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

WebAssembly Workshop

Διαδικτυακή μεταγλώττιση και ανάπτυξη κώδικα

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΓΕΩΡΓΙΟΣ ΖΑΡΑΒΙΝΟΣ
ΣΑΒΒΑΣ ΛΕΟΥΣΗΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιανουάριος 2021

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

WebAssembly Workshop

Διαδικτυακή μεταγλώττιση και ανάπτυξη κώδικα

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΓΕΩΡΓΙΟΣ ΖΑΡΑΒΙΝΟΣ
ΣΑΒΒΑΣ ΛΕΟΥΣΗΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 15η Ιανουαρίου 2021.

Νικόλαος Σ. Παπασπύρου Καθηγητής Ε.Μ.Π.	Κωνσταντίνος Σαγώνας Αν. Καθηγητής Ε.Μ.Π.	Γεώργιος Γκούμας Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιανουάριος 2021

.....

ΓΕΩΡΓΙΟΣ ΖΑΡΑΒΙΝΟΣ

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

.....

ΣΑΒΒΑΣ ΛΕΟΥΣΗΣ

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

Περίληψη

Σκοπός της παρούσας εργασίας είναι η συγκέντρωση εργαλείων, πιο συγκεκριμένα μεταγλωττιστών από γνωστές γλώσσες προγραμματισμού όπως οι C, C++ και η Go σε Web Assembly. Αυτό γίνεται με σκοπό προγράμματα και εφραμογές που είναι γραμμένα σε αυτές τις γλώσσες, να μπορούν να τρέχουν και σε προγράμματα περιήγησης ιστού.

Για χρόνια η Javascript ηταν η μοναδική γλώσσα που έτρεχε στα προγράμματα περιήγησης ιστού. Πρόσφατα, μια νέα χαμηλού επιπέδου γλώσσα έκανε την εμφάνισή της και υπόσχεται πολλά πλεονεκτήματα με το βασικότερο να είναι η δυνατότητα για εκτέλεση εφαρμογών πιο υψηλής απόδοσης σε ιστοσελίδες. Επιτρέπει στους προγραμματιστές front-end να αξιοποιήσουν μια υπάρχουσα γλώσσα που είναι πιθανότατα πιο οικεία σε αυτούς όπως η C++ ή απλά να χρησιμοποιήσει υπάρχοντα κομμάτια κώδικα, όπως αλγόριθμους αναζήτησης, τα οποία πλέον μπορούν να τρέχουν σε μια ιστοσελίδα σχεδόν τόσο γρήγορα όσο μια εγγενής εφαρμογή.

Η αρχιτεκτονική που χρησιμοποιήσαμε για την υλοποίηση του εργαλείου μας είναι αυτή των Microservices. Αυτή η αρχιτεκτονική μάς βοήθησε να διαχωρίσουμε την συνολική εφαρμογή σε μικρότερα επιμέρους τμήματα τα οποία λειτουργούν ανεξάρτητα το ένα από το άλλο αλλά συνδιάζονται για να επιτευχθεί η πλήρης λειτουργεία του εργαλείου. Πιο συγκεκριμένα, το Docker είναι αυτό το εργαλείο που μας επέτρεψε τον διαχωρισμό των εφαρμογών και κατέστησε δυνατή τη δυνατότητα συμπύκνωσης και εκτέλεσης όλων των επιμέρους εφαρμογών μικρά απομονωμένα περιβάλλοντα που ονομάζονται κοντέινερ σε έναν κεντρικό υπολογιστή.

Για την τελική υλοποίηση του στόχου μας, οι βασικές τεχνολογίες που χρησιμοποιήθηκαν είναι αρκετές. Τα εργαλεία ανάπτυξης μας είναι το Docker και το Firebase. Στο backend της εφαρμογής μας έχει χρησιμοποιηθεί το NGINX, το Flask της Python με το uWSGI, η MongoDB, ο Kafka μαζί με το Zookeeper καθώς και οι 2 βασικοί μας μεταγλωττιστές, το Emscripten (για C και C++) και ο μεταγλωττιστής για την Go. Τέλος, στο frontend χρησιμοποίησαμε την Angular. Όλες αυτές οι τεχνολογίες συνεργάζονται μεταξύ τους με σκοπό την εύρυθμη λειτουργία της εφαρμογής.

Το τελικό προϊόν της εργασίας μας είναι μια ιστοσελίδα – ένα εργαλείο για προγραμματιστές. Ουσιαστικά, δίνουμε την δυνατότητα στον οποιοδήποτε να γράψει και να αποθηκεύσει την εφαρμογή του στη γλώσσα της αρέσκειάς του. Έπειτα μπορεί να την δει να τρέχει μέσα στο πρόγραμμα περιήγησης που χρησιμοποιεί χωρίς κανέναν κόπο και χωρίς να απαιτείται να διαθέτει ιδιαίτερες γνώσεις σχετικά με την Web Assembly και τις εφαρμογές διαδικτύου γενικότερα. Πρόκεται λοιπόν για ένα εργαλείο που έχει στόχο την εξοικείωση των προγραμματιστών με την με τον χώρο των δικτυακών εφαρμογών με το ελάχιστο δυνατό κόστος και τη μέγιστη ευκολία.

Λέξεις κλειδιά

Προγράμματα περιήγησης ιστού, Μεταγλωττιστές, Web Assembly, Ιστοσελίδες, Διαδίκτυο.

Abstract

The purpose of this diploma dissertation is to integrate tools, more specifically compilers from well-known programming languages such as C, C++ and Go to Web Assembly. The reason we did this is for enabling programs and applications coded in these languages also run in web browsers.

For many years, browsers used to run explicitly JavaScript. Recently, a new low-level language has emerged and promises many benefits, most notably the ability to run higher-performance web applications. It allows front-end developers to take advantage of an existing language that is probably more familiar to them, such as C++, or simply use existing code snippets, such as search algorithms, that can now run on a web page almost as fast as a native application.

The architecture we used to implement our tool is that of Microservices. This architecture helped us to split the entire application into smaller sub-sections that operate independently of each other but are combined to achieve the full functionality of the tool. More specifically, Docker is the tool that allowed us to separate applications and made it possible to compact and run all individual applications in small isolated environments called containers on a single server.

For the final realization of our goal, the basic technologies used are several. Our development tools are Docker and Firebase. In the backend of our application we have used NGINX, Python Flask with uWSGI, MongoDB, Kafka together with Zookeeper and of course our 2 main compilers, Emscripten (for C and C++) and the Golang compiler. Finally, at frontend we used Angular. All these technologies work together for the smooth operation of the application.

The final product of our work is a website – a tool for developers. Essentially, we enable anyone to write and save their application in their favorite language. Then the developer can see it running in the browser effortlessly and without being required to have special knowledge about Web Assembly and web applications in general. It is therefore a tool that aims to familiarize developers with web applications at the lowest possible cost and maximum convenience.

Key words

Browsers, Compilers, Web Assembly, Web pages Web Developer.

Ευχαριστίες

Πρώτα από όλα θα θέλαμε να ευχαριστήσουμε τον επιβλέποντά μας, κ. Νίκο Παπασπύρου, για την καθοδήγηση του, για το ότι ήταν πάντα στο πλάι μας όποτε τον χρειαστήκαμε και μας βοήθησε να πάρουμε αποφάσεις για τα επόμενα βήματα της επαγγελματικής μας καριέρας. Θα θέλαμε επίσης, να ευχαριστήσουμε τους κ. Γιώργο Γκούμα και κ. Κωνσταντίνο Σαγώνα, οι οποίοι συμπληρώνουν την τριμελή εξεταστική επιτροπή, καθώς και όλους τους καθηγητές στο τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Η/Υ, καθώς με τη βοήθειά τους φτάσαμε στο σημείο που βρισκόμαστε σήμερα, να ασχολούμαστε με αυτό που αγαπάμε.

Τέλος, θα θέλαμε φυσικά να ευχαριστήσουμε τις οικογένειές μας για την υποστήριξη τους καθ' όλη τη διάρκεια των σπουδών μας αλλά και τους φίλους - συμφοιτητές μας για όλα όσα περάσαμε μαζί τους όλα αυτά τα χρόνια, που δεν θα ήταν ποτέ ίδια χωρίς αυτούς.

Γεώργιος Ζαραβίνος, Σάββας Λεούσης,

Αθήνα, 15η Ιανουαρίου 2021

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-6-20, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Ιανουάριος 2021.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Περιεχόμενα

Κατάλογος σχημάτων

Σχήματα στο αγγλικό κείμενο

Κεφάλαιο 1

Εισαγωγή

Για πολλά χρόνια, τα προγράμματα περιήγησης χρησιμοποιούσαν αποκλειστηκά την JavaScript σαν γλώσσα για όλες τις ενέργειες που εκτελούσαν. Κάθε προγραμματιστής έπρεπε να μάθει να προγραμματίζει σε αυτήν τη μοναδική γλώσσα για να έχει τη δυνατότητα να τρέχει εφαρμογές σε οποιοδήποτε πρόγραμμα περιήγησης. Όμως, το 2015, εμφανίστηκε μια εντελώς νέα γλώσσα με την ονομασία Web Assembly. Ο κύριος στόχος αυτής της γλώσσας ήταν να επιτρέψει την εκτέλεση εφαρμογών πιο υψηλής απόδοσης σε ιστοσελίδες. Ωστόσο, το πιο συναρπαστικό μέρος είναι ότι, τώρα, μπορούμε να έχουμε εργαλεία (μεταγλωττιστές) που συγκεντρώνουν τις περισσότερες από τις αγαπημένες μας γλώσσες, όπως C, C ++, Python, Rust κλπ. στη Web Assembly. Έτσι, έχουμε την δυνατότητα να γράφουμε κώδικα όπως συνηθίζαμε και τον βλέπουμε να τρέχει μέσα στο πρόγραμμα περιήγησής μας σχεδόν χωρίς να κάνουμε τίποτα παραπάνω.

Καθώς κάναμε κάποια έρευνα, καταλάβαμε ότι θα ήταν πολύ χρήσιμο αν μπορούμε να ενσωματώσουμε όλα αυτά τα εργαλεία μεταγλώτισσης σε Web Assembly σε ένα. Με αυτόν τον τρόπο, κάποιος που θέλει να εξοικειωθεί με τη Web Assembly και τις εφαρμογές που εκτελούνται στα πρόγραμμα περιήγησης, μπορεί να έχει την ευκαιρία να γράψει, να αποθηκεύσει και να εκτελέσει την εφαρμογή του στην αγαπημένη του γλώσσα σε ένα μέρος, χωρίς να χρειάζεται να κάνει αναζήτηση για να βρει κάθε εργαλείο που απαιτείται, ανεξάρτητα από το περιβάλλον στο οποίο δουλεύει. Το **Web Assembly Workshop** είναι ακριβώς αυτό και είμαστε πολύ περήφανοι που επιτύχαμε τον στόχο μας, φέρνοντας κάτι χρήσιμο με πολλές δυνατότητες ανάπτυξης στα χέρια όλων μας!

1.1 Web Assembly

1.1.1 Τι είναι η Web Assembly

Η Web Assembly (συντομογραφία Wasm) είναι μια γλώσσα προγραμματισμού χαμηλού επιπέδου για μια εικονική μηχανή που βασίζεται σε στοίβα. Αυτή η γλώσσα έχει σχεδιαστεί με σκοπό την μεταγλώτισση εφαρμογών από άλλες γνωστές γλώσσες υψηλού επιπέδου όπως C / C ++ / Rust. Με αυτό τον τρόπο επιτρέπεται η ανάπτυξη εφαρμογών ιστού με χρήση και άλλων γλωσσών πέρα από την Javascript. Πρόκειται για μια ριζικά διαφορετική προσέγγιση για την ανάπτυξη λογισμικού ιστό, σε αντίθεση με την τυπική χρήση βιβλιοθηκών JavaScript που πολλές φορές δεν δίνουν ή δίνουν αρκετά αργές λύσεις σε αρκετά υπάρχοντα προβλήματα. Τέσσερα μεγάλα προγράμματα περιήγησης έχουν ήδη υιοθετήσει την Web Assembly, γεγονός το οποίο αποτελεί ένα τεράστιο βήμα προς την επίτευξη της ανάπτυξης εφαρμογών διαδικτύου υψηλής απόδοσης.

1.1.2 Γιατί είναι σημαντική η Web Assembly

Η WebAssembly είναι μόνο η δεύτερη γλώσσα (μετά τη Javascript) που μπορεί να τρέξει στα προγράμματα περιήγησης ιστού. Μάλιστα, η πρώτη έχει παγιδευτεί σε αμέτρητα ζητήματα

συμμόρφωσης με πρότυπα, σοβαρά προβλήματα απόδοσης καθώς και σε έντονη δυσκινησία στα πλαίσια που προκαλούνται συχνά περισσότερα προβλήματα από αυτά που επιλύονται μακροπρόθεσμα. Οπότε, μετά από μια πορεία 25 ετών, είναι καιρός πλέον να φτάσει στο προσκήνειο μια νέα γλώσσα για την οποία μάλιστα, φαίνεται να υπάρχουν πολλές προσδοκίες.

1.1.3 Πώς λειτουργεί

Η WebAssembly αποτελεί ένα σετ αρχιτεκτονικής εικονικών εντολών, η οποία επιτρέπει σε έναν εξειδικευμένο προγραμματιστή να δημιουργήσει λειτουργικές μονάδες που φορτώνουν γρήγορα και εκτελούνται σχεδόν τόσο γρήγορα όσο τα μεταγλωττισμένα C ή C ++, σαν να συντάχθηκαν αυτές οι εφαρμογές απευθείας στο ίδιο το πρόγραμμα περιήγησης ιστού.

1.1.4 Επόμενα Βήματα

Η Web Assembly δείχνει πολλές δυνατότητες για τη γεφύρωση του χάσματος μεταξύ των κομματιών κώδικα που αφορούν τον client και τον server των εφαρμογών ιστού, το οποίο είναι ιδιαίτερα σημαντικό καθώς μπαίνουμε σε μια εποχή κατανεμημένων υπολογιστών και ανοιχτών προτύπων ιστού. Καθώς διατίθενται περισσότερες τοπικές πηγές δεδομένων και ενέργειας, η αξιοποίηση της απίστευτης δύναμης των σύγχρονων προσωπικών υπολογιστικών συσκευών θα είναι ένα σημαντικό βήμα προς τη σωστή κατεύθυνση προς ένα πιο προσιτό, παραγωγικό και διασκεδαστικό μέλλον. Παρόλο που αυτή η τεχνολογία μπορεί να μην προσφέρει άμεσα κέρδη για όλους όσους μπορούν να τη χρησιμοποιήσουν, για όσους έχουν λόγο να την υιοθετήσουν νωρίς, υπάρχουν τεράστια πλεονεκτήματα που θα αρχίσουν να φαίνονται αμέσως. Αυτό ισχύει ιδιαίτερα στην περίπτωση του AssemblyScript, καθώς επιτρέπει στους προγραμματιστές front-end να αξιοποιήσουν μια υπάρχουσα γλώσσα που είναι πιθανότατα πιο οικεία σε αυτούς όπως η C++ ή η Rust για παράδειγμα. Με το AssemblyScript, ένας προγραμματιστής front-end θα μπορούσε, για παράδειγμα, να μεταφέρει εύκολα πολλές σημαντικές υπάρχουσες συναρτήσεις, όπως αυστηρούς βρόχους για αλγόριθμους αναζήτησης ή παιχνίδια με τεχνητή νοημοσύνη, σε μια εξαιρετικά γρήγορη μεταγλωττισμένη δυαδική μορφή που λειτουργεί σχεδόν τόσο γρήγορα όσο μια εγγενής εφαρμογή (και πιθανώς γρηγορότερα ανάλογα με τιε γνώσεις του προγραμματιστή).

1.2 Ο σκοπός της διπλωματικής

Ο σκοπός της διπλωματικής μας είναι να δημιουργήσουμε ένα εργαλείο που μπορεί να προσφέρει σε έναν προγραμματιστή μια σειρά από μεταγλωττιστές πολλών διάσημων γλωσσών όπως C, C++, Go, Rust κ.λπ. σε Web Assembly. Καταφέραμε να επιτύχουμε αυτόν τον στόχο δημιουργώντας έναν server που ενσωματώνει όλα αυτά τα εργαλεία σε ένα. Με αυτόν τον τρόπο, ένας προγραμματιστής μπορεί να γράψει κώδικα στην αγαπημένη του γλώσσα και στη συνέχεια να παράξει τον αντίστοιχο κώδικα Web Assembly και να τον τρέξει σε ένα πρόγραμμα περιήγησης χωρίς προσπάθεια. Δεν χρειάζεται να μελετήσει και να εξασκηθεί για να μάθει να κωδικοποιεί στη Web Assembly και αυτό είναι και το μεγαλύτερο πλεονέκτημα που θέλαμε να προσφέρουμε μέσα από την εργαλείο μας. Τώρα ο καθένας μπορεί να γράψει κώδικα και να τον εκτελέσει στο πρόγραμμα περιήγησής του. Οι αλγόριθμοι και τα γραφικά μπορούν πλέον να εκτελούνται σε προγράμματα περιήγησης πολύ πιο γρήγορα από ό,τι πριν με την "καθαρή" Javascript και αυτή η δυνατότητα δίνεται πλέον απλόχερα σε όλους τους προγραμματιστές μέσω της ιστοσελίδας μας.

Κεφάλαιο 2

Εργαλεία ανάπτυξης

2.1 Microservices

Τι είναι ένα microservice?

Τα Microservices είναι ένα αρχιτεκτονικό στυλ που βοηθάει στην ανάπτυξη μιας εφαρμογής ως μια συλλογή υπηρεσιών που είναι:

- Ιδιαίτερα διατηρήσιμες και ελεγχόμενες
- Χαλαρά συνδεδεμένες
- Ανεξάρτητα αναπτυσσόμενες
- Οργανωμένες γύρω από επιχειρηματικές δυνατότητες

Η αρχιτεκτονική των microservices επιτρέπει την ταχεία και αξιόπιστη παράδοση μεγάλων και πολύπλοκων εφαρμογών. Επιτρέπει επίσης σε έναν οργανισμό να εξελίσσει την τεχνολογίες που χρησιμοποιεί.

Ποια είναι τα πλεονεκτήματα;

Αυτή η λύση έχει πολλά οφέλη:

- Επιτρέπει τη συνεχή παράδοση και ανάπτυξη μεγάλων, πολύπλοκων εφαρμογών.
  - Βελτιωμένη συντήρηση - κάθε υπηρεσία είναι σχετικά μικρή και έτσι είναι ευκολότερο να κατανοηθεί και να αλλάξει
  - Μεγαλύτερη ευκολία στις δοκιμές - οι υπηρεσίες είναι μικρότερες και γρηγορότερες
  - Μεγαλύτερη ευκολία ανάπτυξης - οι υπηρεσίες μπορούν να αναπτυχθούν ανεξάρτητα
  - Επιτρέπει την οργάνωση στην προσπάθεια ανάπτυξης από πολλές, αυτόνομες ομάδες. Κάθε ομάδα κατέχει και είναι υπεύθυνη για μία ή περισσότερες υπηρεσίες. Κάθε ομάδα μπορεί να αναπτύξει, να δοκιμάσει, να αναπτύξει και να κλιμακώσει τις υπηρεσίες της ανεξάρτητα από όλες τις άλλες ομάδες.
- Κάθε microservice είναι σχετικά μικρό:
  - Ευκολότερο για έναν προγραμματιστή να κατανοήσει την λειτουργία του
  - Το IDE είναι γρήγορο και κάνει τους προγραμματιστές πιο παραγωγικούς
  - Η εφαρμογή τρέχει πιο γρήγορα, γεγονός που καθιστά τους προγραμματιστές πιο παραγωγικούς και επιταχύνει την ανάπτυξη
- Βελτιωμένη απομόνωση σφαλμάτων. Για παράδειγμα, εάν υπάρχει διαρροή μνήμης σε μία υπηρεσία, τότε θα επηρεαστεί μόνο αυτή η υπηρεσία. Οι άλλες υπηρεσίες θα συνεχίσουν να χειρίζονται αιτήματα. Συγκριτικά, ένα αντίστοιχο πρόβλημα σε μία εφαρμογή μονολιθικής αρχιτεκτονικής μπορεί να καταστρέψει ολόκληρο το σύστημα.

- Εξαλείφει οποιαδήποτε μακροπρόθεσμη δέσμευση σε μια στοίβα τεχνολογίας. Κατά την ανάπτυξη μιας νέας υπηρεσίας μπορεί να επιλέχθεί μια νέα στοίβα τεχνολογίας. Ομοίως, όταν γίνονται σημαντικές αλλαγές σε μια υπάρχουσα υπηρεσία, υπάρχει η δυνατότητα να ξαναγραφεί χρησιμοποιώντας μια νέα στοίβα τεχνολογίας.

Ποια είναι τα μειονεκτήματα;

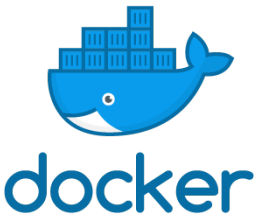
Αυτή η λύση έχει επίσης ορισμένα μειονεκτήματα:

- Οι προγραμματιστές πρέπει να αντιμετωπίσουν την πρόσθετη πολυπλοκότητα της δημιουργίας ενός κατανεμημένου συστήματος:
  - Οι προγραμματιστές πρέπει να υλοποιήσουν τον μηχανισμό επικοινωνίας μεταξύ των υπηρεσιών και να αντιμετωπίσουν το φαινόμενο της μερικής αποτυχίας
  - Η εφαρμογή αιτημάτων που καλύπτουν πολλές υπηρεσίες είναι πιο δύσκολη
  - Ο έλεγχος των αλληλεπιδράσεων μεταξύ υπηρεσιών είναι πιο δύσκολος
  - Η εφαρμογή αιτημάτων που καλύπτουν πολλές υπηρεσίες απαιτεί προσεκτικό συντονισμό μεταξύ των ομάδων
  - Τα εργαλεία προγραμματιστή / IDE διευκολύνουν περισσότερο τη δημιουργία μονολιθικών εφαρμογών και δεν παρέχουν καλή υποστήριξη για την ανάπτυξη κατανεμημένων εφαρμογών.
- Πολυπλοκότητα ανάπτυξης. Στην παραγωγή, υπάρχει επίσης η λειτουργική πολυπλοκότητα της ανάπτυξης και διαχείρισης ενός συστήματος που αποτελείται από πολλές διαφορετικές υπηρεσίες.
- Αυξημένη κατανάλωση μνήμης. Η αρχιτεκτονική των microservices αντικαθιστά N μονολιθικές εφαρμογές με πλήθος υπηρεσιών NxM. Εάν κάθε υπηρεσία εκτελείται με το δικό της JVM (ή ισοδύναμο), τότε υπάρχει η επιβάρυνση M χρόνων αναλογικά με τους χρόνους εκτέλεσης JVM. Επιπλέον, εάν κάθε υπηρεσία εκτελείται με το δική της VM (π.χ. παρουσία EC2), όπως για παράδειγμα συμβαίνει στο Netflix, το γενικό κόστος είναι ακόμη υψηλότερο.

2.2 Docker

Τι είναι το Docker;

Το Docker είναι μια ανοιχτή πλατφόρμα για ανάπτυξη, αποστολή και εκτέλεση εφαρμογών. Το Docker επιτρέπει τον διαχωρισμό των εφαρμογών από την υποδομή, ώστε να μπορεί να παραδίδεται το λογισμικό πιο γρήγορα. Με το Docker, είναι δυνατό να γίνει η διαχείριση της υποδομής με τον ίδιο τρόπο που γίνεται η διαχείριση των εφαρμογών. Αξιοποιώντας τις μεθοδολογίες του Docker για αποστολή, δοκιμή και ανάπτυξη κώδικα, μπορεί να μειωθεί σημαντικά η καθυστέρηση μεταξύ της σύνταξης κώδικα και της εκτέλεσης του στην παραγωγή.



Γιατί είναι το Docker χρήσιμο?

Το Docker παρέχει τη δυνατότητα συμπύκνωσης και εκτέλεσης μιας εφαρμογής σε ένα απομονωμένο περιβάλλον που ονομάζεται κοντέινερ. Η απομόνωση και η ασφάλεια επιτρέπουν την εκτέλεση πολλών κοντέινερ ταυτόχρονα σε έναν κεντρικό υπολογιστή. Τα κοντέινερ είναι ελαφριά, επειδή δεν χρειάζονται το επιπλέον φορτίο ενός hypervisor, αλλά τρέχουν απευθείας στον

πυρήνα του κεντρικού υπολογιστή. Αυτό σημαίνει ότι υπάρχει η δυνατότητα για εκτέλεση περισσότερων κοντέινερ σε έναν δεδομένο hardware από ό,τι εάν γινόταν χρήση εικονικών μηχανών. Τα κοντέινερ Docker μπορούν να τρέξουν ακόμα και σε κεντρικούς υπολογιστές που είναι πραγματικά εικονικές μηχανές!

Το Docker παρέχει εργαλεία και μια πλατφόρμα για τη διαχείριση του κύκλου ζωής των κοντέινερ σας:

- Αναπτύξτε την εφαρμογή σας και όλα τα στοιχεία της χρησιμοποιώντας κοντέινερ
- Το κοντέινερ γίνεται η μονάδα διανομής και δοκιμής της εφαρμογής σας
- Όταν είστε έτοιμοι, αναπτύξτε την εφαρμογή σας στο περιβάλλον παραγωγής σας, ως κοντέινερ. Αυτό λειτουργεί το ίδιο εάν το περιβάλλον παραγωγής σας είναι ένα τοπικό κέντρο δεδομένων, ένας πάροχος cloud ή ένας συνδιασμός των δύο.

Πώς χρησιμοποιείται το Docker στην εργασία μας

Τα microservices που χρησιμοποιούνται στο εργαλείο μας είναι κοντέινερ και μπορούν να χρησιμοποιηθούν χρησιμοποιώντας το εργαλείο Docker Compose, το οποίο μας επιτρέπει να ενορχηστρώσουμε και να αναπτύξουμε ταυτόχρονα όλες τις απαραίτητες υπηρεσίες για να κάνουμε την εφαρμογή ιστού μας να λειτουργεί ομαλά.

Το Compose είναι ένα εργαλείο για τον καθορισμό και την εκτέλεση εφαρμογών Docker πολλαπλών κοντέινερ. Με το Compose, χρησιμοποιείτε ένα αρχείο YAML με το οποίο εκτελείται η διαμόρφωση των υπηρεσιών της εφαρμογής. Στη συνέχεια, με μία μόνο εντολή, δημιουργούνται και ξεκινούν όλες οι υπηρεσίες με βάση τη διαμόρφωση.

Το `docker-compose.yml` που χρησιμοποιείται για την ανάπτυξη του backend βρίσκεται παρακάτω:

```
version: '2'
services:
  flask:
    image: 'tiangolo/uwsgi-nginx-flask:python3.8-2020-08-16'
    environment:
      - USER
      - LISTEN_PORT=80
    restart: always
    volumes:
      - './nginx_flask_uwsgi:/app'
      - './requirements.txt:/requirements.txt'
      - '/tmp:/tmp'
      - '/etc/ssl:/etc/ssl'
    ports:
      - '443:443'
    command: bash -c "pip install -r /requirements.txt && ../start.sh"
    tty: true
    stdin_open: true
  mongo:
    image: 'mongo:4.4.2'
    environment:
      - MONGO_INITDB_DATABASE=wasm
    restart: always
    volumes:
      - './dump:/dump'
```

```
    ports:
      - '27017:27017'
    command: bash -c "docker-entrypoint.sh mongod"
  mongo-seed:
    image: 'stefanwalther/mongo-seed:master'
    environment:
      - MONGODB_HOST=mongo
      - MONGODB_PORT=27017
    volumes:
      - './dump:/data'
    depends_on:
      - mongo
    command: mongorestore --host mongo --port 27017 /data --drop
  mongo-express:
    image: 'mongo-express:0.54'
    environment:
      - ME_CONFIG_SITE_GRIDFS_ENABLED=true
    restart: always
    ports:
      - '8081:8081'
    depends_on:
      - mongo
  zookeeper:
    image: 'wurstmeister/zookeeper:latest'
    ports:
      - '2181:2181'
    restart: always
    tty: true
    stdin_open: true
  kafka:
    image: 'wurstmeister/kafka:2.13-2.6.0'
    ports:
      - '9092:9092'
    environment:
      KAFKA_ADVERTISED_LISTENERS: 'INSIDE://:9092,OUTSIDE://:9094'
      KAFKA_LISTENERS: 'INSIDE://:9092,OUTSIDE://:9094'
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: 'INSIDE:PLAINTEXT,OUTSIDE:PLAINTEXT'
      KAFKA_INTER_BROKER_LISTENER_NAME: INSIDE
      KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
      KAFKA_BROKER_ID: '1'
      KAFKA_CREATE_TOPICS: 'C_TOPIC:1:1,GO_TOPIC:1:1'
      KAFKA_MAX_REQUEST_SIZE: 104857600
      KAFKA_MESSAGE_MAX_BYTES: 104857600
    restart: always
    volumes:
      - '/var/run/docker.sock:/var/run/docker.sock'
    depends_on:
      - zookeeper
    tty: true
    stdin_open: true
  emscripten_compiler:
```

```
    image: 'robertaboukhalil/emsdk:1.39.1'
    volumes:
      - '/tmp:/tmp'
    restart: always
    command: bash
    stdin_open: true
    tty: true
  emscripten_consumer:
    image: 'python:3.8'
    environment:
      - USER
    volumes:
      - '/tmp:/tmp'
      - './:/app'
      - '/var/run/docker.sock:/var/run/docker.sock'
    command: >-
      bash -c "pip install -r /app/requirements.txt && python
      /app/consumer/consumer.py -l C -t 5"
    depends_on:
      - kafka
      - emscripten_compiler
    tty: true
    stdin_open: true
    restart: always
  emscriptenpp_compiler:
    image: 'robertaboukhalil/emsdk:1.39.1'
    volumes:
      - '/tmp:/tmp'
    command: bash
    stdin_open: true
    tty: true
    restart: always
  emscriptenpp_consumer:
    image: 'python:3.8'
    environment:
      - USER
    volumes:
      - '/tmp:/tmp'
      - './:/app'
      - '/var/run/docker.sock:/var/run/docker.sock'
    command: >-
      bash -c "pip install -r /app/requirements.txt && python
      /app/consumer/consumer.py -l C++ -t 5"
    depends_on:
      - kafka
      - emscripten_compiler
    tty: true
    stdin_open: true
    restart: always
  golang_compiler:
    image: 'golang:1.15.5'
```

```
    volumes:
      - '/tmp:/tmp'
      - './consumer/assets:/assets'
    command: bash
    stdin_open: true
    tty: true
    restart: always
  golang_consumer:
    image: 'python:3.8'
    environment:
      - USER
    volumes:
      - '/tmp:/tmp'
      - './:/app'
      - '/var/run/docker.sock:/var/run/docker.sock'
    command: >-
      bash -c "pip install -r /app/requirements.txt && python
      /app/consumer/consumer.py -l Go -t 5"
    depends_on:
      - kafka
      - golang_compiler
    tty: true
    stdin_open: true
    restart: always
```

Σχήμα 2.1: Docker Compose YAML file used for our project.

Εκτελώντας την εντολή `docker compose up`, όλες οι υπηρεσίες μας σε κοντέινερ ξεκινούν μία προς μία, καθεμία ως ξεχωριστή διαδικασία και σχηματίζεται έτσι ένα δίκτυο Docker που μπορεί να χρησιμοποιηθεί για εσωτερική επικοινωνία μεταξύ των υπηρεσιών.

2.3 Firebase

Το Firebase είναι η πλατφόρμα ανάπτυξης εφαρμογών της Google που βοηθά στη δημιουργία, τη βελτίωση και την ανάπτυξη μιας εφαρμογής ιστού. Χρησιμοποιούμε αυτήν την πλατφόρμα για να φιλοξενήσουμε την εφαρμογή web Angular frontend μέσω του Firebase Hosting.

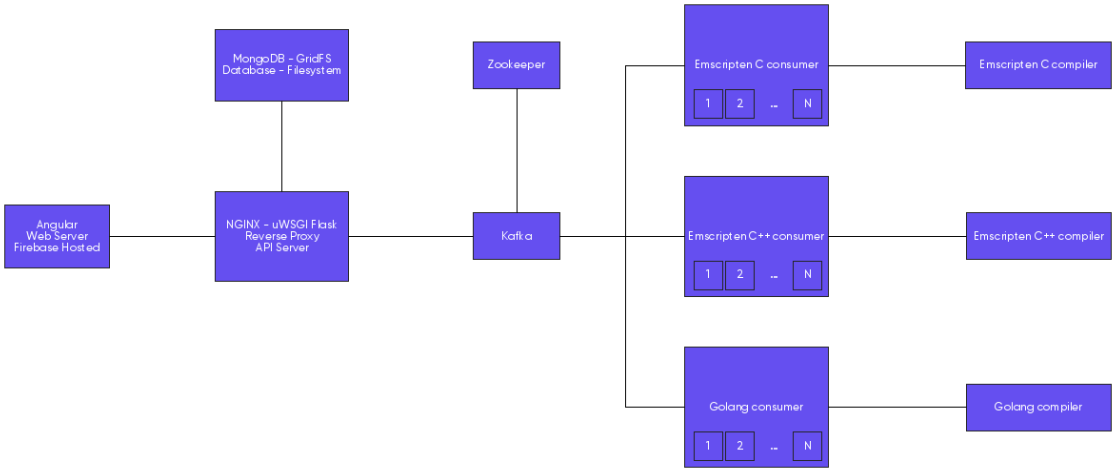
Με μία μόνο εντολή, μπορείτε να αναπτύξετε γρήγορα εφαρμογές ιστού και να προβάλλετε τόσο στατικό όσο και δυναμικό περιεχόμενο σε ένα παγκόσμιο δίκτυο παράδοσης περιεχομένου.

Το Firebase Hosting έχει δημιουργηθεί για τον σύγχρονο προγραμματιστό ιστού. Οι ιστότοποι και οι εφαρμογές είναι πιο ισχυρές από ποτέ με την εξέλιξη των JavaScript front-end frameworks όπως η Angular και κάποια στατικά εργαλεία δημιουργίας όπως το Jekyll. Είτε αναφερόμαστε σε ανάπτυξη μιας απλής ιστοσελίδας εφαρμογής, είτε μίας σύνθετης εφαρμογής web, το Firebase Hosting παρέχει την υποδομή, τις δυνατότητες και τα εργαλεία που είναι απαραίτητα στην ανάπτυξη και διαχείριση ιστότοπων και εφαρμογών.

Κεφάλαιο 3

Υποδομή εργασίας

Σε αυτό το κεφάλαιο θα απαριθμήσουμε και θα εξοικειωθούμε με όλα τα στοιχεία και τις τεχνολογίες που χρησιμοποιούνται στην εργασία μας από το backend έως το frontend. Η δομή του εργασίας έχει σχεδιαστεί έχοντας κατά νου πολλαπλά συνδεδεμένα microservices, προκειμένου να εξυπηρετήσει τον στόχο μας. Μια γρήγορη παρουσίαση των λειτουργιών αυτών των συνδεδεμένων υπηρεσιών γίνεται παρακάτω:



Σχήμα 3.1: Υποδομή των micro-services του WebAssembly Workshop.

Κείμενο στα αγγλικά

Chapter 1

Introduction

For many years, browsers used to run explicitly JavaScript. Every developer had to learn to code in this one and only language in order to have an application running on a browser. But, in 2015, a completely new language appeared named Web Assembly. The main goal of this language was to enable high-performance applications on web pages. Nevertheless, the most exciting part is that, now, we can have tools (compilers) that compile most of our favorite languages like C, C++, Python, Rust, etc. to Web Assembly. Thus, we have the opportunity to code as we used to code and see our code running right inside our browser almost without doing anything.

As we did some research, we figured out that it would be very useful if we could integrate all these compiling tools into one. In this way, someone who wants to get familiar with Web Assembly and browser-running applications, can have the opportunity to code, save and run his application in his favorite language in one place, without having to search to find every tool and dependency needed, regardless of the working environment. **Web Assembly Workshop** is exactly that and we are very proud we reached our goal, bring something useful with many possibilities for development in your hands!

1.1 Web Assembly

1.1.1 What is Web Assembly

WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable target for compilation of high-level languages like C/C++/Rust, enabling deployment on the web for client and server applications. This is a radically different approach to front-end software development on the web, in contrast to the typical use of heavy JavaScript libraries with layers of compatibility workarounds for issues which may not even exist in five or ten years. Four major browsers plus node have adopted it, which is a huge step towards finally achieving cross-browser compatibility, with high performance web applications being the default rather than the exception.

1.1.2 Why is Web Assembly Important

WebAssembly is only the second language (after Javascript) to be natively understood by web browsers, with the first having been caught up in endless waves of standards compliance issues, serious performance problems, conflicting notions of how to go about implementing solutions, and giant cumbersome frameworks that often cause more problems than they solve in the long run. So, after a good 25 year run, it's about time that at least one other language gets a shot at it.

1.1.3 How it works

WebAssembly is a virtual instruction set architecture (virtual ISA), which effectively allows a skilled developer to build modules that load quickly and run nearly as fast as compiled C or C++, as if these functions were compiled directly into the web browser itself.

1.1.4 Next steps

WebAssembly shows a lot of potential for bridging the gap between client and server components of web applications, which is especially important as we enter an age of distributed computing and open web standards. As more local sources of data and energy become available, leveraging the incredible power of modern personal computing devices will be an important step in the right direction towards a more accessible, productive, and entertaining future. While this technology may not provide immediate returns for everyone who might consider using it, for those with a reason to adopt early there are huge advantages which will start to pay off right away. This is especially in the case of AssemblyScript, as it allows front-end developers to leverage an existing language that is likely more familiar to them like C++ or Rust for example. With AssemblyScript, a front-end developer could, for example, migrate all performance-critical functions, such as tight loops for search algorithms or game AI, into an ultra-fast compiled binary format that runs almost as fast as a native application (and potentially faster depending on the programmer).

1.2 The Goal of this Project

The purpose of our thesis is to create a tool (integrator) that can offer a programmer a bunch of compilers of many famous languages such as C, C++, Go, Rust etc. to WebAssembly. We managed to achieve this goal by creating a server that integrates all these tools in one. By that way, a programmer can write code to his preferred language and then he can generate the corresponding WebAssembly script and run it on a browser without effort. He does not have to study and practice to learn to code in WebAssembly and that is the most important thing. Now everyone can code and run this code in his browser. Algorithms and graphics can now run in browsers a lot faster than before with Javascript and anybody can do it easily with our tool.

Chapter 2

Deployment tools

2.1 Microservices

What is a microservice?

Microservices, also known as the microservice architecture, is an architectural style that structures an application as a collection of services that are:

- Highly maintainable and testable
- Loosely coupled
- Independently deployable
- Organized around business capabilities

The microservice architecture enables the rapid, frequent and reliable delivery of large, complex applications. It also enables an organization to evolve its technology stack.

What are the benefits?

This solution has a number of benefits:

- Enables the continuous delivery and deployment of large, complex applications.
  - Improved maintainability - each service is relatively small and so is easier to understand and change
  - Better testability - services are smaller and faster to test
  - Better deployability - services can be deployed independently
  - It enables you to organize the development effort around multiple, autonomous teams. Each (so called two pizza) team owns and is responsible for one or more services. Each team can develop, test, deploy and scale their services independently of all of the other teams.
- Each microservice is relatively small:
  - Easier for a developer to understand
  - The IDE is faster making developers more productive
  - The application starts faster, which makes developers more productive, and speeds up deployments
- Improved fault isolation. For example, if there is a memory leak in one service then only that service will be affected. The other services will continue to handle requests. In comparison, one misbehaving component of a monolithic architecture can bring down the entire system.

- Eliminates any long-term commitment to a technology stack. When developing a new service you can pick a new technology stack. Similarly, when making major changes to an existing service you can rewrite it using a new technology stack.

What are the drawbacks?

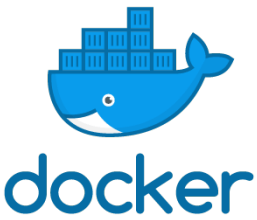
This solution has a number of drawbacks:

- Developers must deal with the additional complexity of creating a distributed system:
  - Developers must implement the inter-service communication mechanism and deal with partial failure
  - Implementing requests that span multiple services is more difficult
  - Testing the interactions between services is more difficult
  - Implementing requests that span multiple services requires careful coordination between the teams
  - Developer tools/IDEs are oriented on building monolithic applications and don't provide explicit support for developing distributed applications.
- Deployment complexity. In production, there is also the operational complexity of deploying and managing a system comprised of many different services.
- Increased memory consumption. The microservice architecture replaces N monolithic application instances with NxM services instances. If each service runs in its own JVM (or equivalent), which is usually necessary to isolate the instances, then there is the overhead of M times as many JVM runtimes. Moreover, if each service runs on its own VM (e.g. EC2 instance), as is the case at Netflix, the overhead is even higher.

2.2 Docker

What is Docker?

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.



Why is Docker useful?

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow you to run many containers simultaneously on a given host. Containers are lightweight because they don't need the extra load of a hypervisor, but run directly within the host machine's kernel. This means you can run more containers on a given hardware combination than if you were using virtual machines. You can even run Docker containers within host machines that are actually virtual machines!

Docker provides tooling and a platform to manage the lifecycle of your containers:

- Develop your application and its supporting components using containers
- The container becomes the unit for distributing and testing your application
- When you're ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two

How Docker is used in our Project

The microservices used in the project are containerized and deployable using the Docker Compose tool, which enables us to orchestrate and deploy at once all the necessary microservices in order to make our web app running smoothly.

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.

The `docker-compose.yml` used for the deployment of our backend infrastructure is found below:

```yaml
version: '2'
services:
  flask:
    image: 'tiangolo/uwsgi-nginx-flask:python3.8-2020-08-16'
    environment:
      - USER
      - LISTEN_PORT=80
    restart: always
    volumes:
      - './nginx_flask_uwsgi:/app'
      - './requirements.txt:/requirements.txt'
      - '/tmp:/tmp'
      - '/etc/ssl:/etc/ssl'
    ports:
      - '443:443'
    command: bash -c "pip install -r /requirements.txt && ../start.sh"
    tty: true
    stdin_open: true
  mongo:
    image: 'mongo:4.4.2'
    environment:
      - MONGO_INITDB_DATABASE=wasm
    restart: always
    volumes:
      - './dump:/dump'
    ports:
      - '27017:27017'
    command: bash -c "docker-entrypoint.sh mongod"
  mongo-seed:
    image: 'stefanwalther/mongo-seed:master'
    environment:
      - MONGODB_HOST=mongo
      - MONGODB_PORT=27017
    volumes:
```

```
      - './dump:/data'
    depends_on:
      - mongo
    command: mongorestore --host mongo --port 27017 /data --drop
  mongo-express:
    image: 'mongo-express:0.54'
    environment:
      - ME_CONFIG_SITE_GRIDFS_ENABLED=true
    restart: always
    ports:
      - '8081:8081'
    depends_on:
      - mongo
  zookeeper:
    image: 'wurstmeister/zookeeper:latest'
    ports:
      - '2181:2181'
    restart: always
    tty: true
    stdin_open: true
  kafka:
    image: 'wurstmeister/kafka:2.13-2.6.0'
    ports:
      - '9092:9092'
    environment:
      KAFKA_ADVERTISED_LISTENERS: 'INSIDE://:9092,OUTSIDE://:9094'
      KAFKA_LISTENERS: 'INSIDE://:9092,OUTSIDE://:9094'
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: 'INSIDE:PLAINTEXT,OUTSIDE:PLAINTEXT'
      KAFKA_INTER_BROKER_LISTENER_NAME: INSIDE
      KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
      KAFKA_BROKER_ID: '1'
      KAFKA_CREATE_TOPICS: 'C_TOPIC:1:1,GO_TOPIC:1:1'
      KAFKA_MAX_REQUEST_SIZE: 104857600
      KAFKA_MESSAGE_MAX_BYTES: 104857600
    restart: always
    volumes:
      - '/var/run/docker.sock:/var/run/docker.sock'
    depends_on:
      - zookeeper
    tty: true
    stdin_open: true
  emscripten_compiler:
    image: 'robertaboukhalil/emsdk:1.39.1'
    volumes:
      - '/tmp:/tmp'
    restart: always
    command: bash
    stdin_open: true
    tty: true
  emscripten_consumer:
    image: 'python:3.8'
```

```
    environment:
      - USER
    volumes:
      - '/tmp:/tmp'
      - './:/app'
      - '/var/run/docker.sock:/var/run/docker.sock'
    command: >-
      bash -c "pip install -r /app/requirements.txt && python
      /app/consumer/consumer.py -l C -t 5"
    depends_on:
      - kafka
      - emscripten_compiler
    tty: true
    stdin_open: true
    restart: always
  emscriptenpp_compiler:
    image: 'robertaboukhalil/emsdk:1.39.1'
    volumes:
      - '/tmp:/tmp'
    command: bash
    stdin_open: true
    tty: true
    restart: always
  emscriptenpp_consumer:
    image: 'python:3.8'
    environment:
      - USER
    volumes:
      - '/tmp:/tmp'
      - './:/app'
      - '/var/run/docker.sock:/var/run/docker.sock'
    command: >-
      bash -c "pip install -r /app/requirements.txt && python
      /app/consumer/consumer.py -l C++ -t 5"
    depends_on:
      - kafka
      - emscripten_compiler
    tty: true
    stdin_open: true
    restart: always
  golang_compiler:
    image: 'golang:1.15.5'
    volumes:
      - '/tmp:/tmp'
      - './consumer/assets:/assets'
    command: bash
    stdin_open: true
    tty: true
    restart: always
  golang_consumer:
    image: 'python:3.8'
```

```
137     environment:
138       - USER
139     volumes:
140       - '/tmp:/tmp'
141       - './:/app'
142       - '/var/run/docker.sock:/var/run/docker.sock'
143     command: >-
144       bash -c "pip install -r /app/requirements.txt && python
145       /app/consumer/consumer.py -l Go -t 5"
146     depends_on:
147       - kafka
148       - golang_compiler
149     tty: true
150     stdin_open: true
151     restart: always
```

Figure 2.1: Docker Compose YAML file used for our project.

By running the `docker compose up` command, all our containerized services are getting started and deployed one by one, each one as a separate process, and form a Docker network that we can use for internal communication between our services.

2.3 Firebase

Firebase is Google's application development platform that helps building, improving, and growing a web app. We utilize this platform in order to host our Angular frontend web app via Firebase Hosting.

Firebase Hosting is production-grade web content hosting for developers. With a single command, you can quickly deploy web apps and serve both static and dynamic content to a global CDN (content delivery network).

Firebase Hosting is built for the modern web developer. Websites and apps are more powerful than ever with the rise of front-end JavaScript frameworks like Angular and static generator tools like Jekyll. Whether you are deploying a simple app landing page or a complex Progressive Web App (PWA), Hosting gives you the infrastructure, features, and tooling tailored to deploying and managing websites and apps.

Chapter 3

Project Infrastructure

In this chapter we are going to list and get familiar with all the components and technologies used in our project from backend to frontend. The project structure is designed having multiple connected micro-services in mind, in order to serve its goal. A rough design of those connected micro-services can be shown below:

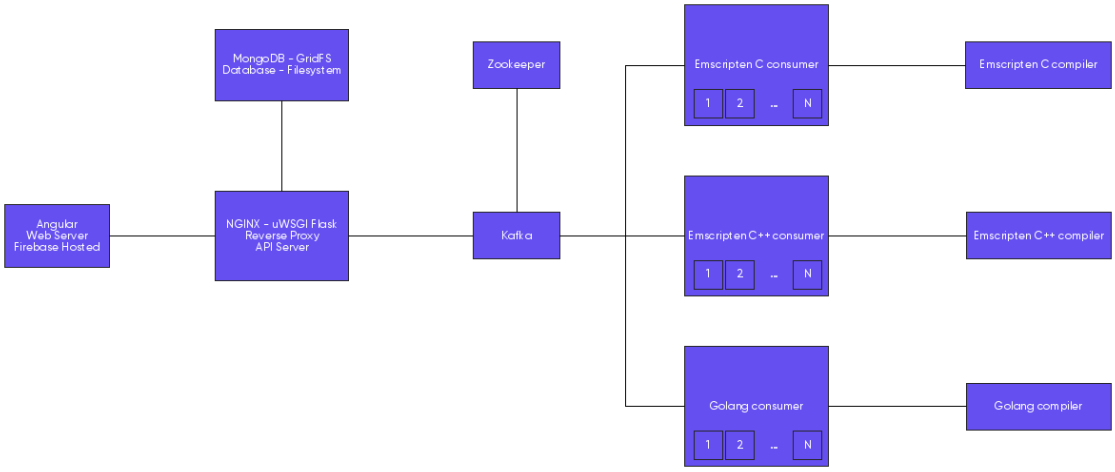


Figure 3.1: WebAssembly Workshop's micro-service infrastructure.

3.1 Backend

3.1.1 NGINX

What is NGINX?

NGINX is open source software for web serving, reverse proxying, caching, load balancing, media streaming, and more. It started out as a web server designed for maximum performance and stability. In addition to its HTTP server capabilities, NGINX can also function as a proxy server for email (IMAP, POP3, and SMTP) and a reverse proxy and load balancer for HTTP, TCP, and UDP servers.

What is a Reverse Proxy Server?

A proxy server is a go-between or intermediary server that forwards requests for content from multiple clients to different servers across the Internet. A reverse proxy server is a type of proxy server that typically sits behind the firewall in a private network and directs client requests to the appropriate backend server. A reverse proxy provides an additional level

of abstraction and control to ensure the smooth flow of network traffic between clients and servers.

Common uses for a reverse proxy server include:

- Load balancing – A reverse proxy server can act as a "traffic cop," sitting in front of your backend servers and distributing client requests across a group of servers in a manner that maximizes speed and capacity utilization while ensuring no one server is overloaded, which can degrade performance. If a server goes down, the load balancer redirects traffic to the remaining online servers.
- Web acceleration – Reverse proxies can compress inbound and outbound data, as well as cache commonly requested content, both of which speed up the flow of traffic between clients and servers. They can also perform additional tasks such as SSL encryption to take load off of your web servers, thereby boosting their performance.
- Security and anonymity – By intercepting requests headed for your backend servers, a reverse proxy server protects their identities and acts as an additional defense against security attacks. It also ensures that multiple servers can be accessed from a single record locator or URL regardless of the structure of your local area network.

How NGINX is used in our Project

NGINX provides us a great solution as a reverse proxy server, redirecting all incoming client requests to our uWSGI-Flask app, our API server. Also, it applies any header information required in all requests, either incoming or outgoing, resolving CORS issues and resolving the requests' origin.

NGINX and our uWSGI-Flask app share the same container, as they are built in a common Docker image. Therefore, this container is obliged for both running our NGINX web server, and our Flask application alongside it.

3.1.2 Flask - uWSGI

What is Flask?

Flask is a web framework, it's a Python module that lets you develop web applications easily. It does have many features like url routing, template engine. It is a WSGI web app framework. The Web Server Gateway Interface (Web Server Gateway Interface, WSGI) has been used as a standard for Python web application development. WSGI is the specification of a common interface between web servers and web applications.
Flask is often referred to as a microframework. It is designed to keep the core of the application simple and scalable. Instead of an abstraction layer for database support, Flask supports extensions to add such capabilities to the application.

What is uWSGI?

In order to deploy a web application written in Python, you would typically need two supporting components. The first is a traditional web server such as **NGINX** to perform basic web server tasks such as caching, serving static content, and handling inbound connections. The second is an application server such as uWSGI. In this context, an application server is a service that acts as a middleware between the application and the traditional web server. The role of an application server typically includes starting

the application, managing the application, as well as handling incoming connections to the application itself. With a web-based application, this means accepting HTTP requests from the web server and routing those requests to the underlying application.

uWSGI is an application server commonly used for Python applications. However, uWSGI supports more than just Python; it supports many other types of applications, such as ones written in Ruby, Perl, PHP, or even Go. Even with all of these other options, uWSGI is mostly known for its use with Python applications, partly because Python was the first supported language for uWSGI. Another thing uWSGI is known for is being performant.

How Flask is used in our Project

Flask is the web framework our backend API server is actually implemented with. In our Python Flask codebase resides the core logic and functionality of our backend infrastructure. Flask can process our requests and do all the CRUD operations needed in order to manipulate our **MongoDB** database, use **Kafka producers** to send projects to the corresponding **Kafka topic**, for compilation by our **compiler containers**, and also return the compiled `WASM` and `JS` files back to the clients, in order to be executed in their browsers.

3.1.3 MongoDB

What is MongoDB?

MongoDB is a document-oriented **NoSQL** database used for high volume data storage. Instead of using tables and rows as in the traditional relational databases, **MongoDB** makes use of collections and documents. Documents consist of key-value pairs which are the basic unit of data in **MongoDB**. Collections contain sets of documents and function which is the equivalent of relational database tables.

How MongoDB is used in our Project?

MongoDB plays a key role in our project as it stores all user accounts and their projects for future editing and compilation.
All files are stored with a unique id. Each project is a nested structure containing all file ids for each level. When we need to get a file, Flask runs some recursive functions in order to resolve the contents of all the files needed. **GridFS** is responsible for resolving each unique id. This way, we get back the initial project for editing or sent it to the user.

3.1.4 Kafka

What is Kafka?

Kafka is a distributed streaming platform that is used publish and subscribe to streams of records. It is used for fault tolerant storage and replicates topic log partitions to multiple servers. It is designed to allow your apps to process records as they occur. **Kafka** is fast and uses IO efficiently by batching and compressing records. Its purpose is to decouple data streams and stream data into data lakes, applications, and real-time stream analytics systems.

How does Kafka work?

Applications (producers) send messages (records) to a Kafka node (broker) and said messages are processed by other applications called consumers. Said messages get stored in a topic and

consumers subscribe to the topic to receive new messages.

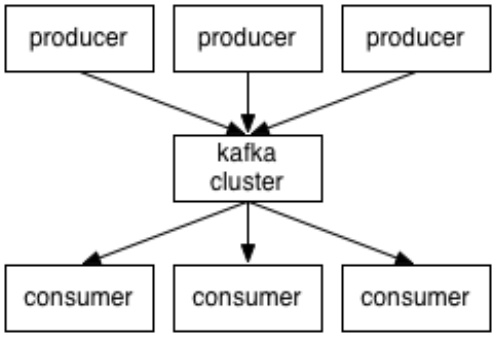


Figure 3.2: Communication between producers and consumers via the Kafka cluster.

As topics can get quite big, they get split into partitions of a smaller size for better performance and scalability. (ex: say you were storing user login requests, you could split them by the first character of the user's username) Kafka guarantees that all messages inside a partition are ordered in the sequence they came in. The way you distinct a specific message is through its offset, which you could look at as a normal array index, a sequence number which is incremented for each new message in a partition.

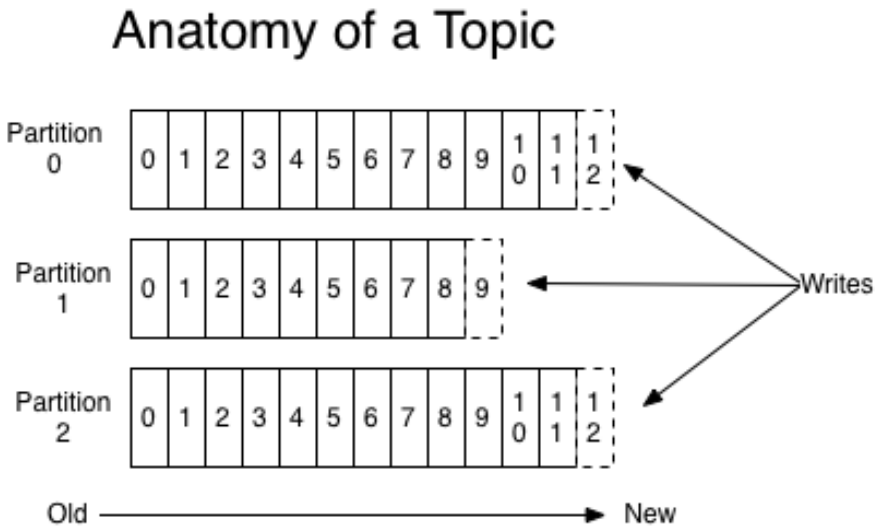


Figure 3.3: Brief design of a Kafka topic.

Kafka follows the principle of a dumb broker and smart consumer. This means that Kafka does not keep track of what records are read by the consumer and delete them but rather stores them a set amount of time (e.g one day) or until some size threshold is met. Consumers themselves poll Kafka for new messages and say what records they want to read. This allows them to increment/decrement the offset they're at as they wish, thus being able to replay and reprocess events.

It is worth noting that consumers are actually consumer groups which have one or more consumer processes inside. In order to avoid two processes reading the same message twice, each partition is tied to only one consumer process per group.

How Kafka is used in our Project?

Kafka is used in our project as intermediate between **Flask** and the **compilers**. In our project **Kafka** consists of three topics, one for each language:

- C topic
- C++ topic
- Go topic

When a valid compilation request is sent, **Flask** collects the project (with all its files) from our database and send them to suitable topic. More specifically, a new producer is created that sends a message containing the project that is ready for compilation. On the other side, when the responsible consumer of this topic is freed and identifies that **Kafka** contains new information, it consumes the message and triggers the appropriate compiler. Each consumer has five threads, so that five projects maximum can be compiled at the same time for better performance. The importance of **Kafka** is observed when there is a situation of heavy load.

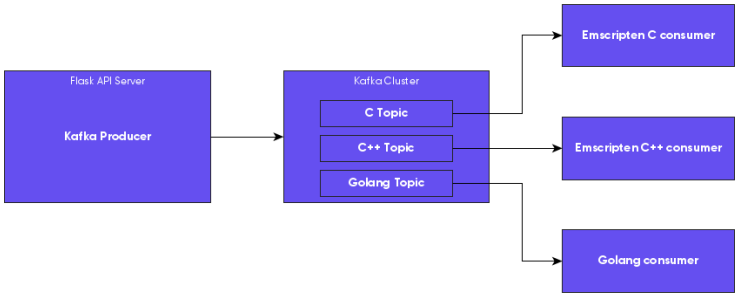


Figure 3.4: Brief design of Kafka topics and consumers in our project.

Kafka is responsible for balancing the load and maintain a more stable user experience.

3.1.5 Zookeeper

What is Zookeeper?

Zookeeper is a top-level software developed by Apache that acts as a centralized service and is used to maintain naming and configuration data and to provide flexible and robust synchronization within distributed systems. Zookeeper keeps track of status of the Kafka cluster nodes and it also keeps track of Kafka topics, partitions etc.

Zookeeper itself is allowing multiple clients to perform simultaneous reads and writes and acts as a shared configuration service within the system. The Zookeeper atomic broadcast (ZAB) protocol is the brains of the whole system, making it possible for Zookeeper to act as an atomic broadcast system and issue orderly updates.

How does Zookeeper work?

The data within **Zookeeper** is divided across multiple collection of nodes and this is how it achieves its high availability and consistency. In case a node fails, **Zookeeper** can perform instant failover migration; e.g. if a leader node fails, a new one is selected in real-time by polling within an ensemble. A client connecting to the server can query a different node if the first one fails to respond.

Why is Zookeeper necessary for Apache Kafka?

- **Controller election**

The controller is one of the most important broking entity in a **Kafka** ecosystem, and it also has the responsibility to maintain the leader-follower relationship across all the partitions. If a node by some reason is shutting down, it's the controller's responsibility to tell all the replicas to act as partition leaders in order to fulfill the duties of the partition leaders on the node that is about to fail. So, whenever a node shuts down, a new controller can be elected and it can also be made sure that at any given time, there is only one controller and all the follower nodes have agreed on that.

- **Configuration Of Topics**

The configuration regarding all the topics including the list of existing topics, the number of partitions for each topic, the location of all the replicas, list of configuration overrides for all topics and which node is the preferred leader, etc.

- **Access control lists**

Access control lists or ACLs for all the topics are also maintained within Zookeeper.

- **Membership of the cluster**

Zookeeper also maintains a list of all the brokers that are functioning at any given moment and are a part of the cluster.

3.2 Compilers

3.2.1 Emscripten

What is Emscripten?

Emscripten is a complete open source compiler toolchain to **WebAssembly**. Using Emscripten you can:

- Compile C and C++ code, or any other language that uses LLVM, into WebAssembly, and run it on the Web, Node.js, or other wasm runtimes.
- Compile the C/C++ runtimes of other languages into WebAssembly, and then run code in those other languages in an indirect way (for example, this has been done for Python and Lua).

Practically any portable C or C++ codebase can be compiled into **WebAssembly** using **Emscripten**, ranging from high-performance games that need to render graphics, play sounds, and load and process files, through to application frameworks like Qt. Emscripten generates small and fast code! Its default output format is **WebAssembly** , a highly optimizable executable format, that runs almost as fast as native code, while being portable and safe. **Emscripten** does a lot of optimization work by careful integration with LLVM, Binaryen, Closure Compiler, and other tools.

How Emscripten is used in our Project?

Emscripten is our compiler for C and C++ written projects. Our goal was to have a great compiler in our backend so that the user doesn't need to study a bunch of information in order to use our online tool. **Emscripten** was an excellent solution as it has everything a user needs to create big things as simple as it can be with the power of C and C++.
Emscripten is running in both C and C++ compiler containers. When the appropriate consumer triggers the compiler into the container, **Emscripten** compiles the project based on instructions given into a *Makefile* or a *compile.sh* file the user can write. If none of them exists, standard single-file compilation is executed.

3.2.2 Golang

What is Golang?

Go is a compiled language. This means we must run our source code files through a compiler, which reads source code and generates a binary, or executable, file that is used to run the program. Examples of other popular compiled languages include C, C++, and Swift. Programs written in these languages are transformed into machine code and can perform extremely fast.

Now, speaking about the WASM Golang compiler, all we have to say is that we use exactly the same compiler passing some extra commands that are responsible for the `WASM` output.

How Golang compiler is used in our Project?

Golang compiler is our compiler for Go written projects. No further knowledge is required about **WebAssembly**. The user can code in Go as he codes locally in his computer and run all of his projects just inside his browser!
For **Golang** ther is only one container using the compiler and same as Emscripten, when the appropriate consumer triggers the compiler into the container, **Golang** compiler compiles the project based on instructions given into a *Makefile*. If id does not exist, standard single-file compilation is executed again.

3.3 Frontend

3.3.1 Angular

What is Angular?

Angular is a web development platform built in **TypeScript** that provides developers with robust tools for creating the client side of web applications. Released in 2010 and formerly known as AngularJS, **Angular** is a JavaScript framework that was initially geared toward building single-page applications. At the time, the popularity of SPAs was growing rapidly, and so did the popularity of the AngularJS framework. In 2016, Google presented a new, fully rewritten version of this tool, with new features that included semantic versioning based on the MAJOR.MINOR.PATCH scheme, a command-line interface (CLI), and an entirely component-based architecture.

Why use Angular?

So, what are the top benefits developers can get from using Angular in their projects? The strong sides of Angular include:

- **Detailed documentation**

  Angular boasts detailed documentation where developers can find all necessary information without asking their colleagues. As a result, developers can quickly come up with technical solutions and resolve emerging issues.

- **Support by Google**

  A lot of developers consider Google support another benefit of Angular, making the platform trustworthy. At ng-conf 2017, the developers of Angular confirmed that Google will support Angular on a long-term basis.

- **Great ecosystem of third-party components**

  The popularity of Angular has resulted in the appearance of thousands of additional tools and components that can be used in Angular apps. As a result, you can get additional functionality and productivity improvements.

- **Component-based architecture**

  Angular implements a component-based architecture. According to this architecture, an app is divided into independent logical and functional components. These components can easily be replaced and decoupled as well as reused in other parts of an app. In addition, component independence makes it easy to test a web app and ensure that every component works seamlessly.

- **Ahead-of-time compiler**

  Angular's AOT compiler converts TypeScript and HTML into JavaScript during the build process. This means that code is compiled before the browser loads your web app so that it's rendered much faster. An AOT compiler is also much more secure than a just-in-time (JIT) compiler.

- **CLI**

  It is probably the most beloved feature for the majority of Angular developers. It automates the whole development process making app initialization, configuration, and

development as easy as possible. The **Angular CLI** allows you to create a new Angular project, add features to it, and run unit and end-to-end tests with a few simple commands. It not only increases code quality but also greatly facilitates development.

- **Ivy Renderer**

  Ivy Renderer translates an app's components and templates into JavaScript code that can be displayed by a browser. The main characteristic of this tool is its tree shaking technique. During rendering, Ivy removes unused code to make it clearer and smaller. As a result, web apps load faster.

- **Angular Material**

  This collection of **Material Design** elements optimized for Angular lets developers easily integrate UI components.

- **Dependency injection**

  Dependency injection is quite an arguable advantage of Angular. In plain English, dependency injection refers to one object supplying the dependencies of another object. These dependencies define how various components are connected and show how changes in one part of the code affects other parts. On the one hand, using dependency injection makes code more readable and maintainable. It can greatly reduce the time spent testing and hence cut the costs of web development. Starting from version 2, Angular provides developers with a separate tree of dependency injectors that can be changed or replaced without reconfiguring all components. But on the other hand, dependency injection may be time-consuming and it may be hard to create dependencies for components.

Chapter 4

Common Use Cases

4.1 Build and Run Project

4.1.1 User Story

- For simple **C**, **C++** and **Go** programs, the user needs to do nothing special than just write the initial program and just press **Build & Run**.
- For bigger projects or **SDL** projects for **C** and **C++**, the user must write a *Makefile* and (not necessary in all occasions) a *compile.sh* file for telling the compiler the instructions for compiling the project. The full procedure on how to create these files are described in our info file that is auto-generated on project creation.

After the compilation is complete, the user can see the project running below the editor, in the output console. If the project includes graphics, a canvas is used to draw the expected output. For the project input, we use classic Javascript prompts for **C** and **C++** and the browser console for **Go** projects respectively.

4.1.2 Step by Step Data Flow

When the user hits the **Build & Run** button, a request is sent from his machine to our server that is handled by **Nginx** and satisfied write after by **Flask**. **Flask** is responsible for collecting all files of project from **MongoDB** in a nested format and send them to the appropriate compiler through **Kafka**. For each compiler there are 5 **Kafka** consumers (5 threads) that are responsible for getting all files from **Flask** and copy them in a specific directory into the appropriate compiler container. Finally, the consumer orders the compiler to proceed with the compilation process. At the end of compilation, the compiler crates a *logs.txt* file including all the compiler output on both cases (successful or unsuccessful compilation).

Alongside, Flask is waiting for *logs.txt* to be created into the corresponding directory of the compiler container. When this happens, **Flask** sends a response to the initial compile request that includes the compilation status (success or not). When **Angular** on user machine gets the response, another request is triggered that is satisfied again by **Flask**. Now, **Flask** has to modify the *output.js* produced by the compiler accordingly so that it can be executed by **Angular** without errors. After that, **Flask** serves the *output.js* file so that **Angular** can execute it.

- For **C** and **C++** projects, at the execution of *output.js*, *output.wasm* is requested alongside with maybe some other files (eg *.data*). In this case **Angular** sends the suitable request and these files are also served from **Flask**.
- For **Go** projects, we need both *output.js* and *output.wasm* in order to instantiate wasm and run the project. If more files are requested from *output.js* are again served from **Flask**.

After the compilation is complete, the user can get all files of his project (both source code and files produced after compilation).

4.2 Build and Download Project

4.2.1 User Story

- For simple **C**, **C++** and **Go** programs, the user needs again to simply write the initial program and just press **Build & Download**.
- For bigger projects or **SDL** projects for **C** and **C++**, the user must write a *Makefile* and (not necessary in all occasions) a *compile.sh* file for telling the compiler the instructions for compiling the project as described in the previous use case.

4.2.2 Step by Step Data Flow

When the user hits the **Build & Run** button, a request is sent from his machine to our server that is handled by **Nginx** and satisfied write after by **Flask**. **Flask** is responsible for collecting all files of project from **MongoDB** in a nested format and send them to the appropriate compiler through **Kafka**. For each compiler there are 5 **Kafka** consumers (5 threads) that are responsible for getting all files from **Flask** and copy them in a specific directory into the appropriate compiler container. Finally, the consumer orders the compiler to proceed with the compilation process. At the end of compilation, the compiler crates a *logs.txt* file including all the compiler output on both cases (successful or unsuccessful compilation).

Alongside, Flask is waiting for *logs.txt* to be created into the corresponding directory of the compiler container. When this happens, **Flask** checks for the compilation status (success or not). If compilation is successful, **Flask** creates a **zip** including all files (source and produced files) and serves it so that **Angular** on user machine can receive them. Finally, **Angular** downloads the compressed **zip** file into user machine. The user can now have all files created and edited on our site locally and in addition, he can use the output files (*output.wasm, output.js* etc.) however he wants without dependency on our online tool.

Bibliography

[Abou20] Robert Aboukhalil, *Level up with WebAssembly*, Electronic book, May 2020. Available from https://www.levelupwasm.com/.

[Berg20] Tim Berglund, "Kafka Introduction", https://kafka.apache.org/intro, 2011-2020.

[Cane16] Benjamin Cane, "Getting Every Microsecond Out of uWSGI", https://rollout.io/blog/getting-every-microsecond-out-of-uwsgi/, 2016.

[Hark18] Liliia Harkushko, "Angular: Best Use Cases and Reasons To Opt For This Tool", https://yalantis.com/blog/when-to-use-angular/, 2018.

[Hyke20] Solomon Hykes, "Docker overview", https://docs.docker.com/get-started/overview/, 2013-2020.

[Kozl17] Stanislav Kozlovski, "Thorough Introduction to Apache Kafka", https://hackernoon.com/thorough-introduction-to-apache-kafka-6fbf2989bbc1, 2017.

[Rama20] Naveen Ramanathan, "WebAssembly: Introduction to WebAssembly using Go", https://golangbot.com/webassembly-using-go/, 2020.

[Rich20] Chris Richardson, "What are microservices?", https://microservices.io/, 2020.

[Rona20] Armin Ronacher, "Flask (web framework)", https://en.wikipedia.org/wiki/Flask_(web_framework), 2010-2020.

[Rung21] Krishna Rungta, "What is MongoDB? Introduction, Architecture, Features & Example", https://www.guru99.com/what-is-mongodb.html, 2021.

[Syso20a] Igor Sysoev, "What Is a Reverse Proxy Server?", https://www.nginx.com/resources/glossary/reverse-proxy-server/, 2013-2020.

[Syso20b] Igor Sysoev, "What is NGINX?", https://www.nginx.com/resources/glossary/nginx/, 2013-2020.

[Zaka15] Alon Zakai, "Emscripten Documentation", https://emscripten.org/, 2015.