Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης
Πολυτεχνική Σχολή
Τμήμα Ηλεκτρολόγων Μηχανικών &
Μηχανικών Υπολογιστών
Τομέας Ηλεκτρονικής και Υπολογιστών
Εργαστήριο Επεξεργασίας Πληροφορίας και
Τεχνολογίας Λογισμικού (ISSEL)

Διπλωματική Εργασία

Δημιουργία Συστήματος 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 πλαίσιο αυτής, καθιστούν επιτακτική την ανάγκη ύπαρξης συστημάτων που θα ελέγχουν την εύρυθμη λειτουργία τους. Πιο συγκεκριμένα μιλάμε για την ελέξιλη 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 υπηρεσίες (SaaS - Software as a Service) που ζουν στον Διαδίκτυο (World Wide Web). Η λειτουργία αυτών μπορεί να ελεχθεί με διάφορους τρόπους. Από Unit Testing, στο πλαίσιο του κύκλου ανάντυξης του λογισμικού (continuous integration, continuous deployment cycle) προκειμένου να ελεχθεί 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 παρέχει τη 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 αποκριθεί ορθά στα αιτήματα που δέχεται, και σχετικά με το χρόνο που χρειάστηκε προκειμένου 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 errors during the software's development process

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

June 2023

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

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
GCS ⟶ Google Cloud Storage
CPU ⟶ Central Processing Unit
RAM ⟶ Random Access Memory

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 σημεντικότερο 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 σημεντικό αντικείμενο μελέτης

[1]. Πέρα από τον οικομικό 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 Παρακολούθση (Active Monitoring)	**Παθητική Παρακολούθηση (Passive Monitoring)**
• Στηρίζεται σε συνθετικά API calls	• Αναλύει δεδομένα πραγματικών χρηστών
• Παράγει δεδομένα για συγκεκριμένες πτυχές του δικτύου	• Πλήρης εικόνα της απόδοσης του δικτύου
• Μπορεί να μετρήσει την κίνηση εντός και εκτός του δικτύου	• Μετράει κίνηση μόνο εντός του δικτύου
• Μπορεί να εντοπίσει προβλήματα πριν ακόμα μπορέσουν να τα εντοπίσουν οι χρήστες	• 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 πρωτού 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. Ακόμα αναλύονται τα αποτελέσματα που παρήχθησαν καθόλη της διάρκεια των πειραμάτων που διενεργήθηκαν.

1.3 Διαρθρωση 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 αποτελέσματα μετρήσεων που έγιναν στο τελικό σύστημα καθώς και εικόνες της γραφικής διεπαφής που υλοποιήσαμε.

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

2
Θεωρητικό Υπόβαθρο

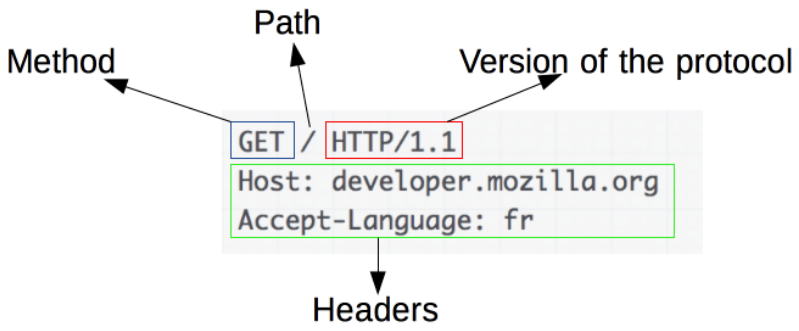
Στο κεφάλαιο αυτό θα παρουσιαστούν εργαλεία που χρησιμοποιήθηκαν για την υλοποίηση του Συστήματος Ενεργής Παρακολούθησης, καθώς και έννοιες και τεχνολογίες που αξιοποιήθηκαν για το σκοπό αυτό.

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, η οποία φαίνεται και στο 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 υπήρχε
- **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**. Καθώς όμως, οι υπόλοιπες αυτές οι μέθοδοι, δεν χρησιμοποιούνται τόσο συχνά στην πράξη, δεν θα αναλυθούν περαιτέρω.

2.1.2 Εκδόσεις HTTP

Η πρώτη έκδοση του HTTP, παρόλλο που δεν είχε κάποια συγκεκριμένο τίτλο, εκ των υστέρων ονομάστηκε HTTP/0.9. Αποτελεί την πιο απλή έκδοση του πρωτοκόλλου. Δεν υποστηρίζονταν headers και 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 υπαρχουσες 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 Speedy (SPDY) 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 protocoll), 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 αιτήται, μάλλον θα ζητήσει εντός της ιδίας συνεδρίας.

Τέλος, πρέπει να αναφερθούμε στην τελευταία, αν και όχι ακόμα ευρέως διαδεδομένη, έκδοση HTTP/3.0. Η βασική διαφορά με τους πρωκατόχους του είναι ότι αλλάζει το πρωτόκολλο επικοινωνίας που χρησιμοποιεί όλα αυτά τα χρόνια, από TCP (Transfer Communication Protocoll) σε έναν συνδυασμό UDP (User Datagram Protocoll) και QUIC (Quick UDP Internet Connections), 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 κατάστασεις (status codes) αποτελούν μέρος της απάντησης του server. Επιτρέπουν στον χρήστη να καταλάβει με μία ματιά αν το αίτημα που έχει 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 αποσταλλεί το αίτημα
- **Εύρος 400-499**: Ανεπιτυχές αίτημα, που οφείλεται στον client. Ένα σύνηθες παράδειγμα είναι η αίτηση πρόσβασης σε προστατευόμενους πόρους χωρίς κάποιου είδους αυθεντικοποίηση, 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 Προγραμμάτων (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 λειουργήσει σωστά ο server αλλά και αντίστροφά, 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 APIs, μπορούμε να διακρίνουμε τις εξής κατηγορίες:

- **Ανοιχτά (Open)**: Σε αυτού του τύπου διεπαφών λογισμικού, έχει ελεύθερη πρόσβαση ο καθένας, χωρίς να απαιτείται κάποια παραπάνω πληροφορία που αφορά την αυθεντικοποίηση ή ταυτοποίηση του χρήστη. Επειδή ακριβώς η πληροφορία που παρέχεται (ανοιχτά) είναι τεράστια, χρησιμοποιούνται πολύ συχνά σε έρευνες, όπως η [6] που αξιοποιεί ελεύθερα προσβάσιμους 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 προκειμένουν 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 APIs).

Αξίζει να σημειωθεί σε αυτό το σημείο ότι ο τρόπος δημιουργίας, η δομή, καθώς και τα πρωτόκολλα που χρησιμοποιούν τα διαφόρα 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, σύχρονες 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 ένα 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 αποστέλλει 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. Προτάθηκε 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 υπάρξη 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). Αυτές σχετίζονται με την ύπαρξη μεθόδων εντός του συστήματος που θα επιτρέπουν τη δημιουργία δεδομένων και την τροποποίηση, ανάγνωση, και διαγραφή ήδη υπάρχουσας πληροφορίας.

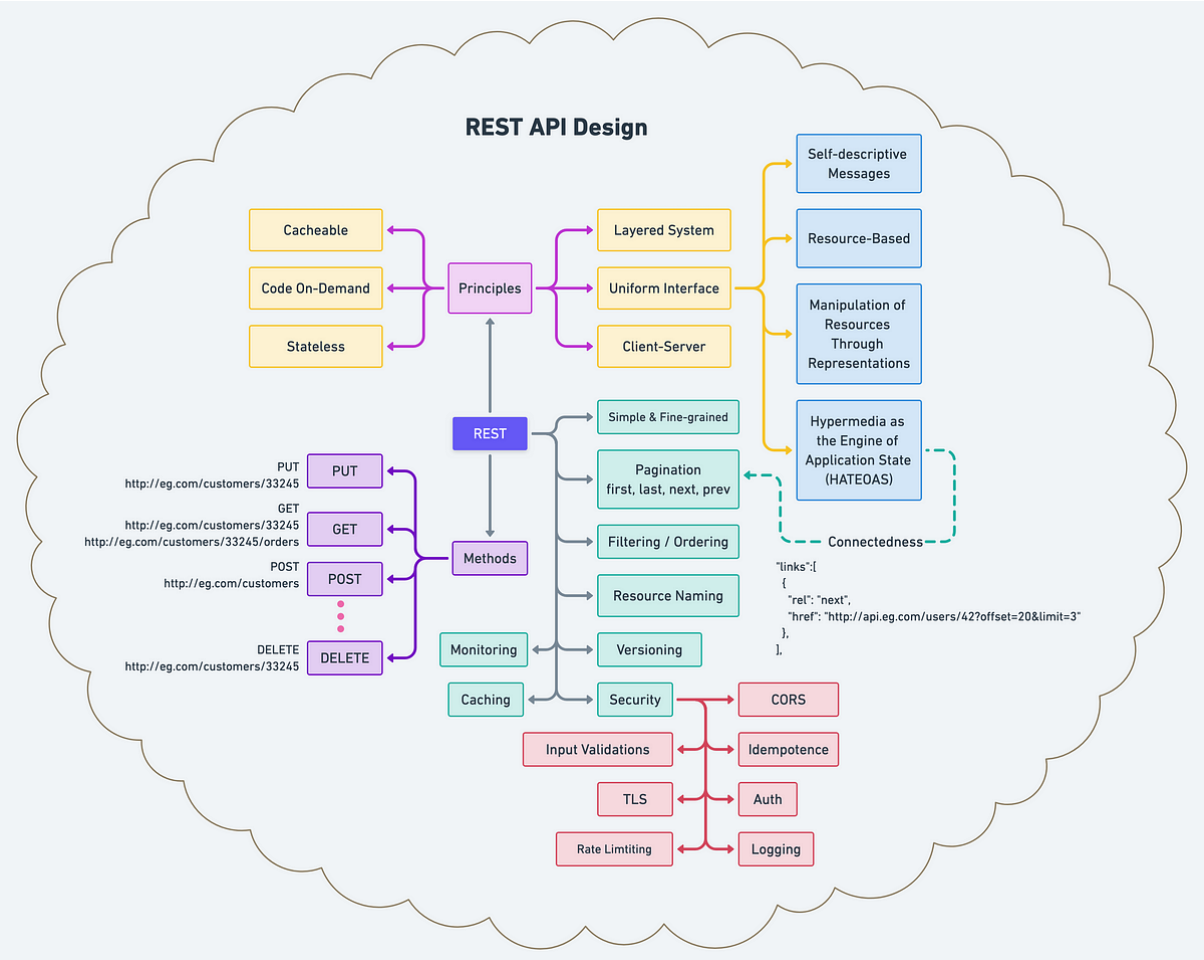
Για να χαρακτηριστεί ένα API ώς RESTful θα πρέπει να ικανοποιεί τα παρακάτω, ενώ στο σχήμα 2.2 μπορούμε να τα δούμε σε ένα γενικότερο πλαίσιο:

- **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 ελαφρής, 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 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 προγμάμματα 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 αποθήκευση πληροφορίας καταφέρνουν να γλιτώσουν χώρο στο τελικό προϊόν. Αυτό σε συνδύασμό 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.

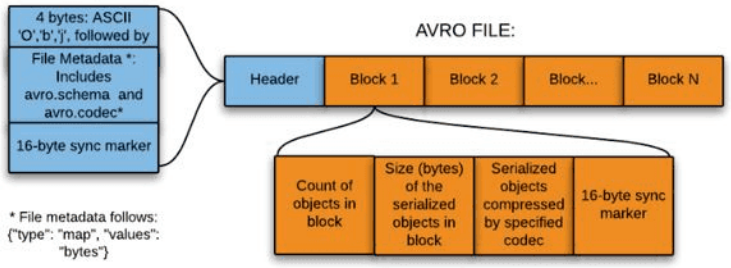
- Επιτρέπει την τροποποίηση και εξέλιξη του σχήματος των δεδομένων (schema evolution), δίνοντας τη δυνατότητα για αλλαγές και data migration που πολλές φορές καθίστανται επιτακτικά στη διάρκεια ζωής ενός προγράμματος που συνεχώς εξελίσσεται.

- 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, $i = 0, ..., k$

1. **Αριθμητική Μέση Τιμή**: Αποτελεί μία από τις πιο βασικές μετρικές που χρησιμοποιείται ευρέως σε στατιστικές μελέτες. Περιγράφει την τάση του εκάστοτε συνόλου δεδομένων που μελετάμε να κυμαίνεται γύρω από μία τιμή.
$$mean(x) = \frac{\sum_{n=1}^{k} x_n}{k} \tag{2.1}$$

2. **Διάμεσος**: Είναι η τιμή που διαχωρίζει το υψηλότερο μισό από το κάτω μισό ενός διατεταγμένου συνόλου δεδομένων
$$median(x) = \begin{cases} x_{\frac{n+1}{2}} & \text{για n περιττό} \\ \frac{x_{\frac{n}{2}} + x_{\frac{n}{2}+1}}{2} & \text{για n ζυγό} \end{cases} \tag{2.2}$$

3. **Τυπική Απόκλιση Πληθυσμού**: Περιγράφει το κατά πόσο απέχουν το σύνολο των δεδομένων, κατά μέση τιμή, από τον **Μέσο Όρο**
$$std(x) = \sqrt{\frac{\sum_{i=1}^{n}(x_i - mean(x))^2}{n-1}} \tag{2.3}$$

4. **Τεταρτημόρια**: Όπως και η διάμεσος αποτελούν τις τιμές που διαχωρίζουν το σύνολο, των διατεταγμένων κατά αύξουσα σειρά, δεδομένων στο 25% και 75% αντίστοιχα
$$\begin{cases} q1(x) = mean(x_i) & \text{όπου i = 0, ..., k/2} \\ q3(x) = mean(x_j) & \text{όπου j = k/2 + 1, ..., k} \end{cases} \tag{2.4}$$

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.

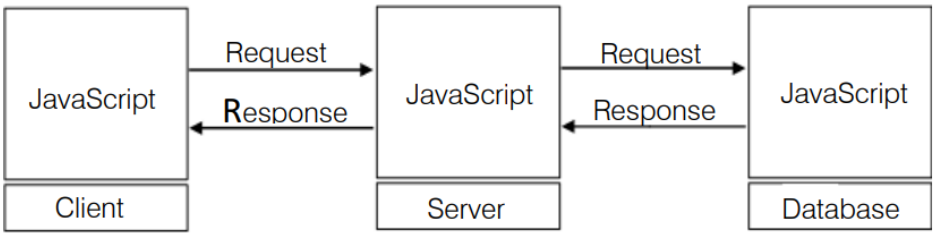
2.5.1 Node.js

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 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 συγγραφής 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. Μερικά από τα πιο γνωστά backend framerworks 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 διαχείρηση 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 έργο λογισμικού ανοιχτού κώδικα (open source project), 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 διαχείρηση 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 διαχείρηση αυτών. Εξ ορισμού, εφαρμογές γραμμένες σε JavaScript τρέχουν σε ένα νήμα (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	↺	status	cpu	mem	user	watching
0	lychte-server	default	N/A	fork	10432	66s	0	online	0%	66.4mb	stavs	disabled
1	phiphisus-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 BJSON (Binary JSON) και παρέχουν πληθώρα τύπων δεδομένων που μπορούν να αποθηκευτούν (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:

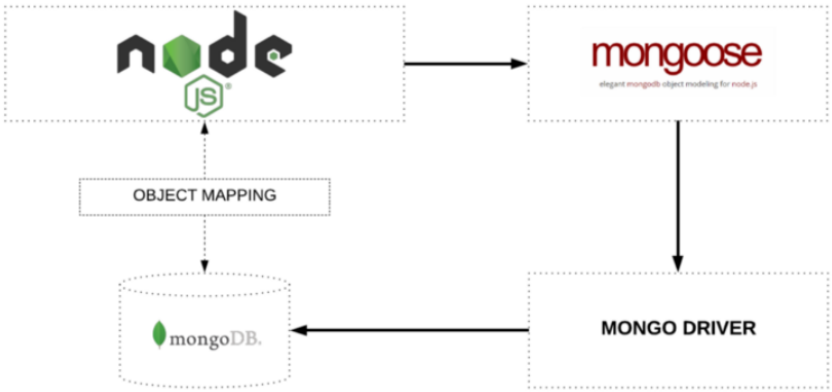
- **Document Oriented Storage**: η αποθήκευση και διαχείριση των δεδομένων (στο πλαίσιο της εφαρμογής) είναι εύκολη καθώς στηρίζεται σε δεδομένα σε μορφή JSON.

- **Ευρετήρια (Indexes)**: Υπάρχει η δυνατότητα, κατά τη διαδικασία του στησίματος της βάσης (αλλά και μετέπειτα), να οριστούν ευρετήρια σε πεδία που χρησιμοποιούνται συχνά σε queries προκειμένου να βελτιωθεί η 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 περισσότερους 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.

- **Εύκολη ενσωμάτωση**: Η mongo διαθέτει βιβλιοθήκες στο περιβάλλον του node.js που καθιστούν τη χρήση και το στήσιμό της πολύ απλά. 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 ώς 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 προαναφέρθηκε θέλουμε να αποθηκεύουμε ιστορικά δεδομένα, δεδομένα δηλαδή που δεν θα εμφανίζονται άμεσα στον χρήστη, αλλά θα χρησιμοποιούνται για τον υπολογισμό μετρικών και στατιστικά σημαντικών αποτελεσμάτων, στο σύνολο όλης της μέχρι τώρα αποθηκευμένης πληροφορίας.

Σε αυτό λοιπόν το σημείο έρχεται το Google Cloud Storage (GCS), το οποίο παρέχει μεταξύ άλλων, δυνατότητες Αποθήκευσης Αρχείων (File Storage). Τα δεδομένα αυτά δεν χρειάζεται να υπακούν 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 διαβάσει 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

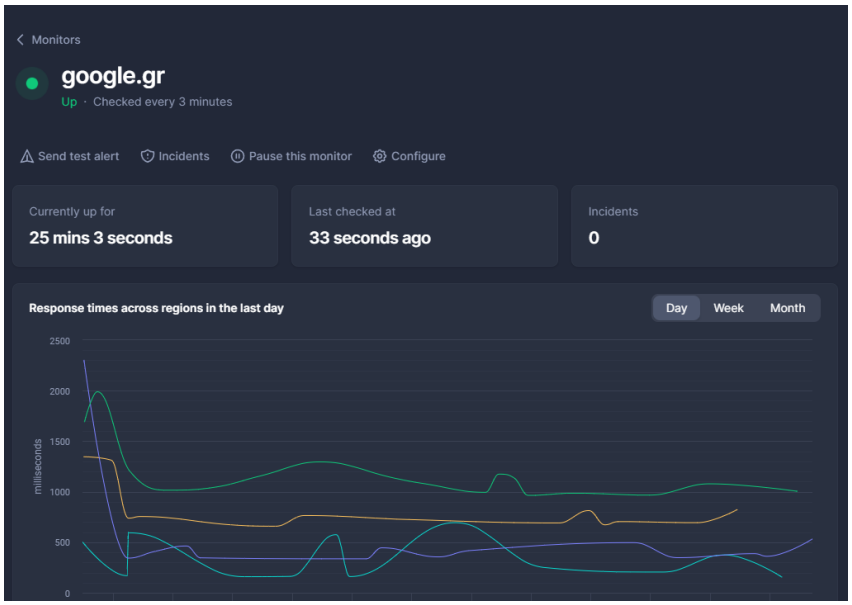
Βιβλιογραφική Αναζήτηση Τεχνολογιών Αιχμής

Πριν αναλυθεί η υλοποίηση του συστήματος που δημιουργήσαμε για την Ενεργή Παρακολούθηση Εφαρμογών ως Υπηρεσίες (SaaS) και ιστοσελιδών που εδρεύουν στο διαδίκτυο, θα πασουσιάσουμε τεχνολογίες που έχουν χτιστεί ήδη για τον 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 projects 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.

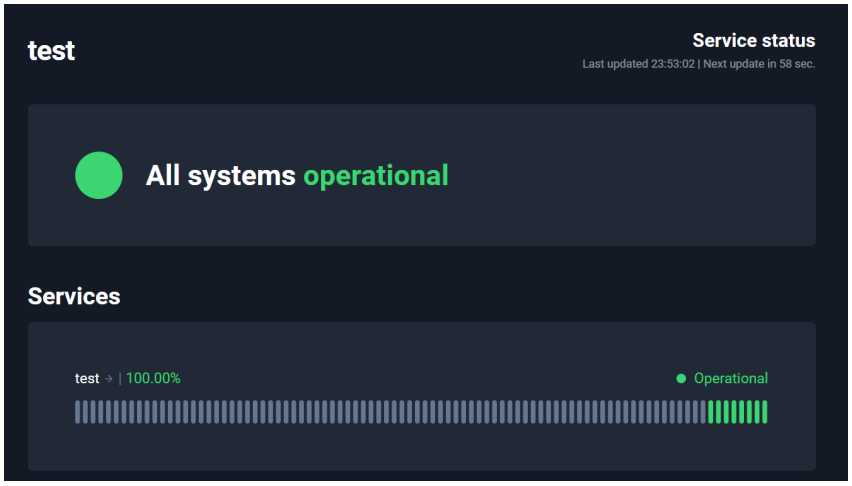
- **Better Uptime**: Προσφέρει εύκολη ενσωμάτωση των ιστοσελιδών που θέλει κανείς να παρακολουθήσει. Λειτουργεί κάνοντας ping κάθε τριάντα δευτερόλεπτα στο url που ορίζει ο χρήστης και παρουσιάζει τα παραγόμενα δεδομένα σε ευπαρουσίαστα διαγράμματα (σχήμα 3.1). Ένα από τα μεγάλα πλεονεκτήματα που έχει 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 τους 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 προκειμένη περίπτωση του συστήματος παρακολούθησης). Πέρα από αυτά μπορεί να καθορίσει την επιθυμητή



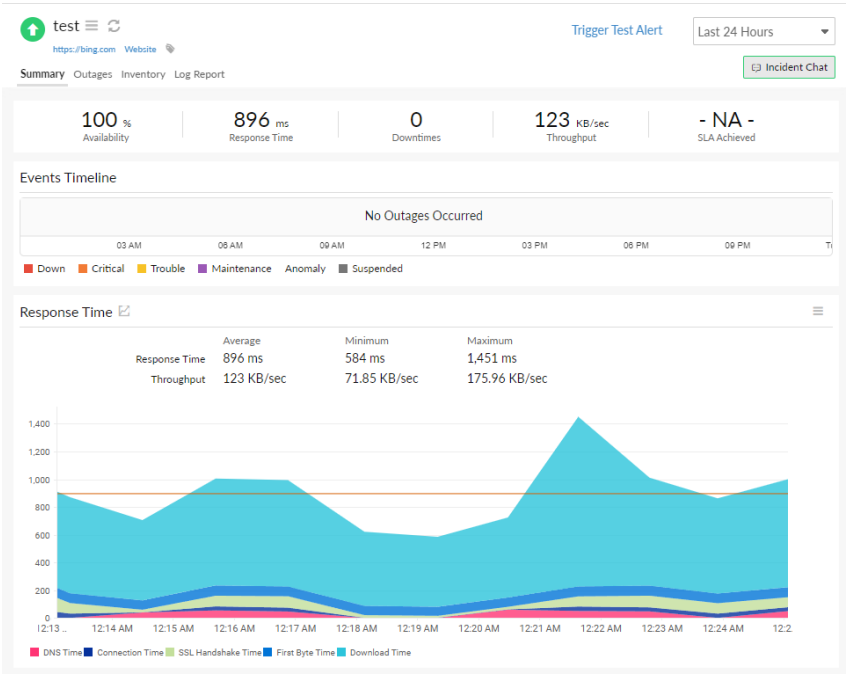
Σχήμα 3.1: Παράδειγμα χρήσης του εργαλείου Better Uptime

http κατάσταση της απόκρισης της ιστόσελίδας 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)



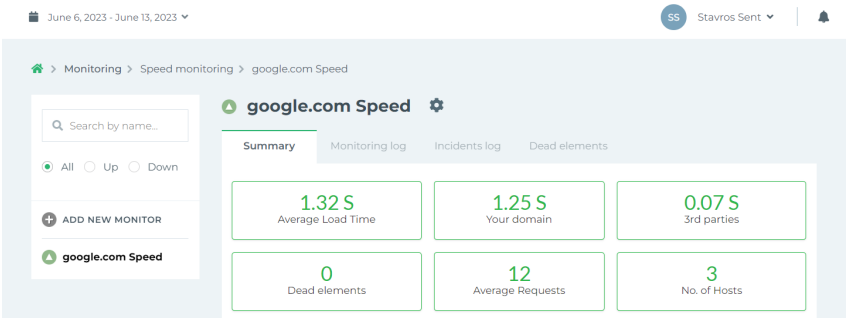
Σχήμα 3.2: Παράδειγμα χρήσης του εργαλείου Uptime Robot

- **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 καθιστα τα δεδομένα που συλλέγονται πιο εύκολα στην κατανόηση και οπτικοποίηση.
- **Uptimia**: Πέρα από τα κλασικού τύπου http αιτήματα, μπορεί να κάνει ελέγχους παρακολούθησης (uptime monitoring) σε DNS, UDP, TCP και email με



Σχήμα 3.3: Παράδειγμα χρήσης του εργαλείου Site24x7

απόσταση έως και τριάντα δευτερολέπτων (μεταξύ αιτημάτων). Αξίζει, να σημειωθεί, ότι η συγκεκριμένη εφαρμογή έχει δυνατότες και παθητικής παρακολούθησης (RUM). Αρχικά επιλέγεται το site το οποίο ο χρήστης θέλει να παρακολουθήσει, καθώς και τα δεδομένα για το οποία επιθυμεί να ενημερώνεται ή να παρακολουθεί. Αυτά σχετίζονται, κυρίως με σφάλματα ή καταστάσεις στις οποίες βρίσκεται το σύστημα και μπορεί να δηλώνουν κάποιο πρόβλημα. Καταστάσεις όπως είναι η μειωμένη απόδοση του υπό μελέτη συστήματος 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 τοποθείται στην αρχή της ιστοσελίδας την οποία θέλουμε να παρακολουθήσουμε. Με αυτό τον τρόπο δίνεται η δυνατότητα συλλογής δεδομένων πραγματικών χρηστών.

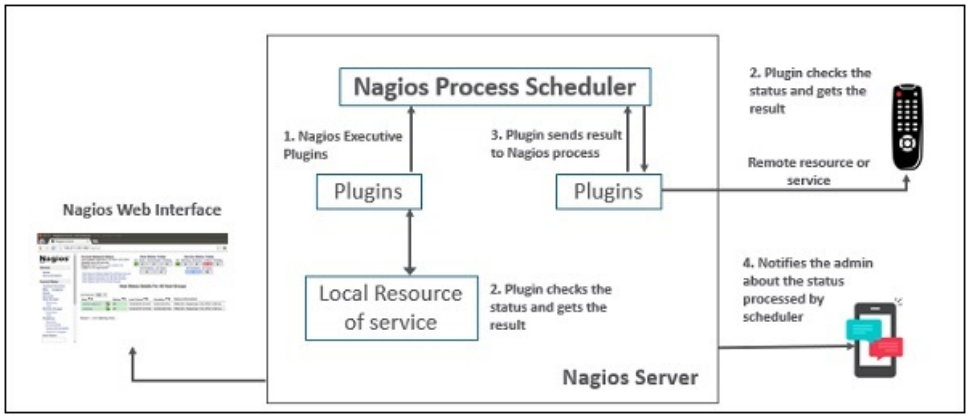


Σχήμα 3.4: Παράδειγμα χρήσης του εργαλείου Uptimia

Παραπάνω αναφέρθηκαν μερικά μόνο κάποια από τα εργαλεία που υπάρχουν σήμερα για την Ενεργη 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 projects, είτε ως projects μίας μεγαλύτερης ομάδας από developers. Έτσι και στο πλαίσιο της Ενεργής Παρακολούθησης μερικά από τα πιο γνωστά και δημοφιλή μεταξύ developers αποθετήρια αποτελούν τα:

- Upptime[1]: Αποτελεί μία ενδιαφέρουσα προσέγγιση στο πρόβλημα της διαχείρησης 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.



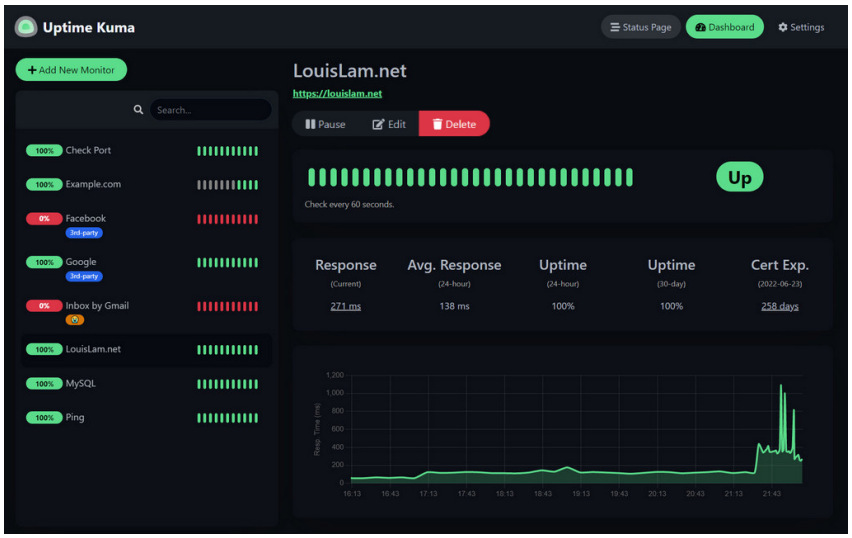
Σχήμα 3.5: Αρχιτεκτόνική Συστήματος Nagios

- Nagios[2]: Είναι ένα πρόγραμμα γραμμένο στη γλώσσα προγραμματισμού C. Πέρα από το backend διαθέτει και Γραφικό Περιβάλλον Χρήστη (Graphical User Interface - GUI). Αποτελείται από ένα σύστημα ενός server (nagios server), ο οποίος λειτουργεί σαν ένας scheduler που στέλνει σήματα για να ξεκινήσει

[1]αποθετήριο κώδικα Upptime: *https://github.com/upptime/upptime*
[2]αποθετήριο κώδικα Nagios: *https://github.com/NagiosEnterprises/nagioscore*

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 στο σχήμα 3.5

- Kuma Uptime[3]: Αποτελεί μία εύκολή, στη χρήση και στήσιμο, self-hosted εφαρμογή γραμμένη σε JavaScript για Ενεργή Παρακολούθηση. Για την αποστολή των αιτημάτων σε απομακρυσμένες (στο διαδίκτυο) σελίδες χρησιμοποιεί **child_proccesses**, ένα api δήλαδή που περιλαμβάνει η Node.js για τη δημιουργία διεργασιών (processes) εντός άλλων διεργασιών. Η εφαρμογή περιλαμβάνει backend και frontend, στο οποίο εμφανίζονται 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 θα θέλαμε να διαθέτει ιδανικά ένα σύστημα Παρακολούθησης, βάσει όλων αυτών που είδαμε μέχρι τώρα, είναι τα εξής:

1. Σταθερό και Αξιόπιστο σύστημα διαχείρησης 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]αποθετήριο κώδικα Kuma Uptime: *https://github.com/louislam/uptime-kuma*

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 απο 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. Απο 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

	Τρέχουσα Διπλωματική	**Better Uptime**	**Uptime Robot**	**Site24x7**	**Uptimia**	**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 και τους λόγους που μας ώθησαν 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 server, ώστε να εξυπηρετεί απομακρυσμένους χρήστες που συνδέονται μέσω του δικτύου. Το σύστημα επιπροσθέτως θα πρέπει να μπορεί να στέλνει αιτήματα σε εξωτερικούς servers, ιστοσελίδες και γενικά εφαρμογές που επικοινωνούν με το διαδίκτυο. Σε αυτό το σημείο, τονίζουμε την αναγκαιότητα ύπαρξης μίας βάσης δεδομένων που θα αποθηκεύει πληροφορία 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 "**L**ightweight **Y**et **C**onfigurable **H**TTP **T**raffic **E**xpert", που περιγράφει την λειτουργία και μόνο μερικά από τα πολλά χαρακτηριστικά του συστήματος μας.

4.1 Version 0.0.1

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

server), 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.

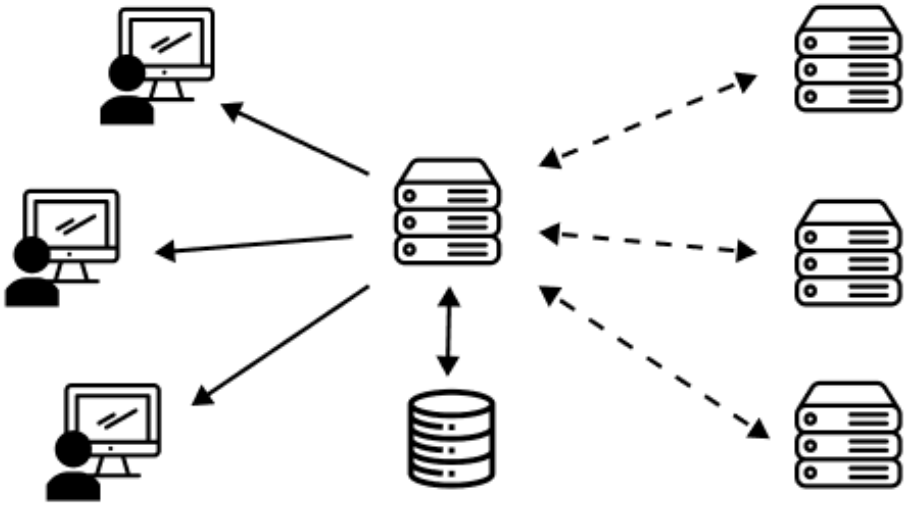
Ο server αρχικά στεγάζει το api της εφαρμογής. Μέσα από αυτό, κάθε χρήστης που έχει πρόσβαση στο σύστημα (σωστά credentials) μπορεί να αντλεί πληροφορία ήδη αποθηκευμένη στη βάση ή να την τροποποιεί και να δημιουργεί νέα. Τα δεδομένα που μπορεί να παράξει είναι περιορισμένα, βέβαια, καθώς υπάρχουν collections στα οποία γράφει μόνο το ίδιο το backend. Για να συνεχίσουμε και να δούμε το υπόλοιπο σύστημα θα πρεπει 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 ιστότοπου. 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 server υπό τη μορφή HTTP request. Αυτός χρησιμοποιεί το μήνυμα που έλαβε, που περιέχει το url και όποια ακόμα πληροφορία απαιτείται (headers, body), για να κάνει μία δοκιμαστική κλήση στο σύστημα που ο χρήστης έχει επιλέξει. Αν το αίτημα εκτελεστεί επιτυχώς, το αποθηκεύει στη βάση δεδομένων και ξεκινάει έναν απλό scheduler ο οποίος είναι υπεύθυνος για τον έλεγχου του συγκεκριμένου url.

Καταλήγουμε έτσι στη δεύτερη, μεγάλη, λειτουργία του server, το scheduling. Κάθε φορά που αποθηκεύεται ένα καινούργιο αίτημα για παρακολούθηση, ξεκινάει μία νέα διεργασία εντός αυτής που ήδη τρέχει (αυτής του server). 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 συλλέγονται, 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 server.

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 server, λειτουργούν αυτόνομα, χωρίς να υπάρχει κάποιος τρόπος να αναφερ-



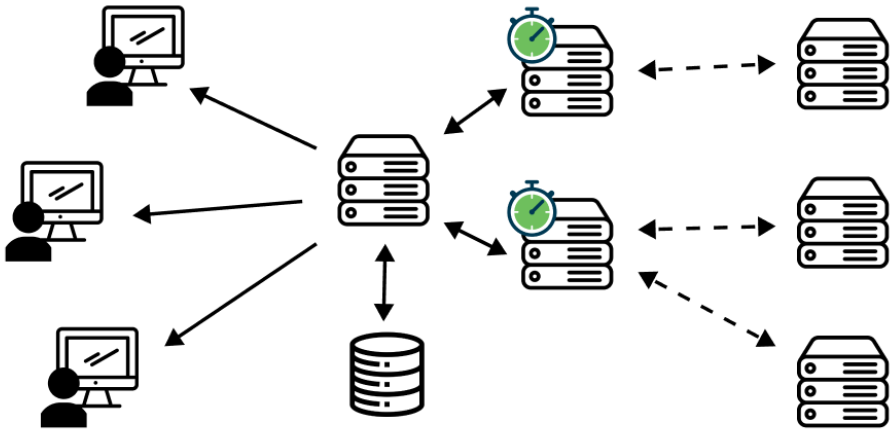
Σχήμα 4.1: Διάγραμμα Πρώτης Υλοποίησης Συστήματος Ενεργής Παρακολούθησης. Ένας express server, με μία mongo βάση δεδομένων εξυπηρετούν χρήστες, ενώ παράλληλα στέλνουν αιτήματα σε εξωτερικά συστήματα προς παρακολούθηση

θείς σε 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 server, καθώς οι καινούργιες διεργασίες θα είχαν διαφορετικό id. Οι ”καινούργιες διεργασίες” εδώ χαρακτηρίζουν ένα σύνολο διεργασιών, που θα ξεκινούσε ο server, 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 επεκταθεί το σύνολο των αιτημάτων που διαχειρίζονται, θα πρέπει να προστεθεί κι ένας δεύτερος server, που ίσως δεν χρειάζεται να επιτελεί κάποια λειτουργία, 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 server. Αυτό ακριβώς θα μελετήσουμε και στην επόμενη 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, τον διαμοιρασμό των λειτουργιών. Όπως προαναφέραμε θα πρέπει να υπάρχει ένας server που θα εξυπηρετεί τους χρήστες μέσω των προγραμματιστικών διεπαφών που παρέχει, και ένας ή παραπάνω schedulers, 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 server και μία mongo βάση δεδομένων. Ο server επικοινωνεί με δύο schedulers οι οποίοι είναι υπεύθυνοι για τον έλεγχο εξωτερικών προς το σύστημα εφαρμογών/server

Το βασικό συστατικό του Lychte, ο server δηλαδή παραμένει όπως είχε. Έχει δηλαδή routes μέσα από τα οποία εξυπηρετεί τα 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.

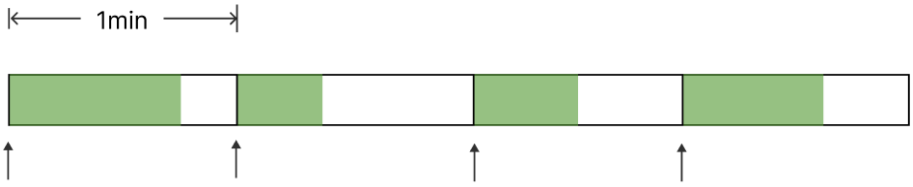
Αυτό που αλλάζει σε σχέση με την προηγούμενη υλοποίηση, είναι ο τρόπος που διευθετεί την εκτέλεση των επαναλαμβανόμενων αιτημάτων που πρέπει να κάνει προς τα εξωτερικά υπό παρακολούθηση συστήματα. Όπως είπαμε νωρίτερα, δεν είναι πλέον υπεύθυνος για το scheduling των ενεργειών που πρέπει να γίνουν, αλλά 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, βάσει αυτών, συνεχώς αι-

τήματα προς άλλα εξωτερικά συστήματα. Ο τρόπος λειτουργίας του είναι αρκετά παρόμοιος με αυτή του scheduler της πρώτης υλοποίησης, με κάποιες μικρές διαφορές. Κάθε φορά που λαμβάνει κάποιο αίτημα από τον κεντρικό server, κάνει ένα δοκιμαστικό request στο url που πρόκειται να ελεγχθεί, για να διαπιστωθεί η ορθότητά του. Έπειτα ξεκινάει έναν timer που εκτελεί το αίτημα στα χρονικά διαστήματα που ορίζει ο χρήστης. Ο χρόνος αυτός θα μπορούσε να φτάνει μέχρι και το ένα δευτερόλεπτο, 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 σχήματα 4.3 και 4.4.

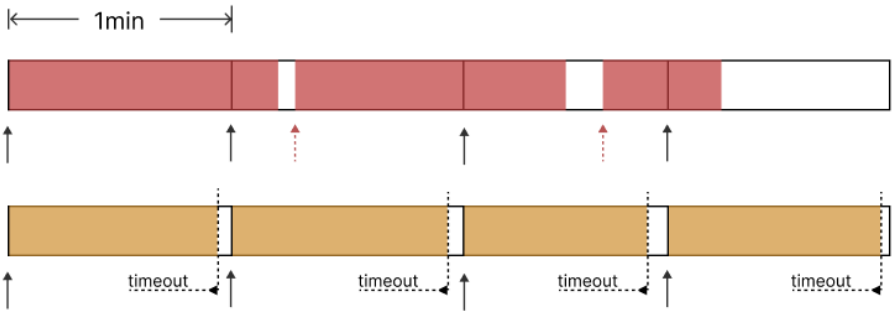
Το πρώτο περιγράφει την εκτέλεση ενός αιτήματος σε χρονικό διάστημα ενός λεπτού. Όπως παρατηρούμε το αίτημα εκτελείται επιτυχώς σε αυτό το μήκος χρόνου, με αποτέλεσμα, όταν έρθει η στιγμή που πρέπει να επαναεκτελεστεί, αυτό να γίνει χωρίς περιττές καθυστερήσεις. Αν υποθέσουμε ότι ο χρόνος που μεσολάβησε, μεταξύ του πρώτου και του δεύτερου αιτήματος, είναι ακριβώς εξήντα δευτερόλεπτα, μπορούμε να πούμε ότι η απόκλιση 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 timeous 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.



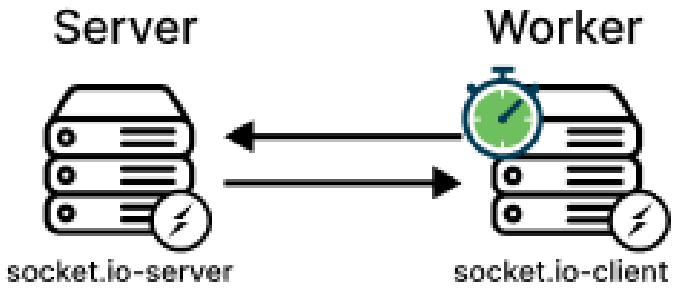
Σχήμα 4.4: Σύγκριση χρήσης ή μη timeouts στον κύκλο ζωής εκτέλεσης ενός αργού επαναλαμβανόμενου αιτήματος

Μία ακόμα αλλαγή ως προς την πρώτη υλοποίηση αφορά την επικοινωνία μεταξύ server 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. Οι λειτουργίες, όμως, αυτές, 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 server και worker, ώστε να μπορεί να ενημερώνεται ο server (όταν το ζητήσει) για την κατάσταση των διάφορων workers που είναι συνδεμένοι 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 ο server, θα ρωτούσε όλους τους συνδεδεμένους workers 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 server και ο socket.io-client οι workers. Κάθε φορά που θα ξεκινάει τη λειτουργία του ένας server, 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 στον 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 server είναι κυρίως οι ταχύτητες που προσφέρουν και η αμφίδρομη επκοινωνία. Υπό περιτπώσεις 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 server του δικού μας κλειστού συστήματος βλέπουμε ότι δεν έχει νόημα η δημιουργία ολόκληρου HTTP server στο πλαίσιο λειτουργίας ενός worker.



Σχήμα 4.5: Επικοινωνία server-worker μέσω websockets

Η παραπάνω υλοποίηση έχει πολλά πλεονεκτήματα σε σχέση με την πρώτη. Αρχικά με τον καταμερισμό των δύο βασικών λειτουργιών μειώνουμε 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 μηχάνημα 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 server 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.

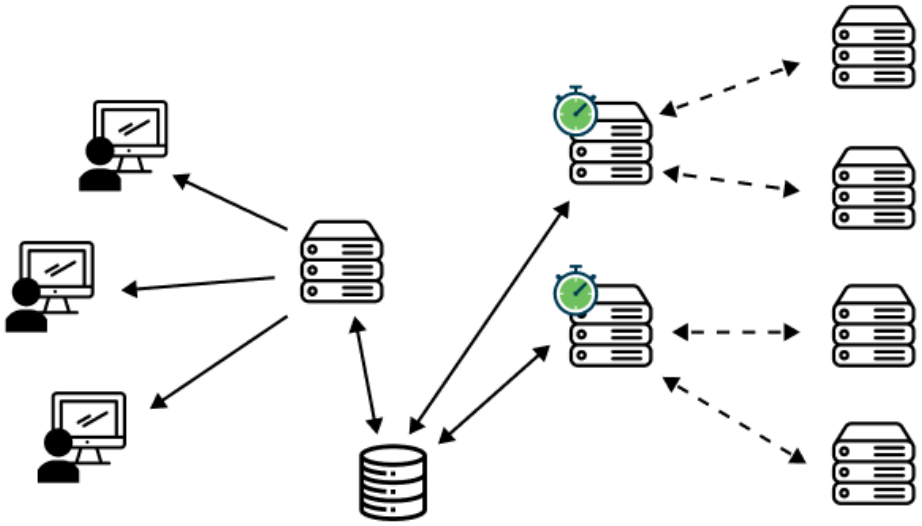
Ένας ακόμα τομέας στον οποίο συνεισφέρει σε μεγάλο βαθμό η διάσπαση των λειτουργιών αφορά την κλιμάκωση, και βασικά την οριζόντια κλιμάκωση του συστήματος. Αν δούμε ότι ο server υστερεί και δυσκολεύεται να ανταπεξέλθει στα

αιτήματα των χρηστών, μπορούμε να φτιάξουμε έναν καινούργιο server. Από την άλλη, αν παρατηρήσουμε ότι οι workers αργούν ή έχουν υπερφορτωθεί μπορούμε να δημιουργήσουμε νέους workers και να διαμοιράσουμε τα αιτήματα εξίσου στο πλήθος των ενεργών schedulers 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 υλοποίησής 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 server και κάθε worker του συστήματος συνδέονται με τη βάση χωρίς να υπάρχει άμεση επικοινωνία μεταξύ τους. Ο server 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 server είναι σχεδόν ίδια χωρίς το κομμάτι των websockets. Πλέον δεν υπάρχει άμεση επικοινωνία μεταξύ server 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.

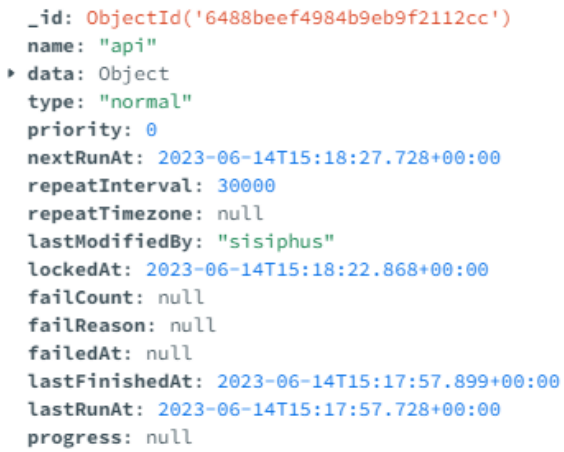
Πριν συνεχίσουμε στην επεξήγηση της ακριβής λειτουργίας του ανανεωμένου συστήματος, θα πρέπει να αναφερθούμε στους workers/schedulers. Η λειτουργία τους έχει εξελιχθεί σε σχέση με τις προηγούμενες εκδόσεις. Έχοντας πλέον ένα κοινό σημείο αναφοράς, τη Mongo βάση, μπορούμε να φτιάξουμε πιο ''έξυπνους'' schedulers. Στις άλλες υλοποιήσεις μας κάθε φορά που ερχόταν ένα νέο αίτημα (ένα νέο url που θα πρέπει να ελέγχουμε δήλαδή), 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 n δευτερόλεπτα και ψάχνει από τη βάση ποια αιτήματα πρέπει να εκτελεστούν. Πιο συγκεκριμένα, κάθε φορά που αποθηκεύουμε ένα αίτημα προς έλεγχο στη βάση, δημιουργούμε ένα ακόμα αντικείμενο που αφορά ένα job. Σε αυτό το αντικείμενο αποθηκεύονται πληροφορίες σχετικά με το αίτημα 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. Η λειτουργία αυτής είναι να ψάχνει στη βάση, και συγκεκριμένα στο collection απο το οποίο διαβάζει αποκλειστικά ο συγκεκριμένος worker, προκειμένου να βρει jobs που πρέπει να εκτελέστουν. Αν εντοπιστεί κάποιο του οποίου ο αποθηκευμένος χρόνος, που πρέπει να ξαναεκτελεστεί, έχει ξεπεραστεί χρονικά, σημαίνει ότι πρέπει να εκτελέσει το αίτημα, τα χαρακτηριστικά του οποίου έχει αποθηκευμένα. 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 σημεία 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. Ο χρήστης εισάγει κάποιο url το οποίο επιθυμεί να ελέγχεται ανά κάποιο συγκεκριμένο χρονικό διάστημα. Πέρα από την εισαγωγή της διεύθυνσης της



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

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 παραμετροποίησης μπορεί να καθορίσει την επιθυμητή κατάσταση απόκρισης του αιτήματος (εξ ορισμού **Status Code 200**) και ένα response body, βάσει του οποίου, αν το επιθυμει, 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.

2. Για να γίνει όμως η αποθήκευση θα πρέπει πρώτα να ξεκινήσει ένα post αίτημα στον server του συστήματός μας, στο σώμα του οποίου θα περιέχονται όλες οι επιλογές που έχει κάνει ο χρήστης. Πριν, όμως, ο server το αποθηκεύσει στη βάση, ψάχνει τον πιο ελεύθερο, από άποψη φόρτου, worker. Αυτό γίνεται μετρώντας τις καταχωρήσεις των collections που αντιστοιχούν σε κάθε έναν από αυτούς. Αφού εντοπιστεί και επιλεχθεί 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 τη ροή του συστήματος, μεταφερόμαστε στους workers και πιο συγκεκριμένα σε αυτόν που επιλέχθηκε στο προηγούμενο βήμα. Η λειτουρία αυτού του τμήματος του Lychte μελετήθηκε

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

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 server) γνωρίζει που πρέπει να απευθύνεται το αίτημα που εκτελεί, καθώς και τις παραμέτρους που ίσως αυτό χρειάζεται. Επίσης είναι υπεύθυνο δυνητικά για το έλεγχο της ορθότητας της απόκρισης του εξωτερικού συστήματος, μέσα από την σύγκριση της με τα στοιχεία που έδωσε ο χρήστης κατά τη δημιουργία του Api, καθώς και για την ενημέρωση του χρήστη σε περίπτωση αποτυχίας του αιτήματος, μέσω email, εφόσον ο ίδιος το επιθυμεί. 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 διήρκησε) 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

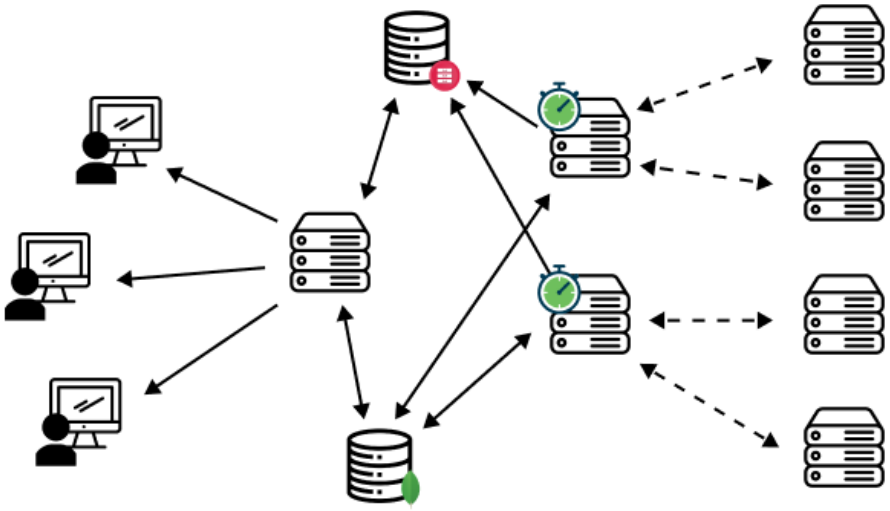
Σχήμα 4.9: Παράδειγμα εγγραφής **Response** στη βάση. Αριστερά βλέπουμε ένα επιτυχές response και δεξιά ένα που απέτυχε λόγω σφάλματος από τη μεριά του υπό παρακολούθηση συστήματος (το καταλαβαίνουμε από το πεδίο hasError που είναι true).

και μόνο 1.440 εγγραφές. Αν αυτά συνεχίζουν να μαζεύονται, τότε ακόμα και τα πιο απλά queries στο μοντέλο των αποκρίσεων θα καθυστερούν, καθιστώντας έτσι την εφαρμογή λιγότερο ανταποκρίσιμη. Για την αποφυγή συσσώρευσης περιττής, μετά από ορισμένο χρονικό διάστημα, πληροφορίας, έχουμε βάλει 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 ψάχνει τα Apis ανάλογα με τον worker στον οποίο τρέχει. Έπειτα μαζεύει όλες 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 ολα 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 άλλης μορφής βάση δεδομένων, η Google Cloud Storage.

Πριν σβήσουμε δεδομένα που κρίνουμε ότι δεν είναι απαραίτητο να υπάρχουν στη βάση, αλλά έχουν αξία για ιστορική ανάλυση, τα μεταφέρουμε 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.



Σχήμα 4.10: Τελικό διάγραμμα Lychte

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.

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 τη χρήση ενός διαφορετικού τύπου αρχείου σε σχέση με τον κλασσικό 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 πληροφορίας 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 avro 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 Εφαρμογής

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 καναμε 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 αρθμός των αιτημάτων που θέλουμε να ελέγχουμε. Στο σημείο αυτό πρέπει να ανεφερθεί ότι οι μετρήσεις έγιναν σε μηχάνημα 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

Προκειμένου να κάνουμε τις μετρήσεις χρειαζόμαστε κάποια urls τα οποία θα πρέπει να καλούμε ανά τακτά χρονικά διαστήματα, αυτά που κανονικά ορίζει ο χρήστης κατά τη διαδικασία δημιουργίας ενός Api. Για να υπάρχει μία ποικιλία στο πλήθος των ιστοσελιδών και διαδικτυακών apis που παρακολουθούμε και να μελετήσουμε τη λειτουργία του Lychte σε μία κατάσταση όσο το δυνατό πιο κοντά σε πραγματικά σενάρια, δημιουργήσαμε script που γράφει με τυχαίο τρόπο n Apis και Jobs, στα αντίστοιχα collections της βάσης, έχοντας μία προκαθορισμένη λίστα urls, μαζί με στοιχεία που ίσως χρειάζονται για αυθεντικοποίηση (tokens). Μαλιστα, για να ελέγξουμε τη σταθερότητα του συστήματος ως προς το χρόνο μεταξύ των διαδοχικών αιτημάτων ενός Api για διάφορους χρονισμούς, περά από την τυχαιότητα επιλογής του συνδέσμου στον οποίο θα κάνουμε κλήσεις επιλέγεται με τυχαίο τρόπο και ο χρόνος μεταξύ των αιτημάτων (το interval κάθε Api/Job). Επειδή σκοπός των μετρήσεων που κάναμε είναι το stress test του συστήματος, οι χρόνοι που επιλέχθηκαν είναι ένας μεταξύ των τριάντα δευτερολέπτων, ενός, δύο, ή τριών λεπτών.

Πιο συγκεκριμένα τα σενάρια που δοκιμάστηκαν είναι τα εξής:

- $n = 100$ apis
- $n = 500$ apis
- $n = 1.000$ apis
- $n = 5.000$ apis

Κάθε μία από τις παραπάνω διαφορετικές μετρήσεις διήρκησε δώδεκα ώρες. Σε αυτό το μήκος χρόνου, μετρήθηκαν η μέση χρήση/κατανάλωση CPU και RAM καθώς και η μέση απόκλιση του χρόνου μεταξύ των διάφορων διαδοχικών αιτημάτων από τον αναμενόμενο, βάσει του interval κάθε Api, χρόνο που θα έπρεπε να ξαναεκτελεστεί.

5.1.2 Αποτελέσματα

Στον πίνακα 5.2 μπορούμε να δούμε τα αποτελέσματα των μετρήσεών μας.

Παρατηρούμε ότι στα πρώτα τρία σενάρια η χρήση υπολογιστικών πόρων του συστήματος δεν είναι σημαντική, αλλά και η μέση απόκλιση του χρόνου επανεκτέλεσης των Apis είναι πολύ μικρή. Μάλιστα παίρνει αρνητικές τιμές. Αυτό δεν οφείλεται σε κάποιο λάθος στον υπολογισμούς μας, αλλά στον τρόπο λειτουργίας του scheduler. Για να κρίνει, αν πρέπει να εκτελέσει κάποιο αίτημα, έχει ένα μικρό bias που του επιτρέπει να επιλέγει jobs που βρίσκονται πολύ κοντά στο χρόνο που πρέπει να ξανατρέξουν. Η πόλωση αυτή του συστήματος είναι της τάξη nano-δευτερολέπτων και έχει σκοπό να κερδίζει χρόνο για εξομαλύνει πιθανές καθυστερήσεις του συστήματος.

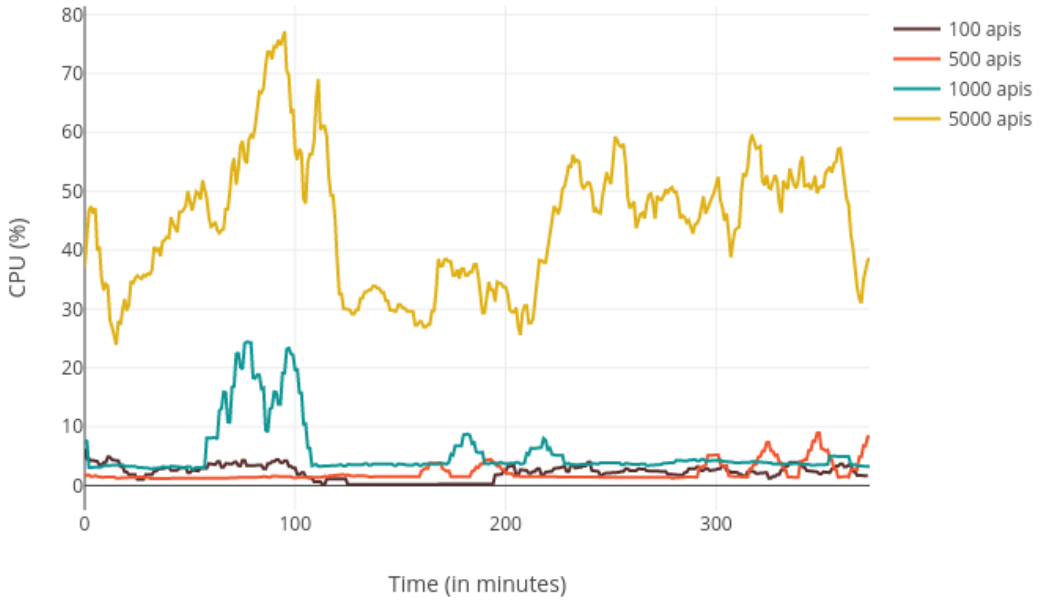
Στην τελευταία σειρά μετρήσεων, που αφορά τα 5.000 apis, που εκτελούνται ταυτόχρονα από έναν και μόνο worker, μπορούμε εύκολα να δούμε ότι το σύστημα συνεχίζει να ανταπεξέρχεται στις ανάγκες του ελέγχου και παρακολούθησης των

apis του (μέσος χρόνος απόκλισης μικρότερος του δευτερολέπτου). Αξίζει να σημειωθεί, όμως, όπως θα δούμε καλύτερα και στη συνέχεια ότι η κατανάλωση των υπολογιστικών πόρων είναι αρκετά μεγάλη, σε βαθμό μάλιστα που ιδανικά θα κάναμε load balancing με έναν από δύο τρόπους. Μεταφέροντας τα αιτημάτα που εκτελεί σε έναν ή περισσότερους άλλους workers, ή βάζοντας άλλους workers να διαβάζουν από το collection της υπό μελέτης διεργασίας.

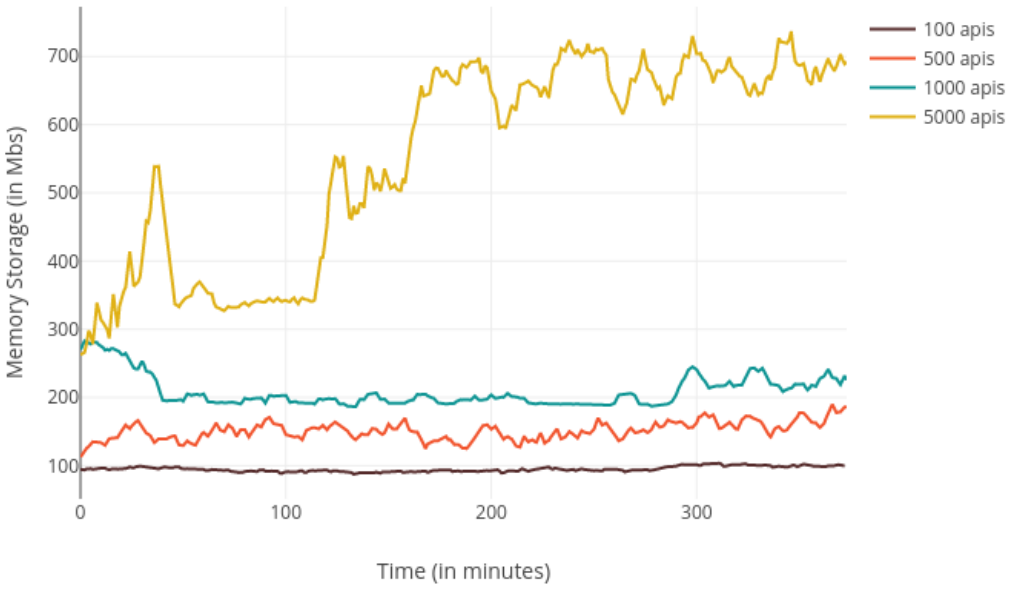
Πίνακας 5.2: Αποτελέσματα Μετρήσεων

	Μέση Απόκλιση	**Μέση Χρήση CPU**	**Μέση Χρήση RAM**
100 apis	-112,62ms	1,22%	102,54Mb
500 apis	-69,32ms	3,26%	171,1Mb
1.000 apis	-47,9ms	7,46%	223,13Mb
5.000 apis	664ms	56,36%	652,45Mb

Παρακάτω παρατίθενται διαγράμματα χρήσης cpu και ram αντίστοιχα για τις τελευταίες έξι ώρες λειτουργίας του Lychte στα τέσσερα σενάρια που αναφέραμε πιο πάνω. Λόγω της ευμετάβλητης φύσης των χαρακτηριστικών που μετράμε στο υπολογιστικό μας σύστημα, στα διαγράμματα παρουσιάζεται ο **Κινούμενος Μέσος Όρος** τους, με παράθυρο δέκα τιμών.



Σχήμα 5.1: Αποτελέσματα χρήσης της CPU.

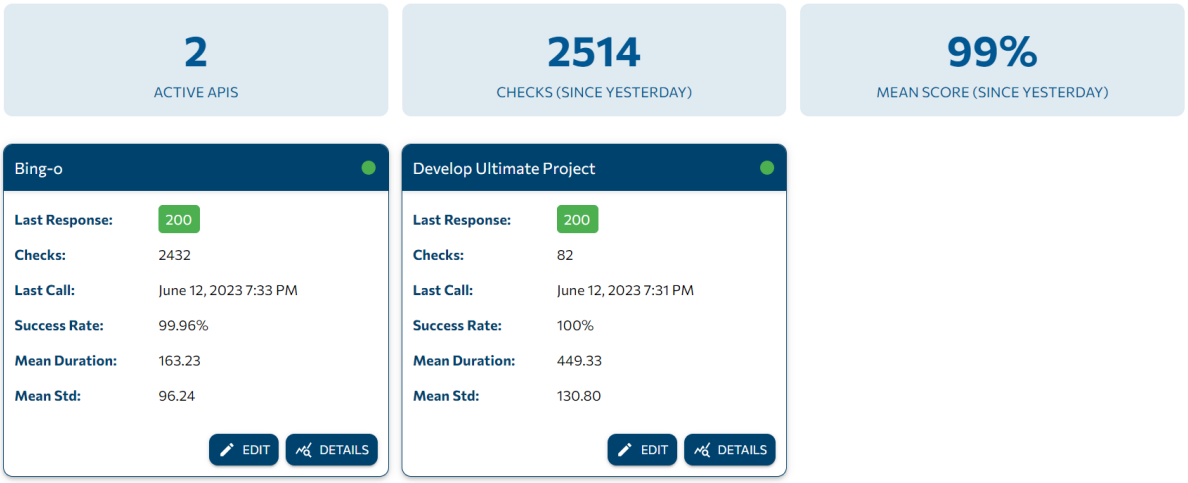


Σχήμα 5.2: Αποτελέσματα χρήσης της RAM

5.2 Επίδειξη Εφαρμογης

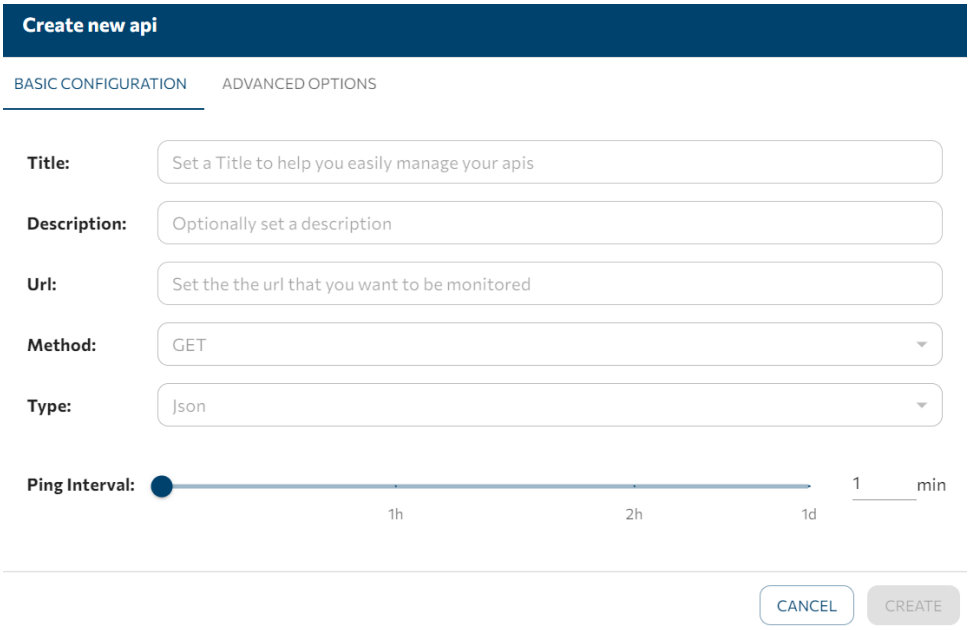
Στη συνέχεια παρατίθενται εικόνες από τη χρήση της γραφικής διεπαφής που υλοποιήσαμε, μέσω της οποίας οι χρήστες μπορούν να επικοινωνούν με το backend.

Στο σχήμα 5.3 μπορούμε να δούμε όλα τα Apis που έχουμε ενεργά, καθώς και κάποια στοιχεία σχετικά με αυτά. Η πληροφορία που δείχνουμε σε αυτήν την εικόνα, αφορά δεδομένα μέχρι μία ημέρα πριν. Αυτό γίνεται για να δώσουμε αρχικά μία γενική εικόνα των υπό μελέτη εξωτερικών συστημάτων, αλλά και να περιορίσουμε το πλήθος της πληροφορίας που θα πρέπει να ληφθεί υπόψη, ώστε να μπορεί να ανταποκρίνεται πιο γρήγορα και να είναι έτσι πιο αποδοτικά τα queries που κάνουμε στη βάση δεδομένων μας. Μερικά από τα στοιχεία που παρουσιάζονται αφορούν το πλήθος των ελέγχων που πραγματοποιήθηκαν εντός των τελευταίων εικοσιτεσσάρων ωρών, η μέση διάρκεια και η τυπική απόκλιση των αποκρίσεων των αιτημάτων που κάναμε, ο μέσος βαθμός επιτυχών αποκρίσεων και η κατάσταση του τελευταίου αποθηκευμένου ελέγχου.

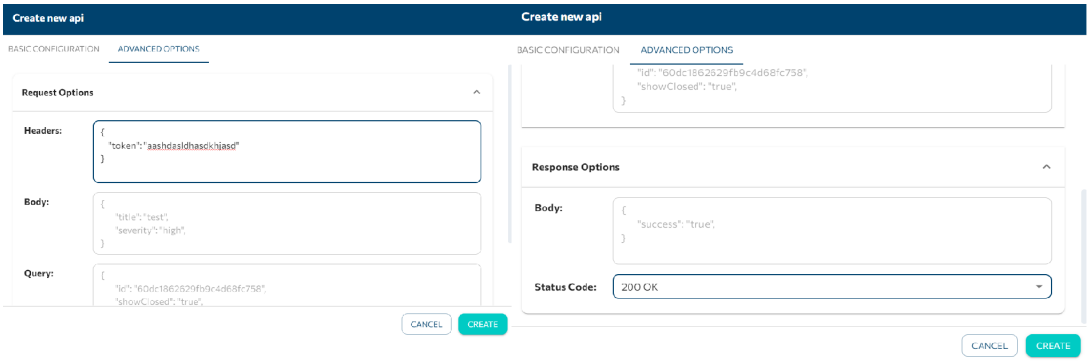


Σχήμα 5.3: Lychte Api Overview

Έπειτα στα σχήματα 5.4 και 5.5, φαίνονται τα στοιχεία που καλείται να συμπληρώσει η χρήστης για να φτιάξει κάποιο καινούργιο Api στο σύστημά μας και να ξεκινήσουν οι έλεγχοι.

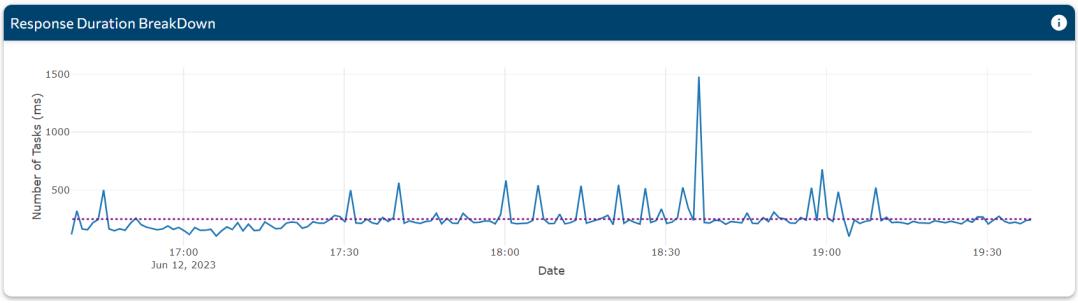


Σχήμα 5.4: Εικόνα Βασικών Στοιχείων προς συμπλήρωση από το χρήση

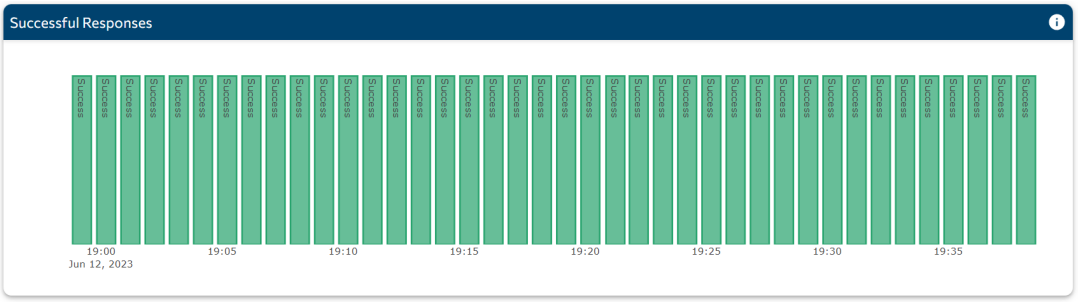


Σχήμα 5.5: Εικόνες Εισαγωγής ”Προηγμένων” στοιχείων για τη δημιουργία νέου ελέγχου

Στη συνέχεια, παρουσιάζονται διαγράμματα που μπορεί να δει ο χρήστης και σχετίζονται, όπως και πριν, με δεδομένα των τελευταίων εικοσιτεσσάρων ωρών. Τα διαγράμματα ανανεώνονται αυτόματα κάθε ένα λεπτό προκειμένου να δείχνουν πάντα την τελευταία έκδοση των δεδομένων που έχουμε αποθηκευμένα. Στο σχήμα 5.6 μπορούμε να δούμε το χρόνο που μεσολάβησε από τη στιγμή που γίνεται κάποιο αίτημα μέχρι να ανταποκριθεί το υπό μελέτη σύστημα, σε βάθος χρόνου ημέρας, μπορούμε, ωστόσο, να επιλέξουμε αν θελουμε να φιλτράρουμε τα δεδομένα στις τελευταίες τρεις, έξι και δώδεκα ώρες. Με μία διακεκομμένη γραμμή αναπαριστούμε το μέσο ώρο του χρόνου που μετρήσαμε. Τέλος, όσον αφορά, τα δεδομένα της τελευταίας μέρας υπάρχει και το ραβδόγραμμα που φαίνεται στο σχήμα 5.7, στο οποίο μπορούμε να δούμε την κατάσταση των τελευταίων πενήντα αποκρίσεων.



Σχήμα 5.6: Διάγραμμα Διάρκειας Αποκρίσεων

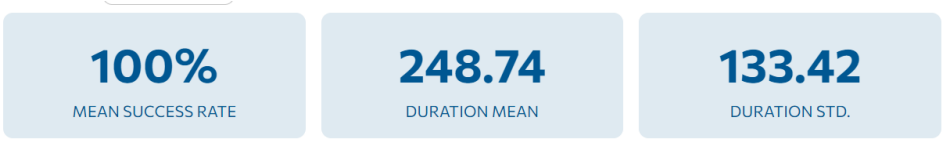


Σχήμα 5.7: Ραβδόγραμμα Κατάστασης Αποκρίσεων

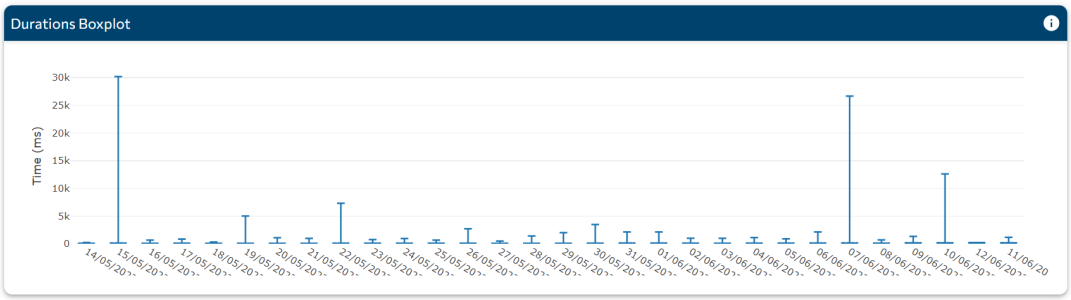
Τέλος, θα δείξουμε διαγράμματα και μετρικές που προκύπτουν από το σύνολο των δεδομένων που υπάρχουν τόσο στη Mongo βάση υπό μορφή documents, όσο και στη Google Cloud βάση, υπό τη μορφή αρχείων. Πιο συγκεκριμένα, αξιοποιούμε τα ήδη υπολογισμένα δεδομένα που υπάρχουν σε αρχεία τις μορφής "**/:apiId/statistics.avro**". Στα δεδομένα που αντλούμε από τα προαναφερθέντα αρχεία, ενσωματώνουμε όσα δεν έχουν αποθηκευτεί ακόμα εκεί και βρίσκονται στη MongoDB. Κάθε φορά που γίνονται οι υπολογισμοί αυτοί, και για να γλιτώσουμε τέτοιους συνεχείς υπολογισμούς, αποθηκεύουμε όσα στοιχεία χρειαζόμαστε σε μία νέα συλλογή δεδομένων στην κύρια βάση μας (ονομασία **Statistics**), που αποθηκεύει στατιστικά. Κάθε φορά που φορτώνει η συγκεκριμένη σελίδα, ελέγχει αν υπάρχει καταχώρηση για το συγκεκριμένο Api στο νέο αυτό collection εντός ενός χρονικού διαστήματος (τριάντα λεπτών). Αν υπάρχει, αντλεί τα δεδομένα από εκεί. Αλλιώς, κάνει τους

απαραίτητους υπολογισμούς και στο τέλος αποθηκεύει τα αποτελέσματα εκεί, με αποτέλεσμα τα επόμενα, χρονικά κοντινά, αιτήματα να ικανοποιούνται πιο γρήγορα.

Στα σχήματα 5.8 και 5.9 παρουσιάζονται μετρικές που υπολογίζονται στο σύνολο των ιστορικών δεδομένων και ένα διάγραμμα Θηκογραμμάτων της διάρκειας απόκρισης των ελέγχων που πραγματοποιήθηκαν, ομαδοποιημένα σε διαστήματα ημέρας.



Σχήμα 5.8: Μετρικές που αφορούν το σύνολο των δεδομένων (ιστορικά δεδομένα)



Σχήμα 5.9: Διάγραμμα Θηκογραμμάτων (Boxplots) της διάρκειας απόκρισης αιτημάτων για κάθε μέρα (ιστορικά δεδομένα)

6 Συμπεράσματα και Μελλοντικές Επεκτάσεις

6.1 Συμπερασματα

Συνοψίζοντας, παρουσιάζονται συνοπτικά οι υλοποιήσεις μας, τα πλεονεκτήματα και τα μειονεκτήματα κάθε μίας.

- **1η Υλοποίηση**: Το ίδιο μηχάνημα λειτουργεί σαν server και scheduler ταυτόχρονα. Κάθε καινούργιο αίτημα προς παρακολούθηση ξεκινάει μία διεργασία εντός του server που εκτελείται επαναληπτικά, με ρυθμό ένα συγκεκριμένο χρονικό διάστημα. Η υλοποίηση αυτή λειτουργεί, αλλά, εμφανίζει προβλήματα:
  - **οριζόντιας κλιμάκωσης** λόγω της εξάρτησης που υπάρχει μεταξύ server και worker
  - **απόδοσης**, καθώς το φορτίο υπό το οποίο μπορεί να βρίσκεται ο server επηρεάζει τη λειτουργία των jobs που εκτελούνται στο παρασκήνιο
  - έλλειψης ενός **κοινού σημείου αναφοράς**.
- **2η Υλοποίηση**: Το σύστημα χωρίζεται σε δύο ανεξάρτητες οντότητες. Πλέον, υπάρχει, στην πιο απλή εκδοχή του συστήματος, ένας server και περισσότεροι από ένας workers. Ο server στεγάζει το API της εφαρμογής και είναι υπεύθυνος για την επικοινωνία των χρηστών της εφαρμογής με το σύστημά μας και οι workers εκτελούν αποκλειστικά ελέγχους προς τα εξωτερικά συστήματα που ορίζουν οι χρήστες. Η επικοινωνία μεταξύ server και workers σε αυτή την υλοποίηση είναι αμφίδρομη, καθώς αξιοποιούμε το πρωτόκολλο των Websockets. Έτσι ο server μπορεί να στέλνει αιτήματα (προς έλεγχο) στους workers, και οι workers με τη σειρά τους μπορούν να ενημερώνουν για την

κατάσταση τους (πλήθος ενεργών αιτημάτων που διαθέτουν), τον server. Το βασικό μειωνέκτημα της συγκεκριμένης υλοποίησης είναι η έλλειψη ενός κοινού σημείου αναφοράς. Ενός σημείου μέσω του οποίου θα μπορεί να ελεχθεί το σύνολο του συστήματος χωρίς να χρειάζεται να υπάρχει επικοινωνία μεταξύ των οντοτήτων του Lychte.

- **3η και Τελική Υλοποίηση**: Υιοθετεί αρκετές αρχές της προηγούμενης υλοποίησης. Server και workers αποτελούν ανεξάρτητες οντότητες. Πλεόν, όμως, η επικοινωνία αυτών δεν είναι άμεση, αλλά το ρόλο του μεσολαβητή αναλαμβάνει η βάση δεδομένων. Ο server επικοικωνεί με τους workers δημιουργώντας αντικείμενα σε συγκεκριμένα collections ή τροποποιώντας τα κατάλληλα, όταν γίνεται κάποια αντίστοιχη ενέργεια από τους χρήστες. Οι workers λειτουργούν πιο έξυπνα, αποφεύγοντας την εκτέλεση αχρείαστων timers και schedulers, κρατώντας μόνο όσους από αυτούς χρειάζεται για την ορθή λειτουργία του. Επίσης, εισάγεται μία ακόμα βάση δεδομένων στο σύστημα, σκοπός της οποίας είναι η αποθήκευση ιστορικών δεδομένων σε μορφή αρχείων. Έτσι κάποια δεδομένα μπορούν να διαγραφούν από την κεντρική βάση και να μεταφερθούν σε αυτή, για να κάνουν την πρώτη πιο αποδοτική και γρήγορη στην ανταπόκριση.

Καταλήγοντας στην τελική υλοποίησή μας, κάναμε κάποιες μετρήσεις, για να ελέγξουμε την απόδοση και την αξιοπιστία του συστήματος, και κυρίως της λειτουργίας των workers. Ξεκινώντας με ένα σχετικά μικρό πλήθος αιτημάτων που τρέχουν παράλληλα (100 στο πλήθος), και αυξάνοντας τα σταδιακά (φτάσαμε στα 5.000), είδαμε ότι η μέση απόκλιση του αναμενόμενου χρόνου εκτέλεσης των διάφορων αιτημάτων παραμένει χαμηλή, φτάνοντας μάλιστα σε ορισμένες περιπτώσεις σε αρνητικές τιμές. Όσον αφορά στην κατανάλωση πόρων του υπολογιστικού συστήματος, παρατηρούμε ότι αυξάνοντας το πλήθος των αιτημάτων που εκτελούνται παράλληλα από έναν worker, αυξάνονται, μετά από ένα ορισμένο σημείο, κατά πολύ οι απαιτήσεις του συστήματος. Αυτό μας οδηγεί στο συμπέρασμα, ότι πάντα θα πρέπει να υπάρχουν διαθέσιμοι workers στους οποίους θα μπορεί να γίνεται καταμερισμός των ενεργών αιτημάτων, ώστε να αποφεύγονται καταστάσεις υπερφόρτωσης του συστήματος.

6.2 Μελλοντικες Επεκτασεις

Όπως είδαμε, το σύστημα Lychte που υπολοποιήσαμε στο πλαίσιο της διπλωματικής αυτής εργασίας δουλεύει σε ένα αρκετά ικανοποιητικό επίπεδο, με τα αποτελέσματα των μετρήσεων που κάνουμε να το αποδεικνύουν.

Υπάρχει όμως ακόμα χώρος για βελτίωση. Όπως είδαμε, όταν το πλήθος των εξωτερικών συστημάτων που θέλουμε να ελέγχουμε αυξάνεται θα πρέπει να αυξάνεται αντίστοιχα και το πλήθος των workers που λειτουργούν προς την εξυπηρέτηση αυτών. Μία λύση στο πρόβλημα αυτό, και για την αποφυγή ύπαρξης καταστάσεων που ένας worker δυσλειτουργεί ή το υπολογιστικό σύστημα στο οποίο ζει δεν μπορεί να ικανοποιήσει τις ανάγκες του ως προς τους πόρους που απαιτεί, είναι η αυτόματη ενημέρωση κάποιου admin, που διαχειρίζεται το σύστημα, προκειμένου

να δημιουργήσει και άλλους worker. Είναι σημαντικό ακόμα να ερευνηθεί στο πλαίσιο αυτό το κομμάτι της κύριας βάσης δεδομένων που χρησιμοποιούμε. Η παρούσα υλοποίηση αξιοποιεί τις δυνατότητες της MongoDB. Υπάρχουν όμως πλήθος άλλων noSql βάσεων, όπως για παράδειγμα του Redis, η χρήση του οποίου θα μπορούσε να βελτιώσει την απόδοση και την ταχύτητα των queries στη βάση. Ακόμα, θα μπορούσε να ερευνηθεί το κατά πόσο θα είχε νόημα ο συνδυασμός μίας κύριας και μίας δευτερεύουσας βάσης (πέραν αυτής του Google Cloud Storage), που θα χρησιμοποιούνται από τις δύο διαφορετικές οντότητες του συστήματός μας, προκειμένου να διαμοιραστεί ο φόρτος των λειτουργιών που κατά βάση εκτελούν οι workers.

Ένας ακόμα τομέας στον οποίο υστερεί το σύστημα, είναι το περιορισμένο πλήθος των τύπων ελέγχου που μπορούμε να κάνουμε. Σε μελλοντική έκδοση, θα μπορούσαμε να ενσωματώσουμε αιτήματα ελέγχου PORTS, TCP συνδέσεων, αποστολής μηνυμάτων μέσω του πρωτοκόλλου επικοινωνίας MQTT, καθώς και ελέγχους συνδέσεων και αποστολής μηνυμάτων σε socket.io server. Επιπλέον, θα μπορούσαμε να επεκτείνουμε τα μέσα ενημέρωσης (σε περίπτωση προβλήματος) των χρηστών. Το τωρινό σύστημα παρέχει δυνατότητες ενημέρωσης μέσω mail, αλλά θα μπορούσαμε ακόμα να επιτρέψουμε τη σύνδεση άλλων μορφών επικοινωνίας, όπως είναι οι διάφορες messaging εφαρμογές (Slack, Discord).

Πέρα από τη βελτίωση του backend του συστήματος, θα μπορούσαμε ακόμα να προσθέσουμε παραπάνω διαγράμματα, αξιοποιώντας περαιτέρω τα αποθηκευμένα δεδομένα μας και δείχνοντας παραπάνω χρήσιμη πληροφορία στο χρήστη.

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.