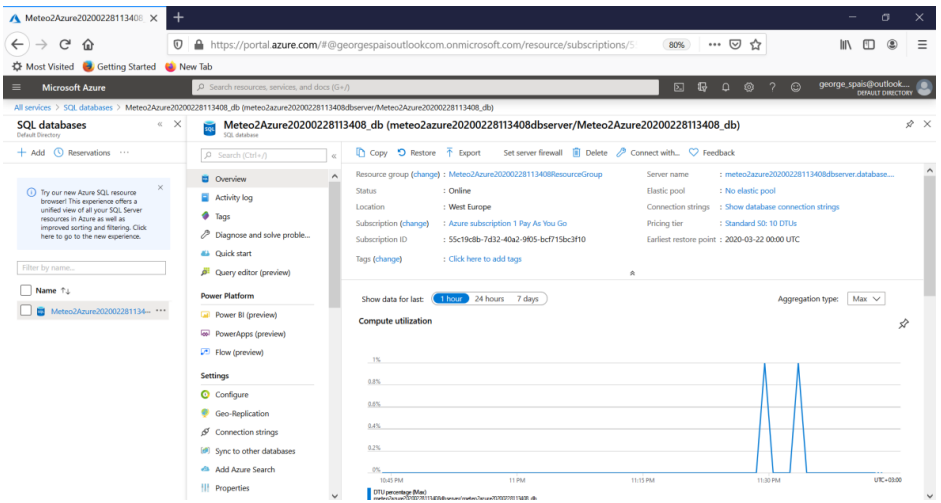
5.5 SQL Database management

Αρχικά επιλέγουμε, κατηγορία Databases και μετά SQL Database (Εικόνα 44) και βλέπουμε την SQL Database *(Meteo2Azure20200228113408_db)* που έχουμε δημιουργήσει για τις ανάγκες του project.

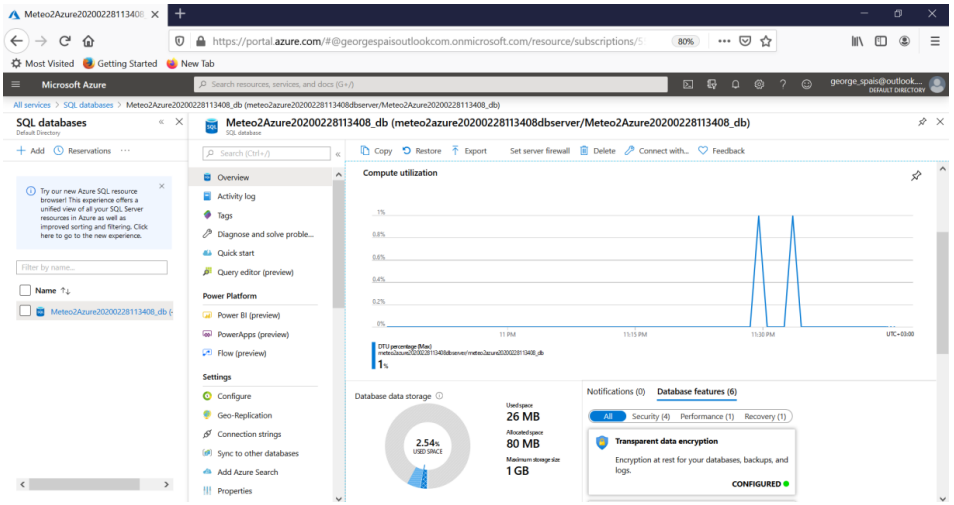


Εικόνα 44 Azure Portal Sql Databases

Την Εικόνα 45 βλέπουμε όταν επιλέξουμε το SQL Database *(Meteo2Azure20200228113408_db)*



Εικόνα 45 Azure Portal Meteo2Azure Server



Εικόνα 46 Azure Portal Meteo2Azure Overview

Στην παραπάνω Εικόνα (Εικόνα 46), μας δίνεται η δυνατότητα με γραφήματα να ελέγξουμε την κίνηση στη βάση, καθώς επίσης και τη χωρητικότητά της. Υπάρχουν επίσης και διάφορες πληροφορίες, όπως το όνομα του Server που τη φιλοξενεί, το subscription plan κλπ. Δεν μπορούμε όμως να δούμε τη δομή της και τα περιεχόμενά της. Για να το δούμε αυτό, χρειαζόμαστε εργαλεία διαχείρισης dataset όπως το SQL Server Management Studio (SSMS). Μέσα από SSMS θα δώσουμε τη διεύθυνση του SQL Server και θα συνδεθούμε με τα credentials που ορίσαμε κατά τη δημιουργία του SQL Server και της βάσης.

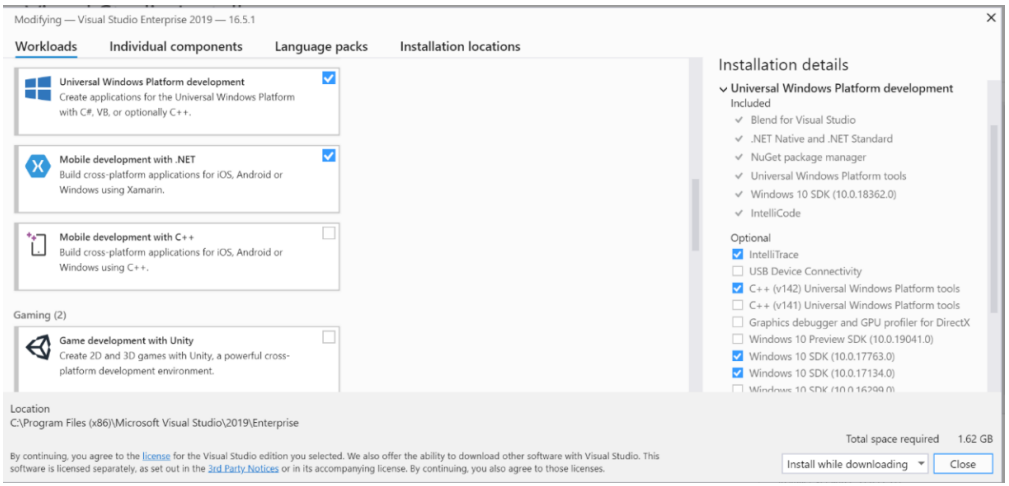
Κεφάλαιο 6 - Εφαρμογή Raspberry

Η εφαρμογή που εκτελείται τοπικά στο Raspberry είναι υπεύθυνη για την επικοινωνία του Raspberry με τους αισθητήρες, τη συλλογή των μετρήσεων και την αποστολή τους στην εφαρμογή ASP .NET API που εκτελείται στο Azure.

6.1 Απαραίτητα SDK's και Add-ons για Windows 10 IoT applications

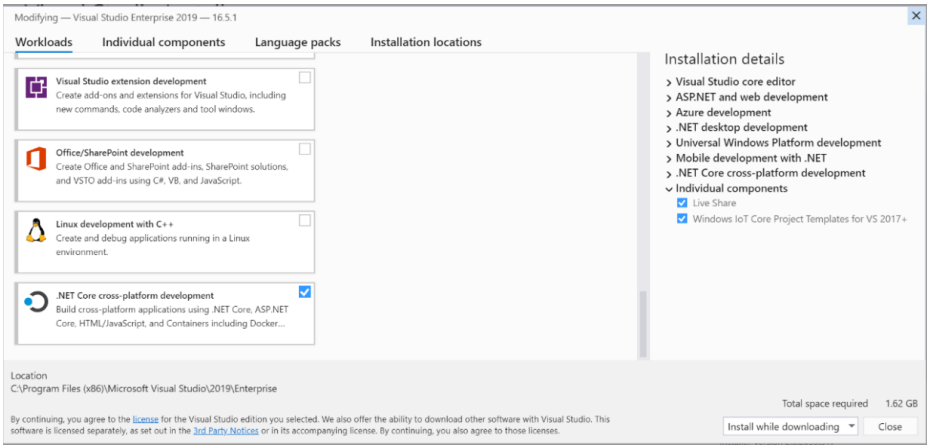
Οι εφαρμογές που εκτελούνται σε Raspberry PI με λειτουργικό σύστημα Windows 10 IoT Core, είναι εφαρμογές UWP (Universal Windows Platform). Για να δημιουργήσουμε μία εφαρμογή UWP στο Visual Studio 2019, θα πρέπει πρώτα από το Visual Studio 2019 Installer να εγκαταστήσουμε το Windows 10 SDK, και επιπλέον κάποια πρόσθετα templates για Windows 10 IoT Core projects.

Στο Visual Studio 2019 Installer επιλέγουμε στα Workloads, *Universal Windows Platform development*, και στα installation details ότι φαίνεται στην παρακάτω εικόνα (Εικόνα 47).



Εικόνα 47 SDK's

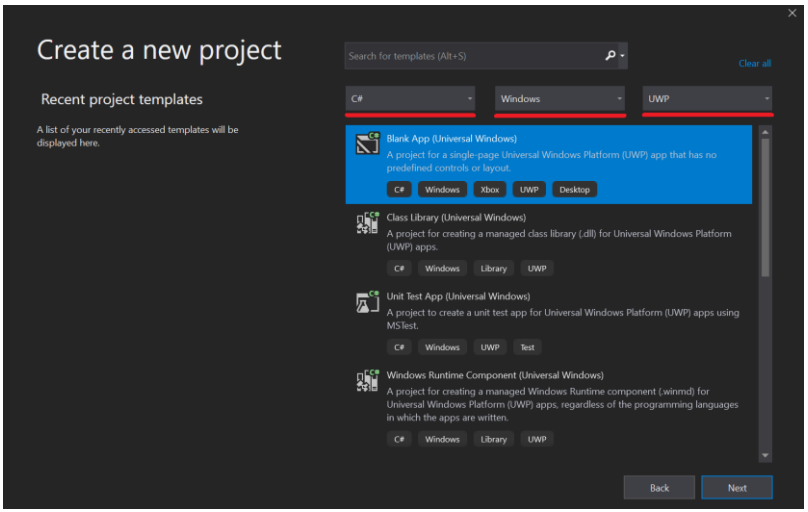
Επίσης στα Installation details, στα Individual components, επιλέγουμε *Windows IoT Core Project Templates for VS 2017+ (Εικόνα 48).*



Εικόνα 48 SDK's

6.2 Δημιουργία project στο Visual Studio 2019

Ξεκινάμε το Visual Studio 2019 και επιλέγουμε **Create New Project**. Επιλέγουμε γλώσσα **C#**, platform **Windows** και project type **UWP** (όπως δείχνουν οι κόκκινες γραμμές στην παρακάτω εικόνα 49). Τέλος, επιλέγουμε **Blank App (Universal Windows)** και *Next*.

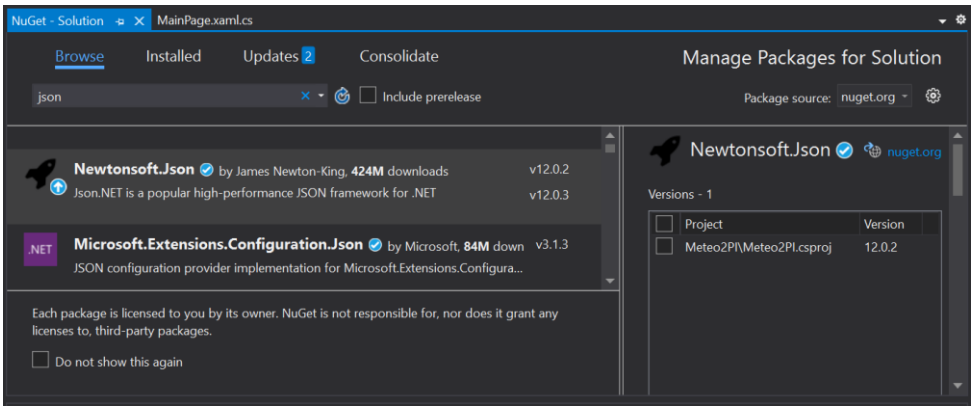


Εικόνα 49 Visual Studio Create project

Στη συνέχεια δίνουμε το όνομα του project και το path του. Πατάμε *Next* και μετά το VS μας ρωτάει ποιες εκδόσεις των Windows 10 θέλουμε να κάνει target η εφαρμογή μας. Επιλέγουμε version 1903 και minimum version 17763.

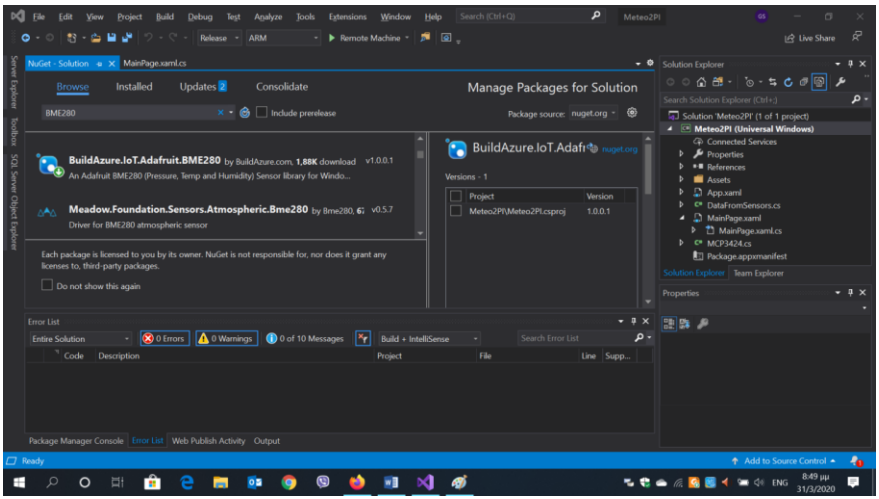
Μόλις δημιουργηθεί το project, επιλέγουμε από το menu: *Tools->NuGet Package Manager->Manage NuGet Packages for Solution*.

Επιλέγουμε Browse και αναζητούμε το package **Newtonsoft.Json** (Εικόνα 50). Αυτό είναι μία πολύ διαδεδομένη βιβλιοθήκη για JSON. Επειδή η Microsoft στη .NET δεν υποστηρίζει JSON, η βιβλιοθήκη από τη Newtonsoft είναι απαραίτητη. Οπότε επιλέγουμε και εγκαθιστούμε το Newtosoft.Json.



Εικόνα 50 Visual Studio Manage Solution Packages

Όταν ολοκληρωθεί η εγκατάσταση του Newtonsoft.Json, αναζητούμε το **BuildAzure.IoT.Adafruit.BME280** (Εικόνα 51)



Εικόνα 51 Visual Studio Manage Solution Packages

Αυτό το package είναι μία βιβλιοθήκη για τον αισθητήρα θερμοκρασίας-υγρασίας-πίεσης BME280 της Bosch η οποία είναι βασισμένη στη βιβλιοθήκη που συστήνει η Bosch, και υπάρχει και στο GitHub.

6.3 MainPage.xaml.cs

Παρακάτω φαίνονται ορισμένα namespaces που πρέπει να δηλωθούν στο MainPage.xaml.cs:

```
using Windows.Devices.Gpio;                    πρόσβαση στις GPIO του Raspberry


using System.Diagnostics;                      περιέχει τη Debug.WriteLine
using BuildAzure.IoT.Adafruit.BME280; πρόσβαση στον BME280 αισθητήρα
using Newtonsoft.Json;                         πρόσβαση στη JSON βιβλιοθήκη
using System.Net.Http;                         methods για χρήση HTTP protocol
using System.Net.Http.Headers;                 methods για χρήση HTTP protocol
using Windows.Media.Capture;                   methods για χρήση της camera
using Windows.Media.MediaProperties;           methods για χρήση της camera
using Windows.Storage;                         methods για local storage στο PI
using System.Threading.Tasks;                  Task library
using Windows.Storage.Streams;                 methods για local storage στο PI
```

Στη συνέχεια, στη MainPage class δηλώνουμε μερικά objects για τη λειτουργία της camera (MediaCapture), του GpioController, των GpioPin, του αισθητήρα θερμοκρασίας-υγρασίας-πίεσης BME280, του ADC (MCP3424), τον timer σύμφωνα με τον οποίο θα παίρνονται μετρήσεις, καθώς και μερικές variables για τη μέτρηση της ταχύτητας του αέρα, το ύψος βροχής κλπ. Στη μεταβλητή *interval* δηλώνουμε κάθε πότε θα λειτουργεί ο timer. Εδώ έχουμε δηλώσει κάθε 15 λεπτά. Η class MCP3424 δεν ανήκει σε κάποια βιβλιοθήκη, είναι δική μας και θα αναλυθεί αργότερα.

```
MediaCapture captureManager;
GpioController gpiocontroller;
GpioPin windspeedPin;
GpioPin rainPin;
BME280Sensor sensor1;
MCP3424 ADC;
DispatcherTimer timer1;

string i2cAqs = null;
int WindSpeedCount = 0;
int RainCount = 0;
```

```
int interval = 15; //15 min interal //30000 msec;  // 30 sec
int windspeedBF;
double windspeedKMH;
double voltage;
double rainheight;
```

6.3.1 Method OnNavigatedTo()

Η *OnNavigatedTo()* είναι μία method των Windows, η οποία εκτελείται όταν φορτώνεται η βασική Page της εφαρμογής μας. Την κάνουμε override και προσθέτουμε δικό μας κώδικα, καθώς θέλουμε μόλις ξεκινάει η εφαρμογή να γίνονται κάποια initializations σε devices και να ξεκινάει ο timer. Γενικά η OnNavigatedTo είναι το κατάλληλο μέρος για εργασίες που θέλουμε να γίνονται κατά την εκκίνηση μίας εφαρμογής. Επίσης τη δηλώνουμε σαν **async**, καθώς περιέχει κλήσεις σε άλλες async methods.

```
protected override async void OnNavigatedTo(NavigationEventArgs e)
{
    base.OnNavigatedTo(e);

    captureManager = new MediaCapture();
    await captureManager.InitializeAsync();

    // camera will NOT capture without Preview !!!!!
    var captureElement = new CaptureElement();
    captureElement.Source = captureManager;
    await captureManager.StartPreviewAsync();

    gpiocontroller = GpioController.GetDefault();

    sensor1 = new BME280Sensor();
    await sensor1.Initialize();

    ADC = new MCP3424();
    await ADC.Initalize();
    ADC.WriteADC();
    ADC.ReadADC();

    timer1 = new DispatcherTimer();
    //timer1.Interval = TimeSpan.FromMilliseconds(interval);
    timer1.Interval = TimeSpan.FromMinutes(interval);
    timer1.Tick += Timer1_Tick;

    SetupWindSpeedMeasurement();
    SetupRainMeasurement();

    timer1.Start();
```

```
}
```

Οι classes MediaCapture και CaptureElement παρέχουν λειτουργικότητα για τη λήψη φωτογραφιών, ήχου και video από την camera. Αρχικά δημιουργούμε ένα MediaCapture object, το κάνουμε initialize, δημιουργούμε ένα CaptureElement object, το κάνουμε bind με το CaptureManager ξεκινάμε preview. Παρόλο που εμείς δε χρειαζόμαστε camera preview στην εφαρμογή μας, πρέπει υποχρεωτικά να δημιουργήσουμε ένα preview, διαφορετικά η κάμερα δε θα κάνει capture! Αυτό ήταν ένα πρόβλημα που μας απασχόλησε αρκετά, μέχρι να βρούμε αυτή τη πληροφορία.

Στη συνέχεια γίνεται initialize ο GPIO Controller, ο BME280 και το MCP3424. Η Initialize method του BME280 ανήκει στη βιβλιοθήκη, ενώ η Initalize method του MCP3424 είναι δική μας και ανήκει σε δική μας βιβλιοθήκη και θα αναλυθεί αργότερα στο αντίστοιχο κεφάλαιο. Στη συνέχεια γίνεται ένα write και ένα read στο ADC, με σκοπό να προγραμματιστεί το mode λειτουργίας του.

Στη συνέχεια δημιουργούμε τον timer, ο οποίος θα λειτουργεί σε διαστήματα των 15 λεπτών. Timer1_Tick είναι η συνάρτηση που θα εκτελείται σε κάθε tick του timer (κάθε 15 λεπτά).

Τέλος, γίνεται setup οι ρυθμίσεις για τη μέτρηση ταχύτητας αέρα και ύψους βροχής με τις αντίστοιχες methods (που θα αναλυθούν αργότερα) και ξεκινάει ο timer με την εντολή timer1.Start().

6.3.2 SetupWindSpeedMeasurement()

Η method αυτή κάνει τις απαραίτητες ρυθμίσεις για τη μέτρηση της ταχύτητας του ανέμου.

```
private void SetupWindSpeedMeasurement()
{
   windspeedPin = gpiocontroller.OpenPin(18);
   windspeedPin.SetDriveMode(GpioPinDriveMode.InputPullUp);
   windspeedPin.ValueChanged += WindspeedPin_ValueChanged;
}
```

Με την method *OpenPin(18)*, «ανοίγουμε» το GPIO pin 18 για χρήση από εμάς. Με την *SetDriveMode()* το ρυθμίζουμε σαν Input με χρήση εσωτερικής αντίστασης PullUp, δίνοντας σαν παράμετρο *GpioPinDriveMode.InputPullUp*.

Το ValueChanged event ενεργοποιείται κάθε φορά που υπάρχει αλλαγή στην τιμή (κατάσταση) του pin. Δηλαδή, κάθε φορά που του pin γίνεται από High->Low είτε από

Low->High, το event ενεργοποιείται και θα εκτελεστεί ο event handler *WindSpeedPin_ValueChanged*, ο οποίος θα κάνει τις απαραίτητες ενέργειες για να γίνει η μέτρηση.

6.3.3. WindspeedPin_ValueChanged()

Αυτός ο event handler, απλώς θα αυξήσει έναν μετρητή, που μετράει τις αλλαγές στην κατάσταση του pin. Αυτό είναι χρήσιμο όπως αναλύθηκε νωρίτερα στην περιγραφή του αισθητήρα ταχύτητας αέρα, για τον υπολογισμό της ταχύτητας.

```
private void WindspeedPin_ValueChanged(GpioPin sender,
                                       GpioPinValueChangedEventArgs args)
{
  WindSpeedCount++;
}
```

6.3.4 SetupRainMeasurement()

Η method αυτή κάνει τις απαραίτητες ρυθμίσεις για τη μέτρηση του ύψους βροχής.

```
private void SetupRainMeasurement()
{
  rainPin = gpiocontroller.OpenPin(23);
  rainPin.SetDriveMode(GpioPinDriveMode.InputPullUp);
  rainPin.ValueChanged += RainPin_ValueChanged;
}
```

Με την method OpenPin(23), «ανοίγουμε» το GPIO pin 23 για χρήση από εμάς. Με την SetDriveMode() το ρυθμίζουμε σαν Input με χρήση εσωτερικής αντίστασης PullUp, δίνοντας σαν παράμετρο GpioPinDriveMode.InputPullUp.

Το ValueChanged event ενεργοποιείται κάθε φορά που υπάρχει αλλαγή στην τιμή (κατάσταση) του pin. Δηλαδή, κάθε φορά που του pin γίνεται από High->Low είτε από Low->High, το event ενεργοποιείται και θα εκτελεστεί ο event handler RainPin_ValueChanged, ο οποίος θα κάνει τις απαραίτητες ενέργειες για να γίνει η μέτρηση.

6.3.5 RainPin_ValueChanged()

Η λειτουργία του event handler του αισθητήρα ύψους βροχής είναι παρόμοια με του αισθητήρα ταχύτητας αέρα.

Η διαφορά τους είναι ότι επειδή, όπως αναλύθηκε νωρίτερα στη περιγραφή του αισθητήρα ύψους βροχής, μας ενδιαφέρει μόνο η μετάβαση από High->Low, πρέπει να μετράμε μόνο τα falling edges. Τα event arguments, μας δίνουν αυτή τη πολύτιμη πληροφορία μέση της property *Edge*. Επομένως ελέγχουμε απλώς αν το Edge της αλλαγής που συνέβη στο pin είναι *GpioPinEdge.FallingEdge* και μόνο τότε αυξάνουμε τον μετρητή.

```
private void RainPin_ValueChanged(GpioPin sender,
                                  GpioPinValueChangedEventArgs args)
{
  if (args.Edge == GpioPinEdge.FallingEdge)
    RainCount++;
}
```

6.3.6 Timer1_Tick()

Αυτή η method είναι ο event handler του timer. Εκτελείται κάθε 15 λεπτά , που είναι το interval που έχουμε δηλώσει, παίρνει τις μετρήσεις από όλους τους αισθητήρες, κάνει capture μία φωτογραφία από την camera, ομαδοποιεί όλα αυτά τα data σε ένα object, και τα στέλνει στο API service που τρέχει στο Azure όπου εκεί αποθηκεύονται στη database.

```
private async void Timer1_Tick(object sender, object e)
{
  windspeedKMH = GetWindSpeedKMH();
  windspeedBF = GetWindSpeedBF(windspeedKMH);
  rainheight = GetRainHeight();

  ADC.WriteADC();
  ADC.ReadADC();
  voltage = ADC.GetVoltage();
  ADC.GetWindDirection(voltage);
  string direction = ADC.direction;
  double degrees = ADC.degress;

  Single temp = await sensor1.ReadTemperature();
  Single humidity = await sensor1.ReadHumidity();
  Single pressure = await sensor1.ReadPressure() / 100;

  ImageEncodingProperties imgFormat = ImageEncodingProperties.CreateJpeg();
  // create storage file in local app storage
```

```
        var myPictures = await
Windows.Storage.StorageLibrary.GetLibraryAsync(Windows.Storage.KnownLibraryId.Pictures);
        StorageFile file = await myPictures.SaveFolder.CreateFileAsync("photo.jpg",
CreationCollisionOption.ReplaceExisting);

        // take photo to file
        await captureManager.CapturePhotoToStorageFileAsync(imgFormat, file);

        // take photo to stream and convert it to Base64 for JSON serialization
        MemoryStream ms = new MemoryStream();
        IRandomAccessStream ras = ms.AsRandomAccessStream();
        await captureManager.CapturePhotoToStreamAsync(imgFormat, ras);
        byte[] imageBytes = ms.ToArray();
        string base64String = Convert.ToBase64String(imageBytes);

        DataFromSensors data = new DataFromSensors();
        data.DateTime = DateTime.Now.ToString("dd/MM/yyyy HH:mm:ss");
        data.Temperature = Math.Round(temp, 1).ToString();
        data.Humidity = Math.Round(humidity).ToString();
        data.AtmPressure = Math.Round(pressure).ToString();
        data.WindSpeedKMH = Math.Round(windspeedKMH, 1).ToString();
        data.WindSpeedBF = windspeedBF.ToString();
        data.RainMM = rainheight.ToString();
        data.WindDirection = direction;
        data.WindDirectionDegrees = degrees.ToString();
        data.RawPhotoBase64 = base64String;

        await SendDataToAzure(data);

        Debug.WriteLine("{0} wind ticks {1} km/h {2} BF {3} rain ticks {4} mm {5} V {6} {7} deg", WindSpeedCount,
windspeedKMH, windspeedBF, RainCount, rainheight, voltage, direction, degrees);
        Debug.WriteLine("{0} C {1} % {2} mbar", temp, humidity, pressure);

        WindSpeedCount = 0;
        RainCount = 0;
    }
```

Αρχικά καλούνται οι *GetRainHeight(), GetWindSpeedKMH()* και *GetWindSpeedBF()* οι οποίες θα πάρουν μέτρηση ύψους βροχής και ταχύτητας αέρα σε km/h και σε BF αντίστοιχα. Όλες θα αναλυθούν αργότερα.

Στη συνέχεια γίνονται ένα Write και ένα Read στο ADC για να πάρουμε μέτρηση τάσης από τον αισθητήρα κατεύθυνσης ανέμου. Ακολουθεί μία κλήση στην *GetVoltage()* η οποία υπολογίζει την πραγματική τάση από την τιμή μέτρησης του ADC, όπως αναλύθηκε νωρίτερα στο κεφάλαιο για το MCP3424. Η *GetWindDorection()* παίρνει σαν παράμετρο την τάση που μετρήσαμε, και υπολογίζει την κατεύθυνση του αέρα σε μοίρες αλλά και περιγραφικά (π.χ. N, NW, κλπ). Όλες οι παραπάνω methods θα αναλυθούν στη συνέχεια.

Αμέσως μετά καλούνται οι methods *ReadTemperature()*, *ReadHumidity()* και *ReadPressure()* οι οποίες μας δίνουν τις μετρήσεις θερμοκρασίας, υγρασίας και πίεσης από το BME280. Επειδή η μέτρηση του αισθητήρα για την πίεση είναι σε Pascal και επειδή εμείς τη θέλαμε σε mbar που είναι πιο κοινή μονάδα μέτρησης στους μετεωρολογικούς σταθμούς, το αποτέλεσμα της ReadPressure το κάνουμε /100. Και οι τρεις methods είναι ασύγχρονες (οπότε καλούνται με await) και ανήκουν στη βιβλιοθήκη του BME280 που εγκαταστήσαμε ως nuget package.

Στη συνέχεια ακολουθούν μερικές γραμμές κώδικα που αφορούν το image capture από την camera. Με το *ImageEncodingProperties* object δηλώνουμε το format που θέλουμε να έχει η εικόνα που θα κάνει capture η camera. Επιλέγουμε jpeg καλώντας *ImageEncodingProperties.CreateJpeg()*.

Αμέσως μετά με την κλήση της method *GetLibraryAsync()* δηλώνουμε το path που θα αποθηκευτεί η εικόνα που θα κάνει capture η camera. Ορίζουμε με την παράμετρο *Windows.Storage.KnownLibraryId.Pictures* σαν φάκελο αποθήκευσης το Pictures special folder. Να σημειωθεί εδώ, ότι τα Windows IoT Core δεν είναι φιλικό λειτουργικό ως προς την τοπική αποθήκευση αρχείων. Επιτρέπεται η αποθήκευση αρχείων τοπικά στο Raspberry μόνο σε λίγα και συγκεκριμένα folders. Δεν υπάρχει Save και Save Dialog Box και δεν μπορούμε να αποθηκεύσουμε αρχεία τοπικά όπου θέλουμε. Αφού δηλώσουμε το path αποθήκευσης, δίνουμε το όνομα το αρχείο (photo.jpg) και δηλώνουμε ότι θέλουμε κάθε νεότερο αρχείο να κάνει replace τυχών υπάρχοντα, τη χρήση της παραμέτρου *CreationCollisionOption.ReplaceExisting*.

Στη συνέχεια η camera κάνει capture μία εικόνα και την αποθηκεύει σε local file με τη κλήση στη method *CapturePhotoToStorageFileAsync()*. Η method αυτή είναι ασύγχρονη και με δύο παραμέτρους της δίνουμε το image type και το file path που θα αποθηκευτεί.

Η εικόνα αυτή που έκανε capture η camera θέλουμε να κωδικοποιηθεί κατάλληλα και να αποσταλεί και στο Azure. Επειδή αυτή η διαδικασία μετατροπής από το αρχείο είναι αρκετά πολύπλοκη, αναγκάζουμε τη camera να κάνει ένα δεύτερο capture αμέσως μετά από το πρώτο. Αυτή τη φορά χρησιμοποιούμε τη method *CapturePhotoToStreamAsync()* η οποία αντί για file θα στείλει την εικόνα σε ένα MemoryStream που έχουμε δημιουργήσει αμέσως πριν τη κλήσης της (με τις *MemoryStream* και *IRandomAccessStream*) . Η method αυτή είναι ασύγχρονη και με δύο παραμέτρους της δίνουμε το image type και το memory stream που θα αποθηκευτεί.

Με την εντολή *byte[] imageBytes = ms.ToArray()* μετατρέπουμε το memorystream που περιέχει την εικόνα, σε byte array. Στη συνέχεια αυτό το byte array το μετατρέπουμε σε ένα string κωδικοποιημένο σε κώδικα Base64, με την εντολή

string base64String = Convert.ToBase64String(imageBytes). Η κωδικοποίηση Base64 χρησιμοποιείται για τη μεταφορά εικόνων με το πρωτόκολλο HTTP. Αυτός είναι και ο λόγος που το επιλέξαμε για τη μεταφορά της εικόνας στο Azure, καθώς το πρωτόκολλο που χρησιμοποιούμε στα REST APIs είναι το HTTP.

Στο σημείο αυτό έχουμε όλα τα δεδομένα, τα οποία πρέπει να τα ομαδοποιήσουμε για να τα αποστείλουμε στο Azure. Φτιάχνουμε ένα object τύπου *DataFromSensors* και το γεμίζουμε με τις μετρήσεις μας. Η δομή της class DataFromSensors θα αναλυθεί αργότερα.

Η αποστολή του object με τα δεδομένα στο azure, γίνεται στη συνέχεια με την κλήση της method *SendDataToAzure()*, η οποία είναι ασύγχρονη και παίρνει σαν παράμετρο το object με τα δεδομένα. Η λειτουργία της θα αναλυθεί αργότερα.

Ακολουθούν δύο Debug.WriteLine εντολές, οι οποίες είναι για debugging λόγους και δεν επηρεάζουν την εκτέλεση του προγράμματος.

Στο τέλος, πριν επιστρέψει η Timer1_Tick, μηδενίζονται οι μετρητές της ταχύτητας αέρα και ύψους βροχής.

6.3.7 GetWindSpeedKMH()

Η method αυτή υπολογίζει την ταχύτητα του αέρα σε km/h βάση του μετρητή WindSpeedCount, ο οποίος όπως είδαμε αυξάνεται στον event handler του αισθητήρα ταχύτητας. Ο τύπος υπολογισμού έχει ήδη αναλυθεί στο κεφάλαιο του αισθητήρα ταχύτητας βροχής. Χρειαζόμαστε πόσα ticks ανα δευτερόλεπτο έγιναν. Και αυτά τα πολλαπλασιάζουμε επί 1.2 km/h. Το interval*60 γίνεται, διότι το interval είναι σε λεπτά, και πρέπει να μετατραπεί σε δευτερόλεπτα.

```
private double GetWindSpeedKMH()
{
  return ((double)(WindSpeedCount * 1.2)) / (double)(interval * 60);
}
```

6.3.8 GetWindSpeedBF()

Η method αυτή παίρνει σαν παράμετρο τη ταχύτητα του αέρα σε km/h που υπολογίσαμε με την GetWindSpeedKMH(), και την μετατρέπει σε BF, βάση γνωστού πίνακα αντιστοιχίας km/h σε BF.

```
private int GetWindSpeedBF(double windspeedKMH)
{
  int BF = 0;
```

```
    if (windspeedKMH <= 1)
        BF = 0;
    else if (windspeedKMH > 1 && windspeedKMH <= 5)
        BF = 1;
    else if (windspeedKMH > 5 && windspeedKMH <= 11)
        BF = 2;
    else if (windspeedKMH > 11 && windspeedKMH <= 19)
        BF = 3;
    else if (windspeedKMH > 19 && windspeedKMH <= 28)
        BF = 4;
    else if (windspeedKMH > 28 && windspeedKMH <= 38)
        BF = 5;
    else if (windspeedKMH > 38 && windspeedKMH <= 49)
        BF = 6;
    else if (windspeedKMH > 49 && windspeedKMH <= 61)
        BF = 7;
    else if (windspeedKMH > 61 && windspeedKMH <= 74)
        BF = 8;
    else if (windspeedKMH > 74 && windspeedKMH <= 88)
        BF = 9;
    else if (windspeedKMH > 88 && windspeedKMH <= 102)
        BF = 10;
    else if (windspeedKMH > 102 && windspeedKMH <= 117)
        BF = 11;
    else if (windspeedKMH > 117)
        BF = 12;

    return BF;
}
```

6.3.9 GetRainHeight()

Η method αυτή υπολογίζει το ύψος βροχής σε mm βάση των ticks του αισθητήρα, σύμφωνα με τον τύπο υπολογισμού που αναλύθηκε στο κεφάλαιο του αισθητήρα βροχής. Πολλαπλασιάζουμε δηλαδή, απλώς τα ticks επί 0.2794 που αντιστοιχούν στα mm βροχής σε κάθε tick του αισθητήρα.

```
private double GetRainHeight()
{
    return (double)(RainCount * 0.2794);
}
```

6.3.10 SendDataToAzure()

Αυτή η method, όπως δείχνει και το όνομά της, στέλνει τα δεδομένα μας στον server στο Azure. Αυτό επιτυγχάνεται με μία POST HTTP instruction, που περιέχει το DataFromSensors object με τις μετρήσεις και κωδικοποιημένη την εικόνα από την camera. Το DataFromSensors object με τα δεδομένα μας, έχει περαστεί σαν παράμετρος στην SendDataToAzure κατά την κλήση της.

```
private async Task SendDataToAzure(DataFromSensors data)
{
   HttpClient client = new HttpClient();
   client.BaseAddress = new Uri("https://meteo2azure20200228113408.azurewebsites.net");
   string json = JsonConvert.SerializeObject(data);
   StringContent content = new StringContent(json);
   content.Headers.ContentType = new MediaTypeHeaderValue("application/json");
   HttpResponseMessage response = null;

   try
   {
      response = await client.PostAsync("api/DataFromSensors", content);
   }
   catch
   {
      throw new Exception();
   }

   string response_data = await response.Content.ReadAsStringAsync();
   client.Dispose();

}
```

Δημιουργούμε ένα *HttpClient* object και με την property *BaseAddress* του δίνουμε τη διεύθυνση του server στο Azure.

Η method *JsonConvert.SerializeObject()* ανήκει στη βιβλιοθήκη της NewtonSoft που εγκαταστήσαμε αρχικά. Όπως δείχνει και το όνομά της, κάνει serialize με JSON format το object που παίρνει σαν παράμετρο και επιστρέφει ένα string, το οποίο είναι το object σε JSON format.

Στη συνέχεια η *StringContent* παρέχει το HTTP content βάση του JSON string που δημιουργήθηκε προηγουμένως, και στη συνέχεια με την *ContentType* ορίζουμε τον τύπο του content. Στη περίπτωσή μας JSON.

Το object *HttpResponeMessage* θα περιέχει στη συνέχεια το response της HTTP Post εντολής που θα στείλουμε.

Στο σημείο αυτό είμαστε έτοιμοι να στείλουμε την HTTP Post εντολή. Εκτελούμε τη method *PostAsync()*, η οποία είναι member της HttpClient, δίνοντας σαν παραμέτρους το path του URL και το HTTP content. Η method είναι ασύγχρονη, επομένως την καλούμε με *await*. Η κλήση γίνεται μέσα σε ένα *try {} catch {}* block. Όταν εκτελεστεί επιτυχώς, τα δεδομένα μας στέλνονται μέσω HTTP πρωτοκόλλου στο service που εκτελείται στο Azure, και συγκεκριμένα στη διεύθυνση:

https://meteo2azure20200228113408.azurewebsites.net/api/DataFromSensors

Τέλος, με την κλήση της *response.Content.ReadAsStringAsync()*, παίρνουμε σε string το HTTP response, και στη συνέχεια καλούμε την *Dispose()* του HttpClient object για να κλείσει το HTTP connection.

6.4 DataFromSensors class

Η DataFromSensors class περιέχει τα δεδομένα των μετρήσεων, καθώς και την εικόνα που έκανε capture η camera κωδικοποιημένη σε string κατά Base64.

```
public class DataFromSensors
{
    public string DateTime;
    public string Temperature;
    public string Humidity;
    public string AtmPressure;
    public string WindSpeedKMH;
    public string WindSpeedBF;
    public string WindDirection;
    public string WindDirectionDegrees;
    public string RainMM;
    public string RawPhotoBase64;
}
```

Όλα τα members είναι τύπου string. Πρώτο είναι το timestamp (ημερομηνία και ώρα μέτρησης, θερμοκρασία σε ^{o}C, υγρασία σε %, ατμοσφαιρική πίεση σε mbar, ταχύτητα ανέμου σε km/h, ταχύτητα ανέμου σε BF, κατεύθυνση ανέμου περιγραφικά (π.χ. N, NW κλπ), κατεύθυνση ανέμου σε μοίρες, ύψος βροχής σε mm και τέλος η εικόνα κωδικοποιημένη σε Base64.

Η class αυτή με **ακριβώς** με την ίδια μορφή πρέπει να υπάρχει και στην εφαρμογή API service που εκτελείται στο Azure, για να μπορεί να γίνει deserialize το object που ελήφθη από την παρούσα εφαρμογή στο Raspberry.

6.5 MCP3424 class

Η MCP3424 class αποτελεί τη βιβλιοθήκη που φτιάξαμε για τη λειτουργία του MCP3424 analog-to-digital converter. Περιέχει methods για το initialization του I^2C και του ADC. Επίσης περιέχει methods για την ανάγνωση και την εγγραφή στο chip, καθώς και την μετατροπή της μετρούμενης τιμής σε Volt. Τέλος περιέχει method που μετατρέπει τη τάση σε κατεύθυνση ανέμου.

```
using Windows.Devices.Enumeration;
using Windows.Devices.I2c;

  class MCP3424
  {
    private I2cDevice adc;

    private byte[] readbuffer;
    private double LSBVolt = 0.001;  // 1mV for 12bit resolution
    private double Gain = 1;

    public string direction;
    public double degress;
                ...

                ...

  }
```

Τα δύο namespaces *Windows.Devices.Enumaration* και *Windows.Devices.I2c* παρέχουν τις λειτουργίες (classed και methods) για τη διαχείριση του I^2C hardware. Το field object *I2cDevice* μας παρέχει πρόσβαση στον I2C controller του Raspberry. Ο byte array *readbuffer* περιέχει τα data που διαβάστηκαν από το ADC. Το *LSBVolt* field είναι τύπου double και περιέχει την τιμή 0.001 (1mV), η οποία αντιστοιχεί στο LSB για 12bit ανάλυση που θα χρησιμοποιήσουμε, όπως αναλύθηκε στο κεφάλαιο για το MCP3424.

Τέλος, τα δύο public fields *direction* και *degrees* θα περιέχουν τις υπολογισμένες τιμές της κατεύθυνσης του αέρα περιγραφικά και σε μοίρες. Αυτές είναι οι τιμές που επιστρέφονται από τη βιβλιοθήκη.

6.5.1 Initalize()

Η method *Initialize()* κάνει όλες τις αρχικοποιήσεις και ρυθμίσεις για το I^2C bus. Είναι ασύγχρονη και επιστρέφει Task.

```
    public async Task Initalize()
    {
```

```
    string i2cAqs = null;
    i2cAqs = I2cDevice.GetDeviceSelector();

    DeviceInformationCollection dis;
    dis = await DeviceInformation.FindAllAsync(i2cAqs);
    int I2C_DEVICE_ADDR = 0x68;

    I2cConnectionSettings settings;
    settings = new I2cConnectionSettings(I2C_DEVICE_ADDR);
    settings.BusSpeed = I2cBusSpeed.FastMode;

    adc = await I2cDevice.FromIdAsync(dis[0].Id, settings);
}
```

Η κλήση της *I2cDevice.GetDeviceSelector()* επιστρέφει ένα Advanced Query Syntax (AQS) string για όλους τους I^2C bus controllers που υπάρχουν στο σύστημα. Στη συνέχεια η κλήση της *DeviceInformation.FindAllAsync()* με παράμετρο το AQS, απαριθμεί (enumerates) τους I^2C controllers που περιέχονται στο AQS και τους αποθηκεύει στην *DeviceInformationCollection*.

Το field *I2C_DEVICE_ADDR* περιέχει σε δεκαεξαδική μορφή τη διεύθυνση I^2C του MCP3424, την οποία έχουμε επιλέξει από το DIP switch του ADC board όπως είδαμε στο κεφάλαιο του MCP3424.

Με το *I2cConnectionSettings* object δηλώνουμε διάφορα settings (όπως το bus speed) που αφορούν την επικοινωνία μέσω I^2C bus με την I^2C συσκευή συγκεκριμένου address που δίνουμε σαν παράμετρο. Το bus speed στη περίπτωσή μας το δηλώνουμε σε *Fast.Mode*.

Τέλος, με την κλήση της ασύγχρονης method *I2cDevice.FromIdAsync()* και με παραμέτρους το Id του I^2C bus του Raspberry και τα settings, επιστρέφεται ένα object τύπου *I2cDevice* το οποίο είναι ένα handle στο MCP3424 ADC. Με το object αυτό, μπορούμε να επικοινωνήσουμε με το ADC και να στείλουμε read/write εντολές.

6.5.2 ReadADC()

Με τη method αυτή μπορούμε να διαβάσουμε data από το ADC.

```
public void ReadADC()
{
    readbuffer = new byte[4];

    adc.Read(readbuffer);
}
```

Τον read buffer τον ορίζουμε μεγέθους 4 byte. Το μέγεθος αυτό προκύπτει από το datasheet που περιγράφει, όπως είδαμε στο αντίστοιχο κεφάλαιο, πόσα byte επιστρέφει το MCP3424. Ακολουθεί μία κλήση στη Read() method του I2cDevice object η οποία αναλαμβάνει να επικοινωνήσει με το ADC και επιστρέψει στον read buffer που πήρε ως παράμετρο τα data που διάβασε από το ADC. Τα data αυτά είναι η μέτρηση της τάσης που πήρε το ADC σε ψηφιακή μορφή. Η μετατροπή σε volt θα γίνει από άλλη method που θα αναλυθεί στη συνέχεια.

6.5.3 WriteADC()

Με τη method αυτή μπορούμε να γράψουμε data στο ADC. Το MCP3424 έχει μόνο έναν register όπου μπορούμε να γράψουμε. Τον configuration register. Επομένως, η WriteADC method αναφέρεται στην αποστολή και εγγραφή data στον configuration register.

```
public void WriteADC()
{
   byte[] writebuffer = new byte[1];
   writebuffer[0] = 0x80; //10000000

   adc.Write(writebuffer);
}
```

Ορίζουμε έναν write buffer μεγέθους 1 byte. Το μέγεθος αυτό προκύπτει από το datasheet του MCP3424, όπου περιγράφεται, όπως είδαμε, ότι το μέγεθος του configuration register είναι 8 bit. Γράφουμε την δεκαεξαδική τιμή $80_{(16)}$, η οποία αντιστοιχεί στη binary $\mathbf{10000000_{(2)}}$. Όπως είδαμε στο αντίστοιχο κεφάλαιο για το MCP3424, η τιμή αυτή στο configuration register του MCP3424, δίνει εντολή για μια νέα μετατροπή, από είσοδο του CH1, σε one-shot conversion mode, με 240 SPS sample rate (12bits ανάλυση) και PGA Gain x1.

Η κλήση της method *Write()* του I2cDevice object με παράμετρο τον write buffer, αναλαμβάνει την επικοινωνία με το MCP3424 μέσω I^2C bus, και την εγγραφή της τιμής στον configuration register.

6.5.4 GetVoltage()

Η method αυτή μετατρέπει τη τιμή μέτρησης από το ADC σε volt.

```
public double GetVoltage()
{
```

```
        int value = 0;

        //OR gia thn eisagvgi kai shift gia thesi gia to deytero
        value = (value | readbuffer[0]) << 8;
        value = value | readbuffer[1];

        double volts = Math.Round((double)((value * LSBVolt) / Gain), 3);

        return volts;
    }
```

Ο *readbuffer* περιέχει την τιμή μέτρησης η οποία είναι 12 bits. Επειδή ο readbuffer είναι byte array, η τιμή μέτρησης αποτελείται από 2 bytes. Θέλουμε όμως την τιμή ολόκληρη σε μία μεταβλητή και όχι μοιρασμένη σε δύο byte. Για το λόγο αυτό ορίζουμε μία μεταβλητή τύπου int που θα πάρει την τιμή.

Το πρώτο byte *readbuffer[0]* του buffer το κάνουμε OR (|) με την *value*, και μετά shift left 8 bits (<< 8) για να γίνει χώρος και για το δεύτερο byte.

Στη συνέχεια κάνουμε OR το δεύτερο byte *readbuffer[1]* του buffer με την *value* και πλέον στη μεταβλητή value έχουμε τη τιμή της μέτρησης από το ADC.

Στη συνέχεια μετατρέπουμε την τιμή μέτρησης σε volt, σύμφωνα με τον τύπο που δίνεται στο datasheet του MCP3424 όπως περιγράψαμε στο αντίστοιχο κεφάλαιο. Κάνουμε στην τιμή υπολογισμού στρογγυλοποίηση με 3 δεκαδικά ψηφία.

```
double volts = Math.Round((double)((value * LSBVolt) / Gain), 3);
```

Η τιμή αυτή σε volts επιστρέφεται από τη method.

6.5.5 GetWindDirection()

Η method *GetWindDirection()* παίρνει σαν παράμετρο την τάση σε volt που μετρήθηκε από τον αισθητήρα κατεύθυνσης ανέμου, όπως αναλύθηκε νωρίτερα.

Στο κεφάλαιο για τον αισθητήρα κατεύθυνσης ανέμου, δημιουργήσαμε έναν πίνακα αντιστοίχησης τιμών τάσεων με την κατεύθυνση ανέμου σε μοίρες. Ο πίνακας δημιουργήθηκε μετά από υπολογισμούς σύμφωνα με τον μαθηματικό τύπο του διαιρέτη τάσης. Παρόλα αυτά, έγιναν και μετρήσεις με πολύμετρο για όλες τις θέσεις του αισθητήρα κατεύθυνσης ανέμου, δηλαδή για όλες τις περιπτώσεις κατεύθυνσης ανέμου σε μοίρες. Οι τιμές μέτρησης με το πολύμετρο, επαλήθευσαν τις υπολογισμένες τιμές με πάρα πολύ μικρές αποκλίσεις της τάξης των ελάχιστων mV, κάτι το οποίο δικαιολογείται απόλυτα. Για το λόγο αυτό, η τιμή μέτρησης σε volt που λαμβάνουμε από το ADC δεν συγκρίνεται απόλυτα με τη θεωρητική τιμή υπολογισμού βάση του μαθηματικού τύπου,

αλλά με ένα στενό εύρος τιμών γύρω από τη τιμή αυτή. Αυτό το εύρος τιμών υπολογίστηκε βάση των μετρήσεων με το πολύμετρο που αναφέρθηκε νωρίτερα.

```
public void GetWindDirection(double volts)
{
  if ((0.750 <= volts) && (volts < 0.870))  //0.819
  {
    direction = "N";
    degress = 0;
  }
  if ((0.180 < volts) && (volts <= 0.225))  //0.203
  {
    direction = "NNE";
    degress = 22.5;
  }
  if ((0.225 < volts) && (volts < 0.350))  //0.250
  {
    direction = "NE";
    degress = 45;
  }
  if ((0.026 <= volts) && (volts < 0.031))  //0.029
  {
    direction = "ENE";
    degress = 67.5;
  }
  if ((0.031 <= volts) && (volts < 0.039))  //0.032
  {
    direction = "E";
    degress = 90;
  }
  if ((0 < volts) && (volts < 0.026))  //0.022
  {
    direction = "ESE";
    degress = 112.5;
  }
  if ((0.058 < volts) && (volts <= 0.085))  //0.071
  {
    direction = "SE";
    degress = 135;
  }
  if ((0.039 <= volts) && (volts <= 0.058))  //0.046
  {
    direction = "SSE";
    degress = 157.5;
  }
  if ((0.112 < volts) && (volts < 0.150))  //0.124
  {
    direction = "S";
    degress = 180;
  }
```

```
    if ((0.085 < volts) && (volts <= 0.112))  //0.100
    {
      direction = "SSW";
      degress = 202.5;
    }
    if ((0.425 < volts) && (volts < 0.500))  //0.455
    {
      direction = "SW";
      degress = 225;
    }
    if ((0.350 <= volts) && (volts <= 0.425))  //0.408
    {
      direction = "WSW";
      degress = 247.5;
    }
    if ((1.500 < volts) && (volts < 2.000))  //1.800
    {
      direction = "W";
      degress = 270;
    }
    if ((0.870 <= volts) && (volts < 1.100))  //0.978
    {
      direction = "WNW";
      degress = 292.5;
    }
    if ((1.100 <= volts) && (volts <= 1.500))  //1.299
    {
      direction = "NW";
      degress = 315;
    }
    if ((0.500 <= volts) && (volts < 0.750))  //0.592
    {
      direction = "NNW";
      degress = 337.5;
    }
  }
```

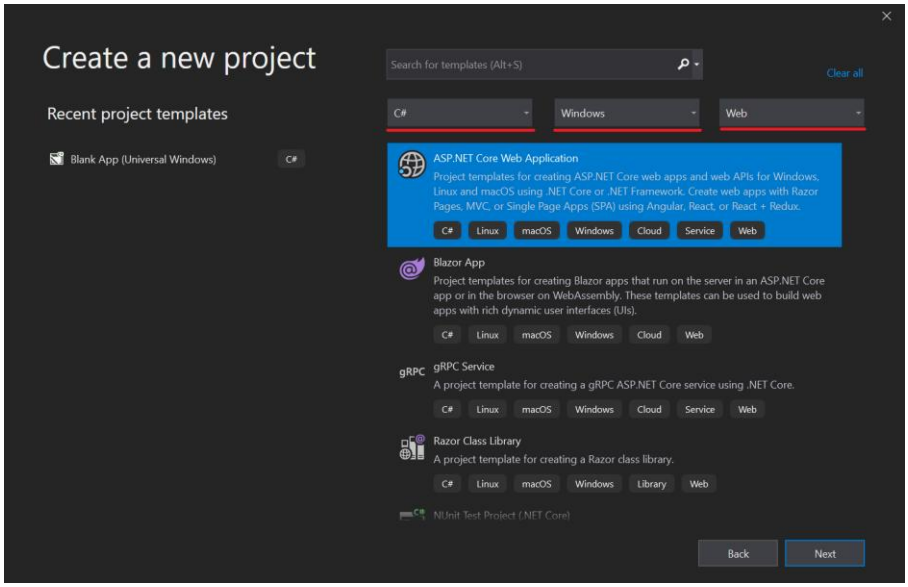
Για κάθε τάση μέτρησης γίνεται ένας έλεγχος με *if* για να καθοριστεί σε ποιο εύρος τιμών ανήκει και ορίζεται αντίστοιχα η κατεύθυνση ανέμου περιγραφικά και σε μοίρες.

Κεφάλαιο 7 -Εφαρμογή ASP .NET REST API

Η εφαρμογή που εκτελείται απομακρυσμένα στο Azure και λαμβάνει από το Raspberry τα δεδομένα των μετρήσεων, είναι μία εφαρμογή ASP .NET core WEB API. Ακολουθεί την αρχιτεκτονική REST API. Η επικοινωνία δηλαδή μεταξύ client (Raspberry) και server (WEB API) γίνεται με messages βασισμένα στο πρωτόκολλο HTTP.

7.1 Δημιουργία project στο Visual Studio 2019

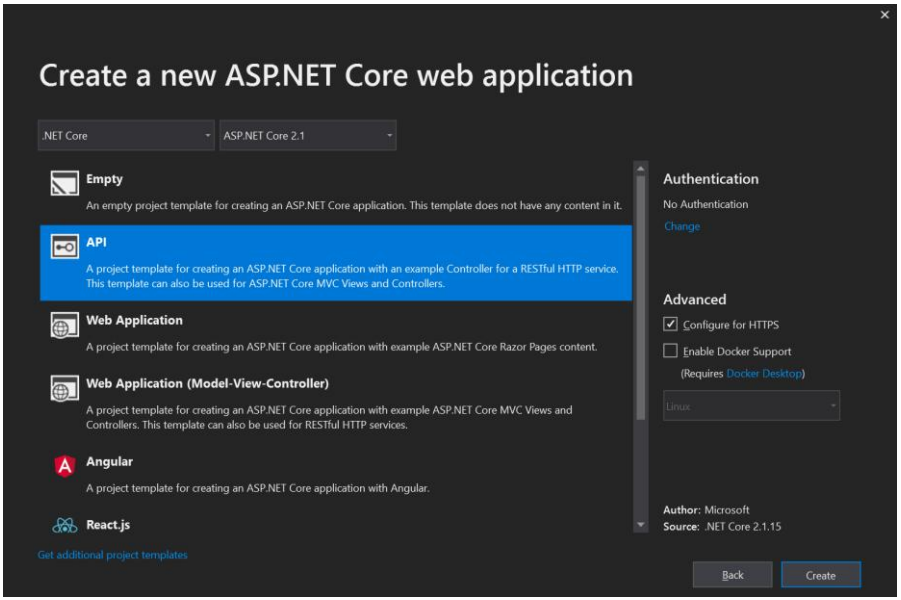
Ξεκινάμε το Visual Studio 2019 και επιλέγουμε **Create New Project**. Επιλέγουμε γλώσσα **C#**, platform **Windows** και project type **WEB** (όπως δείχνουν οι κόκκινες γραμμές στην παρακάτω εικόνα 52). Τέλος, επιλέγουμε **ASP.NET Core Web Application** και *Next*.



Εικόνα 52 Asp.Net Create Project

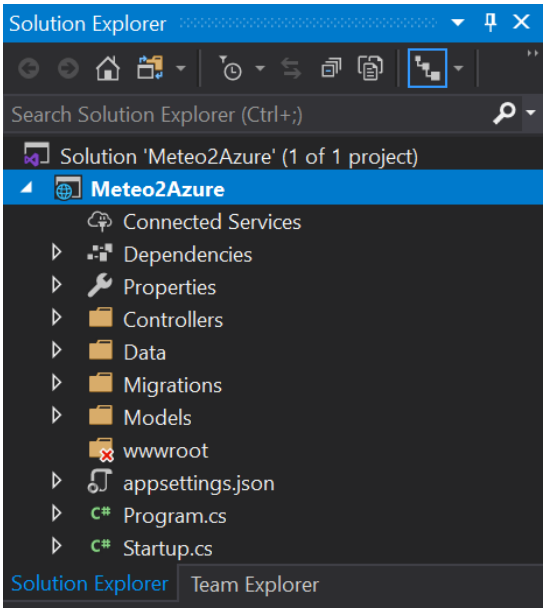
Στη συνέχεια δίνουμε το όνομα του project και το path του και πατάμε *Create*. Στην επόμενη οθόνη (Εικόνα 53) πρέπει να επιλέξουμε ποιο template επιθυμούμε, π.χ. Empty project, API, Web Application κλπ. Επιλέγουμε **API** το οποίο είναι ένα project template για τη δημιουργία μίας ASP.NET Core εφαρμογής με ένα υπόδειγμα Controller για RESTful HTTP service. Αυτό το template μπορεί επίσης να χρησιμοποιηθεί και για ASP.NET Core MVC Views και Controllers.

Δεξιά στις επιλογές *Advanced* κάνουμε check (αν δεν είναι) την επιλογή Configure for HTTPS και τέλος πατάμε *Create*.



Εικόνα 53 Asp.Net Create Web Application

Μόλις ολοκληρωθεί η δημιουργία του project, επιλέγουμε το project στον solution explorer(Εικόνα 54).



Εικόνα 54 Asp.Net Solution Explorer

Στο Solution Explorer βλέπουμε ότι, μεταξύ άλλων, υπάρχουν οι φάκελοι *Controllers* και *Models*. Στον φάκελο Controllers θα αποθηκεύσουμε τον δικό μας Controller για την υποδοχή των δεδομένων μετρήσεων από το Raspberry. Στο φάκελο Models θα αποθηκεύσουμε το δικό μας data model. Αυτό είναι μία class **ακριβώς** ίδια με την class DataFromSensors της Raspberry εφαρμογής, η οποία περιέχει τα δεδομένα μας. Αυτή η class αποτελεί το code first μοντέλο της βάσης δεδομένων μας. Το Entity Framework θα φτιάξει έναν πίνακα στη database μας, που θα έχει την εικόνα της class DataFromSensors.

7.2 Model

Πριν φτιάξουμε έναν δικό μας controller, πρέπει πρώτα να φτιάξουμε το model της εφαρμογής μας. Το Model είναι η class DataFromSensors. Επιλέγουμε με δεξί κλικ τον φάκελο *Models* του project μας στο solution explorer, και μετά *Add*, μετά *Class* και μετά δίνουμε στην class το όνομα **DataFromSensors.cs**.

```
namespace Meteo2Azure
{
  public class DataFromSensors
  {
    public int Id { get; set; }
    public string Temperature { get; set; }
    public string Humidity { get; set; }
    public string AtmPressure { get; set; }
    public string WindSpeedKMH { get; set; }
    public string WindSpeedBF { get; set; }
    public string WindDirection { get; set; }
    public string WindDirectionDegrees { get; set; }
    public string RainMM { get; set; }
    public string RawPhotoBase64 { get; set; }
  }
}
```

Η class αυτή πρέπει να έχει **ακριβώς** το ίδιο όνομα με την αντίστοιχη στην εφαρμογή του Raspberry, και το ίδιο πρέπει να ισχύει και για τα members. **Πρέπει** να έχουν **ακριβώς** τα ίδια ονόματα, ίδιο τύπο και να είναι όλα public.

Η διαφορά εδώ είναι ότι, όλα τα members που αναπαριστούν τα δεδομένα μας πρέπει να είναι **properties** και επιπλέον, το πρώτο member πρέπει υποχρεωτικά να είναι ένα **Id** τύπου int με όνομα **ακριβώς** όπως δίνεται: κεφαλαίο I και μικρό d. Αυτό είναι το Id της εγγραφής στη database και το χρειάζεται το entity framework για να καταλάβει ότι αυτή η class είναι data model που θα τη χρησιμοποιήσει με την τεχνική code first για τη δημιουργία της database και όχι μία απλή τυπική class.

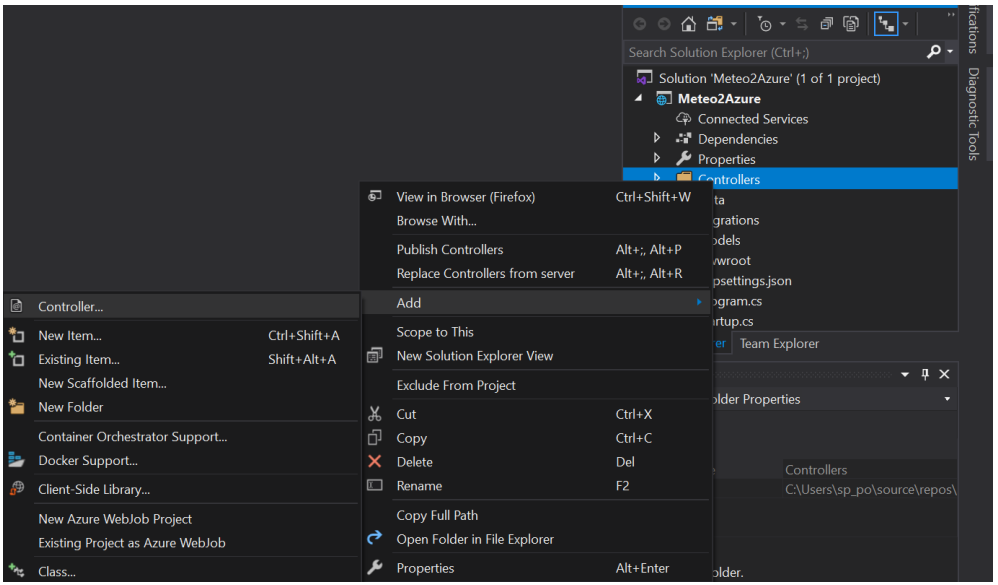
Στο σημείο αυτό πρέπει **<u>οπωσδήποτε</u>** να κάνουμε Build την εφαρμογή, διαφορετικά αργότερα μόλις δημιουργήσουμε τον δικό μας Controller, το Visual Studio δεν θα μπορεί να κάνει bind τον Controller με το Model.

7.3 Δημιουργία DataFromSensorsController Controller

Ο controller σύμφωνα με το MVC pattern είναι μία class όπου κατάλληλες methods δέχονται τα αντίστοιχα HTTP requests και κάνουν τις απαραίτητες ενέργειες που πρέπει να κάνουν. Π.χ. να ενημερώσουν το model ή να καλέσουν το View (όπου υπάρχει).

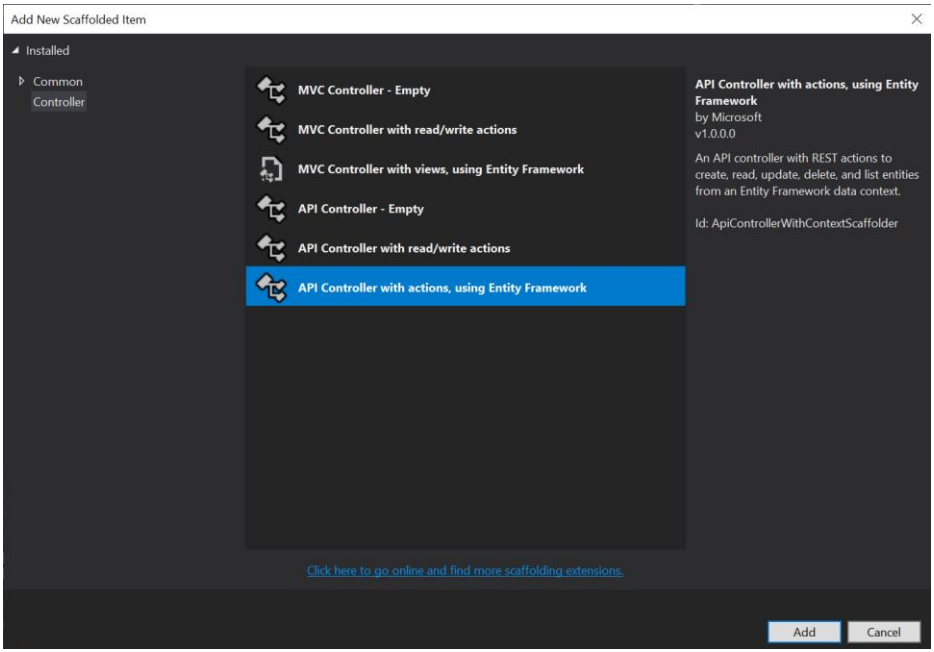
Το Visual Studio μέσω του template που επιλέξαμε κατά τη δημιουργία του project, έχει φτιάξει έναν demo controller, με όνομα *ValuesController.cs*.

Εμείς θα φτιάξουμε έναν δικό μας Controller, που θα υποστηρίζει και τη λειτουργικότητα του Entity Framework. Υπενθυμίζουμε ότι πριν φτιάξουμε τον δικό μας Controller, και αμέσως μετά τη δημιουργία του Model, θα πρέπει να έχουμε κάνει Build το project, διαφορετικά δεν θα μπορέσει να γίνει bind ο Controller με το Model. Στο σημείο αυτό Επιλέγουμε με δεξί κλικ τον φάκελο *Controllers*, μετά *Add* και μετά *Controller*(Εικόνα 55).



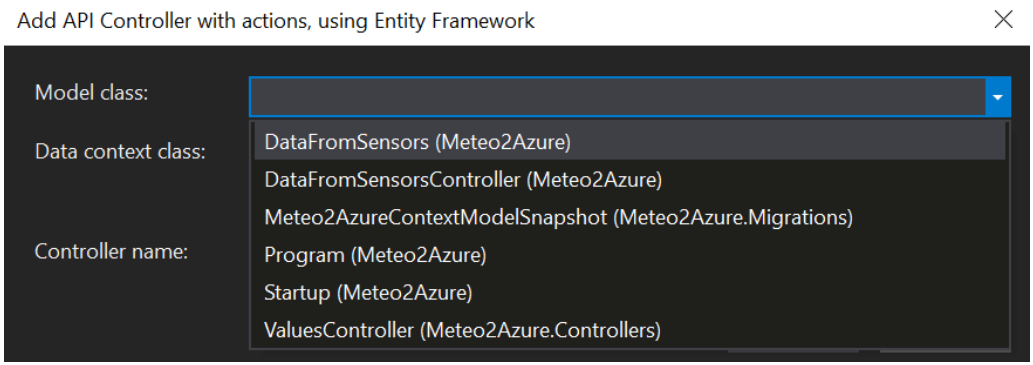
Εικόνα 55 Asp.Net Insert Controller 1

Στην οθόνη που ακολουθεί, επιλέγουμε API Controller with actions, using Entity Framework (Εικόνα 56), πατάμε *Add*.



Εικόνα 56Asp.Net Insert Controller 2

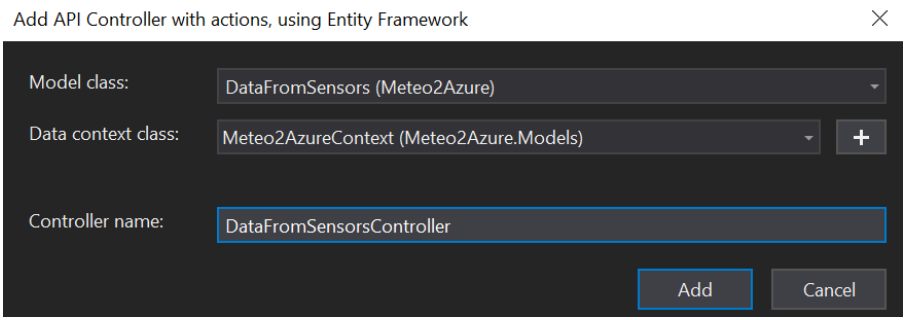
Στη συνέχεια εμφανίζεται το παρακάτω παράθυρο (Εικόνα 57), όπου πρέπει να διαλέξουμε τη Model class που θα συσχετιστεί με τον controller που δημιουργούμε. Επιλέγουμε *DataFromSensors (Meteo2Azure)*. Αν δεν εμφανίζεται η model class DataFromSensors, σημαίνει ότι δεν έχουμε κάνει Build το project, όπως αναφέρθηκε νωρίτερα. Κάνουμε Build και ξανακάνουμε την ίδια διαδικασία δημιουργίας Controller.



Εικόνα 57 Asp.Net Insert Controller 3

Μόλις επιλέξουμε τη Model class, αυτόματα θα συμπληρωθεί ένα προτεινόμενο όνομα για τον Controller. Το αφήνουμε όπως έχει, και πρέπει θυμόμαστε ότι όταν θα καλούμε τον Controller για επικοινωνία, το όνομα που θα χρησιμοποιούμε είναι το όνομα του Controller χωρίς τη λέξη Controller. Π.χ. το όνομα του Controller είναι DataFromSensorsController. Για να αναφερθούμε στον Controller θα χρησιμοποιούμε τη λέξη DataFromSensors case sensitive.

Στο Data Context Class (Εικόνα 58) πατάμε το "+" και επιλέγουμε το προτεινόμενο όνομα.



Εικόνα 58 Asp.Net Insert Controller 4

7.4 DataFromSensorsController.cs

Όταν δημιουργηθεί ο Controller, περιέχει έτοιμο template κώδικα για RESTful API με κάποιες βασικές λειτουργίες για όλα τα HTTP instructions. Εμείς χρειαζόμαστε το POST instruction που θα δέχεται τα data από το Raspberry. Επίσης για λόγους διαχείρισης και debug χρειαζόμαστε και τα GET, DELETE.

```
using Microsoft.EntityFrameworkCore;
using Meteo2Azure.Models;


    public class DataFromSensorsController : ControllerBase
    {
        private readonly Meteo2AzureContext _context;

        public DataFromSensorsController(Meteo2AzureContext context)
        {
            _context = context;
        }
}
```

Το namespace *Microsoft.EntityFrameworkCore* παρέχει τη λειτουργικότητα του Entity Framework. Το namespace *Meteo2Azure.Models* παρέχει πρόσβαση στο data context. Η class *DataFromSensorsController* περιέχει τις methods που αποκρίνονται στα HTTP requests . Περιέχει ένα private member τύπου *Meteo2AzureContext* το οποίο παρέχει πρόσβαση στο data context, δηλαδή στη **database**.

7.4.1 PostDataFromSensors()

Η method αυτή είναι υπεύθυνη για να απαντάει στα HTTP Post instructions που δέχεται ο controller. Υπάρχει και ένα σχετικό attribute στη δήλωση της method.

Δέχεται σαν παράμετρο ένα object τύπου *DataFromSensors* επειδή περιμένει αυτά τα δεδομένα να της έρθουν μέσω HTTP από τον client. Πράγματι, όπως είδαμε, ο client (το Raspberry) στέλνει μία Post HTTP εντολή, που περιέχει τα δεδομένα μέσα σε ένα DataFromSensors object, μορφοποιημένα κατά JSON. Μόλις το ASP.NET runtime πάρει την POST εντολή που απευθύνεται στον συγκεκριμένο controller, κάνει de-serialize τα JSON δεδομένα και ξαναφτιάχνει το αρχικό DataFromSensors object. Μετά κοιτάζει αν ο controller περιέχει κατάλληλη POST method που να δέχεται σαν παράμετρο DataFromSensors και την εκτελεί.

```
// POST: api/DataFromSensors
[HttpPost]
public async Task<IActionResult> PostDataFromSensors([FromBody] DataFromSensors dataFromSensors)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    _context.DataFromSensors.Add(dataFromSensors);
    await _context.SaveChangesAsync();

    return CreatedAtAction("GetDataFromSensors", new { id = dataFromSensors.Id }, dataFromSensors);
}
```

Αφού γίνει ένας σύντομος έλεγχος ότι το model είναι valid, καλούμε την *_context.DataFromSensors.Add(dataFromSensors)*. Μέσω αυτής κλήσης, έχουμε πρόσβαση στο data context, το οποίο όπως είπαμε είναι ένα abstraction της database μας, και συγκεκριμένα στον πίνακα DataFromSensors. Επομένως στη database και στον πίνακα DataFromSensors κάνει προσθέτει με *Add* ότι έχει σαν παράμετρο. Δηλαδή ένα DataFromSensors object, το οποίο είναι τα data που ήρθαν από το Raspberry.

Μόλις εκτελεστεί η *Add()* καλούμε την ασύγχρονη method *SaveChangesAsync()*, η οποία είναι το contex επομένως επιδρά στη database, και συγκεκριμένα, όπως φαίνεται από το όνομά της κάνει στην database ότι αλλαγές κάναμε (Add στη περίπτωσή μας).

Με την εντολή return επιστρέφεται και το response στον client δηλαδή στο Raspberry. Συγκεκριμένα το response είναι όλο το object που αποθηκεύτηκε στη βάση. Μπορούμε φυσικά να ορίσουμε ότι θέλουμε σαν response.

7.4.2 GetDataFromSensors()

Η method αυτή είναι υπεύθυνη για να απαντάει στα HTTP Get instructions που δέχεται ο controller. Υπάρχει και ένα σχετικό attribute στη δήλωση της method.

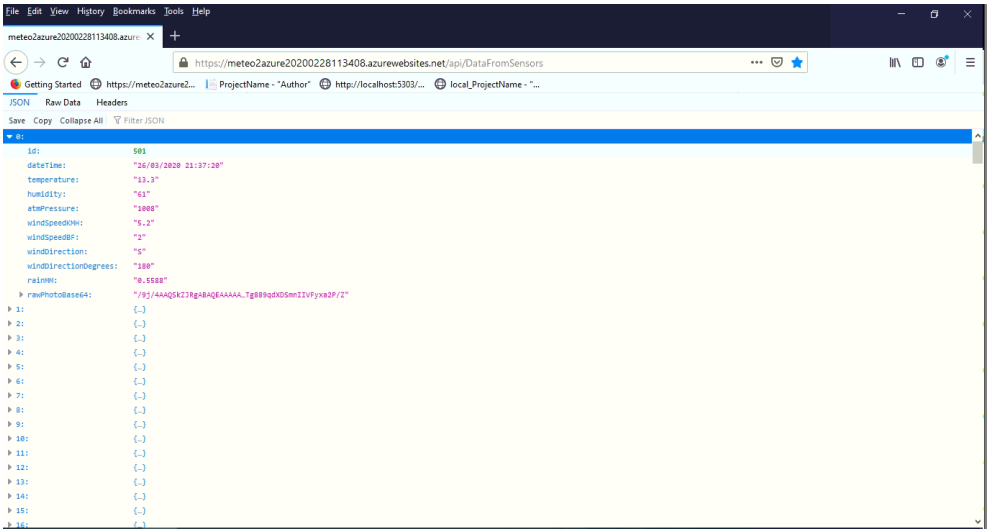
```
public IEnumerable<DataFromSensors> GetDataFromSensors()
{
  return _context.DataFromSensors;
}
```

Η συγκεκριμένη *GetFromSensors()* δεν έχει παραμέτρους, επομένως επιστρέφει όλα τα data έχει η database (όλα τα records). Αυτό γίνεται με την κλήση *_context.DataFromSensors*. Για το λόγο αυτό τιμή επιστροφής της *GetDataFromSensors()* είναι IEnumerable<DataFromSensors>.

Αυτή η method θα κληθεί αν από έναν browser ζητήσουμε εξυπηρέτηση από την παρακάτω διεύθυνση:

https://meteo2azure20200228113408.azurewebsites.net/api/DataFromSensors

Θα πάρουμε την παρακάτω απάντηση (Εικόνα 59), όπου φαίνονται όλες οι records της database μορφοποιημένες από τον browser (και όχι από εμάς) σαν JSON. Αν στον browser επιλέξουμε Raw Data θα εμφανιστούν όπως στέλνονται από τον server. Το πρώτο record φαίνεται αναλυτικά.



Εικόνα 59 Json Raw Data View

7.4.3 GetDataFromSensors(int id)

Αυτή η method είναι ένα overload της GetDataFromSensor() και δέχεται μία παράμετρο τύπου *int*. Η παράμετρος είναι το Id του record που θέλουμε να διαβαστεί από τη database και αποσταλεί στον client.

```
public async Task<IActionResult> GetDataFromSensors([FromRoute] int id)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var dataFromSensors = await _context.DataFromSensors.FindAsync(id);

    if (dataFromSensors == null)
    {
        return NotFound();
    }

    return Ok(dataFromSensors);
}
```

Η υλοποίηση είναι παρόμοια με την αντίστοιχη method χωρίς παράμετρο. Η διαφορά είναι ότι η πρόσβαση στο data context (database) γίνεται με μία Find εντολή με παράμετρο το Id που θέλουμε: *_context.DataFromSensors.FindAsync(id)*Ένα δεν βρεθεί το ζητούμε Id, η συνάρτηση επιστρέφει NotFound(), διαφορετικά επιστρέφει το object με τα data σε raw μορφή.

7.4.4 DeleteDataFromSensors()

Η method αυτή λαμβάνει και χειρίζεται την DELETE HTTP instruction. Παίρνει σαν παράμετρο μία τιμή int που αντιστοιχεί στο Id που θέλουμε να διαγράψουμε από τη database.

```
// DELETE: api/DataFromSensors/5
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteDataFromSensors([FromRoute] int id)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var dataFromSensors = await _context.DataFromSensors.FindAsync(id);
    if (dataFromSensors == null)
    {
        return NotFound();
    }

    _context.DataFromSensors.Remove(dataFromSensors);
    await _context.SaveChangesAsync();

    return Ok(dataFromSensors);
}
```

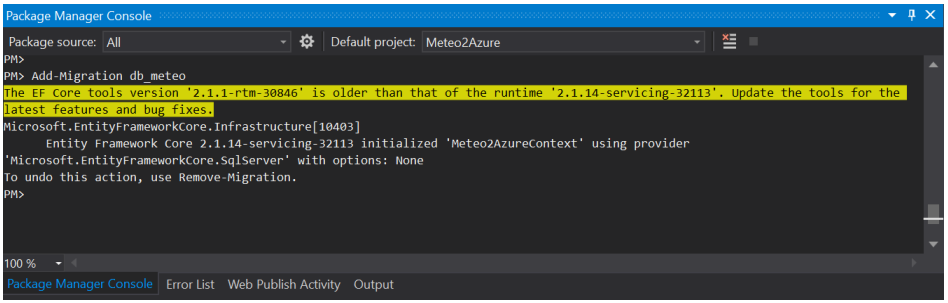
Γίνεται μία κλήση στην *_context.DataFromSensors.FindAsync(id)* για να λάβουμε από τη database την εγγραφή που θέλουμε να διαγράψουμε. Αν δεν βρέθηκε εγγραφή με το ζητούμενο Id, η method επιστρέφει NotFound() response. Διαφορετικά αν υπάρχει, καλείται η method *_context.DataFromSensors.Remove(dataFromSensors)* η οποία διαγράφει την εγγραφή από τη database. Στη συνέχεια καλούμε την SaveChangesAsync() για να αποθηκευτούν οι αλλαγές στη database. Τέλος στέλνεται ένα OK response συμπεριλαμβανομένου του object που διαγράφτηκε. Μπορούμε όμως αν θέλουμε να στείλουμε μόνο OK response χωρίς τα data, χρησιμοποιώντας την Ok() overload χωρίς παραμέτρους.

7.5 Δημιουργία τοπικής database

Στο σημείο αυτό, έχει δημιουργηθεί το Model και ο Controller. Τώρα είμαστε έτοιμοι να δημιουργήσουμε τη τοπική database, η οποία θα είναι τύπου *MS SQL localdb*.

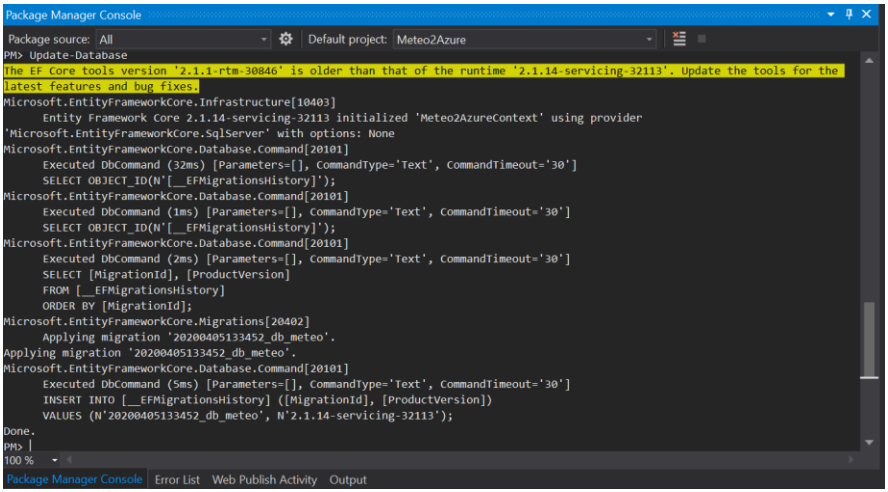
Επιλέγουμε Tools->NuGet Package Manager->Package Manager Console.

Στο παράθυρο console που ανοίγει χαμηλά στη οθόνη του Visual Studio, γράφουμε **Add-Migration db_meteo**.



Εικόνα 60 Local Database 1

Μόλις δημιουργηθεί επιτυχώς το migration (Εικόνα 60), γράφουμε **Update-Database** για να δημιουργηθεί ο πίνακας βάση το Model στη database.



Εικόνα 61 Local Database 2

Η παραπάνω οθόνη (Εικόνα 61), δείχνει ότι έγινε επιτυχώς η δημιουργία. Τώρα είμαστε πλέον να αρχίσουμε να γράψουμε data στη database. Αυτή η διαδικασία Add-Migrations και Update-Database, πρέπει να γίνεται κάθε φορά που γίνεται μία αλλαγή στη Model class, έτσι ώστε να ενημερώνεται και η database για τις αλλαγές.

7.6 Meteo2AzureContext.cs

Η *Meteo2AzureContext* class κληρονομεί από την DbContext και έχει μία property member, την DataFromSensors, η οποία είναι τύπου

DbSet<*Meteo2Azure.DataFromSensors*>. Αυτή η property μας δίνει πρόσβαση στα data της βάσης δεδομένων.

```
using Microsoft.EntityFrameworkCore;
using Meteo2Azure;

namespace Meteo2Azure.Models
{
  public class Meteo2AzureContext : DbContext
  {
    public Meteo2AzureContext (DbContextOptions<Meteo2AzureContext> options)
      : base(options)
    {
    }

    public DbSet<Meteo2Azure.DataFromSensors> DataFromSensors { get; set; }
  }
}
```

Η DataFromSensors property είναι αυτή που καλείται από τον Controller για να προσπελάσει τα δεδομένα της database, όπως είδαμε νωρίτερα.

Ο constructor της Meteo2AzureContext class, παίρνει σαν παράμετρο κάποια *DbContextOptions* τα οποία ορίζονται κατά την εκκίνηση της εφαρμογής, στο αρχείο *Startup.cs* όπως θα δούμε σε λίγο.

7.7 Startup.cs

Το αρχείο αυτό δημιουργείται από το Visual Studio μόλις δημιουργούμε το project. Περιέχει την class Startup η οποία περιέχει διάφορες methods που εκτελούνται από το runtime.

Η method *ConfigureServices*, μεταξύ άλλων, περιέχει μία κλήση στην *services.AddDbContext*, η οποία προσθέτει ένα database service στην εφαρμογή, κάνοντάς το register σε ένα data context. Σαν παράμετρο παίρνει μία action, η οποία κάνει μία κλήση στη method *GetConnectionString("Meteo2AzureContext")*. Όπως φανερώνει το όνομά της, η method αυτή βρίσκει το connection string της database. Το connection string είναι ορισμένο με το όνομα *Meteo2AzureContext* στο αρχείο appsettings.json του project, το οποίο δημιουργήθηκε από το Visual Studio, όπως θα δούμε σε λίγο.

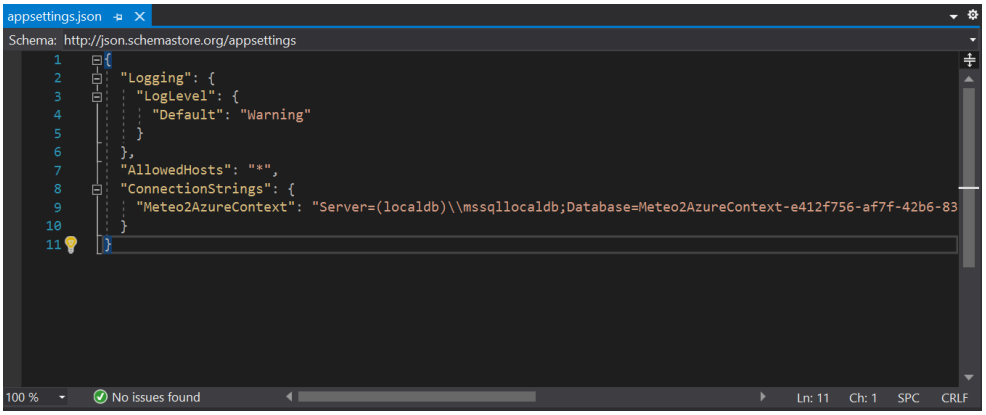
```
// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

  services.AddDbContext<Meteo2AzureContext>(options =>
```

```
            options.UseSqlServer(Configuration.GetConnectionString("Meteo2AzureContext")));
    }
```

7.8 appsettings.json

Το αρχείο αυτό (Εικόνα 62) δημιουργείται από το Visual Studio κατά τη δημιουργία του project. Είναι το αρχείο ρυθμίσεων της εφαρμογής.



Εικόνα 62 appsettings.json

Μία από τις ρυθμίσεις είναι το database connection string:

```
 "ConnectionStrings": {
   "Meteo2AzureContext": "Server=(localdb)\\mssqllocaldb;Database=Meteo2AzureContext-e412f756-af7f-
42b6-83ec-9fd5730ab279;Trusted_Connection=True;MultipleActiveResultSets=true"
```

Στη περίπτωση αυτή, που η database είναι τοπικά στο υπολογιστή μας και όχι ακόμα στο Azure, παρατηρούμε ότι η database είναι η localdb.

Αργότερα, όταν θα κάνουμε την εφαρμογή deploy στο Azure, το connection string θα αλλάξει στο αντίστοιχο για τη σύνδεση με την απομακρυσμένη database.

Κεφάλαιο 8 Εφαρμογή ASP .NET Web Application

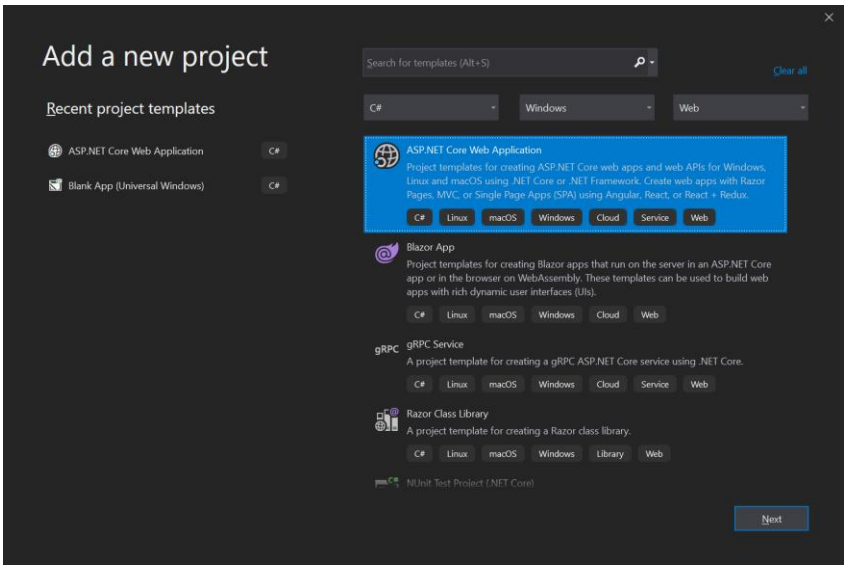
Στο προηγούμενο κεφάλαιο αναλύθηκε η Web API εφαρμογή, η οποία εκτελείται στο Azure, λαμβάνει τα δεδομένα από τον client (Raspberry) και τα αποθηκεύει στη database η οποία βρίσκεται επίσης στο Azure.

Στο Azure θα εκτελείται και ένα δεύτερο application service, το οποίο θα διαβάζει data (μετρήσεις) από την database και θα εμφανίζει τις μετρήσεις και τη φωτογραφία σε μία ιστοσελίδα. Θα είναι δηλαδή ένα ASP .NET Web Application. Αυτή η εφαρμογή θα ακολουθεί το MVC pattern.

Εφόσον το API app service και το WEB app service μοιράζονται μία κοινή database, μπορούμε να τα έχουμε στο ίδιο solution σαν ξεχωριστά project.

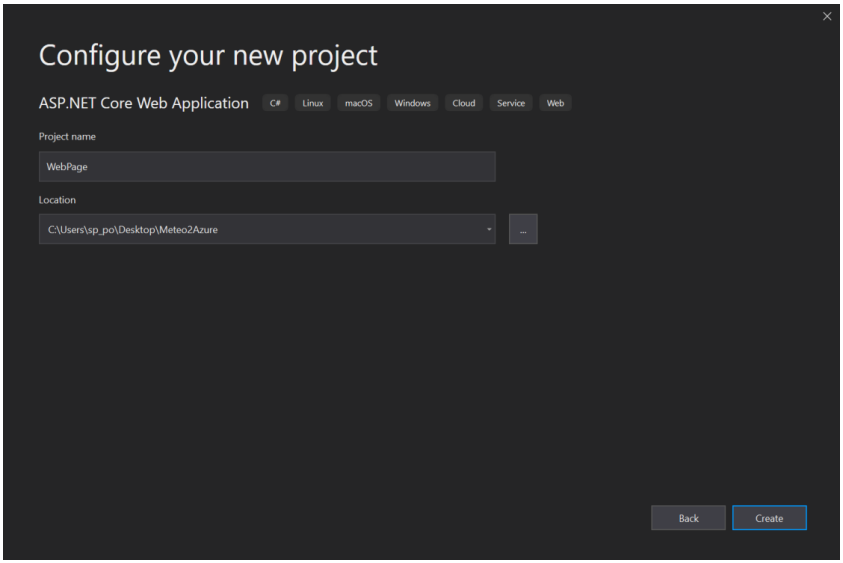
8.1 Δημιουργία project στο Visual Studio

Ανοίγουμε το Meteo2Azure solution. Σε αυτό υπάρχει ήδη το project Meteo2Azure, το οποίο είναι το API application. Με δεξί κλικ πάνω στο solution επιλέγουμε,
Add->New Project. Στη συνέχεια επιλέγουμε στο παράθυρο με τα templates επιλέγουμε **C#->Windows->Web** και τέλος το project type **ASP .NET Core Web Application** (Εικόνα 63) και *Next*.



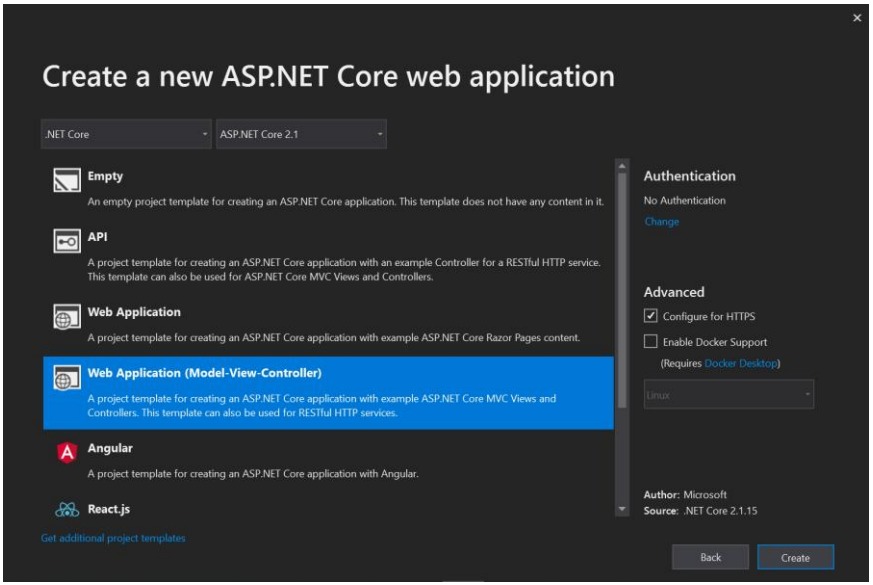
Εικόνα 63 Create ASP .NET Core Web Application 1

Στην επόμενη οθόνη (Εικόνα 64) επιλέγουμε το όνομα του project, *WebPage* στη περίπτωσή μας, και μετά πατάμε *Create*.



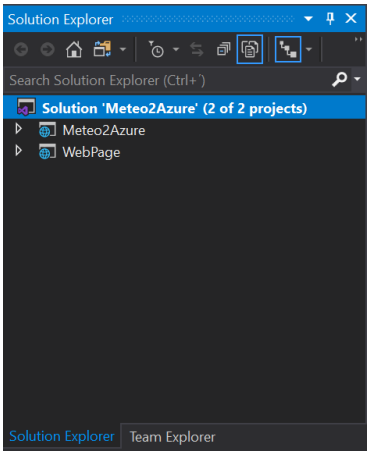
Εικόνα 64 Create ASP .NET Core Web Application 2

Στην επόμενη οθόνη (Εικόνα 65) επιλέγουμε τον τύπο της εφαρμογής που θέλουμε. Επιλέγουμε: **Web Application (Model-View-Controller)**



Εικόνα 65 Create ASP .NET Core Web Application 3

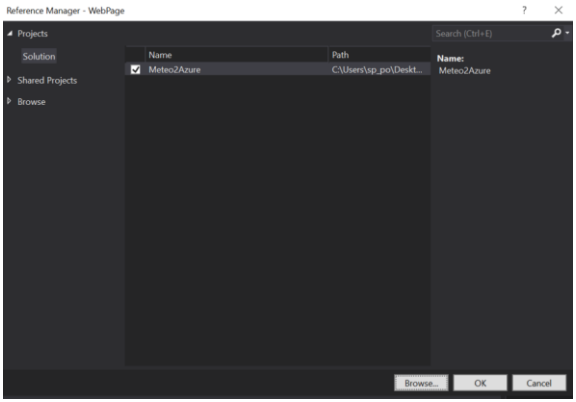
Όταν ολοκληρωθεί η δημιουργία του project, στο solution explorer βλέπουμε και τα δύο project (Εικόνα 66).



Εικόνα 66 Create ASP .NET Core Web Application 4

8.2 Model

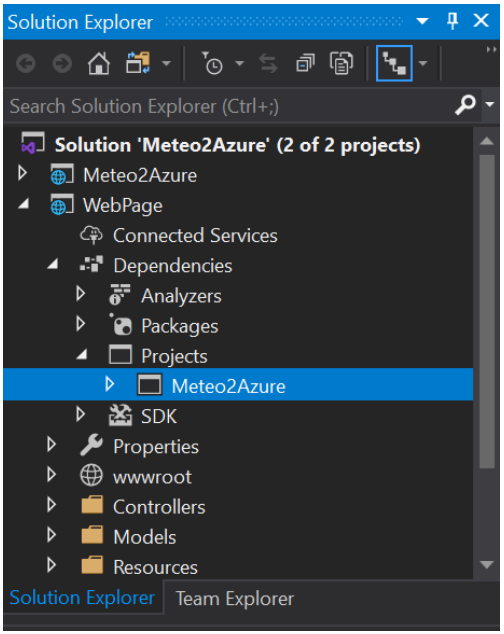
Στην εφαρμογή αυτή δεν υπάρχει ξεχωριστό Model. Τα δεδομένα που πρέπει να εμφανίσει βρίσκονται στην database της εφαρμογής API. Η εφαρμογή API βρίσκεται στο ίδιο solution, οπότε μπορούμε να κάνουμε reference το project της API εφαρμογής στο project της Web εφαρμογής. Με τον τρόπο αυτό, θα έχουμε πρόσβαση στη database καθώς θα έχουμε πρόσβαση στο data context το *Meteo2AzureContext*. Πατάμε δεξί κλικ στην επιλογή Dependencies του project WebPage και επιλέγουμε **Add Reference**.



Εικόνα 67 ASP .NET Core Reference

Όπως φαίνεται παραπάνω (Εικόνα 67), επιλέγουμε **Projects** και μετά πατάμε *Browse*. Επιλέγουμε το API project (Meteo2Azure), επιλέγουμε το σχετικό checkbox και τέλος πατάμε OK.

Τέλος, ελέγχουμε αν φαίνεται στην επιλογή *Dependencies* του Solution Explorer (Εικόνα 68):



Εικόνα 68 ASP .NET Core Dependencies

8.3 Δημιουργία WeatherController Controller

Ο controller της Web εφαρμογής είναι υπεύθυνος για την εμφάνιση της ιστοσελίδας με τις μετρήσεις.

Αφού πρώτα κάνουμε Build το project WebPage, πατάμε δεξί κλικ πάνω στον φάκελο Controllers και επιλέγουμε **Add->Controller**. Στη συνέχεια επιλέγουμε **MVC Controller – Empty** και πατάμε *Add*. Δίνουμε στον controller το όνομα **WeatherController**. Προσθέτουμε οπωσδήποτε το **using Meteo2Azure.Models**, έτσι ώστε να έχουμε πρόσβαση στο data context άρα και στη database του Meteo2Azure project.

```
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using Meteo2Azure.Models;
```

```
// For more information on enabling MVC for empty projects, visit
https://go.microsoft.com/fwlink/?LinkID=397860

namespace WebPage.Controllers
{
  public class WeatherController : Controller
  {
    private readonly Meteo2AzureContext _context;

    public WeatherController(Meteo2AzureContext context)
    {
      _context = context;
    }

    [HttpGet]
    public IActionResult Index()
    {
      // Select Latest Data
      ViewData["LatestData"] = _context.DataFromSensors.OrderByDescending(x => x.Id)
        .First();

      // Select 6 Hour Data
      ViewData["HoursData"] = _context.Select6HourData();

      return View();
    }
  }
}
```

Ο Controller έχει μία μόνο method, την *Index()*, η οποία θα εκτελεστεί όταν η εφαρμογή λάβει από τον client μία GET εντολή για την ιστοσελίδα *Index*, η οποία είναι η κύρια σελίδα της εφαρμογής.

Επίσης υπάρχει ένα member τύπου *Meteo2AzureContext*, το οποίο μας δίνει πρόσβαση στο data context.

8.3.1 Index()

Η method *Index()* καλείται όταν ο controller λάβει μία HTTP GET εντολή για την ιστοσελίδα Index.

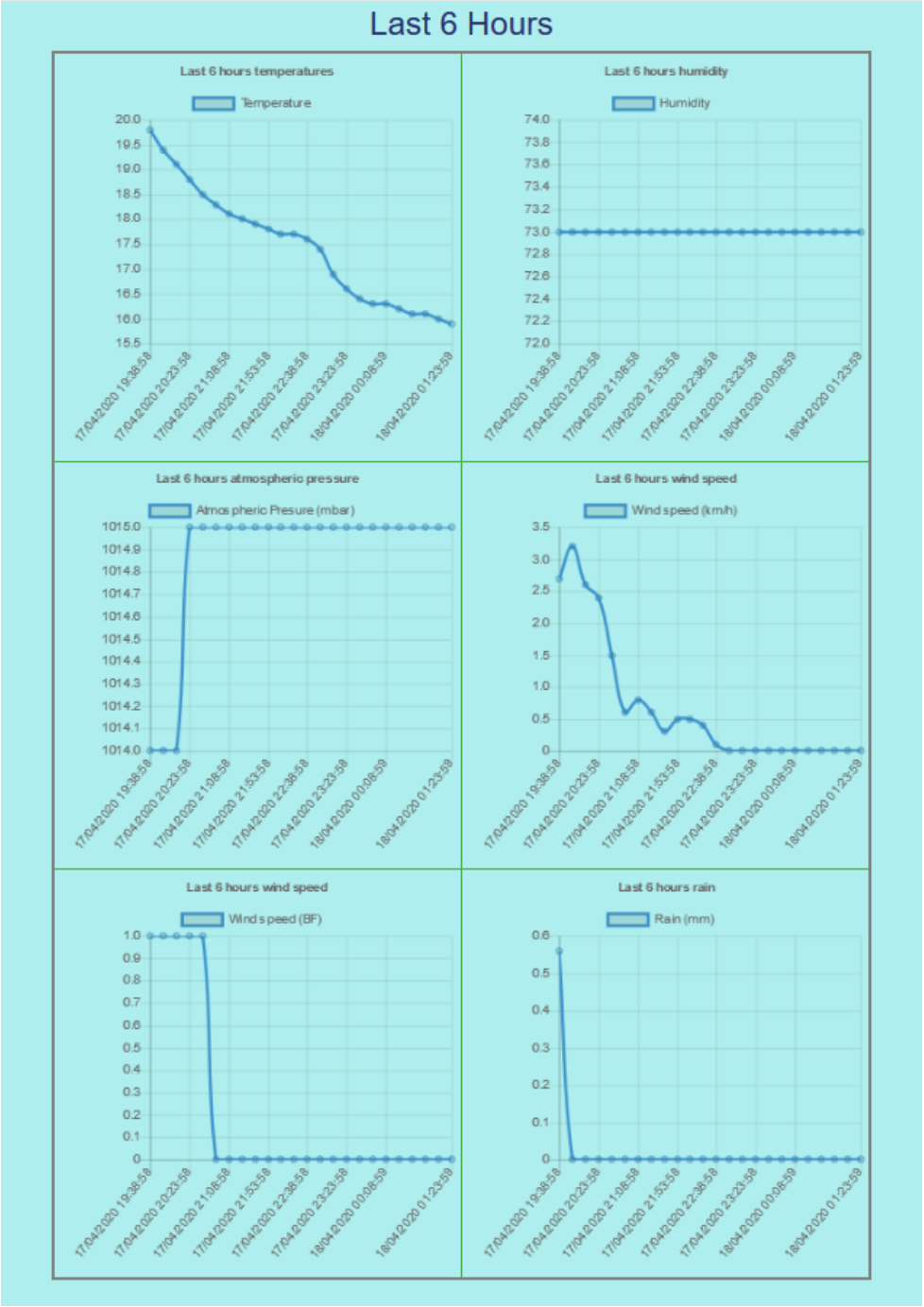
```
[HttpGet]
public IActionResult Index()
{
   // Select Latest Data
   ViewData["LatestData"] = _context.DataFromSensors.OrderByDescending(x => x.Id)
      .First();
   // Select 6 Hour Data
   ViewData["HoursData"] = _context.Select6HourData();
   return View();
}
```

Όπως φαίνεται, η method Index() αρχικά διαβάζει από τη βάση όλα τα records, τα ταξινομεί κατά φθίνουσα σειρά και παίρνει το πρώτο record το οποίο και επιστρέφεται. Αυτό αντιστοιχεί στο πιο πρόσφατο record, δηλαδή στις πιο πρόσφατες μετρήσεις.

ViewData["LatestData"] = _context.DataFromSensors.OrderByDescending(x => x.Id) .First()

Μετά καλεί την method *Select6HourData()*, η οποία επιστρέφει τις μετρήσεις των τελευταίων 6 ωρών. Τα ViewData[“LastestData”] και Viewata[“HoursData”] είναι ο σύνδεσμος του controller με το View. Με τον τρόπο αυτό στέλνονται, τα δεδομένα του Model (database) από τον Controller στο View.Τέλος, γίνεται κλήση της *View()* method, η οποία θα δημιουργήσει την ιστοσελίδα (Εικόνες 69 και 70) και θα την στείλει στον client.

Εικόνα 69 Web Browser View (Κεντρική οθόνη)



Εικόνα 70 Web Browser View (Μετρήσεις)

8.4 Startup.cs

Όπως και στο API application, στο αρχείο Startup.cs υπάρχει η method *ConfigureServices()* η οποία καλείται από το runtime και προσθέτει services στην εφαρμογή. Θα προσθέσουμε και εδώ μία κλήση στην method *services.AddDbContext()*, η οποία θα δημιουργήσει ένα database service και θα πάρει το connection string, όπως αναλύθηκε στην αντίστοιχη παράγραφο της API εφαρμογής.

```
// Add database context
services.AddDbContext<Meteo2AzureContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("Meteo2AzureContext")));
```

Το connection string είναι το ίδιο με την API εφαρμογή *(Meteo2AzureContext)*, καθώς η database είναι κοινή και για τα δύο applications.

8.5 appsettings.json

Όπως αναλύθηκε στην αντίστοιχη παράγραφο της API εφαρμογής, σε αυτό το αρχείο ρυθμίσεων της εφαρμογής, δηλώνουμε το connection string της database που θέλουμε να έχει πρόσβαση η εφαρμογή.

```
"ConnectionStrings": {
   "Meteo2AzureContext": "Server=(localdb)\\mssqllocaldb;Database=Meteo2AzureContext-e412f756-af7f-
42b6-83ec-9fd5730ab279;Trusted_Connection=True;MultipleActiveResultSets=true"
}
```

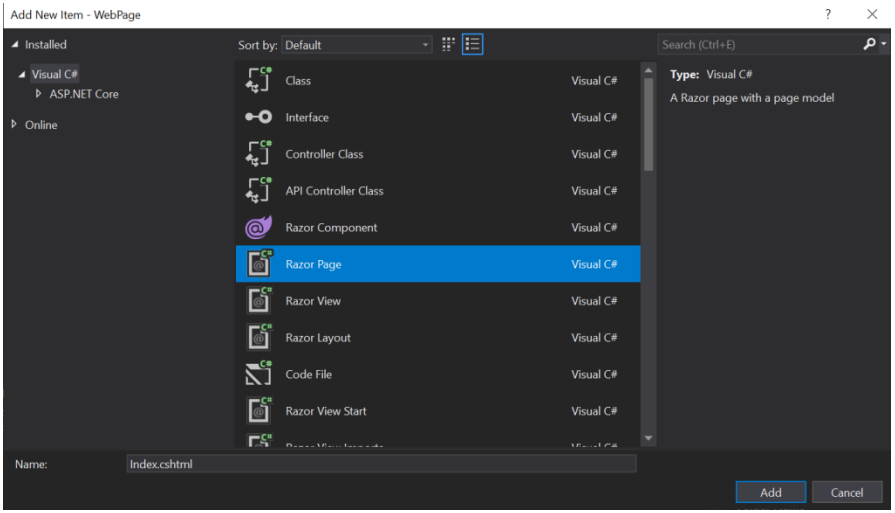
Από το αρχείο αυτό διαβάζει το connection string η AddDbContext() της Startup.cs που αναλύθηκε νωρίτερα.

8.6 View

Σύμφωνα με το MVC pattern, στο View υπάρχει ότι αφορά την αναπαράσταση των δεδομένων. Σε αυτήν την ASP .NET WEB application, το View είναι η ιστοσελίδα που θα εμφανίσει τα δεδομένα μας (τις μετρήσεις).

Με δεξί κλικ πάνω στον φάκελο Views, επιλέγουμε **Add->New Folder**. Δίνουμε οπωσδήποτε το όνομα το controller που θέλουμε να συσχετίσουμε με αυτό το View. Επιλέγουμε όνομα folder **Weather**.

Με δεξί κλικ πάνω στον φάκελο Weather, επιλέγουμε **Add->New Item** και στην οθόνη που φαίνεται παρακάτω (Εικόνα 71), επιλέγουμε **Razor Page**. Δίνουμε όνομα για τη σελίδα μας **Index.cshtml**.



Εικόνα 71 Create Razor Page

Τα αρχεία *cshtml* είναι Razor Pages. Έχουν κώδικα HTML και Javascript αν (χρειάζεται) και ταυτόχρονα έχει και κώδικα C#. Αυτό γίνεται, για να υπάρχει μία επικοινωνία του View με τον Controller και το Model.

Εκτός από το *Index.cshtml*, χρήσιμα είναι και τα αρχεία *_Layout.cshtml* και *_ViewImports.cshtml*.

Συνοπτικά, ο ρόλος είναι ο εξής:

- **_ViewImports.cshtml**: Εδώ δηλώνουμε σε ποια C# namespaces θέλουμε να έχει πρόσβαση η Razor Page μας.
- **_Layout.cshtml**: Περιέχει το κύριο σώμα της HTML σελίδας μας
- **Index.cshtml**: Περιέχει τον κώδικα HTML και Javascript που θα εμφανίσει τα δεδομένα.

8.6.1 ViewImports.cshtml

Στο αρχείο δηλώνουμε ανάλογα με τις απαιτήσεις της εφαρμογής μας, τα namespaces στα οποία χρειαζόμαστε πρόσβαση. Ο wizard του Visual Studio έχει ορίσει τα βασικά. Εμείς προσθέτουμε τα namespaces **Meteo2Azure** και **Meteo2Azure.Models**.

Με το Meteo2Azure θα έχουμε πρόσβαση στον Controller, άρα και στα δεδομένα που διαβάστηκαν από την database (lastest και 6hours) και είναι έτοιμα για εμφάνιση στην σελίδα. Με το Meteo2Azure.Models έχουμε πρόσβαση στο data conext.

```
@using WebPage
@using WebPage.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

@using Meteo2Azure
@using Meteo2Azure.Models

@using Microsoft.AspNetCore.Builder
@using Microsoft.AspNetCore.Localization
@using Microsoft.AspNetCore.Mvc.Localization
@using Microsoft.Extensions.Options
```

Παρατηρούμε ότι μπροστά από τα **using** υπάρχει το σύμβολο **@**. Με αυτό τον τρόπο ο Razor καταλαβαίνει ότι πρόκειται για C# εντολές και όχι για HTML ή Javascript.

8.6.2 Layout.cshtml

Ο wizard του Visual Studio έχει προσθέσει στο αρχείο κώδικα HTML που αφορούν το body μίας demo ιστοσελίδας. Εμείς κρατήσαμε τα βασικά μέρη και προσθέσαμε κώδικα που αφορά style και formatting του πίνακα εμφάνισης των δεδομένων, γραμματοσειρές του πίνακα και το body κλπ.

```
<style>
  table{
    border-collapse: collapse;
    border: 3px solid #808080 ;


  }
  td {
    border: 1px solid #4CAF50;
    font: bold 22px calibri;
  }
  th {
    border: 1px solid #ffffff;

    background-color: #4CAF50;
    font: 22px calibri;
    color: white;
```

```
    }
    body {
        background-color: paleturquoise;
        font: bold 18px arial, verdana;
        color: #273171

    }
</style>
```

8.6.3 Index.cshtml

Στο αρχείο αυτό υπάρχει ο κώδικας της σελίδας Index.html. Περιέχει κώδικα HTML και Javascript για την εμφάνιση των δεδομένων και των γραφημάτων. Η σελίδα αποτελείται από έναν πίνακα με τις τρέχουσες τιμές των μετρήσεων και από γραφήματα των μετρήσεων, τα οποία αναπαριστούν τις μετρήσεις των τελευταίων 6 ωρών.

Στην αρχή του αρχείου βρίσκονται κάποιες δηλώσεις, οι οποίες είναι κώδικας C#.

```
@{
    ViewData["Title"] = "ProjectName";
}

@{
    DataFromSensors latestData = (DataFromSensors)ViewData["LatestData"];
    List<DataFromSensors> hoursData = (List<DataFromSensors>)ViewData["HoursData"];
}
@{
    var date = DateTime.Now.ToShortDateString();

}
```

Με τον παραπάνω κώδικα, δημιουργείται ένα object με όνομα **latestdata** τύπου DataFromSensors και παίρνει τιμές από το αντίστοιχο DataFromSensors object του Controller, μέσω του Dictionary **ViewData[“LatestData”]**.

Αντίστοιχα, δημιουργείται μία List<DataFromSensors> με όνομα **hoursData**, η οποία παίρνει τιμές από την αντίστοιχη List του Controller, μέσω του Dictionary **ViewData[“HoursData”]**.

Τέλος, αποθηκεύουμε στη μεταβλητή **date** την τρέχουσα ημερομηνία και ώρα. Η ημερομηνία και ώρα θα εμφανιστεί στην ιστοσελίδα με την παρακάτω εντολή:

```
<h3>Today's date is: @date </h3>
```

Με τη δήλωση **@date** μέσα στον HTML κώδικα, αναφερόμαστε στην τρέχουσα ημερομηνία. Έτσι μπορούμε να περάσουμε C# μεταβλητές μέσα σε κώδικα HTML.

Με την παρακάτω γραμμή, δηλώνουμε ότι θα χρησιμοποιήσουμε την javascript βιβλιοθήκη *Chart.js* για την δημιουργία γραφημάτων.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/Chart.js/2.5.0/Chart.min.js">
</script>
```

Στη συνέχεια ακολουθεί ο κώδικας που εμφανίζει τον πίνακα με τις τρέχουσες τιμές και δεσμεύεται ο χώρος για την εμφάνιση των γραφημάτων. Ο χώρος που δεσμεύεται για τα γραφήματα είναι ένας πίνακας, στα κελιά του οποίου ορίζουμε canvas πάνω στο οποία θα εμφανιστούν τα γραφήματα.

```
<div class="weather-content">
  <div>
    <h3 align="center" class="weather-h1">Current weather conditions</h3>
    <table class="weather-table" align="center">
      <tbody>
        <tr>
          <th><b> Date and time</font></b></th>
          <td>@latestData.DateTime</font></td>
        </tr>
        <tr>
          <th><b> Temperture</font></b></th>
          <td>@latestData.Temperature oC</font></td>
        </tr>
        <tr>
          <th><b> Humidity</font></b></tdbgcolor=yellow>
          <td>@latestData.Humidity %</font></td>
        </tr>
        <tr>
          <th><b> Pressure</font></b></th>
          <td>@latestData.AtmPressure mbar</font></td>
        </tr>
        <tr>
          <th><b> Wind Speed km/h</font></b></th>
          <td>@latestData.WindSpeedKMH km/h</font></td>
        </tr>
        <tr>
          <th><b> Wind Speed BF</font></b></th>
          <td>@latestData.WindSpeedBF BF</font></td>
        </tr>
        <tr>
          <th><b> Wind Direction</font></b></th>
          <td>@latestData.WindDirection</font></td>
        <tr>
          <th><b> Wind Direction Degrees</font></b></th>
          <td>@latestData.WindDirectionDegrees Degrees</font></td>
        </tr>
        <tr>
```

```
            <th><b>  Rain</font></b></th>
            <td>@latestData.RainMM mm</font></td>
          </tr>
          <tr>
            <th><b>  Live photo</font></b></th>
            <td><img src=" @latestData.RawPhotoBase64" /></td>
          </tr>
        </tbody>
      </table>
    </div>

    <div>
      <h1 class="weather-h1">Last 6 Hours</h1>
      <small class="weather-small"></small>
      <div>
        <table class="weather-table" align="center">
          <tbody>
            <tr>
              <td><canvas id="line-chart1" width="400" height="400"></canvas></td>
              <td><canvas id="line-chart2" width="400" height="400"></canvas></td>
            </tr>
            <tr>
              <td><canvas id="line-chart3" width="400" height="400"></canvas></td>
              <td><canvas id="line-chart4" width="400" height="400"></canvas></td>
            </tr>
            <tr>
              <td><canvas id="line-chart5" width="400" height="400"></canvas></td>
              <td><canvas id="line-chart6" width="400" height="400"></canvas></td>
            </tr>
          </tbody>
        </table>
        <div style="clear:both"></div>
      </div>
    </div>
</div>
```

Οι τρέχουσες τιμές εμφανίζονται με προσπέλαση των αντίστοιχων member του latestData object, όπως π.χ. *@latestData.Temperature*.

Για την εμφάνιση της live εικόνας, δηλώνουμε την παρακάτω εντολή:

```
<img src=" @latestData.RawPhotoBase64" />
```

Με την εντολή αυτή, δηλώνουμε ότι τα δεδομένα που περιέχει η *@latestData.RawPhotoBase64*, είναι jpeg εικόνα κωδικοποιημένη με base64. Επομένως τα δεδομένα θα αποκωδικοποιηθούν πρώτα από Base64 σε binary και έτσι θα εμφανιστεί η εικόνα.

Τα canvas για την εμφάνιση των charts, δηλώνονται, όπως φαίνεται και παραπάνω, ως εξής:

```
<canvas id="line-chart1" width="400" height="400"></canvas>
```

Με το **id** θα καθορίζουμε στη συνέχεια, σε ποιο canvas θα εμφανιστεί κάθε chart.

Στη συνέχεια ακολουθεί ο κώδικας σε Javascript, ο οποίος θα δημιουργήσει τα γραφήματα χρησιμοποιώντας τη βιβλιοθήκη που αναφέραμε νωρίτερα.

```
<script type="text/javascript">
  new Chart(document.getElementById("line-chart1"), {
  type: 'line',
  data: {
     labels: ["@hoursData[0].DateTime", "@hoursData[1].DateTime", "@hoursData[2].DateTime",
"@hoursData[3].DateTime", "@hoursData[4].DateTime", "@hoursData[5].DateTime",
        "@hoursData[6].DateTime", "@hoursData[7].DateTime", "@hoursData[8].DateTime",
"@hoursData[9].DateTime", "@hoursData[10].DateTime", "@hoursData[11].DateTime",
        "@hoursData[12].DateTime", "@hoursData[13].DateTime", "@hoursData[14].DateTime",
"@hoursData[15].DateTime", "@hoursData[16].DateTime", "@hoursData[17].DateTime",

"@hoursData[18].DateTime","@hoursData[19].DateTime","@hoursData[20].DateTime","@hoursData[21].Date
Time","@hoursData[22].DateTime","@hoursData[23].DateTime"],
    datasets: [{
       data:
[@hoursData[0].Temperature,@hoursData[1].Temperature,@hoursData[2].Temperature,@hoursData[3].Temp
erature,@hoursData[4].Temperature,@hoursData[5].Temperature,

@hoursData[6].Temperature,@hoursData[7].Temperature,@hoursData[8].Temperature,@hoursData[9].Temp
erature,@hoursData[10].Temperature,@hoursData[11].Temperature,

@hoursData[12].Temperature,@hoursData[13].Temperature,@hoursData[14].Temperature,@hoursData[15].T
emperature,@hoursData[16].Temperature,@hoursData[17].Temperature,

@hoursData[18].Temperature,@hoursData[19].Temperature,@hoursData[20].Temperature,@hoursData[21].T
emperature,@hoursData[22].Temperature,@hoursData[23].Temperature],
       label: "Temperature",
       borderColor: "#3e95cd",
       fill: false
     }
    ]
  },
  options: {
    title: {
      display: true,
      text: 'Last 6 hours temperatures'
    }
  }
});
```

```
 new Chart(document.getElementById("line-chart2"), {
 type: 'line',
 data: {
    labels: ["@hoursData[0].DateTime", "@hoursData[1].DateTime", "@hoursData[2].DateTime",
"@hoursData[3].DateTime", "@hoursData[4].DateTime", "@hoursData[5].DateTime",
      "@hoursData[6].DateTime", "@hoursData[7].DateTime", "@hoursData[8].DateTime",
"@hoursData[9].DateTime", "@hoursData[10].DateTime", "@hoursData[11].DateTime",
      "@hoursData[12].DateTime", "@hoursData[13].DateTime", "@hoursData[14].DateTime",
"@hoursData[15].DateTime", "@hoursData[16].DateTime", "@hoursData[17].DateTime",

"@hoursData[18].DateTime","@hoursData[19].DateTime","@hoursData[20].DateTime","@hoursData[21].Date
Time","@hoursData[22].DateTime","@hoursData[23].DateTime"],
   datasets: [{
      data:
[@hoursData[0].Humidity,@hoursData[1].Humidity,@hoursData[2].Humidity,@hoursData[3].Humidity,@hour
sData[4].Humidity,@hoursData[5].Humidity,

@hoursData[6].Humidity,@hoursData[7].Humidity,@hoursData[8].Humidity,@hoursData[9].Humidity,@hours
Data[10].Humidity,@hoursData[11].Humidity,

@hoursData[12].Humidity,@hoursData[13].Humidity,@hoursData[14].Humidity,@hoursData[15].Humidity,@h
oursData[16].Humidity,@hoursData[17].Humidity,

@hoursData[18].Humidity,@hoursData[19].Humidity,@hoursData[20].Humidity,@hoursData[21].Humidity,@h
oursData[22].Humidity,@hoursData[23].Humidity],
      label: "Humidity",
      borderColor: "#3e95cd",
      fill: false
    }
   ]
 },
 options: {
  title: {
    display: true,
    text: 'Last 6 hours humidity'
  }
 }
});

..
..
..</script>
```

Για τη δημιουργία του γραφήματος, εκτελούμε ένα **new Chart()** και στις παραμέτρους του δηλώνουμε:

- Με **document.getElementById()**, το id του σημείου που εμφανιστεί το chart π.χ. *line-chart1*.
- Με **type**, τον τύπο του chart. Δηλώνουμε line.

- Με **data**, τα δεδομένα του γραφήματος. Labels είναι τα δεδομένα του X άξονα, και datasets τα δεδομένα του Y άξονα. Η προσπέλαση των δεδομένων γίνεται με αναφορά στα αντίστοιχα members της List *hoursData*, π.χ. *@hoursData[0].Temperature*.
- Με **label**, τον τίτλο του γραφήματος.
- Με **borderColor**, ορίζουμε το χρώμα του περιθωρίου.
- Με **fill**, αν είναι fill τα borders.

Για κάθε ένα από τα 6 charts που χρειαζόμαστε (θερμοκρασία, υγρασία, ατμοσφαιρική πίεση, ταχύτητα αέρα σε km/h, ταχύτητα αέρα σε BF, ύψος βροχής), θα κάνουμε ένα αντίστοιχο new Chart() με τις αντίστοιχες τιμές μετρήσεων.

Κεφάλαιο 9 -Εφαρμογή REST API Consumer

Στη φάση της ανάπτυξης της ASP .NET API εφαρμογής, υπάρχει η ανάγκη για μία εφαρμογή η οποία θα κάνει consume το API. Συγκεκριμένα, χρειάζεται μία εφαρμογή η οποία να στέλνει POST HTTP εντολές στο service, με δεδομένα εικονικών μετρήσεων, με σκοπό να προσομοιώνει το Raspberry. Με τον τρόπο αυτό δε χρειάζεται να λειτουργεί το Raspberry συνεχώς, και επιπλέον μπορούμε να εκτελούμε το API service τοπικά στον υπολογιστή όπου τρέχει ο consumer, με σκοπό να γίνεται εύκολο και γρήγορο debugging.

Για τις ανάγκες της ανάπτυξης, δημιουργήθηκε αρχικά η εφαρμογή consumer για Windows 10. Στη συνέχεια, επειδή υπήρχε η ανάγκη να συνεχιστεί η ανάπτυξη σε υπολογιστή με Windows 7, έγινε τροποποίηση του Windows 10 consumer έτσι ώστε να τρέχει σε Windows 7. Και οι δύο εκδόσεις θα αναλυθούν στη συνέχεια.

9.1 Windows 10 consumer

Η εφαρμογή του consumer για Windows 10 είναι μία UWP desktop εφαρμογή. Διαθέτει απλώς ένα button, το οποίο όταν πατηθεί γίνεται η επικοινωνία με το service. Ο κώδικας μοιάζει πολύ με τον κώδικα της method SendDataToAzure() της εφαρμογής Raspberry.

Υπάρχει δηλωμένη η class DataFromSensors, όπως ακριβώς θα την δημιουργούσε το Raspberry και όπως την περιμένει και το API service.

```
        public class DataFromSensors
        {
    public string Temperature;
    public string Humidity;
    public string AtmPressure;
    public string WindSpeedKMH;
    public string WindSpeedBF;
    public string WindDirection;
    public string WindDirectionDegrees;
    public string RainMM;
    public string RawPhotoBase64;
}


private async void Button_Click(object sender, RoutedEventArgs e)
{
    DataFromSensors data = new DataFromSensors();
    data.Temperature = "16";
    data.Humidity = "40";
```

```
        data.AtmPressure = "80";
        data.WindDirection = "WN";
        data.WindDirectionDegrees = "13";
        data.WindSpeedKMH = "10";
        data.WindSpeedBF = "2";
        data.RainMM = "1";
        data.RawPhotoBase64 = "photos10";

        HttpClient client = new HttpClient();
        client.BaseAddress = new Uri("http://localhost:5303/");
        //client.BaseAddress = new Uri("https://Meteo2Azure20191023120206.azurewebsites.net");
        string json = JsonConvert.SerializeObject(data);
        StringContent content = new StringContent(json);
        //StringContent content = new StringContent(DeviceDataJson);
        content.Headers.ContentType = new MediaTypeHeaderValue("application/json");
        HttpResponseMessage response = null;

        try
        {
          response = await client.PostAsync("api/DataFromSensors", content);
        }
        catch
        {

          throw new Exception();
        }

        var dialog = new MessageDialog("Data sent to Azure!");
        await dialog.ShowAsync();

        string response_data = await response.Content.ReadAsStringAsync();
        client.Dispose();

    }
```

Όταν πατηθεί το button, δημιουργείται ένα object τύπου DataFromSensors και βάζουμε στα members εικονικές τιμές μετρήσεων.

Στη συνέχεια ορίζουμε ένα HttpClient object το οποίο θα αναλάβει την επικοινωνία με το service. Δίνουμε τη διεύθυνση του service. Σε περίπτωση που αυτό εκτελείται τοπικά η διεύθυνση είναι η localhost:5303 (το port number θα είναι διαφορετικό σε άλλον υπολογιστή), διαφορετικά αν εκτελείται στο Azure δηλώνουμε την αντίστοιχη διεύθυνση.

Αμέσως μετά, γίνεται JSON serialize το object με τα data, και δηλώνουμε στο HTTP header το content type ως JSON.

Στη συνέχεια μέσα σε ένα try…catch block, γίνεται η κλήση της PostAsync με παράμετρο το path με το όνομα του service που απευθυνόμαστε (DataFromSensors).

Αν όλα έγιναν σωστά, παίρνουμε κατάλληλο μήνυμα σε DialogBox και λαμβάνουμε το response string.

Τέλος, κλείνουμε την HTTP σύνδεση *Dispose()*.

9.2 Windows 7 consumer

Η εφαρμογή του consumer για Windows 7, ακολουθεί την ίδια λογική με την αντίστοιχη για Windows 10. Υπήρχαν όμως δυσκολίες και πολλά εμπόδια στη χρήση της **HttpClient** στα Windows 7.

Τελικά επιλέχτηκε αντί της HttpClient, η **WebClient** από το namespace *System.Net*. Επομένως, η μόνη διαφορά είναι στις member methods όπου αντί της HttpClient.PostAsync, χρησιμοποιήσαμε την **WebClient.UploadString()**. Και οι δύο έχουν το ίδιο αποτέλεσμα. Στέλνουν μία HTTP POST εντολή στον server. Τα data και στην εφαρμογή αυτή είναι JSON formatted.

Μία απαραίτητη προσθήκη στον κώδικα για Windows 7, έχει να κάνει με το security protocol. Τα Windows 7 υποστηρίζουν διαφορετικό security protocol στις HTTP συνδέσεις από το default του .NET Framework 4.7 που χρησιμοποιούμε. Επομένως, πρέπει να δηλώσουμε στην εφαρμογή να χρησιμοποιεί security protocol TLS 1.2.

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
    DataFromSensors data = new DataFromSensors();
    data.Temperature = "2020";
    data.Humidity = "2020";
    data.AtmPressure = "2020";
    data.WindDirection = "40";
    data.WindDirectionDegrees = "50";
    data.WindSpeedKMH = "60";
    data.WindSpeedBF = "70";
    data.RainMM = "80";
    data.RawPhotoBase64 = "90";

    ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12;

    WebClient webClient = new WebClient();
    //webClient.BaseAddress = "https://Meteo2Azure20191023120206.azurewebsites.net";
    webClient.BaseAddress = "http://localhost:5303";
    //webClient.BaseAddress = "http://localhost:44364";
    var url = "api/DataFromSensors";
```

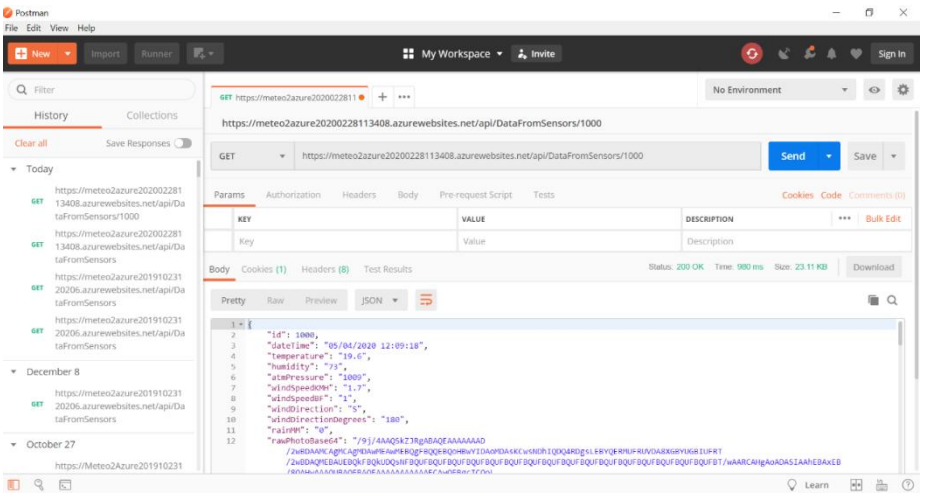
```
        webClient.Headers[HttpRequestHeader.ContentType] = "application/json";
        string dataString = JsonConvert.SerializeObject(data);

        string response;

        try
        {
            response = webClient.UploadString(url, dataString);
            MessageBox.Show("Data sent to Azure!");
        }
        catch (Exception ex)
        {
            while (ex != null)
            {
                MessageBox.Show(ex.Message);
                ex = ex.InnerException;
            }
        }
    }
```

9.3 Postman

Παράλληλα με τις API consumer εφαρμογές που αναλύθηκαν, χρησιμοποιήθηκε και η εφαρμογή **POSTMAN.** Με αυτή την εφαρμογή, μπορούμε να στείλουμε HTTP εντολές σε κάποιο service και να δούμε το response. Προσφέρει πολύ ωραίο περιβάλλον και πολύ καλή λειτουργικότητα. Γενικά είναι ένα πάρα πολύ καλό εργαλείο για να κάνει κανείς consume API.



Εικόνα 72 Postman

Στο παραπάνω παράδειγμα (Εικόνα 72), έχουμε στείλει μία HTTP GET εντολή στη διεύθυνση του API, ζητώντας το στοιχείο με Id 1000. Αυτό επιστρέφεται με το response, και εμφανίζεται σε μορφή JSON από το Postman.

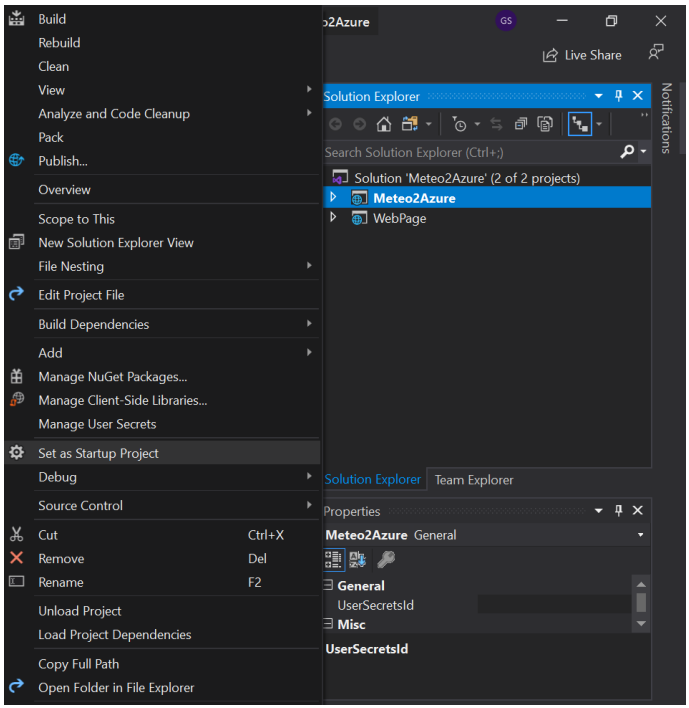
Κεφάλαιο 10 - ASP .NET applications debugging and deployment

Κατά τη διάρκεια της ανάπτυξης των δύο ASP .NET εφαρμογών (API και WEB apps), είναι φυσικά απαραίτητο να γίνεται παράλληλα debugging. Στη φάση της ανάπτυξης της API εφαρμογής είναι απαραίτητο να εκτελείται τοπικά με τοπική database για να μπορεί να γίνει debugging. Στη φάση ανάπτυξης της WEB εφαρμογής, είναι απαραίτητο να εκτελούνται και οι δύο εφαρμογές. Και η API και η WEB, καθώς αλληλοεπιδρούν μεταξύ τους.

Μόλις οι εφαρμογές λειτουργούν κανονικά, μπορεί να γίνει deployment στο Azure. Το Visual Studio παρέχει πολύ υψηλού επιπέδου λειτουργικότητα για το deployment στο Azure.

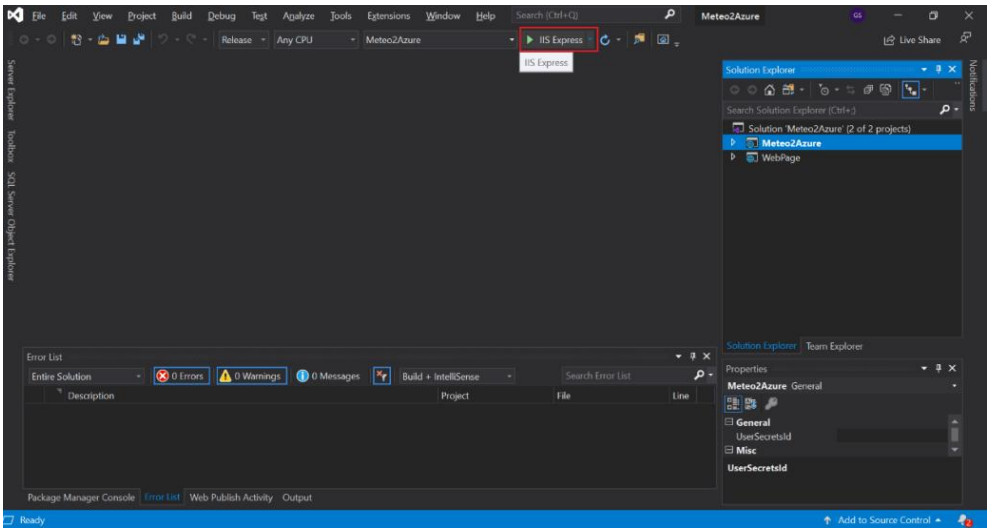
10.1 API και WEB application debugging

Προκειμένου να κάνουμε debugging στην API εφαρμογή, επιλέγουμε με δεξί κλικ στο solution explorer το project Meteo2Azure και μετά επιλέγουμε **Set as Startup Project** (Εικόνα 73).



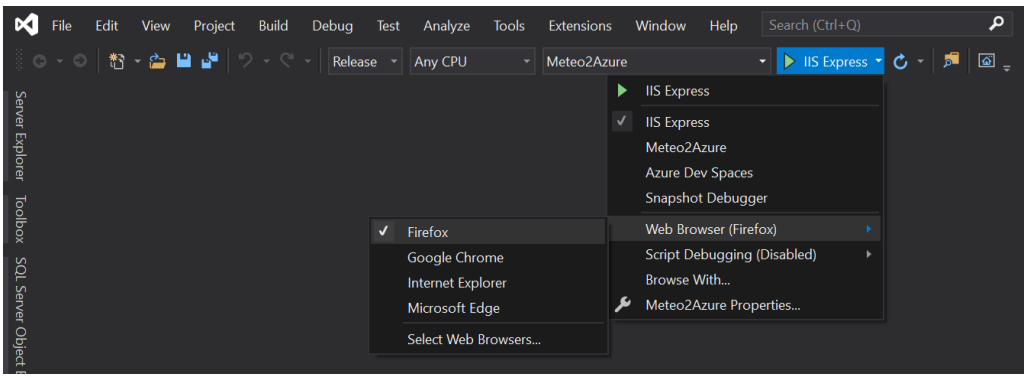
Εικόνα 73 Debugging

Επειδή η εφαρμογή API είναι ουσιαστικά ένα Web Service, για να εκτελεστεί η εφαρμογή χρειάζεται έναν Web Server. Για να εκτελεστεί τοπικά, εκτελείται πρώτα ο IIS Express ο οποίος είναι μία μικρή (Lite) έκδοση του IIS Server της Microsoft. Αμέσως μετά μέσω του IIS Express θα εκτελεστεί η εφαρμογή μας. Για να ξεκινήσει η εκτέλεση, πατάμε το button IIS Express στο menu του Visual Studio (Εικόνα 74).



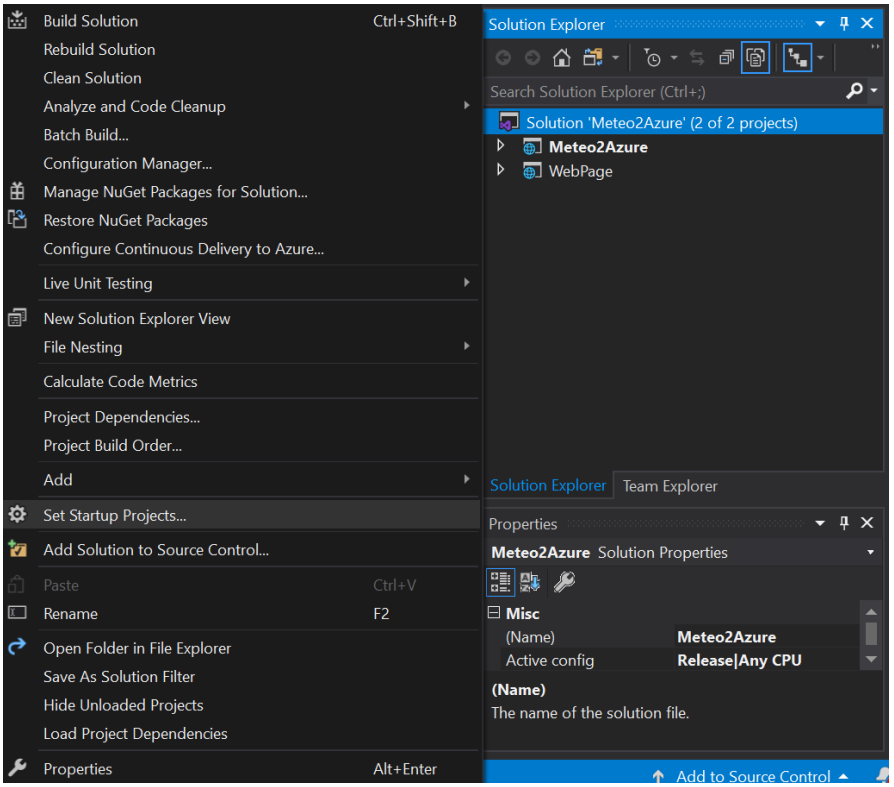
Εικόνα 74 Run IIS Express

Μόλις η εφαρμογή μας αρχίσει να εκτελείται, θα ανοίξει αυτόματα ένας browser για να μπορούμε αν θέλουμε να στείλουμε κάποια HTTP GET εντολή κλπ. Μπορούμε να ορίσουμε εμείς ποιος browser θα ανοίξει, πατώντας πάνω στο IIS Express button το βελάκι δεξιά, επιλέγοντας μετά **Web Browser** και μετά τον browser που θέλουμε (Εικόνα 75).



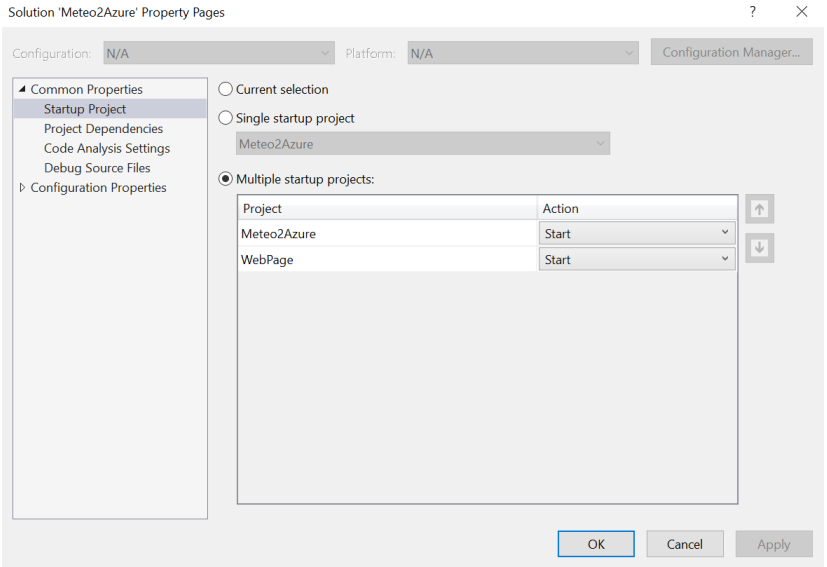
Εικόνα 75 Browser View

Για το debugging της WEB εφαρμογής, η διαδικασία είναι η ίδια με την API εφαρμογή, με τη διαφορά ότι, επειδή η WEB εφαρμογή έχει πρόσβαση στη database η οποία ανήκει στην API εφαρμογή, πρέπει να εκτελούνται και οι δύο εφαρμογές ταυτόχρονα. Για να γίνει αυτό πρέπει να επιλέξουμε σαν Startup project και τα δύο. Για να γίνει αυτό, επιλέγουμε με δεξί κλικ το Solution στο Solution Explorer, και μετά επιλέγουμε **Set Startup Projects** (Εικόνα 76).



Εικόνα 76 Multitasking

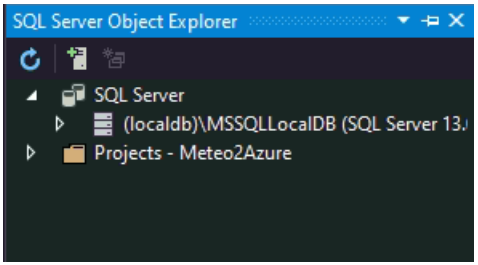
Στο παράθυρο που θα ανοίξει (Εικόνα 77), επιλέγουμε **Multiple startup projects** και στη συνέχεια στον πίνακα κάτω από την επιλογή, στη στήλη **Action** επιλέγουμε **Start** και για τα δύο projects (Meteo2Azure και WebPage).



Εικόνα 77 Multiple startup projects

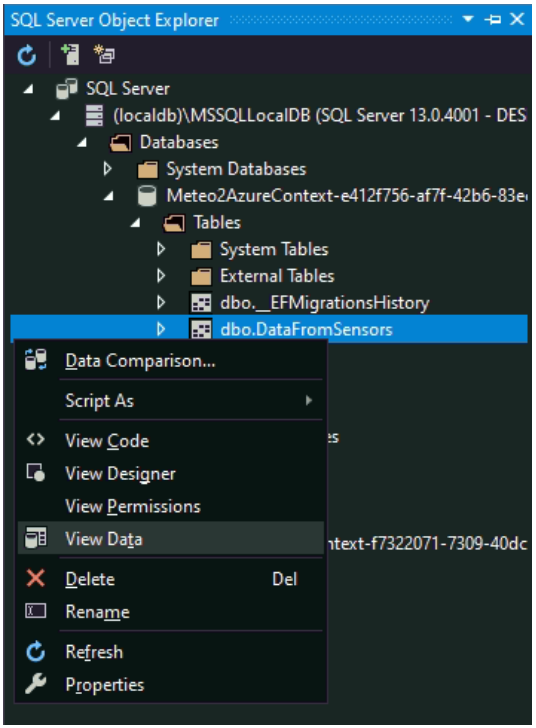
10.2 Database local management

Στη φάση της ανάπτυξης της εφαρμογής, μπορούμε να έχουμε πρόσβαση στην local database μέσα από το Visual Studio. Από το menu του Visual Studio επιλέγουμε **View->SQL Server Object Explorer** (Εικόνα 78).



Εικόνα 78 Database Local Managment

Στη συνέχεια επιλέγουμε **SQL Server**, μετά **(localdb)\MSSQLLocalDB**, μετά **Databases** και στη συνέχεια επιλέγουμε **Meteo2AzureContext** η οποία είναι η βάση μας τοπικά στον υπολογιστή μας. Στη συνέχεια επιλέγουμε **Tables** και θα εμφανιστούν οι πίνακες περιέχει η βάση μας. Επιλέγουμε τον **dbo.DataFromSensors** (Εικόνα 79) ο οποίος είναι ο πίνακας με τις μετρήσεις μας.



Εικόνα 79 View dbo.DataFromSensors

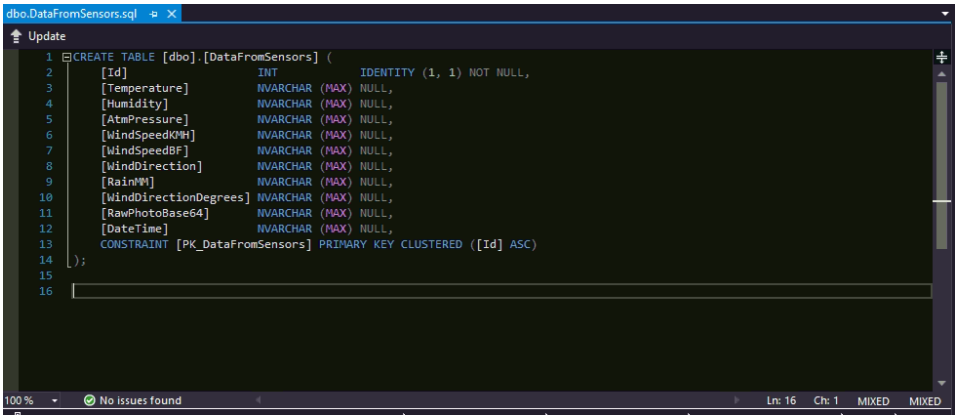
Με δεξί κλικ πάνω στον πίνακα μπορούμε να επιλέξουμε **View Data** για να δούμε τα δεδομένα του πίνακα (Εικόνα 80), **View Code** για να δούμε τον SQL κώδικα που δημιουργεί τον πίνακα (Εικόνα 81) και **View Designer** για να δούμε τον designer του πίνακα (Εικόνα 82).

dbo.DataFromSensors [Data]

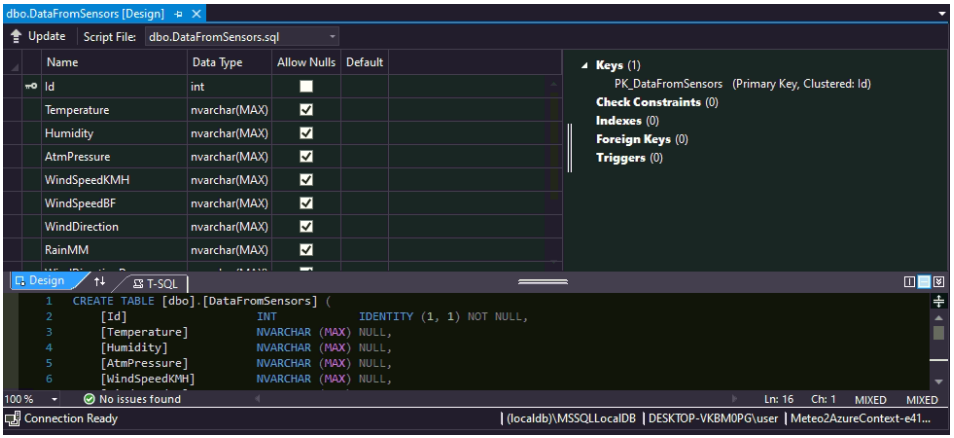
Max Rows: 1000

Id	Temperature	Humidity	AtmPressure	WindSpeedKMH	WindSpeedBF	WindDirection	RainMM	WindDirection...	R
1	1	2	3	6	7	4	8	5	9
1003	gspa	gspa	gspa	gspa	gspa	gspa	gspa	gspa	gs
2003	2020	Hum2020	Atm2020	KMH2020	BF2020	Dir2020	Rain2020	Deg2020	ph
2004	20	21	23	26	27	24	28	25	ph
2005	30	31	33	36	37	34	38	35	ph
2006	30	31	33	36	37	34	38	35	ph
2007	40	41	43	46	47	44	48	45	ph
2008	50	51	53	56	57	54	58	55	ph
2009	60	61	63	66	67	64	68	65	ph
2010	Tem60	Hum61	Pres63	KMH66	BF67	Dire64	Rain68	Degrees65	ph
2011	Tem_17	Hum_61	Pres1000	KMH_29	BF_6	Dire_NN	Rain_0	Degrees_9065	ph
2012	Tem_-10	Hum_50	Pres1020	KMH_10	BF_1	Dire_BB	Rain_0	Degrees_90	ph
2013	-10	50	1020	10	1	BB	0	90	ph
2014	9	35	92	13	3	AA	6	125	ph
2015	-3	85	99	50	5	S	2	3	ph
3004	Tem=-6	Hum=50	Presure=1000	KMH=40	BF=4	Directon=NW	Rain=1	Degrees=45	ph

Εικόνα 80 View Data



Εικόνα 81 View Code



Εικόνα 82 View Designer

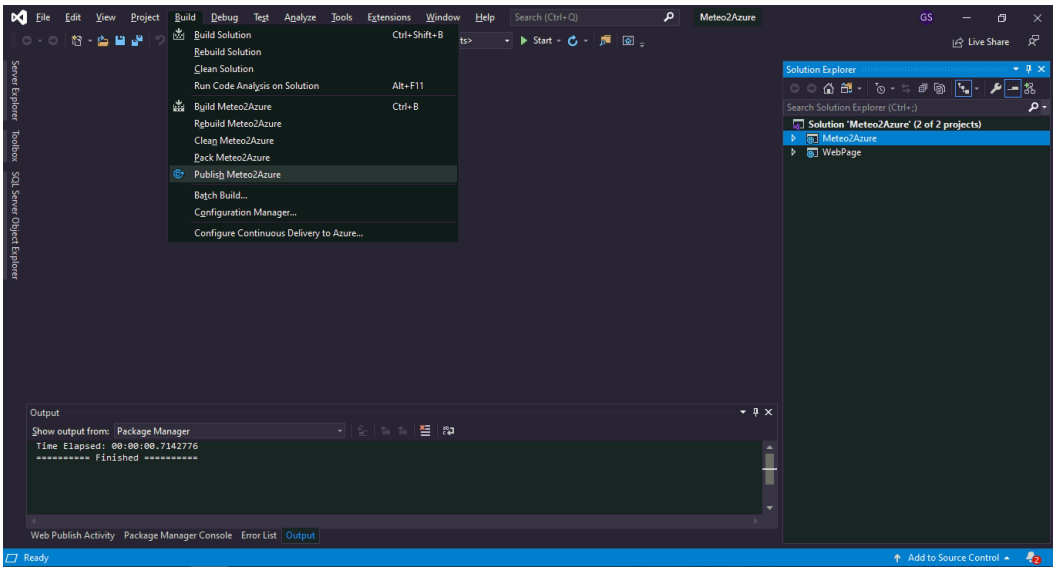
Από τον SQL Server Object Explorer μπορούμε να αποκτήσουμε πρόσβαση και σε απομακρυσμένες databases, πατώντας πάνω αριστερά στο παράθυρο **Add SQL Server**. Στη συνέχεια θα πρέπει να δώσουμε τη διεύθυνση του SQL Server και τα credentials για να συνδεθούμε.

10.3 Application deployment

Όταν ολοκληρωθεί η ανάπτυξη και το debugging μπορούμε να κάνουμε deploy τις δύο ASP .NET εφαρμογές στο Azure.

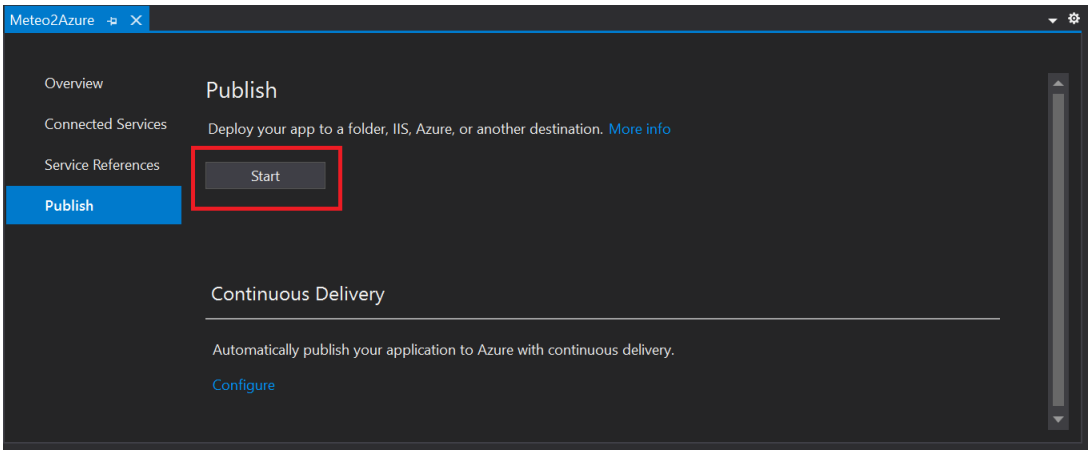
10.3.1 API app deployment

Επιλέγουμε πρώτα το Meteo2Azure project (API app) και στο menu **Build** επιλέγουμε **Publish** (Εικόνα 83)**.**



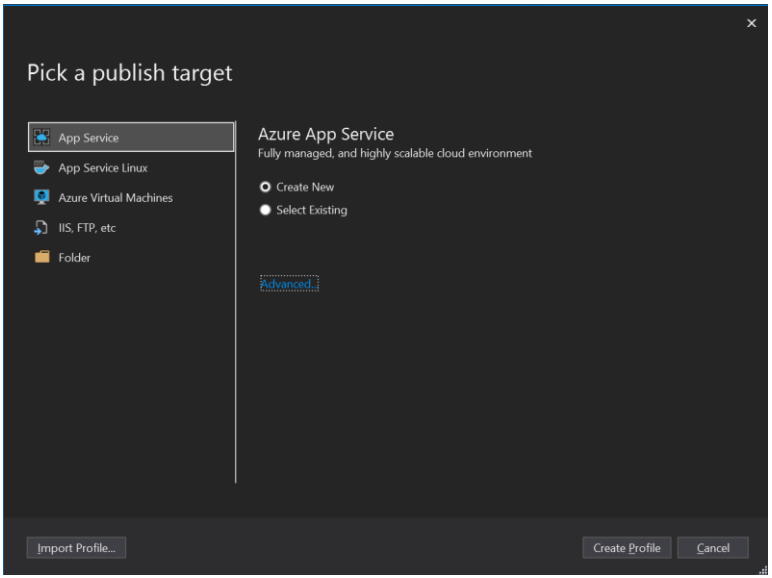
Εικόνα 83 Publish MeteoAzure

Στη συνέχεια επιλέγουμε **Publish** και μετά **Start** (Εικόνα 84).



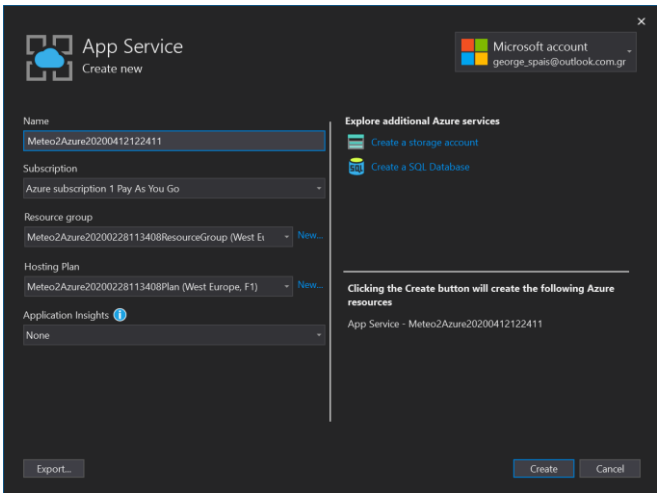
Εικόνα 84 Start Publish MeteoAzure

Στη επόμενη οθόνη (Εικόνα 85) παρακάτω, επιλέγουμε **App Service**, **Create New** και μετά **Create Profile**.

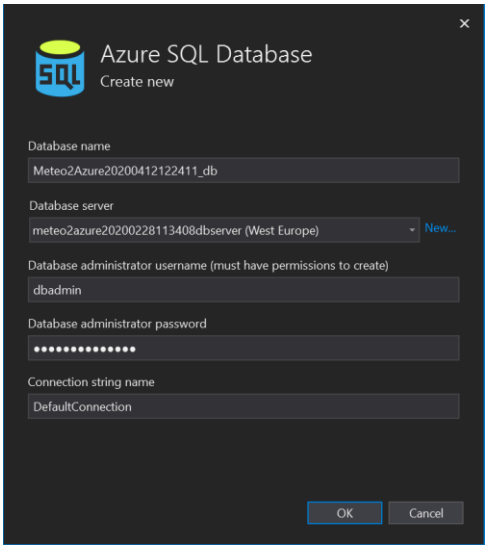


Εικόνα 85 Create Profile IIS

Μόλις πατήσουμε **Create Profile**, το Visual Studio θα ζητήσει τα credentials του Azure account μας. Μόλις τα δώσουμε, στην παρακάτω οθόνη (Εικόνα 86) δίνουμε τα στοιχεία το service. Αφήνουμε το default name και στα **Resource Group** και **Hosting Plan** επιλέγουμε κατά προτίμηση *Western Europe* και *Free Plan (F1)*. Στη συνέχεια δεξιά επιλέγουμε **Create a SQL Database**. Στο παράθυρο που ανοίγει δίνουμε τα στοιχεία που αφορούν την database. Μόλις ολοκληρώσουμε την καταχώρηση των στοιχείων για την database.



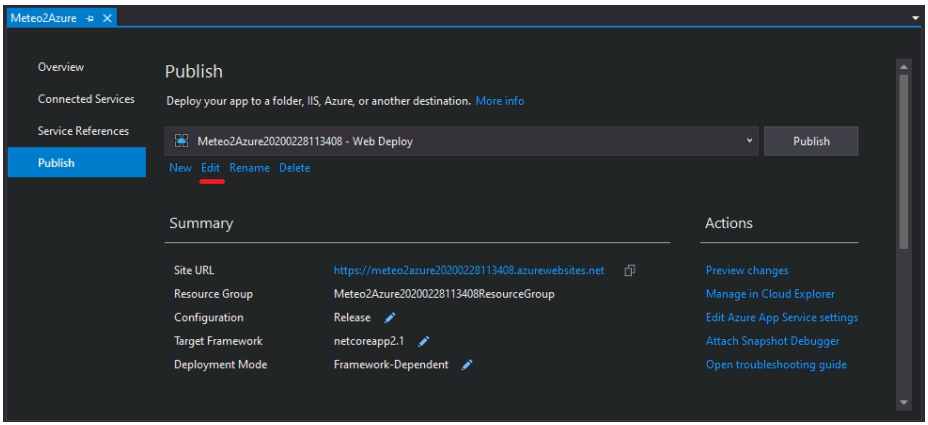
Εικόνα 86 Azure App Service settings



Εικόνα 87 Azure SQL Database settings

Μετά τις παραπάνω ρυθμίσεις, πατάμε **OK** στην οθόνη της SQL και **Create** στην προηγούμενη οθόνη του App Service. Το Visual Studio θα δημιουργήσει το app service και την database στο Azure.

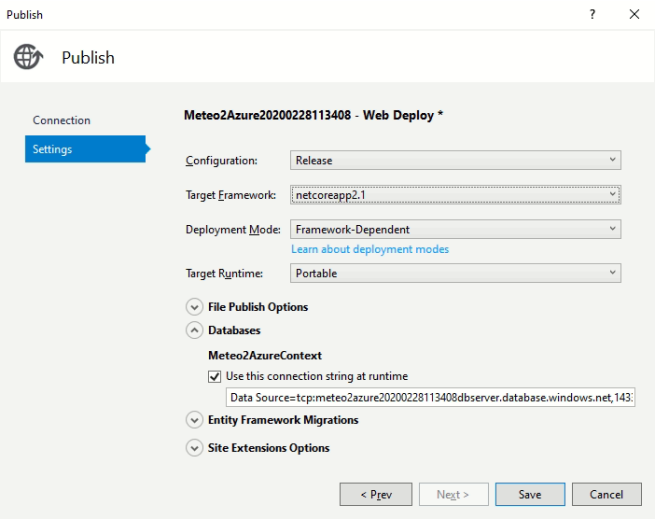
Στην συνέχεια επιλέγουμε ξανά Publish και μετά κάτω από το όνομα του service επιλέγουμε **Edit** (Εικόνα 88).



Εικόνα 88 App Service

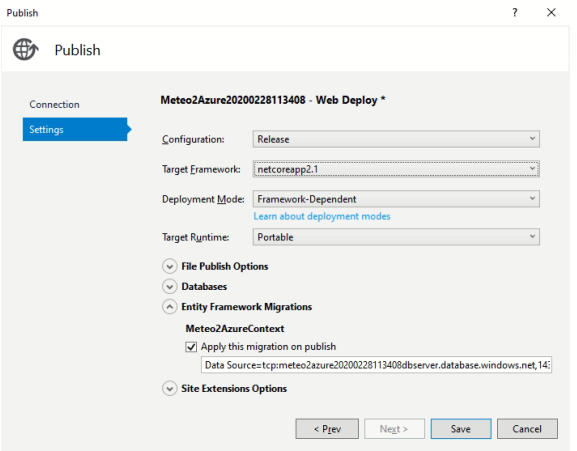
Στο παράθυρο που θα ανοίξει βλέπουμε τα connection settings του service, όπως τον server, το site name, τα username και password και το destination URL. Υπάρχει επίσης ένα **Validate Connection** button, με το οποίο μπορούμε να ελέγξουμε τη σύνδεση με το service στο Azure.

Στη συνέχεια επιλέγουμε αριστερά **Settings** και μετά τη ρύθμιση **Databases**. Ελέγχουμε ότι το connection string αναφέρεται στη διεύθυνση της SQL database στο Azure και ενεργοποιούμε το check box **Use this connection string at runtime** (Εικόνα 89).



Εικόνα 89 Web Deploy 1

Στη συνέχεια, επιλέγουμε ακριβώς από κάτω τη ρύθμιση **Entity Framework Migrations**, ελέγχουμε ότι για Data Source έχει τη διεύθυνση της SQL database στο Azure, και ενεργοποιούμε το check box **Apply this migration on publish** (Εικόνα 90).



Εικόνα 90 Web Deploy 2

Στη συνέχεια επιλέγουμε Save και κάνουμε ξανά publish το project. Μόλις ολοκληρωθεί το deployment, το service και ο SQL server εκτελούνται στο Azure και

αναμένουν requests από κάποιον client. Μπορούμε να κάνουμε δοκιμές, είτε με την εφαρμογή consumer που έχουμε φτιάξει, είτε με το POSTMAN όπως αναλύθηκε νωρίτερα.

10.3.2 WEB app deployment

Μόλις ολοκληρωθεί το deployment της API εφαρμογής, κάνουμε την ίδια διαδικασία για την WEB app.

Επιλέγουμε για **Resource Group** και για **Hosting** Plan τα ίδια με την API εφαρμογή.

Στα Publish settings επιλέγουμε για **Database** και για **Entity Framework Migrations** τα ίδια με την API εφαρμογή. Εάν είναι κενά, τα αντιγράφουμε από τα settings της API εφαρμογής. Αυτό γίνεται διότι, το data context και η database είναι κοινά μεταξύ των API και WEB εφαρμογών, όπως αναλύθηκε νωρίτερα.

Τέλος, επιλέγουμε **Save** και **Publish**.

Με την ολοκλήρωση του deployment και της WEB εφαρμογής, εκτελούνται στο Azure και τα δύο services (API και WEB). Επίσης εκτελείται και ο SQL Server με την database, στην οποία έχουν πρόσβαση και τα δύο services. Στο σημείο αυτό, η εγκατάσταση του cloud μέρους του project έχει ολοκληρωθεί και το Raspberry μπορεί πλέον να επικοινωνεί με το cloud API service για την αποστολή των μετρήσεων. Επίσης, ο χρήστης μπορεί από έναν browser της επιλογής του να βλέπει την ιστοσελίδα με τις μετρήσεις από το cloud WEB service.

Βιβλιογραφία

[1] Illustrated C# 7, 5th Edition, Daniel Solis and Cal Schrotenboer, Apress 2018

[2] ASP .NET Core in Action, Andrew Lock, Manning Publications Co. 2018

[3] Javascript in easy steps, 5th Edition, Mike McGrath, In Easy Steps Limited 2015

[4] Javascript and JQuery, 1st Edition, Jon Duckett, Wiley 2014

[5] Tutorial: Create your first ASP.NET Core App using Entity Framework with Visual Studio2019
https://docs.microsoft.com/en-us/visualstudio/get-started/csharp/tutorial-aspnet-core-ef-step-01?view=vs-2019

[6] Razor Pages with Entity Framework Core in ASP.NET Core - Tutorial 1 of 8
https://docs.microsoft.com/en-us/aspnet/core/data/ef-rp/intro?tabs=visual-studio&view=aspnetcore-3.1

[7] Setting up a Raspberry Pi
https://docs.microsoft.com/en-us/windows/iot-core/tutorials/rpi

[8] Call a Web API From a .NET Client (C#)
https://docs.microsoft.com/en-us/aspnet/web-api/overview/advanced/calling-a-web-api-from-a-net-client

[9] Code First to a New Database
https://docs.microsoft.com/en-us/ef/ef6/modeling/code-first/workflows/new-database?redirectedfrom=MSDN

[10] Entity Framework Code First and ASP.NET Web API
https://blogs.msdn.microsoft.com/jasonz/2012/07/23/my-favorite-features-entity-framework-code-first-and-asp-net-web-api/

[11] Create a REST API with Attribute Routing in ASP.NET Web API 2
https://docs.microsoft.com/en-us/aspnet/web-api/overview/web-api-routing-and-actions/create-a-rest-api-with-attribute-routing

[12] Attribute Routing in ASP.NET Web API 2
https://docs.microsoft.com/en-us/aspnet/web-api/overview/web-api-routing-and-actions/attribute-routing-in-web-api-2

[13] Action Results in Web API 2
https://docs.microsoft.com/en-us/aspnet/web-api/overview/getting-started-with-aspnet-web-api/action-results

[14] Tutorial: Build an ASP.NET app in Azure with SQL Database
https://docs.microsoft.com/en-us/azure/app-service/app-service-web-tutorial-dotnet-sqldatabase

[15] Tutorial: Migrate SQL Server to a single database or pooled database in Azure SQL Database offline using DMS
https://docs.microsoft.com/en-us/azure/dms/tutorial-sql-server-to-azure-sql

[16] Code First to a New Database
https://docs.microsoft.com/en-us/ef/ef6/modeling/code-first/workflows/new-database

[17] BME280Combined humidity and pressure sensor datasheet v1.3, Bosch Sensortec 2016

[18] MCP3422 18-Bit, Multi-Channel ΔΣ Analog-to-Digital Converter with I2C™ Interface and On-Board Reference datasheet, Microchip Technology Inc. 2009

[19] Weather Sensor Assembly p/n 80422 datasheet, Argent Data Systems

[20] Complete Raspberry Pi Weather Station
https://www.instructables.com/id/Complete-Raspberry-Pi-Weather-Station/

[21] Raspberry Pi Weather Station – Part 2
https://www.bc-robotics.com/tutorials/Raspberry-pi-weather-station-part-2/

[22] Build your own weather station
https://projects.Raspberrypi.org/en/projects/build-your-own-weather-station

[23] 10 Chart.js example charts to get you started
https://tobiasahlin.com/blog/chartjs-charts-to-get-you-started/#2-line-chart