ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

ΤΜΗΜΑ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Κατεύθυνση μεταπτυχιακού: Προηγμένα πληροφορικά συστήματα

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ανάπτυξη και ενορχήστρωση υπηρεσιών πληροφοριακών συστημάτων με χρήση serverless αρχιτεκτονικών

Υπεύθυνος καθηγητής: Κυριαζής Δημοσθένης

Ονοματεπώνυμο	**Αριθμός Μητρώου**
Μπατζιόπουλος Σπυρίδων	**ΜΕ1833**

1. Περιεχόμενα

1. ΠΕΡΙΕΧΟΜΕΝΑ **2**
2. ΕΙΣΑΓΩΓΗ **6**
3. ΑΡΧΙΤΕΚΤΟΝΙΚΕΣ ΚΑΙ ΠΑΡΟΧΟΙ ΥΠΗΡΕΣΙΩΝ **8**
3.1 ΑΝΑΔΡΟΜΙΚΗ ΑΝΑΦΟΡΑ 8
3.2 ΜΟΝΤΕΛΑ ΥΠΗΡΕΣΙΩΝ ΝΕΦΟΥΣ 10
3.2.1 Υποδομές ως υπηρεσία – Infrastructure as a service *12*
3.2.2 Κοντέινερ ως υπηρεσία - Container as a service *13*
3.2.3 Λειτουργικότητα ως υπηρεσία – Function as a service *13*
3.2.4 Πλατφόρμα ως υπηρεσία – Platform as a service *14*
3.2.5 Λογισμικό ως υπηρεσία – Software as a service *15*
3.3 ΣΤΟΧΟΙ ΤΗΣ ΕΡΓΑΣΙΑΣ 16
4. CONTAINERS ΚΑΙ DOCKER **17**
4.1 ΒΑΣΙΚΕΣ ΕΝΝΟΙΕΣ DOCKER ΚΑΙ CONTAINERS 17
4.2 CONTAINERS ΚΑΙ VIRTUAL MACHINES 19
4.3 ΕΡΓΑΛΕΙΑ ΕΓΚΑΤΑΣΤΑΣΗ ΚΑΙ ΛΕΙΤΟΥΡΓΙΑ 21
4.3.1 Εγκατάσταση docker *21*
4.3.2 Λειτουργία και βασικές εντολές *22*
4.3.3 Multi container εφαρμογές *24*
5. QUARKUS **27**
5.1 ΚΥΡΙΑ ΧΑΡΑΚΤΗΡΙΣΤΙΚΑ ΤΟΥ QUARKUS 27
5.2 ΑΝΑΠΤΥΞΗ ΕΦΑΡΜΟΓΩΝ ΜΕ QUARKUS 29
5.2.1 Jvm και native build *31*
6. APACHE OPENWHISK **34**
6.1 ΑΡΧΙΤΕΚΤΟΝΙΚΗ ΚΑΙ ΛΕΙΤΟΥΡΓΙΕΣ 34
6.1.1 Actions *35*
6.1.2 Triggers rules και events *36*
6.1.3 Key components *37*
6.2 ΕΓΚΑΤΑΣΤΑΣΗ OPENWHISK (LOCAL KUBERNETES) 39
7. KUBERNETES **45**
7.1 ΑΡΧΙΤΕΚΤΟΝΙΚΗ KUBERNETES 46
7.1.1 Δομικά στοιχεία Kubernetes server και worker *47*
7.2 ΕΓΚΑΤΑΣΤΑΣΗ LOCAL KUBERNETES CLUSTER (DOCKER FOR DESKTOP VERSION) 52
8. ΔΗΜΙΟΥΡΓΙΑ ΚΑΙ ΠΑΡΑΜΕΤΡΟΠΟΙΗΣΗ RPI KUBERNETES CLUSTER **56**
8.1 ΕΞΟΠΛΙΣΜΟΣ ΚΑΙ ΕΓΚΑΤΑΣΤΑΣΗ 56
8.1.1 Raspberry pi *56*
8.1.2 Εγκατάσταση και παραμετροποίηση Raspberry Pi OS : *57*
8.2 ΔΗΜΙΟΥΡΓΙΑ KUBERNETES CLUSTER 61
8.2.1 Δημιουργία server κόμβου *62*
8.2.2 Δημιουργία agent κόμβων *62*
8.2.3 Απομακρυσμένη πρόσβαση στο cluster *64*
8.3 CLOUDFLARE ΚΑΙ SSL SETUP 66
8.3.1 Dns proxy & port forward *66*
8.3.2 Let's Encrypt ssl certificates *67*
9. ΑΡΧΙΤΕΚΤΟΝΙΚΗ ΚΑΙ ΕΚΤΕΛΕΣΗ ΕΦΑΡΜΟΓΩΝ ΣΤΟ CLUSTER **69**

9.1 Αρχιτεκτονικη προσεγγιση 69
9.2 Quarkus Weather App 73
9.3 Slack Configuration 77
9.4 Openwhisk setup και demo 79

10. ΣΥΝΟΨΗ 81

11. ΒΙΒΛΙΟΓΡΑΦΙΑ 83

Ευχαριστίες

Ολοκληρώνοντας την εργασία θα ήθελα να εκφράσω τις θερμές μου ευχαριστίες στον υπεύθυνο και επιβλέποντας καθηγητή κ. Κυριαζή Δημοσθένη για την βοήθεια την υπομονή και την καθοδήγηση που μου προσέφερε καθ' όλο το διάστημα της εκπόνησης της διπλωματικής εργασίας.

Οι συμπληρώσεις, τα σχόλια και οι συμβουλές του συνέβαλαν να καταφέρω να ολοκληρώσω την εργασία.

2. Εισαγωγή

Την τελευταία δεκαετία, ένας από τους τομείς ο οποίος βίωσε την μεγαλύτερη ανάπτυξη είναι αυτός της τεχνολογίας. Πολλές επιχειρήσεις επένδυσαν στον εκσυγχρονισμό των υπηρεσιών τους αποδίδοντας στους καταναλωτές περισσότερο ποιοτικά προϊόντα με χαμηλότερο κόστος.

Μεγάλο μέρος των αλλαγών της δεκαετίας έγιναν στον τομέα του λογισμικού και των διαδικτυακών υπηρεσιών σε μια προσπάθεια να εναρμονιστούν με την έννοια του «ψηφιακού μετασχηματισμού». Οι αλλαγές αυτές επηρέασαν σε μεγάλο βαθμό την καθημερινότητα όλων των εμπλεκομένων στον κύκλο ζωής της παραγωγής λογισμικού.

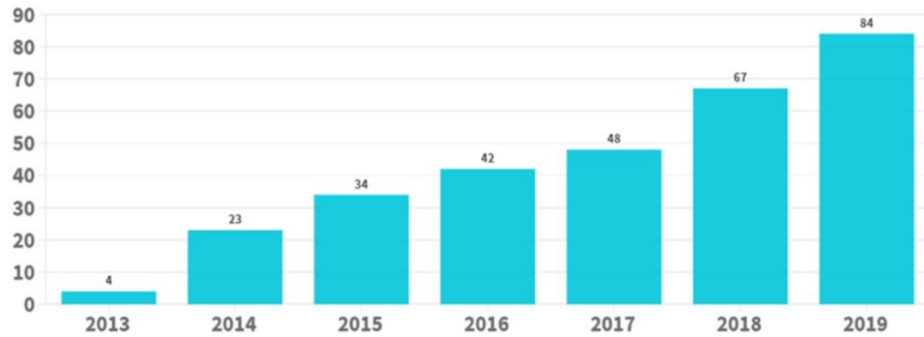
Δημιουργήθηκαν νέοι ρόλοι και επεκτάθηκαν οι ήδη υπάρχοντες καθ' όλη την διάρκεια της προσαρμογής αυτής. Ρόλοι όπως οι «dev ops» οι οποίοι δεν ήταν απαραίτητοι στο παρελθόν, έγιναν αναπόσπαστο κομμάτι της διαδικασίας ανάπτυξης λογισμικού και υπηρεσιών.

Ένας από τους κύριους λόγους αυτού του τεχνολογικού βήματος είναι η συνεχής αύξηση της ζήτησης νέων λύσεων οι οποίες φέρνουν κάθε εταιρία ή οργανισμό ένα βήμα πιο κοντά στον ψηφιακό μετασχηματισμό. Φυσικά μεγάλο ρόλο έπαιξε και η ανάγκη για μεγαλύτερη αξιοπιστία, επεκτασιμότητα, αποδοτικότητα και μείωση του κόστους.

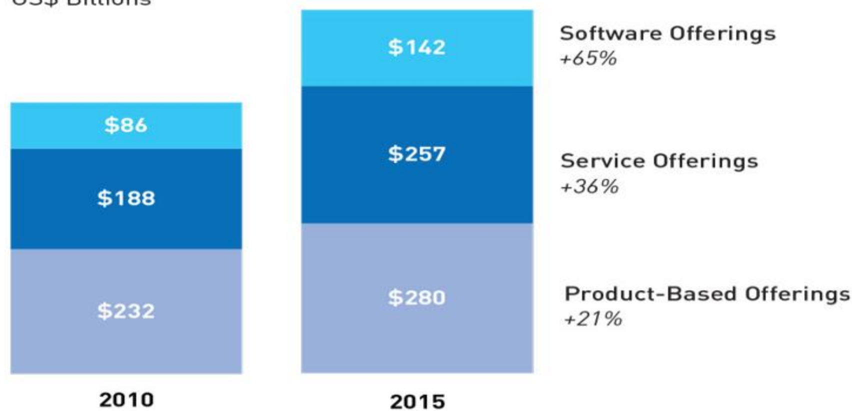
Τι είναι λοιπόν ο ψηφιακός μετασχηματισμός που πυροδότησε όλη αυτή την αλληλουχία και οδήγησε στην διαμόρφωση των προτύπων στην παραγωγή λογισμικού και υπηρεσιών;

Ως ψηφιακό μετασχηματισμό μπορούμε να ορίσουμε γενικά (καθώς διαφέρει ανά περίπτωση οργανισμού) την ενσωμάτωση ψηφιακών τεχνολογιών σε όλους τους τομείς μιας επιχείρησης / οργανισμού με αποτέλεσμα την δημιουργία υπεραξίας στον τρόπο που αυτές παρέχουν υπηρεσίες στους καταναλωτές. Πρόδρομος αυτού του μετασχηματισμού μπορούμε να υποθέσουμε ότι ήταν όλες οι νεοφυείς επιχειρήσεις, καθώς ήταν ευκολότερο να προσαρμοστούν στις αλλαγές έχοντας μικρότερο κόστος σε κάθε περίπτωση αποτυχίας του εγχειρήματος, οι οποίες κλήθηκαν να χτίσουν ανταγωνιστικές νέες υπηρεσίες.

Στα γραφήματα που ακολουθούν από την «PWC» και το «startupgenome» μπορούμε να δούμε αρχικά την διάθεση των επενδυτών να αυξήσουν τις χρηματοδοτήσεις σε startup οι οποίες δημιουργούν καινούργιους δρόμους μέσω της έρευνας και ανάπτυξης και αφετέρου την μεγάλη αύξηση στην ζήτηση νέου λογισμικού και υπηρεσιών ως απόρροια του ψηφιακού μετασχηματισμού που αναφέραμε:



2.1 Startup raised funds over year



2.2 2016 Global Innovation 1000

3. Αρχιτεκτονικές και πάροχοι υπηρεσιών

3.1 Αναδρομική αναφορά

Για να γίνει κατανοητή η συνολική εικόνα της εξέλιξης του τρόπου φιλοξενίας αλλά και παροχής υπολογιστικών πόρων θα κάνουμε επιγραμματικά μια αναδρομή στους κυριότερους σταθμούς της μέχρι σήμερα πορείας της.

Μέχρι και την προηγούμενη δεκαετία τον υπολογιστικό φόρτο των διεργασιών διαφόρων εφαρμογών και υπηρεσιών (όπως και την φιλοξενία αυτών) επωμίζονταν φυσικά μηχανήματα (server) οι οποίοι σε μερικές περιπτώσεις ήταν απλώς workstation μηχανήματα ενώ σε κάποιες άλλες περιπτώσεις ήταν dedicated server μηχανήματα.

Στην συνέχεια, την εμφάνιση της έκανε η δυνατότητα «εικονικοποίησης» - virtualization σύμφωνα με την οποία πάνω σε ένα φυσικό μηχάνημα δημιουργούσαν περισσότερα εικονικά μηχανήματα τα οποία μοιράζονταν όλους τους πόρους του φυσικού μηχανήματος είτε κατ αναλογία είτε στις περισσότερες περιπτώσεις ανάλογα τις ρυθμίσεις που είχαν δεχθεί.

Με τον τρόπο αυτό πέτυχαν να μειώσουν το κόστος της φιλοξενίας εφαρμογών και δεδομένων καθώς μπορούσαν να μοιράσουν κατ ανάγκη τους πόρους του φυσικού μηχανήματος στα διαφορετικά εικονικά που είχαν ρυθμιστεί να λειτουργούν πάνω από αυτό.

Το σημείο αυτό χρονικά αποτέλεσε το σημείο όπου αυτό που προσδιορίζουμε τον όρο νεφουπολογιστική αρχίζει να κάνει την εμφάνιση του με περισσότερο δομημένο τρόπο.
Τα μηχανήματα αυτά επιμερίζονταν τον φόρτο όλων των διεργασιών στους επεξεργαστές τους αλλά και ως προς την αποθήκευση δεδομένων και αρχείων σε βάσεις δεδομένων αποθηκευτικούς χώρους (storage).

Τον Αύγουστο του 2006 η Amazon ίδρυσε την θυγατρική της «Amazon Web Services» και παρουσίασε μέσω αυτής το «Elastic Compute Cloud». Η συγκεκριμένη υπηρεσία παρείχε με τη μορφή ενοικίασης εικονικές μηχανές στις οποίες ο καταναλωτής δύναται να τρέξει τις εφαρμογές του.

Τον Απρίλιο του 2008 η Google παρουσιάζει σε δοκιμαστική μορφή το «Google App Engine» δίνοντας ως τίτλο της λειτουργίας του ότι είναι μια πλατφόρμα νεφουπολογιστικής και ένα πλαίσιο διαδικτύου το οποίο προορίζονταν για την δημιουργία και φιλοξενίας εφαρμογών διαδικτύου οι οποίες θα φιλοξενούνταν στις εγκαταστάσεις (data center) της ίδιας της εταιρίας (Google).

Στις αρχές του 2008 επίσης δημιουργήθηκε το 1ο λογισμικό ανοιχτού κώδικα για την εγκατάσταση και παραμετροποίηση ιδιωτικών και υβριδικών δικτύων νέφους. Το όνομα του λογισμικού ήταν το «OpenNebula» της NASA το οποίο και είχε χρηματοδοτηθεί από ευρωπαϊκό πρόγραμμα.

Τον Φεβρουάριο του 2010 είναι η σειρά της Microsoft να παρουσιάσει την δική της λύση στην νεφουπολογιστική, και το όνομα του project της είναι Azure και αποτελεί ως και σήμερα έναν από τους βασικούς συναγωνιστές στο ράλι επιδόσεων και καινοτομίας στον κόσμο της νεφουπολογιστικής.

Τον Ιούλιο του 2010 η συνεργασία της NASA με την Rackspace γεννά το OpenStack.O ρόλος του OpenStack είναι να βοηθήσει τις εταιρίες και οργανισμούς που διέθεταν υπηρεσίες νεφουπολογιστικής πάνω σε προτυποποιημένο υλικό.
Τον Μάρτιο του 2011 η IBM ανακοινώνει το SmartCloud πλαίσιο για να υποστηρίζει το Smart Planet project.

Τον Μάιο του 2012 η Google παρουσιάζει την δοκιμαστική έκδοση της δικής της υπηρεσίας νέφους με την ονομασία Google Compute Engine η οποία διατέθηκε στην αγορά σε πλήρως παραγωγική μορφή τον Δεκέμβριο του 2013.
Τον Ιούνιο του 2012 η Oracle έρχεται στο προσκήνιο με την διάθεση του Oracle Cloud. Είναι η πρώτη προσπάθεια που γίνεται ώστε οι χρήστες του να έχουν πρόσβαση σε διαφορετικά επίπεδα μέσω της ενσωμάτωσης μιας σειράς υπηρεσιών. Οι υπηρεσίες αυτές είναι: «Λογισμικό ως υπηρεσία (Software as a Service)», «Πλατφόρμα ως υπηρεσία (Platform as a Service)» και «Υποδομή ως υπηρεσία (Infrastructure as a Service)».

Το μεγαλύτερο μερίδιο της αγοράς αυτή την στιγμή κατέχουν Google, Microsoft και Amazon με την IBM και Oracle να συμπληρώνουν την σειρά προτίμησης στην αγορά της υπολογιστικής νέφους.

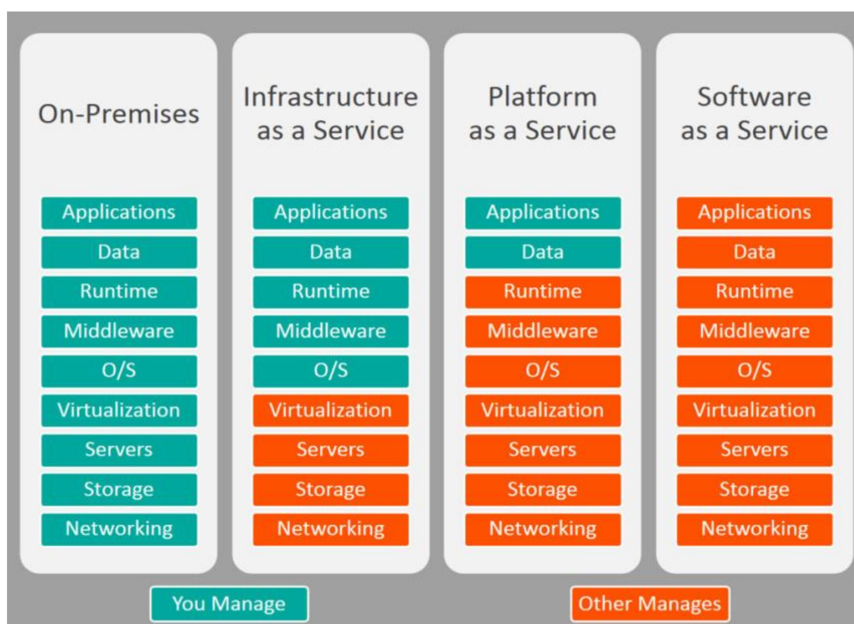
Όπως αναφέραμε τα μοντέλα που προσφέρονται ως υπηρεσίες νέφους, στην συντριπτική πλειοψηφία τους, ανήκουν σε μία από τις παρακάτω κατηγορίες :

- Υποδομή ως Υπηρεσία – Infrastructure as a Service (IaaS)
- Container as a Service (CaaS)
- Πλατφόρμα ως Υπηρεσία – Platform as a Service (PaaS)
- Λειτουργικότητα ως Υπηρεσίας – Function as a Service (FaaS)
- Λογισμικό ως Υπηρεσία – Software as a Service (SaaS)

Στην συνέχεια θα αναλύσουμε τα μοντέλα αυτά καθώς και τι εξυπηρετεί καθένα από αυτά.

3.2 Μοντέλα υπηρεσιών νέφους

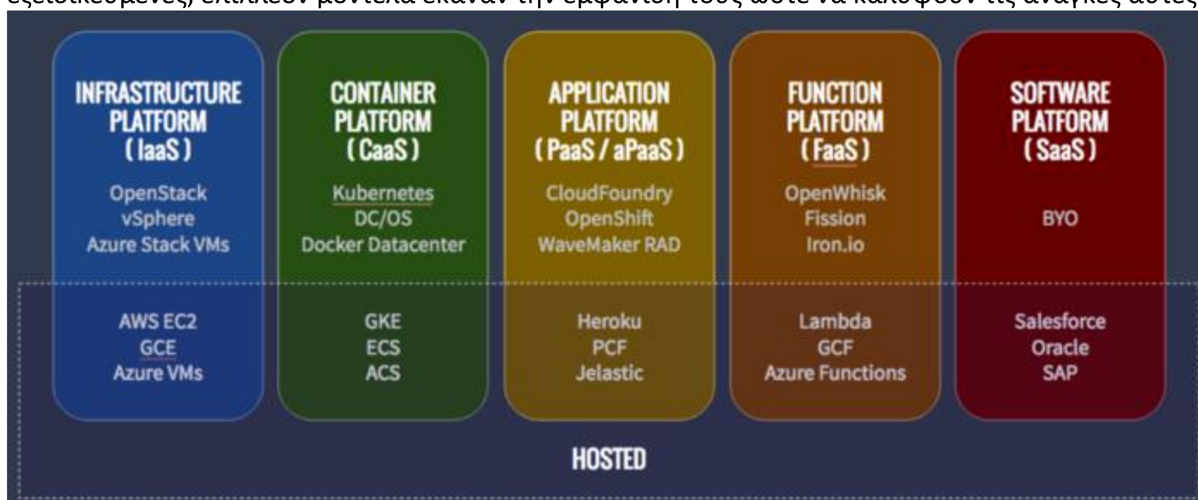
Στο παρακάτω σχήμα γίνεται μια σχετική αναφορά στις διαφοροποιήσεις των μοντέλων των υπηρεσιών νέφους σχετικά με τα επίπεδα τα οποία διαχειρίζεται ο χρηστής (πελάτης) και τα επίπεδα που διαχειρίζεται η υπηρεσία (cloud services provider).



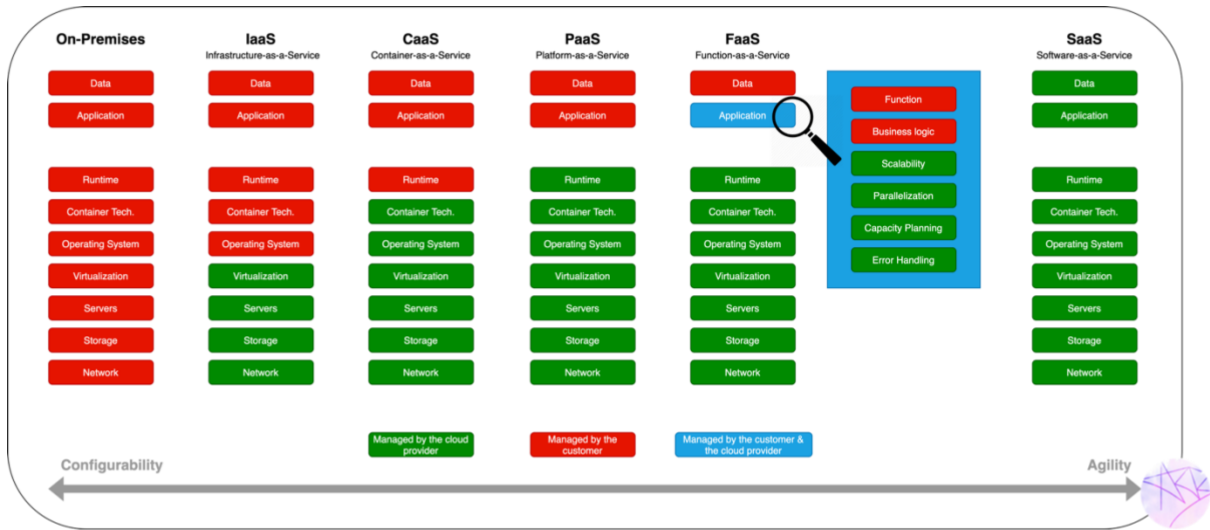
3.1 On premise to cloud services

Γίνεται επομένως αντιληπτό η διαφοροποίηση από μια εγκατάσταση στην οποία οι υποδομές βρίσκονται εντός της εταιρίας ή του χρήστη, σε μια εγκατάσταση σε πάροχο υπηρεσιών νέφους για παράδειγμα στο μοντέλο υποδομές ως υπηρεσία [1].

Η παραπάνω εικόνα αναφέρεται στα περισσότερο χρησιμοποιούμενα μοντέλα υπηρεσιών των παρόχων. Στην συνέχεια όμως, καθώς οι απαιτήσεις των χρηστών γίνονταν περισσότερο εξειδικευμένες, επιπλέον μοντέλα έκαναν την εμφάνιση τους ώστε να καλύψουν τις ανάγκες αυτές.



3.2 Cloud services examples



3.3 cloud OSI model explained

3.2.1 Υποδομές ως υπηρεσία – Infrastructure as a service

Οι υπηρεσίας υποδομής νέφους είναι δομημένες ώστε να παρέχουν υπολογιστικούς πόρους αυτοματοποιημένα και με κλιμακούμενα.

Το μοντέλο «υποδομές ως υπηρεσία» είναι άρρηκτα συνδεδεμένο με την παροχή πρόσβασης και την δυνατότητα επίβλεψης στοιχείων όπως οι υπολογιστικοί πόροι, τη δικτυακή υποδομή και λειτουργία, την αποθήκευση και άλλες υπηρεσίες.

Κύριο χαρακτηριστικό του μοντέλου είναι η δυνατότητα να παρέχει (διάθεση μέσω πώλησης) στον χρήστη – πελάτη τους απολύτους απαραίτητους πόρους που εξυπηρετούν τις ανάγκες του χωρίς να χρειάζεται να πληρώσει για το σύνολο της φυσικής υποδομής όπως σε μια φυσική εγκατάσταση (on premise).
Η «υποδομή ως υπηρεσία» παρέχει στον χρήστη υποδομές υπολογιστικών πόρων συμπεριλαμβανομένων και άλλων στοιχείων όπως εξυπηρετητές, λειτουργικά συστήματα, συστήματα αποθήκευσης (storage) μέσω τεχνολογίας εικονικοποίησης (virtualization).Στις περισσότερες περιπτώσεις οι εξυπηρετητές νέφους ελέγχονται από τον χρήστη – πελάτη μέσω ενός διαχειριστικού περιβάλλοντος ή ενός τυποποιημένου πλαισίου ανάπτυξης και υλοποίησης (API).

Οι χρήστες του μοντέλου «υποδομή ως υπηρεσία» συνεχίζουν να έχουμε άμεση πρόσβαση στους εξυπηρετητές και τα αποθηκευτικά μέσα όπως σε μία παραδοσιακή εγκατάσταση με την πολύ σημαντική διαφορά ότι οι πόροι αυτοί του διατίθενται μέσω εικονικών κέντρων δεδομένων (data center).

Σε αντίθεση με τα άλλα 2 μοντέλα που προσφέρουν οι υπηρεσίες νέφους, «πλατφόρμα ως υπηρεσία» και «λογισμικό ως υπηρεσία», ο χρήστης – πελάτης είναι υπεύθυνος για την διαχείριση και παραμετροποίηση στοιχείων όπως εφαρμογές, περιβάλλοντα εκτέλεσης εφαρμογών, λειτουργικά συστήματα, ενδιάμεσο λογισμικό και δομών αποθήκευσης.

Στην αντίθετη πλευρά ο πάροχος της υπηρεσίας φροντίζει για την διαχείριση των εξυπηρετητών, των δικτύων, των φυσικών μέσων αποθήκευσης και την εικονικοποίηση των μηχανημάτων. Σε ορισμένες περιπτώσεις παρέχονται επίσης περισσότερες υπηρεσίες εκτός του περιβάλλοντος εικονικοποιησης όπως εξωτερικές βάσεις δεδομένων.

Πλεονεκτήματα του μοντέλου «υποδομές ως υπηρεσία» :

- Είναι το πιο ευέλικτο μοντέλο παροχής υπολογιστικών πόρων νέφους.
- Επιτρέπει την εύκολη επέκταση και εγκατάσταση στοιχείων όπως αποθηκευτικός χώρος, επεξεργαστική ισχύ, εξυπηρετητών και δικτύων
- Δυνατότητα αγοράς των φυσικών μηχανημάτων (dedicated hardware) αναλόγως των αναγκών και της χρήσης των διαθέσιμων πόρων.
- Δίνει στον χρήστη καθολικό έλεγχο της υποδομής.
- Οι πόροι μπορούν επίσης να εξαγοραστούν από τον χρήστη (dedicated resource consumption) αναλόγως των αναγκών του.
- Είναι υπηρεσία που παραμετροποιείται και επεκτείνεται με εύκολο τρόπο από την πλευρά του παρόχου.

Καταλήγουμε στο συμπέρασμα ότι το συγκεκριμένο μοντέλο υπηρεσίας νέφους αποτελεί μια συμφέρουσα λύση για χρήστες οι οποίοι θέλουν να μεταφέρουν το βάρος της διαχείρισης των

φυσικών μηχανημάτων εκτός της δικής τους εποπτείας καθώς και να αποφύγουν την αγορά ακριβών και ενεργειακά απαιτητικών μηχανημάτων επομένως στρέφονται προς μια λογική της δυναμικής οριοθέτησης και κοστολόγησης των πόρων που πραγματικά χρειάζονται για να εξυπηρετήσουν τις ανάγκες τους [1].

3.2.2 Κοντέινερ ως υπηρεσία - Container as a service

Το «container as a service», είναι ένα μοντέλο υπηρεσιών νέφους που επιτρέπει στους χρήστες να χρησιμοποιήσουν containers για να διαχειριστούν τις εφαρμογές τους. Το συγκεκριμένο μοντέλο προσφέρει στους χρήστες την δυνατότητα να οργανώσουν τις εφαρμογές τους μέσα σε container ή και container clusters και με αυτό τον τρόπο να έχουν περισσότερο δομημένη διαχείριση τους.

Οι υπηρεσίες του μοντέλου αυτού παρέχεται είτε μέσω virtualization είτε μέσω API είτε μέσω web interface σε κάποιο portal του παρόχου (πχ IBM, Microsoft Azure κλπ).

Πλεονεκτήματα μοντέλου «Container as a service»

- Υψηλή μεταφερσιμότητα (οι εφαρμογές δεν εξαρτώνται από το λειτουργικό σύστημα ούτε τους φυσικούς πόρους και μπορούν να μεταφερθούν σε οποιοδήποτε μηχάνημα είτε υπηρεσία νέφους).
- Ασφάλεια, καθώς οι εφαρμογές που τρέχουν εντός των container είναι απομονωμένες από τους υπόλοιπους containers αλλά ακόμη και το host μηχάνημα που τους φιλοξενεί.
- Εύκολη επεκτασιμότητα καθώς οι containers μπορούν να γίνουν «scale» με πολύ απλό τρόπο

Την γρήγορη επέκταση του συγκεκριμένου μοντέλου βοήθησε καταλυτικά η αύξηση στην χρήση εργαλείων όπως είναι το docker αλλά και συστήματα «orchestration» όπως είναι το Kubernetes για τα οποία θα αναφερθούμε εκτενώς στην συνέχεια της εργασίας.

3.2.3 Λειτουργικότητα ως υπηρεσία – Function as a service

Το συγκεκριμένο μοντέλο έχει πάρει επίσης μεγάλη έκταση καθώς επιτρέπει στους προγραμματιστές να τρέχουν συγκεκριμένα «block» κώδικα χωρίς να τους απασχολούν οι υπολογιστικοί πόροι που απαιτούνται όπως στα υπόλοιπα μοντέλα υπηρεσιών νέφους[2].

Υπάρχουν αρκετές ομοιότητες με τον μοντέλο «Platform as a service» με βασική διαφορά όμως ότι εδώ μπορεί να τρέξουν κυρίως μικρές εφαρμογές που κατά κύριο λόγο αποτελούν ενέργειες που εκτελούνται όταν κάποιο συμβάν λαμβάνει χώρα.

Παραδείγματα τέτοιων υπηρεσιών είναι το Azure functions, το AWS Lamda και το Openwhisk το οποίο θα χρησιμοποιήσουμε για το τεχνικό σκέλος της εργασίας.

3.2.4 Πλατφόρμα ως υπηρεσία – Platform as a service

Το μοντέλο «πλατφόρμα ως υπηρεσία» προσφέρει στοιχεία νέφους σε συγκεκριμένο λογισμικό και χρησιμοποιείται κατά κύριο λόγο για ανάπτυξη λογισμικού και εφαρμογών
Παρέχει ένα συγκεκριμένο πλαίσιο ανάπτυξης, το οποίο στην πλειοψηφία των παρόχων υπηρεσιών νέφους δίνεται με την μορφή επιλογής ανάλογα με την γλώσσα προγραμματισμού και τις απαιτήσεις της εφαρμογής[1].

Οι προγραμματιστές με χρήση αυτού του πλαισίου αναπτύσσουν τις εφαρμογές τους χωρίς να χρειάζεται να γνωρίζουν ολόκληρο το μοντέλο της υποδομής ούτε να συντηρούν τα διάφορα επίπεδα αυτής που βρίσκονται χαμηλότερα από το περιβάλλον εκτέλεσης των εφαρμογών που έχουν επιλέξει (πχ. Application server).

Το μοντέλο παράδοσης της συγκεκριμένης υπηρεσίας νέφους είναι παρόμοια με αυτή που θα περιγράψουμε για το μοντέλο «λογισμικό ως υπηρεσία». Η διαφορά τους είναι ότι το παρόν μοντέλο (πλατφόρμα ως υπηρεσία) παρέχει μια πλατφόρμα στον προγραμματιστή μέσω του διαδικτύου η οποία διαχειρίζεται από αυτόν μέσω διαχειριστικού περιβάλλοντος είτε σε μερικές περιπτώσεις γραμμής εντολών (cli).

Με χρήση αυτής της υπηρεσίας νέφους ο χρήστης – πελάτης διευκολύνεται στον σχεδιασμό και ανάπτυξη εφαρμογών χρησιμοποιώντας ειδικά πλαίσια προγραμματισμού ανάλογα με τις απαιτήσεις και τις ανάγκες του. Προσφέρεται κοινώς ένα περιβάλλον που εγγυάται την επεκτασιμότητα του και την υψηλή διαθεσιμότητα όπως κάθε άλλη υπηρεσία νέφους.

Πλεονεκτήματα του μοντέλου «Πλατφόρμα ως υπηρεσία» :

- Μείωση του κόστους και της πολυπλοκότητας κατά την ανάπτυξη εφαρμογών
- Επεκτασιμότητα
- Υψηλή διαθεσιμότητα
- Σχεδιασμός και ανάπτυξη εφαρμογών από τους προγραμματιστές χωρίς την υποχρέωση συντήρησης της υποδομής που το φιλοξενεί
- Αυτοματοποίηση επιχειρησιακών διαδικασιών
- Εύκολη μετάβαση σε υβριδικό μοντέλο
- Ελαχιστοποίηση το όγκου εργασίας τους προγραμματιστή κατά τον κύκλο σχεδιασμού και ανάπτυξης των εφαρμογών του

Καταλήγουμε στο συμπέρασμα ότι σε κάποιες περιπτώσεις η χρήση της υπηρεσίας νέφους «Πλατφόρμα ως υπηρεσία» αποφέρει αρκετά πλεονεκτήματα και σε ορισμένες από αυτές είναι απαραίτητη.

Σε έργα στα οποία συνεργάζονται ομάδες προγραμματιστών και απαιτείται ένα πλαίσιο ώστε να καταφέρουν όλοι, στο σύνολο τους, να ολοκληρώσουν το κομμάτι που τους έχει ανατεθεί, η «πλατφόρμα ως υπηρεσία» αποτελεί εξαιρετική λύση καθώς προσφέρει ευελιξία και ταχύτητα καθ' όλη την διάρκεια της διαδικασίας.

Το μεγάλο σαφώς πλεονέκτημα που αποκομίζει ο χρήστης είναι η μείωση του κόστους και η ταχύτερη σχεδίαση και ανάπτυξη εφαρμογών.

3.2.5 Λογισμικό ως υπηρεσία – Software as a service

Το «λογισμικό ως υπηρεσία» είναι το πιο διαδεδομένο και ευρέως χρησιμοποιούμε μοντέλο υπηρεσιών νέφους. Η υπηρεσίας αυτή διαθέτει εφαρμογές σε χρήστες – πελάτες οι οποίες διαχειρίζονται από τον πάροχο[1].

Οι περισσότερες εφαρμογές του μοντέλου εκτελούνται στον περιηγητή του χρήστη – πελάτη χωρίς να χρειαστεί οποιαδήποτε εγκατάσταση στο δικό του μηχάνημα.

Ελαχιστοποιώντας επομένως την ανάγκη για τοπική εγκατάσταση των εφαρμογών που θα χρησιμοποιήσει ο χρήστης και η απεμπλοκή του από την αντιμετώπιση τεχνικών ζητημάτων, τα οποία βαραίνουν τον πάροχο της υπηρεσίας, δημιουργεί μια ιδιαίτερα ελκυστική λύση.

Πλεονεκτήματα του μοντέλου «Λογισμικό ως υπηρεσία» :

- Μείωση του χρόνου και του κόστους για εργασίες όπως εγκατάσταση, διαχείριση και αναβάθμιση του λογισμικού.
- Επαναπροσδιορισμός των καθηκόντων των τεχνικών ομάδων σε περισσότερο σημαντικά θέματα εντός μιας εταιρίας.
- Προσφορά κεντρικής διαχείρισης των εφαρμογών.
- Φιλοξενία εφαρμογών σε απομακρυσμένο διακομιστή.
- Προσβάσιμο μέσω διαδικτύου και του περιηγητή του χρήστη.

Το συμπέρασμα που αντλείται από την επεξήγηση του συγκεκριμένου μοντέλου είναι ότι οι λύσεις που δίνει, κυρίως σε μικρομεσαίες επιχειρήσεις είναι σημαντικές.

Ενδείκνυται να χρησιμοποιείται από χρήστες – εταιρίες οι οποίες ασχολούνται με βραχυπροθέσμα έργα και απαιτούν συνεργασία διαφορετικών μερών για την ολοκλήρωση τους. Ένα παράδειγμα που αντικατοπτρίζει την παραπάνω αναφορά είναι ένα σύστημα παρακολούθησης εργασιών από διαφορετικούς ανθρώπους και ρόλους στο οποίο παρέχεται πρόσβαση σε αυτούς μέσω του περιηγητή τους ή ακόμη και το κινητό τους τηλέφωνο.

3.3 Στόχοι της εργασίας

Βασικός στόχος της εργασίας είναι να εξεταστεί η χρήση τόσο των containers μέσω του docker, όσο και του kubernetes για να δημιουργήσουμε ένα περιβάλλον το οποίο να μπορεί να υποστηρίξει εφαρμογές serverless.

Θα γίνει μια επεξήγηση των όρων και των λειτουργικοτήτων, τεχνολογιών όπως το Docker, Kubernetes, Quarkus και Openwhisk ώστε να καταλήξουμε αν τελικά η serverless αρχιτεκτονική μας προσφέρει την απαραίτητη ελευθερία να δημιουργήσουμε μικρά block κώδικα τα οποία θα εκτελούνται κάτω από συγκεκριμένες συνθήκες.

Τέλος για το τεχνικό σκέλος θα δημιουργήσουμε ένα Kubernetes cluster με raspberry pi για να δείξουμε ότι ακόμη και με πολύ μικρή επεξεργαστική ισχύ είναι δυνατόν να κάνουμε scale και να εκτελέσουμε ακόμη και πολύπλοκες διεργασίες σε ένα serverless περιβάλλον.

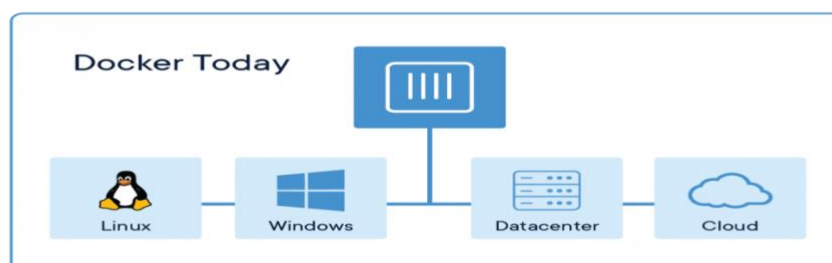
4. Containers και Docker

4.1 Βασικές έννοιες docker και containers

Το docker είναι μια τεχνολογία containers η οποία έγινε διαθέσιμη ως open source το 2013 ως Docker Engine.

Στόχος της τεχνολογίας αυτής είναι να μεγιστοποιήσει την αξία στην χρήση των containers σε διαφορετικά περιβάλλοντα και κυρίως στα Linux συστήματα που τους χρησιμοποιούσαν μέχρι την έλευση του docker μέσω cgroup και namespaces [6].

Στην συνέχεια η Microsoft σε συνεργασία με το docker δημιούργησε το docker for windows για να εκμεταλλευτεί την δυναμική και την ευελιξία στη χρήση του και έτσι το docker δύναται να τρέχει σε όλα τα λειτουργικά συστήματα.



4.1 Docker registry over different os and environments

Container είναι μια τυποποιημένη μονάδα λογισμικού η οποία ενσωματώνει μαζί με το κώδικα όλες τις εξαρτήσεις (dependencies) που χρειάζεται ώστε να λειτουργήσει εύκολα γρήγορα και αξιόπιστα σε οποιοδήποτε υπολογιστικό περιβάλλον (Linux / Windows / Mac Os).

Η εικόνα ενός docker container (docker container image) είναι ένα αυτόνομο, ανεξάρτητο και εύχρηστο εκτελέσιμο πακέτο λογισμικού το οποίο περιλαμβάνει όλες τις εξαρτήσεις και τα εργαλεία που απαιτεί το λογισμικό για να λειτουργήσει όπως βιβλιοθήκες, εκτελέσιμο περιβάλλον, κώδικα και παραμετροποιήσιμα αρχεία.

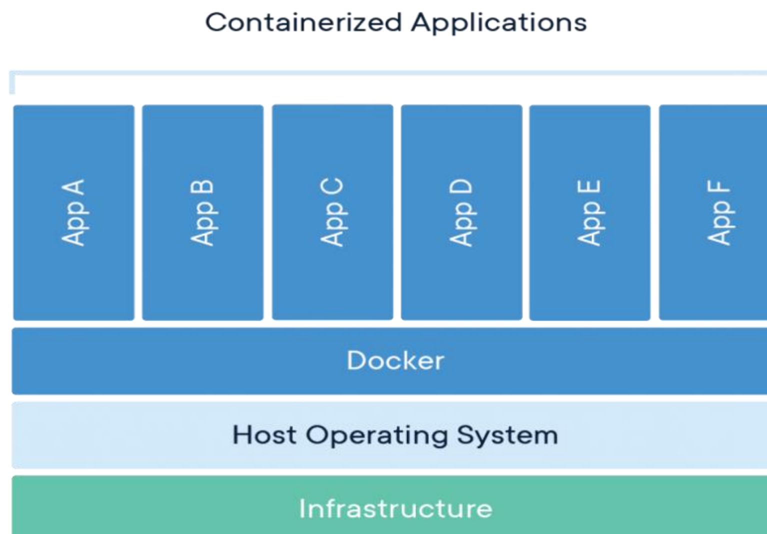
Οι εικόνες που δημιουργούνται μέσω του docker μετατρέπονται σε εκτελέσιμους containers μέσω της μηχανής docker (docker engine), κάτι το οποίο εξασφαλίζει την ανεξαρτησία των εφαρμογών

που τρέχουν από το υπολογιστικό περιβάλλον. Κατά την δημιουργία του, ένας docker container τρέχει απομονωμένος από το λειτουργικό σύστημα που τον φιλοξενεί και έχει την ίδια συμπεριφορά (εγκατάσταση, λειτουργία) ανεξαρτήτως αυτού.

Παρόλο που οι containers σαν έννοια, υπήρχαν στον χώρο της πληροφορικής αρκετά χρόνια πριν την εμφάνιση του docker, αυτό δεν ήταν αρκετό ώστε να πάρουν αρκετή προσοχή από τους προγραμματιστές, καθώς τα περισσότερα προβλήματα της εικονικοποίησης αλλά και της διαχείρισης πόρων τα επέλυαν οι εικονικές μηχανές (virtual machines).

Με την έλευση του docker οι containers πήραν μεγαλύτερη προσοχή και έγιναν ο τυποποιημένος τρόπος που χρησιμοποιούνται έκτοτε οι containers. Αυτό που προσφέρουν οι containers που τρέχουν πάνω σε docker μηχανή είναι:

- Τυποποίηση καθώς το docker engine έχει οριστεί ως η κύρια μηχανή για containers επομένως έχουν εύκολη και αξιόπιστη μεταφερσιμότητα.
- Καταναλώνουν τον ελάχιστο δυνατό χώρο και τους ελάχιστους δυνατούς πόρους καθώς συνεργάζονται με το λειτουργικό σύστημα που τους φιλοξενεί επομένως δεν χρειάζεται να εσωκλείουν ένα δικό τους όπως τα virtual machines.
- Ασφάλεια καθώς οι container παραμένουν απομονωμένοι από το λειτουργικό σύστημα που τους φιλοξενεί αλλά και τους υπόλοιπους docker containers που τρέχουν σε αυτό.



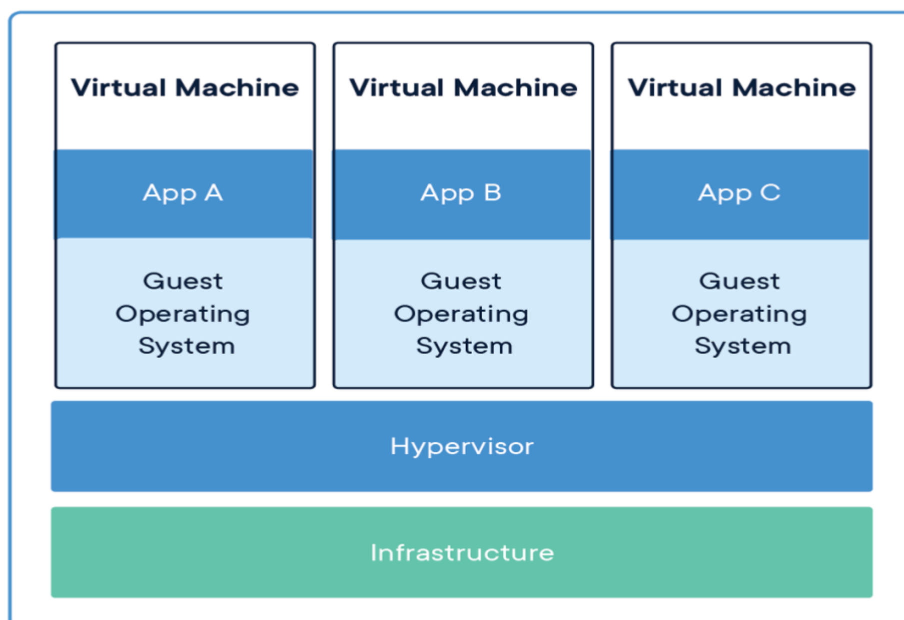
4.2 Containers basic architecture

4.2 Containers και virtual machines

Οι containers και τα virtual machines έχουν αρκετά κοινά στοιχεία αλλά και διαφορές. Στην περίπτωση που συνεργαστούν και τα δύο μαζί τότε η επεκτασιμότητα και η ευελιξία μεγιστοποιείται.

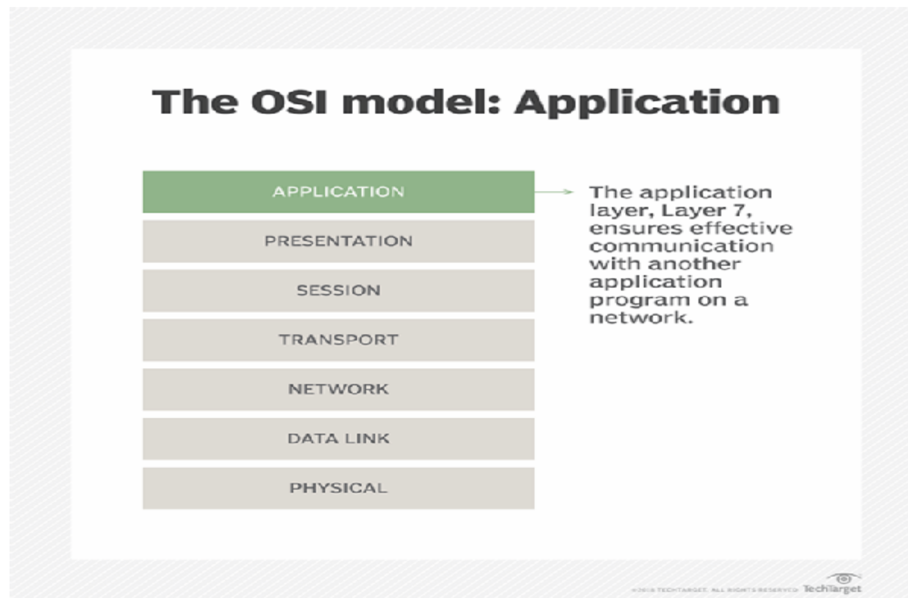
Η βασική χρήση και δύναμη των virtual machines είναι ο διαχωρισμός των πόρων ενός φυσικού μηχανήματος σε πολλά. Με αυτό τον τρόπο το hardware που διαθέτει ένα μηχάνημα μπορεί να επιμεριστεί σε πολλές εικονικές μηχανές που τρέχουν σε αυτό.

Ένα virtual machine εμπεριέχει ολόκληρο το λειτουργικό σύστημα που θέλουμε να χρησιμοποιήσουμε για να τρέξουν οι εφαρμογές μας μαζί με όλες τις εξαρτήσεις τόσο των εφαρμογών όσο και του ίδιου του λειτουργικού συστήματος. Αυτό το καθιστά περισσότερο «βαρύ» για να τρέχουν οι εφαρμογές μας καθώς δεσμεύουν πολύ χώρο από τον δίσκο (αρκετά GB τις περισσότερες φορές) και έχουν σχετικά αργή εκκίνηση.



4.3 Apps in virtual machines architecture

Οι containers είναι μια μικρογραφία του επιπέδου εφαρμογής του OSI (Open Systems Interconnection) μοντέλου που απαρτίζεται από τον κώδικα της εφαρμογής αλλά και τις εξαρτήσεις αυτής.

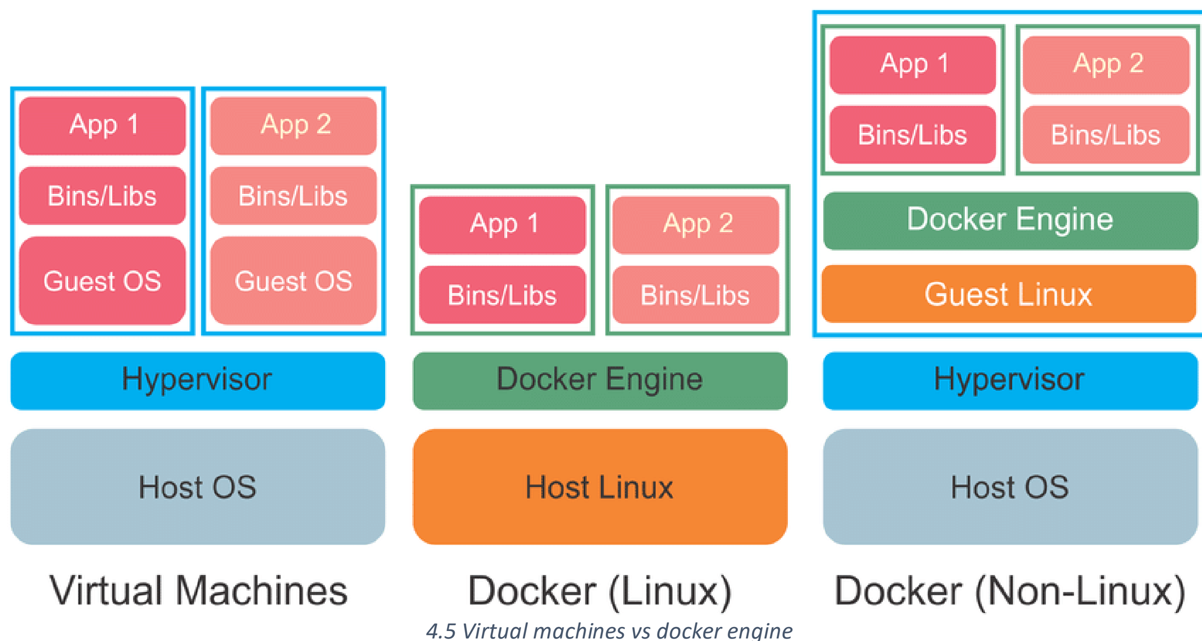


4.4 OSI model

Στο ίδιο μηχάνημα μπορούν να τρέξουν πολλαπλοί containers, οι οποίοι χρησιμοποιούν τον ίδιο φλοιό του λειτουργικού συστήματος που τους φιλοξενεί. Είναι επίσης απομονωμένοι από το υπόλοιπο μηχάνημα, καταλαμβάνουν λιγότερο χώρο στο δίσκο, εκκινούν με πολύ μεγαλύτερη ταχύτητα από τα virtual machines.

Φυσικά είναι σύνηθες οι δύο αυτές τεχνολογίες να συνεργάζονται να προσφέρονται λύσεις που υποστηρίζουν virtual machines πάνω σε φυσικά μηχανήματα, τα οποία με τη σειρά τους φιλοξενούν πολλούς container.

Αυτή η αρχιτεκτονική προσφέρει μεγαλύτερη ευελιξία στον καταμερισμό των πόρων και την συντήρηση των εφαρμογών.



4.5 Virtual machines vs docker engine

4.3 Εργαλεία εγκατάσταση και λειτουργία

Αυτό που χρειαζόμαστε για να χρησιμοποιήσουμε το docker σε μηχανήματα που δεν είναι Linux, είναι το docker server και το docker client. Το docker server ή docker deamon είναι η μηχανή που χρησιμοποιεί το docker και είναι υπεύθυνη για την δημιουργία images, την εκτέλεση container κλπ [6].

Από την άλλη πλευρά το docker client (γνωστό και ως docker CLI) είναι το εργαλείο που χρησιμοποιούμε σε non Linux λειτουργικά συστήματα για να επικοινωνήσουμε με τον docker server.

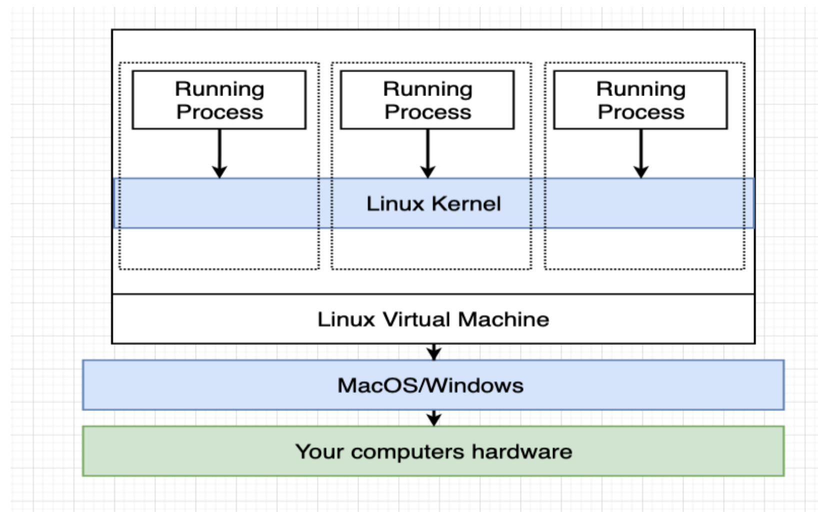
4.3.1 Εγκατάσταση docker

Για να εγκαταστήσουμε το docker σε Mac OS ή Windows ακολουθούμε την παρακάτω διαδικασία:

1. Πηγαίνουμε στην επίσημη σελίδα του docker στον σύνδεσμο: https://www.docker.com/get-started
2. Κατεβάζουμε το εκτελέσιμο αρχείο για το λειτουργικό μας σύστημα
3. Τρέχουμε το εκτελέσιμο αρχείο και ακολουθούμε τις οδηγίες εγκατάστασης
4. Όταν ολοκληρωθεί η εγκατάσταση επιβεβαιώνουμε ότι έχει γίνει σωστά ανοίγοντας ένα terminal ή PowerShell και εκτελώντας την εντολή: **docker ps**

Με αυτό τον τρόπο εγκαθιστούμε τόσο το docker server όσο και το docker cli στο μηχάνημα μας. Στις νεότερες εκδόσεις docker for desktop έχει προστεθεί το docker for Kubernetes στο οποίο μπορούμε να προσομοιώσουμε την λειτουργία ενός Kubernetes cluster στο μηχάνημα για development σκοπούς, το οποίο θα εξετάσουμε στην συνέχεια.

Η αρχιτεκτονική στην οποία βασίζεται το docker for desktop περιγράφεται στο παρακάτω διάγραμμα:



4.6 Virtual machines vs docker engine

4.3.2 Λειτουργία και βασικές εντολές

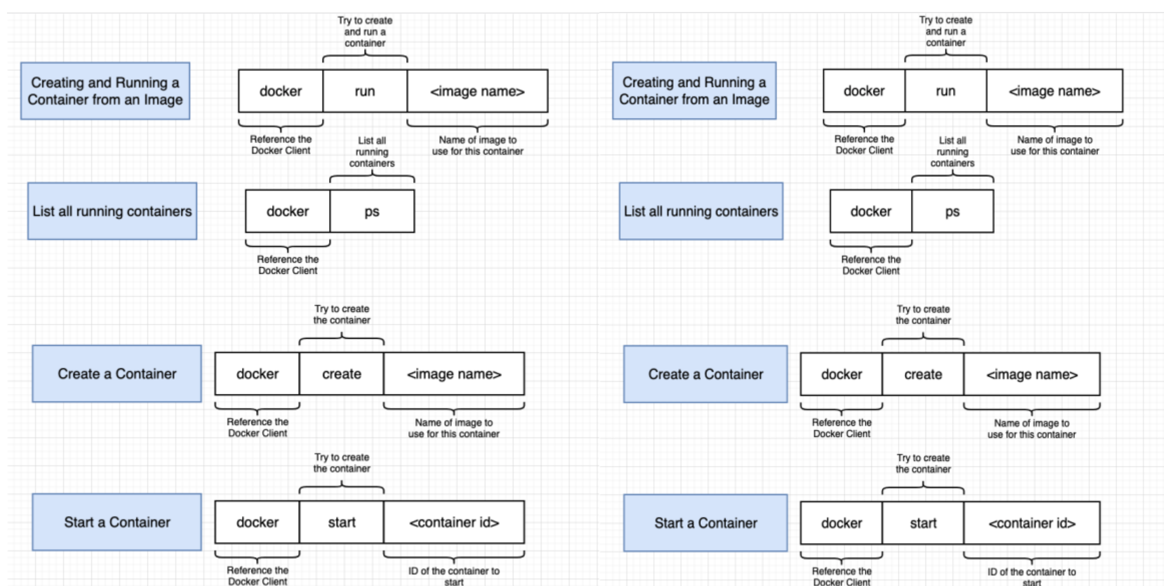
Όπως αναφέρθηκε προηγουμένως το docker χρησιμοποιείται για να κάνει build docker images και να τρέχει τους container που παράγονται από αυτές. Το official repository για docker images είναι το docker hub, στο οποίο μπορούμε να βρούμε όλα τα επίσημα docker image διαφόρων εφαρμογών, αλλά και images από διαφορετικούς χρήστες.

Στο docker hub επίσης μπορούμε να ανεβάζουμε και τα δικά μας image αν θέλουμε να τα κοινοποιήσουμε ή να μπορεί κάποιος να έχει πρόσβαση σε αυτά.

Οι βασικές εντολές που χρησιμοποιούμε κυρίως στο docker είναι:

- ***docker ps***: Μας δίνει την λίστα με τους container που τρέχουν στο μηχάνημα μας.
- ***docker images***: Μας δίνει τα διαθέσιμα images που υπάρχουν αποθηκευμένα στην cache του συστήματος μας
- ***docker create {image-name}:*** Με την εντολή αυτή δημιουργούμε έναν container από ένα image που βρίσκεται είτε στην cache του συστήματος μας είτε στο docker hub. Η εντολή αυτή αναζητεί στο σύστημα αρχείο το αρχείο **Dockerfile**
- ***docker start {container-id}:*** Με αυτή την εντολή ξεκινάει ένας container. Το id του container είναι αυτό που παράγεται από την προηγούμενη εντολή (docker create)
- ***docker run {image-name}:*** Με την εντολή αυτή γίνεται build ένα image και τρέχει ο container που παράγεται από αυτό. Ουσιαστικά η συγκεκριμένη εντολή τρέχει και τις 2 προηγούμενες μαζί. Δημιουργεί έναν container από το image name και τον εκτελεί.

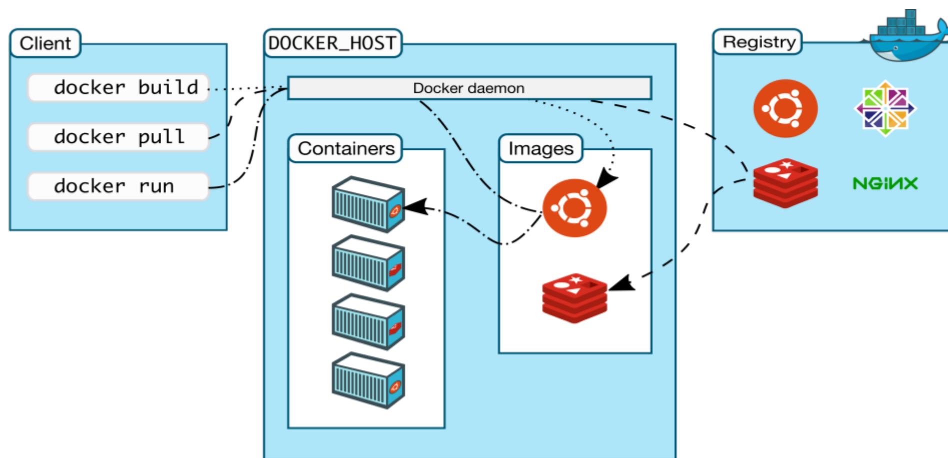
- ***docker stop {container-id}:*** Η εντολή αυτή δίνει σήμα στον συγκεκριμένο container να τερματίσει τις διεργασίες που τρέχει ώστε να σταματήσει να λειτουργεί. Δίνει ένα χρονικό περιθώριο δηλαδή στον container να προλάβει να ολοκληρώσει (αν είναι εφικτό) όποιες διεργασίες του είναι σε εξέλιξη.
- ***docker kill {container-id}:*** Η συγκεκριμένη εντολή δίνει σήμα στον container να σταματήσει άμεσα την λειτουργία του χωρίς να περιμένει να ολοκληρωθούν διεργασίες που είναι σε εξέλιξη
- ***docker logs {container-id}:*** Στην περίπτωση αυτή μας επιστρέφει τα logs που έχουν παραχθεί από τον ίδιο τον container.



4.7 Docker commands

Κάθε εντολή που τρέχει από τον docker client για να εκτελεστεί μια διεργασία στον docker server ακολουθεί την ίδια αρχιτεκτονική. Ο docker server αναζητά πρώτα το image που τη ζητήθηκε από το cache layer που έχει.

Στην περίπτωση που βρει το image τότε εκτελεί την εντολή το ήδη υπάρχον. Στην αντίθετη περίπτωση(όπου δεν βρεθεί η εικόνα είτε είναι διαφορετική έκδοση από αυτή που ζητήθηκε) το πρώτο πράγμα που κάνει εξ ορισμού είναι να ζητήσει την συγκεκριμένη εικόνα από το docker hub το οποίο είναι όπως αναφέραμε το επίσημο docker image repository. Στην συνέχεια εκτελεί: ***docker pull {image-name}*** για να κατεβάσει την εικόνα και να την αποθηκεύσει στο cache layer και τρέχει την αρχική εντολή που του δόθηκε πάνω σε αυτή την εικόνα.



4.8 Docker runtime architecture

4.3.3 Multi container εφαρμογές

Άλλη μια πολύ σημαντική προσφορά του docker είναι ότι μας επιτρέπει να δημιουργήσουμε multi container εφαρμογές. Αυτό μας δίνει την δυνατότητα να δημιουργήσουμε πολλές ξεχωριστές εφαρμογές κάθε μια από τις οποίες θα εκτελείτε σε διαφορετικό container.

Την δυνατότητα αυτή την προσφέρει το docker μέσω του docker compose. Το docker compose είναι ένα εργαλείο το οποίο μας επιτρέπει να παραμετροποιήσουμε διαφορετικούς containers ώστε να επικοινωνούν μεταξύ τους. Η διαδικασία αυτή πραγματοποιείται με τη χρήση ενός docker-compose.yml αρχείου το οποίο περιέχει όλες τις οδηγίες που είναι απαραίτητες.

Το docker compose μας δίνει επίσης την δυνατότητα να δημιουργήσουμε ένα εσωτερικό δίκτυο μέσω του οποίου θα επικοινωνούν οι container, να δημιουργήσουμε volumes στο μηχάνημα που τους φιλοξενεί ώστε να κάνουμε persist πληροφορία η οποία θα παραμένει ακόμη και μετά την λήξη της λειτουργίας τους και να δημιουργούμε environmental μεταβλητές τις οποίες θα διαβάζουν οι containers κατά την λειτουργία τους.

Ένα docker-compose.yml αρχείο έχει την παρακάτω δομή:

docker-compose.yml

```yaml
version: '3'
services:
  postgres:
    image: 'postgres:latest'
  redis:
    image: 'redis:latest'
  nginx:
    restart: always
    build:
      dockerfile: Dockerfile.dev
      context: ./nginx
    ports:
      - '3050:80'
  api:
    build:
      dockerfile: Dockerfile.dev
      context: ./server
    volumes:
      - /app/node_modules
      - ./server:/app
    environment:
      - REDIS_HOST=redis
      - REDIS_PORT=6379
      - PGUSER=postgres
      - PGHOST=postgres
      - PGDATABASE=postgres
      - PGPASSWORD=postgres_password
      - PGPORT=5432
  client:
    build:
      dockerfile: Dockerfile.dev
      context: ./client
    volumes:
      - /app/node_modules
      - ./client:/app
  worker:
    environment:
      - REDIS_HOST=redis
      - REDIS_PORT=6379
    build:
      dockerfile: Dockerfile.dev
      context: ./worker
    volumes:
      - /app/node_modules
      - ./worker:/app
```

4.9 Docker compose yaml

Αν το εξετάσουμε θα δούμε τα εξής:

- Το συγκεκριμένο docker compose περιέχει 5 services. Τα 3 εξ αυτών είναι λογισμικό που χρησιμοποιούμε πολύ συχνά στις εφαρμογές μας. Πρόκειται για ένα cache layer το ρόλο του οποίου εδώ παίζει το Redis, μια βάση δεδομένων postgress και ένα load balancer που είναι ο nginx.
- Το service «worker» χρησιμοποιεί volumes. Δηλαδή για παράδειγμα στην συγκεκριμένη περίπτωση αποθηκεύονται στο host μηχάνημα κάτω από το path /worker όλες οι πληροφορίες που υπάρχουν είτε γίνονται generate κάτω από το path /app που βρίσκεται μέσα στον συγκεκριμένο container.
- Στο service «nginx» βλέπουμε ότι γίνεται expose προς το host μηχάνημα η πόρτα 3050 η οποία εσωτερικά του δικτύου που έχει δημιουργηθεί από το docker οδηγεί στην πόρτα 80.
- Τα services τα οποία έχουμε επεξεργαστεί είτε δεν υπάρχουν στο docker hub (πχ services τα οποία έχουν τον κώδικα μας), δίνεται στο όνομα του Dockerfile μέσα στο build section ώστε το docker compose να ξέρει τι πρέπει να εκτελέσει για να δημιουργήσει τον συγκεκριμένο container / service.
- Γίνεται χρήση environmental μεταβλητών και τους αποδίδονται τιμές έτσι ώστε τα επιμέρους dockerfile των container να έχουν ως reference τις συγκεκριμένες μεταβλητές κατά την δημιουργία και την λειτουργία τους.

Οι βασικές εντολές που χρησιμοποιούμε στο docker-compose είναι παρόμοιες με αυτές του docker client όμως μας δίνουν περισσότερη ευελιξία στην διαχείριση πολλαπλών container:

- ***docker-compose ps:*** Μας δίνει την λίστα με όλους τους container που τρέχουν και διαχειρίζεται το docker compose
- ***docker-compose build:*** Με την εντολή αυτή γίνονται build όλες οι υπηρεσίες που περιέχει το docker compose δηλωμένες μέσα στο yml αρχείο του

- ***docker-compose up:*** Η εντολή αυτή δημιουργεί και εκτελεί όλους τους container που διαχειρίζεται

docker-compose down: Σταματάει την λειτουργία όλων των container και απενεργοποιεί όποιες παραμέτρους δικτύου είτε volume χρησιμοποιούνται για την λειτουργία των container στο μηχάνημα που τους φιλοξενεί.

5. Quarkus

Το Quarkus είναι ένα full-stack, Kubernetes native εργαλείο της Java το οποίο δημιουργήθηκε κυρίως για γλώσσες προγραμματισμού που εκτελούνται εντός του JVM (Java Virtual Machine) και δόθηκε ως λογισμικό ανοιχτού κώδικα σε beta μορφή τον Μάρτιο του 2019.

Έκτοτε από την έκδοση 0.1.0 beta το Quarkus είναι πλέον σε final έκδοση (1.11.1) έχοντας προσελκύσει πολλούς προγραμματιστές κυρίως για τις επιδόσεις του σε Kubernetes περιβάλλον, την ευκολία στην ανάπτυξη κώδικα και τις υπόλοιπες δυνατότητες που προσφέρει η serverless αρχιτεκτονική του.

Σχεδιάστηκε με τέτοιο τρόπο ώστε να εξυπηρετεί την πλειοψηφία των αναγκών των προγραμματιστών κατά την φάση της υλοποίησης, κάτι που μέχρι την δημιουργία του κανένα από τα ανταγωνιστικά προϊόντα δεν προσέφερε.

Έχουν ενσωματώσει εντός του framework, δημοφιλής βιβλιοθήκες της Java όπως αυτές του Eclipse MicroProfile, του Spring, του Apache Camel, drivers για τις περισσότερες βάσεις δεδομένων, loggers, openApi κλπ. Επίσης έχουν αναπτύξει δικές τους βιβλιοθήκες οι οποίες διευκολύνουν την διασύνδεση με άλλες τεχνολογίες και συστήματα όπως είναι το Apache Kafka, RabbitMQ και διάφορες άλλες message driven και CQRS τεχνολογίες.

H Red Hat όταν δημοσίευσε τον κώδικα και το Quarkus ως open source λογισμικό είχε σαν κύριο στόχο να προσελκύσει όσο το δυνατόν περισσότερους προγραμματιστές γύρω από αυτό. Γρήγορα δημιουργήθηκε μια μεγάλη κοινότητα που βοήθησε στην περεταίρω ανάπτυξη του εργαλείου, καθώς και στην επίλυση προβλημάτων του.

Κατάφερε πέρα από τους ανθρώπους της ίδιας της εταιρίας να δημιουργήσει μια ενεργή κοινότητα περίπου 500 προγραμματιστών.

5.1 Κύρια χαρακτηριστικά του Quarkus

Κάποια από τα κυριότερα χαρακτηριστικά του Quarkus είναι τα παρακάτω:

1. Το Quarkus σχεδιάστηκε από developers για developers.

   Η ομάδα που σχεδίασε αρχικά το εργαλείο αυτό είχε σαν μοναδικό στόχο να βοηθήσει τους developers να αναπτύσσουν κώδικα ταχύτερα και πολύ πιο εύκολα, χωρίς να τους προβληματίζουν τα επιμέρους χαρακτηριστικά του περιβάλλοντος στο οποίο προγραμματίζουν.

   Ο χρόνος που χρειάζεται ο χρήστης για να ξεκινήσει να γράφει τον κώδικα του, όπως θα δούμε και στην συνέχεια, είναι σχεδόν αμελητέος. Η Red Hat εξαρχής προσάρμοσε την αρχιτεκτονική του εργαλείου με τέτοιο τρόπο ώστε οι πιο συχνά χρησιμοποιούμενες βιβλιοθήκες να αποτελέσουν μέρος του Quarkus και να ενσωματώνονται σε αυτό με ελάχιστο έως καθόλου παραμετροποίηση.

Έδωσε την ευελιξία στου προγραμματιστές που το χρησιμοποιούν να μπορούν να γράψουν κώδικα σε παραπάνω από 1 γλώσσα (Java, Kotlin, Scala κλπ), καθώς και την δυνατότητα να τρέξουν τις εφαρμογές τους είτε σε jvm mode είτε να τα κάνουν native build και να τις τρέξουν σε Kubernetes native mode.

Τέλος το Quarkus προσφέρει τεχνολογία live coding, κάτι το οποίο υπήρχε μέχρι προσφάτως μόνο σε JavaScript τεχνολογίες. Με αυτό τον τρόπο αυξάνεται η παραγωγικότητα του χρήστη στο μέγιστο δυνατό καθώς οι αλλαγές που κάνει γίνονται compile και deploy άμεσα χωρίς να χρειαστεί να το κάνει με χειροκίνητο τρόπο.

2. Φιλοσοφία «containers first»

Άλλη μια βασική φιλοσοφία πάνω στην οποία έχει δημιουργηθεί το quarkus είναι να εξυπηρετεί όλες τις απαιτήσεις για την ανάπτυξη εφαρμογών οι οποίες τρέχουν μέσα σε containers.

Αυτό που έχει επιτευχθεί ακολουθώντας την παραπάνω φιλοσοφία είναι να επιτύχουν πολύ χαμηλές καταναλώσεις σε μνήμη και εξαιρετικά χαμηλό χρόνο εκκίνησης των εφαρμογών εντός των container.

Ο προσδιορισμός της αρχιτεκτονικής του ίδιου του εργαλείου για να υποστηρίξει τις απαιτήσεις του GraalVM είναι κομβικής σημασίας. Και εδώ παρατηρείται ίσως η μοναδική δυσκολία που ίσως αντιμετωπιστεί από τους προγραμματιστές καθώς το Quarkus στο native build του για Kubernetes αποφεύγει τις κλάσεις στις οποίες χρησιμοποιείται reflection (reflection είναι η αναζήτηση μεθόδων ή πεδίων μιας κλάσης εντός του runtime χωρίς η ίδια να έχει δημιουργηθεί στον class loader).

Οι επιδόσεις που δίνει η Red Hat για την χρήση του Quarkus σε jvm ή native mode συγκριτικά με οποιοδήποτε άλλο αντίστοιχο εργαλείο του ανταγωνισμού είναι ιδιαίτερα εντυπωσιακές[9].

5.1 Quarkus memory consumption and boot time

3. Σύγχρονη και ασύγχρονη μεθοδολογία

Τέλος το Quarkus ενοποιεί τον σύγχρονο και ασύγχρονο τρόπο ανάπτυξης κώδικα μέσα από το ίδιο το εργαλείο.

Οι χρήστες μπορούν να επιλέξουν αν θα χρησιμοποιήσουν το «imperative» mode στις εφαρμογές τους, δηλαδή την κλασσική μορφή client – server εφαρμογών είτε θα χρησιμοποιήσουν το «reactive» mode όπου η εφαρμογή βασίζεται σε ασύγχρονους τρόπους ανταλλαγής πληροφορίας.

Στην πρώτη περίπτωση παρέχονται όλες οι γνωστές http client βιβλιοθήκες με blocking threads και σύγχρονη επικοινωνία ενώ στην δεύτερη περίπτωση παρέχονται βιβλιοθήκες για message driven αρχιτεκτονικές όπως evet bus, Apache Kafka, Apache Camel κλπ.

5.2 Ανάπτυξη εφαρμογών με Quarkus

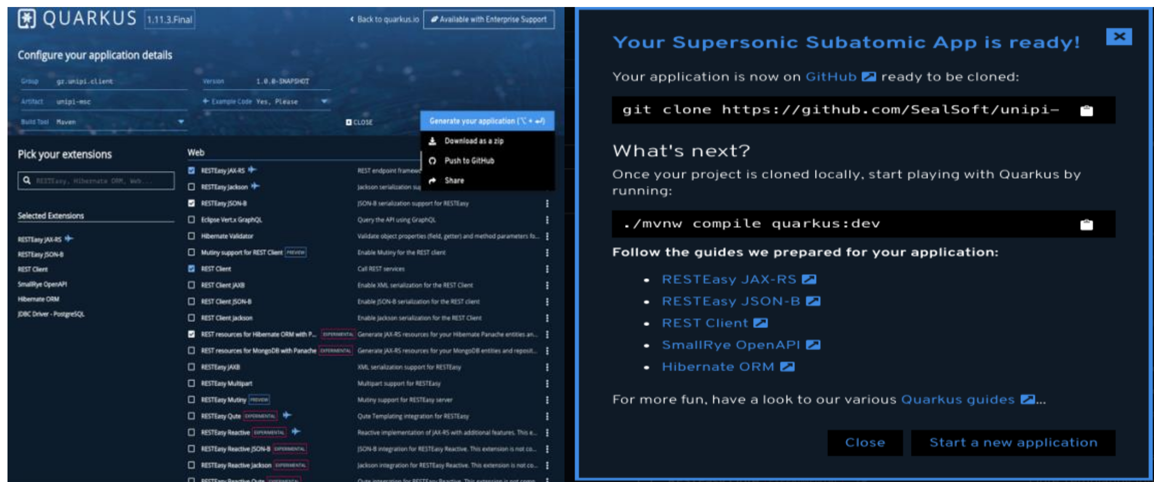
Για να αναπτύξουμε μια εφαρμογή σε Quarkus αυτά που χρειαζόμαστε είναι τα εξής:

1. Έναν code editor ή κατά προτίμηση ένα ide(integrated development environment) κάποια εκ των οποίων έχουν plugin για την ανάπτυξη εφαρμογών με quarkus ή ακόμη και απλώς με maven.
2. Χρειαζόμαστε οπωσδήποτε μια έκδοση JDK (Java Development Kit) το οποίο να είναι έκδοση 8 ή μεταγενέστερη. Ως open source λύσεις για το JDK στην κορυφή βρίσκεται το OpenJDK καθώς αυτό της Oracle έχει περιορισμούς για χρήση της εφαρμογής σε παραγωγικό περιβάλλον.
3. Τέλος χρειαζόμαστε μια έκδοση του maven ή του Gradle στο μηχάνημα μας ώστε να μπορέσει να γίνει build η εφαρμογή μαζί με τις εξαρτήσεις της. Το Quarkus προτείνει maven έκδοση 3.6.2 και μεταγενέστερες.

Μετά την εγκατάσταση των παραπάνω εργαλείων στο λειτουργικό μας σύστημα για να δημιουργήσουμε μια εφαρμογή, το Quarkus παρέχει έναν πολύ εύκολο τρόπο μέσω του web initializer που βρίσκεται στον σύνδεσμο: https://code.quarkus.io/

Από το εκεί επιλέγουμε όλες τις εξαρτήσεις της εφαρμογής μας, αν θέλουμε να χρησιμοποιεί maven ή grade ως build tool, καθώς και τις υπόλοιπες ρυθμίσεις (maven artifact και group).

Στην συνέχεια από το μενού «Generate your application» επιλέγουμε τον τρόπο που θέλουμε να δημιουργηθεί ο σκελετός της εφαρμογής μαζί με τις εξαρτήσεις της. Μπορούμε να επιλέξουμε να την στείλουμε κατευθείαν στο github repository μας ώστε από εκεί έπειτα να την κάνουμε clone και να την χειριστούμε στο μηχάνημα μας.

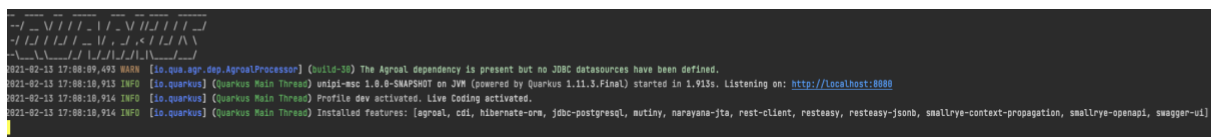


5.2 Quarkus initializer with GitHub

Στην περίπτωση μας επιλέγουμε «Push to github» ώστε η εφαρμογή να πάει σε public repository. Στην συνέχεια το παράθυρο που εμφανίζεται μας ενημερώνει για την τοποθεσία της εφαρμογής και τον url με το οποίο θα την κάνουμε clone στο μηχάνημα μας.

Αφού κάνουμε ***clone*** την εφαρμογή μας, την ανοίγουμε στο IDE που χρησιμοποιούμε και εκτελούμε σε ένα terminal την εντολή: ***mvn clean package quarkus:dev***

Η παραπάνω εντολή κάνει build την εφαρμογή και στην συνέχεια την τρέχει στο μηχάνημα ως jvm build χωρίς να χρησιμοποιήσει το native build του GraalVM. Όταν εκκινήσει η εφαρμογή για να την δοκιμάσουμε εκτελούμε πάλι σε ένα terminal την εντολή: ***curl http://localhost:8080/unipi***


5.3 Quarkus run app dev jvm mode

5.2.1 Jvm και native build

Για να κάνουμε την εφαρμογή μας build σε jvm mode, πάμε αρχικά στον root φάκελο της εφαρμογής και σε ένα terminal δίνουμε την εντολή: ***docker build -t unipi/jvm -f src/main/docker/Dockerfile.jvm .***

Το πρώτο σκέλος της εντολής δίνει στο image που δημιουργείται το unipi/jvm ως repository name ενώ το δεύτερο σκέλος δίνει το path όπου βρίσκεται το αντίστοιχο Dockerfile. Η τελεία που χρησιμοποιείται στο τέλος είναι για να πει στο docker client ότι θέλουμε να χρησιμοποιήσει τα sources από τον φάκελο που βρισκόμαστε και έπειτα[9].

Για να δούμε ότι το image έχει δημιουργηθεί αρκεί να τρέξουμε την εντολή: ***docker images***
Στη συνέχεια για να τρέξουμε τον container του image που δημιουργήσουμε εκτελούμε την εντολή: ***docker run --rm --name unipi-jvm -p 8080:8080 unipi/jvm:latest***

Στην συγκεκριμένη εντολή αυτό που συμβαίνει σειριακά είναι να αφαιρέσουμε προηγούμενο container με την ίδια ονομασία, να δώσουμε στον container την ονομασία «unipi-jvm», να κάνουμε expose την πόρτα 8080 του container προς την ίδια πόρτα στο μηχάνημα μας και να του δώσουμε στο τέλος την εικόνα από την οποία θα δημιουργηθεί ο container[9].

Για να ελέγξουμε την λειτουργία του εκτελούμε την εντολή: ***docker ps*** και στην συνέχεια για να δούμε τα resources που χρησιμοποιεί σε πραγματικό χρόνο εκτελούμε την εντολή ***docker stats | grep jvm***

```
→  unipi-msc git:(master) ✗ docker run --rm --name unipi-jvm -p 8080:8080 unipi/jvm:latest
exec java -Dquarkus.http.host=0.0.0.0 -Djava.util.logging.manager=org.jboss.logmanager.LogManager -javaagent:/opt/agent-bond/agent-bond.jar=jmx_exporter{{9779:/opt/agent-bond/jmx_exporter_config.yml}} -XX:+UseParallelGC -XX:GCTimeRatio=4 -XX:AdaptiveSizePolicyWeight=90 -XX:MinHeapFreeRatio=20 -XX:MaxHeapFreeRatio=40 -XX:+ExitOnOutOfMemoryError -cp . -jar /deployments/app.jar
2021-02-13 17:48:49,028 INFO  [io.quarkus] (main) unipi-msc 1.0.0-SNAPSHOT on JVM (powered by Quarkus 1.11.3.Final) started in 2.697s. Listening on: http://0.0.0.0:8080
2021-02-13 17:48:49,066 INFO  [io.quarkus] (main) Profile prod activated.
2021-02-13 17:48:49,066 INFO  [io.quarkus] (main) Installed features: [agroal, cdi, hibernate-orm, jdbc-postgresql, mutiny, narayana-jta, rest-client, resteasy, resteasy-jsonb, smallrye-context-propagation, smallrye-openapi]
```

5.4 Quarkus run jvm image

Τώρα θα ακολουθήσουμε την ίδια διαδικασία για να κάνουμε build το image σε native μορφή ώστε να δούμε τι διαφορές υπάρχουν τόσο στο startup time όσο και στο memory management.

Στο native image θα δώσουμε την ονομασία unipi-native και θα τρέξουμε build με το αντίστοιχο Dockerfile για native. Το native build θα χρειαστεί αρκετά περισσότερο χρόνο για να εκτελεστεί. Για να ξεκινήσουμε τον container στην συνέχεια θα ακολουθήσουμε την ίδια διαδικασία με την διαφορά ότι θα κάνουμε expose τον container στην πόρτα 8081 ώστε να μην έχουμε conflict με τον jvm container.

Παρατηρώντας τα boot time των 2 container βλέπουμε ότι η διαφορά είναι πολύ μεγάλη καθώς το jvm build εκκίνησε σε 1,22sec ενώ το native build εκκίνησε σε 0,038s. Στην συνέχεια βλέποντας στα resources που καταναλώνουν βλέπουμε ότι το jvm build καταναλώνει 97,84mb ram ενώ το native build μόλις 6,93mb.

Σημαντική είναι επίσης η χρήση της CPU καθώς το jvm καταναλώνει το 1.64% της επεξεργαστικής ισχύς ενώ το native μόλις το 0.12%.

Καταλήγουμε λοιπόν ότι σε ένα Kubernetes περιβάλλον η διαφορά μεταξύ των 2 container είναι πολύ μεγάλη τόσο στην κατανάλωση των πόρων όσο και στον χρόνο εκκίνησης[9].

```
  2 years ago     110MB
→  unipi-msc git:(master) ✗ docker run --rm --name unipi-jvm -p 8080:8080 unipi/jvm:latest
exec java -Dquarkus.http.host=0.0.0.0 -Djava.util.logging.manager=org.jboss.logmanager.LogManager -javaagen
t:/opt/agent-bond/agent-bond.jar=jmx_exporter{{9779:/opt/agent-bond/jmx_exporter_config.yml}} -XX:+UseParal
lelGC -XX:GCTimeRatio=4 -XX:AdaptiveSizePolicyWeight=90 -XX:MinHeapFreeRatio=20 -XX:MaxHeapFreeRatio=40 -XX
:+ExitOnOutOfMemoryError -cp . -jar /deployments/app.jar
2021-02-13 20:47:52,837 INFO  [io.quarkus] (main) unipi-msc 1.0.0-SNAPSHOT on JVM (powered by Quarkus 1.11.
3.Final) started in 1.255s. Listening on: http://0.0.0.0:8080
2021-02-13 20:47:52,851 INFO  [io.quarkus] (main) Profile prod activated.
2021-02-13 20:47:52,852 INFO  [io.quarkus] (main) Installed features: [agroal, cdi, hibernate-orm, jdbc-pos
tgresql, mutiny, narayana-jta, rest-client, resteasy, resteasy-jsonb, smallrye-context-propagation, smallry
e-openapi]
```

```
→  unipi-msc git:(master) ✗ docker images
Error response from daemon: Bad response from Docker engine
→  unipi-msc git:(master) ✗ docker ps
Error response from daemon: Bad response from Docker engine
→  unipi-msc git:(master) ✗ docker run --rm --name unipi-native -p 8081:8080 unipi/native:
latest
2021-02-13 20:47:29,337 INFO  [io.quarkus] (main) unipi-msc 1.0.0-SNAPSHOT native (powered
 by Quarkus 1.11.3.Final) started in 0.038s. Listening on: http://0.0.0.0:8080
2021-02-13 20:47:29,337 INFO  [io.quarkus] (main) Profile prod activated.
2021-02-13 20:47:29,337 INFO  [io.quarkus] (main) Installed features: [agroal, cdi, hibern
ate-orm, jdbc-postgresql, mutiny, narayana-jta, rest-client, resteasy, resteasy-jsonb, sma
llrye-context-propagation, smallrye-openapi]
```

5.5 Quarkus jvm / native container startup

```
Last login: Sat Feb 13 22:50:11 on ttys003
→  ~ docker stats | grep unipi
2ad4b5ed7289   unipi-jvm
       0.27%     97.84MiB / 5.809GiB   1.64%     866B / 0B     37MB / 0B        28
c3feecc60b4d   unipi-native
       0.01%     6.934MiB / 5.809GiB   0.12%     1.05kB / 0B   2.14MB / 0B      13
```

5.6 Quarkus jvm vs native memory footprint

6. Apache Openwhisk

Το Apache Openwhisk αποτελεί άλλη μια serverless και functions as a service (FaaS) open source λύση που προσέφεραν σε συνεργασία IBM και Adobe η οποία δύναται να αξιοποιηθεί είτε σε κάποιο cloud provider (πχ managed Kubernetes cluster) είτε ακόμη και on premise σε οποιοδήποτε data center.

Η πρώτη έκδοση του Openwhisk που δόθηκε ήταν 0.9.0-incubating 30/6/2016 ως beta και στην συνέχεια στις 31/10/2020 δόθηκε η πρώτη τελική του έκδοση 1.0.0.

Σε σύγκριση με τα υπόλοιπα αντίστοιχα serverless προϊόντα, το Openwhisk δίνει ακόμη περισσότερη ευελιξία, εύκολη και αποτελεσματική κλιμάκωση (scaling) ανάλογα με το περιβάλλον στο οποίο τρέχει και μπορεί να υποστηρίξει πολλά request ταυτόχρονα.

Η IBM δίνει στους χρήστες της επίσης την δυνατότητα να χρησιμοποιήσουν την enterprise version του Openwhiksk, η οποία τρέχει στο δικό τους cloud περιβάλλον και μέσω του dashboard που παρέχει να δώσει ακόμη μεγαλύτερη ελευθερία στην επεξεργασία του. Το περιβάλλον αυτό της IBM ονομάζεται IBM Bluemix.

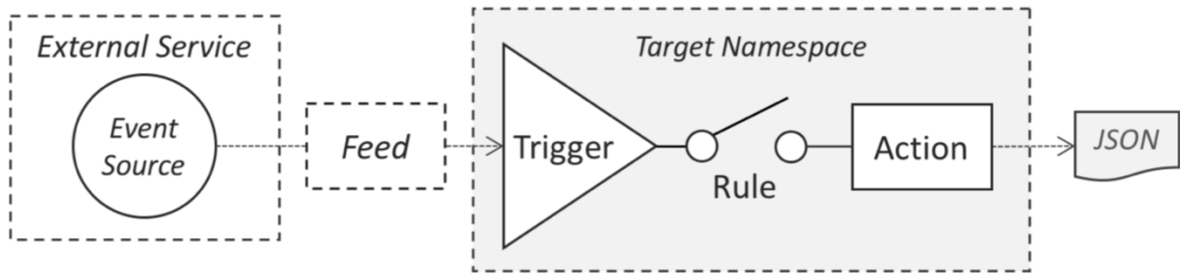
6.1 Αρχιτεκτονική και λειτουργίες

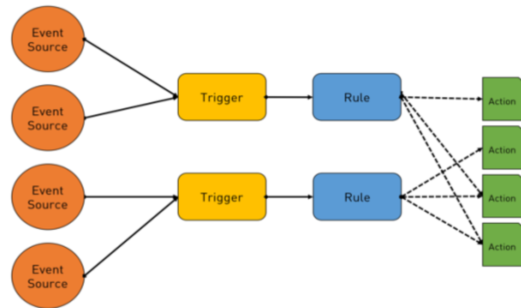
Το Openwhisk έχει σχεδιαστεί για να χρησιμοποιείται ως ένα περιβάλλον στο οποίο να τρέχουν μικρά ή μεγαλύτερα κομμάτια κώδικα (actions) τα οποία πυροδοτούνται από triggers ή events.

Στην αρχιτεκτονική αυτή, οι προγραμματιστές προσδιορίζουν τις ενέργειες που οι οποίες θα εκτελούνται ως αυτόνομα προγράμματα. Στην συνέχεια ορίζονται οι triggers οι οποίο εκτελούνται από ενέργειες που συμβαίνουν συνήθως εκτός του οικοσυστήματος του Openwhisk όπως πχ αλλαγές σε μια βάση δεδομένων, τρίτα συστήματα που ενημερώνουν για αλλαγές ή κλήσεις που δέχθηκαν κλπ.

Actions και Triggers είναι ανεξάρτητα μεταξύ τους και ένα trigger δεν γνωρίζει αν είναι συνδεδεμένο με κάποιο action. Στην αρχιτεκτονική του Openwhisk είναι δυνατό ένα trigger να καταλήγει στην ενεργοποίηση ενός ή περισσοτέρων actions. Αυτή την διαδικασία την γνωρίζει μόνο το σύνολο της εφαρμογής κατά την εκτέλεση του.

Τα actions και triggers συνδέονται μεταξύ τους μέσω των rules. Τα rules αποτελούν συνθήκες που πρέπει να πληρούνται ώστε η ενεργοποίηση ενός trigger να δρομολογηθεί προς την εκτέλεση ενός ή περισσοτέρων actions [16].





6.1 Openwhisk workflow process

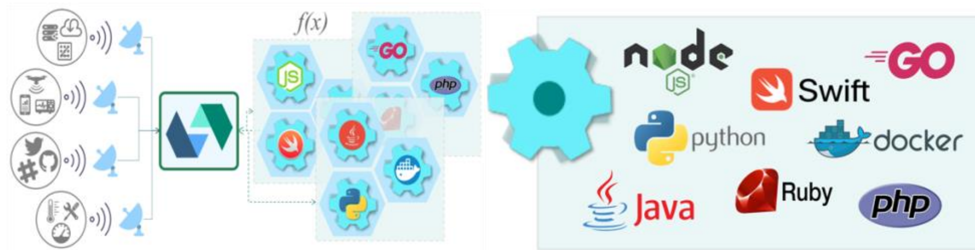
6.1.1 Actions

Τα actions αποτελούν stateless κώδικα ο οποίος τρέχει πάνω στην πλατφόρμα του Openwhisk. Ένα action για παράδειγμα θα μπορούσε να χρησιμοποιηθεί ως απάντηση σε ένα API call, ένα GitHub pull request, ένα νέο εισερχόμενο μήνυμα στο slack ή το Twitter.

Ένα action μπορεί να έχει πολλές μορφές εντός της πλατφόρμας του Openwhisk. Μπορεί να είναι από μια απλή συνάρτηση που γράφτηκε σε μια από τις υποστηριζόμενες γλώσσες, μπορεί να είναι ένα binary εκτελέσιμο αρχείο ή ακόμη και εκτελέσιμα αρχεία που βρίσκονται μέσα σε docker containers.

Το runtime του Openwhisk υποστηρίζει πληθώρα runtime sdk για γλώσσες προγραμματισμού. Μεταξύ αυτών είναι όλες οι δημοφιλείς γλώσσες όπως java, python, JavaScript, Go, Ruby κλπ. Μεταξύ αυτών προστίθενται και έτοιμοι docker containers που έχουν δημιουργηθεί από διάφορους χρήστες. Η μεγάλη ευελιξία όμως που προσθέτει το Openwhisk είναι η δυνατότητα κάποιος να δημιουργήσει το δικό του runtime sdk αν η γλώσσα που επιθυμεί δεν υποστηρίζεται. Επίσης η ίδια η πλατφόρμα έχει ενσωματωμένα ήδη αρκετά actions που μπορούν να παραμετροποιηθούν και να χρησιμοποιηθούν από τους χρήστες.

Τέλος, κάτι που υποστηρίζει ακόμη το Openwhisk σχετικά με τα actions είναι ότι επιτρέπει στους χρήστες του να δημιουργήσει αλληλουχίες από αυτά ακόμη και αν αυτά είναι σε διαφορετικές γλώσσες γραμμένα και αναλαμβάνει η πλατφόρμα την ενορχήστρωση τους.



6.2 Openwhisk runtimes

6.1.2 Triggers rules και events

Τα triggers του Openwhisk είναι κανάλια τα οποία έχουν οδηγίες για ενεργοποιηθούν από events που συμβαίνουν σε τρίτα συστήματα. Στην αρχιτεκτονική δομή της πλατφόρμας ο trigger είναι ουσιαστικά ένας υποδοχέας των πληροφοριών πριν αυτές δρομολογηθούν εσωτερικά για να καταλήξουν πιθανόν στην εκτέλεση ενός ή περισσοτέρων actions.

Τα rules με την σειρά τους είναι ο συνδετικός κρίκος μεταξύ trigger και action. Χωρίς την παρουσία των rules στην υποδομή, ακόμη και αν ενεργοποιούνται συνεχώς οι triggers από events, δεν είναι εφικτό να φτάσει η πληροφορία στα actions ώστε να εκτελεστούν. Τα rules λειτουργούν σαν δρομολογητές μεταξύ των 2 οντοτήτων. Περιέχουν μέσα συνθήκες που αναλόγως αν ή όχι ικανοποιούνται μεταφέρεται ή όχι η πληροφορία στα actions για να εκτελεστούν [16].

Τα event sources είναι υπηρεσίες ή πηγές οι οποίες παράγουν event τα οποία μεταφέρουν συγκεκριμένη πληροφορία. Σκοπός της πλατφόρμας είναι να εξυπηρετήσει τις event / message driven εφαρμογές οπότε τα event sources αποτελούν την εναρκτήριο δύναμη όλης της διαδικασίας μέχρι την εκτέλεση ενός action.

6.1.3 Key components

Nginx

O nginx αποτελεί ένας από τους πιο γνωστούς web server ανοιχτού λογισμικού. O nginx εντός του Openwhisk λειτουργεί ως reverse proxy καθώς κάνει expose όλα τα client api μέσω http / https request. Χρησιμοποιείται επίσης για να πιστοποιεί την υποδομή – SSL.

Κάθε request που φτάνει προς την υποδομή του Openwhisk, ακόμη και αυτά τα οποία προέρχονται από το cli, περνούν πρώτα από τον reverse proxy. Ως προς την kubernetes υποδομή αξίζει να αναφέρουμε ότι ο nginx γίνεται εύκολα scale επειδή είναι stateless.

Controller

Το αμέσως επόμενο component που κατευθύνεται ένα request είναι ο controller / gatekeeper. O controller αρχικά σε πρώτο στάδιο κάνει authenticate και authorize κάθε request. Στην συνέχεια ενορχηστρώνει εσωτερικά όλα το Openwhisk api και αποφασίζει αναλόγως με το path, namespace κλπ πως και με ποιο τρόπο θα προωθήσει το request.

Σε kubernetes cluster που λειτουργούν παραγωγικά χρησιμοποιούνται περισσότερα από ένα controller instances για να μπορούν να εξυπηρετούν πολλαπλά ταυτόχρονα request.

CouchDB

H CouchDB είναι μια open source NoSQL βάση δεδομένων, η οποία εξυπηρετεί την αποθήκευση δεδομένων σε μορφή JSON. Στο Openwhisk περιβάλλον η CouchDB έχει αποθηκευμένη όλη την πληροφορία που αφορά το σύνολο του – credentials, metadata, actions, rules, triggers,namespaces κλπ. O controller για να κάνει authenticate και authorize το request, αναζητεί τα credentials που βρίσκει στους headers του request μέσα στην CouchDB.

Στην συνέχεια αποφασίζει για το path που θα καταλήξει το request πάλι επιβεβαιώνοντας το namespace και το api path που βρίσκει στο request με αυτά που αντιστοιχούν στα actions, triggers που βρίσκει μέσα στην βάση δεδομένων. Κάθε ενέργεια που πραγματοποιείται εντός του Openwhisk αποθηκεύεται στην CouchDB. Αυτό μπορεί να αφορά από την δημιουργία νέο χρήστη μέχρι το invocation ενός action.

Kafka

Το Kafka είναι μια distributed event streaming πλατφόρμα. Η βασική της λειτουργία είναι να χειρίζεται και να διανέμει δεδομένα μεταξύ servers και client σε ένα οικοσύστημα μέσω tcp κάνοντας dispatch messages ή events.

Στο σύστημα του Openwhisk, το Kafka είναι ο συνδετικός κρίκος μεταξύ του controller και των invoker. Δέχεται και ρυθμίζει τα μηνύματα που δέχεται από τον controller. Όταν τα μηνύματα αποστέλλονται από το Kafka προς έναν invoker και παραδοθούν τότε ο controller επιστρέφει πίσω ένα activation id.

Λόγω της ασύγχρονης λειτουργίας του τα thread που χρησιμοποιούνται δεν γίνονται block μέχρι να περιμένουν να απαντήσει. Για να παρακάμψει κάποιος αυτή την λειτουργία πρέπει να το δηλώσει ρητά στο request του με την παράμετρο ***blocking = true.***

ZooKeeper

Το ZooKeeper είναι ένα σύστημα που χρησιμοποιείται κατά κόρον μαζί με το Kafka σε ασύγχρονα περιβάλλοντα όπως του Openwhisk. Κύρια εργασία του είναι η συντήρηση της παραμετροποίησης ενός συστήματος σε ένα κεντρικοποιημένο περιβάλλον ώστε να μπορεί να συγχρονίζει συνεχώς τα υπόλοιπα distributed συστήματα που λειτουργούν γύρω από αυτό.

Στο Openwhisk το ZooKeeper διαχειρίζεται το Kafka cluster, ελέγχοντας συνεχώς τους κόμβους του αλλά και τα θέματα, μηνύματα που διανέμονται από αυτό.

Invoker

Ο Invoker είναι το τελευταίο στάδιο όπου καταλήγει ένα request στο Openwhisk. Ο invoker δέχεται το request, επεξεργάζεται τα στοιχεία του και αποφασίζει πως πρέπει να εκτελεστεί. Σε κάθε περίπτωση όλα καταλήγουν στην εκτέλεση ενός docker container.

Ο invoker παίρνει τις πληροφορίες που χρειάζεται από την CouchDB ώστε να γνωρίζει το runtime και τον κώδικα που θα χρειαστεί να τρέξει μέσα στον container. Όταν η εκτέλεση του container ολοκληρωθεί, τότε το activation id που παράγεται το αποθηκεύει πάλι στην CouchDB.

Επίσης διαχειρίζεται τον τρόπο με τον οποίο θα εκτελεστεί ένας container. Έχει την δυνατότητα να τρέξει έναν container που ήδη είναι ενεργός, να τρέξει έναν container ο οποίος έχει ήδη δημιουργηθεί αλλά δεν είναι ενεργός είτε να δημιουργήσει έναν καινούργιο όπου είναι και η πιο χρονοβόρα διαδικασία.

Docker

Σχεδόν όλη η υποδομή του Openwhisk βασίζεται σε containers. Όλα τα components που περιγράψαμε μέχρι στιγμής είναι docker containers. Η πλειοψηφία των docker images που χρησιμοποιεί το Openwhisk είναι κάτω από δικό του repository στο docker hub.

API Gateway

Το τελευταίο key component του Openwhisk που θα αναφέρουμε και θα χρησιμοποιήσουμε είναι το API Gateway. Η βασική του λειτουργία είναι να κάνει expose actions και triggers που έχουμε δημιουργήσει στο web μέσω http routing.

6.2 Εγκατάσταση Openwhisk (local kubernetes)

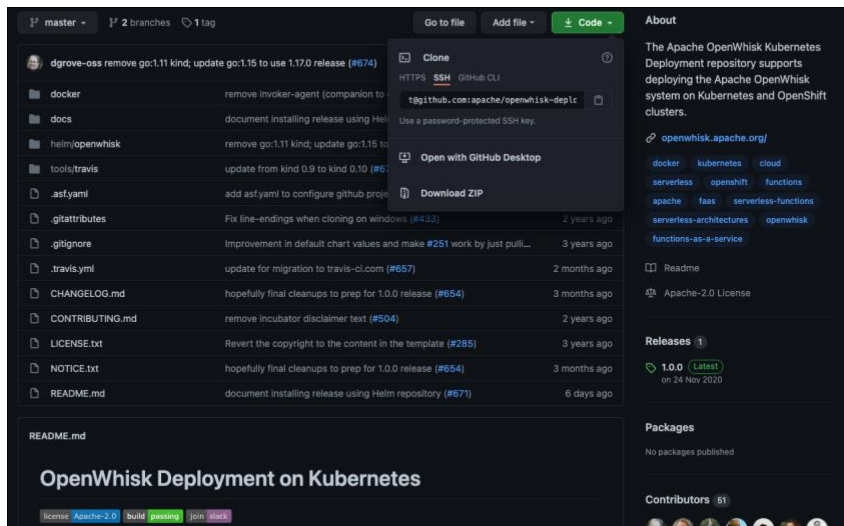
Για να εγκαταστήσουμε το Openwhisk στο μηχάνημα μας θα χρησιμοποιήσουμε αρχικά το docker for desktop. Ενεργοποιώντας το Kubernetes το οποίο έρχεται ενσωματωμένο ως υπηρεσία με το docker for desktop θα κάνουμε deploy το Openwhisk σε αυτό το dev cluster.

Πέραν αυτού θα χρειαστούμε και το helm ως ενδιάμεσο εργαλείο το οποίο θα μας βοηθήσει να κάνουμε deploy το Openwhisk στο cluster.

1. Windows
    a. Για να εγκαταστήσουμε το helm σε windows μηχάνημα ακολουθούμε τον σύνδεσμο: https://github.com/helm/helm/releases
    b. Εκεί βρίσκουμε την τελευταία stable έκδοση, επιλέγουμε και κατεβάζουμε το αρχείο zip.
    c. Κάνουμε unzip το αρχείο στο μηχάνημα μας.
    d. Σε ένα PowerShell εκτελούμε την εντολή ***helm init*** και ενεργοποιούμε το helm.
2. Mac OS
    a. Η εγκατάσταση του helm σε Mac OS προτείνεται να γίνει μέσω το homebrew.
    b. Από το terminal εκτελούμε την εντολή ***brew install kubernetes-helm***
    c. Στην συνέχεια για να επιβεβαιώσουμε την σωστή εγκατάσταση του εκτελούμε την εντολή ***helm version*** και για να το εκκινήσουμε την εντολή ***helm init***
3. Linux
    a. Για Linux διανομή η διαδικασία είναι αντίστοιχη με αυτή των windows στο πρώτο βήμα.
    b. Αφού κάνουμε unzip το αρχείο που κατεβάσαμε μεταφέρουμε το **helm** αρχείο στα service binaries με την εντολή ***sudo mv linux-amd64/helm usr/local/bin***
    c. Για να επιβεβαιώσουμε και εδώ την εγκατάσταση χρησιμοποιούμε την εντολή ***helm version***

Όταν ολοκληρωθεί η εγκατάσταση του helm προχωράμε με το Openwhisk deployment. Αρχικά θα κάνουμε clone από το official GitHub repository το project που θα γίνει deploy. Κατευθυνόμαστε στον σύνδεσμο: https://github.com/apache/openwhisk-deploy-kube

Κάνουμε στην συνέχεια copy το clone command URL και σε κάποια θέση του δίσκου που επιθυμούμε, ανοίγουμε ένα terminal και εκτελούμε:
git clone git@github.com:apache/openwhisk-deploy-kube.git



6.3 Openwhisk helm kubernetes deploy project

Δημιουργείται με αυτόν τον τρόπο ένα νέο project στο μηχάνημα μας, το οποίο θα ανοίξουμε με έναν code editor είτε με το IDE. Στο root folder του project δημιουργούμε ένα νέο αρχείο με όποια ονομασία θέλουμε και κατάληξη ***.yaml***

Στην συγκεκριμένη περίπτωση το αρχείο το ονομάσαμε ***owlocal.yaml***

Μέσα στο αρχείο αυτό γράφουμε τις παρακάτω εντολές:

```
{..} owlocal.yaml
1   whisk:
2     ingress:
3       type: NodePort
4       apiHostName: 192.168.65.3
5       apiHostPort: 31001
6
7   nginx:
8     httpsNodePort: 31001
```

6.3 Openwhisk kubernetes user yaml file

Το apiHostName είναι η cluster ip την οποία μπορούμε να την βρούμε εκτελώντας την εντολή:
kubectl describe nodes | grep IP

```
spyros@spyross-mini: ~
→  ~ kubectl describe nodes | grep IP
  InternalIP:  192.168.65.3
→  ~
```

6.4 Find internal kubernetes ip

Στην συνέχεια θέλουμε αρχικά να δημιουργήσουμε στο Kubernetes cluster ένα καινούργιο namespace για να κάνουμε σε αυτό deploy το Openwhisk και να δώσουμε στα nodes που θέλουμε να έχουν πρόσβαση στα actions τον ρόλο invoker. Αυτά τα κάνουμε σειριακά με τις παρακάτω εντολές:

- ***kubectl create namespace openwhisk***
- ***kubectl label nodes –all openwhisk-role=invoker***

Για να ολοκληρώσουμε την εγκατάσταση κατευθυνόμαστε στο root φάκελο του deploy-kube project που κάναμε clone και εκτελούμε την εντολή:

helm install -name openwhisk --namespace openwhisk -f owlocal.yaml ./helm/Openwhisk

Η εντολή μας επιστρέφει ότι εκτελέστηκε, όμως χρειάζεται λίγος χρόνος ώστε να ολοκληρωθεί. Για να ενημερωνόμαστε για την κατάσταση των pod που δημιουργήσαμε μπορούμε να εκτελούμε την εντολή:

kubectl --namespace openwhisk get pod

Όταν ολοκληρωθεί η εκκίνηση όλων των container θα δούμε το παρακάτω response:

```
spyros@spyross-mini: ~
→ ~ kubectl describe nodes | grep IP
  InternalIP:  192.168.65.3
→ ~ kubectl --namespace openwhisk get pod
NAME                                          READY   STATUS      RESTARTS   AGE
openwhisk-alarmprovider-687b588f44-ccn9m      1/1     Running     2          15h
openwhisk-apigateway-bf6d6784b-dxtjw          1/1     Running     2          15h
openwhisk-controller-0                        1/1     Running     2          15h
openwhisk-couchdb-6957d4dfb7-qjcf7            1/1     Running     2          15h
openwhisk-gen-certs-pkjn8                     0/1     Completed   0          15h
openwhisk-init-couchdb-g4mk9                  0/1     Completed   0          15h
openwhisk-install-packages-8wjkm              0/1     Completed   0          15h
openwhisk-invoker-0                           1/1     Running     2          15h
openwhisk-kafka-0                             1/1     Running     2          15h
openwhisk-kafkaprovider-6c69bd4788-rbndv      1/1     Running     2          15h
openwhisk-nginx-665b58845f-65zpd              1/1     Running     2          15h
openwhisk-redis-74c4b564c9-p6frb              1/1     Running     2          15h
openwhisk-wskadmin                            1/1     Running     2          15h
openwhisk-zookeeper-0                         1/1     Running     2          15h
owdev-wskadmin                                1/1     Running     1          11h
wskopenwhisk-invoker-00-1-prewarm-nodejs10    1/1     Running     0          130m
wskopenwhisk-invoker-00-2-prewarm-nodejs10    1/1     Running     0          130m
→ ~
```

6.5 Openwhisk running containers in cluster

Το επόμενο βήμα είναι να εγκαταστήσουμε στο μηχάνημα μας το Openwhisk-cli για να μπορούμε να αλληλοεπιδρούμε με αυτό.

Κατευθυνόμαστε στο σύνδεσμο: https://github.com/apache/openwhisk-cli/releases και από εκεί κατεβάζουμε την έκδοση που αντιστοιχεί στο λειτουργικό μας σύστημα.

Ακολουθούμε την ίδια λογική με το helm, οπότε κάνουμε unzip το αρχείο και μπαίνουμε στον root φάκελο που έχει δημιουργηθεί. Μπορούμε όπως με το helm να τοποθετήσουμε το αρχείο **wsk** μέσα στα user binaries για να έχουμε πρόσβαση σε αυτό από οποιοδήποτε σημείο του μηχανήματος.

Πριν πάμε να ορίσουμε τα βασικά properties του Openwhisk, θα δημιουργήσουμε έναν νέο χρήστη μέσα στο Openwhisk admin ο οποίος θα μπορεί να διαχειριστεί το namespace που φτιάξαμε προηγουμένως.

Εκτελούμε λοιπόν τις παρακάτω εντολές:

kubectl exec -ti --namespace openwhisk openwhisk-wskadmin – bash

Η παραπάνω εντολή μας βάζει εντός του Openwhisk admin container ώστε να μπορούμε να τρέξουμε εσωτερικά τις εντολές που χρειάζεται. Η επόμενη εντολή είναι η δημιουργία του χρήστη

wskadmin user create -ns openwhisk unipi

Η εντολή αυτή μας δίνει ως αποτέλεσμα ένα auth key για τον χρήστη unipi που μόλις δημιουργήθηκε και είναι της μορφής xxxx:yyyyy. Κρατάμε αυτό το κλειδί καθώς στην συνέχεια θα το χρησιμοποιήσουμε.

```
root@openwhisk-wskadmin: /
root@openwhisk-wskadmin:/# wskadmin user create -ns openwhisk unipi
2361dba7-290c-46e4-829c-b247619cd15e:r2ZV3rN5Z0crv2QheTULyvzZYN9DvqaVR4uBOrfsDZpQjoUW2wFiVhGkPJ3M5ZWI
root@openwhisk-wskadmin:/#
```

6.6 Openwhisk create new user

Η τελευταία εντολή που θα τρέξουμε είναι για να δώσουμε στο Openwhisk τα στοιχεία που χρειάζεται. Στην περίπτωση μας αυτά είναι το apihost και το auth που πήραμε από το προηγούμενο βήμα. Το apihost είναι η IP ή το domain name απ' όπου γίνεται expose το Kubernetes cluster προς τον έξω κόσμο. Στην περίπτωση μας επειδή είμαστε σε dev περιβάλλον αυτό είναι το localhost ή 127.0.0.1 και η πόρτα που αντιστοιχεί στον ngnix του Openwhisk είναι η 31001 που του δώσαμε στο yaml αρχείο κατά την εγκατάσταση.

Επομένως εκτελούμε τώρα την εντολή:

wsk property set --apihost https://127.0.0.1:31001 --auth 2361dba7-290c-46e4-829c-b247619cd15e:r2ZV3rN5Z0crv2QheTULyvzZYN9DvqaVR4uBOrfsDZpQjoUW2wFiVhGkPJ3M5ZWI

Για να βεβαιωθούμε ότι παραμετροποιήθηκε σωστά εκτελούμε στην συνέχεια:

wsk -i property get –all

```
spyros@spyross-mini: ~/Desktop/diploma/openwhisk/OpenWhisk_CLI-1.1.0-mac-amd64
→  OpenWhisk_CLI-1.1.0-mac-amd64 ./wsk -i property get --all
whisk API host          https://127.0.0.1:31001
whisk auth              2197fd95-207c-4d5d-bdbe-7e329c86bfc1:VPfS957SZzdJha24dZZGVJdCSr3PcRohlLlGUED1hhfi7q
85PnORMtV2NjZvlrkO
whisk namespace         openwhisk
client cert
Client key
whisk API version       v1
whisk CLI version       2020-10-02T00:33:38.010+0000
whisk API build         2020-10-07-10:25:41Z
whisk API build number  20201007a
→  OpenWhisk_CLI-1.1.0-mac-amd64
```

6.7 Openwhisk apply user credential through wsk cli

Η τελευταία δοκιμή που θα κάνουμε θα είναι ένα deploy ενός action στο Openwhiksk και μετά ένα invoke για να δούμε ότι λειτουργεί. Θα χρησιμοποιήσουμε το quarkus native app που δημιουργήσαμε νωρίτερα και θα κάνουμε deploy τον docker container.

Οπότε εκτελούμε τις ακόλουθες εντολές:

wsk -i action create echo-quarkus --docker nheidloff/quarkus-serverless:1

και στην συνέχεια:

wsk -i action invoke --blocking echo-quarkus --param name unipi


```
→  OpenWhisk_CLI-1.1.0-mac-amd64 ./wsk -i action invoke --blocking echo-quarkus --param name unipi
ok: invoked /_/echo-quarkus with id d44074bbae4947778074bbae493777f0
{
    "activationId": "d44074bbae4947778074bbae493777f0",
    "annotations": [
        {
            "key": "path",
            "value": "openwhisk/echo-quarkus"
        },
        {
            "key": "waitTime",
            "value": 2257
        },
        {
            "key": "kind",
            "value": "blackbox"
        },
        {
            "key": "timeout",
            "value": false
        },
        {
            "key": "limits",
            "value": {
                "concurrency": 1,
                "logs": 10,
                "memory": 256,
                "timeout": 60000
            }
        },
        {
            "key": "initTime",
            "value": 22
        }
    ],
    "duration": 35,
    "end": 1613389143332,
    "logs": [],
    "name": "echo-quarkus",
    "namespace": "openwhisk",
    "publish": false,
    "response": {
        "result": {
            "name": "unipi"
        },
        "size": 16,
        "status": "success",
        "success": true
    },
    "start": 1613389143297,
    "subject": "openwhisk",
    "version": "0.0.1"
}
```


6.8 Openwhisk cli action invocation with response

Το response από το action μας δείχνει ότι έχει γίνει deploy και λειτουργεί κανονικά.

7. Kubernetes

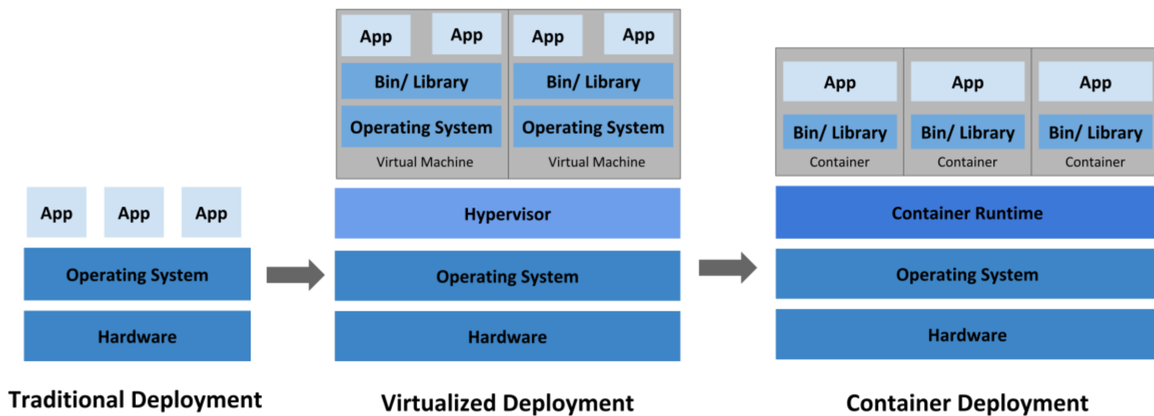
Το Kubernetes είναι ένα σύστημα ανοιχτού λογισμικού το οποίο αναπτύχθηκε αρχικά από την Google. Στόχος του είναι η διαχείριση εφαρμογών containers μέσα σε ένα cluster περιβάλλον.

Το Kubernetes έγινε διαθέσιμο για πρώτη φορά στους χρήστης τον Ιούνιο του 2014. Ο πιο συχνός ορισμός που του δίνεται είναι ως πλατφόρμα ενορχήστρωσης containers.

Σε ένα βασικό επίπεδο αναφορά, το Kubernetes είναι ένα σύστημα στο οποίο εκτελούνται και διαχειρίζονται διάφορες εφαρμογές οι οποίες τρέχουν μέσα σε containers σε πολλά διαφορετικά μηχανήματα (φυσικά μηχανήματα ή VM) τα οποία συγκροτούν ένα cluster. Διαχειρίζεται πλήρως όλο τον κύκλο ζωής των εφαρμογών και των υπηρεσιών που φιλοξενεί δίνοντας την δυνατότητα για υψηλή διαθεσιμότητα, ευέλικτη διαχείριση πόρων αλλά και κλιμάκωσης αυτών.

Οι χρήστες του Kubernetes μπορούν να καθορίσουν με ποιο τρόπο θα εκτελούνται οι εφαρμογές, πως θα αλληλοεπιδρούν με άλλες εφαρμογές ή με συστήματα και υπηρεσίες εκτός του cluster. Έχουν την δυνατότητα να κλιμακώσουν ή αποκλιμακώσουν τις εφαρμογές τους και να επαναφέρουν προηγούμενες εκδόσεις χωρίς χρονοβόρες διαδικασίες είτε μέσω των διεπαφών που δίνει το ίδιο το σύστημα (Kubernetes dashboard) είτε ακόμη και μέσω command line εντολών.

Το Kubernetes απέκτησε μεγάλη δημοτικότητα καθώς έλυνε προβλήματα αλλά και εξαφάνιζε περιορισμούς που εμφανίζονταν από την συνεχή αύξηση της χρήσης των containers στην ανάπτυξη και εκτέλεση των εφαρμογών.



7.1 Runtime comparison

7.1 Αρχιτεκτονική Kubernetes

Για να γίνει κατανοητή η λειτουργία και οι δυνατότητες του Kubernetes χρειάζεται να αναλυθεί πως δομείται σε υψηλό επίπεδο. Έχει σημασία να θεωρηθεί αρχικά ως ένα σύστημα το οποίο είναι βασισμένο σε πολλά επίπεδα καθένα από τα οποία αναλαμβάνει να μειώσει την πολυπλοκότητα που υπάρχει στα προηγούμενα επίπεδα.

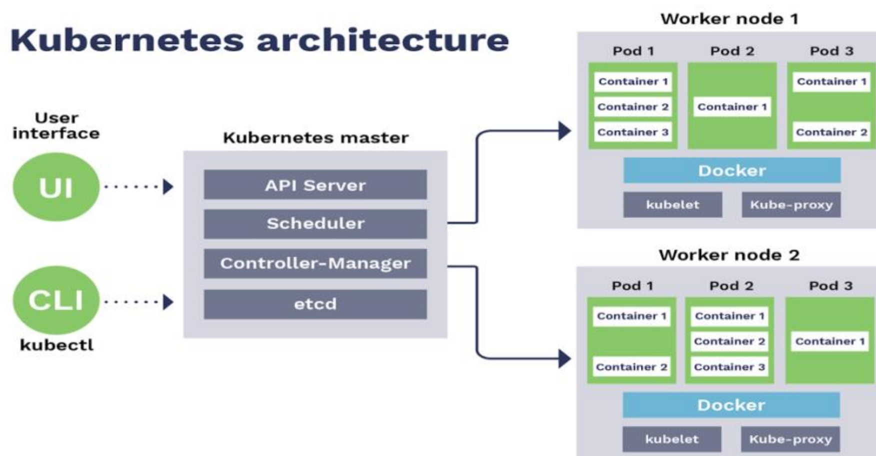
Σε πολύ υψηλό επίπεδο αυτό που κάνει το Kubernetes είναι να συνδέει σε ένα cluster, φυσικά μηχανήματα είτε virtual machines, τα οποία επικοινωνούν μεταξύ τους μέσω ενός εσωτερικού κοινού δικτύου. Πάνω σε αυτό το cluster λειτουργούν και παραμετροποιούνται όλες οι λειτουργικότητες και τα component του Kubernetes.

Τα μηχανήματα που απαρτίζουν το cluster έχουν ένα πολύ συγκεκριμένο ρόλο εντός του οικοσυστήματος. Απολύτως απαραίτητο είναι ένα από αυτά να λειτουργεί ως ο **master server**.

Ο ρόλος του master server ή αλλιώς main node είναι κομβικός για όλη την λειτουργία του cluster. Ο κόμβος αυτός λειτουργεί ως το κύριο και μοναδικό σημείο επικοινωνίας του συνόλου του cluster με χρήστες ή υπηρεσίες εκτός αυτού. Ο ρόλος του τον καθιστά υπεύθυνο για την διαχείριση των υπολοίπων κόμβων ή agents, την ενορχήστρωση όλων των διεργασιών που εκτελούνται εντός του cluster, τον συνεχή έλεγχο των άλλων κόμβων ώστε να επεμβαίνει στο «work split» αλλά και στην περίπτωση που κάποιος από αυτούς παρουσιάσει κάποιο πρόβλημα.

Οι agents, δηλαδή οι υπόλοιποι κόμβοι που είναι συνδεδεμένοι και αποτελούν μέρος του cluster είναι υπεύθυνοι να δέχονται και να εκτελούν εργασίες όπως αυτές τους ορίζονται από τον κεντρικό κόμβο. Οι οδηγίες δίνονται πάντα από τον κεντρικό κόμβο και αφορούν την δημιουργία ή καταστροφή containers, και την διαχείριση του δικτύου και των πόρων.

Όλοι οι κόμβοι του cluster τρέχουν εσωτερικά κάποια container runtime, στις περισσότερες περιπτώσεις αυτό είναι το Docker, ώστε να επιτύχουν την απομόνωση, τη συντήρηση και την ευελιξία των εφαρμογών που εκτελούν [17].



7.1.1 Δομικά στοιχεία Kubernetes server και worker

Όπως είδαμε και προηγουμένως, ο Kubernetes server, λειτουργεί ως το βασικό στοιχείο ελέγχου όλου του Kubernetes cluster. Για την διαχείριση του cluster από τους χρήστες είναι και το βασικό στοιχείο με το οποίο επικοινωνούν με το σύνολο του cluster.

Ο **Kubernetes server** αποτελείται από επιμέρους δομικά στοιχεία. Τα στοιχεία αυτά είναι τα εξής [18]:

- etcd
- kube-apiserver
- kube-controller-manager
- kube-scheduler
- cloud-controller-manager

Συνοπτικά θα αναφερθούμε στα παραπάνω στοιχεία

Etcd

Αποτελεί ένα από τα βασικά δομικά στοιχεία, τα οποία χρειάζεται το Kubernetes για να λειτουργήσει. Είναι ένας εσωτερικός χώρος που δεσμεύεται για να αποθηκεύει όλες τις παραμέτρους που απαιτούνται για την διαμόρφωση του συνόλου του cluster και αποτελείται από ένα «λεξικό» που περιέχει λήμματα σε μορφή key-value pairs.

Τα δεδομένα διαμόρφωσης τα οποία έχει αποθηκευμένα είναι προσπελάσιμα από όλους τους κόμβους του cluster και περιέχουν πληροφορίες όπως οδηγίες διαμόρφωσης και δικτύου προσπέλασης.

Οι πληροφορίες από το την αποθήκη δεδομένων είναι εύκολα προσπελάσιμη μέσω ενός HTTP / JSON API αλλά και μέσω διεπαφών που προσφέρει το ίδιο το Kubernetes.

Api-server

Μια άλλη πολύ σημαντική υπηρεσία του cluster είναι ο Api-server. Αποτελεί το κύριο στοιχείο που επιτρέπει την διαχείριση του φόρτου εργασίας των μονάδων του cluster. Επίσης διασφαλίζει τον συγχρονισμό και την διανομή των πληροφοριών και οδηγιών που βρίσκονται στο etcd.

Ο Api-server υλοποιεί ένα RESTful interface οπότε δίνει την δυνατότητα προσπέλασης του σε πολλούς διαφορετικούς μηχανισμούς από γλώσσες προγραμματισμού μέχρι και bash scripts. Ο προεπιλεγμένος μηχανισμός για την επικοινωνία ενός μηχανήματος με τον Api-server και κατ’ επέκταση του cluster είναι το **kubectl** που αποτελεί ένα εργαλείο γραμμής εντολών

Kube-controller-manager

Ο controller manager είναι μια υπηρεσία του Kubernetes που κυρίως διαχειρίζεται άλλους controllers οι οποίοι ελέγχουν και ρυθμίζουν την κατάσταση του cluster και ρυθμίζουν τον φόρτο εργασίας του.

Οι λεπτομέρειες των εργασιών που εκτελεί ο controller manager περιγράφονται και καταγράφονται στο etcd. Όταν παρουσιαστεί κάποια αλλαγή, τότε διαβάζει τις πληροφορίες που περιγράφουν την αλλαγή αυτή και φροντίζει να εκτελεστούν οι οδηγίες. Ένα παράδειγμα χρήσης του είναι πχ στην περίπτωση που κάποιο deployment config αρχείο ορίσει επιπλέον pod για ένα συγκεκριμένο container που τρέχει στο cluster. Τότε ο controller manager αναλαμβάνει να διαχειριστεί αυτή την αλλαγή και να αυξήσει τον αριθμό των pod.

Kube-scheduler

Ο kube scheduler είναι η διεργασία που κατανέμει τον φόρτο εργασίας σε πραγματικό χρόνο σε όλο το cluster. Αναγνωρίζει τις ανάγκες διάθεσης πόρων από τις διεργασίες που εκτελούνται μέσα σε ένα συγκεκριμένο χρονικό διάστημα και αναθέτει τις διεργασίες αυτές σε έναν ή περισσότερους κόμβους.

Επίσης γνωρίζει πλήρως όλους τους διαθέσιμους πόρους του cluster και κάθε μηχανήματος ξεχωριστά ώστε να τους υπερβαίνει και να μοιράζει το φόρτο με τέτοιο τρόπο ώστε όλοι οι κόμβοι να εξυπηρετούν σε ισορροπία.

Cloud-controller-manager

Ο cloud controller είναι η υπηρεσία του συστήματος η οποία παρέχει την δυνατότητα το Kubernetes να μπορεί να λειτουργήσει εξίσου και να εκμεταλλευτεί διαφορετικά περιβάλλοντα που το φιλοξενούν όπως για παράδειγμα όλα τα managed Kubernetes που προσφέρουν οι διάφοροι cloud providers.

Ο συγχρονισμός του και η παραμετροποίηση του στα διαφορετικά αυτά cloud περιβάλλοντα γίνεται μέσω του API του. Παραδείγματα Kubernetes cloud provider είναι το Azure Kubernetes, AWS Kubernetes service κλπ.

Τα επόμενα δομικά στοιχεία που θα αναφέρουμε είναι αυτά που περιέχονται στα **Kubernetes nodes** [18]:

- container runtime (docker runtime)
- kubelet
- kube-proxy

container runtime

Το πρώτο στοιχείο το οποίο είναι απαραίτητο για κάθε node server ή node ή agent είναι το container runtime. Το πιο δημοφιλές runtime σε αυτή την κατηγορία είναι το Docker παρόλο που υπάρχουν και ανταγωνιστικά προϊόντα μικρότερου όμως βεληνεκούς.

Ο ρόλος του runtime είναι η γενική διαχείριση των container που εκτελούνται καθώς και των εφαρμογών που βρίσκονται μέσα σε αυτός. Κάθε node εντός του cluster διαθέτει έναν τουλάχιστον container που εκτελείται μέσα σε αυτό. Το container runtime είναι το στοιχείο που κατευθύνει την λειτουργία αυτών των container με τις οδηγίες οι οποίες του έχουν ανατεθεί από το server node.

Kubelet

Το kubelet είναι η υπηρεσία που αναλαμβάνει την επικοινωνία του συγκεκριμένου κόμβου με τους υπόλοιπους κόμβους του cluster. Είναι υπεύθυνη για την μετάδοση των πληροφοριών από και προς τις υπηρεσίες ελέγχουν (master node) καθώς και την προσπέλαση του etcd για ανάγνωση ή και εγγραφή των απαραίτητων παραμέτρων λειτουργίας του κόμβου και των εφαρμογών του.

Η υπηρεσία αυτή συνδέεται και επικοινωνεί με τα υπόλοιπα στοιχεία για του master κόμβου για ζητήματα όπως η αυθεντικοποίηση του κόμβου μέσα στο cluster και την λήψη οδηγιών λειτουργίας. Οι οδηγίες του δίνονται μέσω των yaml αρχείων που γίνονται deploy από διαχειριστές και χρήστες και αναλαμβάνει την εκτέλεση και παύση των container μέσω του container runtime.

Kube-proxy

Το kube proxy είναι μια ενδιάμεση υπηρεσία η οποία αποτελείται από έναν proxy, ο οποίος είναι υπεύθυνος για την διαχείριση της προώθησης των request προς τους σωστούς container και να κάνει ένα load balancing στα request που δέχεται προς τους container του συγκεκριμένου node.

Στην συνέχεια θα αναφερθούμε συνοπτικά σε αντικείμενα που πλαισιώνουν ένα kubernetes cluster και είναι επίσης βασικά στοιχεία της λειτουργίας του:

- Pods
- Deployments
- Services
- Volumes
- Persistent volumes
- Ingress nginx
- Jobs
- Namespaces
- Annotations
- Labels
- Replication controllers / sets

Καθώς ο αριθμός των στοιχείων που απαρτίζουν το kubernetes είναι πολύ μεγάλος και σε κάποια από αυτά η πολυπλοκότητα τους δεν μπορεί να εξεταστεί πλήρως στα πλαίσια της εργασίας, θα εξετάσουμε τα βασικά που θα χρησιμοποιηθούν και στο cluster που θα δημιουργήσουμε.

Pods

Τα pod αποτελούν βασικό δομικό στοιχείο ενός cluster. Κάθε node απαρτίζεται από ένα ή περισσότερα pods τα οποία φιλοξενούν τους container. Οι containers οι οποίοι βρίσκονται στο ίδιο pod μοιράζονται συνήθως το δίκτυο και το σύστημα αρχείων που τους παρέχεται και τις περισσότερες φορές οι containers αυτοί είναι στενά συνδεδεμένοι μεταξύ τους ή αλληλεξαρτώμενοι.

Deployments

Τα συγκεκριμένα αντικείμενα εμπεριέχουν στη δομή τους μια συλλογή από pods καθώς και τον αριθμό των αντιγράφων κάθε pod που απαιτείται να εκτελεστεί. Τα deployments είναι ο τρόπος που συνίσταται για να δημιουργούμε pods σε ένα kubernetes cluster.

Οι εντολές που υπάρχουν σε ένα deployment αρχείο αναφέρουν πόσα pod θέλουμε να τρέχουμε ανά container και το kubernetes cluster φροντίζει να κρατάει αυτό τον αριθμό σταθερό. Αν παρατηρηθεί μια δυσλειτουργία σε κάποιο από αυτά τότε το kubernetes είτε προσπαθεί να κάνει επανεκκίνηση της λειτουργίας τους είτε το καταστρέφει και δημιουργεί ένα νέο για να διατηρηθεί ο αριθμός των ζητούμενων pod.

Οι εντολές των deployments δίνονται από τους διαχειριστές ή και χρήστες, σε yaml αρχεία και ακολουθούν συγκεκριμένο πρότυπο.

Service

Το service είναι ένα στοιχείο το οποίο εξασφαλίζει την επικοινωνία μεταξύ των pod. Βασικός του ρόλος είναι να παρέχει ένα κεντρικό σημείο επικοινωνίας από τον έξω κόσμο προς το pod και να διασφαλίζει ότι η επικοινωνία αυτή θα καταλήξει στο σωστό σημείο ακόμη και αν το ίδιο το Pod έχει κάποιο σφάλμα, έχει υποστεί αναβάθμιση κλπ.

Το σημαντικότερο σενάριο στο οποίο ένα service είναι απολύτως απαραίτητο είναι όταν ένα από τα pod στα οποία προωθείται ένα request εκείνη την στιγμή έχει πρόβλημα λειτουργίας. Το service γνωρίζει εξαρχής που πρέπει να ανακατευθύνει το request ώστε να εξυπηρετηθεί.

Αυτό συμβαίνει καθώς κάθε pod εντός του cluster έχει μια μοναδική ip. Στο service όμως περιγράφεται το όνομα της υπηρεσίες μόνο και η πόρτα που εξυπηρετεί. Επομένως μέσω του service discovery εσωτερικά ενός κόμβου μπορεί να βρει σε ποια pod εκτελείται εκείνη την στιγμή η συγκεκριμένη υπηρεσία που χρειάζεται.

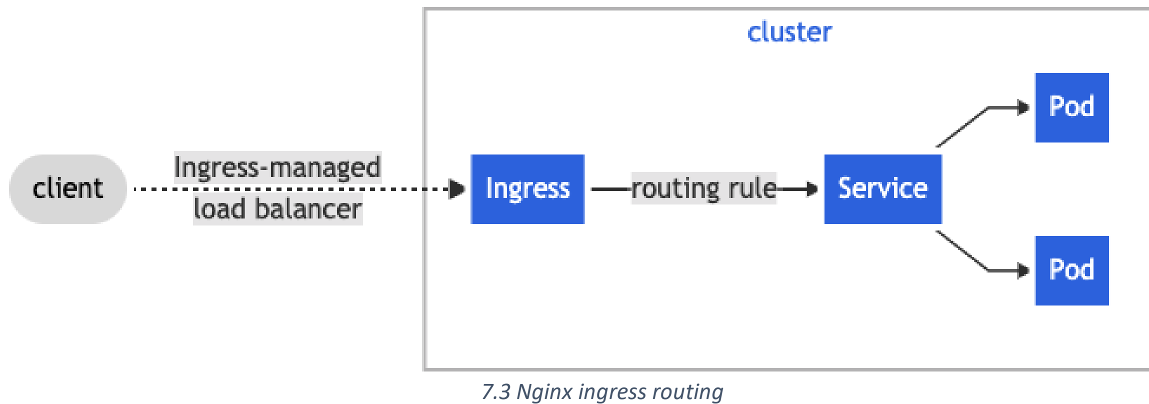
Τα service επίσης δίνονται από τους διαχειριστές ή τους χρήστες του συστήματος και είναι yaml αρχεία.

Ingress

Όπως αναφέρθηκε προηγουμένως το service είναι υπεύθυνο για την εσωτερική επικοινωνία των pods εντός του kubernetes cluster. Για να προσπελάσουμε όμως αντικείμενα που βρίσκονται εντός του cluster από εξωτερικό περιβάλλον χρειαζόμαστε το ingress.

Το kubernetes ingress είναι μια μικρογραφία ενός nginx η οποία κατευθύνει τα HTTP και HTTPS αιτήματα που προέρχονται εκτός του cluster προς τα services.

Τα παραπάνω στοιχεία αλληλοεπιδρούν μεταξύ τους όπως φαίνεται στην παρακάτω εικόνα:



7.3 Nginx ingress routing

Volumes

Το volumes αποτελούν μονάδες του kubernetes οι οποίες εξασφαλίζουν την κοινή χρήση δεδομένων από τους containers ενός pod. Στην περίπτωση που γίνει χρήση ενός volume εντός του pod τότε δεσμεύεται ένας κοινόχρηστος χώρος στον δίσκο ώστε να έχουν πρόσβαση όλοι οι containers.

Ακόμη και σε περιπτώσεις όπου κάποιος container χρειαστεί να επανεκκινήσει λόγω κάποιου προβλήματος, οι υπόλοιποι συνεχίζουν και έχουν κανονικά πρόσβαση στον κοινόχρηστο αυτό χώρο αποθήκευσης.

Η κύρια αδυναμία των volumes είναι στην περίπτωση που καταστρέφεται ένα pod και αναδημιουργείται. Στην περίπτωση αυτή τα δεδομένα που προυπήρχαν χάνονται και το νέο pod που δημιουργείται δεσμεύει εκ νέου χώρο στον δίσκο για τα δεδομένα.

Persistent volumes

Το παραπάνω πρόβλημα των volumes, λύνεται με την χρήση των persistent volumes. Τα συγκεκριμένα στοιχεία αφορούν κυρίως στοιχεία αποθήκευσης τα οποία είναι απαραίτητα ακόμη και αν καταστραφεί το pod όπως μια βάση δεδομένων.

Τα persistent volumes δεσμεύουν επίσης χώρο στο δίσκο αλλά δεν τον δένουν με την λειτουργία ενός συγκεκριμένου pod. Οπότε ακόμη και στην περίπτωση όπου το pod καταστραφεί και δημιουργηθεί καινούργιο, τα δεδομένα παραμένουν.

Ένας ακόμη ρόλος του persistent volume είναι να ελέγχει συνεχώς ότι το μέγεθος του volume δεν ξεπερνά το διαθέσιμο που έχει οριστεί κατά την δημιουργία του kubernetes cluster.

7.2 Εγκατάσταση local kubernetes cluster (docker for desktop version)

Για το demo περιβάλλον και για να έχουμε μια πρώτη εικόνα των στοιχείων και του τρόπου λειτουργίας ενός kubernetes cluster θα γίνει χρήση του kubernetes for docker desktop[7].

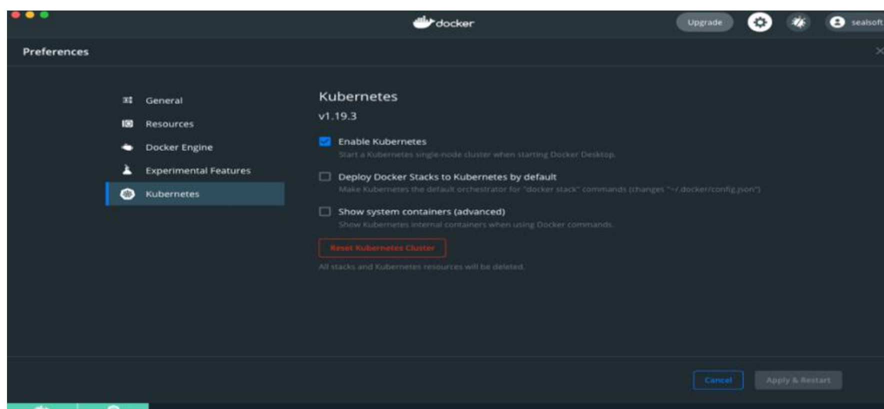
Το περιβάλλον αυτό δημιουργεί μόνο 1 κόμβο ο οποίος παίζει το ρόλο και του server και του worker και είναι κατάλληλος για development σε πρώτο επίπεδο. Όπως είδαμε και προηγουμένως στο συγκεκριμένο kubernetes cluster κάναμε deploy το Openwhisk.

Πέρα από το docker for desktop, μια ακόμη λύση για τοπική εγκατάσταση και χρήση ενός kubernetes cluster είναι το minikube. Το minikube είναι επίσης ένα εργαλείο διαθέσιμο για όλα τα λειτουργικά συστήματα το οποίο μπορούμε να χρησιμοποιήσουμε για να δημιουργήσουμε ένα single node kubernetes cluster.

Για να ενεργοποιήσουμε το Docker desktop kubernetes στο λειτουργικό μας σύστημα ακολουθούμε τα παρακάτω βήματα:

- Εκκινούμε το docker
- Επιλέγουμε από το μενού του docker το «preferences» και μεταφερόμαστε στο tab «kubernetes».
- Επιλέγουμε το checkbox «Enable kubernetes» και στην συνέχεια «apply & restart»

Το docker κάνει restart και ενεργοποιεί ταυτόχρονα το single node kubernetes.



7.4 Enable Kubernetes – Docker desktop version

Για να επιβεβαιώσουμε ότι το kubernetes έχει εκκινήσει σωστά εκτελούμε σε terminal την εντολή:
kubectl cluster-info

7.5 Kubernetes cluster info IP

Στην συνέχεια θα κάνουμε deploy το kubernetes ingress nginx για να μπορέσουμε να προσπελάσουμε το cluster από το μηχάνημα μας μέσω του dashboard που θα γίνει deploy στο επόμενο βήμα.

Η εντολή που πρέπει να τρέξουμε στο terminal είναι:
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v0.44.0/deploy/static/provider/cloud/deploy.yaml

Αμέσως το kubernetes μας ενημερώνει ότι το deploy ολοκληρώθηκε. Στην παραπάνω εντολή βλέπουμε τον τρόπο με τον οποίο μέσω του kubectl και των yaml αρχείων δίνουμε οδηγίες στο kubernetes server. Η εντολή **apply** χρησιμοποιείται στις πλειοψηφία των περιπτώσεων για να δώσουμε κάποιο yaml αρχείο που οδηγεί σε deployment.

Η επόμενη εντολή είναι για να διαπιστώσουμε ότι ο ingress nginx είναι σε λειτουργία:

kubectl get pods -n ingress-nginx

7.6 Ingress nginx namespace running pods

Για να δούμε την IP την οποία χρησιμοποιεί το cluster εκτελούμε:

kubectl describe nodes | grep InternalIP

Τέλος θα εγκαταστήσουμε το kubernetes dashboard. Επειδή το περιβάλλον που χρησιμοποιούμε είναι για development σκοπό θα χρειαστεί να απενεργοποιήσουμε τα certificates ώστε να έχουμε πρόσβαση σε αυτό χωρίς login του χρήστη με token[7].

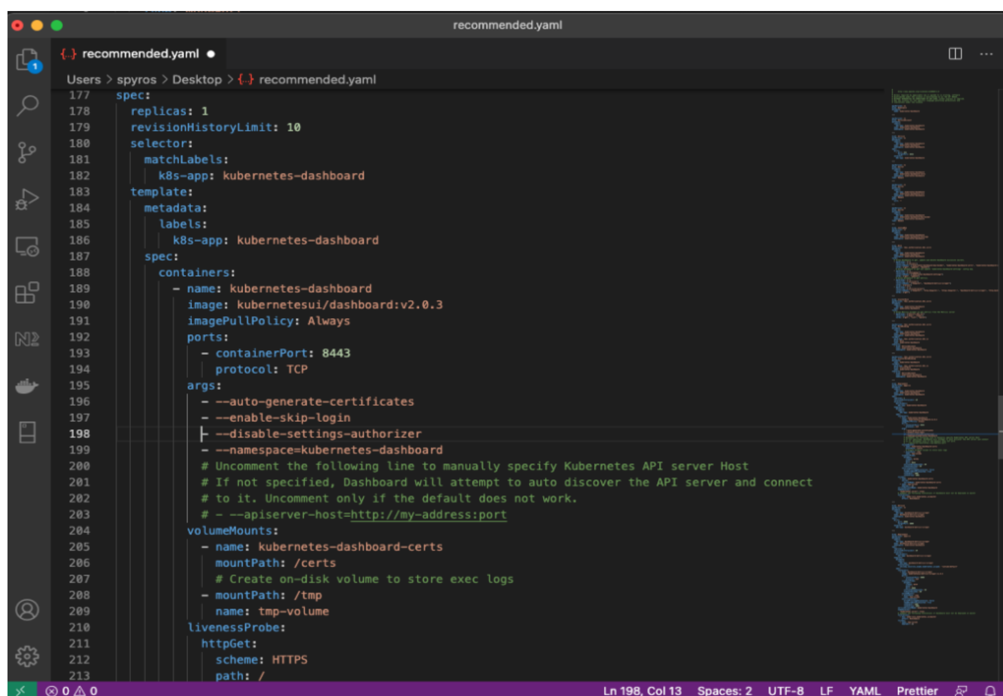
Αρχικά κατεβάζουμε τοπικά στο μηχάνημα μας το yaml αρχείο που βρίσκεται στην διεύθυνση:

https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.3/aio/deploy/recommended.yaml -Outfile kubernetes-dashboard.yaml

Ανοίγουμε το αρχείο σε ένα code editor για να το επεξεργαστούμε και στο section **args** προσθέτουμε τις τιμές:

1. ***- --auto-generate-certificates***
2. ***- --enable-skip-login***
3. ***- --disable-settings-authorizer***

Οι παραπάνω τιμές μας επιτρέπουν να μπορούμε να κάνουμε login στο dashboard χωρίς authentication. Το dashboard θα χρησιμοποιηθεί μόνο εντός του δικτύου και δεν θα βγαίνει στο internet.



7.7 Kubernetes dashboard yaml configuration

Αποθηκεύουμε το αρχείο και εκτελούμε στη συνέχεια σε terminal την εντολή:

kubectl apply -f recommended.yaml

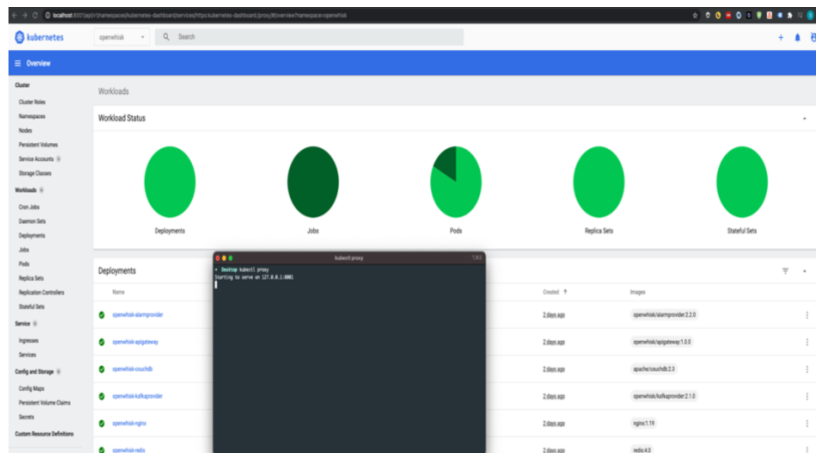
για να γίνει deploy στο cluster
Για να το τρέξουμε εκτελούμε την εντολή:

kubectl proxy

και στην συνέχεια όταν ξεκινήσει κατευθυνόμαστε στο URL:

[*http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/*](http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/)

Επιλέγουμε «skip» για να μην δώσουμε token στο login και μεταφερόμαστε στην κεντρική σελίδα απ' όπου βλέπουμε την κατάσταση στην οποία βρίσκεται αυτή τη στιγμή στο cluster μας



7.8 kubectl proxy & Kubernetes dashboard ui interface

Με αυτό τον τρόπο εκτελούμε μια πρώτη development εγκατάσταση ενός single node kubernetes στο μηχάνημα μας.

8. Δημιουργία και παραμετροποίηση Rpi kubernetes cluster

Για το τεχνικό σκέλος της εργασίας θα εγκαταστήσουμε ένα kubernetes cluster πάνω σε 4 κόμβους. Σχεδόν όλοι οι cloud providers παρέχουν την δυνατότητα να δημιουργήσει κάποιος το δικό του kubernetes cluster μέσα από κάποιο portal, διεπαφές, ακόμη και μέσω κάποιου command line interface που παρέχουν.

Η διαφορά στις παραπάνω δύο περιπτώσεις είναι κυρίως ότι ο cloud provider παρέχει συνήθως μια managed λύση, αφαιρώντας την πολυπλοκότητα της εγκατάστασης και της συντήρησης του cluster από τον χρήστη.

Σκοπός της εργασίες είναι η δημιουργία αυτού του cluster σε private περιβάλλον, και η εξέταση των δομικών του στοιχείων και της αλληλεπίδρασης μεταξύ τους.

8.1 Εξοπλισμός και εγκατάσταση

Για την δημιουργία του kubernetes cluster χρησιμοποιήσαμε τον παρακάτω εξοπλισμό:

- 4 x Raspberry pi 4 (4Gb ram)
- 4 x 64Gb SD cards
- 1 Netgear switch με 4 POE ports
- 5 x Cat-5 ethernet cables
- 4 x Raspberry pi POE HAT

8.1.1 Raspberry pi

Το raspberry pi είναι ένας single board υπολογιστής, ο οποίος αρχικά δημιουργήθηκε για εκπαιδευτικούς σκοπούς από το Raspberry Pi Foundation.

Βγήκε στην παραγωγή για πρώτη φορά το 2012 με επεξεργαστή ενός πυρήνα και συχνότητα 700MHz και μόλις 256Mb RAM. Στην συνέχεια συνεχώς εξελισσόταν και έγινε μια πολύ δημοφιλής λύση για IOT κυρίως εφαρμογές.

Σήμερα υπάρχει πλέον με επεξεργαστή 4 πυρήνων 64bit και RAM που φτάνει τα 8GB στην μεγαλύτερη του έκδοση. Το κόστος του είναι εξαιρετικά χαμηλό καθώς εκτείνεται από τα $5 στο pi zero και φθάνει ως τα $75 στην μεγαλύτερη έκδοση με 4 πυρήνες και 8GB RAM.

Για την εργασία θα χρησιμοποιήσουμε την ενδιάμεση έκδοση με τους 4 πυρήνες και 4GB ram. Επίσης για να αποφύγουμε την χρήση πολλών τροφοδοτικών για την λειτουργία τους, χρησιμοποιήσαμε το POE HAT.

Το POE HAT προέρχεται από «Hardware attached on top» και «power over ethernet». Το POE HAT είναι η πλακέτα που χρησιμοποιούμε, συνδέεται στο GPIO του raspberry και παρέχει τροφοδοσία ρεύματος μέσω του ethernet καλωδίου.

Για να υποστηρίξουμε τον τρόπο αυτό τροφοδοσίας χρησιμοποιήσαμε ένα switch με POE ports το οποίο παρέχει 55W / port, που είναι επαρκές ώστε να λειτουργήσει πλήρως το Raspberry.

8.1.2 Εγκατάσταση και παραμετροποίηση Raspberry Pi OS :

Θα χρειαστεί να εγκαταστήσουμε και στις 4 συσκευές μας το official λειτουργικό σύστημα του raspberry, γνωστό και ως rasbian ή αλλιώς raspberry pi os. Για τα raspberry υπάρχουν αρκετά διαθέσιμα λειτουργικά συστήματα όπως noobs, ubuntu και γενικά αρκετές από τις διανομές του Linux οι οποίες έχουν γίνει port για arm συσκευές ώστε να λειτουργούν σε hardware όπως αυτό του raspberry.

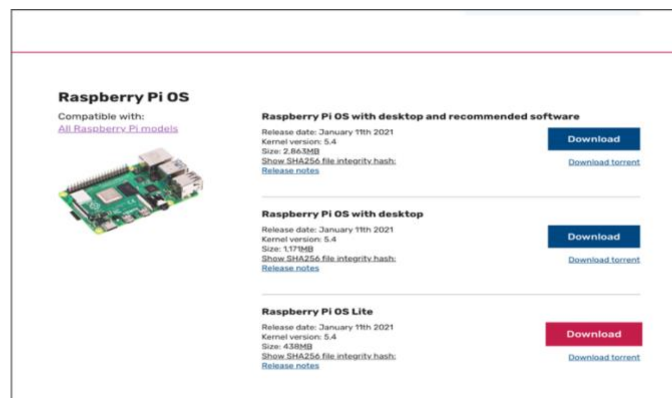
Ωστόσο επειδή το επίσημο λειτουργικό του είναι το rasbian θα επιλέξουμε αυτό ώστε να έχουμε συχνά updates και αρκετά μεγάλη κοινότητα διαθέσιμη.

Συγκεκριμένα το λειτουργικό μας θα είναι η headless έκδοση. Το raspberry pi os προσφέρει 3 διαφορετικές εκδόσεις.

1. Raspberry pi os lite
2. Raspberry pi os
3. Raspberry pi os full

Εμείς θα επιλέξουμε την πρώτη έκδοση καθώς δεν χρειαζόμαστε γραφικό περιβάλλον ούτε επιπλέον λογισμικό στο λειτουργικό μας αλλά μόνο τα απολύτως απαραίτητα ώστε να λειτουργήσει ως server.

Για να κατεβάσουμε την τελευταία έκδοση μεταφερόμαστε στην σελίδα:
https://www.raspberrypi.org/software/operating-systems/
Και στην έκδοση Raspberry Pi OS Lite επιλέγουμε «Download». Στο μηχάνημα μας κατεβαίνει η έκδοση ως zip αρχείο.



8.1 Raspberry pi official OS versions

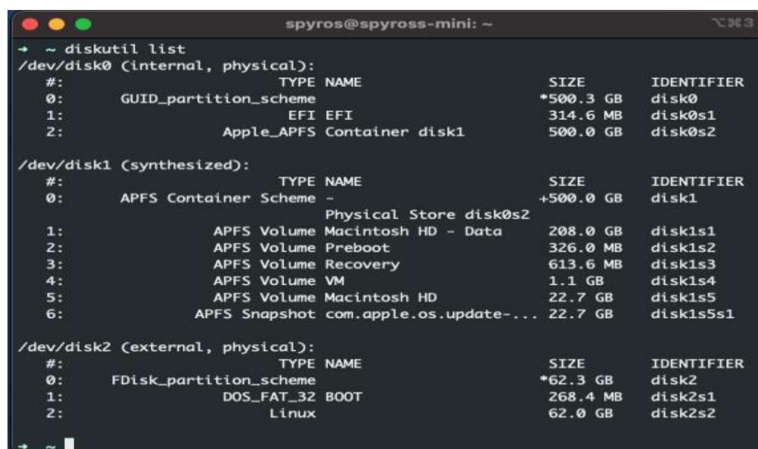
Κάνουμε unzip το αρχείο και τοποθετούμε το .img αρχείο που περιέχει σε ένα φάκελο σε οποιοδήποτε σημείο του συστήματος μας εξυπηρετεί.

Για να προχωρήσουμε με την εγκατάσταση χρειαζόμαστε ένα SD card reader ή κάποιο adapter για να εισάγουμε τις SD κάρτες και να αντιγράψουμε εκεί το .img αρχείο.

Ένας εύκολος τρόπος να εγκαταστήσουμε το raspberry pi os στην SD κάρτα είναι το λογισμικό etcher, πληροφορίες για το οποίο μπορούμε να βρούμε στον σύνδεσμο: https://www.balena.io/etcher/

Σε UNIX συστήματα ο προτεινόμενος τρόπος είναι μέσω του disk utility σε terminal το οποίο θα ακολουθήσουμε και εδώ στα παρακάτω βήματα:

- Εισάγουμε την sd κάρτα στο μηχάνημα μας.
- Όταν το μηχάνημα μας αναγνωρίσει την κάρτα που μόλις βάλαμε, σε ένα terminal δίνουμε την εντολή: ***diskutil list*** για να δούμε που έχει γίνει mount.



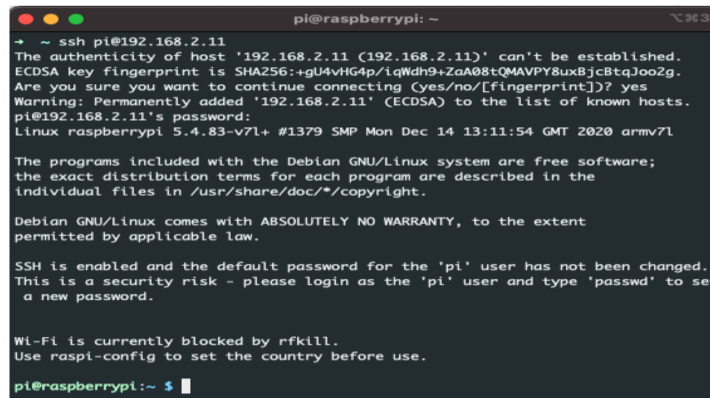
8.2 Mounted SD card partition

- Στην περίπτωση μας η κάρτα έχει γίνει mount στο **/dev/disk2**. Η επόμενη μας κίνηση είναι να κάνουμε unmount την κάρτα με την εντολή: ***diskutil unmountDisk /dev/disk2***
- Στην συνέχεια πηγαίνουμε στον φάκελο που βρίσκεται το .img αρχείο που κατεβάσαμε και από εκεί εκτελούμε την εντολή: ***sudo dd bs=1m if=2021-01-11-raspios-buster-armhf-lite.img of=/dev/rdisk2; sync*** . Δίνουμε το password του χρήστη που μας ζητάει και προχωράει στην αντιγραφή του περιεχομένου.
- Μια ακόμη ενέργεια που χρειάζεται να γίνει ώστε να έχουμε ssh πρόσβαση στο Rpi είναι να δημιουργήσουμε μέσα στο boot φάκελο ένα αρχείο ssh. Για να το πετύχουμε αυτό εκτελούμε την εντολή: ***touch /Volumes/boot/ssh***
- Κάνουμε πάλι unmount την κάρτα και την αφαιρούμε από το μηχάνημα μας.

Την διαδικασία αυτή θα πρέπει να την ακολουθήσουμε για όλες τις sd κάρτες που θα χρησιμοποιήσουμε (4 στην δική μας περίπτωση).

Όταν ολοκληρωθεί η διαδικασία τοποθετούμε τις κάρτες στα rpi και τα εκκινούμε. Θα χρειαστούν κάποιες ακόμη ρυθμίσεις για να δώσουμε static ip και να ενεργοποιήσουμε τα cgroup ώστε να μπορούμε μετά να στήσουμε το kubernetes.

Ξεκινάμε με ssh στα rpi. Η προκαθορισμένη πόρτα για ssh είναι οι 22 και username: **pi** / password: **raspberry**

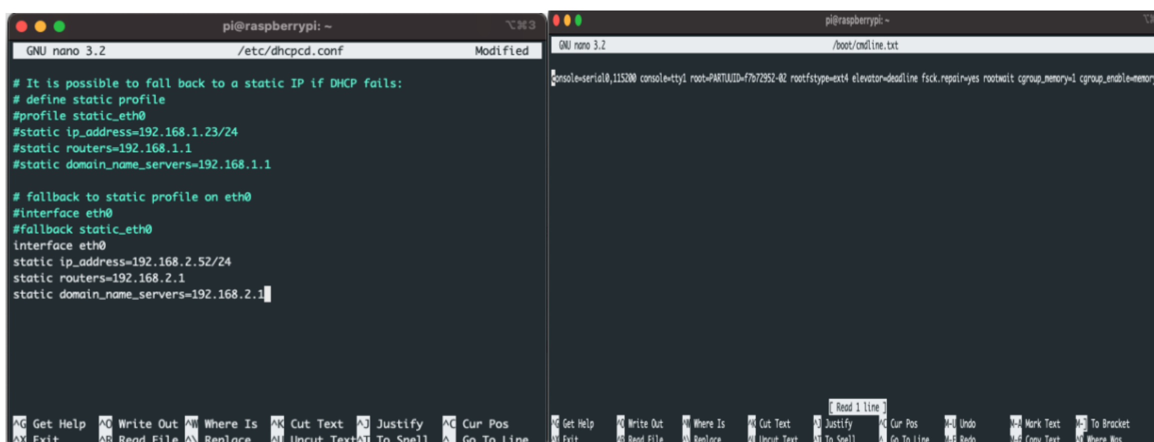


8.3 First ssh

Θα ξεκινήσουμε να επεξεργαστούμε το αρχείο dhcpcd.conf. Το path που βρίσκεται αυτό το αρχείο είναι: ***/etc/dhcpcd.conf***

Θα προσθέσουμε την ip που θέλουμε να έχει στο εσωτερικό δίκτυο καθώς και την ip του router που στην συγκεκριμένη περίπτωση είναι και ο dns. Επειδή είναι συνδεδεμένα στο δίκτυο ethernet το interface πάνω στο οποίο θα δηλώσουμε τις τιμές μας είναι το eth0.

Επίσης θα προσθέσουμε τις τιμές: ***cgroup_memory=1*** και ***cgroup_enable=memory*** στο αρχείο ***/boot/cmdline.txt***



8.4 setting ip and enable cgroup

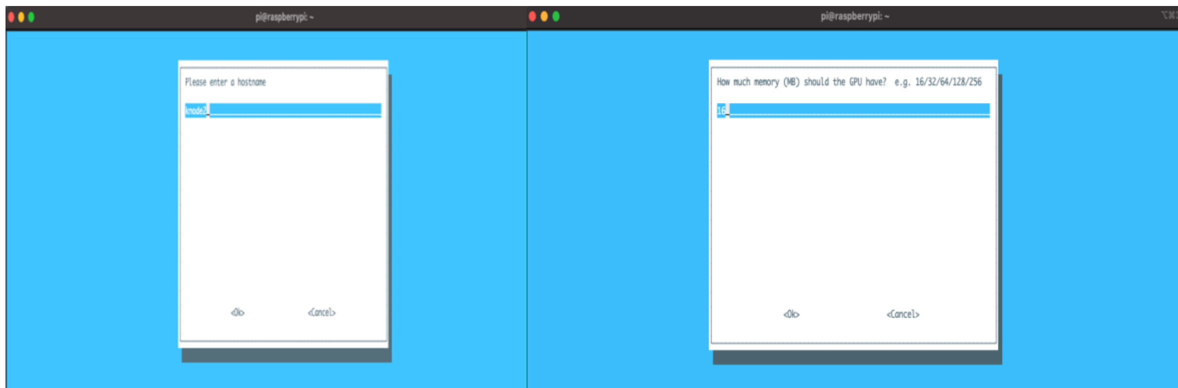
Για να κατανοήσουμε πως θα παραμετροποιήσουμε συνολικά το cluster στα επόμενα βήματα θα χρησιμοποιήσουμε το παρακάτω σενάριο:

- Το server node θα πάρει hostname: kmaster και ip: 192.168.2.50
- Τα υπόλοιπα node θα πάρουν hostname: knode1-3 και ip: 192.168.2.51-53

Προχωράμε με την αλλαγή του hostname, και μείωση της μνήμης της gpu. Εκτελούμε την εντολή: ***sudo raspi-config***

Στην οθόνη που μας εμφανίζεται για να αλλάξουμε το hostname επιλέγουμε «***System Options***» και στην συνέχεια «***Hostname***». Αλλάζουμε την ονομασία σε αυτή που επιθυμούμε και επιλέγουμε «OK». Μετά επιλέγουμε «***Performance Options***» και από εκεί «***GPU Memory***» και αλλάζουμε την τιμή σε 16 και «OK».

Επιλέγουμε «Finish» και μας ζητάει να κάνει επανεκκίνηση όπου του επιλέγουμε «YES» ώστε να πάρει όλες τις νέες ρυθμίσεις που του έχουμε προσθέσει. Την διαδικασία αυτή την ακολουθούμε επίσης για όλα τα rpi του cluster μας.



8.5 Configure hostname & gpu memory

Το τελευταίο που θα κάνουμε για την ολοκλήρωση της εγκατάστασης του λειτουργικού είναι να κάνουμε update όλα τα πακέτα του ώστε να πάρουμε τις τελευταίες αναβαθμίσεις του. Αυτό το κάνουμε με την εντολή: ***sudo apt update && sudo apt dist-upgrade***

8.2 Δημιουργία Kubernetes cluster

Το επόμενο βήμα μετά την ολοκλήρωση της εγκατάστασης του λειτουργικού συστήματος είναι η εγκατάσταση και η ρύθμιση του kubernetes cluster. Για το συγκεκριμένο cluster θα χρησιμοποιήσουμε μια light έκδοση του kubernetes, την οποία ανέπτυξε η Rancher Labs και ονομάζεται k3s.

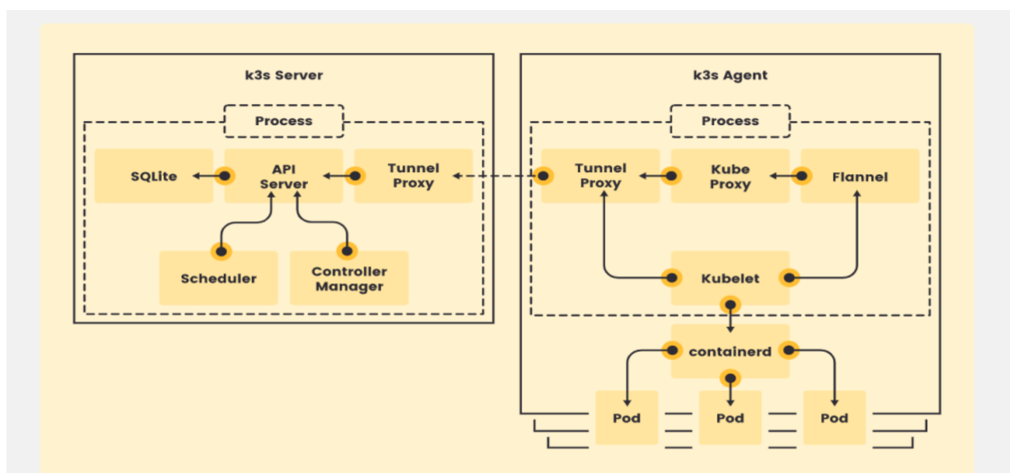
Το k3s είναι μια επίσημα αναγνωρισμένη έκδοση του kubernetes (k8s) που χρησιμοποιείται κυρίως για ARM επεξεργαστές, όπως στην περίπτωση μας στα raspberry pi. Το κυριότερο πλεονέκτημα του είναι ότι το k3s αποτελείται από μόλις ένα αρχείο με μέγεθος μικρότερο από 100mb και η διαδικασία εγκατάστασης και παραμετροποίησης του είναι πολύ πιο ξεκάθαρη.

Περιέχει το σύνολο των βασικών στοιχείων όπως και το k8s όπως api server, kube proxy, controller manager, scheduler κλπ. Έχουν αφαιρεθεί όλα τα στοιχεία τα οποία δεν ήταν απαραίτητα και έχουν προστεθεί άλλα στοιχεία με μικρότερη κατανάλωση σε μνήμη και επεξεργαστική ισχύ.

Ένα από τα παραδείγματα είναι η αφαίρεση του ingress nginx που χρησιμοποιείται για την επικοινωνία του cluster από τον έξω κόσμο και την θέση του πήρε το Traefic ingress controller με πολύ μικρότερες απαιτήσεις σε hardware.

Επίσης αφαιρέθηκε το etcd που ήταν η βάση δεδομένων στην οποία τοποθετούνταν όλες οι οδηγίες για την λειτουργία του συνόλου του cluster και την θέση του πήρε μια βάση δεδομένων SQLite η οποία είναι πολύ δημοφιλής ειδικά σε κινητές συσκευές.

Η αρχιτεκτονική του k3s ακολουθεί την παρακάτω δομή [14]:



8.6 K3s architecture

Τέλος αξίζει στην αναφορά μας προς το k3s να τονίσουμε πως υποστηρίζεται πλήρως από την πλειοψηφία των cloud providers. Αυτό σημαίνει ότι αν θελήσουμε στο μέλλον να μεταφέρουν την υποδομή μας σε κάποια cloud υπηρεσία αυτό θα είναι εφικτό.

8.2.1 Δημιουργία server κόμβου

Η εγκατάσταση του kubernetes cluster ξεκινάει από τον κόμβο που θα ορίσουμε ως server. Στον συγκεκριμένο κόμβο έχουμε δώσει hostname: kserver και ip: 192.168.2.50 κατά την παραμετροποίηση των raspberry.

Η εντολή που θα χρειαστούμε για να ξεκινήσει η εγκατάσταση είναι:

curl -sfL https://get.k3s.io | sh -

Η εγκατάσταση θα διαρκέσει μερικά δευτερόλεπτα και στο τέλος θα πάρουμε το μήνυμα «Starting k3s-server». Για να ελέγξουμε ότι η διαδικασία ολοκληρώθηκε με επιτυχία, αφού αφήσουμε μερικά ακόμη δευτερόλεπτα ώστε να εκκινήσουμε όλα τα απαραίτητα process, εκτελούμε την εντολή:

sudo kubectl get nodes

Η επόμενη ενέργεια μας είναι να εξάγουμε το cluster token που έχει δημιουργηθεί ώστε να το δηλώσουμε στην συνέχεια στους agent για να μπορέσουν να πιστοποιηθούν στο δίκτυο. Για να γίνει αυτό εκτελούμε την εντολή:

sudo cat /var/lib/rancher/k3s/server/node-token

Την τιμή που θα μας επιστρέψει η εντολή αυτή, θα την κρατήσουμε καθώς θα χρησιμοποιηθεί στην συνέχεια.

8.7 Retrieve k3s node token

8.2.2 Δημιουργία agent κόμβων

Μετά την ολοκλήρωση του server ξεκινάμε με το παραμετροποίηση των agent nodes. Η διαδικασία που ακολουθεί θα επαναληφθεί και στους 3 κόμβους με ρόλο agent του cluster.

Η διαδικασία είναι παρόμοια με αυτή που ακολουθήσαμε στον server με την διαφορά ότι εδώ θα χρειαστεί να δηλώσουμε την ip του server και να δώσουμε το token ώστε να ενσωματώσουμε τον κόμβο αυτό στο cluster.

Αφού κάνουμε πάλι ssh στον κόμβο, δίνουμε την παρακάτω εντολή:

curl -sfL https://get.k3s.io | K3S_URL=https://192.168.2.50:6443 K3S_TOKEN={token} sh -

Στην τιμή του {token} δίνουμε την τιμή που κάναμε στο προηγούμενο βήμα extract από τον server.

```
pi@node3: ~
pi@node3:~ $ curl -sfL https://get.k3s.io | K3S_URL=https://192.168.2.50:6443 K3
S_TOKEN=K1020971c730d000c7adef2801020fb65d266b79e20197419f3695b67e055851fd8::ser
ver:b24363f2b314b32604e467afcd35c655 sh -
[INFO]  Finding release for channel stable
[INFO]  Using v1.20.2+k3s1 as release
[INFO]  Downloading hash https://github.com/rancher/k3s/releases/download/v1.20.
2+k3s1/sha256sum-arm.txt
[INFO]  Downloading binary https://github.com/rancher/k3s/releases/download/v1.2
0.2+k3s1/k3s-armhf
[INFO]  Verifying binary download
[INFO]  Installing k3s to /usr/local/bin/k3s
[INFO]  Creating /usr/local/bin/kubectl symlink to k3s
[INFO]  Creating /usr/local/bin/crictl symlink to k3s
[INFO]  Creating /usr/local/bin/ctr symlink to k3s
[INFO]  Creating killall script /usr/local/bin/k3s-killall.sh
[INFO]  Creating uninstall script /usr/local/bin/k3s-agent-uninstall.sh
[INFO]  env: Creating environment file /etc/systemd/system/k3s-agent.service.env
[INFO]  systemd: Creating service file /etc/systemd/system/k3s-agent.service
[INFO]  systemd: Enabling k3s-agent unit
Created symlink /etc/systemd/system/multi-user.target.wants/k3s-agent.service →
/etc/systemd/system/k3s-agent.service.
[INFO]  systemd: Starting k3s-agent
pi@node3:~ $
```

8.8 Init k3s worker and apply server token and ip

Μετά την ολοκλήρωση αυτού του βήματος σε όλους τους agent, επιβεβαιώνουμε το cluster ότι είναι σωστά ρυθμισμένο κάνοντας ssh στο master και δίνοντας την εντολή:

sudo kubectl get nodes

```
pi@kserver: ~
pi@kserver:~ $ sudo kubectl get nodes
NAME      STATUS   ROLES                  AGE    VERSION
knode1    Ready    <none>                 30h    v1.20.2+k3s1
knode2    Ready    <none>                 25h    v1.20.2+k3s1
kserver   Ready    control-plane,master   33h    v1.20.2+k3s1
node3     Ready    <none>                 113s   v1.20.2+k3s1
pi@kserver:~ $
```

8.8 get nodes verify

Για να πάρουμε περισσότερες πληροφορίες για όλους τους κόμβους χρησιμοποιούμε την εντολή: ***sudo kubectl describe nodes***

8.2.3 Απομακρυσμένη πρόσβαση στο cluster

Το επόμενο βήμα είναι να ενεργοποιήσουμε την πρόσβαση στο kubernetes cluster από κάποιο άλλο μηχάνημα του δικτύου μας για να αποφύγουμε την ανάγκη να χρειάζεται να κάνουμε ssh ή να ανοίξουμε ssh tunnel στο server node ώστε να τρέχουμε εκεί τις εντολές[14].

Για την διαδικασία αυτή στο μηχάνημα που χρησιμοποιούμε για απομακρυσμένη πρόσβαση θα χρειαστεί να εγκαταστήσουμε το kubernetes cli ή αλλιώς kubectl. Στο κεφάλαιο που παρουσιάσαμε το Openwhisk δόθηκαν οι απαραίτητες οδηγίες πως να εγκατασταθεί για όλα τα λειτουργικά συστήματα. Αναφέραμε με βήματα την εγκατάσταση του ***kubernetes-helm*** το οποίο είναι το εργαλείο που χρειαζόμαστε και σε αυτή την περίπτωση.

Ως πρώτο βήμα θα χρειαστεί να πάρουμε το configuration yaml από το server κόμβο και να το μεταφέρουμε στο local μηχάνημα μας. Η τοποθεσία που βρίσκεται το yaml αρχείο του cluster είναι στο /etc/rancher/k3s/k3s.yaml. Εκτελούμε λοιπόν την εντολή:

sudo cat /etc/rancher/k3s/k3s.yaml

και αντιγράφουμε το περιεχόμενο του αρχείου.

```
pi@kserver:~ $ sudo cat /etc/rancher/k3s/k3s.yaml
```

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUJkekNDQVIyZ0F3SUJBZ0lCQURBS0JnZ3Foa2p
PUFFRREFqQWpNU0V3SHdZRFZRUUREQmhyTTNNdGMyVnkKZG1WeUxXTmhRREUyTVRNMk5ESXhPRGt3SGhjTk1qRXdNakU0TURrMU5qSTVXaGNOTX
pFd01qRTJNRGsxTmpJNQpXakFqTVNFd0h3WURWUVFEREJock0zTXRjMlZ5ZG1WeUxXTmhRREUyTVRNMk5ESXhPRGt3V1RBVEJnY3Foa2pPClBRS
UJCZ2dxaGtqT1BRTUJCd05DQUFUVGFjdWNPY3lweDR4ZFE5b29INTZhQy9vWGhzbE1GY09MekJxRDBQNlgKMVFxNXZmVkRqS05PR2RtbG1UNE9i
WDAreU05d3dlSXVuNzNrTUNianZLYllvMEl3UURBT0JnTlZIUThCQWY4RQpCQU1DQXFRd0R3WURWUjBUQVFIL0JBVXdBd0VCL3pBZEJnTlZIUTR
FRmdRVXV4blhvS0o5bzdQSU9mVDZqSWxrClZxSkNjL0l3Q2dZSUtvWkl6ajBFQXdJRFNBQXdSUUlnYUNXb3Y0ek5RRGJBYUxMYXFjaGlvMjBtTU
RCdGJsblMKa2p2WEFjRDdhb2NDSVFEYnMwME84MzdLY1RtUnU4S1BWS2R6RlQ2aVBpZXFZZTZ5K2x4SjBBUWRKUT09Ci0tLS0tRU5EIENFUlRJR
klDQVRFLS0tLS0K
    server: https://127.0.0.1:6443
  name: default
contexts:
- context:
    cluster: default
    user: default
  name: default
current-context: default
kind: Config
preferences: {}
users:
- name: default
  user:
    client-certificate-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUJrakNDQVRlZ0F3SUJBZ0lJZUFHamM5VjdMTWd3Q2
dZSUtvWkl6ajBFQXdJd0l6RWhNQjhHQTFVRUF3d1kKYXpOekxXTnNhV1Z1ZEMxallVQXhOakV6TmpReU1UZzVNQjRYRFRJeE1ESXhPREE1TlRZe
U9Wb1hEVEl5TURJeApPREE1TlRZeU9Wb3dNREVYTUJVR0ExVUVDaE1PYzNsemRHVnRPbTFoYzNSbGNuTXhGVEFUQmdOVkJBTVRESE41CmMzUmxi
VHBoWkcxcGJqQlpNQk1HQnlxR1NNNDlBZ0VHQ0NxR1NNNDlBd0VIQTBJQUJLTkFzbUI2SldEcm5yYTAKb3l4SUhXRVFsSGNpZVdqTlVkeUtPR29
6RXRwV3BzWmZ0QWxwY2E0M1c0RVptSTBSaUF5cWg3Y3QrM3hjZnVLVwpRRUJpRnp1alNEQkdNQTRHQTFVZER3RUIvd1FFQXdJRm9EQVRCZ05WSF
NVRUREQUtCZ2dyQmdFRkJRY0RBakFmCkJnTlZIU01FR0RBV2dCUUhkTGlQMEpTR3BvUXhWTEkrTjRNTWxsTkx6ekFLQmdncWhrak9QUVFEQWdOS
kFEQkcKQWlFQTBFQTlFRWZOdVVPcXdnaU55NmZYVGZhY2w3NlZnYkdXdUxQUFNjenFCQVlDSVFENUVlNGREdzdMOGx0Tgo4cjZuTTBsSGowV09E
alRwS3BveDk1WFFLZzJDelE9PQotLS0tLUVORCBDRVJUSUZJQ0FURS0tLS0tCi0tLS0tQkVHSU4gQ0VSVElGSUNBVEUtLS0tLQpNSUlCZGpDQ0F
SMmdBd0lCQWdJQkFEQUtCZ2dxaGtqT1BRUURBakFqTVNFd0h3WURWUVFEREJock0zTXRZMnhwClpXNTBMV05oUURFMk1UTTJOREl4T0Rrd0hoY0
5NakV3TWpFNE1EazFOakk1V2hjTk16RXdNakUyTURrMU5qSTUKV2pBak1TRXdId1lEVlFRRERCaHJNM010WTJ4cFpXNTBMV05oUURFMk1UTTJOR
El4T0Rrd1dUQVRCZ2NxaGtqTwpQUUlCQmdncWhrak9QUU1CQndOQ0FBUzU4UlNoWFlqKy9nMldrSTFaaFcwK3NNQVNUMC90OVVDdldHeW9NUFVR
CjZxZGQ4MVNsNjBOajlDdXZIN2dxZFpZS2tNditDZXZpT0JvWWJOVWQvRHBBbzBJd1FEQU9CZ05WSFE4QkFmOEUKQkFNQ0FxUXdEd1lEVlIwVEF
RSC9CQVV3QXdFQi96QWRCZ05WSFE0RUZnUVVCM1M0ajlDVWhxYUVNVlN5UGplRApESlpUUzg4d0NnWUlLb1pJemowRUF3SURSd0F3UkFJZ0pHdj
l4MzVkeldOQ0JiVmZ5UmVmRnZYekV4RHlmYkw4CjVoR1cycFhkczN3Q0lDWUQ0VTRBYXJad0phOG9BUktwV3luZlVueDZJaUpXVDYzTVlad3VQd
1AzCi0tLS0tRU5EIENFUlRJRklDQVRFLS0tLS0K
    client-key-data: LS0tLS1CRUdJTiBFQyBQUklWQVRFIEtFWS0tLS0tCk1IY0NBUUVFSUdnZDlkMXFXa2FMZWFFaEJHZzRhUW1VUm1yTl
hpeG1IdEU4QXp6YmxJQ0FvQW9HQ0NxR1NNNDkKQXdFSG9VUURRZ0FFbzBDeVlIb2xZT3VldHJTakxFZ2RZUkNVZHlKNWFNMVIzSW80YWpNUzJsY
W14bCswQ1dseApyamRiZ1JtWWpSR0lES3FIdHkzN2ZGeCs0cFpBUUdJWE93PT0KLS0tLS1FTkQgRUMgUFJJVkFURSBLRVktLS0tLQo=
```

```
pi@kserver:~ $
```

8.9 k3s yaml configuration file

Στο μηχάνημα μας στο root folder υπάρχει ένα αρχείο, που έχει προκύψει από την εγκατάσταση του kubernetes helm του προηγούμενου κεφαλαίου με την ονομασία ***.kube/config***

Το αρχείο αυτό ως τώρα περιέχει τις πληροφορίες για την σύνδεση με το local kubernetes που στήσαμε στο docker for desktop. Αυτό που θα κάνουμε για να μην χρειαστεί να εγκαταστήσουμε περισσότερα εργαλεία, καθώς δεν υπάρχει επίσημος τρόπος να διαχειριζόμαστε πολλαπλά cluster από το ίδιο μηχάνημα αλλά μόνο 3rd party εργαλεία, είναι να μετονομάσουμε το config αρχείο σε **config-local**

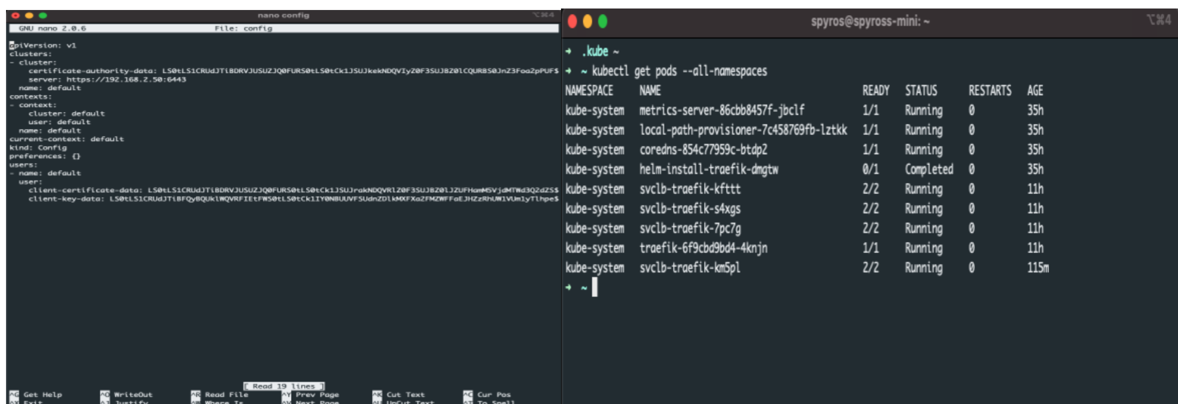
Αυτό το κάνουμε εκτελώντας την εντολή **mv config config-local.**

Στην συνέχεια θα δημιουργήσουμε ένα νέο config αρχείο και θα αντιγράψουμε μέσα το yaml από το kubernetes cluster. Για να δημιουργήσουμε το αρχείο εκτελούμε την εντολή: ***nano config***

Κάνουμε επικόλληση το yaml εντός του αρχείου και αλλάζουμε την τιμή localhost ή 127.0.0.1 στο πεδίο «**server**» με την ip ή το hostname του server μας, δηλαδή ***192.168.2.50 ή kmaster***

Αποθηκεύουμε και βγαίνουμε από τον editor.

Για να ελέγξουμε ότι η ρύθμιση έγινε με σωστό τρόπο εκτελούμε την εντολή: ***kubectl get nodes*** στο μηχάνημα μας. Στην συνέχεια για να δούμε όλα τα pods που τρέχουν σε όλα τα namespaces εκτελούμε την εντολή: ***kubectl get pods --all-namespaces***



8.10 edit local kubeconfig and verify remote access

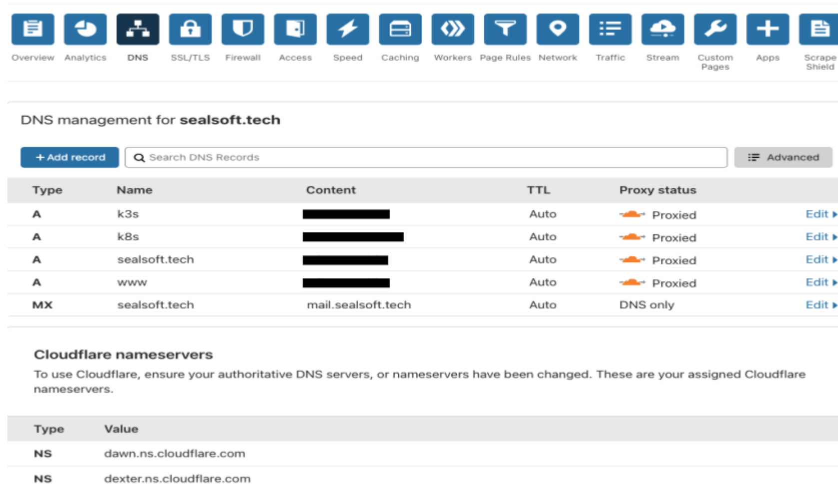
8.3 Cloudflare και SSL setup

8.3.1 Dns proxy & port forward

Η επόμενη ενέργεια που θα χρειαστεί να κάνουμε είναι να χρησιμοποιήσουμε το Cloudflare ως dns proxy. Ουσιαστικά θέλουμε το domain name που έχουμε στην κατοχή μας να προωθεί τα request προς το kubernetes cluster.

Η πρώτη κίνηση για να το επιτύχουμε αυτό είναι να δημιουργήσουμε έναν λογαριασμό στο Cloudflare και από εκεί να δηλώσουμε το domain name μας. Στην συνέχεια αφού γίνει verify το domain name, στο tab «***dns***» θα χρειαστεί να κάνουμε forward ένα subdomain μας προς την local ip address. Σημαντικό είναι να σημειώσουμε ότι αν δεν έχουμε static ip τότε αυτή περιοδικά αλλάζει από τον πάροχο επομένως η διαδικασία αυτή θα είναι επαναλαμβανόμενη.

Η ενημέρωση των dns χρειάζεται από μερικά λεπτά μέχρι κάποιες ώρες, επομένως υπάρχει πιθανότητα να μην έχουμε άμεση πρόσβαση στο kubernetes cluster μέσω του domain name. Υπό κανονικές συνθήκες δεν θα χρειαζόμαστε να έχουμε πρόσβαση στο cluster μέσω internet και θα είχαμε αποφύγει την διαδικασία. Στην περίπτωση μας όμως, η κλήση προς το cluster θα προέρχεται από το slack event api.



8.11 Cloudflare dns proxy

Στην συνέχεια παίρνουμε τους name server που δίνει το Cloudflare και τους δηλώνουμε στο domain μας. Αυτό γίνεται από το dashboard της υπηρεσίας από την οποία κατοχυρώσαμε το domain name ώστε τα request να δρομολογούνται προς αυτή την υπηρεσία.

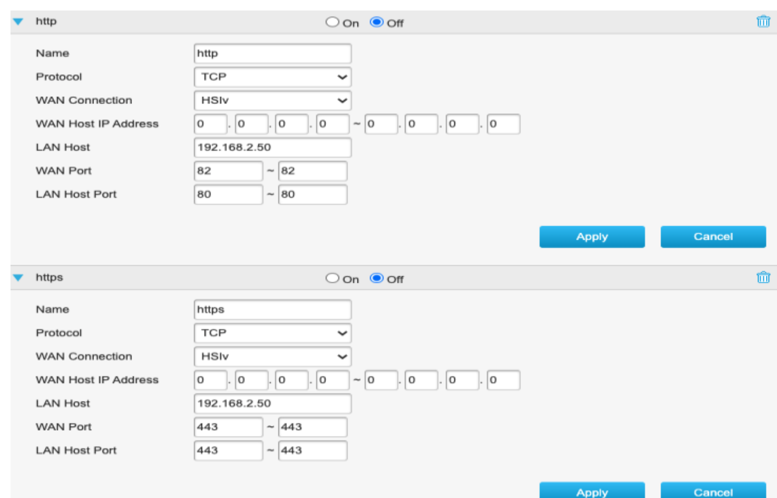
8.12 Declare Cloudflare name servers

Για να επιβεβαιώσουμε ότι πλέον το subdomain ή το domain μας δρομολογούνται στην σωστή ip εκτελούμε σε ένα τερματικό την εντολή: ***dig +short k3s.sealsoft.tech***
Το Cloudflare έχει την δυνατότητα να κρύψει την πραγματική ip που καταλήγει οποιοδήποτε request προς το domain που του έχουμε δηλώσει, οπότε εδώ βλέπουμε ότι το request τερματίζει στους 2 nameserver της υπηρεσίες.

```
→  ~ dig +short k3s.sealsoft.tech
172.67.215.111
104.21.69.232
```

8.13 Verify dns proxy

Απομένει να κάνουμε ανοίξουμε τις αντίστοιχες πόρτες στο router μας ώστε να επιτρέψουμε την πρόσβαση στο δίκτυο μας από το internet. Το interface που παρέχει το κάθε router για port forward διαφέρει, επομένως θα αναφέρουμε μόνο ότι χρειάζεται να γίνει forward η public port 80 στην private port 80 και αντίστοιχα την public port 443 στην private 443. Ως private ip δίνουμε το node server εκεί δηλαδή όπου βρίσκεται και ο ingress nginx load balancer. Στο δικό μας setup αυτό αντιστοιχεί στην ***192.168.2.50***



8.14 Router port forward

8.3.2 Let's Encrypt ssl certificates

Το επόμενο βήμα είναι να εγκαταστήσουμε ένα ssl certificate στον ingress nginx – Traefic ώστε να μπορούμε να επικοινωνούμε με το cluster στην πόρτα 443 πάνω από το https πρωτόκολλο.

Για να κάνουμε issue ένα certificate, αρχικά συνδεόμαστε με ssh στο server μηχάνημα – 192.168.2.50 και θα χρειαστεί να τρέξουμε ένα cert manager pod το οποίο θα μας κάνει στην συνέχεια issue ένα valid ssl certificate από την Let's Encrypt.

Οι οδηγίες για την εγκατάσταση ενός certificate βρίσκονται αναλυτικά στο επίσημη σελίδα του cert-manager: https://cert-manager.io/docs/installation/kubernetes/

Η κύρια διαφορά που υπάρχει είναι ότι το δικό μας cluster είναι arm αρχιτεκτονικής. Συνεπώς θα χρειαστεί να επεξεργαστούμε το cert-manager.yaml και να προσθέσουμε σε κάθε image name ως post fix ***-arm***. Αυτό είναι απαραίτητο σε 3 γραμμές τις οποίες επισυνάπτουμε παρακάτω:

image: quay.io/jetstack/cert-manager-cainjector-arm:v1.2.0
image: quay.io/jetstack/cert-manager-controller-arm:v1.2.0
image: quay.io/jetstack/cert-manager-webhook-arm:v1.2.0

Κάνουμε τις παραπάνω αλλαγές και στην συνέχεια ακολουθούμε πλήρως τις οδηγίες του cert-manager. Στην διαδικασία μας προτρέπει πρώτα να κάνουμε ένα staging issuer για να δοκιμάσουμε ότι οι ip και domain name μας λειτουργούν κανονικά και στην συνέχεια αφού επιτύχει η διαδικασία αυτή να προχωρήσουμε με το production certificate.

9. Αρχιτεκτονική και εκτέλεση εφαρμογών στο cluster

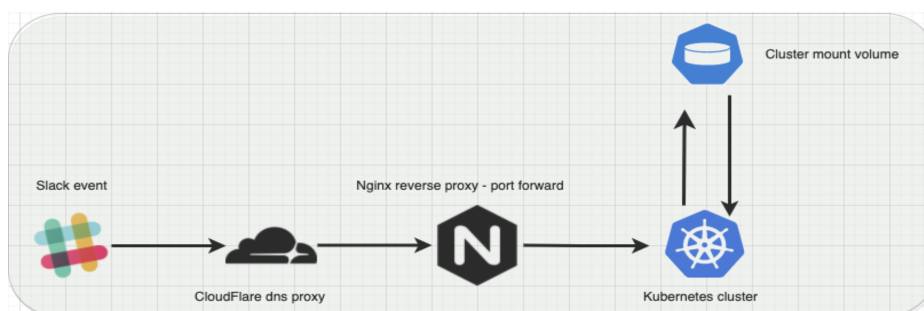
Στο κεφάλαιο αυτό θα εξετάσουμε πως μπορούμε μέσα στο Openwhisk το οποίο θα τρέχει στο kubernetes cluster μπορούμε να αλληλοεπιδράσουμε και να δημιουργήσουμε ένα workflow με το slack.

Στόχος μας θα είναι στην τελική φάση να έχουμε μια σειρά από εφαρμογές εντός του Openwhisk οι οποίες θα διαχειρίζονται κλήσεις από το Slack api και θα απαντούν με event στο ίδιο. Η πρώτη εφαρμογή που θα χρησιμοποιήσουμε θα είναι μια quarkus εφαρμογή η οποία θα γίνει native build όπως δείξαμε και στο προηγούμενο κεφάλαιο.

Η δεύτερη εφαρμογή θα είναι μια εφαρμογή η οποία είναι ενσωματωμένη στο Openwhisk και αποτελεί το integration που θέλουμε με το Slack. Στο τέλος θα δούμε πως αυτές οι εφαρμογές συνδυάζονται μεταξύ τους για να δημιουργηθεί μια αλληλουχία και να επιστραφεί η πληροφορία πίσω στον χρήστη. Όλο το περιβάλλον που χρησιμοποιούμε αλλά και οι ίδιες οι εφαρμογές ακολουθούν την serverless αρχιτεκτονική και λειτουργούν με message queues.

9.1 Αρχιτεκτονική προσέγγιση

Στην λύση που θα παρουσιάσουμε θέλουμε να επιτύχουμε μια συγκεκριμένη αρχιτεκτονική σχεδίαση. Το αρχικό σχήμα που αφορά την γενική εικόνα είναι το παρακάτω διάγραμμα:



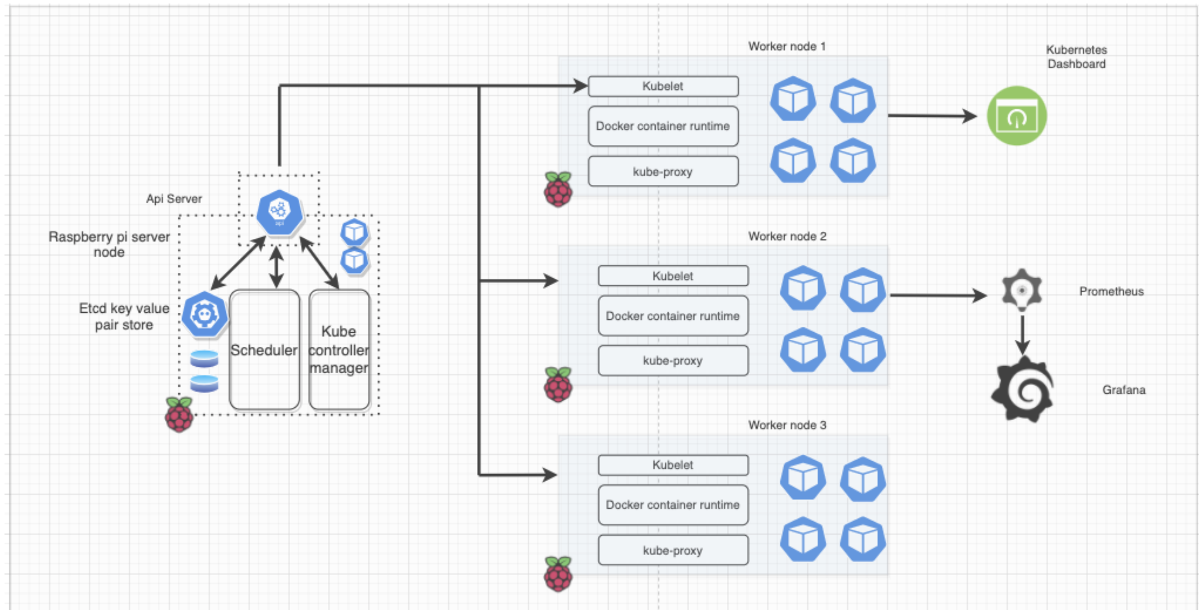
9.1 Overall sequence diagram

Παραπάνω το διάγραμμα απεικονίζει σε high level τον τρόπο με τον οποία θα λειτουργεί και θα αλληλοεπιδρά το slack με το cluster μας. Συγκεκριμένα:

- Το request θα γίνεται trigger μέσω του outgoing webhook api του slack και της λέξης «weather».
- Το webhook api του slack έχει ως αποδέκτη του request το domain μας το οποίο καταλήγει στον dns proxy της Cloudflare.
- Στην Cloudflare το domain μας γίνεται proxy προς την public ip του δικτύου μας. Η κλήση καταλήγει σε έναν nginx proxy ο οποίος θα κάνει forward την πόρτα 80 προς την αντίστοιχη πόρτα του kubernetes cluster όπου θέλουμε να επικοινωνήσουμε (31001)
- Στην υποδομή έχουμε προσθέσει επιπλέον ένα mount storage volume που είναι ένας εξωτερικός σκληρός δίσκος που παίζει το ρόλο του NAS εντός του δικτύου. Ο κύριος λόγος

που έγινε αυτό είναι ότι τα raspberry pi ως δίσκο χρησιμοποιούν SD κάρτες οι οποίες δεν έχουν αντοχή στον χρόνο και σε συνεχόμενα read-write που απαιτείται κυρίως από το etcd του kubernetes και από την CouchDB του Openwhisk

Ακολουθεί το διάγραμμα της αρχιτεκτονικής του kubernetes cluster που έχουμε δημιουργήσει εντός του δικτύου μας:



9.2 Kubernetes cluster architecture

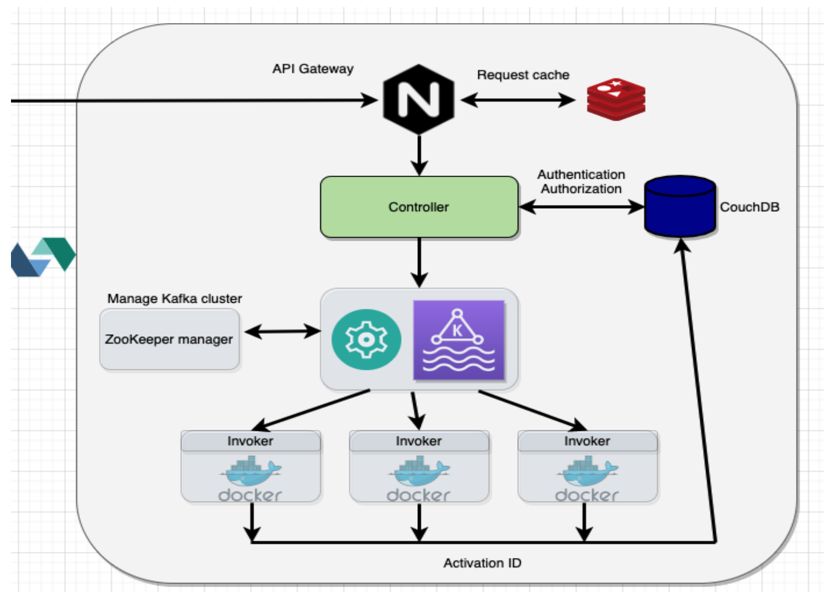
Στο παραπάνω διάγραμμα βλέπουμε το σύνολο του kubernetes cluster το οποίο αποτελείται από 4 κόμβους συνολικά. Αριστερά έχουμε την δομή του kubernetes server κόμβου και δεξιά τα kubernetes worker.

Κάθε worker έχει τον δικό του docker runtime ώστε να μπορεί να δημιουργεί τα images τα οποία του γίνονται assign να εκτελέσει από τον controller-manager του server. Κάθε κλήση που φτάνει στο kubernetes cluster έχει ως πρώτο σημείο αναφοράς τον api server. Ο api server στην συνέχεια ρωτάει συνεχώς το etcd store ώστε να δει που είναι διαθέσιμοι οι απαραίτητοι κόμβοι και σε ποια nodeport μπορεί να επικοινωνήσει μαζί τους.

Ο scheduler είναι υπεύθυνος για να ρυθμίζει ποιες εφαρμογές θα τρέξουν σε ποιους κόμβους και σε πόσα pods. Ο api server επικοινωνεί με τα worker nodes μέσω των kubelet που τρέχουν σε κάθε node ξεχωριστά.

Επίσης τα worker nodes τρέχουν ένα kubernetes dashboard το οποίο μας δίνει μια εποπτική εικόνα του συνόλου του cluster καθώς και ένα Prometheus το οποίο σε συνδυασμό με το Grafana μπορεί να μας δώσει metrics, alerts αλλά και dashboard ui για την κατάσταση του cluster χρησιμοποιώντας queries προς αυτό.

Στο επόμενο διάγραμμα παρουσιάζουμε την υποδομή του Openwhisk και πως αυτό λειτουργεί εντός του cluster:



9.3 Openwhisk architecture

Στο παραπάνω διάγραμμα παρουσιάζουμε το σύνολο του Openwhisk μαζί με όλα τα δομικά του στοιχεία.

Ένα request φθάνει αρχικά στο API Gateway, το οποίο στην συνέχεια το προωθεί στον controller. Σε ένα kubernetes cluster μπορεί να τρέχουν πολλούς controller και σε διαφορετικά μηχανήματα καθώς γίνονται εύκολα scale – stateless.

Ο controller επικοινωνεί με την CouchDB για να κάνει authenticate το request. Το request έχει πάνω του auth headers που αναπαριστούν τα credentials ενός χρήστη της εφαρμογής και πολλών ή συγκεκριμένου namespace. Το CouchDB έχει όλη αυτή την πληροφορία διαθέσιμη.

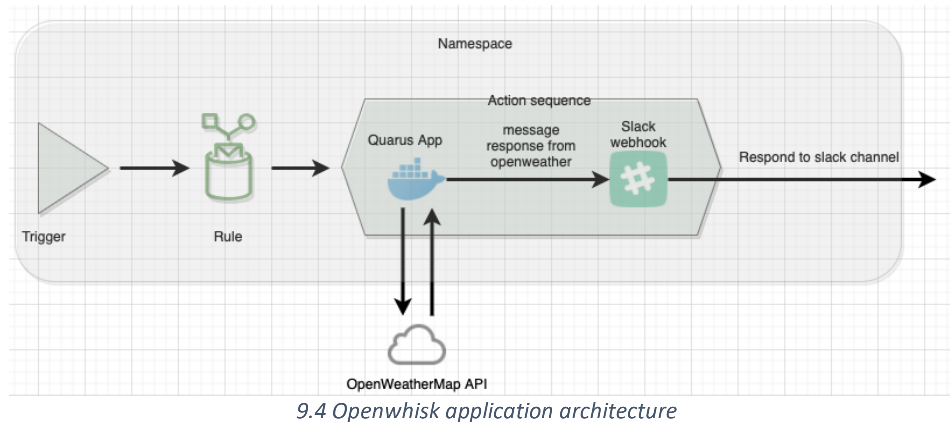
Μετά το πέρας το authentication ο controller αποστέλλει την πληροφορία στο Kafka cluster, το οποίο με την σειρά του το κάνει broadcast προς τους διαθέσιμους invokers. Εδώ ο ZooKeeper έχει τον ρόλο του διαχειριστή του Kafka cluster και επικοινωνεί συνεχώς μαζί του ώστε να βεβαιώνει τον συγχρονισμό του σε όλο το cluster.

Όταν το request καταλήξει εν τέλει στον invoker, αυτός είναι που θα το εκτελέσει. Αρχικά θα ζητήσει από το CouchDB τον κώδικα που πρέπει να κάνει inject στον docker container που θα χρειαστεί να τρέξει την εφαρμογή. Στην συνέχεια θα εντοπίσει αν τρέχει ήδη κάποιος container στην cache του ώστε να μην χρειαστεί να δημιουργήσει κάποιον καινούργιο.

Στην περίπτωση που υπάρχει ήδη έτοιμος container είτε κάνει warm up και εκτελεί το request είτε αν είναι ενεργός ο container και διαθέσιμος εκτελεί απευθείας τον κώδικα. Μετά το πέρας της εκτέλεσης κάνει expose ένα activation id το οποίο το αποθηκεύει και στην CouchDB ώστε να ενημερώσει για την επιτυχή εκτέλεση του.

Στην περίπτωση τώρα που δεν έχει κάποιο docker container στην διάθεση του ζητάει το αντίστοιχο image από το docker hub ή όποιο άλλο docker repository έχουμε δηλώσει ως διαθέσιμο. Κάνει build το image, εκκινεί τον container και στο τέλος τρέχει τον κώδικα. Ακολουθεί η ίδια διαδικασία με το activation id.

Το τελευταίο διάγραμμα που θα αναλύσουμε είναι αυτό της εφαρμογής μας:



9.4 Openwhisk application architecture

Στο τελευταίο μας διάγραμμα παρουσιάζουμε τον τρόπο με τον οποίο θα λειτουργούν οι εφαρμογές εντός του Openwhisk.

Στο πρώτο στάδιο το request θα ενεργοποιεί τον trigger μέσω του api gateway και του REST endpoint που έχει δημιουργηθεί. O trigger με την σειρά του θα είναι συνδεδεμένος με ένα rule το οποίο όταν ενεργοποιείται καταλήγει στην κλήση ενός action sequence.

Τα δύο actions που θα χρησιμοποιήσουμε είναι το quarkus application, που είναι ουσιαστικά ο rest client για να επικοινωνούμε με το OpenWeatherMap API και το slack webhook το οποίο ανήκει στα build in actions του Openwhisk.

Επιλέξαμε να το εκτελέσουμε ως sequence of actions διότι υπάρχει εξάρτηση μεταξύ των δυο επιμέρους actions που το απαρτίζουν. Όταν ο invoker κάνει inject τον κώδικα στον quarkus container αυτός με την σειρά του θα κάνει ένα GET request εκτός του cluster στο API του OpenWeatherMap.

Το response από το api θα είναι και το input parameter στο slack webhook. Το slack webhook action με την σειρά του θα πάρει το response και θα το κάνει POST στο incoming webhook api του slack κάτω από συγκεκριμένο URL το οποίο θα του δώσουμε κατά την δημιουργία του.

Με αυτό τον τρόπο ολοκληρώνεται ο κύκλος του πρώτου διαγράμματος και το μήνυμα που γίνεται POST από το Openwhisk action εμφανίζεται στο slack κανάλι σε μορφή κειμένου.

9.2 Quarkus Weather App

Το πρώτο σκέλος της εφαρμογής μας θα είναι ένας client ο οποίος θα επικοινωνεί με ένα api για να λαμβάνει δεδομένα καιρού.

Τα δεδομένα καιρού θα τα ζητάμε από το api του https://openweathermap.org/ στο οποίο χρειάζεται απλώς ο χρήστης να κάνει έναν λογαριασμό ώστε να πάρει api key. Εξετάζοντας στην συνέχεια το api documentation βλέπουμε ότι χρειαζόμαστε μόνο την ονομασία της πόλης που μας ενδιαφέρει ώστε να πάρουμε τα δεδομένα καιρού.

API call

```
api.openweathermap.org/data/2.5/weather?q={city name}&appid={API key}
```

```
api.openweathermap.org/data/2.5/weather?q={city name},{state code}&appid={API key}
```

```
api.openweathermap.org/data/2.5/weather?q={city name},{state code},{country code}&appid={API key}
```

Parameters

q	required	City name, state code and country code divided by comma, use ISO 3166 country codes. You can specify the parameter not only in English. In this case, the API response should be returned in the same language as the language of requested location name if the location is in our predefined list of more than 200,000 locations.
appid	required	Your unique API key (you can always find it on your account page under the "API key" tab)
mode	optional	Response format. Possible values are xml and html. If you don't use the mode parameter format is JSON by default. Learn more
units	optional	Units of measurement. standard, metric and imperial units are available. If you do not use the units parameter, standard units will be applied by default. Learn more
lang	optional	You can use this parameter to get the output in your language. Learn more

9.5 openweathermap api path and parameters

Οι παράμετροι που θα δώσουμε επιπλέον στον client είναι το «units» για να μας επιστρέψει σε metric τις τιμές. Συνεπώς η κλήση θα είναι ***GET*** και της μορφής ***https://api.openweathermap.org/data/2.5/weather?q={city name}&appid={API key}&units=metric***

Για το Quarkus project θα χρειαστεί να μορφοποιήσουμε αυτό που είδαμε σε προηγούμενο κεφάλαιο. Ο κώδικας της εφαρμογής τον οποίο θα δούμε λίγο συνοπτικά βρίσκεται στο GitHub: https://github.com/SealSoft/unipi-msc

Θα χρησιμοποιήσουμε επίσης μόνο βιβλιοθήκες που υποστηρίζονται από το quarkus και το GraalVM ώστε να μπορέσουμε να κάνουμε native build. Το project έχει την παρακάτω δομή:

9.6 quarkus client package and class structure

Ξεκινάμε από τα 2 Dockerfile που θα χρειαστούμε για την εφαρμογή. Το πρώτο είναι το Dockerfile.jvm και το δεύτερο το Dockerfile.native. Δεν θα μπούμε σε πολλές λεπτομέρειες για τα συγκεκριμένα αρχεία, αξίζει όμως να αναφέρουμε ότι το δεύτερο Dockerfile είναι τροποποιημένο ώστε να μην χρειάζεται το μηχάνημα που θα γίνει το build να έχει εγκατεστημένο το GraalVM καθώς χρησιμοποιεί ένα ενδιάμεσο image για αυτή την διαδικασία [9].

Στην συνέχεια κάτω από το package ***boundary*** υπάρχουν τα endpoint που γίνονται expose από το quarkus και τον client που είναι το interface όπου το quarkus επικοινωνεί με το openweathermap. Η ιδιαιτερότητα στις 2 κλάσεις ***InitResource*** και ***WeatherResource*** είναι τα path που έχουν.

Το Openwhisk για να μπορέσει να λειτουργήσει με το quarkus χρειάζεται ένα path: ***/init*** στο οποίο ελέγχει αν ο container έχει δημιουργηθεί και μπορεί να απαντήσει (health check) και ένα path: ***/run*** το οποίο εκτελείται όταν φτάσει κάποια κλήση προς το συγκεκριμένο action.

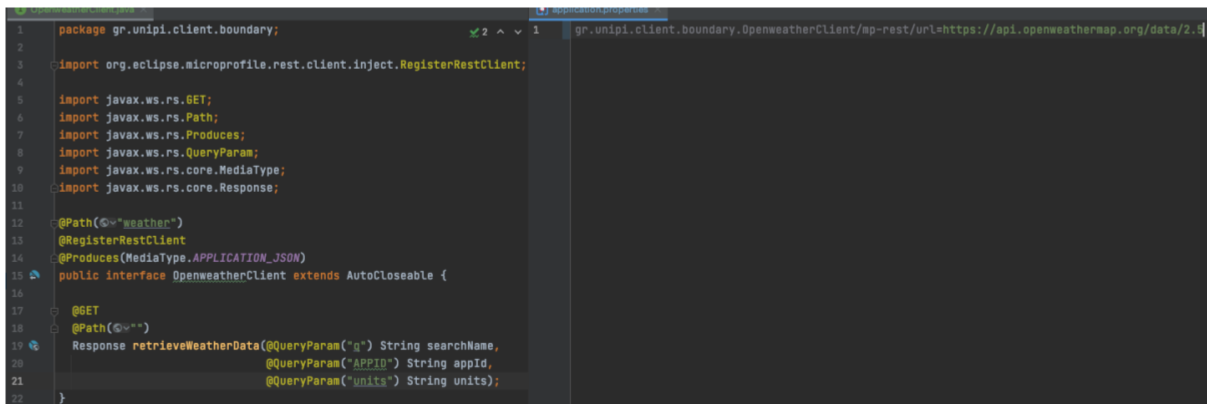
Στο /init επιστρέφουμε ένα κενό response και στο /run τρέχει η εφαρμογή μας. Η δομή τους είναι η παρακάτω:

```java
package gr.unipi.client.boundary;

import javax.ws.rs.POST;
import javax.ws.rs.Path;

@Path("init")
public class InitResource {

  @POST
  public String init() { return ""; }
}
```

```java
package gr.unipi.client.boundary;

import gr.unipi.client.control.WeatherService;
import gr.unipi.client.entity.ExportValue;
import gr.unipi.client.entity.InputValue;

import javax.inject.Inject;
import javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("run")
public class WeatherResource {

  @Inject
  WeatherService service;

  @POST
  @Consumes(MediaType.APPLICATION_JSON)
  @Produces(MediaType.APPLICATION_JSON)
  public ExportValue retrieveWeatherData(InputValue inputValue) {
    return service.getSpotWeatherData(inputValue.getValue().getText().split(" ")[1]);
  }
}
```

9.7 quarkus client REST boundary

Το interface ***OpenweatherClient*** δημιουργεί το path προς το api που είδαμε προηγουμένως. Για να λειτουργήσει ο client σωστά θα χρειαστεί το stable κομμάτι του path να το δηλώσουμε στο αρχείο ***application.properties.*** Η δομή τους φαίνεται στην παρακάτω εικόνα:



9.8 quarkus REST client interface

Βλέπουμε στο αρχείο ***application.properties*** έχουμε ακολουθήσει το package name και έχουμε δηλώσει το όνομα του interface του client με suffix ***/mp-rest/url*** . Αυτός είναι ο τρόπος να μπορέσει κατά την εκτέλεση το quarkus action να διαβάσει το path [8].

Βλέπουμε επίσης ότι στο interface έχουμε δηλώσει τις παραμέτρους που χρειάζονται στο api για να λειτουργήσει.

Στο package ***entity*** βρίσκονται όλα τα μοντέλα / αντικείμενα που χειριζόμαστε στην εφαρμογή. Για να μπορέσουμε να κάνουμε σωστά integration με το slack στην συνέχεια θα χρειαστεί να διαμορφώσουμε την απάντηση που παίρνουμε από το OpenWeatherMap api για να την περάσουμε ως εισερχόμενο μήνυμα στο επόμενο action.

Κάτω από το package ***control*** βρίσκεται ο μοναδικός controller που θα χρειαστούμε για την εφαρμογή μας. Αποτελεί τον συνδετικό κρίκο μεταξύ του rest endpoint της εφαρμογής μας και του OpenWeatherMap api. Λαμβάνει τις παραμέτρους από την κλήση του action, τις μετατρέπει στην μορφή που χρειάζεται το OpenWeatherMap και καλεί τον client. Όταν ο client στην συνέχεια επιστρέψει τα αποτελέσματα, ο controller μορφοποιεί πάλι τα αποτελέσματα και τα εξάγει ως response στην διαδικασία που τον κάλεσε.

Η δομή του είναι η παρακάτω:

```java
package gr.unipi.client.control;

import gr.unipi.client.boundary.OpenweatherClient;
import gr.unipi.client.entity.ExportValue;
import gr.unipi.client.entity.WeatherResponse;
import org.eclipse.microprofile.config.inject.ConfigProperty;
import org.eclipse.microprofile.rest.client.inject.RestClient;

import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;

@ApplicationScoped
public class WeatherService {

  @Inject
  @RestClient
  OpenweatherClient client;

  @Inject
  @ConfigProperty(name = "API_KEY")
  String API_KEY;

  public ExportValue getSpotWeatherData(String name) {
    ExportValue exportValue = new ExportValue();
    exportValue.setTextValue(client.retrieveWeatherData(name, API_KEY, units: "metric").readEntity(WeatherResponse.class));
    return exportValue;
  }
}
```

9.9 quarkus main weather service - singleton

Για να βεβαιωθούμε ότι λειτουργεί η εφαρμογή πριν την κάνουμε build ακολουθούμε την ίδια διαδικασία που είχαμε ακολουθήσει και με την προηγούμενη sample εφαρμογή. Δοκιμάζουμε την εφαρμογή σε dev mode με την εντολή: ***mvn clean package quarkus:dev*** για να εκκινήσει τοπικά στην πόρτα 8080 και κάνουμε ένα POST request στο url: ***localhost:8080/run*** με body:

```json
{
  "value": {
    "text": "weather Athens",
    "trigger_word": "weather"
  }
}
```

Αυτό είναι ένα request με την μορφή που θα έρχεται από το slack request. Το response είναι επίσης μορφοποιημένο όπως μπορεί να το διαβάσει το slack όταν του το επιστρέψουμε και είναι της μορφής:

```json
{
  "text": "clear sky,  Temperature: 6.95 Real feel: 3.1 Max Temperature: 8.89 Min Temperature: 8.89 Humidity: 64.0 Wind Speed: 2.8km/h"
}
```

Όταν ολοκληρώσουμε τις δοκιμές χρειάζεται να κάνουμε build το image και να το κάνουμε push στο docker hub. Θα δημιουργήσουμε και τα δύο image build (jvm & native) και θα τα ανεβάσουμε με αντίστοιχο tag name.

Η εντολή για να κάνουμε build το image σε jvm είναι: ***docker build -t sealsoft/openwhisk-quarkus:jvm -f src/main/docker/Dockerfile.jvm .***

Για να γίνει build σε native απλώς αλλάζουμε το Dockerfile και η εντολή παραμένει ίδια: ***docker build -t sealsoft/openwhisk-quarkus:native -f src/main/docker/Dockerfile.native .***

Όταν ολοκληρωθούν οι διαδικασίες κάνουμε push στο docker hub ώστε να μπορέσει να βρει τα image εκεί το Openwhisk όταν του τα δηλώσουμε ως actions. Οι εντολές είναι οι παρακάτω:

Jvm: ***docker push sealsoft/openwhisk-quarkus:jvm***
Native: ***docker push sealsoft/openwhisk-quarkus:native***

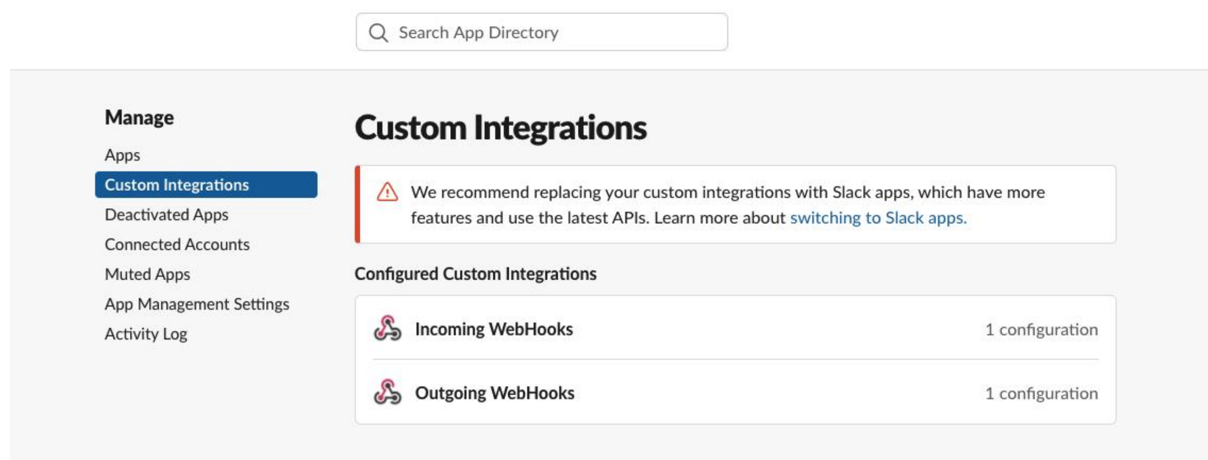
Και τα δύο image όπως και ο κώδικας είναι public. Μπορούμε να τα βρούμε στο docker hub στο url: ***https://hub.docker.com/repository/docker/sealsoft/openwhisk-quarkus***

9.3 Slack Configuration

Για να συνδεθούμε με το slack αρχικά θα πρέπει να έχουμε έναν λογαριασμό και να δημιουργήσουμε ένα κανάλι. Πάμε στον σύνδεσμο: https://slack.com/get-started#/createnew και δημιουργούμε ένα λογαριασμό είτε συνδεόμαστε με λογαριασμό που ήδη έχουμε. Για την διαδικασία αρκεί ένα Gmail. Στην συνέχεια ακολουθούμε τις οδηγίες και δημιουργούμε ένα νέο κανάλι.

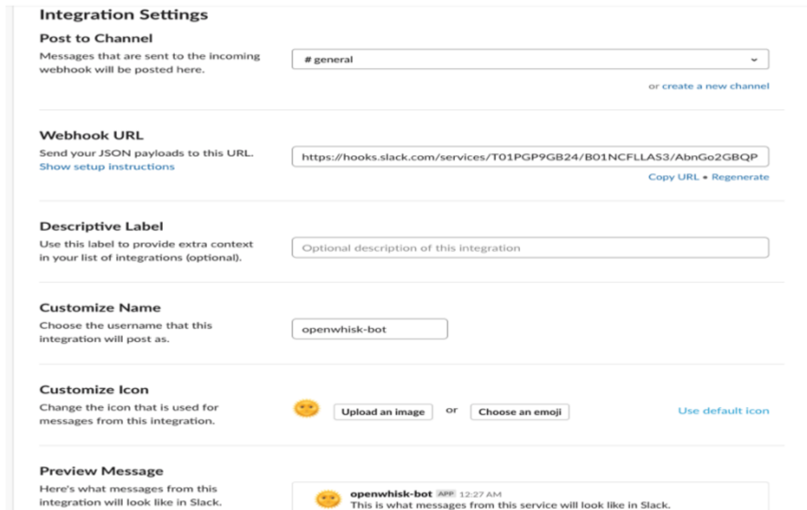
Το κανάλι για την εργασία το ονομάσαμε openwhisk και βρίσκεται στον σύνδεσμο: https://openwhisk-workspace.slack.com/

Αυτό που θα χρειαστεί να κάνουμε, είναι να ενεργοποιήσουμε τα incoming και outgoing webhooks που προσφέρονται στο slack ως εφαρμογές για να συνδέσουμε τα 2 api. Θα τα βρούμε στο path ***/apps/manage/custom-integrations*** ώστε να τα ενεργοποιήσουμε.



9.10 slack custom integrations panel

Αρχικά ρυθμίζουμε τα incoming webhooks στα οποία δηλώνουμε το κανάλι στο οποίο θέλουμε να λαμβάνουμε τα εισερχόμενα μηνύματα και το όνομα που θέλουμε να εμφανίζεται ως αποστολέας. Τέλος μας δίνει ένα επιπλέον URL το οποίο θα το χρειαστούμε ώστε να στείλουμε στην συνέχεια μήνυμα στο general channel του slack.



9.11 incoming webhooks configuration

Στην συνέχεια επιλέγουμε τα outgoing messages. Σε αυτή την φόρμα επιλέγουμε πάλι το κανάλι από το οποίο θα αποστέλλουμε το μήνυμα, την λέξει που θα πυροδοτεί την κλήση, το URL όπου θα καταλήγει η κλήση με το μήνυμα και ένα token για να γίνεται authentication από τον λήπτη του μηνύματος.

Στο πεδίο «trigger word» έχουμε βάλει την λέξη ***weather.*** Αυτό πρακτικά σημαίνει ότι κάθε μήνυμα που θα ξεκινάει με αυτή την λέξη θα κάνει κλήση προς το Openwhisk rest api με το περιεχόμενο του μηνύματος.

Το URL που έχουμε θα δούμε στην συνέχεια πως προκύπτει όταν θα ρυθμίσουμε το Openwhisk να δέχεται και αυτό με την σειρά του κλήσεις στο rest api του.

9.12 outgoing webhooks configuration

9.4 Openwhisk setup και demo

Μετά την ενεργοποίηση του slack Webhook και την δημιουργία του docker image του Quarkus μένει να γίνουν deploy οι εφαρμογές στο Openwhisk και προχωρήσουμε σε όλες τις απαραίτητες ρυθμίσεις.

Γυρνάμε στο τερματικό και θα ξεκινήσουμε από το quarkus action. Για να γίνει deploy το quarkus ως action στο Openwhisk εκτελούμε: ***wsk -i action create openweather-quarkus --docker sealsoft/openwhisk-quarkus:native***

Εδώ δημιουργείται το πρώτο action με ονομασία openweather-quarkus στο οποίο δίνουμε ως runtime το docker image που ανεβάσαμε στο docker hub. Για να ελέγξουμε ότι δημιουργήθηκε εκτελούμε: ***wsk action list | grep openweather***

```
→ OpenWhisk_CLI ./wsk -i action create openweather-quarkus --docker sealsoft/openwhisk-quarkus:jvm
ok: created action openweather-quarkus
→ OpenWhisk_CLI ./wsk -i action list | grep openweather
/openwhisk/openweather-quarkus                                          private blackbox
→ OpenWhisk_CLI
```

9.13 deploy quarkus docker image as action

Στην συνέχεια θα κάνουμε copy το υπάρχον system action του Openwhisk για το slack post. Το συγκεκριμένο action βρίσκεται κάτω από το package ***whisk.system/slack/post*** οπότε εκτελούμε την παρακάτω εντολή: ***wsk action create --copy openweather-slack /whisk.system/slack/post -p url https://hooks.slack.com/services/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***

Βλέπουμε ότι στην παράμετρο url δίνουμε αυτό που μας έκανε generate το slack στα incoming webhooks προηγουμένως. Αν ζητήσουμε το action list θα δούμε τώρα ότι έχουμε 2 action διαθέσιμα.

```
→ OpenWhisk_CLI ./wsk -i action create --copy openweather-slack /whisk.system/slack/post -p url https://hooks.slack.com/services/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
ok: created action openweather-slack
→ OpenWhisk_CLI ./wsk -i action list | grep openweather
/openwhisk/openweather-slack                                   private nodejs:10
/openwhisk/openweather-quarkus                                 private blackbox
→ OpenWhisk_CLI
```

9.14 copy slack system action

Προχωράμε με την δημιουργία ενός sequence από actions. Με τον τρόπο αυτό το Openwhisk μας δίνει την δυνατότητα να συνδέσουμε πολλά actions τα οποία θέλουμε να τρέξουν σειριακά. Στην περίπτωση μας θα τρέξει πρώτα το quarkus να πάρει τα δεδομένα από το OpenWeatherMap api και στην συνέχεια θα τα περάσει αυτά ως input στο openweather-slack action ώστε να στείλει το μήνυμα στο slack κανάλι.

Για να δημιουργήσουμε την αλληλουχία αυτή εκτελούμε: ***wsk action create openweather --sequence /openwhisk/openweather-quarkus,/openwhisk/openweather-slack***

Το action αυτό θέλουμε να το κάνουμε διαθέσιμο μέσω web και να δημιουργήσουμε ένα api gateway ώστε να μπορούμε να το καλέσουμε απομακρυσμένα μέσω του Openwhisk rest api. Για να τα επιτύχουμε αυτά χρειαζόμαστε τις δυο παρακάτω εντολές:

- ***wsk action update openweather --web true***
- ***wsk api create /openweather POST openweather***

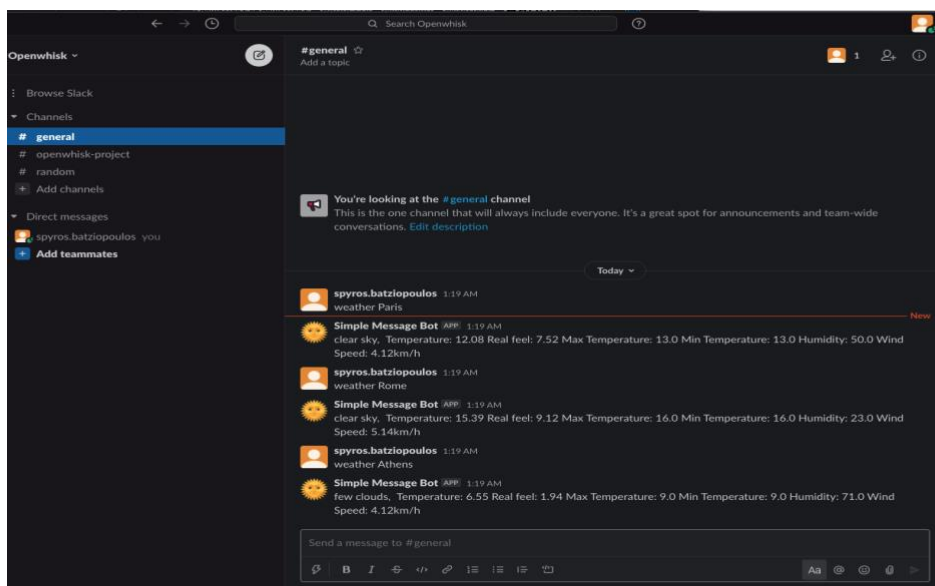
Η δεύτερη εντολή δημιουργεί ένα rest endpoint με path /openweather το οποίο δέχεται POST request και η κλήση του καταλήγει στο openweather action, που είναι η αλληλουχία των δύο action.

Αυτό μας επιτρέπει από το slack το μήνυμα που ξεκινάει με την λέξη «weather» να κάνει κλήση προς το endpoint αυτό. Η απάντηση της δεύτερης εντολής είναι το endpoint που θα χρειαστούμε. Στην περίπτωση μας, όπως είδαμε χρησιμοποιήσαμε το Cloudflare για να δηλώσουμε το domain στην ip μας και κάναμε forward την πόρτα 80 και 443 στην 31001 εσωτερικά οπότε το URL που θα χρειαστούμε είναι k3s.sealsoft.tech/ και μετά το path που μας έδωσε το Openwhisk χωρίς την πόρτα.

```
→ OpenWhisk_CLI ./wsk -i action update openweather --web true
ok: updated action openweather
→ OpenWhisk_CLI ./wsk -i api create /openweather POST openweather
ok: created API /openweather POST for action /_/openweather
https://192.168.65.3:31001/api/2197fd95-207c-4d5d-bdbe-7e329c86bfc1/openweather
```

9.15 create api for openweather sequence

Το τελευταίο κομμάτι της εργασίας είναι η δοκιμή της εφαρμογής μας στο slack:



9.16 slack demo

10. Σύνοψη

Βασικός στόχος αυτής της διπλωματικής εργασίας ήταν να παραθέσει συνοπτικά όλες τις έννοιες που συνθέτουν ένα περιβάλλον ενορχήστρωσης υπηρεσιών και να εκτελέσει σε αυτό, μπλοκ κώδικα που βασίζονται σε serverless αρχιτεκτονική.

Εξετάσαμε σε ικανοποιητικό βαθμό έννοιες όπως το docker και το kubernetes, οι οποίες αποτελούν δομικούς λίθους ενός περιβάλλοντος ενορχήστρωσης. Στην συνέχεια είδαμε δύο open source συστήματα, τα οποία ακολουθούν την serverless αρχιτεκτονική κατά την ανάπτυξη εφαρμογών. Αυτό που καταθέσαμε σαν πρόταση στην εφαρμογή στην συνέχεια ήταν ένας συνδυασμός αυτών των δύο serverless συστημάτων.

Το Openwhisk είναι ένα πολύ δυνατό εργαλείο ανάπτυξης εφαρμογών το οποίο υποστηρίζει πολλά και διαφορετικά runtime και φυσικά το σύνολο σχεδόν της υποδομής τους είναι docker containers. Από την άλλη πλευρά το Quarkus το οποίο υποστηρίζει μόνο το jvm runtime, έχεις πολύ καλές επιδόσεις όταν εκτελείται εντός ενός kubernetes cluster. Ενοποιώντας τα δύο open source συστήματα σκοπός μας ήταν να έχουμε την δυναμική και την ανεξαρτησία που προσφέρει το Openwhisk σε συνδυασμό με Quarkus native builds ως docker container το οποίο θα αποτελεί και το πρώτο block κώδικα που θα εκτελεστεί.

Μια από τις μεγαλύτερες δυσκολίες που αντιμετωπίσαμε ήταν η αδυναμία του Openwhisk να λειτουργήσει σε arm επεξεργαστές όπως αυτόν του raspberry pi. To Openwhisk έχει φροντίσει να παρέχει στους χρήστες του όλα τα images που χρειάζονται μέσω του επίσημου repository του στο docker hub. Το πρόβλημα σε αυτό είναι ότι τα περισσότερα images έχουν γίνει build για amd64 αρχιτεκτονική.

Για να αντιμετωπιστεί αυτό, καθώς το Openwhisk δεν δίνει κάποια λύση, κάναμε build τα περισσότερα από τα images του σε arm64 αρχιτεκτονική με runtime που τρέξαμε εντός του raspberry server. Ο συγκεκριμένος τεχνικός περιορισμός δεν παρουσιάστηκε στην εργασία διότι επέκτεινε κατά πολύ εκτός της θεματικής ενότητας που εξετάζαμε.

Μια ακόμη παρατήρηση σχετικά με το Openwhisk είναι η διαχείριση των πόρων που κάνει εντός ενός kubernetes cluster. Χρειάζεται να παρέχεται σε περιπτώσεις custom cluster μια μονάδα δίσκου την οποία χρειάζεται απαραίτητα να χρησιμοποιήσει ως persistent volume και να γίνει mount από όλα τα nodes.

Το ίδιο ακριβώς configuration μεταφέρθηκε σε ένα custom cluster στο cloud μετά το πέρας των δοκιμών τοπικά ώστε να δούμε πως λειτουργεί χωρίς να χρειαστεί να κάνουμε μεγάλες αλλαγές. Μετά από αυτή την διαδικασία το συμπέρασμα που καταλήγουμε είναι ότι για την ασφαλή και απρόσκοπτη λειτουργία του Openwhisk σε ένα kubernetes cluster χρειάζεται 2 server nodes και αναλόγως του φόρτου εργασίας του 4-6 worker nodes.

Τέλος θα είχε μεγάλο ενδιαφέρον, ως μελλοντική έρευνα και ανάπτυξη, η διαχείριση σε ένα αντίστοιχο περιβάλλον περισσότερο πολύπλοκων εφαρμογών με μεγαλύτερους φόρτους εργασίας. Επίσης σε ένα kubernetes cluster θα είχε ιδιαίτερη σημασία να μελετηθεί εκτενώς το horizontal scaling στους container του Openwhisk κατά την διαχείριση ταυτόχρονα πολλών αλλά και περισσότερο απαιτητικών request.

Και τα δύο προϊόντα – Openwhisk και Quarkus – παρόλο που είναι σε stable release βρίσκονται ακόμη σε φάση ανάπτυξης και επέκτασης. Είναι σημαντικό να αναφέρουμε ότι το native build του

Quarkus στο Openwhisk είχε χαμηλότερο χρόνους warm-up και start-up ακόμη και από JavaScript εφαρμογές οι οποίες χρησιμοποιούν NodeJS runtime.

11. Βιβλιογραφία

[1] Andrei Palade, Aqeel H. Kazmi, Siobhan Clarke, “An evaluation of Open Source Serverless Computing Frameworks Support at the Edge” [online] available: https://www.researchgate.net/publication/332980072_An_Evaluation_of_Open_Source_Serverless_Computing_Frameworks_Support_at_the_Edge

[2] Joel Scheuner, Philipp Leither, “Function-as-a-Service performance evaluation” [online] available: https://www.sciencedirect.com/science/article/pii/S0164121220301527

[3] “Using a service to expose your app”, Kubernetes [online] available: https://kubernetes.io/docs/tutorials/kubernetes-basics/expose/expose-intro/

[4] “Externalizing config using MicroProfile, ConfigMaps and secrets, Kubernetes [online] avaialbe: https://kubernetes.io/docs/tutorials/configuration/configure-java-microservice/configure-java-microservice/

[5] Michele Sciabarra, “Learning Apache OpenWhisk – Developing Open Serverless Solutions [book]

[6] “Container runtime”, Docker [online] available: https://www.docker.com/products/container-runtime

[7] “Kubernetes advantages over docker desktop”, Docker io [online] available: https://www.docker.com/products/kubernetes

[8] “Using Rest client”, Red Hat Quarkus, [online] available: https://quarkus.io/guides/rest-client

[9] “Building native executables”, Red Hat Quarkus [online] available: https://quarkus.io/guides/rest-client

[10] Christopher Barnatt, “A brief guide to cloud computing” [book]

[11] “Prometheus monitor system” Prometheus [online] available: https://prometheus.io

[12] “System metrics with Grafana” [online] available: https://github.com/grafana

[13] “Benefits of cloud computing” Salesforce [online] available: https://www.salesforce.com/products/platform/best-practices/benefits-of-cloud-computing/

[14] “kubernetes lite version for arm” k3s [online] available https://k3s.io

[15] “Traefic ingress controller and load balancing” Rancher labs [online] https://rancher.com/docs/k3s/latest/en/networking/#traefik-ingress-controller

[16] “OpenWhisk programming model” Apache Openwhisk [online] https://openwhisk.apache.org/documentation.html

[17] Marini, Alberto. "Deployment of container orchestration and function-as-a-service functions with node selection method on hybrid cluster architecture." (2020).

[18] Kristiani, Endah, et al. "Implementation of an edge computing architecture using openstack and kubernetes." *International Conference on Information Science and Applications*. Springer, Singapore, 2018.