



Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης
Πολυτεχνική Σχολή
Τμήμα Ηλεκτρολόγων Μηχανικών &
Μηχανικών Υπολογιστών
Τομέας Ηλεκτρονικής και Υπολογιστών

Διπλωματική Εργασία

Σύστημα για την Αυτόματη Παρακολούθηση του Χρόνου Λειτουργίας και Απόκρισης ενός Ιστότοπου

Εκπόνηση:
Σεντονάς Σταύρος
ΑΕΜ: 9386

Επίβλεψη:
Καθ. Συμεωνίδης Ανδρέας
Δρ. Παπαμιχαήλ Μιχαήλ
Υπ. Δρ. Καρανικιώτης Θωμάς

Θεσσαλονίκη, Ιούνιος 2023

Περίληψη

Η εξέλιξη της τεχνολογίας και της πληθώρας εφαρμογών που αναπτύσσονται στα πλαίσια αυτής, καθιστούν επιτακτική την ανάγκη ύπαρξης συστημάτων που θα ελέγχουν την εύρυθμη λειτουργία τους. Πιο συγκεκριμένα μιλάμε για την εξέλιξη στο χώρο του διαδικτύου και των δομών που έχουν υλοποιηθεί πάνω σε αυτό.

Πλέον αναφερόμαστε σε ένα συνεχώς αυξανόμενο και ευρύ δίκτυο web εφαρμογών - λογισμικών ως υπηρεσίας (SaaS - Software as a Service) που ζουν στον Διαδίκτυο (World Wide Web). Η λειτουργία αυτών μπορεί να ελεγχθεί με διάφορους τρόπους. Από Unit Testing, στο πλαίσιο του κύκλου ανάπτυξης του λογισμικού (continuous integration, continuous deployment cycle) προκειμένου να ελεγχθεί λειτουργικά το σύστημα για την αποφυγή bugs, μέχρι και Παρακολούθηση Δικτύου (Network Monitoring), για να επιβεβαιωθεί η σωστή λειτουργία των συστημάτων καθόλη της διάρκεια του κύκλου ζωής τους.

Η παρούσα διπλωματική εστιάζει στην ανάπτυξη ενός συστήματος Παρακολούθησης Δικτύου και κατεπέκταση εφαρμογής που θα δίνει της δυνατότητα στους χρήστες της να παρακολουθούν, εύκολα, την ομαλή λειτουργία των διαδικτυακών σελιδών τους, είτε αυτά είναι εφαρμογές, είτε απλά στατικές σελίδες. Το σύστημα στηρίζεται στη βασική μέθοδο εντοπισμού διαθεσιμότητας μίας ιστοσελίδας, γνωστή και ως ping. Κάνοντας ping μπορούμε να πάρουμε χρήσιμη πληροφορία σχετικά με το αν το υπό μελέτη σύστημα μπορεί να ανταποκριθεί και εφόσον ανταποκριθεί σχετικά με το χρόνο που μεσολάβησε μέχρι να απαντήσει. Συνεχίζοντας την λογική πορεία ενός τέτοιου συστήματος μπορούμε ακόμα στο μήνυμα που στέλνουμε να έχουμε πληροφορία που θα επηρεάζει την απάντηση που θα περιμέναμε να δούμε, έχοντας έτσι έναν ακόμα μηχανισμό για την αναγνώριση και αποφυγή πιθανών bugs, ή λαθών κατά τη διαδικασία ανάπτυξης λογισμικών ως υπηρεσία.

Title

Development of a System for Uptime Status Monitoring

Abstract

The evolution of technology and the abundance of applications developed within this framework make it imperative to have systems that will control their smooth operation. More specifically, we are talking about monitoring in the realm of the internet and the structures implemented on it.

Today we refer to a constantly growing and extensive network of web applications - software as a service (SaaS) - that reside on the Internet (World Wide Web), whose operations can be monitored in various ways. From unit testing, within the software development cycle (continuous integration, continuous deployment) to ensure that the system functions are running properly and avoid bugs, to network monitoring, to verify the correct functioning of the systems throughout their life cycle.

This thesis focuses on the development of a Network Monitoring system and a web application that will enable its users to easily monitor how their systems operate, whether they are applications or simply static pages. The system is based on the basic method of checking the availability of a website, known as ping. By pinging, we can obtain useful information regarding whether the system under study can respond and, if so, the time it takes to respond. Continuing the logical progression of such a system, we can further enhance the message we send with predetermined data to check the response of the system and verify the returned data, thereby identifying and avoiding potential bugs or error during the software's development process

Sentonas Stavros

Electrical & Computer Engineering Department,
Aristotle University of Thessaloniki, Greece

June 2023

Περιεχόμενα

Περίληψη	3
Abstract	4
Ακρωνύμια	10
1 Εισαγωγή	12
1.1 Περιγραφή του Προβλήματος	13
1.2 Σκοπός - Συνεισφορά της Διπλωματικής Εργασίας	14
1.3 Διάρθρωση της Αναφοράς	14
2 Θεωρητικό Υπόβαθρο	16
2.1 Hypertext Transfer Protocol	16
2.1.1 Μέθοδοι	17
2.1.2 Εκδόσεις HTTP	17
2.1.3 Κωδικοί Κατάστασης	18
2.2 API	19
2.2.1 RESTful API	21
2.3 Avro	22
2.4 Στατιστικές μετρικές	24
2.5 Εργαλεία	25
2.5.1 Node.js	25
2.5.2 PM2	26
2.5.3 MongoDB	26
2.5.4 Google Cloud Storage	28
3 Βιβλιογραφική Αναζήτηση Τεχνολογιών Αιχμής	30
4 Υλοποιήσεις	37
4.1 Version 0.0.1	37
4.2 Version 0.1.0	40
4.3 Version 1.0.0	44
5 Μετρήσεις και Επίδειξη Συστήματος	51
5.1 Μετρήσεις	51
5.1.1 Περιγραφή	51
Βιβλιογραφία	56

Κατάλογος Σχημάτων

2.1	Βασική Δομή ενός αιτήματος http	17
2.2	Αρχές και Καλές Πρακτικές Σχεδίασης REST API	22
2.3	Δομή ενός avro αρχείου	23
2.4	Σχεδίαση συστήματος με χρήση Node.js	25
2.5	Παράδειγμα Παρακολούθησης διεργασιών με τη χρήση του PM2	26
2.6	Mongoose, ένα abstract layer μεταξύ Node και mongo	27
3.1	Παράδειγμα χρήσης του εργαλείου Better Uptime	31
3.2	Παράδειγμα χρήσης του εργαλείου Uptime Robot	31
3.3	Παράδειγμα χρήσης του εργαλείου Site24x7	32
3.4	Παράδειγμα χρήσης του εργαλείου Uptimeia	32
3.5	Αρχιτεκτονική Συστήματος Nagios	33
3.6	Παράδειγμα χρήσης του εργαλείου Kuma Uptime	34
4.1	Διάγραμμα πρώτης Υλοποίησης	39
4.2	Διάγραμμα δεύτερης Υλοποίησης	40
4.3	Κύκλος Ζωής εκτέλεσης ενός επαναλαμβανόμενου Αιτήματος	41
4.4	Σύγκριση χρήσης ή μη timeouts στον κύκλο ζωής εκτέλεσης ενός αργού επαναλαμβανόμενου αιτήματος	42
4.5	Επικοινωνία σέρβερ-worker μέσω websockets	43
4.6	Διάγραμμα Τελικής Υλοποίησης	44
4.7	Παράδειγμα εγγραφής ενός Job στη βάση	46
4.8	Παράδειγμα εγγραφής ενός Api στη βάση	46
4.9	Παράδειγμα εγγραφών Response στη βάση	48
4.10	Τελικό διάγραμμα Lychte	49
4.11	Schemas πληροφορίας που αποθηκεύουμε σε avro αρχεία	50
5.1	Αποτελέσματα χρήσης της CPU	52
5.2	Αποτελέσματα χρήσης της RAM	52
5.3	Lychte Api Overview	53
5.4	Lychte Create Api Check	53
5.5	Lychte Create Api Advanced Check	54
5.6	Lychte Create Api Advanced v2 Check	54
5.7	Διάγραμμα Διάρκειας Αποκρίσεων	55
5.8	Διάγραμμα Κατάστασης Αποκρίσεων	55
5.9	Μετρικές που αφορούν το σύνολο των δεδομένων (ιστορικά δεδομένα)	55

5.10 Διάγραμμα Θηκογραμμάτων (Boxplots) της διάρκειας απόκρισης αι- τημάτων για κάθε μέρα (ιστορικά δεδομένα)	55
----------------------------------------------------------------------------------------------------------------------------	----

Κατάλογος Πινάκων

1.1	Χαρακτηριστικά Ενεργής και Παθητικής Παρακολούθησης	13
3.1	Σύγκριση Εργαλείων Ενεργής Παρακολούθησης	35
5.1	Τεχνικά χαρακτηριστικά Υπολογιστικού Συστήματος	51
5.2	Αποτελέσματα Μετρήσεων	53

Ακρωνύμια Εγγράφου

Παρακάτω παρατίθενται ορισμένα από τα πιο συχνά χρησιμοποιούμενα ακρωνύμια της παρούσας διπλωματικής εργασίας:

RUM → Real User Monitoring
API → Application Programming Interface
SaaS → Software as a Service
HTTP → Hypertext Transfer Protocol
GCP → Google Cloud Storage

1

Εισαγωγή

Τα τελευταία χρόνια, ο κλάδος του Διαδικτύου προσεγγίζει ένα μεγαλύτερο κομμάτι ανθρώπων, τόσο από τη μεριά του καταναλωτή όσο και από τη μεριά του παραγωγού. Όσο αφορά τον καταναλωτή οι δυνατότητες που του προσφέρονται μπορούν να διακριθούν στους εξής τομείς:

- **Επικοινωνία:** το διαδίκτυο παρέχει τη δυνατότητα άμεσης επικοινωνίας μεταξύ μεγάλων αποστάσεων, που δεν περιορίζεται μόνο στο ακουστικό ερέθισμα, αλλά επιτρέπει και την μετάδοση οπτικο-ακουστικής πληροφορίας
- **Πρόσβαση Πληροφορίας:** ίσως το σημαντικότερο αγαθό που προσφέρει το διαδίκτυο είναι η πληθώρα πληροφορίας που στεγάζει. Μηχανές Αναζήτηση (search engines), Online Βάσεις Δεδομένων (online databases), και άλλου είδους εφαρμογών εκπαιδευτικού χαρακτήρα που δίνουν πρόσβαση σε άτομα που το επιθυμούν, να κάνουν έρευνα
- **Ποιότητα ζωής:** σε αυτή την κατηγορία περιλαμβάνονται όλες εκείνες οι υπηρεσίες που διευκολύνουν την καθημερινότητα των χρηστών. Online αγορές (eshops) που γλιτώνουν την αναμονή σε ουρές ή ακόμα επιτρέπουν την εύκολη αγορά προϊόντων από απομακρυσμένες περιοχές του πλανήτη, ψυχαγωγία και πρόσβαση σε υπηρεσίες που επιταχύνουν ενέργειες που υπό άλλες περιπτώσεις θα ήταν χρονοβόρες (online banking, πληρωμή λογαριασμών, κρατήσεις ξενοδοχείων/εισητηρίων)

Από τη μεριά του παραγωγού, τα μέσα που υπάρχουν για την ανάπτυξη τέτοιων εφαρμογών/υπηρεσιών/συστημάτων μέρα με τη μέρα αυξάνονται. Η ραγδαία εξέλιξη στον χώρο των cloud υποδομών, καθιστά ευκολότερη και επισπεύδει τόσο την δημιουργία διαδικτυακών εφαρμογών, και σελιδών σε ένα γενικότερο πλαίσιο, όσο και την μεγέθυνση και αύξηση αυτών (scale up). Μάλιστα η επιλογή κατάλληλου παρόχου τέτοιων υπηρεσιών αποτελεί ένα αρκετά σημαντικό αντικείμενο μελέτης

[1]. Πέρα από τον οικονομικό παράγοντα θα πρέπει να προσμετρηθούν οι παροχές, τα πλεονεκτήματα αλλά και η αποδοτικότητα που κάθε ένας προσφέρει.

Βλέποντας λοιπόν το πόσο συνυφασμένη είναι η ζωή του σύγχρονου ανθρώπου με το δίκτυο αλλά και τις δυνατότητες και τα μέσα που έχει ο καθένας για να αναπτύξει εφαρμογές σε αυτό, καθίσταται επιτακτική η ανάγκη ύπαρξης μηχανισμών που θα αναγνωρίζουν σφάλματα (bugs) και θα επιβλέπουν την ορθή λειτουργία των υπό μελέτη συστημάτων καθόλη τη διάρκεια ζωής τους.

1.1 ΠΕΡΙΓΡΑΦΗ ΤΟΥ ΠΡΟΒΛΗΜΑΤΟΣ

Η Παρακολούθηση (Monitoring) ενός συστήματος που "ζει" στο χώρο του διαδικτύου μπορεί να γίνει κυρίως με δύο τρόπους:

- **Ενεργή Παρακολούθηση (Active Monitoring):** έχει περισσότερο προγνωστικό και προληπτικό χαρακτήρα. Συχνά αναφέρεται και ως **Συνθετική παρακολούθηση (Synthetic Monitoring)**, λόγω της φύσης των ενεργειών της. Ουσιαστικά δημιουργεί πλασματικά api calls και όχι πραγματικά δεδομένα χρηστών προκειμένου να ελεγχθεί η απόκριση του υπό μελέτη συστήματος. Η συχνότητα αποστολής των συνθετικών αιτημάτων συνήθως ρυθμίζεται από το χρήστη.
- **Παθητική Παρακολούθηση (Passive Monitoring):** παρέχει μία πιο πλήρη εικόνα σχετικά με πως χρησιμοποιούνται οι πόροι του δικτύου καταγράφοντας, αποθηκεύοντας και αναλύοντας τα δεδομένα του χρήστη. Για αυτό πολλές φορές αναφέρεται στη βιβλιογραφία ως **Παρακολούθηση Πραγματικών Χρηστών (Real User Monitoring - RUM)**. Έτσι μπορεί κανείς να εντοπίσει τις τάσεις χρήσης του δικτύου για τη βελτίωση και βελτιστοποίησή του συστήματος.

Πίνακας 1.1: Χαρακτηριστικά Ενεργής και Παθητικής Παρακολούθησης

Ενεργή Παρακολούθηση (Active Monitoring)	Παθητική Παρακολούθηση (Passive Monitoring)
<ul style="list-style-type: none"> • Στηρίζεται σε συνθετικά API calls • Παράγει δεδομένα για συγκεκριμένες πτυχές του δικτύου • Μπορεί να μετρήσει την κίνηση εντός και εκτός του δικτύου • Μπορεί να εντοπίσει προβλήματα πριν ακόμα μπορέσουν να τα εντοπίσουν οι χρήστες 	<ul style="list-style-type: none"> • Αναλύει δεδομένα πραγματικών χρηστών • Πλήρης εικόνα της απόδοσης του δικτύου • Μετράει κίνηση μόνο εντός του δικτύου • Εντοπίζει προβλήματα που εμφανίζονται εκείνη τη στιγμή

Και οι δύο μέθοδοι έχουν πλεονεκτήματα και μειονεκτήματα, τα οποία φαίνονται και στον παραπάνω πίνακα 1.1. Όπως είναι εμφανές η παθητική παρακολούθηση γίνεται πάνω στο σύστημα που θέλουμε να μελετήσουμε, πράγμα το οποίο σήμαινει ότι σαν εξωτερικοί παράγοντες στο σύστημα δεν θα μπορέσουμε να προσφέρουμε ανάλογες υπηρεσίες. Για το λόγο αυτό συνεχίζουμε την ανάλυση στο πλαίσιο της Ενεργής Παρακολούθησης Δικτύων.

Οι βασικοί λόγοι που χρειάζονται τέτοιου είδους υπηρεσίες όπως αναφέρεται και στα [2], [3] είναι οι εξής:

- Βελτίωση προβλημάτων που σχετίζονται με την απόδοση του συστήματος πρώτου τα βιώσουν οι πραγματικοί χρήστες του συστήματος
- Ύπαρξη κάποιας μονάδας αξιολόγησης της απόδοσης του
- Αξιολόγηση του συστήματος υπό μεγαλύτερο φορτίο
- Διασφάλιση της Συμφωνίας Επιπέδου Υπηρεσιών (Service Level Agreement - SLA), μεταξύ του παρόχου υπηρεσιών και των χρηστών
- Παρέχει χρήσιμα δεδομένα ακόμα και σε καινούργια συστήματα που ακόμα μπορεί να μην έχουν χρήστες

1.2 ΣΚΟΠΟΣ - ΣΥΝΕΙΣΦΟΡΑ ΤΗΣ ΔΙΠΛΩΜΑΤΙΚΗΣ ΕΡΓΑΣΙΑΣ

Η παρούσα διπλωματική εργασία μελετά τη χρήση σύγχρονων τεχνολογιών για τη δημιουργία ενός συστήματος Ενεργής Παρακολούθησης (Active Monitoring) σε συνδυασμό με μία SaaS εφαρμογή που θα παρουσιάζει μέσα από διαγράμματα τα αποτελέσματα της ανάλυσης της πληροφορίας που εξάγεται.

Εξετάζονται διάφοροι τρόποι και υλοποιήσεις που δοκιμάστηκαν κατά τη διάρκεια εκπόνησης της διπλωματικής αυτής εργασίας, και τέλος θα αναλύσουμε τα αποτελέσματα που παράξαμε καθόλη της διάρκεια των πειραμάτων που διενεργήθηκαν.

1.3 ΔΙΑΦΘΡΩΣΗ ΤΗΣ ΑΝΑΦΟΡΑΣ

Η διάρθρωση της παρούσας διπλωματικής εργασίας είναι η εξής:

- **Κεφάλαιο 2:** Περιγράφονται τα βασικά εργαλεία και θεωρητικά στοιχεία στα οποία βασίστηκαν οι υλοποιήσεις
- **Κεφάλαιο 3** Αναφορά συστημάτων που ήδη χρησιμοποιούνται και παράθεση διαφορών με την υλοποίησή μας
- **Κεφάλαιο 4** Περιγραφή των υλοποιήσεων και πλήρης περιγραφή του τελικού συστήματος
- **Κεφάλαιο 5** Παρουσιάζονται τα τελικά συμπεράσματα.

- **Κεφάλαιο 6** Προτείνονται θέματα για μελλοντική μελέτη, αλλαγές και επεκτάσεις.

2

Θεωρητικό Υπόβαθρο

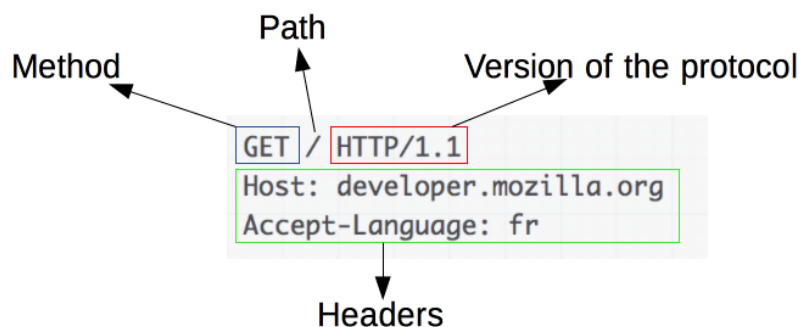
Στο κεφάλαιο αυτό θα παρουσιαστούν εργαλεία που χρησιμοποιήθηκαν για την υλοποίηση του Συστήματος Ενεργής Παρακολούθησης, καθώς και έννοιες και τεχνολογίες που αξιοποιήθηκαν για το σκοπό αυτό.

2.1 HYPERTEXT TRANSFER PROTOCOL

Το πρωτόκολλο επικοινωνίας HTTP (Hypertext Transfer Protocol) αποτελεί το πιο διαδεδομένο και ευρέως γνωστό πρωτόκολλο στο χώρο του διαδικτύου. Αναπτύχθηκε από τους Tim Berners-Lee και την ομάδα του το 1990 και από τότε έχει περάσει πολλές αλλαγές προκειμένου να μπορεί να ανταπεξέλθει στις ολοένα και συνεχώς αυξανόμενες ανάγκες του σήμερα.

Αποτελεί τη βάση κάθε μετάδοσης πληροφορίας στο διαδίκτυο. Στηρίζεται στην επικοινωνία δύο υπολογιστών, ενός που κάνει τα αιτήματα (client) και ενός που απαντά σε αυτά (server). Στο τέλος της επικοινωνίας στην μεριά του παραλήπτη θα υπάρχει ανακατασκευασμένο ένα ολοκληρωμένο αρχείο, από τα διάφορα υποαρχεία που μαζεύτηκαν, που μπορεί να είναι αρχεία ήχου, εικόνας, video. Τα αιτήματα αυτού που ξεκινάει την επικοινωνία ονομάζονται requests, ενώ οι απαντήσεις του αποστολέα responses.

Η βασική δομή ενός http αιτήματος [σχήμα 2.1](#) περιληπτικά περιλαμβάνει τη μέθοδο (method) του αιτήματος, που περιγράφει τη βασική λειτουργία του, το μονοπάτι (path) στο οποίο θα επικοινωνήσει με τον server, την έκδοση του πρωτοκόλλου που θα χρησιμοποιηθεί και τέλος headers προκειμένου να κρίνει ο server αν πρέπει να απαντήσει ή όχι πίσω στον client



Σχήμα 2.1: Βασική Δομή ενός αιτήματος http

2.1.1 Μέθοδοι

Πιο συγκεκριμένα οι βασικές μέθοδοι που παρέχει το http και οι συνήθεις λειτουργίες τους είναι οι εξής:

- **GET**: παίρνει πληροφορία από τον server
- **POST**: υποβάλλει πληροφορία, προκαλώντας αλλαγές στον τρόπο λειτουργίας του server. Σχετίζεται συχνά με τη δημιουργία πληροφορίας που προηγουμένως δεν υφίστατο
- **PUT**: όπως και πριν στέλνει πληροφορία στον παραλήπτη υπολογιστή, αλλά αυτή τη φορά επηρεάζει πόρους που ήδη υπήρχαν στο σύστημα. Σχετίζεται συχνά με την τροποποίηση ήδη υπάρχουσας πληροφορίας
- **DELETE**: διαγράφει από το σύστημα του server το συγκεκριμένο πόρο.

Αξίζει να σημειωθεί ότι πέρα από τις τέσσερις αυτές βασικές μεθόδους υπάρχουν και άλλες όπως είναι η **PATCH** που αποτελεί ειδική περίπτωση της **PUT**, η **HEAD** που αποτελεί ειδική περίπτωση της **GET**, καθώς και άλλες που σχετίζονται με τη σύνδεση μεταξύ server και client. Αυτές είναι οι **CONNECT**, **OPTIONS** και **TRACE**

2.1.2 Εκδόσεις HTTP

Η πρώτη έκδοση του HTTP παρόλο που δεν είχε κάποια συγκεκριμένο τίτλο, εκ των υστέρων το αποκάλεσαν HTTP/0.9. Αποτελεί την πιο απλή έκδοση του πρωτοκόλλου. Δεν υποστηρίζονταν headers, κωδικοί κατάστασης (status codes). Εξυπηρετούσε μόνο GET αιτήματα και η μοναδική απάντηση που μπορούσε να επιστρέψει ήταν hypertext αρχεία. Κάθε φορά που ο server ανταποκρινόταν και έστελνε απάντηση η επικοινωνία με τον client έκλεινε κατευθείαν.

Στη συνέχεια και με την ανάπτυξη του διαδικτύου προστέθηκαν και άλλες λειτουργίες. Πέριξ του 1996, με τη επόμενη έκδοση του πρωτοκόλλου (HTTP/1.0) τα αιτήματα πλέον συνοδεύονταν από headers, μεταπληροφορία σχετικά με τη

κατάσταση του αιτήματος, τον τύπο της πληροφορίας που περιμένουμε να έρθει (stylesheets, media, hypertext) καθώς και την έκδοση του HTTP που χρησιμοποιήθηκε στη συγκεκριμένη επικοινωνία. Επιπλέον πέρα από τη GET μέθοδο υπάρχει η δυνατότητα για POST και PUT, δημιουργία και τροποποίηση πληροφορίας δηλαδή.

Στη συνέχεια το HTTP/1.1 προσπαθεί να βελτιώσει τις ήδη υπάρχουσες δυνατότητες κάνοντας την επικοινωνία μεταξύ server και client πιο αποδοτική. Αντί να κλείνει η επικοινωνία μετά από κάθε μήνυμα, η σύνδεση παραμένει ανοιχτή γλιτώνοντας έτσι μία σταθερή καθυστέρηση που υπήρχε σε κάθε αίτημα

Φτάνοντας στο σήμερα, μιλάμε για το HTTP/2.0 [4]. Αξιοποιώντας το πρωτόκολλο SPDY (Speedy) που αναπτύχθηκε κάποια χρόνια πριν την κυκλοφορία του, και κτίζοντας πάνω σε αυτό, κατάφερε να μειώσει τους χρόνους επικοινωνίας server-client. Μερικοί από τους τρόπους που επιτυγχάνεται αυτό είναι η μετατροπή του http από text πρωτόκολλο, σε δυαδικό (binary protocol), επιτρέποντας έτσι χρήση καλύτερων και αποδοτικότερων τεχνικών επικοινωνίας. Επιπλέον συμπιέζει τους headers (header compression) καθώς αποτελούν πληροφορία που επαναλαμβάνεται όταν τα αιτήματα στον server είναι συνεχή. Ο server ακόμα, αποκτά έναν μηχανισμό (server-push) που του επιτρέπει να προωθεί πληροφορία στον client (στην cache του client συγκεκριμένα), που δεν έχει ζητήσει ακόμα, αλλά βάση αυτού που αιτείται, μάλλον θα ζητήσει εντός του ίδιας συνεδρίας.

Τέλος, πρέπει να αναφερθούμε στην τελευταία, αν και όχι ακόμα ευρέως διαδεδομένη, έκδοση HTTP/3.0. Η βασική διαφορά με τους προκάτοχους του είναι ότι αλλάζει το πρωτόκολλο επικοινωνίας που χρησιμοποιεί όλα αυτά τα χρόνια, από TCP (Transfer Communication Protocol) σε έναν συνδυασμό UDP (User Datagram Protocol) και QUIC, μίας νέας τεχνολογίας που λύνει το πρόβλημα και βελτιστοποιεί τόσο το πρόβλημα της ασφάλειας των επικοινωνιών (TLS handshakes), όσο και της απώλειας πληροφορίας που μπορεί να υπήρχε λόγω UDP, πρωτοκόλλου που είναι γνωστό για την ταχύτερη απόδοσή του σε σχέση με το TCP, αλλά και το γεγονός ότι είναι πιο επιρρεπές σε σφάλματα. Η νέα αυτή έκδοση από τα αποτελέσματα του [5] φαίνεται να έχει ήδη καλύτερους χρόνους σε σχέση με τις παλαιότερες εκδόσεις και ήδη το 28% του διαδικτύου αξιοποιεί τις δυνατότητές του.

2.1.3 Κωδικοί Κατάστασης

Οι κωδικοί κατάστασης (status codes) αποτελούν μέρος της απάντησης του server. Σου επιτρέπουν με μία ματιά να καταλάβεις αν το αίτημα που έχεις κάνει έχει επιστρέψει σωστά, ή έχει γίνει κάποιο λάθος στη μεριά του server. Υπάρχουν πέντε μεγαλύτερες κατηγορίες που στεγάζουν όλες τις υποπεριπτώσεις αυτών. Πιο συγκεκριμένα:

- **Εύρος 100-199:** Υποδηλώνουν ενημερωτική απάντηση σχετικά με τη λειτουργία του server
- **Εύρος 200-299:** Επιτυχή αιτήματα.
- **Εύρος 300-399:** Υποδηλώνουν την ανακατεύθυνση του μηνύματος του client. Συνήθως συνοδεύονται από το νέο url στο οποίο πρέπει να αποστείλουν το αίτημα

- **Εύρος 400-499:** Ανεπιτυχές αίτημα, που οφείλεται στον client. Ένα σύνθημα παράδειγμα είναι αίτησης πρόσβασης σε προστατευόμενους πόρους χωρίς κάποιου είδους αυθεντικοποίησης, ή χωρίς τα σωστά στοιχεία για αυθεντικοποίηση
- **Εύρος 500-599:** Ανεπιτυχές αίτημα, που οφείλεται στον server.

2.2 API

Ένα ακόμα πολύ σημαντικό συστατικό του διαδικτύου αποτελούν οι Διεπαφές Εφαρμογών Προγραμμαμάτων (Application Programming Interfaces - APIs). Πρόκειται για το σύνολο των ορισμών, κανόνων και πρωτοκόλλων για τη δημιουργία και ενσωμάτωση μίας εφαρμογής. Ουσιαστικά λειτουργεί ως το συμβόλαιο μεταξύ ενός συστήματος παροχής υπηρεσίας και του χρήστη του συστήματος αυτού, καθορίζοντας την απαραίτητη πληροφορία που απαιτεί για να λειτουργήσει σωστά ο server αλλά και αντίστροφα, καθορίζοντας την απαραίτητη πληροφορία που απαιτεί ο client στην απάντηση που θα του επιστραφεί.

Είναι εμφανές λοιπόν ότι σε κάθε περίπτωση, αν θέλει κανείς να αλληλεπιδράσει με κάποιο σύστημα είτε για να αντλήσει πληροφορία, είτε για να στείλει πληροφορία (αποθήκευση ή τροποποίηση ήδη υπάρχουσας) θα πρέπει να υπάρχουν κανόνες που ορίζουν και καθιστούν πιο εύκολη και απλή την επικοινωνία αυτή.

Η έννοια του api δεν περιορίζεται φυσικά μόνο στο πλαίσιο του διαδικτύου, αλλά σε όλων των ειδών εφαρμογές που υπάρχει επικοινωνία ενός κεντρικού σημείου (server) με κάποιον χρήστη (client) που θέλει να κάνει χρήση των υπηρεσιών που αυτό προσφέρει.

Όσων αφορά τη διαθεσιμότητα και ασφάλεια των API, μπορούμε να διακρίνουμε τις εξής κατηγορίες:

- **Ανοιχτά (Open):** Σε αυτού του τύπου διεπαφών λογισμικού, έχει ελεύθερη πρόσβαση καθένας, χωρίς να απαιτείται κάποια παραπάνω πληροφορία που αφορά την αυθεντικοποίηση ή ταυροποίηση του χρήστη. Επειδή ακριβώς η πληροφορία που παρέχεται (ανοιχτά) είναι τεράστια, χρησιμοποιούνται πολύ συχνά σε έρευνες, όπως αυτή [6] που αξιοποιεί ελεύθερα προσβάσιμα πόρους για να αναλύσει τα πιο διαδεδομένα APIs.
- **Δημόσια (Public):** Μοιάζουν πολύ με τα **Ανοιχτού Τύπου API**, με τη μόνη διαφορά, τον περιορισμό πρόσβασης σε ορισμένα σημεία που απαιτούν κλειδιά προκειμένου να γίνει σε αντίθεση με πριν αυθεντικοποίηση και ταυτοποίηση.
- **Ιδιωτικά (Private):** Αφορούν διεπαφές λογισμικού που αναπτύσσονται και χρησιμοποιούνται μόνο εντός ενός κλειστού πλαισίου, όπως θα μπορούσε να είναι μία επιχείρηση ή κάποιο πανεπιστήμιο που παρέχει ορισμένες υπηρεσίες μόνο εντός του χώρου του. Πολλές φορές στη βιβλιογραφία αναφέρονται και ως **Εσωτερικά (Internal APIs)**

- **Εταιρικά (Partner):** Είναι περιορισμένα στο πλήθος χρηστών που έχουν πρόσβαση σε αυτό. Χρησιμοποιούνται για επικοινωνία μεταξύ συστημάτων εταιριών/επιχειρήσεων για την ανάπτυξη και αξιοποίηση εφαρμογών και υπηρεσιών. Συνήθως η ασφάλεια σε όλες αυτές τις αλληλεπιδράσεις είναι αρκετά πιο αυστηρή.
- **Σύνθετα (Composite):** Συνδυάζουν περισσότερα από ένα API αιτημάτων σε ένα, κάνοντας έτσι την επικοινωνία πιο αποδοτική (κερδίζοντας χρόνο από πολλά διαδοχικά API).

Αξίζει να σημειωθεί σε αυτό το σημείο ότι ο τρόπος δημιουργίας, η δομή καθώς και τα πρωτόκολλα που χρησιμοποιούν τα διάφορα APIs που υπάρχουν στο διαδίκτυο δεν είναι πάντα κοινά. Υπάρχουν κάποιες γνωστές αρχιτεκτονικές και πρωτόκολλα όπως θα δούμε και στη συνέχεια, που πέρα από τη δομή των συστημάτων παρέχουν και κανόνες "Καλύτερων Πρακτικών" (Best Practices). Οι πιο γνωστές αρχιτεκτονικές παρατίθενται στη συνέχεια:

- **SOAP (Simple Object Access Protocol):** Χρησιμοποιούνταν ευρέως στο παρελθόν πριν την εμφάνιση του REST. Η επικοινωνία μεταξύ server και client επιτυγχάνεται με την αποστολή αρχείων XML.
- **RPC (Remote Procedure Call):** Αποτελεί πρωτόλλο επικοινωνίας που επιτρέπει σε ένα σύστημα (client) να καλεί διαδικασίες (procedures) ή συναρτήσεις σε ένα άλλο σύστημα (server) είτε αυτά είναι στο ίδιο δίκτυο, είτε απομακρυσμένα. Σκοπός είναι να κάνει τα δύο συστήματα να επικοινωνούν σαν να βρίσκονται στο ίδιο υπολογιστικό σύστημα. Επιγραμματικά η διαδικασία επικοινωνίας δύο συστημάτων ξεκινάει με ένα αίτημα στον server να εκτελέσει κάποια συγκεκριμένη λειτουργία, παρέχοντας ορίσματα που ίσως χρειαστούν για αυτό. Ο server τέλος επιστρέφει την απάντηση της παραπάνω διαδικασίας στον client. Σήμερα το πρωτόκολλο αυτό χρησιμοποιείται σε διάφορες εφαρμογές και αναπτύσσονται μάλιστα συνεχώς, σύγχρονες τεχνολογίες που πατώντας πάνω στη βασική λειτουργία του πρωτοκόλλου κάνουν περαιτέρω βελτιστοποιήσεις (gRPC).
- **REST (Representational State Transfer):** Η πιο διαδεδομένη μορφή API στο διαδίκτυο. Η λειτουργία του είναι απλή. Η χρήστης στέλνει αίτημα σε έναν απομακρυσμένο σύστημα (server). Το σύστημα αντιδράει και βάσει των δεδομένων που του στέλνει ο χρήστης, τρέχει διαδικασίες ώστε να παράξει το επιθυμητό αποτέλεσμα, το οποίο τέλος το αποστέλει στον client.
- **Websocket:** Αποτελεί πρωτόκολλο που επιτρέπει την αμφίδρομη επικοινωνία μεταξύ client και server. Σε αντίθεση με το REST API, που κάθε φορά που γίνεται ένα αίτημα στον server πρέπει να αποστέλλονται οι κατάλληλοι headers και να γίνεται η κατάλληλη σύνδεση μεταξύ των δύο συστημάτων, με τη χρήση των websockets η σύνδεση αυτή εκτελείται μόνο μία φορά στο αρχικό αίτημα που αποστέλλεται. Αξίζει να σημειωθεί ακόμα ότι θεωρείται ιδανικό για εφαρμογές που απαιτούν άμεση ενημέρωση [7] (real-time-applications) τόσο για τον προαναφερθέντα λόγο, βάσει του οποίου μειώνεται δραστικά ο

χρόνος επικοινωνίας μεταξύ των εμπλεκόμενων, όσο και για το γεγονός ότι μηνύματα μπορούν να σταλούν και από τη μεριά του client προς τον server αλλά και το ανάποδο (server σε client) χωρίς να χρειάζεται να γίνει κάποιο αίτημα πρώτα.

2.2.1 RESTful API

Πρόκειται για ένα τύπο αρχιτεκτονικής λογισμικού για APIs, που αποτελείται από κατευθυντήριες γραμμές και βέλτιστες πρακτικές για τη δημιουργία επεκτάσιμων εφαρμογών στο διαδίκτυο. Πρωτάθηκε το 2000, από τον Thomas Fielding [8] σαν ένας τρόπος για να κατευθύνει την ανάπτυξη εφαρμογών προκειμένου να υπάρχει ένα κοινός τρόπος "γραφής" ώστε να υπάρξει εξέλιξη στον τομέα αυτό ακόμα πιο γρήγορα και οργανωμένα. Το δομικό συστατικό του είναι το πρωτόκολλο HTTP που χρησιμοποιεί για να πραγματοποιεί κλήσεις μεταξύ συστημάτων.

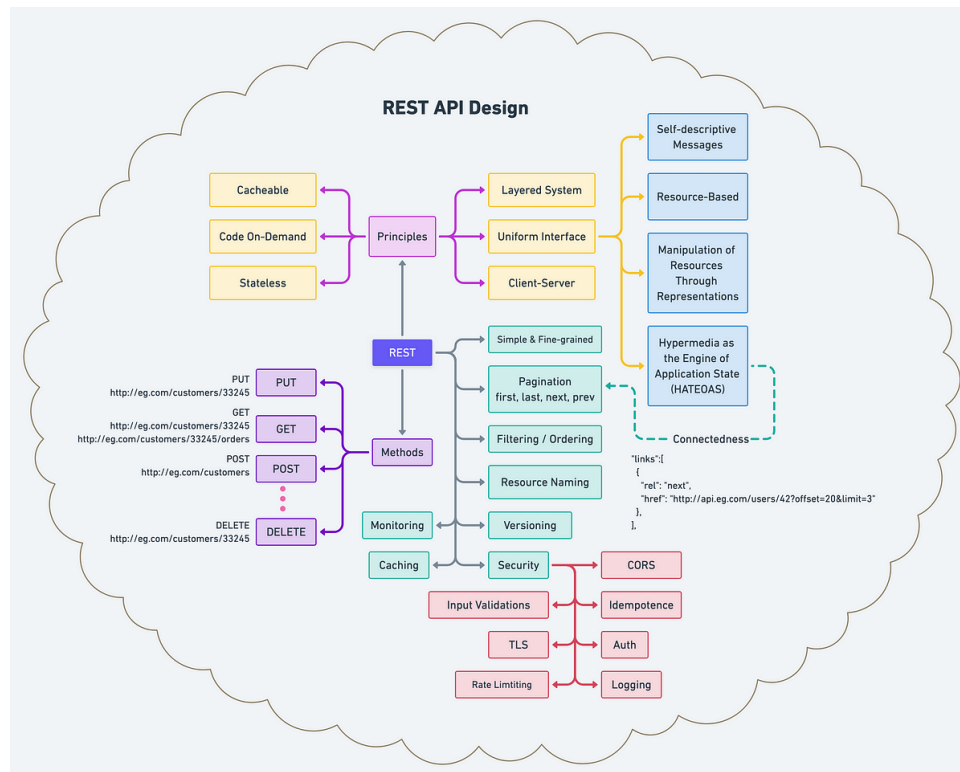
Όπως προαναφέρθηκε ο τρόπος λειτουργίας στην απλούστερη έκδοσή του ξεκινάει με ένα αίτημα κάποιου χρήστη σε ένα σύστημα της επιλογής του, παρέχοντας την απαραίτητη πληροφορία προκειμένου ο server να μπορέσει να αναταποκριθεί επιτυχώς. Έπειτα, και αφού το σύστημα διεκπεραιώσει όλες τις εσωτερικές λειτουργίες του, απαντά μεταφέροντας πίσω στον χρήστη την επιθυμητή πληροφορία και ενημερώνοντας τον σχετικά με την κατάσταση του αιτήματος του (σε περίπτωση που κάτι πήγε λάθος ο χρήστης θα πρέπει να ενημερώνεται αναλόγως).

Ο βασικός τρόπος αλληλεπίδρασης με τους πόρους του συστήματος θα πρέπει να γίνεται μέσα από τις τέσσερις βασικές μεθόδους του HTTP (**GET, POST, PUT, DELETE**), χωρίς αυτό να απαγορεύει τη χρήση των υπόλοιπων. Κάθε σύστημα θα πρέπει τυπικά να υποστηρίζει CRUD λειτουργίες (Create, Read, Update, Delete)

Για να χαρακτηριστεί ένα API ως RESTful θα πρέπει να ικανοποιεί τα παρακάτω, ενώ στο [σχήμα 2.2](#) μπορούμε να τα δούμε σε ένα γενικότερο πλαίσιο:

- **Client-Server:** Τα δύο αυτά υπολογιστικά συστήματα θα πρέπει να είναι ανεξάρτητα. Πρακτικά αυτό σημαίνει ότι ο client θα ασχοληθεί αποκλειστικά και μόνο με το κομμάτι παρουσιάσης της πληροφορίας, που του παρέχει ο server, στον χρήστη μέσα από διεπαφές γραφικού χαρακτήρα, ενώ ο server θα εκτελεί μόνο λειτουργίες που αφορούν την δημιουργία, τροποποίηση και διαγραφή των πόρων του συστήματος. Με αυτό τον διαχωρισμό πλέον ο client είναι πιο ελαφρύς, καθώς διαχειρίζεται μόνο πληροφορία που του έρχεται έτοιμη από το σύστημα, και ο server μπορεί να κάνει πιο εύκολα scaling.
- **Cacheable:** ο client θα πρέπει να αποθηκεύει προσωρινά (cache) τις απαντήσεις που λαμβάνει από τον server για την αποφυγή συνεχών επαναλαμβανόμενων αιτημάτων, βελτιώνοντας έτσι την απόδοση του συστήματος.
- **Stateless:** Η πληροφορία της κατάστασης του συστήματος δεν θα πρέπει να αποθηκεύεται αλλά κάθε αίτημα θα πρέπει να περιέχει την απαραίτητη πληροφορία για να εκτελεστεί από τη μεριά του server. Η πληροφορία αυτή μπορεί να αποτελεί μέρος του ίδιου του url, του body, των headers ή του query.
- **Πολυεπίπεδο Σύστημα (Layered System):** Τα υπολογιστικά συστήματα που συμμετέχουν στη διαδικασία αυτή δεν θα πρέπει να γνωρίζουν αν είναι άμεσα

συνδεδεμένα μεταξύ τους ή υπάρχουν ενδιάμεσοι κόμβοι που παρεμβάλλονται. Το σύστημα θα πρέπει να λειτουργεί δίχως να έχει επίγνωση των γειτόνων του, περιμένοντας απλά την κατάλληλη πληροφορία για να λειτουργήσει, τόσο από τη μεριά του server όσο και από τη μεριά του client.



Σχήμα 2.2: Αρχές και Καλές Πρακτικές Σχεδίασης REST API

2.3 AVRO

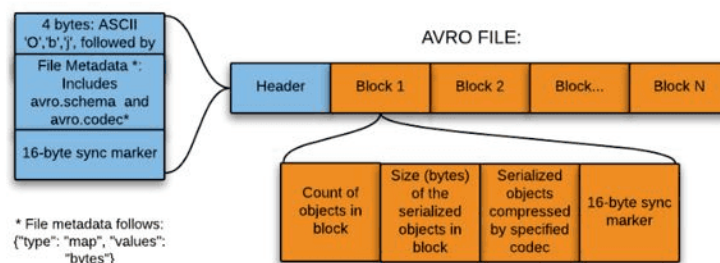
Αποτελεί μία μορφή αποθήκευσης πληροφορίας που λόγω της υλοποίησης του παράγει αρχεία πολύ μικρότερου μεγέθους από ότι θα αναμέναμε σε άλλου είδους μορφές αρχείων. Αναπτύχθηκε αρχικά για τη για την αποθήκευση και τη μετάδοση πληροφορίας εντός του framework Apache Hadoop, αλλά έκτοτε έχει χρήση και σε πλήθος άλλων εφαρμογών. Διαθέτει διεπαφές (API) που επιτρέπουν την ενσωμάτωση της τεχνολογίας αυτής σε προγράμματα γραμμένα σε πλήθος γλωσσών, μερικές εκ των οποίων είναι οι Java, C/C++/C#, Python, PHP, Ruby, Rust και JavaScript.

Η βασικότερη διαφορά μεταξύ αρχείων avro και άλλων αρχείων αποθήκευσης πληροφορίας, όπως είναι τα JSON αρχεία που αποτελούν τον πιο διαδεδομένο τρόπο αποθήκευσης στο χώρο του διαδικτύου, είναι το γεγονός ότι διαθέτουν το schema των δεδομένων (data schema definition) που περιέχουν. Ξέροντας τη μορφή της προς αποθήκευσης πληροφορίας καταφέρνουν να γλιτώσουν χώρο στο τελικό προϊόν. Αυτό σε συνδυασμό με το γεγονός ότι η πληροφορία αποθηκεύεται συμπιεσμένη

σε δυαδική μορφή (binary format) την καθιστά αρκετά σημαντική σε περιπτώσεις που έχουμε μεγάλο πλήθος πληροφορίας που θέλουμε να αποθηκεύσουμε.

Τα βασικότερα πλεονεκτήματα του παρατίθενται στη συνέχεια ενώ η δομή ενός τέτοιου αρχείου φαίνεται στο [σχήμα 2.3](#):

- Είναι χρήσιμος για την μετάδοση πληροφορίας, καθώς η αποθηκευμένη πληροφορία είναι αρκετά συμπυκνωμένη.
- Επιτρέπει την τροποποίηση και εξέλιξη του σχήματος των δεδομένων (schema evolution), δίνοντας τη δυνατότητα για αλλαγές και data migration που πολλές φορές καθίστανται επιτακτικά στη διάρκεια ζωής ενός προγράμματος που συνεχώς εξελίσσεται.
- Διαθέτει πλήθος τύπων δεδομένων απλών (primitive) και σύνθετων (complex) που μπορούν να χρησιμοποιηθούν
 - Primitive: boolean, int, long, float, double, bytes, string
 - Complex: record, enum, array, map, fixed, union
- Το schema εμπεριέχεται σε κάθε αρχείο. Κάνοντας χρήση αυτού κάθε χρήστης που έχει πρόσβαση σε ένα τέτοιο αρχείο μπορεί να διαβάσει τα περιεχόμενα του (εξαρχής binary, μη αναγνώσιμων από τον άνθρωπο) χωρίς να ξέρει από πριν τη μορφή της πληροφορίας.



Σχήμα 2.3: Δομή ενός avro αρχείου

2.4 ΣΤΑΤΙΣΤΙΚΕΣ ΜΕΤΡΙΚΕΣ

Στην υποενότητα αυτή θα δούμε τον τρόπο υπολογισμού μερικών κλασικών μετρικών της στατιστικής που χρησιμοποιούνται για την εξαγωγή συμπερασμάτων στο σύνολο των δεδομένων που αποθηκεύουμε. Σε όλους τους παρακάτω τύπους έχουμε δεδομένα x_i όπου, $i = 0, \dots, k$

1. Αριθμητική Μέση Τιμή:

$$mean(x) = \frac{\sum_{n=1}^k x_n}{k} \quad (2.1)$$

2. Διάμεσος:

$$median(x) = \begin{cases} x_{\frac{n+1}{2}} & \text{για } n \text{ περιττό} \\ \frac{x_{\frac{n}{2}} + x_{\frac{n}{2}+1}}{2} & \text{για } n \text{ ζυγό} \end{cases} \quad (2.2)$$

3. Τυπική Απόκλιση Πληθυσμού:

$$std(x) = \sqrt{\frac{\sum_{i=1}^n (x_i - mean(x))^2}{n-1}} \quad (2.3)$$

4. Τεταρτημόρια:

$$\begin{cases} q1(x) = mean(x_i) & \text{όπου } i = 0, \dots, k/2 \\ q3(x) = mean(x_j) & \text{όπου } j = k/2 + 1, \dots, k \end{cases} \quad (2.4)$$

2.5 ΕΡΓΑΛΕΙΑ

Στην τελευταία υποενότητα αυτού του κεφαλαίου θα μιλήσουμε για τα εργαλεία που χρησιμοποιήθηκαν για την υλοποίηση του συστήματος Αυτόματης Παρακολούθησης που υλοποιήσαμε στο πλαίσιο της διπλωματικής αυτής εργασίας.

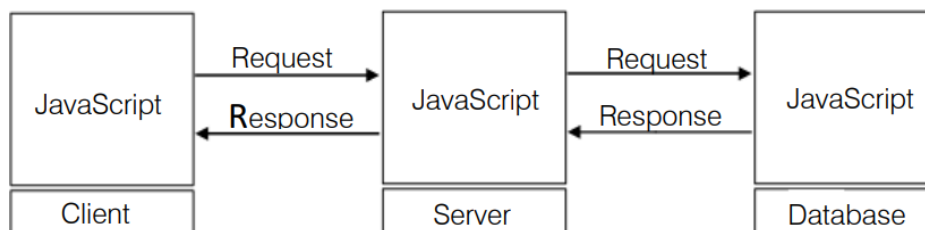
2.5.1 Node.js

Είναι ένα περιβάλλον εκτέλεσης (runtime) της προγραμματιστικής γλώσσας JavaScript. Πληθώρα μηχανικών λογισμικού το χρησιμοποιούνε σήμερα, για την γρήγορη και, σχετικά με άλλα εργαλεία, εύκολη ανάπτυξη διαδικτυακών εφαρμογών. Η Node.js μάλιστα διαθέτει το δικό της package manager (NPM - Node Package Manager), στον οποίο διαρκώς προστίθενται καινούργιες βιβλιοθήκες από χρήστες για χρήστες.

Είναι σχεδιασμένο για να μπορεί να χτίζει κλιμακούμενες (scalable) εφαρμογές, καθώς η αρχιτεκτονική του, επιτρέπει τη σύνδεση πολλών εξωτερικών συστημάτων/χρηστών και την εξυπηρέτηση αυτών ταυτόχρονα. Κάθε φορά που γίνεται κάποια σύνδεση στην εφαρμογή και κατεπακόλουθω κάποιο αίτημα στο σύστημα εκτελείται μία callback συνάρτηση προκειμένου να μπορέσει να απαντήσει πίσω. Όσο το σύστημα δεν δέχεται αιτήματα "κοιμάται" και περιμένει το επόμενο που θα έρθει.

Όλα όσα προαναφέρθηκαν την καθιστούν κατάλληλη για τη δημιουργία συστημάτων server. Αξίζει να σημειωθεί μάλιστα ότι για το σκοπό αυτό υπάρχουν πλήθος βιβλιοθηκών που παρέχουν έτοιμες συναρτήσεις και διεπαφές που κάνουν την διαδικασία ανάπτυξης ακόμα πιο εύκολη και γρήγορη. Πολλές φορές μάλιστα, κάποιες βιβλιοθήκες πέρα από βοηθητικές συναρτήσεις και υπορουτίνες, επηρεάζουν τον τρόπο σύγγραφής κώδικα, μέσα από APIs που παρέχουν. Αυτές χαρακτηρίζονται ως frameworks, και επιταχύνουν τόσο τη διαδικασία ελέγχου (testing), όσο και τη διαδικασία ανάπτυξης (development) του λογισμικού. Μερικές από τα πιο γνωστά backend frameworks είναι τα: Express, Jest, Koa, Socket.io, Meteor, Loopback

Τα παραπάνω σε συνδυασμό με το ότι η JavaScript είναι μία ευρέως διαδεδομένη και πολυχρησιμοποιούμενη γλώσσα προγραμματισμού στο διαδίκτυο δίνει την ευκαιρία σε developers να αναπτύξουν μία εφαρμογή πλήρως (frontend και backend) με τη χρήση ενός κοινού εργαλείου. Ένα τυπικό παράδειγμα εφαρμογής που μπορεί να αναπτυχθεί με τον τρόπο αυτό φαίνεται στο [σχήμα 2.4](#).



Σχήμα 2.4: Σχεδίαση συστήματος με χρήση Node.js [9]

2.5.2 PM2

Η διαχείριση διεργασιών (processes) που τρέχουν σε ένα υπολογιστικό σύστημα μπορεί να είναι δύσκολη και αρκετά περίπλοκη για κάποιον που δεν διαθέτει μεγάλη πείρα σε αυτό τον τομέα. Το PM2 ή αλλιώς Process Manager 2, αποτελεί ένα open source πρότζεκτ που κάνει την παραπάνω διαδικασία πιο απλή και κατανοητή, κερδίζοντας χρόνο για τον developer που μπορεί να τον αξιοποιήσει για την ανάπτυξη εφαρμογών.

Με την έννοια διαχείριση διεργασιών, αναφερόμαστε σε όλες εκείνες τις ενέργειες που σχετίζονται με τον (πρόωρο ή όχι) τερματισμό και την παρακολούθηση ήδη υπάρχοντων διεργασιών (σχήμα 2.5), αλλά και τη δημιουργία καινούργιων. Προγράμματα όπως το pm2 προσφέρουν ακόμα δυνατότητες όπως είναι η αυτόματη επανεκκίνηση διεργασιών σε περίπτωση σφάλματος αποτρέποντας έτσι την ύπαρξη downtime των εφαρμογών που τρέχουμε.

Ένα από τα πιο χρήσιμα εργαλεία που παρέχει το PM2 είναι η λειτουργία cluster mode. Αν και δεν αναφέρθηκε πιο πάνω ο συγκεκριμένος διαχειριστής διεργασιών εξειδικεύεται σε Node.js εφαρμογές και στη διαχείριση αυτών. Εξ ορισμού εφαρμογές γραμμένες σε JavaScript τρέχουν σε ένα thread στο υπολογιστικό σύστημα που εκτελούνται. Χρησιμοποιώντας όμως το cluster mode, μπορούμε να ξεκινήσουμε πολλαπλές διεργασίες που θα λειτουργούν ταυτόχρονα και θα μοιράζουν το φόρτο κατάλληλα (load-balancing) ώστε το τελικό σύστημα να έχει καλύτερη απόδοση και να μπορεί να εξυπηρετεί περισσότερο κόσμο.

id	name	namespace	version	mode	pid	uptime	u	status	cpu	mem	user	watching
0	lychte-server	default	N/A	fork	10432	66s	0	online	0%	66.4mb	stavs	disabled
1	phiphus-worker	default	N/A	fork	33936	66s	0	online	0%	68.9mb	stavs	disabled
2	sisiphus-worker	default	N/A	fork	34312	66s	0	online	0%	68.7mb	stavs	disabled

Σχήμα 2.5: Παράδειγμα Παρακολούθησης διεργασιών με τη χρήση του PM2

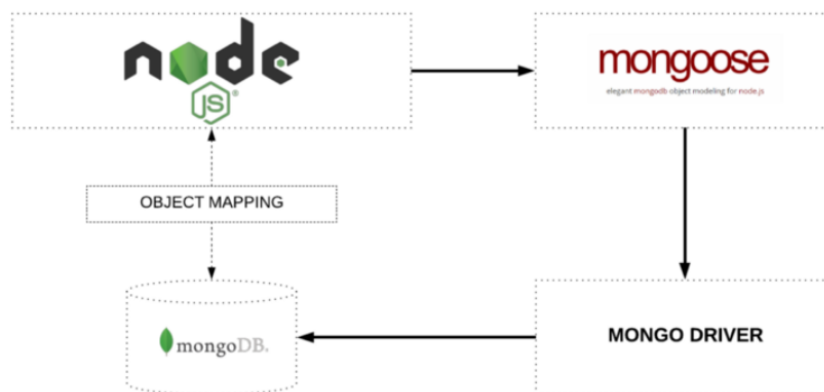
2.5.3 MongoDB

Η MongoDB αποτελεί μία Μη Σχεσιακή Βάση Δεδομένων (Non Relational Database), που χρησιμοποιεί documents, για να αποθηκεύσει πληροφορία, αντί στήλες και γραμμές όπως θα είχαμε σε κλασσικές SQL βάσεις δεδομένων. Είναι σχεδιασμένη για αποθήκευση δεδομένων μεγάλης κλίμακας και παράλληλη επεξεργασία δεδομένων μοιρασμένα σε έναν μεγάλο αριθμό από servers.

Τα documents που αναφέρθηκαν αποτελούν τον ακρογωνιαίο λίθο της Mongo, καθώς αποτελούν τη βασική μονάδα της αποθηκευμένης πληροφορίας. Μορφοποιούνται ως BSON (Binary JSON) και παρέχουν πληθώρα τύπων δεδομένων που μπορείς να αποθηκεύσεις (string, integer, double, boolean, array, object, date, timestamp, null, binary). Κάθε βάση mongo μπορεί να περιέχει μία ή περισσότερες συλλογές (collections), τα δεδομένα των οποίων πρέπει να υπακούουν στο ίδιο σχήμα (schema). Επειδή όμως η βάση αυτή παρέχει δυνατότητες δυναμικού σχήματος (dynamic schema) μπορούμε να αποθηκεύουμε πληροφορία και να προσθέτουμε πεδία (fields) που δεν είχαμε ορίσει από την αρχή δημιουργίας του συστήματος.

Μερικοί από τους λόγους που την επιλέξαμε είναι οι εξής:

- **Document Oriented Storage:** η αποθήκευση και διαχείριση των δεδομένων (στο πλαίσιο της εφαρμογής) είναι εύκολη καθώς στηρίζεται σε δεδομένα σε μορφή JSON.
- **Ευρετήρια (Indexes):** Μπορείς κατά τη διαδικασία του στησίματος της βάσης (αλλά και μετέπειτα) να ορίσεις ευρετήρια σε πεδία που χρησιμοποιούνται συχνά σε queries προκειμένου να βελτιώσεις την απόδοση του συστήματος στο σύνολο. Όσο πιο γρήγορη είναι βάση, τόσο καλύτερη θα είναι η ανταπόκριση του συστήματος.
- **Ομοιοτυπία (Replication) και μεγάλη Διαθεσιμότητα:** Δημιουργώντας πολλαπλά αντίτυπα των αποθηκευμένων δεδομένων σε περισσότερο από έναν servers, μπορούμε να είμαστε σίγουροι ότι θα έχουμε πρόσβαση στην πληροφορία ακόμα και αν η κίνηση (data traffic) της εφαρμογής αυξηθεί.
- **Αυτόματο sharding:** Διασπώντας την αποθηκευμένη πληροφορία και έχοντας τμήματα αυτής σε διαφορετικούς server μας δίνεται η δυνατότητα να κάνουμε οριζόντιο scaling πολύ πιο εύκολα.
- **Εύκολο ενσωμάτωση:** Η mongo διαθέτει βιβλιοθήκες στο περιβάλλον του node.js που καθιστούν τη χρήση και το στήσιμό της πολύ απλό. Πέρα από τη mongo, την επίσημη βιβλιοθήκη που υπάρχει, διατίθεται και η mongoose που αποτελεί μία Object Data Modelling (ODM) βιβλιοθήκη για τη Mongo και τη nodejs (σχήμα 2.6)



Σχήμα 2.6: Η mongoose λειτουργεί ως ένα abstract layer μεταξύ της Node και των driver της mongo προκειμένου η επικοινωνία μεταξύ των δύο να γίνεται πιο εύκολα

2.5.4 Google Cloud Storage

Πέρα από την κλασική βάση δεδομένων που προαναφέραμε θέλουμε να αποθηκεύουμε ιστορικά δεδομένα, δεδομένα δηλαδή που δεν θα δείχνουμε άμεσα στον χρήστη, αλλά θα κρατάμε για να υπολογίζουμε μετρικές και στατιστικά σημαντικά αποτελέσματα στο σύνολο όλης της μέχρι τώρα αποθηκευμένης πληροφορίας.

Σε αυτό λοιπόν το σημείο έρχεται το Google Cloud Storage (GCP), το οποίο παρέχει μεταξύ άλλων δυνατότητες Αποθήκευσης Αρχείων (File Storage). Τα δεδομένα αυτά δεν χρειάζεται να υπακούουν σε κάποιο σχήμα ενώ παράλληλα ο τρόπος αρχειοθέτησης των δεδομένων ταυτίζεται με ένα κλασικό σύστημα αρχείων ενός υπολογιστικού συστήματος. Διαθέτει paths και τύπους αρχείων όπως ακριβώς και οι υπολογιστές και όλες οι συσκευές που χρησιμοποιούμε. Για να διαβάσεις κανείς πληροφορία, χρειάζεται να ξέρει μόνο το μονοπάτι που οδηγεί στο επιθυμητό αρχείο καθώς και τη μορφή αυτού. Οι υποστηριζόμενοι τύποι αρχείων μέχρι τώρα είναι οι εξής:

- Binary
- Flat
- JSON
- Avro
- Parquet

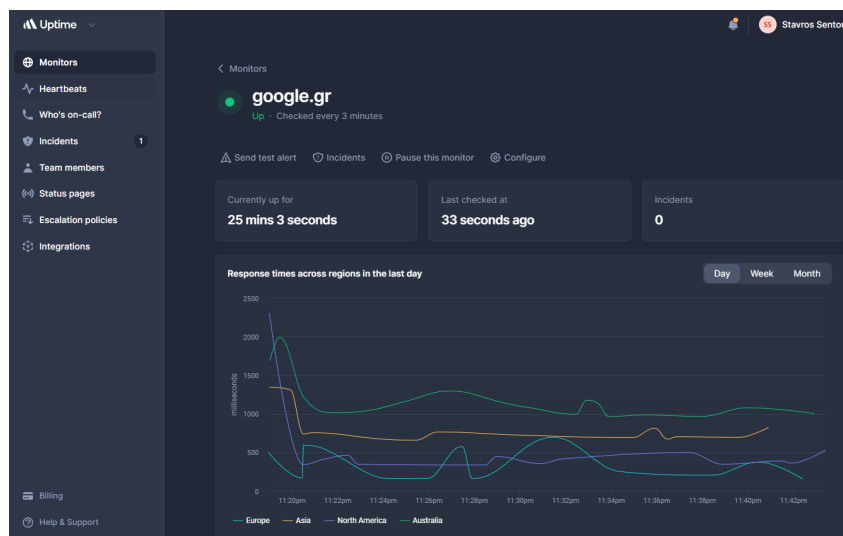
3

Βιβλιογραφική Αναζήτηση Τεχνολογιών Αιχμής

Πριν αναλυθεί η υλοποίηση του συστήματος που δημιουργήσαμε για την Ενεργή Παρακολούθηση Εφαρμογών ως Υπηρεσίες (SaaS) και ιστοσελιδών που εδρεύουν στο διαδίκτυο, θα παρουσιάσουμε τεχνολογίες που έχουν χτιστεί ήδη για το σκοπό αυτό και θα δείξουμε τους τομείς στους οποίους διαφέρει το δικό μας σύστημα.

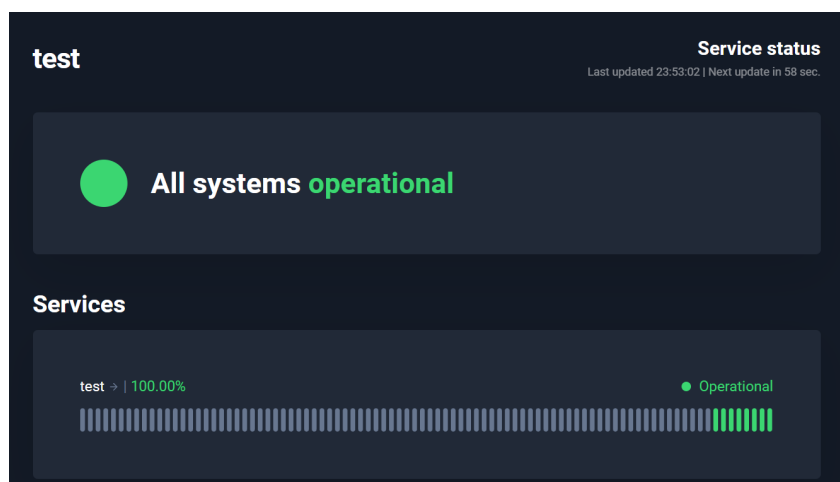
Εφαρμογές τέτοιου τύπου έχουν αναπτυχθεί κυρίως από εταιρίες αλλά υπάρχουν πολλά open source project μικρότερου βεληνεκούς που επιτυγχάνουν τον ίδιο στόχο. Στη συνέχεια θα αναφερθούμε κυρίως στα πιο γνωστά και διαδεδομένα εργαλεία παρουσιάζοντας τα δυνατά τους σημεία και περιγράφοντας τις λειτουργίες που παρέχουν.

- **Better Uptime:** Προσφέρει εύκολη ενσωμάτωση των ιστοσελιδών που θέλει κανείς να παρακολουθήσει. Λειτουργεί κάνοντας ping κάθε τριάντα δευτερόλεπτα στο url που ορίζει ο χρήστης και παρουσιάζει τα παραγόμενα δεδομένα σε ευπαρουσίαστα διαγράμματα (σχήμα 3.1). Ένα από τα μεγάλα πλεονεκτήματα που έχει αφορά τη δυνατότητα για πολλαπλά ping από διαφορετικές περιοχές του κόσμου (Ευρώπη, Ασία, Βόρεια Αμερική, Αυστραλία) ώστε οι χρήστες να διαθέτουν μία πιο πλήρη εποπτεία του υπό μελέτη συστήματος/ιστοσελίδας. Αξίζει να σημειωθεί ότι παρέχει και μηχανισμούς ενημέρωσης για να ειδοποιεί το χρήστη σε περίπτωση μη απόκρισης του συστήματος, μέσα από mail, εφαρμογές chatting και τηλεφωνικών κλήσεων.
- **Uptime Robot:** Επιτρέπει πέρα από την επιλογή του url και την επιλογή παραπάνων παραμέτρων που επηρεάζουν την απάντηση που θα επιστρέψει το υπό μελέτη σύστημα. Οι παράμετροι αυτοί σχετίζονται με τους headers του μηνύματος που αποστέλλεται και πιο συγκεκριμένα με αυτούς που αφορούν την αυθεντικοποίηση του χρήστη (στην προκειμένου περίπτωση του συστήματος παρακολούθησης). Πέρα από αυτά μπορεί να καθορίσει την επιθυμητή



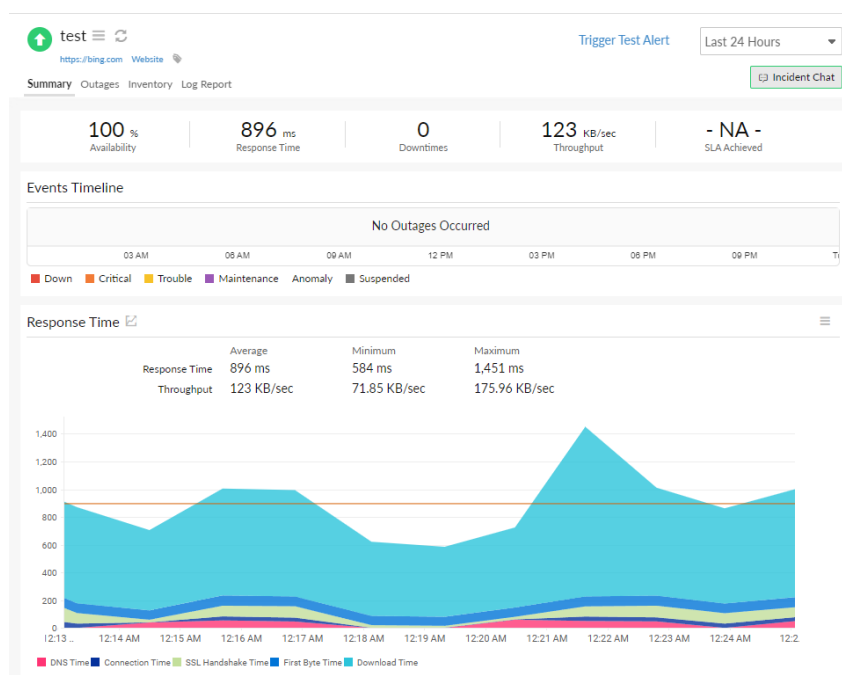
Σχήμα 3.1: Παράδειγμα χρήσης του εργαλείου Better Uptime

http κατάσταση της απόκρισης της ιστοσελίδας και το χρόνο που θα παρεμβάλλεται μεταξύ διαδοχικών αιτημάτων (pings). Τέλος διαθέτει κάποια βασικά διαγράμματα που σχετίζονται με το αν η απόκριση του συστήματος είναι ορθή ή όχι (σχήμα 3.2)



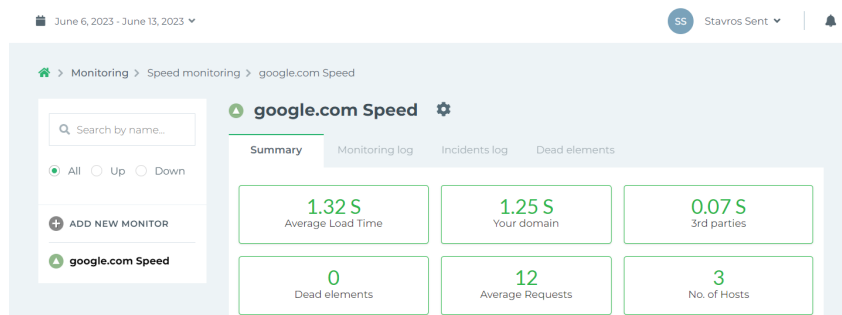
Σχήμα 3.2: Παράδειγμα χρήσης του εργαλείου Uptime Robot

- **Site24x7:** Διαθέτει μετρικές, που αφορούν τη μέγιστη, ελάχιστη τιμή του χρόνου απόκρισης του συστήματος, καθώς και τη μέση τιμή του, ενώ παράλληλα δίνει μία εικόνα του throughput του συστήματος. Δεν λείπει φυσικά και ένα διάγραμμα απόκρισης χρόνου (σχήμα 3.3) που καθιστά τα δεδομένα που συλλέγονται πιο εύκολα στην κατανόηση και οπτικοποίηση.
- **Uptimeia:** Πέρα από τα κλασικού τύπου http αιτήματα μπορεί να κάνει ελέγχους παρακολούθησης (uptime monitoring) σε DNS, UDP, TCP και email με απόσταση έως και τριάντα δευτερολέπτων (μεταξύ αιτημάτων). Αξίζει να



Σχήμα 3.3: Παράδειγμα χρήσης του εργαλείου Site24x7

σημειωθεί ότι η συγκεκριμένη εφαρμογή έχει δυνατότες και παθητικής παρακολούθησης (RUM). Αρχικά επιλέγουμε το site το οποίο θέλουμε να παρακολουθήσουμε καθώς και τα δεδομένα για τα οποία θέλουμε να ενημερωνόμαστε/παρακολουθούμε. Αυτά σχετίζονται κυρίως με σφάλματα ή καταστάσεις στις οποίες βρίσκεται το σύστημα και μπορεί να δηλώνουν κάποιο πρόβλημα. Καταστάσεις όπως η μειωμένη απόδοση του συστήματος εξαιτίας του χρόνου που παίρνει για να φορτώσει δεδομένα ή η απότομη πτώση του πλήθους των χρηστών μίας σελίδας. Για να επιτύχει τέτοιας μορφής ελέγχους παράγει (ανάλογα με τον τύπο των ελέγχων που επιλέγουμε) ένα script γραμμένο σε JavaScript που τοποθετείται στην αρχή της ιστοσελίδας την οποία θέλουμε να παρακολουθήσουμε. Με αυτό τον τρόπο έχουν τη δυνατότητα συλλογής δεδομένων πραγματικών χρηστών.

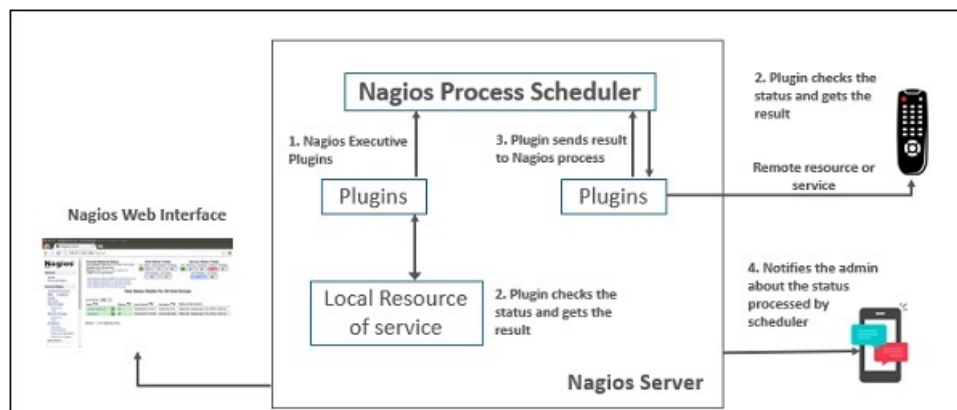


Σχήμα 3.4: Παράδειγμα χρήσης του εργαλείου Uptimeia

Παραπάνω αναφέρθηκαν μερικά μόνο κάποια από τα εργαλεία που υπάρχουν σήμερα για την Ενεργή Παρακολούθηση του Χρόνου και Απόκρισης Ιστοτόπων και Διαδικτυακών Εφαρμογών. Σε αυτά θα πρέπει να προστεθούν πληθώρα εφαρμογών όπως τα: **StatusCake**, **SemaText**, **Uptrends**, **Dotcom-monitor**, **Updown**, **Datadog** **Synthetics**. Οι δυνατότες που προσφέρουν ως επί το πλείστον μπορούν περιγραφούν πλήρως από όσα αναπτύξαμε προηγουμένως, για αυτό το λόγω δεν θα αναφερθούμε περαιτέρω.

Πρέπει σε αυτό το σημείο όμως να τονίσουμε ότι όσα προαναφέρθηκαν αποτελούν προϊόντα εταιριών. Αυτό όμως δεν σταματάει την ανάπτυξη open source projects που υλοποιήθηκαν από χρήστες είτε ως προσωπικά project, είτε ως project μίας μεγαλύτερης ομάδας από developers. Έτσι και στο πλαίσιο της Ενεργής Παρακολούθησης μερικά από τα πιο γνωστά και δημοφιλή μεταξύ developers αποθετήρια αποτελούν τα:

- **Uptime**: Αποτελεί μία ενδιαφέρουσα προσέγγιση στο πρόβλημα της διαχείρισης των schedulers που θα πρέπει να έχει το σύστημα για να κάνει αιτήματα ανά ένα συγκεκριμένο και σταθερό χρονικό διάστημα. Προκειμένου να επιτύχει κάτι τέτοιο αξιοποιεί τις δυνατότητες του GitHub (cloud-based υπηρεσία αποθήκευσης git αποθετηρίων) και των actions που αυτό σου επιτρέπει να εκτελείς κάθε πέντε λεπτά. Έτσι λοιπόν ανά πέντε λεπτά (ελάχιστος χρόνος ελέγχου απόκρισης) τρέχει αυτοματοποιημένα και μέσω του GitHub (ανεξάρτητα από το σύστημα αυτό) μία διαδικασία που κάνει αιτήματα στην σελίδα που ο χρήστης ορίζει. Φυσικά τα αποτελέσματα αυτά αποθηκεύονται και ανά έξι ώρες παράγονται διαγράμματα που μπορείς να τα δεις μέσα από μία σελίδα που παράγεται αυτόματα.

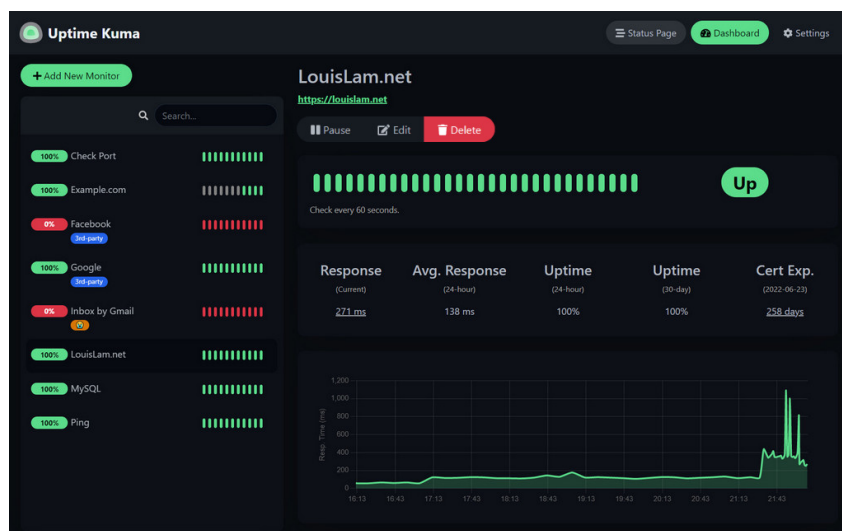


Σχήμα 3.5: Αρχιτεκτονική Συστήματος Nagios

- **Nagios**: Είναι ένα πρόγραμμα γραμμένο στη γλώσσα προγραμματισμού C. Πέρα από το backend, που θα δούμε στη συνέχεια διαθέτει και Γραφικό Περιβάλλον Χρήστη (Graphical User Interface - GUI). Αποτελείται από ένα σύστημα ενός server (nagios server), οποίος λειτουργεί σαν ένας scheduler που στέλνει σήματα για να ξεκινήσει την εκτέλεση plugins στα απομακρυσμένα συστήματα που θέλουμε να παρακολουθήσουμε. Μόλις τα plugins δεχτούν απάντηση επιστρέφουν στον server ο οποίος προωθεί την πληροφορία στο

GUI για να τη δούνε και οι χρήστες. Δεν χρησιμοποιείται κάποια βάση δεδομένων, καθώς η πληροφορία που μαζεύεται αποθηκεύεται μόνο σε logs στο σύστημα που τρέχει την υπηρεσία αυτή. Την αρχιτεκτονική σου συστήματος μπορούμε δούμε στην [σχήμα 3.5](#)

- **Kuma Uptime:** Αποτελεί μία εύκολη να χρησιμοποιήσεις και να στήσεις self-hosted εφαρμογή γραμμένη σε JavaScript για Ενεργή Παρακολούθηση. Για την αποστολή των αιτημάτων σε απομακρυσμένες (στο διαδίκτυο) σελίδες χρησιμοποιεί **child_processes**, ένα api δηλαδή που περιλαμβάνει η Node.js για τη δημιουργία διεργασιών (processes) εντός άλλων διεργασιών. Η εφαρμογή περιλαμβάνει backend και frontend, στο οποίο μπορείς να δεις τα αποτελέσματα του monitoring. Αξίζει να σημειωθεί ότι τα δεδομένα αποθηκεύονται σε βάση SQLite, ένα προσωρινό σύνολο δεδομένων που υφίσταται μόνο στο πλαίσιο εκτέλεσης μίας εφαρμογής. Αποτελεί μία server-less βάση δεδομένων και είναι άρρηκτα συνδεδεμένη με την εφαρμογή στην οποία υπάρχει.



Σχήμα 3.6: Παράδειγμα χρήσης του εργαλείου Kuma Uptime

Κάθε σύστημα που αναφέρθηκε έχει πλεονεκτήματα αλλά και μειονεκτήματα σε σχέση με άλλα. Αυτό που θέλαμε να διαθέτει ιδανικά ένα σύστημα Παρακολούθησης, βάση όλων αυτών που είδαμε μέχρι τώρα, είναι τα εξής:

1. Σταθερό και Αξιόπιστο σύστημα διαχείρισης προγραμματισμού (scheduling system) που θα καθορίζει το πότε πρέπει να ξεκινάνε τα αιτήματα προς τα εξωτερικά υπό παρακολούθηση συστήματα.
2. Κάποια μορφή βάσης δεδομένων στην οποία θα αποθηκεύουμε τα δεδομένα που συλλέγουμε.
3. Χρήση ιστορικών δεδομένων για τον υπολογισμό μετρικών στατιστικής φύσης σε βάθος χρόνου.

4. Προβολή των δεδομένων σε ευπαρουσίαστα και εύπεπτα διαγράμματα που θα βοηθούν το χρήστη να αντιλαμβάνεται γρήγορα την κατάσταση των συστημάτων του
5. Διάθεση τρόπων ειδοποίησης του χρήστη σε περίπτωση που κάποιο από τα συστήματα δεν ανταποκρίνεται
6. Δυνατότητα για οριζόντια κλιμάκωση (**horizontal scaling**)
7. Ελαχιστοποίηση downtime του συστήματος
8. Πλήρης παραμετροποίηση του μηνύματος που αποστέλλεται (κατάλληλη ρύθμιση headers, body και query)
9. Ρύθμιση του χρόνου μεταξύ διαδοχικών μηνυμάτων

Πολλές από τις εφαρμογές που είδαμε καλύπτουν σε κάποιο βαθμό μερικά από τα παραπάνω, αλλά καμία δεν τα καλύπτει όλα ταυτόχρονα. Τα εργαλεία που αναπτύχθηκαν στο πλαίσιο εταιριών έχουν δυνατότητες κλιμάκωσης αλλά στα περισσότερα αν όχι σε όλα μπορείς να δεις πληροφορία μέχρι ένα συγκεκριμένο χρονικό διάστημα πίσω στο χρόνο, κρύβοντας δεδομένα που είτε δεν αποθηκεύουν πλέον είτε δεν μπορούν να αντλήσουν αρκετά γρήγορα. Από την άλλη οι περισσότερες open source εφαρμογές που αναφέραμε δεν μπορούν να κάνουν τόσο εύκολα scaling καθώς είναι φτιαγμένες να δουλεύουν για περιορισμένο πλήθος χρηστών.

Πίνακας 3.1: Σύγκριση Εργαλείων Ενεργής Παρακολούθησης

	Υλοποίηση μας	Better Uptime	Uptime Robot	Site24x7	Uptimeia	Kuma
Σταθερότητα	✓	✓	✓	✓	✓	✓
Βάση Δεδομένων	✓	✓	✓	✓	✓	
Ιστορικά Δεδομένα	✓					
Διαγράμματα	✓	✓		✓	✓	
Ειδοποιήσεις	✓	✓	✓	✓	✓	✓
Ελαχιστοποίηση Downtime	✓	✓	✓	✓	✓	✓
Παραμετροποίηση μηνυμάτων	✓	✓	✓	✓	✓	
Ρύθμιση Χρόνου μεταξύ μηνυμάτων	✓	✓	✓	✓	✓	✓
Οριζόντια Κλιμάκωση	✓	✓	✓	✓	✓	

Στη διπλωματική αυτή εργασία λοιπόν καλούμαστε να δημιουργήσουμε ένα τέτοιο υπολογιστικό σύστημα που θα καλύπτει τα προαναφερθέντα.

4

Υλοποιήσεις

Στο κεφάλαιο αυτό θα αναφερθούμε στις τρεις διαφορετικές εκδόσεις του συστήματος που υλοποιήσαμε μέχρι να καταλήξουμε στην τωρινή και τους λόγους που μας ωθούσαν να κάνουμε αυτές τις αλλαγές.

Πριν ξεκινήσουμε όμως θα πρέπει να ορίσουμε τις βασικές ενέργειες που θα εκτελεί ένα τέτοιο σύστημα. Αρχικά θέλουμε να έχει δυνατότητες σέρβερ, ώστε να εξυπηρετεί απομακρυσμένους χρήστες που συνδέονται μέσω του δικτύου. Το σύστημα επιπροσθέτως θα πρέπει να μπορεί να στέλνει αιτήματα σε εξωτερικούς σέρβερ, ιστοσελίδες και γενικά εφαρμογές που επικοινωνούν με το διαδίκτυο. Σε αυτό το σημείο τονίζουμε την αναγκαιότητα ύπαρξης μίας βάσης δεδομένων που θα αποθηκεύει πληρογορία σχετική με την ορθή λειτουργία του συστήματος. Τέλος και ίσως το πιο σημαντικό, απαιτείται η ύπαρξη κάποιας μορφής scheduling, ενός μηχανισμού δηλαδή που θα ρυθμίζει πότε θα γίνονται τα αιτήματα και ποια από αυτά πρέπει γίνουν.

Όλα αυτά αφορούν αποκλειστικά τη λειτουργία του backend του συστήματος. Πέρα από αυτά θα πρέπει να υπάρχει μία γραφική διεπαφή μέσω της οποίας κάθε χρήστης θα μπορεί να δει τα δεδομένα που παράγονται από τη χρήση του συστήματός.

Πλέον και χάριν συντομίας, στο υπόλοιπο κομμάτι της διπλωματικής αυτής εργασίας, θα αναφερόμαστε στο σύστημα που δημιουργήσαμε, με το όνομα **Lychte** (/licht/). Η ονομασία αυτή προκύπτει από τα αρχικά του "Lightweight Yet Configurable HTTP Traffic Expert", που περιγράφει την λειτουργία και μόνο μερικά από τα πολλά χαρακτηριστικά του συστήματός μας.

4.1 VERSION 0.0.1

Η πρώτη και πιο απλή έκδοση του υπό κατασκευής συστήματος Ενεργής Παρακολούθησης. Αποτελείται από έναν nodejs σέρβερ (και πιο συγκεκριμένα express

σέρβερ), ο οποίος είναι υπεύθυνος για τον χειρισμό όλων των λειτουργιών της εφαρμογής. Αυτός συνδέεται άμεσα με μία MongoDB που θα αποτελέσει τη βάση δεδομένων του συστήματος.

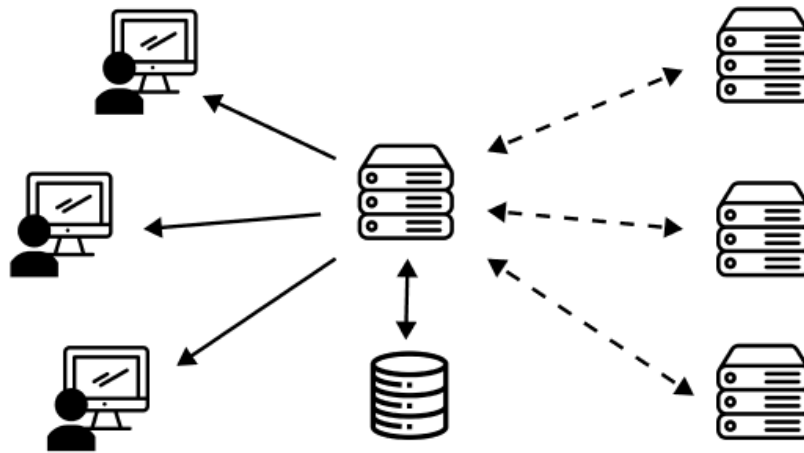
Ο σέρβερ αρχικά στεγάζει το api της εφαρμογής. Μέσα από αυτό κάθε χρήστης που έχει πρόσβαση στο σύστημα (σωστά credentials) μπορεί να αντλεί πληροφορία ήδη αποθηκευμένη στη βάση ή να την τροποποιεί και να δημιουργεί νέα. Τα δεδομένα που μπορεί να παράξει είναι περιορισμένα, βέβαια, καθώς υπάρχουν collections στα οποία γράφει μόνο το backend. Για να συνεχίσουμε και να δούμε το υπόλοιπο σύστημα θα πρέπει να αναφερθούμε στο pipeline της εφαρμογής, στα βήματα δηλαδή που θα ακολουθήσει ο χρήστης εντός αυτής.

Κύρια λειτουργία του Lychte είναι η αυτόματη παρακολούθηση του χρόνου λειτουργίας και απόκρισης ενός ιστοτόπου. Για να γίνει αυτό ο χρήστης θα πρέπει να εισάγει το url, που επιθυμεί να επιβλέπει και να ενημερώνεται σχετικά με τη κατάστασή του, και ένα χρονικό διάστημα βάσει του οποίου θα γίνονται επαναλαμβανόμενα αιτήματα. Έπειτα η πληροφορία αυτή μεταφέρεται στον σέρβερ υπό τη μορφή HTTP request. Αυτός το μήνυμα που περιέχει το url και όποια ακόμα πληροφορία απαιτείται (headers, body) για να μπορέσει να κάνει την κλήση στο εξωτερικό σύστημα που ο χρήστης ορίζει. Τέλος αποθηκεύει το αίτημα στη βάση δεδομένων και ξεκινάει έναν απλό scheduler ο οποίος επαναλαμβάνει το αίτημα στο url που ορίστηκε από το χρήστη ανάλογα με τον διάστημα που επιλέχθηκε.

Καταλήγουμε έτσι στη δεύτερη, μεγάλη, λειτουργία του σέρβερ, το scheduling. Κάθε φορά που αποθηκεύεται ένα καινούργιο αίτημα για παρακολούθηση, ξεκινάει μία νέα διεργασία εντός αυτής που ήδη τρέχει (αυτής του σέρβερ). Η διεργασία παιδί (child process), λαμβάνει κάποια πληροφορία σχετικά με το αίτημα που θα πρέπει να εκτελεί, από την διεργασία που την δημιούργησε. Η τελευταία μάλιστα έχει τη δυνατότητα να σταματήσει τη λειτουργία όσων διεργασιών έχει ξεκινήσει αρκεί να γνωρίζει το pid που χαρακτηρίζει κάθε μία μονοσήμαντα.

Όλες οι διεργασίες εκτελούν την ίδια συνάρτηση τα ορίσματα της οποίας αποτελούν τα χαρακτηριστικά που ορίζει ο χρήστης κατά τη διαδικασία δημιουργίας του αιτήματος προς παρακολούθηση. Η συνάρτηση αυτή είναι υπεύθυνη για να κάνει την κλήση στο εξωτερικό σύστημα, να λάβει την απάντηση, να την αναλύσει (ανάλογα με το είδος της αναμενόμενης απάντησης) και να την αποθηκεύσει, μαζί με κάποιες ακόμα χρήσιμες πληροφορίες, στη βάση δεδομένων. Πέρα από το response που μπορεί να είναι απλό κείμενο (html), αντικείμενο (json) ή κάποιο αρχείο αποθηκεύονται και οι χρονισμοί του αιτήματος, το πότε δηλαδή ξεκίνησε το αίτημα, πότε τελείωσε καθώς και το χρόνο που μεσολάβησε μεταξύ τους. Το τελευταίο θα μπορούσε να υπολογιστεί και εκ των υστέρων σε κάθε αίτημα που το χρειάζεται, καθώς είναι δεδομένο που προκύπτει από αυτά που ήδη υπάρχουν. Εξαιτίας όμως της φύσης του συστήματος και των δεδομένων που μαζεύονται θεωρούμε ότι κρίνεται αναγκαίο να κερδίσουμε επεξεργαστική ισχύ αποφεύγοντας υπολογισμούς για δεδομένα που γνωρίζουμε ότι θα ζητούνται συνεχώς, ώστε να εξηπυρετούνται όσο το δυνατό πιο γρήγορα και άμεσα τα αιτήματα του χρήστη ως προς τον σέρβερ.

Ένα τέτοιο σύστημα μπορεί να λειτουργήσει όπως ακριβώς περιγράψαμε παραπάνω. Έχει όμως κάποια βασικά μειονεκτήματα. Αρχικά δεν υπάρχει κάποιο κοινό σημείο αναφοράς (single point of reference). Όσες διεργασίες υπάρχουν μέσα στο σέρβερ κοινούνται αυτόνομα, χωρίς να υπάρχει κάποιος τρόπος να αναφερθείς σε



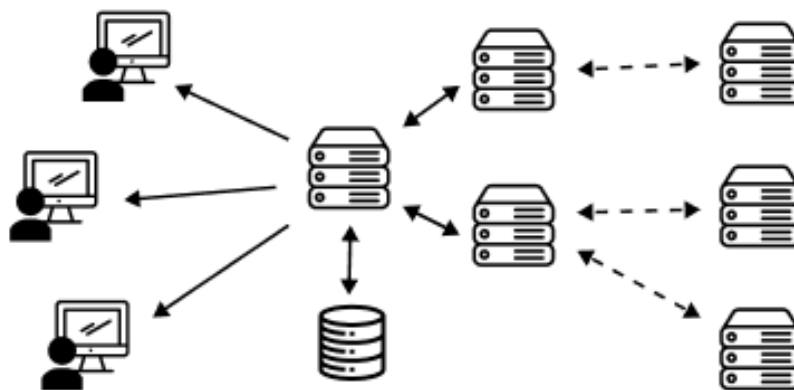
Σχήμα 4.1: Διάγραμμα Πρώτης Υλοποίησης Συστήματος Ενεργής Παρακολούθησης. Ένας express σέρβερ, με μία mongo βάση δεδομένων εξυπηρετούν χρήστες, ενώ παράλληλα στέλνουν αιτήματα σε εξωτερικά συστήματα προς παρακολούθηση

κάποια αν δεν γνωρίζεις εκ των προτέρων το pid της και αν δεν έχεις πρόσβαση στο σύστημα που τις εκτελεί. Επίσης το χαρακτηριστικό αυτό των διεργασιών έχει υπόσταση μόνο εντός του ιδίου περιβάλλοντος εκτέλεσης. Ακόμα και να αποθηκεύαμε το pid της διεργασίας που είναι υπεύθυνο για την εκτέλεση ενός συγκεκριμένου αιτήματος, δεν θα μπορούσαμε να το χρησιμοποιήσουμε κάπως σε περίπτωση που επανεκκινηθεί το υπολογιστικό σύστημα που φιλοξενεί τον σέρβερ, καθώς οι καινούργιες διεργασίες θα είχαν διαφορετικό id. Οι "καινούργιες διεργασίες" εδώ χαρακτηρίζουν ένα σύνολο διεργασιών που θα ξεκινούσε ο σέρβερ μέχρι όλα τα αιτήματα που βρίσκονται αποθηκευμένα στη βάση να ξεκινήσουν να ελέγχονται από κάποιο process.

Αξίζει ακόμα να σημειωθεί ότι το σύστημα αυτό δεν μπορεί να κλιμακωθεί εύκολα, και αποδοτικά, καθώς για να επεκτείνεις το σύνολο των αιτημάτων που μπορείς να διαχειριστείς θα πρέπει να συντηρείς έναν δεύτερο σέρβερ που ίσως να μην χρειάζεται, ώστε να χωρέσουν παραπάνω διεργασίες. Το πρόβλημα αυτό όμως μπορεί να λυθεί σπάζοντας τις δύο βασικές λειτουργίες του Lychte σε δύο διακριτές και σαφώς ανεξάρτητες οντότητες. Ενός scheduler και ενός σέρβερ. Αυτό ακριβώς θα δούμε καινούργιες στη δεύτερη υλοποίηση του συστήματός μας.

4.2 VERSION 0.1.0

Συνεχίζοντας τη συλλογιστική πορεία της πρώτης υλοποίησης, καλούμαστε να λύσουμε ένα από τα πιο βασικά προβλήματα, που προέκυψαν. Ο διαμοιρασμός των λειτουργιών. Όπως προαναφέραμε θα πρέπει να υπάρχει ένας (ή και παραπάνω) σέρβερ που θα εξυπηρετεί τους χρήστες μέσω διεπαφών των προγραμματιστικών διαδικασιών που παρέχει, και ένας (ή παραπάνω) scheduler, που θα εκτελούν αποκλειστικά και μόνο αιτήματα προς τα συστήματα που θέλουμε να ελέγχουμε. Ας δούμε όμως πιο συγκεκριμένα το ανανεωμένο σύστημα ([σχήμα 4.2](#))



Σχήμα 4.2: Διάγραμμα Δεύτερης Υλοποίησης Συστήματος Ενεργής Παρακολούθησης. Αποτελείται στο κέντρο του από έναν express σέρβερ και μία mongo βάση δεδομένων. Ο σέρβερ επικοινωνεί με δύο schedulers οι οποίοι είναι υπεύθυνοι για τον έλεγχο εξωτερικών προς το σύστημα εφαρμογών/σέρβερ

Το βασικό συστατικό του Lychte, ο σέρβερ δηλαδή παραμένει όπως είχε. Έχει δηλαδή routes μέσα από τα οποία εξυπηρετεί αιτήματα των χρηστών που προέρχονται από το frontend του συστήματος. Αυτά αφορούν την αποστολή πληροφορίας προς τους χρήστες προκειμένου να δουν τα δεδομένα που αποθηκεύουμε στη βάση μας, αλλά και την αποθήκευση πληροφορίας που προέρχεται από τους χρήστες που σχετίζεται με τη δημιουργία καινούργιων ελέγχων σε urls που επιθυμούν να παρακολουθούν.

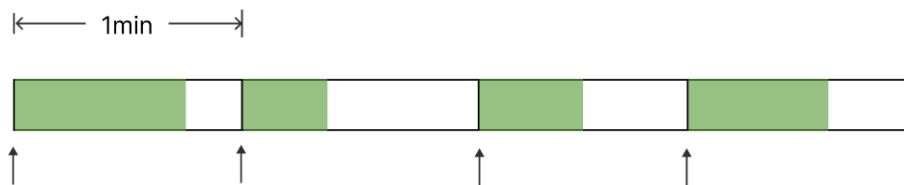
Αυτό που αλλάζει σε σχέση με την προηγούμενη υλοποίηση, είναι ο τρόπος που διευθετεί την εκτέλεση των επαναλαμβανόμενων αιτημάτων που πρέπει να κάνει προς τα εξωτερικά υπό παρακολούθηση συστήματα. Όπως είπαμε νωρίτερα δεν είναι πλέον υπεύθυνος για το scheduling των ενεργειών που πρέπει να γίνουν, αλλά αντιθέτως στέλνει σε ένα άλλο σύστημα (scheduler) τις πληροφορίες που χρειάζεται προκειμένου να ρυθμιστεί και να οργανωθεί κατάλληλα ο έλεγχος των urls που θέλουμε να ελέγχονται.

Πλέον θα αναφερόμαστε στο σύστημα του scheduler ως worker, καθώς αυτό που κάνει είναι να λαμβάνει δεδομένα και να εκτελεί, βάση αυτών, συνεχώς αιτήματα προς άλλα εξωτερικά συστήματα. Ο τρόπος λειτουργίας του είναι αρκετά

παρόμοιος με τη αυτή του scheduler της πρώτης υλοποίησης, με κάποιες μικρές διαφορές. Κάθε φορά που λαμβάνει κάποιο αίτημα από τον κεντρικό σέρβερ, κάνει ένα δοκιμαστικό request στο url που πρόκειται να ελεγχθεί, για να διαπιστωθεί η ορθότητά του. Έπειτα ξεκινάει έναν timer που εκτελεί το αίτημα ανά χρονικό διάστημα που ορίζει ο χρήστης. Ο χρόνος αυτός μπορεί να φτάνει ένα κάτω όριο του ενός δευτερολέπτου, αλλά για λόγους σταθερότητας και εγγύησης ότι η εκτέλεση κάποιου αιτήματος δεν θα μπλοκάρει την εκτέλεση κάποιου άλλου, έχουμε βάλει ένα κάτω όριο των 30 δευτερολέπτων στην υλοποίησή μας. Στην περίπτωση που το αίτημα χρειαστεί χρόνο, για να ικανοποιηθεί, μεγαλύτερο αυτού που έχει οριστεί σαν χρόνος μεταξύ διαδοχικών αιτημάτων διακόπτουμε το αίτημα μέσω timeouts και το αποθηκεύουμε στη βάση σαν αποτυχημένη απόκριση (failed response). Αυτό γίνεται για να εξασφαλίσουμε ότι το χρονικό διάστημα μεταξύ διαδοχικών αιτημάτων θα παραμένει σχετικά σταθερό και η απόλυτη μετατόπιση του χρόνου που υπάρχει μεταξύ των αιτημάτων σε βάθος χρόνου, δεν θα αλλάζει σε μεγάλο βαθμό.

Για να γίνει πιο κατανοητό το παραπάνω αρκεί να μελετήσουμε τα [σχήμα 4.3](#) και [σχήμα 4.4](#)

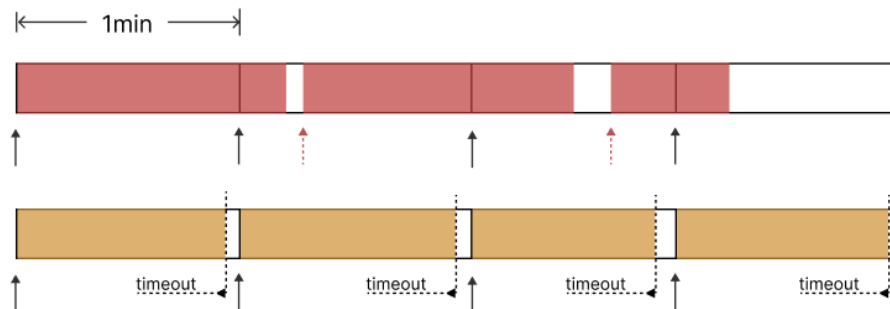
Το πρώτο περιγράφει την εκτέλεση ενός αιτήματος ανά ένα χρονικό διάστημα ενός λεπτού. Όπως παρατηρούμε το αίτημα εκτελείται επιτυχώς σε αυτό το μήκος χρόνου με αποτέλεσμα, όταν έρθει η στιγμή που πρέπει να επαναεκτελεστεί, αυτό να γίνει χωρίς περιττές καθυστερήσεις. Αν υποθέσουμε ότι ο χρόνος που μεσολάβησε μεταξύ του πρώτου και του δεύτερου αιτήματος, είναι ακριβώς εξήντα δευτερόλεπτα, μπορούμε να πούμε ότι η απόκλιση, σε σχέση με τον ένα λεπτό, μεταξύ αυτών είναι μηδενική. Βέβαια κάτι τέτοιο δεν υφίσταται στην πραγματικότητα καθώς οι timers έχουν πάντα μία μικρή απόκλιση, προς τα πάνω ή προς τα κάτω, της τάξης των nanoseconds. Σε βάθος χρόνου η απόκλιση παραμένει κοντά στο μηδέν.



Σχήμα 4.3: Κύκλος Ζωής εκτέλεσης ενός επαναλαμβανόμενου Αιτήματος

Έπειτα ας μελετήσουμε την περίπτωση που το αίτημα χρειάζεται παραπάνω χρόνο για να εκτελεστεί (εξαιτίας καθυστερήσεων του εξωτερικού συστήματος που ελέγχεται). Στο [σχήμα 4.4](#) μπορούμε να δούμε ακριβώς πως θα λειτουργούσε ένας worker στην περίπτωση που υπήρχαν μεγάλες εξωγενείς καθυστερήσεις, χωρίς τη χρήση timeouts αλλά και με τη χρήση αυτών. Όταν αναφερόμαστε σε timeouts περιγράφουμε την πρόωρη έξοδο από τη συνάρτηση που τρέχουμε, στην προκειμένη, την εκτέλεση του αιτήματος. Παρατηρούμε ότι στα δύο χρονικά διαγράμματα έχουμε αρκετά μεγάλες διαφορές. Στο πρώτο περιμένουμε να εκτελεστεί πλήρως το αίτημα, ακόμα και αν αυτό αργήσει παραπάνω από όσο θα έπρεπε. Το επόμενο αίτημα ξεκινάει με κάποια καθυστέρηση ως προς το πρώτο και παρομοίως τα επό-

μενα στη σειρά. Αυτή η διαδικασία εισάγει απόκλιση στους χρόνους των κλήσεων μεταξύ των αιτημάτων που υπό φυσιολογικές συνθήκες αναμένουμε να ισαπέχουν (χρονικά) μεταξύ τους. Απόκλιση η οποία μάλιστα μπορεί συνεχώς να αυξάνεται, καθιστώντας το σύστημά μας αναξιόπιστο ως προς το χρόνο εκτέλεσης και την ακρίβεια των αιτημάτων. Για αυτό το λόγο κρίνουμε αναγκαία τη χρήση timeouts. Το μεινέκτημα σε αυτή την περίπτωση είναι ότι αποθηκεύουμε ως εσφαλμένο ένα πιθανώς σωστό response, λόγω του χρόνου που πήρε για να εκτελεστεί.



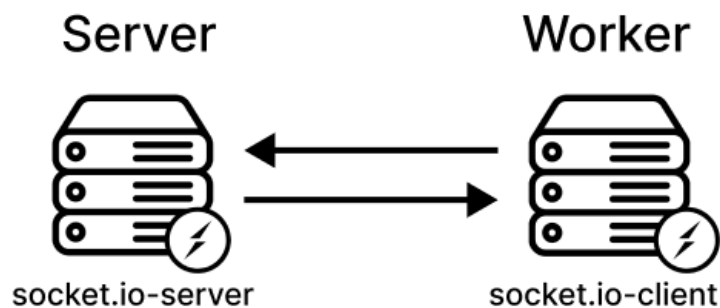
Σχήμα 4.4: Σύγκριση χρήσης ή μη timeouts στον κύκλο ζωής εκτέλεσης ενός αργού επαναλαμβανόμενου αιτήματος

Μία ακόμα αλλαγή ως προς την πρώτη υλοποίηση αφορά την επικοινωνία μεταξύ σέρβερ και worker. Προκειμένου το σύστημα να μπορεί να σπάσει σε πιο διακριτές έννοιες χάνουμε το πλεονέκτημα του να έχουμε όλες τις λειτουργίες εντός του ίδιου συστήματος. Λειτουργίες οι οποίες πλέον για να επιτελούνται πρέπει να υπάρχει ένα τρόπος επικοινωνίας μεταξύ των εμπλεκόμενων. Η επικοινωνία αυτή δεν θα είναι απαραίτητα μονόπλευρη. Η βασική λειτουργία όπως είχαμε δει και από την προηγούμενη υλοποίηση είναι η αποστολή μηνυμάτων στον scheduler, σε αυτή την περίπτωση στον worker. Επεκτείνοντας όμως τις δυνατότητες του συστήματος θα μπορούσαμε να έχουμε αμφίδρομη επικοινωνία μεταξύ σέρβερ και worker, ώστε να μπορεί να ενημερώνεται ο σέρβερ (όταν το ζητήσει) για την κατάσταση των διαφόρων worker που είναι συνδεδεμένοι σε αυτόν. Κάτι τέτοιο θα είχε νόημα αν είχαμε παραπάνω από έναν worker και θα θέλαμε να κάνουμε διαμοιρασμό φόρτου σε καθέναν από αυτούς. Αρχικά θα ρωτούσε όλους τους συνδεδεμένους worker σχετικά με το πλήθος των αιτημάτων που ελέγχουν τη δεδομένη χρονική στιγμή. Έπειτα κάθε κόμβος θα απαντούσε και αυτός με το λιγότερο φόρτο θα επιλεγόταν ως ο κατάλληλος για τη δημιουργία μίας ακόμα διεργασίας ελέγχου ενός url.

Λόγω της αμφίδρομης σχέσης που έχουν τα δύο συστήματα μεταξύ τους, επιλέχθηκε το πρωτόκολλο επικοινωνίας των WebSockets. Πιο συγκεκριμένα ο τρόπος ενσωμάτωσης της τεχνολογίας αυτής πραγματοποιήθηκε με τη χρήση της βιβλιοθήκης **socket.io** που αποτελεί ένα abstract layer πάνω από το πρωτόκολλο WebSockets. Για να υπάρξει επικοινωνία θα πρέπει να υπάρχει από τη μία πλευρά ένας **socket.io-server** και από την άλλη, ένας **socket.io-client**. Στη δική μας εφαρμογή ο **socket.io-server** θα είναι ο σέρβερ και ο **socket.io-client** ο/οι worker. Κάθε φορά που θα ξεκινάει τη λειτουργία του ένας σέρβερ θα δημιουργεί στο τοπικό δίκτυο του έναν **socket server** στον οποίο μπορούν να συνδεθούν και να επικοινωνήσουν όσοι διαθέτουν το url του και τα απαραίτητα credentials. Από την άλλη κάθε φορά που

Ξεκινάει τη λειτουργία του ένας worker θα πρέπει να μπορεί να συνδέεται κατευθείαν στο socket server που υπάρχει. Θα πρέπει δηλαδή να διαθέτει από πριν το url που οδηγεί στον socket server.

Οι λόγοι που επιλέχθηκαν τα WebSockets αντί ενός κλασσικού HTTP σέρβερ είναι κυρίως οι ταχύτητες που προσφέρουν. Υπό περιπτώσεις μπορεί να είναι μέχρι και δέκα φορές πιο γρήγορα σε σχέση με το συμβατικό πρωτόκολλο HTTP. Επίσης από τη στιγμή που κάθε worker θα επικοινωνεί αποκλειστικά με έναν (ή περισσότερους) σέρβερς του δικού μας κλειστού συστήματος βλέπουμε ότι δεν έχει νόημα η δημιουργία ολόκληρου HTTP server στο πλαίσιο λειτουργίας ενός worker.



Σχήμα 4.5: Επικοινωνία σέρβερ-worker μέσω websockets

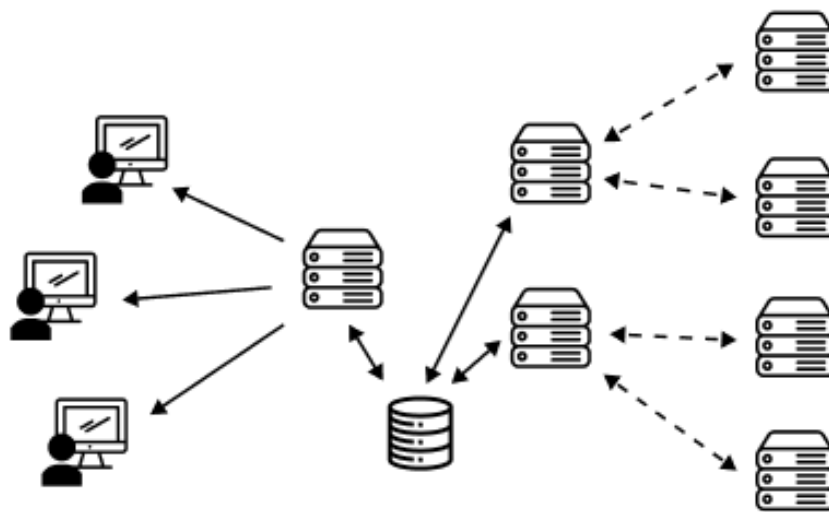
Η παραπάνω υλοποίηση έχει πολλά πλεονεκτήματα σε σχέση με την πρώτη. Αρχικά με τον καταμερισμό των δύο βασικών λειτουργιών κερδίζουμε όσων αφορά την πολυπλοκότητα του συστήματος, κάτι το οποίο θα πρέπει να μας απασχολεί, καθώς όσο πιο πολύπλοκο είναι ένα σύστημα τόσο πιο δύσκολα συντηρείται στη συνέχεια. Πέρα από αυτό, που αφορά κυρίως τη διαδικασία ανάπτυξης λογισμικού και δεν επιφέρει κάποιο άμεσο κέρδος στους χρήστες, θα πρέπει να αναφέρουμε ότι το σύστημα γενικά είναι πιο αποδοτικό, καθώς οι διεργασίες που κάθε worker στεγάζει είναι λιγότερο πιθανό να κολλήσουν και να μπλοκάρουν λόγω φόρτου στο μηχανήμα που εκτελούνται. Τα requests γενικά δεν καταναλώνουν πολλούς πόρους από το σύστημα στο οποίο εκτελούνται, σε αντίθεση με έναν server που μπορεί να δέχεται συνεχώς αιτήματα. Αξίζει να σημειωθεί ότι το api που παρέχει ο σέρβερ του συστήματός μας είναι υπεύθυνο για την αποστολή μεγάλου όγκου πληροφορίας, διαρκώς, που χρησιμοποιείται σε πληθώρα διαγραμμάτων, καθιστώντας τον έτσι απαιτητικό ως προς τους πόρους του υπολογιστικού συστήματος.

Ένας ακόμα τομέας στον οποίο συνεισφέρει σε μεγάλο βαθμό η διάσπαση των λειτουργιών αφορά την κλιμάκωση, και βασικά την οριζόντιο κλιμάκωση του συστήματος. Αν δούμε ότι ο σέρβερ υστερεί και δυσκολεύεται να ανταπεξέλθει στα αιτήματα των χρηστών, μπορούμε να φτιάξουμε έναν καινούργιο σέρβερ. Από την άλλη, αν παρατηρήσουμε ότι οι worker αργούν ή έχουν υπερφορτωθεί μπορούμε να δημιουργήσουμε workers και να διαμοιράσουμε τα αιτήματα εξίσου στο πλήθος των ενεργών scheduler που διαθέτουμε.

Ακόμα και με αυτή την υλοποίηση όμως υπάρχει έλλειψη ενός κοινού σημείου αναφοράς. Ενός κόμβου στο σύστημά μας του οποίου οι αλλαγές θα επηρέαζαν άμεσα τη λειτουργία όλων των οντοτήτων αυτού.

4.3 VERSION 1.0.0

Τέλος φτάνουμε στην τωρινή υλοποίησή μας, στην τελική πλέον έκδοση του Lychte. Πολλές ιδέες από προηγούμενες υλοποιήσεις παρέμειναν, ελαφρώς πάντοτε τροποποιημένες με σκοπό τη βελτίωση του συστήματος. Στο [σχήμα 4.6](#) μπορούμε να δούμε με μία πρώτη ματιά το διάγραμμα του συστήματος



Σχήμα 4.6: Ο σέρβερ και κάθε worker του συστήματος συνδέονται με τη βάση χωρίς να υπάρχει άμεση επικοινωνία μεταξύ τους. Ο σέρβερ εξυπηρετεί αποκλειστικά χρήστες, ενώ οι workers είναι υπεύθυνοι για την αποστολή αιτημάτων στα εξωτερικά συστήματα που θέλουμε να ελέγχουμε

Το τελικό σύστημα μοιάζει σε μεγάλο βαθμό με την προηγούμενη υλοποίηση. Η αρχιτεκτονική του σέρβερ είναι σχεδόν ίδια χωρίς το κομμάτι των websockets. Πλέον δεν υπάρχει άμεση επικοινωνία μεταξύ σέρβερ και workers. Αν κάποια από τις δύο οντότητες πρέπει να μεταφέρει πληροφορία στην άλλη, θα το κάνει γράφοντας σε κατάλληλα πεδία και συλλογές αντικειμένων ή διαγράφοντας πληροφορία εντός της βάσης δεδομένων. Ο τύπος της βάσης εξακολουθεί να είναι noSQL και συγκεκριμένα MongoDB, λόγω της ικανότητάς της να αποθηκεύει μεγάλο πλήθος δεδομένων. Δεδομένων μάλιστα που δεν χρειάζεται να ακολουθούν ένα συγκεκριμένο τύπο σχήματος (schema). Πέρα από αυτά οι δυνατότητες που παρέχει για sharding καθώς και για εύκολη και γρήγορη κλιμάκωση την καθιστούν ιδανική για το σύστημα μας, που συνεχώς αποθηκεύει νέα πληροφορία στη βάση.

Πριν συνεχίσουμε στην επεξήγηση της ακριβούς λειτουργίας του ανανεωμένου συστήματος, θα πρέπει να αναφερθούμε στους workers/schedulers. Η λειτουργία τους έχει εξελιχθεί σε σχέση με τις προηγούμενες εκδόσεις. Έχοντας πλέον ένα

κοινό σημείο αναφοράς, τη Mongo βάση, μπορούμε να φτιάξουμε πιο "έξυπνους" schedulers. Στις άλλες υλοποιήσεις μας κάθε φορά που ερχόταν ένα νέο αίτημα (ένα νέο url που θα πρέπει να ελέγχουμε δηλαδή), η διαδικασία που ακολουθούσαμε ήταν απλή. Ξεκινούσαμε ένα child process που εκτελούσε συνέχεια, ανά χρονικά διαστήματα που όριζε ο χρήστης, ένα αίτημα σε κάποιο εξωτερικό σύστημα. Όταν ο χρήστης επέλεγε να σταματήσει να παρακολουθεί το συγκεκριμένο url, τερματίζαμε το process που έτρεχε τον έλεγχο, καθώς γνωρίζαμε από πριν το pid της διεργασίας. Η παραπάνω διαδικασία δουλεύει καλά σε μικρό πλήθος αιτημάτων. Το πρόβλημα που δημιουργείται είναι ότι φτάνει γρήγορα σε σημείο κορεσμού, στο σημείο εκείνο που οι αποκλίσεις από τους αναμενόμενους χρόνους μεταξύ των διαδοχικών αιτημάτων αυξάνεται συνεχώς.

Η νέα εκδοχή των workers στηρίζεται στην ύπαρξη ενός κύριου scheduler που τρέχει κάθε η δευτερόλεπτα και ψάχνει από τη βάση ποια αιτήματα πρέπει να εκτελεστούν. Ποιο συγκεκριμένα κάθε φορά που αποθηκεύουμε ένα αίτημα προς έλεγχο στη βάση, δημιουργούμε ένα ακόμα αντικείμενο που αφορά ένα job. Σε αυτό το αντικείμενο αποθηκεύεται το id το αιτήματος το οποίο θα πρέπει να εκτελείται, η ημερομηνία που αυτό αποθηκεύτηκε, το χρονικό διάστημα μεταξύ των διαδοχικών αιτημάτων, το πότε πρέπει να εκτελεστεί ξανά το job (χρόνος δημιουργίας + χρονικό διάστημα μεταξύ αιτημάτων), καθώς και κάποια άλλα πεδία που θα εξηγήσουμε στη συνέχεια. Κάθε τέτοιο αντικείμενο αποθηκεύεται σε μία συλλογή αντικειμένων (collection) στην οποία έχει πρόσβαση μόνο ένας worker. Όταν εκτελούμε τον κώδικα ενός worker αρχικά ξεκινάει ένας scheduler που εκτελεί μία διεργασία κάθε δέκα δευτερόλεπτα (χρόνος που επιλέξαμε εμείς). Εντός αυτής ψάχνει στη βάση, και συγκεκριμένα στο collection απο το οποίο διαβάζει αποκλειστικά ο worker, προκειμένου να βρει jobs που πρέπει να εκτελεστούν. Αν εντοπιστεί κάποιο του οποίου ο αποθηκευμένος, που ήδη έχει, χρόνος που πρέπει να ξαναεκτελεστεί έχει ξεπεραστεί χρονικά, σημαίνει ότι πρέπει να εκτελέσει το αίτημα του οποίου το id έχει αποθηκευμένο. Αν κριθεί ότι ένα job πρέπει να ξεκινήσει να εκτελείται, τροποποιούμε στην καταχώρηση του στη βάση ένα πεδίο (locked) που καθορίζει αν το job ήδη τρέχει ή όχι. Στην περίπτωση που αυτό είναι κλειδωμένο, κανένας scheduler δεν θα εκτελέσει το job μέχρι να "ξεκλειδωθεί" και φυσικά να καλυφθεί η πρώτη συνθήκη σχετικά με το χρόνο πλήρωσης και τελευταίας εκτέλεσης του job.

Αξίζει σε αυτό το σημείο να σημειωθεί ότι κάθε worker θα πρέπει να διαθέτει ένα ξεχωριστό όνομα. Αυτό γίνεται γιατί κάθε ένας διαθέτει το δικό του collection στη βάση από το οποίο διαβάζει όλα τα jobs και μετέπειτα αποφασίζει ποια από αυτά θα τρέξουν και πότε.

Στη συνέχεια θα δούμε την πορεία της πληροφορίας στο σύστημα που υλοποιήσαμε.

1. Χρήστης εισάγει κάποιο url το οποίο επιθυμεί να ελέγχεται ανά κάποιο συγκεκριμένο χρονικό διάστημα. Πέρα από την εισαγωγή της διεύθυνσης της ιστοσελίδας έχει τη δυνατότητα να παραμετροποιεί το αίτημά που θα εκτελείται επιλέγοντας headers, body (σε περίπτωση post αιτήματος) query, μέθοδο HTTP (get, put, post, delete) και τύπο απόκρισης (json, text, stream). Επιπλέον μπορεί να επιλέγει το πόσο συχνά θα γίνονται οι διαδοχικοί έλεγχοι. Πιο συγκεκριμένα μπορεί να επιλέξει χρονικά διαστήματα από τριάντα δευτερόλεπτα μέχρι και διάστημα μία ολόκληρης ημέρας. Τέλος, στο κομμάτι της

```
_id: ObjectId('6488beef4984b9eb9f2112cc')
name: "api"
▶ data: Object
  type: "normal"
  priority: 0
  nextRunAt: 2023-06-14T15:18:27.728+00:00
  repeatInterval: 30000
  repeatTimezone: null
  lastModifiedBy: "sisiphus"
  lockedAt: 2023-06-14T15:18:22.868+00:00
  failCount: null
  failReason: null
  failedAt: null
  lastFinishedAt: 2023-06-14T15:17:57.899+00:00
  lastRunAt: 2023-06-14T15:17:57.728+00:00
  progress: null
```

Σχήμα 4.7: Παράδειγμα εγγραφής ενός **Job** στη βάση. Το πεδίο *lockedAt* έχει τιμή διάφορη του null, όταν το job ήδη εκτελείται. Το πεδίο *nextRunAt* υπολογίζεται από το άθροισμα του χρόνου πλήρωσης της προηγούμενης εκτέλεσής του (*lastFinishedAt*) και του interval. Στο πεδίο *data* έχει αποθηκευμένη όλη την πληροφορία που χρειάζεται για να εκτελέσει το αίτημα για το οποίο δημιουργήθηκε το συγκεκριμένο job. Βάσει των πεδίων αυτών κυρίως κρίνεται το αν ο κεντρικός scheduler θα ξεκινήσει τη διεργασία

παραμετροποίησης μπορεί να καθορίσει την επιθυμητή κατάσταση απόκρισης του αιτήματος (εξ ορισμού **Status Code 200**) και ένα response body, βάση του οποίου αν θέλει μπορεί να ελέγχει την απόκριση του εξωτερικού συστήματος (ιδιαίτερα χρήσιμο για τον έλεγχο διαδικτυακών APIs). Όλα αυτά αποθηκεύονται σε ένα collection με την ονομασία **Api**. Στο [σχήμα 4.8](#) μπορούμε να δούμε τη μορφή ενός Api αρχείου αποθηκευμένο στη βάση.

```
_id: ObjectId('646a7191e2e1549853a830')
title: "Develop Ultimate Project"
owner: ObjectId('621353c90edb7029bf8a6cef')
paused: false
queue: "sisiphus"
url: "https://dev-server.cyclopt.services/api/projects/644239156f7d9a658820..."
method: "GET"
type: "json"
interval: 1
▶ options: Object
  ▶ searchParams: Object
    branch: "main"
    token: "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjYyMTM1M2M5MGVhYjcwMjliZm..."
  ▶ headers: Object
  ▶ json: Object
  createdAt: 2023-05-21T19:31:29.621+00:00
  updatedAt: 2023-05-22T00:18:13.447+00:00
  __v: 0
  ▶ responseOptions: Object
    statusCode: 200
    ▶ body: Object
    description: null
```

Σχήμα 4.8: Παράδειγμα εγγραφής ενός **Api** στη βάση

2. Για να γίνει όμως η αποθήκευση θα πρέπει πρώτα να ξεκινήσει ένα post αίτημα στον σέρβερ του συστήματός μας, στο σώμα του οποίου θα περιέχονται όλες οι επιλογές που έχει κάνει ο χρήστης. Πριν όμως το αποθηκεύσει στη βάση, φάχνει τον πιο ελεύθερο από άποψη φόρτου worker. Αυτό γίνεται μετρώντας τις καταχωρήσεις των collection που αντιστοιχούν σε κάθε έναν από αυτούς. Αφού εντοπιστεί και επιλεγθεί ο καταλληλότερος, δημιουργεί ένα *Api* αντικείμενο στη βάση δεδομένων καθώς και ένα *job* στο collection του αντίστοιχου worker.
3. Στη συνέχεια και για να ακολουθήσουμε την ροή του συστήματος, μεταφερόμαστε στους workers και πιο συγκεκριμένα σε αυτόν που επιλέχθηκε στο προηγούμενο βήμα. Η λειτουργία αυτού του τμήματος του Lychte μελετήθηκε παραπάνω. Αυτό το οποίο δεν αναφέραμε είναι το τι ακριβώς κάνει κάθε process που τρέχει εντός του worker. Κάθε τέτοιο child process έχοντας ως δεδομένο ένα *Api* αντικείμενο (αυτό που αποθηκεύτηκε σε προηγούμενο στάδιο από τον σέρβερ) γνωρίζει που πρέπει να απευθύνεται το αίτημα που εκτελεί καθώς και τις παραμέτρους που ίσως αυτό χρειάζεται. Επίσης είναι υπεύθυνο δυνητικά για το έλεγχο τις ορθότητας της απόκρισης του εξωτερικού συστήματος, μέσα από την σύγκριση της με τα στοιχεία που έδωσε ο χρήστης κατά τη δημιουργία του *Api*. Αν ανάλογη πληροφορία δεν υπάρχει αποθηκευμένη στη βάση δεν πραγματοποιούνται έλεγχοι και η απάντηση που λαμβάνουμε αποθηκεύεται στη βάση δεδομένων. Αξίζει να σημειωθεί ότι όταν ο τύπος της απάντησης που επιστρέφεται είναι οποιοσδήποτε πέραν του json, δεν αποθηκεύεται το πραγματικό response, αλλά το αποτέλεσμα της απάντησης, αν δηλαδή εκτελέστηκε επιτυχώς ή όχι (συνεπώς αποθηκεύεται μία Boolean μεταβλητή). Πέρα από αυτά κρατάμε ακόμα τους χρονισμούς του αιτήματος (πότε ξεκίνησε, πότε ολοκληρώθηκε, πόσο διήρκεσε) και δύο ακόμα Boolean μεταβλητές που σχετίζονται με σφάλματα που εμφανίστηκαν κατά την εκτέλεση του (σφάλμα είτε του δικού μας συστήματος είτε του εξωτερικού συστήματος).
4. Τέλος η πληροφορία που αποθηκεύεται σε όλα τα στάδια της διαδικασίας στέλνεται στον client, κάθε φορά που αυτός, χρησιμοποιώντας τη γραφική διεπαφή που δημιουργήσαμε, επιλέγει να δει τα διάφορα διαγράμματα και μετρικές που υπολογίζουμε με τα δεδομένα που αφορούν το αίτημα που θέλει να ελέγξει.

Το σύστημα όμως, όπως είναι τώρα δεν είναι σε θέση να μπορεί να δείχνει επαρκώς γρήγορα ιστορικά δεδομένα. Αν υποθέσουμε ότι έχουμε ένα *Api* που θέλουμε να ελέγχουμε κάθε λεπτό, στο τέλος της ημέρας θα έχουν μαζευτεί από αυτό και μόνο 1440 εγγραφές. Αν αυτά συνεχίζουν να μαζεύονται τότε ακόμα και τα πιο απλά queries στο μοντέλο των αποκρίσεων θα αργούν καθιστώντας έτσι την εφαρμογή λιγότερο ανταποκριτική. Για την αποφυγή συσσώρευσης περιττής, μετά από ορισμένο χρονικό διάστημα, πληροφορίας, έχουμε βάσει επιπρόσθετα σε κάθε worker ένα ακόμα job που αφορά τον καθαρισμό της βάσης από εγγραφές που προέρχονται από το καθένα. Η διεργασία αυτή τρέχει μία φορά την εβδομάδα. Κάθε φορά που εκτελείται φάχνει τα *Apis* ανάλογα με τον worker στον οποίο τρέχει.

```

    _id: ObjectId('64613c6c5995ffaa6b8a363a')
    api: ObjectId('64613b3d0d17fa0523078cc0')
    response: true
    ◀ timings: Object
      start: 1684094059710
      socket: 1684094059710
      lookup: 1684094059762
      connect: 1684094059829
      secureConnect: 1684094059930
      upload: 1684094059931
      response: 1684094060426
      end: 1684094060427
      ◀ phases: Object
        duration: 717
      statusCode: 200
      hasError: false
      hasFailed: false
      createdAt: 2023-05-14T19:54:20.429+00:00
      updatedAt: 2023-05-14T19:54:20.429+00:00
      __v: 0
    _id: ObjectId('648460e6f6a27c2dc306270e')
    api: ObjectId('64616a0caee8d808355d2e96')
    response: null
    statusCode: 400
    hasError: true
    hasFailed: false
    createdAt: 2023-06-10T11:39:18.372+00:00
    updatedAt: 2023-06-10T11:39:18.372+00:00
    __v: 0

```

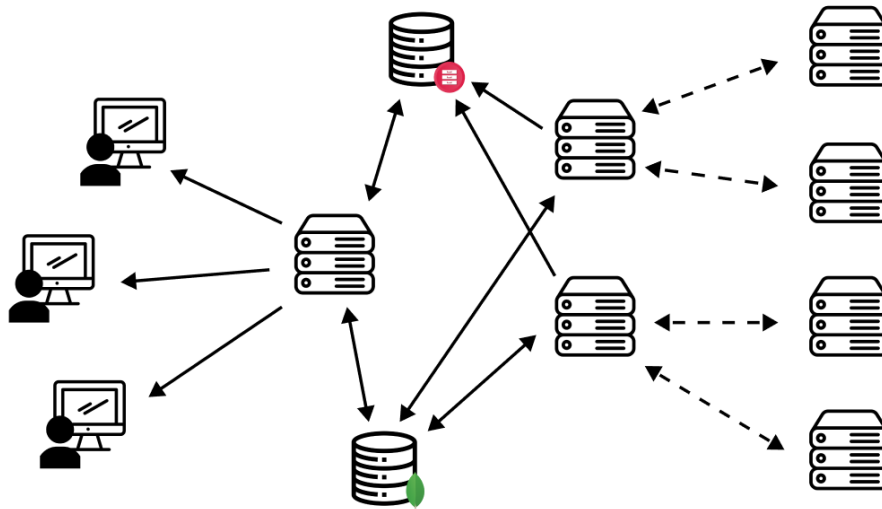
Σχήμα 4.9: Παράδειγμα εγγραφής **Response** στη βάση. Αριστερά βλέπουμε ένα επιτυχές response και δεξιά ένα που απέτυχε λόγω σφάλματος από τη μεριά του υπό παρακολούθηση συστήματος (το καταλαβαίνουμε από το πεδίο hasError που είναι true).

Έπειτα μαζεύει όλα τις αποκρίσεις που σχετίζονται με το κάθε Api ξεχωριστά και ξεκινάει τους υπολογισμούς. Τα δεδομένα που αντλούνται στο πλαίσιο αυτής της διεργασίας στο τέλος διαγράφονται από τη βάση, ώστε όλα τα αιτήματα προς τη βάση να εκτελούνται πιο γρήγορα και αποδοτικά. Σβήνοντας όμως πληροφορία χάνουμε την ιστορικότητα των δεδομένων μας με αποτέλεσμα να χάνουν και οι χρήστες μία πιο γενική εικόνα των ελέγχων που έχουν πραγματοποιηθεί. Στο σημείο αυτό εισάγεται μία άλλη μορφή βάσης δεδομένων, η Google Cloud Storage.

Πριν σβήσουμε οτιδήποτε κρίνουμε ότι δεν είναι απαραίτητο να υπάρχει στη βάση, κάτι που δεν θα δείξουμε σε διαγράμματα άμεσα, το ματεφέρουμε στο Cloud Storage της Google, υπό μορφή αρχείων. Με τον τρόπο αυτό κερδίζουμε χώρο, στη κεντρική Mongo βάση, στη λειτουργία της οποίας στηρίζεται όλο το σύστημα, και γενικά απόδοση καθώς έχει καλύτερες ταχύτητες στο σύνολο.

Πριν συνεχίσουμε στο κομμάτι των αποτελεσμάτων θα αναφερθούμε πιο συγκεκριμένα στις ενέργειες που εκτελεί η παραπάνω διαδικασία:

1. Συλλογή όλων το Responses που υπάρχουν αποθηκευμένα στη βάση, μέχρι μία μέρα πριν την εκτέλεση της διεργασίας καθαρισμού. Αυτό γίνεται προκειμένου να σιγουρευτούμε ότι υπάρχει πάντα πληροφορία στη βάση για την προηγούμενη ημέρα που μπορούμε να δείξουμε.
2. Δημιουργία δύο διαφορετικών πινάκων, ενός που αποθηκεύει του χρόνους διάρκειας των αιτημάτων και ενός που περιέχει Boolean μεταβλητές σχετικά με το αν τα αιτήματα γίνονται επιτυχώς ή όχι.
3. Στη συνέχεια για κάθε μέρα στην οποία έχουμε εγγραφές Responses υπολογίζουμε και αποθηκεύουμε στατιστικά σχετικά με τη μέση τιμή, διάμεσο και τυπική απόκλιση διάρκειας αιτημάτων.



Σχήμα 4.10: Τελικό διάγραμμα Lychte

4. Στο τέλος της διαδικασίας αποθηκεύονται τρία αρχεία για κάθε Api. Ένα στο οποίο περιέχονται τα responses όπως ακριβώς ήταν αποθηκευμένα στη mongodb (σε αρχείο της μορφής `../apiId/raw-responses/MM-DD-YYYY.json`), ένα που περιέχει τους πίνακες που περιγράψαμε στο δεύτερο βήμα (`../apiId/raw-values/MM-DD-YYYY.avro`) και τέλος ένα αρχείο που περιέχει κάποια χρήσιμα στατιστικά για κάθε μέρα που εντοπίστηκε, όπως περιγράφεται στο βήμα τρία (`../apiId/statistics.avro`)

Επειδή τα αρχεία που περιέχουν στατιστικά για κάθε μέρα χρησιμοποιούνται αρκετά συχνά από τη γραφική διεπαφή του Lychte για την παρουσίαση ιστορικών δεδομένων του υπό μελέτη Api, κρίναμε απαραίτητη την χρήση ενός διαφορετικού τύπου αρχείου σε σχέση με τον κλασικό και ευρέως διαδεδομένο στο διαδίκτυο τύπο JSON, το AVRO, για να κερδίσουμε ως προς τον αποθηκευτικό χώρο αλλά και την ταχύτητα ανάγνωσης. Αξίζει να σημειωθεί ότι ένα αρχείο avro σε σχέση με ένα αρχείο json που περιέχουν ακριβώς την ίδια πληροφορία για τα στατιστικά στα οποία αναφερόμαστε, είναι 70% περίπου μικρότερο σε μέγεθος (json αρχείο 250Mb, avro αρχείο 75Mb) Ο λόγος που μπορούμε να χρησιμοποιήσουμε avro αρχεία είναι ότι η μορφή της αποθηκευμένης πληροφορία παραμένει ίδια και δεν αλλάζει. Πιο συγκεκριμένα τα schemas που χρησιμοποιούνται στα αρχεία `"raw-values/MM-DD-YYYY.avro"` και `"statistics.avro"` φαίνονται στο [σχήμα 4.11](#).

```

const statisticsType = Type.forSchema({
  type: "array",
  name: "statistics",
  items: [
    {
      type: "record",
      fields: [
        {
          name: "date",
          type: "string",
        },
        {
          name: "durationMean",
          type: "double",
        },
        {
          name: "durationStd",
          type: "double",
        },
        {
          name: "durationMedian",
          type: "double",
        },
        {
          name: "durationMin",
          type: "double",
        },
        {
          name: "durationMax",
          type: "double",
        },
        {
          name: "durationQuartiles",
          type: [
            "null",
            {
              name: "nestedDurationQuartiles",
              type: "record",
              fields: [
                {
                  name: "q1",
                  type: "double",
                },
                {
                  name: "q3",
                  type: "double",
                },
              ],
            },
          ],
        },
        {
          name: "successRate",
          type: "double",
        },
      ],
    },
  ],
});

const rawValuesType = Type.forSchema({
  type: "record",
  name: "rawValues",
  fields: [
    {
      name: "durations",
      type: [
        "null",
        {
          type: "array",
          items: [
            { type: "long" },
          ],
        },
      ],
    },
    {
      name: "successes",
      type: [
        "null",
        {
          type: "array",
          items: [
            { type: "boolean" },
          ],
        },
      ],
    },
  ],
});

```

Σχήμα 4.11: Schemas πληροφορίας που αποθηκεύουμε σε ανρο αρχεία. Το πρώτο αφορά τα statistics που αποθηκεύονται και αποτελείται από ένα array από objects, τα fields των οποίων σχετίζονται σε μετρικές στατιστικής φύσης. Το δεύτερο schema χαρακτηρίζει τα raw-values που αποθηκεύουμε και αποτελείται από ένα object με δύο arrays

5

Μετρήσεις και Επίδειξη Συστήματος

Στο κεφάλαιο αυτό θα μελετήσουμε την απόδοση του συστήματος Lychte που υλοποιήσαμε στο πλαίσιο της διπλωματικής αυτής εργασίας και στο τέλος θα παρουσιαστούν εικόνες από τη γραφική διεπαφή, την εφαρμογή δηλαδή που αναπτύξαμε.

5.1 ΜΕΤΡΗΣΕΙΣ

Προκειμένου να παρακολουθήσουμε τις ανάγκες και να δοκιμάσουμε τα όρια του Lychte κάναμε μία σειρά διαφορετικών μετρήσεων. Η παράμετρος που αλλάζει σε κάθε μέτρηση είναι ο αριθμός των Apis/Jobs που τρέχει παράλληλα ένας worker. Έτσι σε κάθε διαφορετική μέτρηση κάναμε n καταχωρήσεις Apis και Jobs αντίστοιχα, όπου n ο αριθμός των αιτημάτων που θέλουμε να ελέγχουμε. Στο σημείο αυτό πρέπει να αναφερθεί ότι οι μετρήσεις έγιναν σε μηχανήμα με τα εξής τεχνικά χαρακτηριστικά:

Πίνακας 5.1: Τεχνικά χαρακτηριστικά Υπολογιστικού Συστήματος

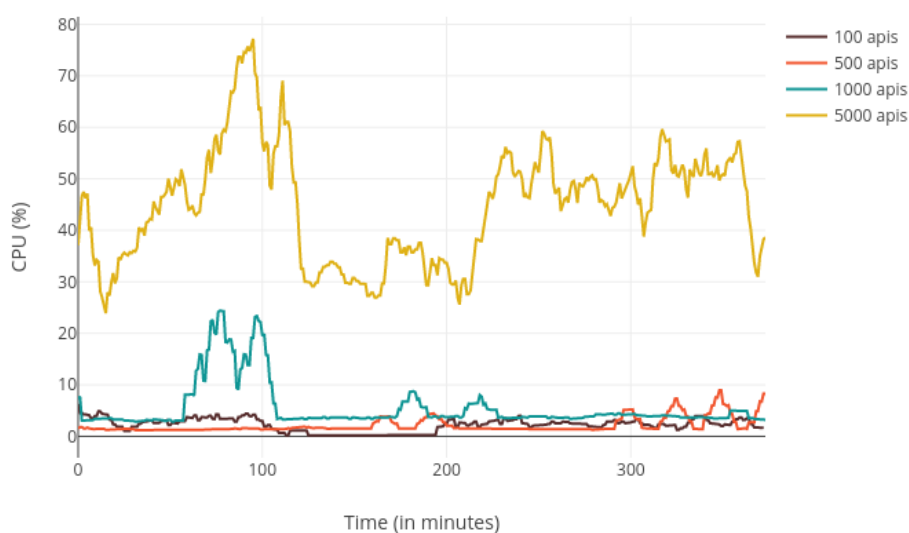
Operating System	WindowsOS
Processor	AMD Ryzen 7 5800H (3.2Ghz)
RAM	16GB

5.1.1 Περιγραφή

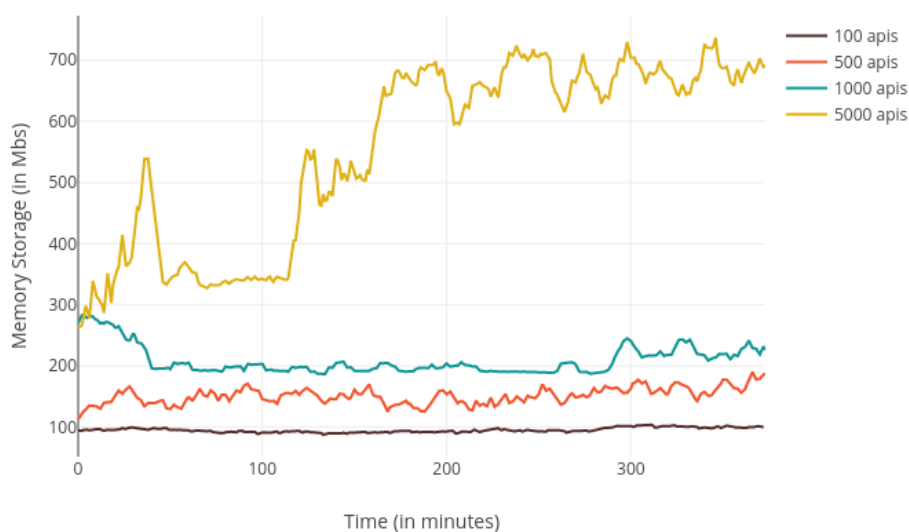
ΓΡΑΨΕ ΓΙΑ ΤΟ ΤΡΟΠΟ ΠΟΥ ΜΠΗΚΑΝ ΤΑ APIS ΣΤΗ ΒΑΣΗ (ΜΕΤΑΞΥ 8 APIS ΕΠΙΛΕΟΝΤΑΝ ΕΝΑ ΣΤΗΝ ΤΥΧΗ ΜΕ INTERVAL 0.5 Ή 1 Ή 2 Ή 3)

ΚΕΦΑΛΑΙΟ 5. ΜΕΤΡΗΣΕΙΣ ΚΑΙ ΕΠΙΔΕΙΞΗ ΣΥΣΤΗΜΑΤΟΣ

ΑΝΑΦΕΡΕ ΌΤΙ ΑΥΤΑ ΑΦΟΡΟΥΝ ΤΙΣ ΤΕΛΕΥΤΑΙΕΣ ΤΡΕΙΣ ΩΡΕΣ ΛΕΙΤΟΥΡΓΙΑΣ, Ή ΟΤΙ ΑΝΑΦΕΡΟΝΤΑΙ ΣΕ ΤΡΕΙΣ ΩΡΕΣ ΛΕΙΤΟΥΡΓΙΑ ΓΕΝΙΚΑ ΜΕΤΡΗΣΗ ΚΑΘΕ ΤΡΙΑΝΤΑ ΔΕΥΤΕΡΟΛΕΠΤΑ



Σχήμα 5.1: Αποτελέσματα χρήσης της CPU



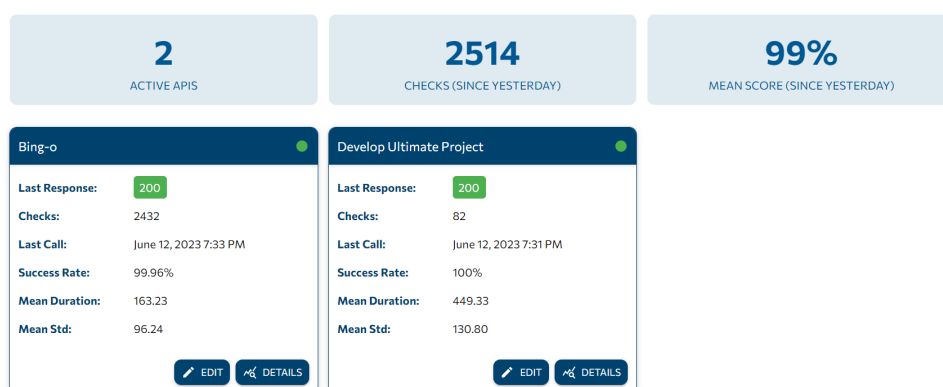
Σχήμα 5.2: Αποτελέσματα χρήσης της RAM

ΓΕΝΙΚΑ ΣΤΑΤΙΣΤΙΚΑ ΑΠΟ ΤΑ ΠΑΡΑΠΑΝΩ ΣΕ ΜΕΓΑΛΥΤΕΡΟ ΒΑΘΟΣ ΧΡΟ-

NOY (12 ΩΡΕΣ)

Πίνακας 5.2: Αποτελέσματα Μετρήσεων

	Μέση Απόκλιση	Μέση Χρήση CPU	Μέση Χρήση RAM
100 apis	-112.62ms	1.22%	102.54Mb
500 apis	-69.32ms	3.26%	171.1Mb
1000 apis	-47.9ms	7.46%	223.13Mb
5000 apis	664ms	56.36%	652.45Mb



Σχήμα 5.3: Lychte Api Overview

The screenshot shows the 'Create new api' form in Lychte. The form is divided into 'BASIC CONFIGURATION' and 'ADVANCED OPTIONS' tabs. The 'BASIC CONFIGURATION' tab is active, showing fields for:

- Title:** A text input field with placeholder text 'Set a Title to help you easily manage your apis'.
- Description:** A text input field with placeholder text 'Optionally set a description'.
- Url:** A text input field with placeholder text 'Set the the url that you want to be monitored'.
- Method:** A dropdown menu currently set to 'GET'.
- Type:** A dropdown menu currently set to 'json'.
- Ping Interval:** A slider control set to 1 minute, with markers for 1h, 2h, and 1d.

 At the bottom right of the form are 'CANCEL' and 'CREATE' buttons.

Σχήμα 5.4: Lychte Create Api Check

The screenshot shows the 'Create new api' form with the 'ADVANCED OPTIONS' tab selected. The 'Request Options' section is expanded, showing three fields: 'Headers', 'Body', and 'Query'. The 'Headers' field contains a JSON object with a 'token' property. The 'Body' field contains a JSON object with 'title' and 'severity' properties. The 'Query' field contains a JSON object with 'id' and 'showClosed' properties. At the bottom right, there are 'CANCEL' and 'CREATE' buttons.

```
Headers: {
  "token": "aashdasldhasdkhjasd"
}

Body: {
  "title": "test",
  "severity": "high",
}

Query: {
  "id": "60dc1862629fb9c4d68fc758",
  "showClosed": "true",
}
```

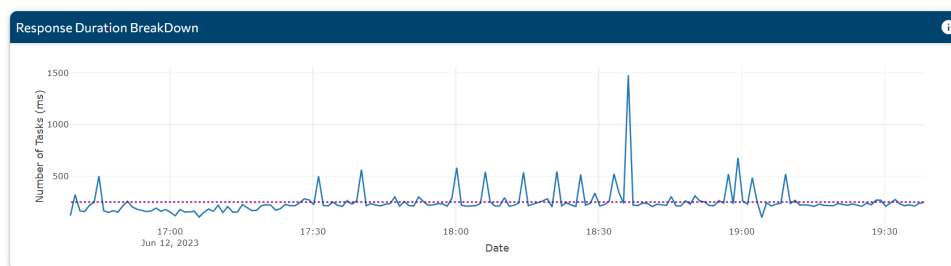
Σχήμα 5.5: Lychte Create Api Advanced Check

The screenshot shows the 'Create new api' form with the 'ADVANCED OPTIONS' tab selected. The 'Response Options' section is expanded, showing two fields: 'Body' and 'Status Code'. The 'Body' field contains a JSON object with a 'success' property. The 'Status Code' field is a dropdown menu showing '200 OK'. At the bottom right, there are 'CANCEL' and 'CREATE' buttons.

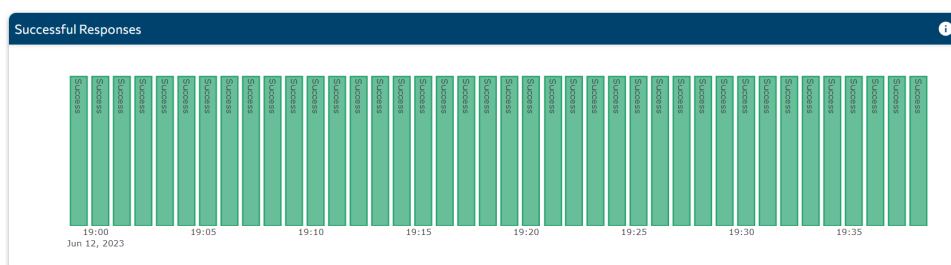
```
Body: {
  "success": "true",
}

Status Code: 200 OK
```

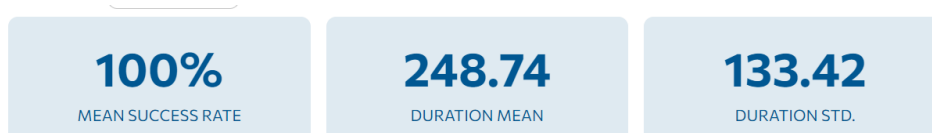
Σχήμα 5.6: Lychte Create Api Advanced v2 Check



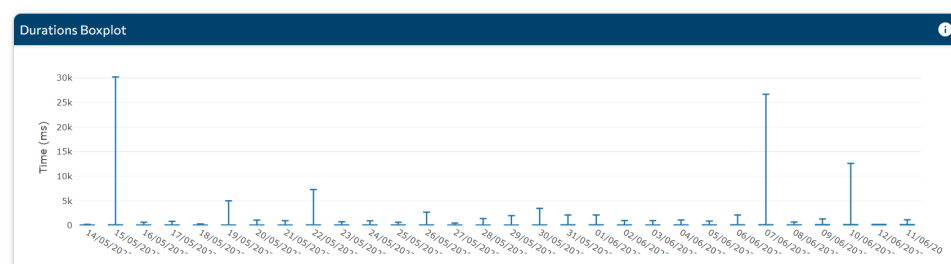
Σχήμα 5.7: Διάγραμμα Διάρκειας Αποκρίσεων



Σχήμα 5.8: Διάγραμμα Κατάστασης Αποκρίσεων



Σχήμα 5.9: Μετρικές που αφορούν το σύνολο των δεδομένων (ιστορικά δεδομένα)



Σχήμα 5.10: Διάγραμμα Θηκογραμμάτων (Boxplots) της διάρκειας απόκρισης αιτημάτων για κάθε μέρα (ιστορικά δεδομένα)

Βιβλιογραφία

- [1] Santhosh S and Narayana Swamy Ramaiah. “*Cloud-Based Software Development Lifecycle: A Simplified Algorithm for Cloud Service Provider Evaluation with Metric Analysis*“. Big Data Mining and Analytics, 6(2):127–138, june 2023.
- [2] Sorin POPA. “*WEB Server monitoring*“. Annals of University of Craiova - Economic Sciences Series, 2(36):710–715, may 2008.
- [3] Siddhesh Vaidya and Prabhat Padhy. “*View towards Synthetic Monitoring using HTTP Archive*“. International Journal of Engineering Research and Technology, august 2022.
- [4] Daniel Stenberg. “*HTTP2 explained*“. ACM SIGCOMM Computer Communication Review, 44:120–128, 07 2014.
- [5] Martino Trevisan, Danilo Giordano, Idilio Drago, and Ali Safari Khatouni. “*Measuring HTTP/3: Adoption and Performance*“, 02 2021.
- [6] Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino. “*An Analysis of Public REST Web Service APIs*“. IEEE Transactions on Services Computing, 14:957–970, 07 2021.
- [7] Liu Qigang and Xiangyang Sun. “*Research of Web Real-Time Communication Based on Web Socket*“. International Journal of Communications, Network and System Sciences, 05:797–801, 01 2012.
- [8] Roy Thomas Fielding and Richard N. Taylor. “*Architectural Styles and the Design of Network-Based Software Architectures*“. PhD thesis, University of California, Irvine, 2000. AAI9980887.
- [9] Hezbullah Shah and Tariq Soomro. “*Node.js Challenges in Implementation*“. Global Journal of Computer Science and Technology, 17:72–83, 05 2017.