



INTERNATIONAL
HELLENIC
UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF INFORMATION
AND ELECTRONIC ENGINEERING

THESIS PROJECT

«Study and development of authorization techniques
for real-time IoT REST APIs»



Customer



Access Token

Student
Savvas Mantzouranidis
Reg. Number: 174883

Supervisor
Periklis Chatzimisios
Professor

September 14, 2021

Thesis Title: Study and development of authorization techniques for real-time IoT REST APIs

Thesis Number: 21221

Student Name: Savvas Mantzouranidis

Supervisor Name: Periklis Chatzimisios

Undertaking Date: 30/03/2021

Completion Date: 14/09/2021

I certify that I am the author of this thesis and that any assistance I have received while preparing it is fully acknowledged and mentioned in the following sections. I have also listed any sources from which I have used data, ideas, images and text, whether they are referenced as is or paraphrased. In addition, I certify that this thesis was prepared by me personally, especially as a diploma thesis, in the Department of Information and Electronic Engineering of IHU.

This work is the intellectual property of the student Savvas Mantzouranidis who prepared it. In the context of the open access policy, the author/creator grants the International Hellenic University the right of reproduction, lending, presentation to the public and digital dissemination of the project internationally, in digital form and in any medium, for teaching and research purposes, with no benefits. Open access to the full text of the work does not in any way imply any intellectual property rights of the author/creator; nor does it allow reproduction, republishing, copying, sale, commercial use, distribution, publication, downloading, uploading, translation, modification in any way, in part or in summary of the work, without the explicit prior written consent of the author/creator.

The approval of this thesis by the Department of Information and Electronic Engineering of the International Hellenic University does not necessarily imply acceptance of the author's views by the Department.

«Dedicated to my grandfather Savvas, in loving memory.»

Prologue

Undertaking this thesis was an obsolete choice that I made with the help of my supervisor. The main subject selected is closely related with the existing and, potentially, future security solutions applied on Multi-Access Edge Computing (MEC) IoT applications. Working on this particular thesis provided me with knowledge that I did not have before, required me to get familiar with state-of-the-art development libraries, tools and equipment, made me realize how enterprise frameworks are designed and built and understand the potential challenges and complications that exist in this field.

The thesis is structured in two main parts, the first of them being the Literature Review where I will be analyzing all the required information in order to have an overall understanding of the main topic. The second part consists of two sections where I will be providing technical details regarding the implementation of the main subject and its integration on a framework running on a private 5G network. Finally, I will be presenting the development environment (testbed) that was used for this project as well as the final results of the experiments that were conducted.

Abstract

Security is definitely one of the most important aspects of IoT applications. In order to accomplish acceptable security levels and system integrity, multiple protocols ought to be combined. In cases like MEC, the threats existing in an IT ecosystem are constantly increasing, thus requiring regular security revisions and implementation of new techniques. Authorization is probably one of the first security measures that should be considered for such scenarios, as it offers mechanisms for controlling each component's access rights for different use cases; this is commonly referred to as "delegated authorization". Although authorization sounds ideal, there are still some vulnerabilities which make the system prone to threats. In order to avoid such situations, a security developer must be constantly informed about the latest attack trends and market challenges so as to act accordingly and apply the needed security updates.

OAuth 2 consists of some core actors and components which all work together and form the overall security model. For the purposes of understanding what authorization is capable of, this thesis demonstrates how the OAuth 2 system was applied on an existing framework functioning on a private 5G network. The requirements of this assignment consist of specific configuration setups in order to enhance security levels while keeping the whole framework's performance as good as possible. The combination of OAuth 2 and Transfer Layer Security (TLS) is mandatory if one desires to exploit HTTPS communications between the various endpoints without risking data leakage. With the usage of the correct tools and configuration, as well as with integration of features like caching and TLS resumption, the authorization mechanism can be applied on any system with minimal impact on the overall performance.

Σχεδίαση και ανάπτυξη τεχνικών εξουσιοδότησης για IoT REST APIs πραγματικού χρόνου

Μαντζουρανίδης Σάββας

Περίληψη

Η ασφάλεια είναι σίγουρα μια από τις πιο σημαντικές πτυχές των εφαρμογών του Διαδικτύου των Πραγμάτων (Internet of Things - IoT). Προκειμένου να επιτευχθούν αποδεκτά επίπεδα ασφάλειας και ακεραιότητας συστήματος, θα πρέπει να συνδυαστούν πολλά πρωτόκολλα. Σε περιπτώσεις εφαρμογών τύπου Multi-Access Edge Computing (MEC), οι απειλές που υπάρχουν σε ένα πληροφοριακό οικοσύστημα αυξάνονται συνεχώς, απαιτώντας συνεχή αναθεώρηση των τακτικών ασφάλειας και εφαρμογή νέων μέτρων. Οι μηχανισμός εξουσιοδότησης αποτελεί ένα από τα πρώτα μέτρα που πρέπει να ληφθούν υπόψη για τέτοια σενάρια, καθώς προσφέρει εργαλεία για τον έλεγχο των δικαιωμάτων πρόσβασης κάθε συσκευής σε διαφορετικές περιπτώσεις χρήσης, γνωστό ως "άδεια κατ' εξουσιοδότηση". Αν και η εξουσιοδότηση ακούγεται ιδανική, εξακολουθούν να υπάρχουν ορισμένες ευπάθειες που καθιστούν το σύστημα επιρρεπές σε απειλές. Προκειμένου να αποφευχθούν τέτοιες καταστάσεις, ένας προγραμματιστής ασφαλείας πρέπει να ενημερώνεται συνεχώς για τις πρόσφατες τάσεις επιθέσεων και προκλήσεις της αγοράς, ώστε να ενεργεί αναλόγως και να εφαρμόζει τις απαραίτητες ενημερώσεις στο σύστημα. Το OAuth 2 αποτελείται από ορισμένους βασικούς παράγοντες και αντικείμενα τα οποία λειτουργώντας όλα μαζί αποτελούν ένα συνολικό μηχανισμό εξουσιοδότησης. Για τους σκοπούς της κατανόησης των δυνατοτήτων της εξουσιοδότησης, αυτή η πτηνυακή εργασία καταδεικνύει πώς εφαρμόστηκε ο μηχανισμός OAuth 2 σε ένα υπάρχον σύστημα, που λειτουργεί πάνω σε ένα ιδιωτικό δίκτυο 5G. Οι απαιτήσεις αυτής εργασίας περιλαμβάνουν συγκεκριμένες ρυθμίσεις διαμόρφωσης προκειμένου να βελτιωθεί η ασφάλεια διατηρώντας, παράλληλα, όσο το δυνατόν καλύτερες αποδόσεις συστήματος. Ο συνδυασμός OAuth 2 και TLS είναι υποχρεωτικός εάν κάποιος επιθυμεί να εκμεταλλευτεί το πρωτόκολλο ασφαλείας HTTPS για τις επικοινωνίες μεταξύ των διαφόρων τελικών σημείων, έτσι ώστε να αποφευχθεί η πιθανή διαρροή δεδομένων. Με τη χρήση των σωστών εργαλείων και διαμόρφωσης, καθώς και με την ενσωμάτωση χαρακτηριστικών όπως η προσωρινή αποθήκευση και η διατήρηση των TLS sessions, η εξουσιοδότηση μπορεί να εφαρμοστεί σε οποιοδήποτε σύστημα με ελάχιστο αντίκτυπο στη συνολική απόδοση.

Acknowledgements

I would like to thank my family and friends for the motivation and mental support they have generously provided me and my supervisor Prof. Periklis Chatzimisios for his excellent mentoring and academic support. Moreover, I must express my gratitude to University of Bristol, and specifically to my Erasmus supervisor Alex Mavromatis for his professionalism, kindness and for the opportunity he gave me to work on such an amazing environment with such an amazing team, and my tech mentor Antonis Vafeas who helped and guided me every single day for the past six months in order to enrich my software development skills. Lastly, I would like to thank my colleague and academic brother, Klodian Bardhi, for the countless hours of debugging together and supporting each other through this amazing journey.

Table of Contents

Prologue	iv
Abstract	v
Περίληψη	vi
Acknowledgements	vii
Table of Contents	viii
List of Figures	x
List of Tables	x
Abbreviations	xi
1 Introduction	1
1.1 Significance of the Study	2
1.1.1 Research Statement	3
1.1.2 Topic Analysis	3
1.2 Contributions	4
1.3 Thesis outline	5
2 Literature Review	7
2.1 Introduction to IoT security	7
2.1.1 Evolution of IoT Security	8
2.1.2 Challenges	9
2.2 MEC Security	10
2.3 Authorization Basics	11
2.3.1 The Idea	11
2.3.2 Terminology	12
2.3.3 Different Authorization Techniques	13
2.4 OAuth 2.0 Framework - RFC 6749	17
2.4.1 Actors	17
2.4.2 Access Token	18
2.4.3 Scopes	20
2.4.4 Refresh Token	20
2.4.5 Token Introspection	21
2.4.6 Token Revocation	21
2.4.7 Authorization Code Grant	22
2.4.8 Client Credentials Grant	24
2.4.9 Device Code Grant	24
2.4.10 Legacy: Implicit Flow Grant	25
2.4.11 Legacy: Resource Owner Password Grant	25
2.5 Vulnerabilities	27
2.5.1 CSRF Attack	27
2.5.2 Theft of authorization codes	28
2.5.3 Theft of tokens	28
2.6 Conclusions and Thesis Motivation	29
3 Implementation and Integration of the OAuth 2 protocol on 5G MEC	31
3.1 Architecture Design and Framework Components	31
3.1.1 Users	31
3.1.2 Dashboard	32
3.1.3 Identity Manager	32
3.1.4 Certificate Manager	33
3.1.5 OAuth 2 Provider	33
3.1.6 Database	34
3.1.7 Middleware API	35
3.1.8 Data Collection REST API	36
3.1.9 IoT Devices	36
3.1.10 Containerized Network and Load Balancer	36
3.2 Challenges	38
3.3 Authorization Integration	39

3.3.1	ORY Hydra	39
3.3.2	Configuration	40
3.3.3	Login and Consent	40
3.4	Integrating authorization with other components	41
3.4.1	Hydra & Dashboard	42
3.4.2	Hydra & Docker	42
3.4.3	Hydra & NGINX	43
4	Performance Evaluation	44
4.1	Testbed Details	44
4.1.1	Requirements	44
4.1.2	Design	46
4.1.3	Private 5G Networking	48
4.2	Device Emulator	48
4.2.1	Python Threading	49
4.2.2	Configuration	49
4.2.3	Preparation Scripts	50
4.2.4	Supervisor & Device Components	51
4.2.5	Measuring the performance	52
4.2.6	Other Applications	53
4.3	Experiment Details	53
4.3.1	Experiment Parameters	53
4.4	Experiment Results	56
4.4.1	Authorization Results	56
4.4.2	TLS Session Results	59
4.4.3	Redis Integration Results	60
4.4.4	Reliability - Packet Loss	61
4.4.5	API Configuration Results - Interoperability	62
5	Conclusion	64
5.1	Results Analysis	65
5.2	Summary of Contributions	66
5.3	Future Work - Recommendations	67
	BIBLIOGRAPHY	69
A	Device Emulator Code	75
A.1	Supervisor	75
A.1.1	Object Structure	75
A.1.2	Create Devices	75
A.1.3	Main Function	76
A.2	Device	76
A.2.1	Object Structure	76
A.2.2	Send Message	77
A.2.3	Main Function	78
B	Database Samples	79
B.1	ORY Kratos	79
B.1.1	Identity Schema	79
B.2	OCB	79
B.2.1	IoT Device Stored Data	79
B.3	Framework's MongoDB	80
B.3.1	User Token Info	80
B.3.2	User-Device-Token Association	80

List of Figures

1.1	Man in the middle attack	2
2.1	The evolution of IoT	7
2.2	MEC deployment scenarios	8
2.3	5G network and potential threats	10
2.4	Yelp’s legacy resource owner password grant authorization	12
2.5	Access control lists	14
2.6	RBAC model	14
2.7	ABAC use case example	15
2.8	PBAC model	16
2.9	OAuth 2 actors	18
2.10	Access token use in request header	19
2.11	Access token as query parameter	19
2.12	Refresh token basic usage	21
2.13	Token introspection result	22
2.14	Authorization code as URL query	22
2.15	Authorization code grant flow	23
2.16	Client credentials grant flow	24
2.17	Device code grant flow	25
2.18	Implicit grant flow	26
2.19	Resource owner password grant flow	27
2.20	CSRF attack	28
2.21	Authorization code steal	29
2.22	Hijacking of access token through the query parameter	30
3.1	Framework’s architecture diagram	32
3.2	Browser status before and after implementing custom SSL certificates	33
3.3	OAuth 2 token usage and handling	34
3.4	Initiating a redis connection in the middleware API	36
3.5	Docker engine architecture	37
3.6	NGINX architecture example	38
3.7	Embedding user’s unique ID inside the access token	41
3.8	NGINX auth_request configuration	42
3.9	Subrequest function in NGINX javascript file	43
4.1	Bristol University testbed equipment	45
4.2	Part of the testbed’s development environment	46
4.3	Outdoor and indoor 5G network equipment	48
4.4	PDE’s JSON configuration file example	50
4.5	PDE’s preparation script operations	51
4.6	CSV log file example	52
4.7	Custom Poisson-like distribution	55
4.8	Experiment D’s performance without TLS resumption and redis caching	57
4.9	Authorization impact: Comparison between experiments A & B	58
4.10	Token refresh visualization	59
4.11	RTT measurements with and w/out TLS resumption	60
4.12	Experiment C’s RTTs with and w/out redis caching	61
4.13	API Config impact: Comparing experiments B & C	63

List of Tables

2.1	Authorization models characteristics	16
4.1	PDE’s configuration parameters	50
4.2	Device possible events and status codes	52
4.3	Experiment parameters	54
4.4	The impact of authorization	57
4.5	Normal transmission vs re-transmission after token refresh	58
4.6	Impact of TLS resumption on experiments C & D	60
4.7	Impact of redis caching on experiment C	61

Abbreviations

ACE	Authentication and Authorization for Constrained Environments
AP	Access Point
CA	Certificate Authority
CORS	Cross Origin Resource Sharing
CSRF	Cross-Site Request Forgery
CSV	Comma-Separated Values
DoS	Denial of Service
GIL	Global Interpreter Lock
IoT	Internet of Things
KPI	Key Performance Indicator
LDAP	Lightweight Directory Access Protocol
MEC	Multi-Access Edge Computing
MNO	Mobile Network Operators
NFV	Network Function Virtualization
OCB	Orion Context Broker
PDE	Python Device Emulator
ppm	Packages Per Minute
RTT	Round Trip Time
SaaS	Software-as-a-Service
SAML	Security Assertion Markup Language
SDK	Software Development Kit
SIL	Smart Internet Lab
SSL	Secure Socket Layer
TLS	Transfer Layer Security
TTL	Time To Live
VM	Virtual Machine
WSL	Windows Linux Subsystem

Chapter 1: Introduction

In the IT world, one of the main topics that is emphasized by a lot of engineers is the issue of security. As new frameworks and technologies get released daily, the number of devices connected to the internet gets even larger with a lot of them, if not most, being part of the IoT spectrum. This tendency results in even more complicated IoT systems, due to both their increased scale and the emergence of smart objects that exhibit semi-autonomous behaviour, for instance, drones, robots and autonomous guided vehicles, smart IoT sensors etc. For this matter, IoT engineers are faced with pressing security challenges which include more risks, vulnerabilities and security attacks. The latter require new and more intelligent approaches to IoT security, notably approaches that are able to tackle complex, unpredictable and sometimes asymmetric attacks at scale [1].

Recently, IoT has been widely applied to social life applications such as smart grids [2], intelligent transportation [3], smart security, and smart homes [4]. Access cards, bus cards and other small applications also belong to this category of devices. IoT applications can bring convenience to people, but personal information can be leaked at any time if personal privacy is not ensured. Therefore, security for this particular matter cannot be ignored. Theft or interruption of communication signals has a direct impact on the security of the complete IoT information. With the wide spreading of IoT, providing richer information increases the risk of such information being exposed. If IoT cannot have a good solution for security issues, it greatly limits its development. So, of all the issues of IoT, security issues are particularly important.

In terms of scalability [5], IoT applications that require a large number of devices are often difficult to implement due to time, memory, processing, and power constraints. For example, calculating daily temperature changes across the country can require millions of devices and lead to unmanageable amounts of data. Additionally, the hardware deployed in IoT often has different performance characteristics, such as sample rate and error distribution, while IoT sensor and actuator components are always very complex. All of these factors contribute to the formation of a heterogeneous [6] IoT network, where IoT data will be deeply heterogeneous. Furthermore, because it is expensive to send huge amounts of raw data over complex and heterogeneous networks, the IoT requires data compression and data fusion in order to reduce the data volume. Therefore, standardization of recognition of IoT data processing will be highly valuable in the future. Additionally, during the course of communication, hackers, malicious software and viruses can interfere with the integrity of data and information. Information insecurity in the development of IoT technology directly threatens the entire IoT system [7].

Starting to develop and implement different security solutions, delegated authorization seems as a promising approach to provide stronger and more configurable access control in the IoT world. Since 2014, the IETF Authentication and Authorization for Constrained Environments (ACE) WG is working towards the standardization of delegated authorization based on OAuth [8] for devices that require low power and little processing.

In IoT authorized access control, user accounts and access control policies are kept on centralized secure servers, not reachable by physical attacks, and IoT devices grant access based on some form of “authorization pass” that the requester must present. Thus, delegated authorization in IoT can allow one to

communicate directly with IoT devices while ensuring the security level of centralized access control servers. However, direct communication with IoT devices has yet to become a reality in actual IoT implementations. The most common communication model involves cloud infrastructures that collect data from IoT devices and makes that data, after processing it, available to the end users. As Beltran et al. mentioned in [9], the Internet Architecture Board has published a few communication protocols specifically designed for IoT in order to facilitate the issue of IoT interoperability. If direct communication between users and IoT devices is achieved, which is probably the most difficult task to fulfill, it will be one of the most promising communication models so far, as it will allow better scalability and privacy of IoT data by storing the collected data directly to the collector device [10].

1.1 Significance of the Study

As IoT applications and 5G is growing, security is becoming even more complicated and require revision on the already existing security protocols. Security and latency are crucial elements in 5G network design [11]. The applications based on 5G entail sending and receiving sensitive command, control, and feedback data. This data must be provided almost instantly while remaining unaffected by malicious actors such as eavesdroppers. Thankfully, 5G contains a multi-tier security architecture [12] which makes the developers able to develop additional security measurements in any level required.

There are several methods for stealing sensitive information from legitimate users, including altering information with malice, denial of service, charging users illegally, and launching a Denial of Service (DoS) attack, among others. There might be fake base stations, man-in-the-middle attack, broken and duplicate NFC cards, brute force attacks on Mifare Classic keys [13], and so on. Although existing higher layer encryption algorithms are nearly flawless, modern computers are quickly evolving, and cracking ciphers is only a matter of time [12].

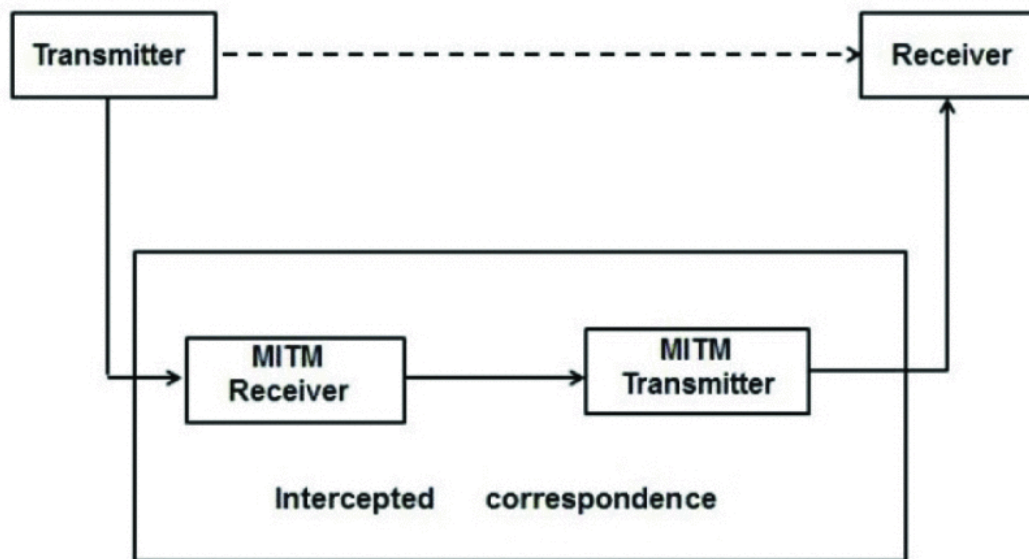


Figure 1.1: Man in the middle attack [14]

The spectrum of authorization choices for 5G bands made by regulators are heavily dependent on reli-

able supply of high-quality 5G services and innovative network deployment models, which is an open issue with developing alternatives for more dynamic methods [15]. While Mobile Network Operators (MNOs) will continue to require assured spectrum access, local micro-operators will require guaranteed local spectrum availability in order to provide high-quality localized 5G services. As a result, obtaining spectrum access rights is a significant problem in the micro-operator idea.

The work done in this thesis aims to, at least for our framework's case, solve this security issue and provide a proper authorization functionality which brings an end-to-end encrypted communication, while rejecting any unauthorized messages and packets received. The authorization protocol used is configured to share as little information to the IoT devices as possible, only providing them with their user's access token and nothing else. The rest of the authorization flow is taking place in our middleware backend service which was exclusively designed in order to examine and handle the different access tokens that arrive along with the receiving HTTP requests, based on their validity.

1.1.1 Research Statement

This thesis is written around a specific use case and model which is not yet defined publicly. The goal was to develop a truly cloud native architecture built on a hardened 5G private network, which is characterized by features like scalability, interoperability, extensive privacy modern light weight security protocols. While there are already some implementations made by other organizations in terms of security, the main aspect that makes this project special is that it is built to function on a 5G network. This means that the security applications should be revisited and redesigned in order to adapt on the cellular nature of the 5G network [16].

One of the light weight security protocols that we needed to apply on this framework was authorization, and specifically the OAuth 2 system which is based on the RFC 6749 [8]. The implementation was not that easy, because of the model that we wanted to achieve for the framework which included the users and their devices, controlled by cloud based architecture with secure communications and exceptional performance. The challenges we faced and the goals we had set for this project will be discussed in detail in some of the following sections.

Authorization techniques are not something new in the industry, but what makes this project special is that we had to apply authorization protection on a framework running on a private 5G cellular network. This means that the network is not congested like a WiFi one, because there are no channels to overlap each other. The 5G cellular network uses licensed bands and a deterministic scheme. It has been improved a lot with the application of MEC and it one of the best choices when combined with IoT applications [17]. Apart from explaining what is an authorization mechanism and how it can be exploited, this thesis is also going to describe the testbed that was developed and used for this matter.

1.1.2 Topic Analysis

The project that is going to be demonstrated in this thesis, follows a unique model based on a cloud native containerised architecture. The concept is as follows: A user wants to deploy a number of IoT devices and sensors wherever needed in order to gather data and use it for various reasons like archiving

past measurements, enabling smart-home like applications, triggering some kind of action based on a sensor's measurement (e.g. enable automated watering for plants based on soil's humidity levels) etc. In order to preview the different devices and their collected data, the users should be able to register and login through a self-service dashboard. This process is to be handled by an identity manager which should be user friendly and also provide the tools for administration and backend configuration.

The process of data collection must be controlled by the user in terms of authorization. The user should choose which of his devices are authorized to collect and send measurements to the data collector on his behalf. This operation can be achieved with the use of access tokens and specifically, the OAuth 2 authorization protocol which is the main topic of this thesis. With their access token distributed to the selected devices, the communication between devices and REST API is ensured to be authorized and can be easily validated through token introspection processes [18].

The communication between the devices and the rest of the framework is going to be performed through the private 5G network that was mentioned earlier. Then, a load balancer will be controlling the various incoming requests, routing them accordingly based on their destination endpoint. The final data collection node will follow the Cross Origin Resource Sharing (CORS) data scheme in order to maintain data homogeneity and device shadowing. Last but not least, in terms of encryption, the framework is protected by the HTTPS communication protocol which is implemented with the use of self-signed SSL certificates and a custom Certificate Authority (CA) that was created and installed on our various endpoints.

1.2 Contributions

A lot of research was made in order to plan, design and execute the authorization implementation presented in this thesis. Details regarding old authorization techniques will be analyzed in Chapter 2 while technical details will be presented in Chapters 3 and 4. The main contributions that were conducted in order to gather the required information and perform all the mandatory implementation are summarised as follows:

1. Research about OAuth 2, IoT and 5G

In order to prepare for this thesis and my internship's assignments, I had to conduct research and get informed about OAuth 2 authorization protocol, IoT security and 5G networking challenges.

2. Implementation of OAuth2 protocol on cloud native platforms

Thoroughly studying the OAuth 2 authorization protocol resulted in the successful implementation of it in a containerized environment and integration with an IoT cloud native platform built on a 5G testbed.

3. Development of a device emulation tool

A tool designed and built specifically for virtually testing a framework. It can emulate any given number of devices for any given number of users but can also act as a penetration testing tool.

4. Load balancer setup on a Docker network

NGINX integration on the Docker containerized engine in order to control access and load balance the network's traffic between the different endpoints.

5. **Manage containerized environments**

Configuration of the Docker's containerized environments so as to achieve better performance, security while maintaining the separation between the different containers.

6. **Coding for REST API provisioning**

Development of a CORS REST API with delegated access control responsible for collecting devices' data and storing them in a database based on specific schemas.

7. **Middleware API design**

Design and development of a middleware API, acting as a bridge/filter/connecting link between the framework's different endpoints (users, database, devices, authorization etc).

8. **Database schema planning**

Constructing MongoDB database schemas, specifically for the particular framework in order to store only the needed information in the best possible format for easier parsing.

9. **Automate experiment execution**

As an extension of the device emulator, we designed a feature for automating the execution of multiple instances of experiments on their respectable servers so as to increase efficiency and simultaneous testing.

10. **Integrate metrics collection**

When conducting the various experiments, we integrated triggers for capturing performance details such as Round Trip Time (RTT), delay, packet loss and consistency confidence levels.

11. **Testbed overview and configuration**

Managing and working on a testbed capable of performing multiple tests at the same time while allowing for extensive configuration so as to function as desired for each particular experiment.

12. **Infrastructure orchestration**

Helped automating configuration, management and coordination of various computing systems, applications and services in order to ease complex tasks and workflows.

1.3 Thesis outline

In the current B.Sc. thesis I will present the main security issues found in IoT and specifically related to projects built based on 5G networks, discuss market challenges that a lot of companies face and how they cope with them. Moreover, I will be mentioning past and current MEC security implementations and how my work is related to them. Then, information regarding authorization techniques will be referenced in order to provide the reader with a complete overview of the main topic so as to have a better understanding of the next sections which include technical details. Different authorization protocols, their main field of

application, advantages, disadvantages and characteristics that make them unique are the main subjects of Chapter 2.

In Chapter 3 there will be specific information in terms of the OAuth 2 authorization implementation and integration on an existing 5G IoT MEC framework and REST API along with some figures which are going to visualize the different flows being executed during the main functionality, Furthermore, Chapter 4 will be focusing on the demonstration of the testbed and the rest of the development tools that were built and used in order to run the experiments that will be analyzed as well, with tables and figures representing their results. Lastly, a summary containing the main contributions, experiment findings and ideas related to future work is going to wrap up this thesis in Chapter 5, hoping to have provided each section with enough details to cover every aspect needed. In the end of this thesis there will also be a list with all the bibliography used as well as a code appendix.

Chapter 2: Literature Review

Before presenting the technical analysis of the main topic, the implementation of the OAuth 2 authorization protocol on a real time IoT REST API, as well as details about the development environment and the challenges we faced, I must firstly go through a review of some important topics regarding security in MEC, current market trends regarding security, authorization history, improvements and different techniques that exists and lastly details about the OAuth 2 protocol that was chosen for this project.

2.1 Introduction to IoT security

The IoT is a rapidly expanding network of common physical things that are linked to the Internet [19]. The ultimate objective of the IoT is to turn Internet-enabled gadgets into a linked environment with digital data available anywhere and at any time [20]. IoT devices, which range in size from small wearables to larges machines and are equipped with sensors and actuators, intelligently detect their environment and act on their own [21]. Based on Cisco information, the estimated number of connected devices to the internet can be as large as 50 billion [22]. Most of these devices are resource constrained by design, with limited memory, processing capacity, and computing power.

The main idea of IoT can be defined in a simple sentence: "a global network of interconnected entities" [23]. However, in recent years, this central idea has been extended in various directions. One of the fundamental actors of IoT, the "things" itself, has evolved to cover different types of devices: from simple RFID labels and sensor devices to complex systems, such as connected cars, consumer devices, such as televisions, cameras and even basic services such as bathrooms [24]. IoT in itself has also been given many different names, which refine and / or expand its scope. Examples include the industrial IoT and the Internet of Everything (including things along with people, processes, data and their connections). Furthermore, the IoT has been closely linked to other paradigms, both because they have similar basic values, and because they use each other [25].

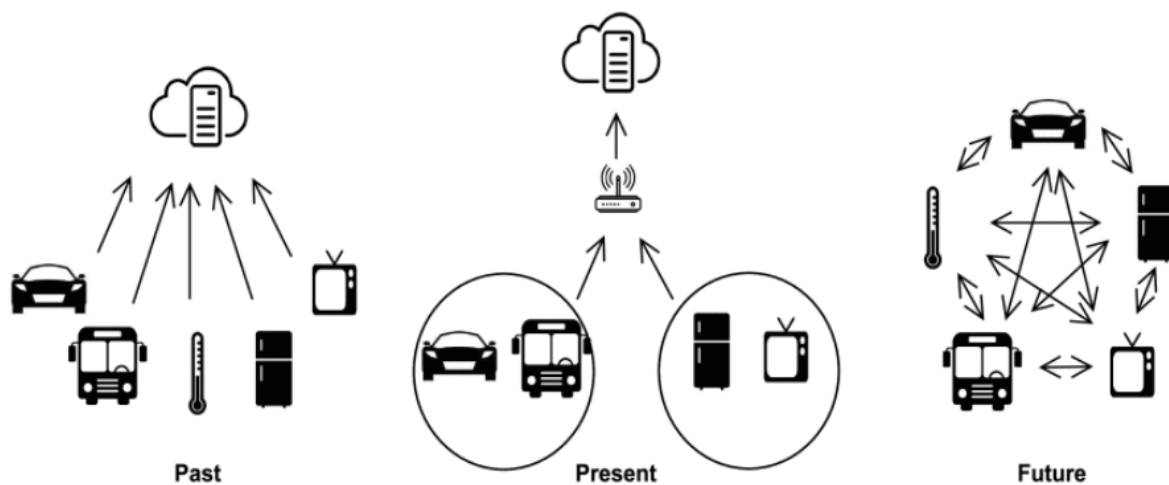


Figure 2.1: The evolution of IoT [20]

In the IoT, the idea of security is also expanding, and it has, of course, grown in importance in recent years [26]. However, as security becomes more important and more attention is devoted to the protection of the IoT ecosystem, it is necessary to have a more thorough understanding of security in the past and present. We can design and build more ideal security methods that are suited to secure our connected future by giving a thorough analysis of how the various IoT security sectors have grown over time and what the current trends are that have noteworthy influence over them.

Various enabling technologies, such as cloud computing, are becoming increasingly important in the development of the IoT paradigm [27]. In the not-too-distant future, IoT data will be generated by billions of devices through device-to-device (D2D) interactions, in which devices are connected to each other and exchange large amounts of data over the Internet. By 2025, the number of linked IoT devices is expected to reach one trillion. According to this estimate, the IoT will generate 11 trillion dollars in annual income by 2025 [28]. As a result, this expansion will raise a number of security concerns that must be addressed.

2.1.1 Evolution of IoT Security

The major long-term goals in a number of IoT security fields were quite apparent from the start: adapting existing and established protocols and algorithms to the context of the IoT, and increasing their performance as much as necessary. User-to-service and device-to-service authentication methods were first adopted from existing protocols developed for the IoT paradigm's building blocks, such as wireless sensor networks. Later, several researchers began to integrate existing federated identification and authentication protocols, such as OpenID and OAuth 2, into their work.

While these protocols make it easier for users/devices to communicate with services, in other situations, such as smart cities and industrial services, direct user-to-device and device-to-device authentication methods are required. Existing concepts such as user biometrics and out-of-band channels were used to this context for this goal. Similarly, the availability of better primitives has facilitated the transition from simpler access control mechanisms such as role-based authorization (RBAC; which is based on entity roles) to other token-based solutions, such as attribute-based authorization (ABAC), whose integration is actively being explored.

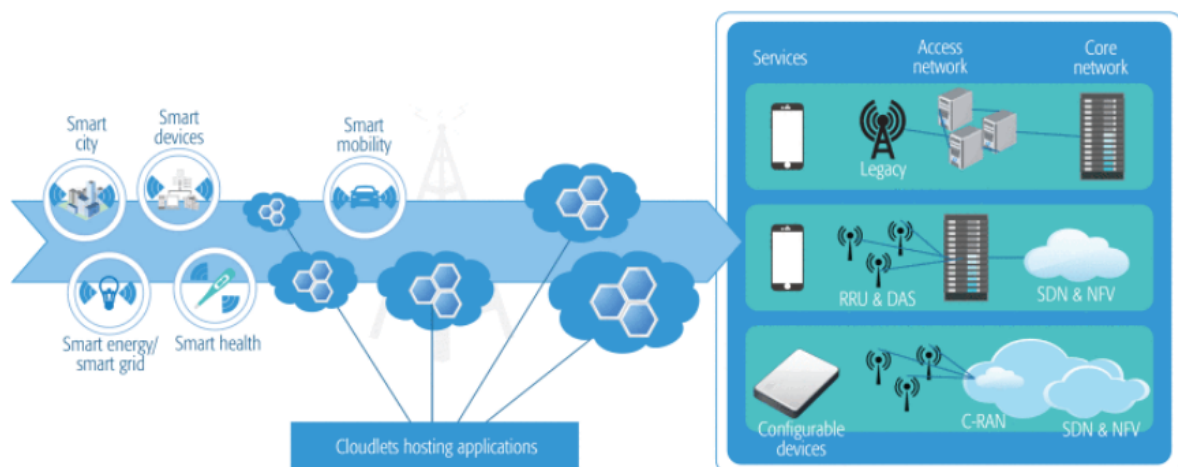


Figure 2.2: MEC deployment scenarios [29]

Finally, there are the concerns of trust and privacy to consider [30]. Both areas have received a lot of attention, with much of the research focused on the mechanisms established for sensor networks as a starting point, and then growing depending on the unique demands of the IoT. The definition of trust models, their integration into generic trust architectures designed for cloud-powered IoT infrastructures [31], and their application in various areas such as access control, intrusion detection systems (IDS), data collection, and usability have all seen advancements in trust mechanisms. Still, these methods must be improved, and trust integration into current IoT systems must be made easier.

In terms of privacy mechanisms, the majority of research has concentrated on determining what privacy means in the context of the Internet of Things, as well as what measures can be implemented. Some research has focused on data privacy, looking at how users may secure their data using methods like homomorphic encryption, which permits computation on encrypted data. Others are looking at other recognized aspects of the topic, such as location privacy, group anonymity, plausible deniability/anonymity, and privacy-aware low-level methods [32].

2.1.2 Challenges

Because 5G will connect every element of life to communication networks, it will require sophisticated security designs and solutions [33]. As a result, we explore and highlight the critical security and privacy issues that 5G networks face. The following are the basic issues in 5G identified by Next Generation Mobile Networks (NGMN) [34] [35] and widely addressed in the literature:

1. **Flash network traffic:** A large number of end-user devices and new "things" will be available (IoT).
2. **Security of radio interfaces:** Encryption keys for radio interfaces are delivered through insecure channels.
3. **User plane integrity:** The user data plane has no cryptographic integrity protection.
4. **Mandated security in the network:** The usage of security measures is optional due to service-driven restrictions on the security architecture.
5. **Roaming security:** Roaming from one operator network to another does not update user-security settings, resulting in security breaches.
6. **DoS attacks on the infrastructure:** There are unencrypted control channels and visible network control components.
7. **Signaling storms:** The non-access stratum (NAS) layer of Third Generation Partnership Project (3GPP) protocols, for example, requires cooperation in distributed control systems.
8. **DoS attacks on end-user devices:** On user devices, there are no security protections in place for operating systems, apps, or configuration data.

2.2 MEC Security

MEC infrastructures enable the deployment of software-only mobile functionalities or Software-as-a-Service (SaaS) applications [36] that run fully on a standardized virtualization platform at or near the network edge. MEC connects network edge devices including Access Points (APs), laptops, base stations, switches, and IP video cameras to provide data storage, processing, and application services. Mobile edge computing can serve the IoT, cyber-physical systems, automotive networks, smart grids, and embedded artificial intelligence with lower latency, location awareness, and mobility support since it is closer to clients than cloud computing [37].

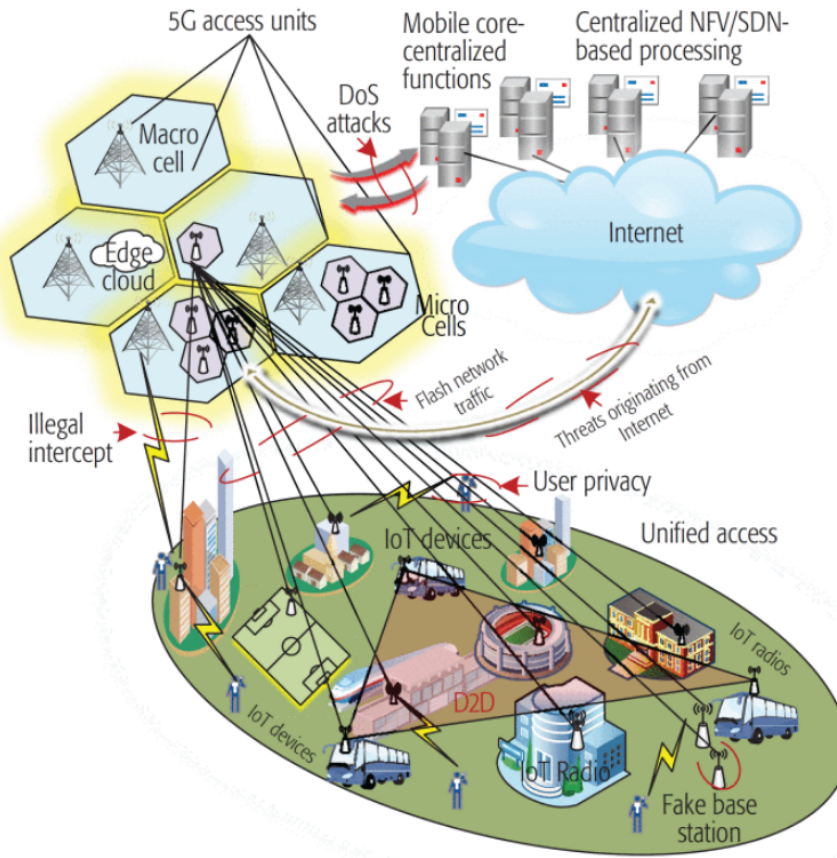


Figure 2.3: 5G network and potential threats [29]

Because edge devices are placed at the edge of heterogeneous networks and are physically closer to attackers, security and data privacy are essential and have become a bottleneck for MEC development. Edge devices are secured by several forms of security mechanisms, which are in general less secure than cloud servers and data centers, due to their restricted compute, energy, communication, and memory resources [38]. Furthermore, MEC systems are made up of dispersed edge devices operated by selfish and autonomous individuals. Owners of edge devices may be concerned about the data saved in their cache, and may even conduct insider assaults to analyze and sell consumer privacy information. As a result, security risks like as wireless jamming, distributed DoS attacks, spoofing attacks using rogue edge and rogue mobile devices, man-in-the-middle attacks, and smart assaults are more likely to affect MEC systems [39].

The majority of MEC's suggested security solutions rely on the strategic use of virtualization technolo-

gies, the redesign of encryption methods, and dynamic data processing point allocation. Because each end node connects to a distinct virtual instance in the cloud through a Virtual Machine (VM), virtualization is a natural choice for protecting cloud services. The separation of each user's virtual connection from other users provides security. One virtualization approach for securing MEC is virtualized in-cloud security [29].

2.3 Authorization Basics

Authorization is a security procedure for defining access stages or client/user privileges for system assets such as computer programs, files, application features, data, and services [40]. This is one of the strategies for granting or denying access to a network resource that gives the client the permission to utilize various resources depending on their uniqueness.

Authorization is a process that allows someone to have or do something. In multi-user systems, a computer system administrator specifies which clients have access to the system and what benefits they have (such as access to certain directory files, access hours, storage capacity, and so on). When someone logs into a PC system or application, the structure or program may need to recognize which assets the customer has access to throughout the session. As a result, authorisation is frequently viewed as both a collection of consents by structure administrators and a genuine examination of the authorization esteem that has been built up when a client is granted access.

Regular consumers value privacy, thus any IoT enabled application installed in any location must be built with strong security mechanisms. Because an IoT solution consists of numerous parts such as embedded devices, user interface elements, cloud computing for data processing, device control, and many others, implementing security and privacy aspects creates a functionality challenge. In order to control the access rights of each device while also providing the user with a clear image of his belongings and who/what device can access them, an authorization mechanism must be exploited. In this subsection I will present some general information about authorization, regarding its evolution and how it was created, some terminology around this field and describe some high level existing authorization variations.

2.3.1 The Idea

The Google Calendar API allowed application developers to access and alter a user's Google Calendar when it was initially published. However, a user could only grant delegated access by giving the app their account username and password, which the application would then utilize via Google's proprietary ClientLogin protocol [41].

Both small and large apps requested passwords from users to get access to their data, thanks to proprietary protocols like ClientLogin and standard protocols like HTTP Basic authentication. This wasn't limited to desktop programs; credentials were being requested by applications all across the Internet. One such application was Flickr, an online photo-sharing service. Flickr was once a stand-alone startup that was purchased by Yahoo! a few years after Google bought Blogger. The prospect of Yahoo! requesting Google user passwords, similarly to Yelp's solution displayed in Figure 2.4, alarmed both companies, prompting the creation of new proprietary methods to address the problem on the Internet.

Are your friends already on Yelp?

Many of your friends may already be here, now you can find out. Just log in and we'll display all your contacts, and you can select which ones to invite! And don't worry, we don't keep your email password or your friends' addresses. We loathe spam, too.

Your Email Service ☐ msn Hotmail ☐ YAHOO! MAIL ☐ AOL Mail ☒ Gmail

Your Email Address (e.g. bob@gmail.com)

Your Gmail Password (The password you use to log into your Gmail email)

[Skip this step](#) [Check Contacts](#)

Figure 2.4: Yelp's legacy resource owner password grant authorization

If an app wanted access to user data, these new protocols, such as Google's AuthSub and Yahoo's BBAuth, would redirect users to an authorization page on the provider's site. Users would log in to their accounts and authorize access, and the app would then be given a token to access the users' data [42].

While this addressed certain security concerns, it came at a cost to developers. Developers who wanted to integrate their apps with many major API providers had to learn and implement different web-based authorisation methods. Startups developing new APIs were hesitant to adopt proprietary authentication systems or design their own unique schemes, which might bring security flaws. Instead, these startups and large API providers agreed that a common protocol was required to ensure the consistency of these web-based authorisation procedures.

2.3.2 Terminology

It is necessary to first comprehend the appropriate terminology in order to fully understand OAuth. We'll start with a few important terms and then introduce some additional ones throughout the thesis.

1. Authentication

Authentication is the process of confirming a user's identification and ensuring that they are who they say they are. For instance, when a police officer asks for someone's identification in the real world, they are confirming their identity by checking that the photo on their ID matches their likeness. Authentication on desktop computers and the internet involves confirming that the person typing is the account's owner. Authentication is usually done by asking for a username and password from the user. The user's claimed identity is represented by the username, and the software program thinks that if the user enters the proper password, they are that user.

2. Federated Authentication

Although many software have their own account system (with usernames and passwords), others rely on third-party services to authenticate users' identities. Federated authentication [43] is the term for this. To authenticate users in a corporate IT environment, apps may rely on an Active Directory server, an Lightweight Directory Access Protocol (LDAP) server, or a Security Assertion

Markup Language (SAML) provider. On the web, apps frequently entrust user authentication to OpenID providers (such as Google). Federation has several advantages for both application developers and consumers. The most widely used open web standard for federated authentication is OpenID.

3. Authorization

The process of confirming that a person has the authority to execute a task, such as reading a document or accessing an email account, is known as authorization. This usually demands valid user identification (authentication) before determining if the user is authorized. For example, when a police officer pulls someone over for speeding, they first verifies the driver's identification using their license, then double-checks the license (expiration date, limitations, etc.) to make sure they are allowed to drive. The same thing happens online: a web application confirms a user's identity by signing them in, and then checks an access control list for each action to verify that they only access the data and services they are permitted to.

4. Delegated Authorization

Delegated authorization [44] helps to offer another person or application permission to act on the initial user's behalf. For example, when a visitor arrives at a posh hotel, they may be offered valet parking. They then give the key to the valet staff and permit him to drive their car on their behalf. OAuth operates in a similar way: a user provides permission to an application to conduct activities on their behalf, and the application may only carry out the actions that the user has allowed them to.

2.3.3 Different Authorization Techniques

In terms of authorization models, there are four different ones which function differently and each one can benefit in different use cases than the others. In this subsection I will be listing these four authorization models that have been established along with their limitations in order to demonstrate how versatile authorization can really be.

1. ACL

An access control list (ACL) [45] is a list that matches users and access permissions to grant authorisation for particular resources. Basic functionality, like as logins, and access to specific items in a database, are controlled using the ACL method. Some of its limitations include:

- Control: An ACL gives access permissions exclusively on the basis of the user's identification, without considering any other factors.
- Administration: Authorization may be time-consuming and difficult to handle, especially in big businesses. When a user requires a large number of particular access permissions, which are provided one by one, and their responsibilities vary, or when resources are added or deleted, for example.
- Scenario: A group of nurses who previously only had access to patient data in their department is suddenly given access to the ICU. More access permissions, including patient data, are

now required for each nurse. Assigning these rights takes a long time and is prone to human mistake.

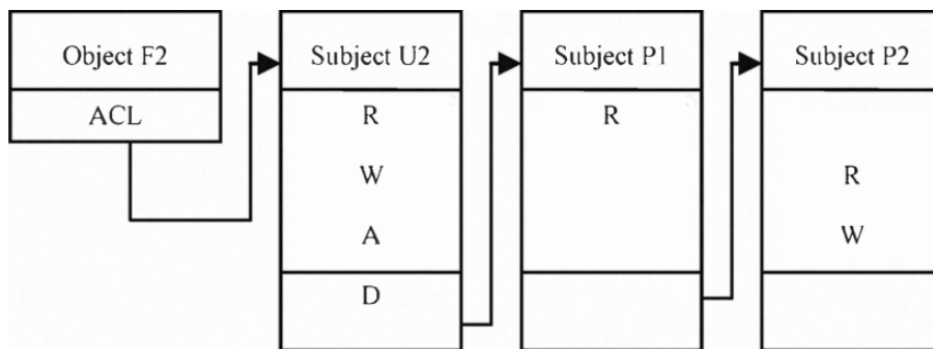


Figure 2.5: Access control lists [45]

In Figure 2.5, R denotes the operation of reading, W denotes the operation of writing, A denotes the operation of adding and D denotes the operation of deleting.

2. RBAC

Role-based access control (RBAC) [46] systems function by establishing fine-grained roles, each with its own set of access privileges, and then assigning each user to one of them. When the rights of a role are changed, the rights of all users who have that role are also changed immediately and safely. When a user's role is changed, their rights are modified as well. Although it sounds ideal, this model has some limitations of its own as well:

- **Control:** RBAC determines authorization only on the basis of the user's position, disregarding any other relevant factors. Access privileges are set in stone and can only be changed by assigning the user to a new position or redefining that role, both of which have consequences for others. Furthermore, RBAC is unable to fulfill the more complicated authorization requirements of IT environments that are linked to third-party infrastructures.
- **Administration:** Of bigger companies, the requirement for more granular access control leads to a rise in specified roles, but as One Identity's Joe Campbell says, "I have seen it over and over again, with time users accumulate roles but never have outdated roles taken away."
- **Scenario:** A group of engineers moves from project to project over the course of a few years. Even though that job is no longer relevant, their access privileges to the initial project remain retained.



Figure 2.6: RBAC model [47]

3. **ABAC** Attribute-based access control (ABAC) [48] systems employ characteristics (such user location or role) to construct a finer-grained access system. Authorization is accomplished automatically at runtime once a policy is created, with no need to identify particular users or groups. Let's examine the limitations of this particular model in the following list:

- **Control:** Attributes are decided locally rather than globally, as is typical in a company (e.g. EU administrators might have different access rights than US administrators).
- **Administration:** Increasing flexibility leads to increased complexity in administration. For a small business that does not employ third-party infrastructure, this may not be the best option.
- **Scenario:** Access to a resource may not be managed universally — access to a resource may be controlled by user credentials in one organization of the company, but by a digital certificate in another. This lack of consistency can lead to redundancy, misunderstanding, and, eventually, access control flaws.

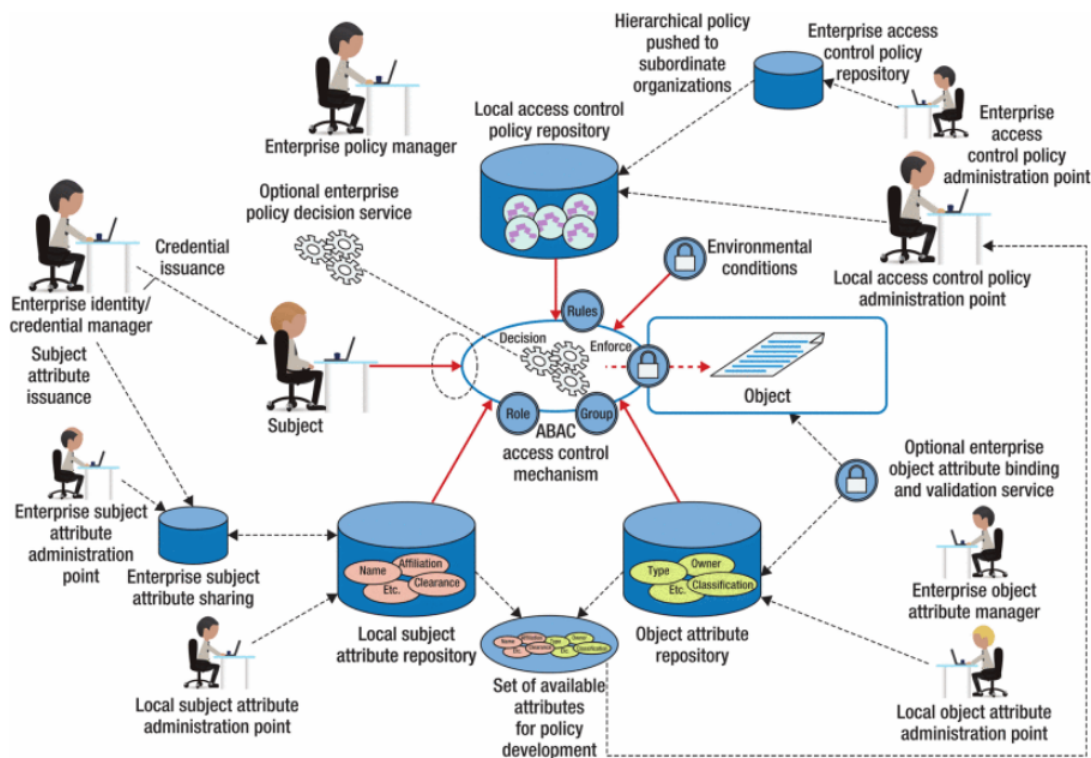


Figure 2.7: ABAC use case example [48]

4. PBAC

Policy-based access control (PBAC) [47] is a dynamic system that gives access based on logical rules that may be changed in real-time, altering user access automatically if necessary. PBAC is a natural extension and evolution of all previous access control systems. PBAC completely covers roles and characteristics, and it is the ideal choice for corporate applications since it can handle different authorization protocols and languages. But every model has some limits, and this one does too:

- **Control:** Because PBAC is so versatile, separate administrators may accidentally design rules that contradict one other.
- **Administration:** Because the technologies currently on the market to administer PBAC are still in their infancy, implementation can be a big undertaking.
- **Scenario:** "Access to the CRM application will only be given to approved personnel who try access from within the company's internal network," an IT Security Manager writes in a policy for employees. The VP of research and development, who has admin privileges to the CRM system and wants field support people to be allowed to use it, then adds a new and conflicting rule that reads, "Access to the CRM application will be permitted to approved employees who try access through a VPN."

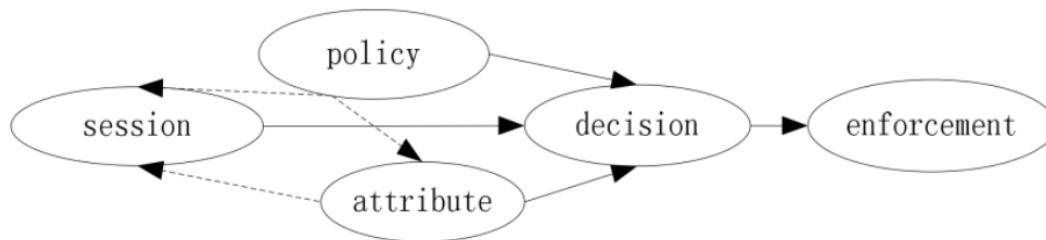


Figure 2.8: PBAC model [47]

The conflict in the case is simply whether an authorized employee may access the CRM both within the company's internal network and when on the road through a virtual VPN, and it can be easily avoided if the PBAC management tools are properly developed. A policy-based approach, on the other hand, would allow the establishment of a policy that alters an employee's access in real-time based on their location — either within or outside the company's internal network.

Table 2.1: Authorization models characteristics

	List-Based	Role-Based	Attribute-Based
ACL	X		
RBAC		X	
ABAC			X
PBAC		X	X

The requirement for finer-grained access restrictions grows with the size of the system, but so does the danger of complication. While there will always be trade-offs when it comes to picking the appropriate authorization model, as well as building and maintaining a specific system, it is apparent that PBAC systems offer the most comprehensive solution. Many of the drawbacks of PBAC may be addressed with the appropriate tools for managing it, allowing one to make use of all of its benefits.

2.4 OAuth 2.0 Framework - RFC 6749

Originating from OAuth version 1.0 [49], OAuth 2.0 is the latest and most commonly used authorization framework used by some of the biggest tech companies like Google, Facebook, Amazon etc. The OAuth protocol was created to address the use-case of "Delegated Authorization" [50]. Access to a user's information does not have to be all or nothing, according to OAuth. When an application just requires a single resource, there is no need to provide it access to all resources (like contact information for example). It solves mostly all of the issues and meets the requirements mentioned in the previous subsections, while providing a wide range of customization and integratability in order to adapt and fit in any situation needed.

OAuth grew so popular that it was adopted as a method of authentication. When a user sought to create a new account on a website, Google (and other companies) started offering the "Sign in with Google" option. Because OAuth was not meant for authentication, this was a terrible idea. To compensate for the lack of authentication capabilities, Google and Facebook implemented their own proprietary hacks to OAuth. The issue was that these proprietary OAuth implementations were not interoperable, resulting in a lack of standardization. By putting a layer on top of the current OAuth protocol, OpenID, which is worth mentioning but will not be described in detail, enhances its capability to incorporate authentication. OpenID Connect, in particular, performs Federated Authentication, which allows users to log in to a site using their Google (or any other platform's) account.

RFC 6749 [8] describes in detail how this whole authorization framework works and what it is consisted of. While one could read the previously mentioned RFC and try to understand the main functionality and use cases of the OAuth 2 protocol, I will try to describe it myself in the following subsections while trying to keep terminology as simple as possible. A lot of figures and tables are going to be included so as to enhance the reader's understanding.

2.4.1 Actors

Clients, resource owners, authorization servers, and protected resources are the four primary actors in an OAuth system [51]. Each of these components is in charge of a separate aspect of the OAuth protocol, and they all work together to make it function.

1. Resource Owner

The entity that has the power to delegate access to the client is known as the resource owner. The resource owner, unlike other components of the OAuth system, is not a piece of software. The resource owner is usually the individual who is utilizing the client software to gain access to something they own. The resource owner communicates with the authorization server via a web browser for at least part of the procedure (more generally known as the user agent). The resource owner may also communicate with the client using a web browser, although this is totally dependent on the client's nature.

2. OAuth Client

An OAuth client is a piece of software that tries to get access to a protected resource on behalf of the resource owner, and it does so via OAuth. The client is typically the simplest component in an OAuth system, thanks to the architecture of the OAuth protocol, and its duties are mostly focused on receiving tokens from the authorization server and utilizing tokens at the protected resource. The client is not required to comprehend the token, nor should it ever need to examine its contents. Instead, the token is used as an opaque string by the client. An OAuth client can be a web application, a native application, or even a JavaScript application that runs in the browser.

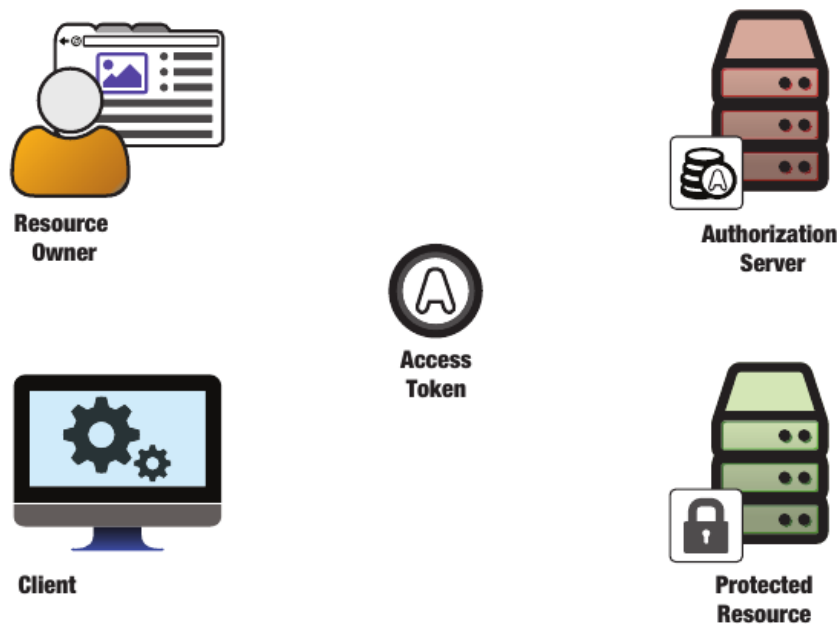


Figure 2.9: OAuth 2 actors [52]

3. Authorization Server

A crucial component of an OAuth system is an HTTP server that functions as an OAuth authorisation server. The authorization server verifies the resource owner and client's identities, offers ways for resource owners to approve clients, and gives tokens to the client. Some authorization servers additionally provide features like token introspection and the ability to remember authorization choices.

4. Protected Resource

Usually, an HTTP server hosts an OAuth protected resource, which can only be accessed with an OAuth token. The protected resource must verify the tokens that have been given to it and decide whether or not to service requests. The protected resource has the ultimate say in whether or not to honor a token in an OAuth architecture.

2.4.2 Access Token

An OAuth access token, also known as a token, is an expiring artifact provided by the authorization server to a client that shows the permissions that have been assigned to the client. The token itself is not defined by OAuth, but it always reflects a combination of the client's requested access, the resource

owner who approved the client, and the privileges granted during that authorization (typically including some indication of the protected resource).

Because OAuth tokens are opaque to the client, the client has no need (and frequently no ability) to inspect the token. The client's responsibility is to transport the token, requesting it from the authorization server and delivering it to the protected resource. Everyone in the system can see the token: the authorization server's role is to issue it, and the protected resource's responsibility is to validate it.

As a result, they must both be capable of understanding the token and what it represents. The client, on the other hand, is entirely unaware of all of this. This method allows the client to be considerably simpler than it would otherwise be, while also allowing the authorization server and protected resource to distribute tokens in a variety of ways.

After obtaining an access token, it can be sent along with a request in a variety of ways. The access token is sent in an HTTP Authorization header as a Bearer token [53], which is the recommended way of proving authorization.

```
GET /tasks/v1/lists/@default/tasks HTTP/1.1
Host: www.googleapis.com
Authorization: Bearer ya29.AHES6ZSdfadgfzX...
```

Figure 2.10: Access token use in request header

The Authorization header is preferable because:

- It is rarely logged by proxy servers and web server access logs.
- The header is almost never cached.
- The header is never saved in the browser cache when making requests from the client.

While the other techniques are specified in the standard, API providers are not obligated to implement any of them, so their mileage may vary.

- **Query parameter:** When libraries make it impossible to alter the Authorization header, using the access token as a URL query parameter is beneficial for debugging. When utilizing the client-side flow and delivering a token in a JSONP request, this method is also useful. For instance:

```
https://www.googleapis.com/tasks/v1/lists/@default/tasks?
callback=outputTasks&access_token=ya29.AHES6ZTh00gsAn4
```

Figure 2.11: Access token as query parameter

- **Form-encoded body parameter:** When an application is unable to change the Authorization header on requests, this is a fallback technique. It should only be used when an HTTP body would usually be transmitted, and it may then be appended as a form parameter to an application/x-www-form-urlencoded body. The Google Tasks API does not support this technique.

2.4.3 Scopes

A collection of rights at a protected resource is represented by an OAuth scope. In the OAuth protocol, scopes are represented by strings, and they may be concatenated into a set using a space-separated list. As a result, the space character cannot be included in the scope value. OAuth does not specify the format or structure of the scope value. Scopes are a key method for restricting a client's access.

The protected resource defines the scopes depending on the API it provides. Clients can request specific scopes, and the authorization server can allow the resource owner to grant or reject such scopes to a specific client at any time throughout the request process. The majority of scopes are additive in nature.

2.4.4 Refresh Token

An OAuth refresh token is similar to an access token in that it is issued by the authorization server to the client and the client has no knowledge of or interest in what is included within the token. The token is never delivered to the protected resource, which is what makes it unique. Instead of engaging the resource owner, the client utilizes the refresh token to request new access tokens.

Why would a client require a refresh token in the first place? An access token in OAuth might stop functioning for a client at any time. The token might have been revoked by the user, expired, or became invalid due to another system event. When the client uses the token and receives an error response, the token is generally found to be incorrect. Of course, the client could have the resource owner reauthorize it, but what if the resource owner isn't there anymore?

The client had no choice but to wait for the resource owner to return in OAuth 1.0. To circumvent this, OAuth 1.0 tokens tended to last indefinitely until expressly canceled. This is a bit of an issue since it expands the attack surface for a stolen token: the attacker can use it indefinitely. Access tokens in OAuth 2.0 have the option of expiring automatically, but we still need a means to access resources when the user isn't there. The refresh token has replaced the long-lived token, however it can no longer be used to get resources; instead, it can only be used to obtain new access tokens, which may then be used to obtain the resources. The refresh token and the access token are both protected in this way, although in different yet complimentary ways.

Refresh tokens also allow the client to narrow the scope of their access. If a client has access to scopes A, B, and C but only requires scope A to perform a certain call, the refresh token may be used to request an access token for only scope A. This allows a smart client to follow the least privilege security principle without burdening less-clever clients with figuring out what rights an API requires. Years of deployment experience have demonstrated that OAuth clients are far from intelligent, but it's still useful to have the advanced functionality available for those who wish to use it. What happens if the refresh token isn't valid? When the resource owner is available, the client can disturb them again. In other words, an OAuth client's fallback state is to perform OAuth again.

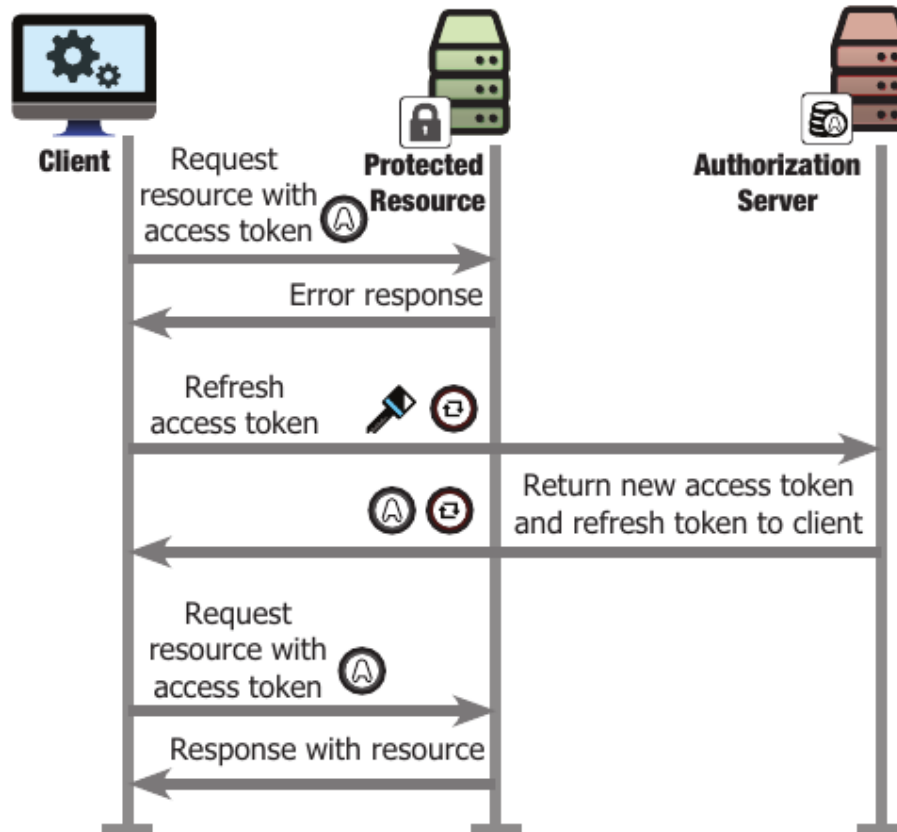


Figure 2.12: Refresh token basic usage [52]

2.4.5 Token Introspection

The OAuth Token Introspection protocol [18] establishes a method for a protected resource to actively query an authorization server about the token's status. Because the authorization server created the token, it is in a unique position to understand the specifics of the permission delegation that the token represents. The protocol is a straightforward extension of OAuth. The token is issued by the authorization server to the client, who then delivers it to the protected resource, which then inspects the token at the authorization server.

The introspection request is a form-encoded HTTP call to the authorization server's introspection endpoint, allowing the protected resource or a middleware API to inquire about the tokens' validity and usage. During this request, the protected resource or middleware API authenticates itself so that the authorization server can determine who is asking the query and may respond differently based on who is asking. The requesting component must authenticate itself, but the introspection specification does not specify how it must do it.

2.4.6 Token Revocation

Based on RFC 7009 [54], OAuth is frequently used to log into a website or service. When an end-user logs out, changes identities, or uninstalls an application, this revocation method allows a client to invalidate

```

{
  "active": true,
  "scope": "offline_access",
  "client_id": "oauth-client",
  "sub": "owner_identifier",
  "exp": 1630677022,
  "iat": 1630676961,
  "nbf": 1630676961,
  "aud": [],
  "iss": "https://example.com/oauth2/token",
  "token_type": "Bearer",
  "token_use": "access_token"
}

```

Figure 2.13: Token introspection result

its tokens. Notifying the authorization server that the token is no longer required allows the authorization server to clear up data (e.g., session data) and the underlying authorization grant connected with the token.

This approach avoids a situation where a legitimate authorization grant for a specific client exists but the end-user is unaware of it. In this approach, token revocation prevents misuse of abandoned tokens and improves end-user experience by removing invalidated authorisation grants from a list of permission grants presented to the authorization server.

2.4.7 Authorization Code Grant

The resource owner is initially forwarded by the application to the API provider's OAuth authorization server in the Authorization Code flow. The authorization server looks to determine if the user is currently logged in. If they do, the authorization server will ask their for permission to view the required information. They will be routed back to the web application after granting access, and an authorization code is added in the URL as the code query parameter (Figure 2.14). The web browser transmits the code to the web server serving as the OAuth client since it is given as a query parameter. Using a server-to-server connection from the application to the authorization server, the authorization code is then exchanged for an access token. The client uses this access token to perform API requests.

```
https://www.example.com/oauth_callback?code=ABC1234
```

Figure 2.14: Authorization code as URL query

The access token is not exposed to the resource owner's browser in this flow. Instead, authorization is done through the browser using an intermediate "authorization code." Before calling protected APIs, this `code` must be swapped for an access token. The exchange procedure will only occur if a valid `client_secret` is included in the request, ensuring the access token's secrecy as long as client security is maintained. Unlike the Implicit flow, which will be discussed in a following subsection, this secrecy extends to the resource owner, which means API calls performed with the access token may be traced back to the client and its developers. Most significantly, because the access token is never delivered through the browser, there is less danger of it being exposed to malicious code via browser history, referer headers,

JavaScript, and other methods.

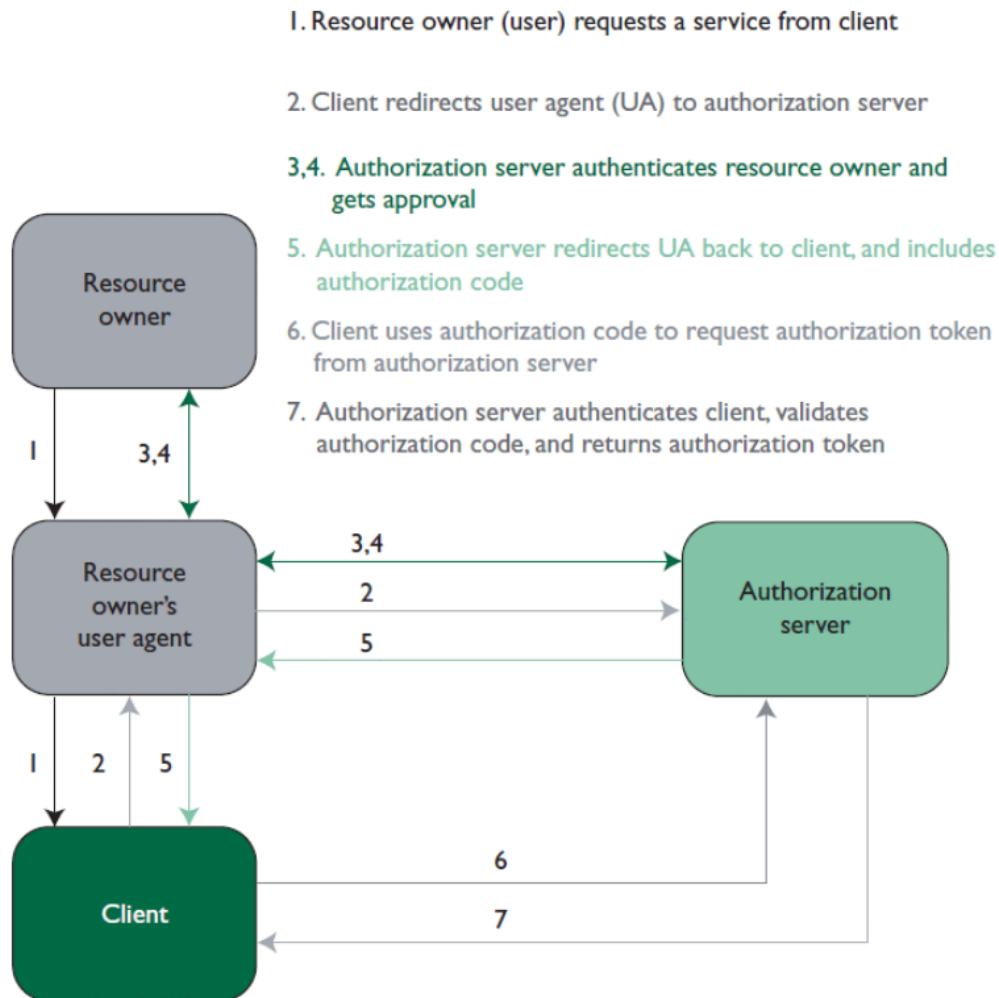


Figure 2.15: Authorization code grant flow [55]

Although the access token is less likely to be leaked because it isn't visible to the browser, many apps that employ this flow will retain long-lived refresh tokens in the application's database or key store to enable "offline" data access. When an application requires long-term offline access to data, there is an extra danger since it establishes a single point of compromise for many users' data. Other flows, such as the flow for client-side web applications, do not have this problem. Despite this added danger, many websites will choose for "offline" data access since their application design makes interacting with the user's browser to get fresh access tokens cumbersome.

The authorization code flow should be used when long-lived access is required, when the OAuth client is a web application server, when accountability for API calls is very important and the OAuth token should not be leaked to the browser, where the user may have access to it.

2.4.8 Client Credentials Grant

The majority of OAuth flows are used to handle delegated authorization, which occurs when a resource owner allows an application access to their data. There are times, however, when the client owns the data and does not require delegated access from a resource owner, or when the application has previously been granted delegated access outside of a regular OAuth procedure.

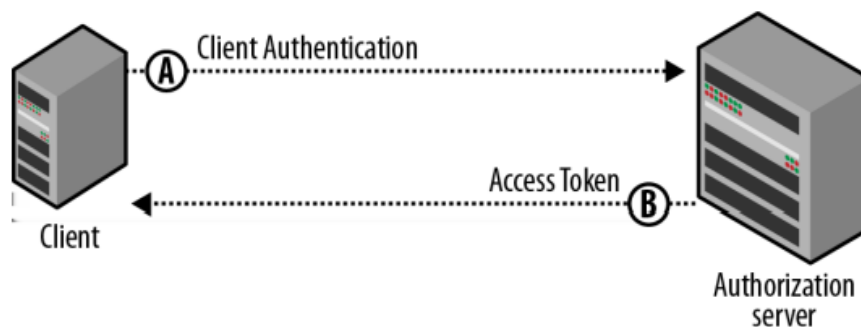


Figure 2.16: Client credentials grant flow [41]

Consider a storage API like Google Drive. A developer is utilizing such an API to create an app with resources (data files, pictures, etc.) hosted outside of their app. The program must read and update these resources, but it must do so on its own behalf rather than on behalf of any specific user. The Client Credentials flow is ideal for this scenario. Without the involvement of any end user, the application can immediately request an access token from the OAuth authorization server. Another use case for the Client Credentials flow is when a resource owner has allowed an application out-of-band access to their resources rather than utilizing a standard OAuth flow.

This flow is dependent on the client's ability to successfully authenticate with the authorization server and the confidentiality of the client's authentication credentials. To authenticate, the client can use an HTTP Basic Authentication header or provide the `client_id` and `client_secret` to the authorization server as POST parameters in the access token request. Other techniques, such as a public/private key pair, SSL/TLS client authentication, and so on, can be used by the authorization server to authenticate the client.

A single set of credentials for a client might offer access to a significant quantity of data, depending on the use case for which the Client Credentials flow is utilized. If a single set of credentials has access to more data, the danger of the credentials being hacked increases. It's important that the credentials used to authenticate the client remain completely private. These credentials should, ideally, be changed on a regular basis.

2.4.9 Device Code Grant

The Device Code Flow, also known as the Device Authorization Grant [56], addresses the situation when user authentication is difficult to enter on the device itself. The flow is begun by the device, but authentication is done out-of-band on a separate device with superior input, such as a browser with a keyboard.

The initial device checks with the authorization server to see if the user authentication has been performed

successfully. Tokens are issued once it is available. The Device Flow is a popular authorisation flow for smart TVs and other devices such as Apple TV.



Figure 2.17: Device code grant flow, as displayed in oauth.com

2.4.10 Legacy: Implicit Flow Grant

This type of authorization grant is considered as legacy and has stopped being developed and supported by most of the companies. For browser-based client-side web applications, the Implicit Grant flow is relatively straightforward. After a user grants the required authorisation, an access token is instantly returned to the application in this flow. As with the server-side Authorisation Code Flow, an intermediary authorization code is not required.

The application does not keep long-lived refresh tokens on a server in the Implicit Grant flow, minimizing the exposure if the server is hacked. In order to "refresh" access tokens on the client, the user must be authorized to the API provider's authorization server, guaranteeing that the value of a leaked access token is time-limited, depending on the OAuth implementation. This flow provides less accountability than the Authorization Code flow since the access token is delivered to the user's web browser. API requests that appear to have come from a third-party app might really have come from the resource owner directly.

The Implicit Flow grant used to be used only when temporary access to data was required, the user was regularly logged into the API provider, the OAuth client was running in the browser and the browser was strongly trusted and there was limited concern if the access token would leak to untrusted users or applications.

2.4.11 Legacy: Resource Owner Password Grant

Just like the previous subsection, this type of authorization grant is considered as legacy and has stopped being supported. The Resource Owner Password Credentials flow allowed to trade a user's username and password for an access token and, if desired, a refresh token. The security characteristics of this flow are considerably different from those of the other OAuth flows. The main distinction is that the program has access to the user's password. This necessitates the user's complete faith in the program.

Although the application has access to the resource owner's password, there are certain security advantages to employing this flow rather than using a username and password to authenticate API requests (via HTTP Basic access authentication or similar). In order to conduct API calls using Basic authentication, an application must have ongoing access to the user's password. If the user no longer wants an application

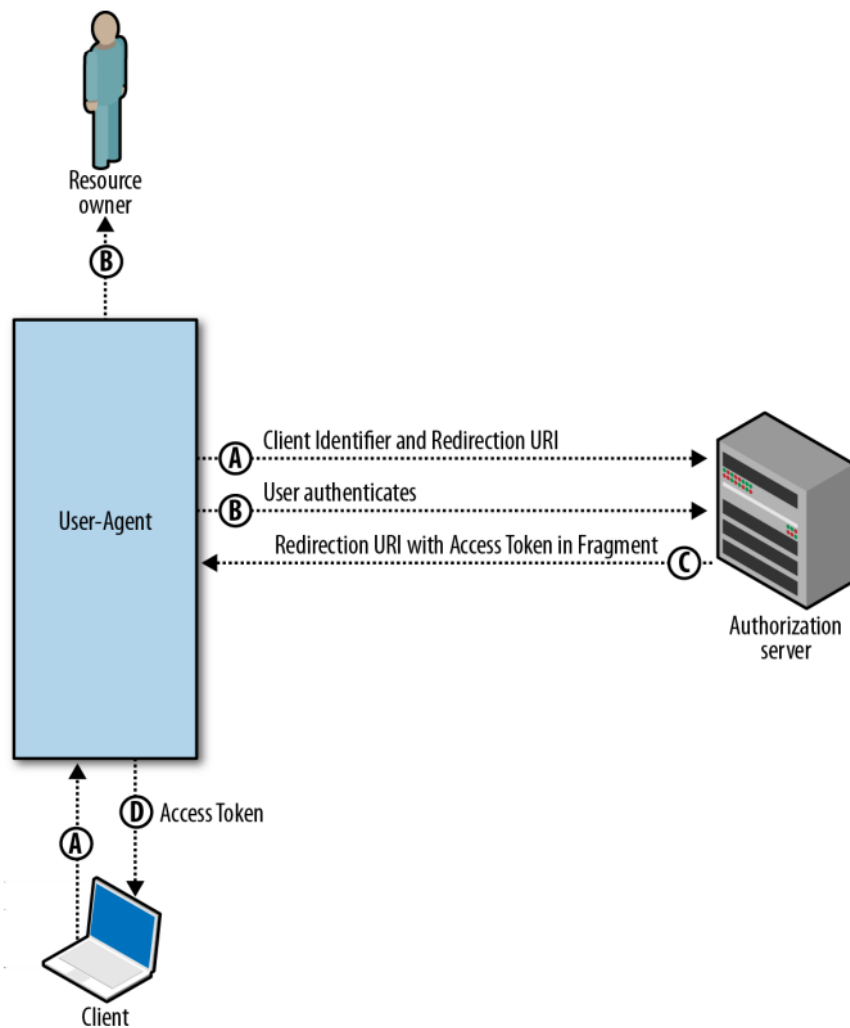


Figure 2.18: Implicit grant flow [41]

to have access to their data, it also needs the user to change their password and reenter the new password in any apps that require it. When using the OAuth Resource Owner Password flow, the application only requires access to the user's credentials once: when the credentials are exchanged for an access token for the first time. This eliminates the need for the app to keep these credentials within the app or on the device, and it also makes canceling access simple.

This flow should be used sparingly since the resource owner's password is revealed to the application. It is exclusively suggested for first-party "official" API provider apps, and it is not open to third-party developer communities. If a user is repeatedly requested to input their password into "official" programs, they may get used to it and therefore vulnerable to phishing efforts by other apps. To alleviate this worry, developers and IT administrators should make it apparent to their users how to tell which applications are "official" and which aren't.

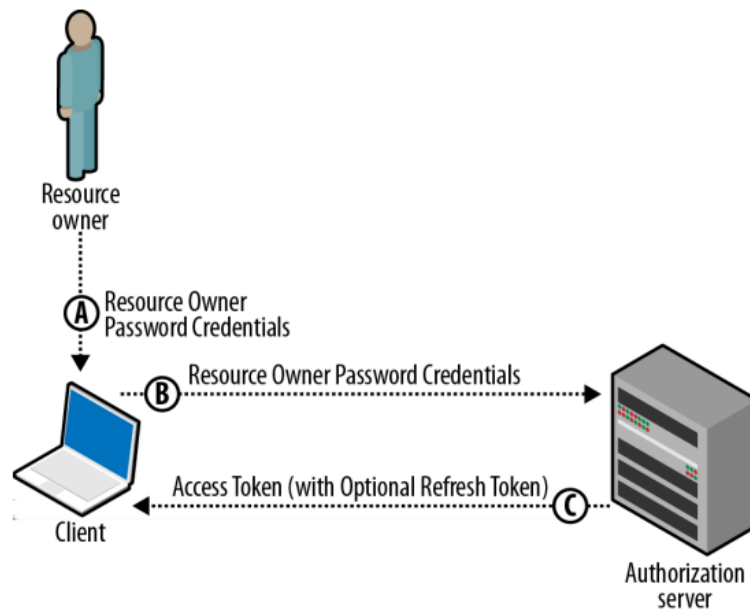


Figure 2.19: Resource owner password grant flow [41]

2.5 Vulnerabilities

In this subsection, I'll go over the various vulnerabilities [57] found in OAuth 2 in the event of a poor deployment. While OAuth 2 is a security standard, its implementation alone does not ensure security. Indeed, everything must be properly installed and configured. Furthermore, certain of the deployment options in the OAuth 2.0 standard might result in incorrect setups. Instead of creating a false feeling of security by claiming that OAuth is a secure protocol (which it is), I'll point out some of the flaws are and how to prevent them [41].

2.5.1 CSRF Attack

Cross-Site Request Forgery (CSRF) is one of the most prevalent threats on the internet [58]. One of the primary reasons for its popularity is because the ordinary developer does not understand the threat, making it an easier target for the attacker. When a malicious application forces the user's browser to do an undesirable action in response to a request to a web site where the user is currently authorized, this is known as CSRF [59]. If a user is signed in to a site that allows them to do a task and an attacker manipulates the user's browser to make a request to one of these task URIs, the task is completed as the logged-in user. Typically, an attacker may insert malicious HTML or JavaScript code into an email or website to request a specific task URI, which the user is unaware of. This procedure is generally described in Figure 2.20.

The most frequent and successful mitigation is to include an unexpected element in every HTTP request, which is the OAuth specification's countermeasure. Let's look at why using the state parameter to avoid CSRF is highly recommended, as well as how to create a good state parameter that can be securely utilized. An OAuth client's mitigation is to construct an unguessable state parameter and transmit it along to the authorisation server's initial request [60]. The standard requires the authorisation server to

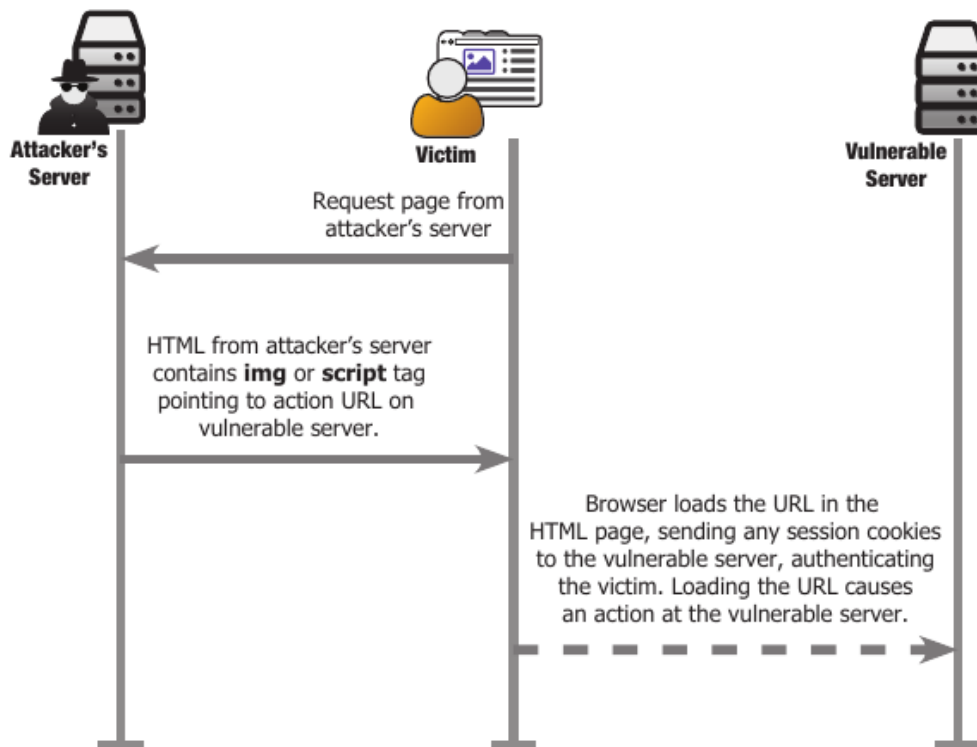


Figure 2.20: CSRF attack [41]

deliver this value as-is as one of the arguments to the correct URI. The client then verifies the value of the state argument when the redirect URI is invoked. The client might end the flow with an error if it is missing or does not match the value initially supplied. This prevents an attacker from injecting their own authorization code into the client of an unwitting victim.

2.5.2 Theft of authorization codes

Can the attacker "take" anything from the resource owner's personal information, such as email, contact information, and so on, if the authorization code is hijacked? Not yet, at least. Keep in mind that the authorization code is just a step between the OAuth client and the access token, which is the attacker's ultimate aim. The client secret is required to exchange the authorization code for an access token, and it must be kept confidential. However, if the client is a public client, it will not have a client secret, allowing anybody to use the authorisation code. With a confidential client, an attacker can either try to get the client secret maliciously, or try to fool the OAuth client into conducting a CSRF attack as displayed in Figure 2.21.

2.5.3 Theft of tokens

The ultimate goal for an attacker that focuses their attention on an OAuth aware target is to steal an access token. The attacker can use the access token to execute a variety of activities that they weren't supposed to be able to accomplish. To consume APIs, OAuth clients submit access tokens to resource servers, as we've already seen. The bearer token is generally supplied as a request header (Authorization: Bearer

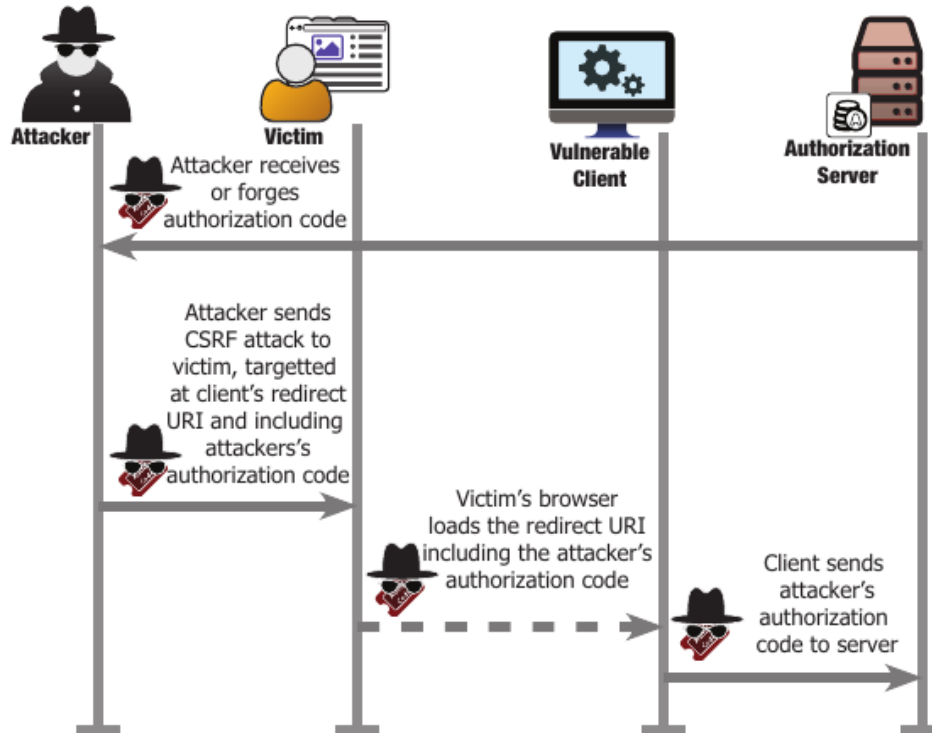


Figure 2.21: Authorization code steal [41]

access token value). RFC 6750 specifies two more methods for passing the bearer token forward. Clients can transmit the access token in the URI using the `access_token` query parameter, according to one of them, the URI query parameter [53]. Although its simplicity tempts someone to utilize it, there are a number of disadvantages, as described in Subsection 2.4.2, to utilizing this approach to submit an access token to a protected site, so these access token transmit methods are ought to be avoided. A potential access token theft that can be performed on the query parameter method is showcased in Figure 2.22.

It is critical that access tokens provided as bearer tokens are not sent via an unsecured channel in the clear. According to the core OAuth standard, access tokens must be transmitted via end-to-end encryption, such as SSL/TLS [61]. TLS, originally known as SSL, is a cryptographic technology for providing network communications security. The protocol secures communications between two parties who are directly linked to each other, and the encryption process results in a private connection because the data is encrypted using symmetric cryptography [62].

2.6 Conclusions and Thesis Motivation

Authorization is definitely a major component of the security field, especially when dealing with IoT applications and private 5G networks. In this section I attempted to share an introduction about security, described the major threats and challenges of the IoT world and then provided a lot of detailed information regarding authorization, different techniques and the OAuth 2 protocol.

OAuth 2 has become a standard in authorization scenarios, offering a plethora of configuration options in order to fit in any system possible without compromises. OpenID Connect covers the issue of authentica-

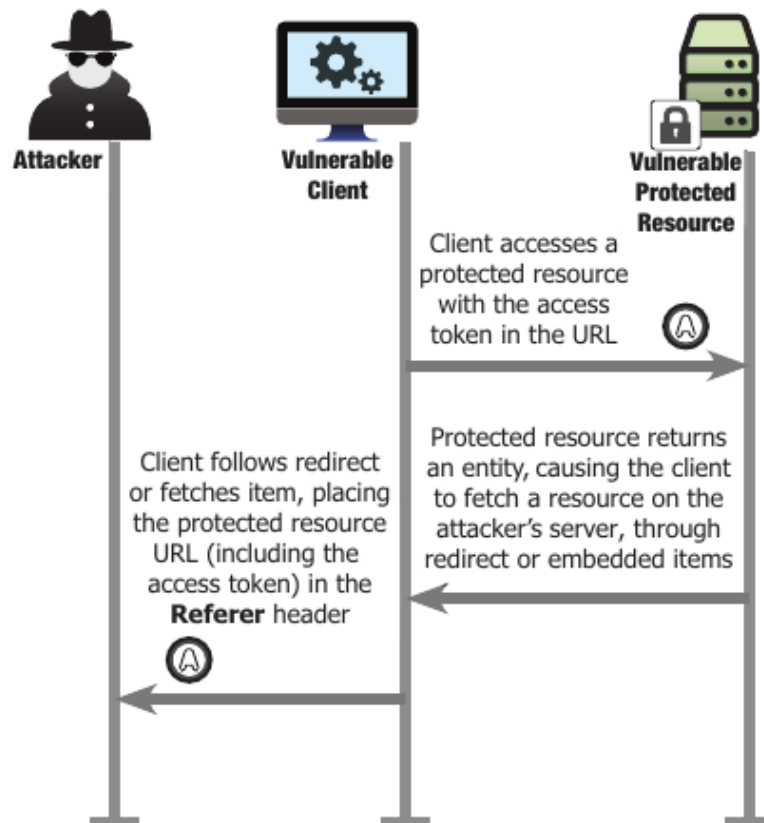


Figure 2.22: Hijacking of access token through the query parameter [41]

tion if that's needed along with the authorization mechanism. Although there has been a lot of revisions and upgrades to OAuth since version 1.0, there are still some vulnerabilities and issues that any developer working with OAuth should be aware of, as described in the previous subsection. With the right tools, configuration and security measures, it is possible for an obsolete authorization-authentication system to be designed and exploited while limiting the security flaws and data leakage scenarios.

With all these in mind, I was inspired to design and build my own instance of authorization system. This was conducted over a six month period while I was having my internship done at the University of Bristol. There was an ongoing development of a framework based on private 5G networking, smart city-like IoT automation and lightweight security with the vision of achieving optimal performance on the final product. I was assigned to implement the authorization protocol using modern technologies and open-source libraries wherever possible.

After researching and getting familiar with the idea behind authorization and understanding the challenges and weaknesses that exist, I was ready to start developing the authorization system I was asked for, based on the OAuth 2 protocol. In the next sections there will be a detailed description about how I proceeded through this project, I will present details on the whole framework that me and my team built and mention all the different security protocols that we needed to combine in order to accomplish the required security levels.

Chapter 3: Implementation and Integration of the OAuth 2 protocol on 5G MEC

When preparing this thesis, I knew that I would have to undertake a practical implementation of the main subject as well as to perform a number of experiments and present their results in order to come to a conclusion and sum up my thoughts and techniques that I decided to make use of. Having said that, I was lucky enough to combine my thesis with my internship at Smart Internet Lab (SIL) at the University of Bristol. I worked in a team of four people with excellent communication, cooperation and equally fair task assignment. Additionally, because SIL is a research lab with state-of-the-art equipment, we had the ability to use very powerful servers and different types of devices that were required in order to setup the whole framework and configure it appropriately.

In order to have a full understanding of the authorization functionality and see how it can be exploited in a real world case, I decided to implement it on the framework mentioned in the previous paragraph, as this was my internship's goal as well; the actual authorization implementation. While the framework was built on a private 5G network, authorization was one of the most important security protocols that needed to be applied in order to achieve the desired control on the communication between the devices and the REST API as we wanted to make sure that all incoming requests from the various devices are made after the user's consent.

In this section of the thesis, I will describe in detail the framework on which the authorization protocol was applied by showcasing various diagrams and explaining why each different component exists and what is their functionality. Additionally, there will be more insight on the specific OAuth 2 provider that I chose to utilize and how it was configured in order to work exactly as needed for this case. There will be technical information and code provided regarding the authorization techniques applied as well as some other important parts of the project that I am allowed to showcase.

It must be noted that I did not build this framework all by myself. A large part of it (not the authorization mechanism) had been already implemented before I started contributing to it when my internship began. For this reason in the following subsections I will be referencing and representing not only myself but also my team, because without them my thesis and the final results would have been significantly less interesting.

3.1 Architecture Design and Framework Components

In this particular subsection, I will demonstrate and analyze all the different components of the project's framework as they are seen in Figure 3.1 and how they relate to/influence one another. There will be technical details regarding the libraries used for each component as well as demonstration of their main functionality wherever needed.

3.1.1 Users

One of the main goals my team and I had was to provide users full control on their devices and stored data while being able to manage which devices are authorized to send information on their behalf. For this particular reason, a user can perform a lot of actions through the dashboard which is going to be

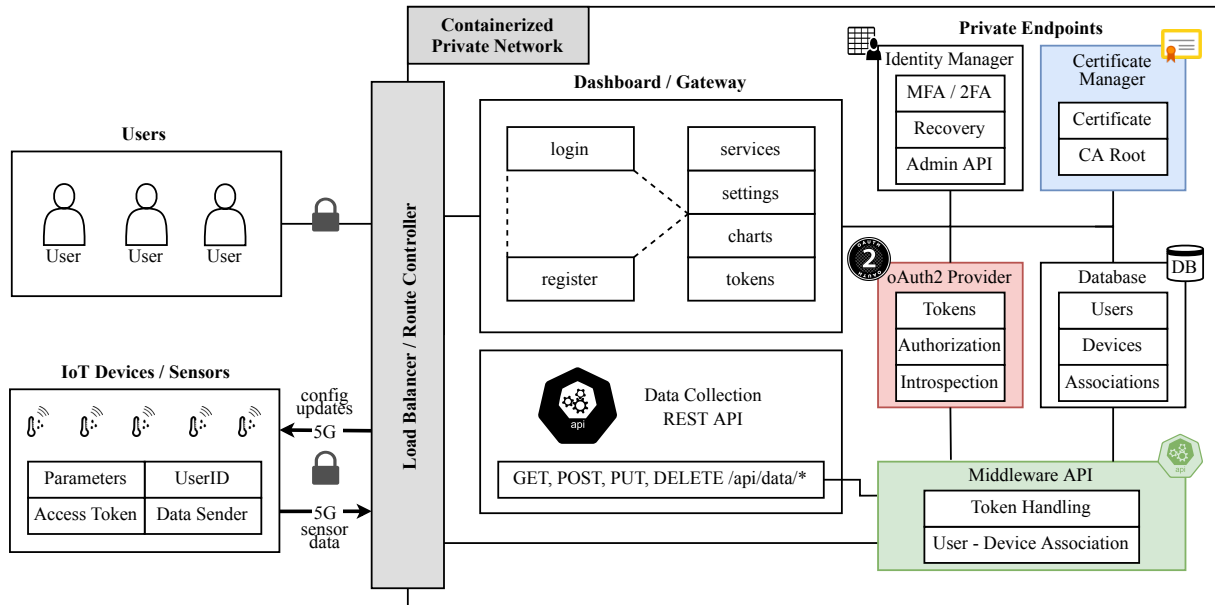


Figure 3.1: Framework's architecture diagram

thoroughly described in its own subsection. The main role of the users is to decide and consent to have their different devices retrieve their unique access token in order to be able to make authorized POST requests to the middleware API. The logic behind the users' actions is to provide them with a self-service user-friendly interface where they will be able to manage their devices, personal information, preview real-time data graphs etc.

3.1.2 Dashboard

The dashboard is a web-based control panel where users can register, login, manage their registered devices, preview stored and real time data, export and visualize metrics based on a selected time frame, allow/reject device authorization and configure their personal information. The whole dashboard is built with Typescript along with ExpressJS for the back-end functionality and Handlebars library for the front-end, but we are currently building a new version of it based on Angular and WebSockets. The dashboard being is casually updated and enhanced in order to provide a more user-friendly experience and offer new features to the end users. The following subsections are going to describe how the dashboard depends heavily on the identity manager, OAuth 2 provider and certificate manager in order to serve all the different features mentioned earlier.

3.1.3 Identity Manager

We decided to use ORY Kratos for the identity management as it is an open-source API friendly system built according to cloud architecture best practices. It packs a wide variety of features that are essential for any modern application, some of which are: Self-Service login and registration, multi-factor authentication based on RFC 6238 [63] and RFC 4226 [64], account verification and recovery, profile management and some useful admin APIs in order to mass import, update and delete identities. ORY is a very developer-friendly framework with systems like Kratos and Hydra which were used in our project.

Another key feature that made Kratos the perfect identity manager for this case is the fact that it can store user information in JSON format which is the same as our database structure. The form of the user identity schema can be found in Subsection B.1.1. Lastly, ORY Kratos can be also controlled through an Software Development Kit (SDK) which makes back-end configuration even easier and more reliable.

3.1.4 Certificate Manager

In terms of encryption, the goal was to implement custom certificates in order to utilize the HTTPS protocol and encrypt the whole communication channel. We used self-signed OpenSSL based certificates that originate from our server, meaning that the CA is our server [65]. Due to the fact that our CA is not a pre-installed third party authority in the devices, we have to manually register the CA as a trusted authority. In order to do that, the device needs to fetch the CA file from our API and subsequently proceed to manually add the CA file to the store. Due to the large number of devices we have to work with, we use a zero-touch tool automating the process of registering the CA in the local trust store therefore the encryption procedure takes place independently for every new device added.

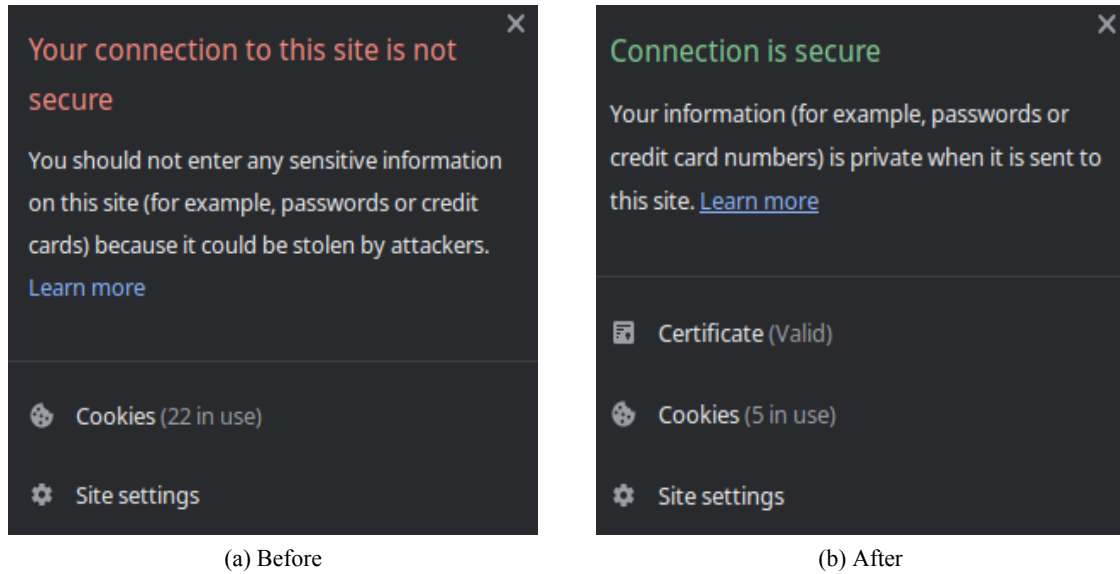


Figure 3.2: Browser status before and after implementing custom SSL certificates

3.1.5 OAuth 2 Provider

After settling for the OAuth 2 protocol as analyzed in Chapter 2, I had to look for an already built provider that makes use of it and can be easily integrated with the rest of the project. Similarly to the identity manager system, ORY offers an authorization system as well, along with all the useful API endpoints and SDK integration tools. Hydra is based on one of the most popular authorization protocols, RFC 6749 [8]. Specifically in our case, each user gets an access - refresh token pair through the authorization code flow, which is then being distributed to their desired IoT devices. Then, the devices make use of their user's access token in order to authenticate themselves and make an authorized POST request to the middleware API. The validation of each device's token is being handled by the introspection endpoint, which follows the RFC 7662 [18]. The whole end-to-end process of POSTing data is being visualized in Figure 3.3 with different colors representing each flow depending on the access token's validity.

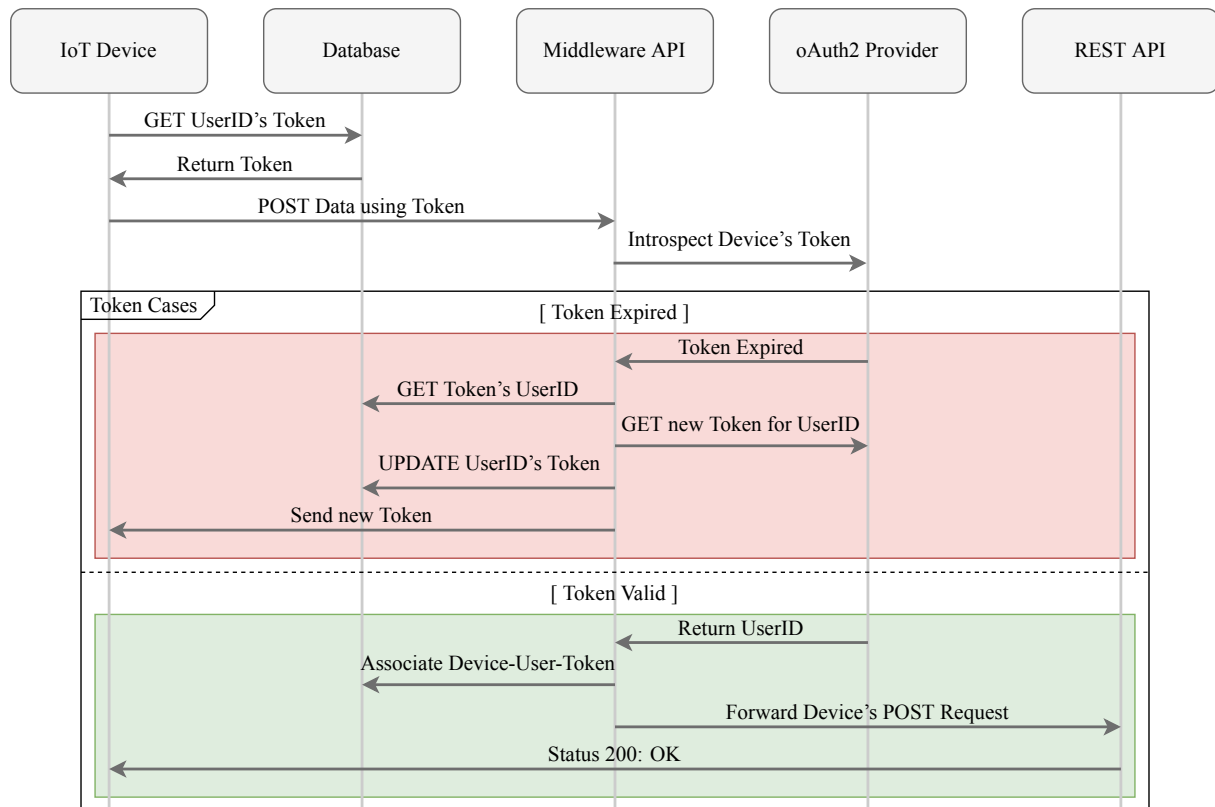


Figure 3.3: OAuth 2 token usage and handling

The key feature of the access tokens mentioned above is that we make sure they contain information about the user they were issued for, their unique userID to be specific. Due to this fact, we are able to share only the users' access token to their different devices, so as to protect as much confidential information as possible. When the devices use their token while POSTing data, the Introspection endpoint checks the token validity and returns the userID information being encrypted in it. This way we can make a number of checks and associations with the user - device - token trio. Lastly, except for the admin's back-end access to the OAuth 2 provider, users can also interact with it through the dashboard, being able to view their current active access token or request a new one in case of expiration.

3.1.6 Database

Being one of the fastest and most versatile database systems, MongoDB was our choice for storing the data of this framework. As mentioned in Subsection 3.1.3, MongoDB stores data in JSON format, thus allowing for easier access and data manipulation from our back-end scripts and SDKs. We use this database system in order to store user identities, user tokens, user-device-token associations, devices' data and performance metrics. Furthermore, we are using a number of database schemas so as to achieve document uniformity. Specific examples can be found in the Annex Section B. In terms of performance, MongoDB is optimized for write performance, and features a specific API for rapidly inserting, updating and deleting data concurrently, thus prioritizing speed over transaction safety, where in other RDBMS systems like MySQL, data needs to be inserted row by row.

3.1.7 Middleware API

Potentially the most important component of all, due to the fact that acts like the connecting link between the rest of the components. An API written in TypeScript, responsible for rejecting any unauthorized communication attempts, handling access token expiration events and replying back to each endpoint with the appropriate status codes and messages. There is a specific flow being followed in this middleware API: When a device makes a POST request using an access token, the API checks whether the token is active or not. If it is, a new association is stored in the DB with the user-device-token trio. The exact model of the device record is presented in Annex Section B.3.2.

Algorithm 1 Middleware API handling device requests

1: procedure POST(<i>token, id, data</i>)	▷ Handle Device POST Requests
2: <i>valid</i> ← introspect(<i>token</i>)	▷ OAuth 2 Introspection Endpoint
3: if <i>valid</i> then	
4: associate(<i>id, user, token</i>)	▷ Save in Redis and/or Database
5: forwardRequest(<i>id, data</i>)	▷ Forward data to REST API
6: response(200, "OK")	
7: else	
8: response(403, "Token Expired")	
9: procedure GET(<i>token, id</i>)	▷ Handle Device GET Requests
10: <i>user</i> ← checkAssociation(<i>token, id</i>)	▷ Check Redis and/or Database
11: if <i>user</i> then	
12: <i>token</i> ← generateToken(<i>user</i>)	▷ OAuth 2 Token Endpoint
13: updateDB(<i>user, token</i>)	▷ Update Redis and/or Database
14: response(200, <i>token</i>)	
15: else	
16: response(401, "Unauthorized")	

Then, it forwards the device's request to the data collection REST API in order to store the sensor's valuable information. If the access token is expired on the other hand, the API checks if the user has a newer token. If they do have one, it responds back to the device providing the new access token to be used; if they don't, then it goes on and gets a new access token for the user by utilizing the refresh token stored in MongoDB. Then, the API updates the user's active token and once again responds to the device with the new access token. This flow being visualized in Figure 3.3, allows the middleware API to keep the users' devices updated with the latest access tokens and also log any real-time errors or misconfigurations in order to maintain the end-to-end desired functionality and successful communication.

Something that should be noted for the middleware API is its ability to cache some of the information in order to limit the database connections. Because the incoming requests might be of large scale, we decided that it would be for the best if we used some kind of caching functionality. That's why we integrated Redis. Redis is an open source, in-memory data structure store, used as a database, cache, and message broker. Redis provides data structures such as strings, hashes, lists, sets, bitmaps, streams etc. so as to fit in any environment that can make use of it. In our case, just like the rest of the endpoints, Redis runs on its separate container with its own dependencies.

Firstly, we create a connection with redis inside middleware API. Then we save everything there in terms of tokens used and user-device-token association. If a token is invalid, the API then connects to MongoDB

and retrieves the refresh token in order to request a new token pair. Then, both MongoDB and redis cache get updated, in order to avoid further database connections for the rest of the devices with the same token. This way we manage to save a lot of unnecessary requests and database connections, thus replying faster to device requests and achieving lower RTTs.

```
const options:
  ClientOpts = {
    host: '172.24.1.20',
    port: 6379 };

let client: RedisClient = createClient(options);
```

Figure 3.4: Initiating a redis connection in the middleware API

3.1.8 Data Collection REST API

As a final destination for data, we used the Orion Context Broker (OCB) which is an implementation of the NGSIV2 REST API binding from Fiware. OCB allows us to manage the entire life cycle of context information including updates, queries, registrations and subscriptions. It is an NGSIV2 server implementation to manage context information and its availability. Using the OCB, we are able to create context elements and manage them through updates and queries. In addition, we can subscribe to context information so when some condition occurs (e.g. the context elements have changed) we receive a notification and act based on that (e.g. notify the user or perform a configuration modification).

One very important aspect of OCB is its CORS feature, which is an HTTP-header based mechanism that allows our devices to indicate their origin and category, in order to be handled accordingly and stored in the the correct database collection. CORS defines a way in which a device and our server can interact to determine whether it is safe to allow the cross-origin request. It allows for more freedom and functionality than purely same-origin requests, but is more secure than simply allowing all cross-origin requests. A real example of how OCB saves each device's data in MongoDB can be found in Annex Section B.2.1.

3.1.9 IoT Devices

Various IoT devices and sensors being spread and distributed in any desired environment that meets the user's needs. They can be of any type, from a raspberry pi module to a simple temperature sensor, all functioning with the same code so as to communicate with our servers though a private 5G network. The only information they have access to is their user's token and the scripts written for them in order to achieve the continuous data POSTing functionality. They can be associated with a user either through the user dashboard or the admin back-end API and start sending data right away. Their main functionality and logic is presented in the Algorithm 2.

3.1.10 Containerized Network and Load Balancer

In terms of network setup, we decided it would be optimal to use a containerized environment as each endpoint is built differently and has unique requirements. A container is a standard unit of software

Algorithm 2 IoT device functionality**Require:** $token \leftarrow$ Owner's Access Token**Require:** $delay \leftarrow$ POST Interval

```

1:  $Session \leftarrow$  Initiate new TLS Session
2: while True do ▷ Infinite Loop
3:    $sendData(Session)$ 
4:    $sleep(delay)$ 
5: function  $sendData(Session)$ 
6:    $r \leftarrow Session.POST(token, id, data)$ 
7:   if  $r == 403$  then ▷ Current Token Expired
8:      $r \leftarrow Session.GET(token, id)$  ▷ Request New Token
9:     if  $r == 200$  then
10:       $token \leftarrow r.token$  ▷ Replace old Token

```

that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. In order to achieve this containerized functionality, we used the Docker engine which is responsible for creating containers based on any given environment configuration (e.g. Ubuntu, Alpine, Python etc), managing the different container images, providing a bridge network between containers and the host machine as well as different IP addresses and ports to each container, and lastly easy access & debugging capabilities on each endpoint separately so as to facilitate the development process. All these features and setup are displayed in Figure 3.5. Docker is ideal for this type of projects because it is fast and reliable, without having to install all the dependencies locally on the host machine. It also ensures the same functionality and behavior across different OSes and computers because its core implementation is always the identical, no matter the environment.

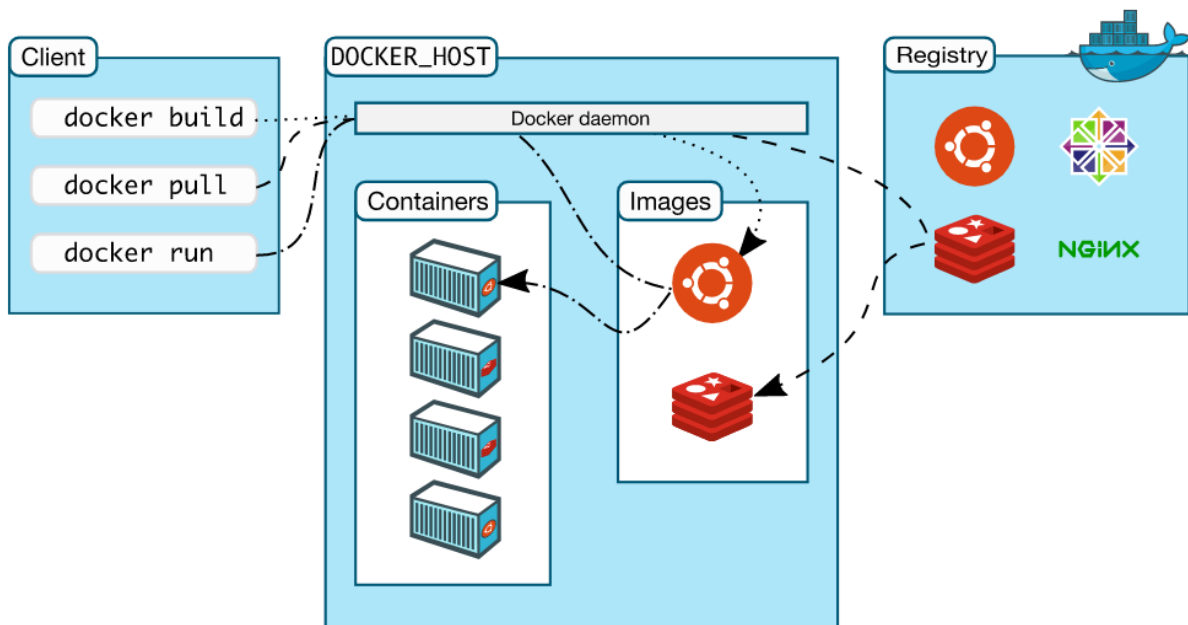


Figure 3.5: Docker engine architecture [66]

One important configuration option that is provided to us with the use of Docker, is controlling the access levels between the containers and their dependence relations. But as the project kept getting bigger, we needed a specific system capable of handling the load balancing for both internal and external commu-

nications. NGINX is probably the most popular and widely used software, built to serve similar cases like ours. It is an open-source project which was firstly released back in 2004. It can be used as a web server, reverse proxy, load balancer, mail proxy and HTTP cache. NGINX uses an asynchronous event-driven approach, rather than threads, to handle requests while its modular event-driven architecture can provide predictable performance under high loads. Some of its core features include: handling a lot of simultaneous connections with low memory footprint, handling static files, index files and auto-indexing, reverse proxy and caching, load balancing, TLS/SSL connections, WebSockets, URL rewriting and more. In terms of mail proxy features, it packs all the popular protocols like SMTP, IMAP, STARTTLS etc. NGINX's general load balancing functionality can be observed in Figure 3.6 and in Figure 3.1 for our specific case.

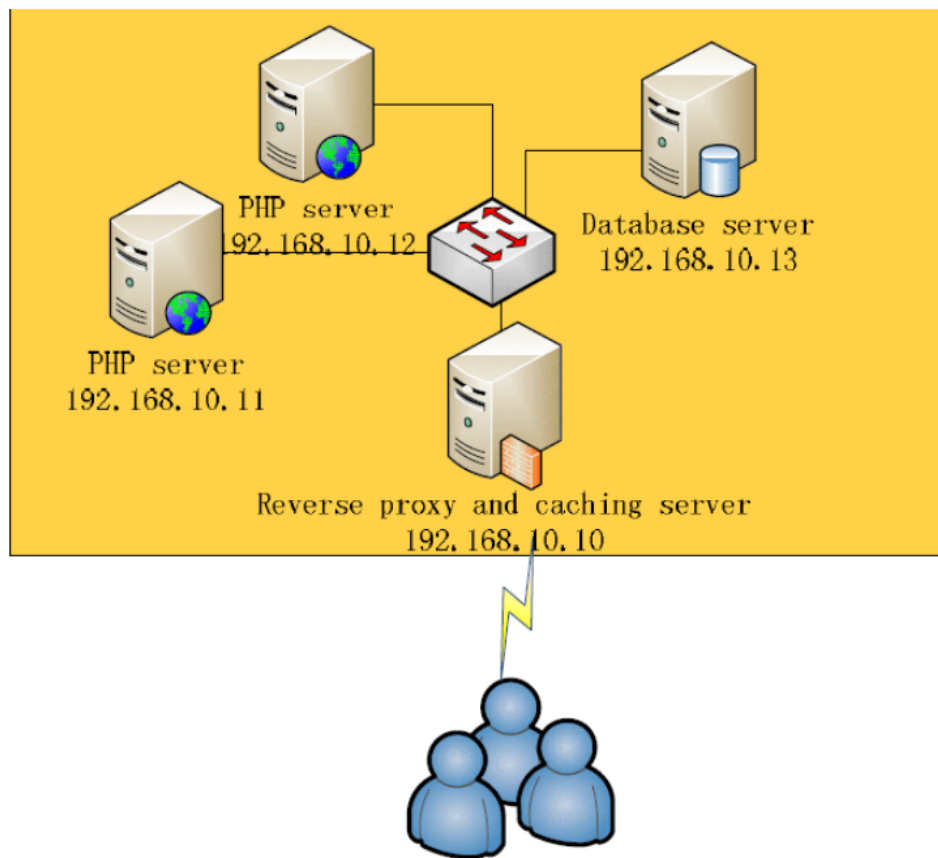


Figure 3.6: NGINX architecture example [67]

3.2 Challenges

Before starting the implementation of the OAuth 2 on the existing framework, there were some challenges that we had to face. These challenges emerged when we were planning the framework's security levels and isolation between the various endpoints, as we wanted to achieve a rigid ecosystem without any data leak risks, while preserving respectable performance. Having said that, we were directed on the correct path of integration and selection of the technologies used. Some of the main challenges are listed as follows:

1. **The system should be able to provide self-service abilities for users to register, authenticate themselves, set the login method and reset their password.** This challenge refers to the identity manager, which was presented in Subsection 3.1.3. Kratos offers all these features, allowing the users to register, login from different interfaces, reset their password and finally authenticate themselves in order to use the dashboard, access the protected sub-pages and have displayed their belonging information.
2. **An OpenID OAuth provider server must be separately configured within the network to provide truly cloud based security and authentication.** ORY Hydra was the key library to face this challenge. As it is developed by the same team as Kratos, the configuration and setup process was quite similar but the implementation had a lot of different approaches. This was mainly caused because we wanted to slightly modify the default operations of the Hydra authorization provider and integrate the whole token issuing process in one single click of the user dashboard. The security is indeed cloud based as the whole authorization endpoint is hosted on a Docker container, protected behind and NGINX load balancer.
3. **The user login information and attributes must be separated from the sensor database and communication between them should be utilised using a REST API in a hardened network.** This was achieved with the use of different MongoDB collections, the OCB's CORS structure and the middleware API's functionality. User login information and attributes are stored in an independent Kratos database, separated from any other kind of data, and can be accessed only from the authentication endpoint. The sensors' data is stored in a specific MongoDB database, following a predefined schema and stored based on the OCB REST API model.
4. **The backend and middleware should be able to provide information on who the user is and his traits or attributes.** The way we managed to fulfill this goal was by using the Hydra tokens, Kratos tokens and session cookies. As mentioned later in Subsection 3.4.1, we are able to extract information from all of these three identification means. We use this user identification information in order to display the correct data on the dashboard (frontend), to store the incoming sensor data in the appropriate database collections (middleware) and make any modification needed in terms of user permissions and associations with sensor devices (backend).

3.3 Authorization Integration

In this subsection a technical analysis will be given based on the authorization library that I chose to use, the way I configured it, the exact flow which needs to be followed for issuing an access tokens and how it was achieved as well as discuss the authorization integration with the rest of the framework and specifically the user dashboard.

3.3.1 ORY Hydra

In Section 3.1.5 I mentioned the authorization library that was used in order to attain the desired results for our security implementation, which is the ORY Hydra system. This authorization library is built based on RFC 6749 [8] and is exactly what we needed for the purpose of my thesis and internship assignment.

Unlike other OAuth 2 implementations, Hydra implements the OAuth and OpenID Connect standard without forcing us to use a "Hydra User Management" (login, logout, profile management, registration), a particular template engine, or a predefined front-end. This allowed us to implement our own user management and login systems, with the specific authentication mechanisms required for our needs. Furthermore, Hydra has a low CPU and memory footprint, short start-up time, and scales effortlessly up and down on many well-known platforms like Docker. With all that being said, I came to the conclusion that this was the perfect fit in terms of authorization for the framework I was working on.

3.3.2 Configuration

As mentioned in Section 3.1.10, the whole framework consists of various separate containers, each one running independently. This case applies on the authorization endpoint as well, with its whole back-end code and sub-systems protected by the NGINX load balancer and being accessed only by the admins and the OAuth 2 clients. In terms of configuration, Hydra offers a lot of options and parameters that can be modified but in our case we only needed to change a few.

Firstly, the Time To Live (TTL) of the access and refresh tokens needs to be configured based on the application's needs. In our case, we decided that 1-hour is a good TTL for the access token, as it provides acceptable security levels with the constant change of the token. As for the refresh token, it must always have a higher TTL if we want it to be used for acquiring a new access token with the same information stored inside it without having to redo the whole authorization code grant flow (Figure 2.15) all over again. For this specific reason, we have set the refresh token's TTL to 30 days. This could definitely be set to a smaller time frame, but it is of small importance. When the refresh token is used, we get a new instance of both the access and refresh tokens, so the old ones are no longer valid.

Secondly, in the YAML configuration file of Hydra we have also included the paths of the various endpoints needed for the authorization flow. Specifically, we have set the login and consent page paths, which are going to be discussed in detail in the next subsection, as well as the token issuer's path which is the NGINX IP of the Hydra container.

3.3.3 Login and Consent

Subsection 2.4 clearly described that there is a login and a consent stage that the user has to go through in order to have an access token issued to them. In this subsection I will analyze how I handled and implemented these two stages and how I managed to include unique user identity information in each access token.

The authorization techniques are mainly used from a host application in order to provide access to an external application to control a user's assets for a specific amount of time (token TTL). In our case, we wanted the same logic to be applied between a user and their devices: If the user allows a device to use their access token, the device should be authorized to collect data on the user's behalf.

For the reason stated in the previous paragraph, there is no need to show the users the login and consent screens. Firstly, the users must be already logged in to the dashboard in order to visit the token sub-page

and secondly, the interface they are interacting with is asking them what devices should be allowed to use their access token. With these two simple techniques, we managed to automate the whole authorization code grant flow in a single click of a mouse from the user's point of view. Let's now observe what is going on at the backend.

We can easily modify what the user has to interact with but we cannot change things like RFC standards. This means that as described in Section 2.4, the OAuth 2 provider will be waiting for a successful login and consent response in order to generate an authorization code. To achieve that, we integrated the Hydra admin SDK into two custom made sub-pages of the user dashboard, one for the login and the other for the consent stage. When a user reaches the token sub-page, they are presented with their current token information. If they have none or an expired token, the user can ask for a new one with the press of a button. When they do this, our first custom page is triggered and after verifying the user's active login session, we make use of the `hydraAdmin.acceptLoginRequest()` function in order to complete the typical OAuth 2 login procedure. Inside this function we are able to set the "subject" value of the access token, in which we have decided to store the user's unique identifier, `userID`. After that, the request gets redirected back to Hydra and then to the consent sub-page.

```
hydraAdmin.acceptLoginRequest(challenge, {
  subject: ai.claims.session.identity.id
})
```

Figure 3.7: Embedding user's unique ID inside the access token

In the consent sub-page, we basically have to perform the same actions as with the login, except for setting the "subject" value of the token. We use the `hydraAdmin.acceptConsentRequest()` so as to accept and forward the successful consent process back to Hydra. With the login and consent actions successfully performed, Hydra replies back with an authorization code which we then use in a final request in order to exchange it for a new access token, specifically made for the particular user and stored in MongoDB using the format showcased in Annex Section B.3.1. With the whole process taking less than a second, the last thing the user has to do is to decide and choose which of their devices are allowed to make use of their access token.

3.4 Integrating authorization with other components

Authorization is a security protocol that can be integrated almost everywhere. Once the backed configuration has been successfully set up along with the various endpoints that are required for all of the OAuth 2 operations, the point where the authorization sequence will take place is up to the developer. This allows us to control the amount of "free ground" an incoming request is allowed to cover before having its access token validated and then proceed as authorized. Due to this freedom, there are a lot of different points where we can integrate the authorization security protocol in our particular framework. All these different authorization possibilities will be described in the following subsections.

3.4.1 Hydra & Dashboard

A very comprehensive custom feature that I wanted to include in the framework, was the ability to access the dashboard's protected sub-pages (the ones that a user can access only after logging in) with the use of their Hydra token. By default, the dashboard supported two ways of confirming a user's active login session: In the case of browser, with the use of session cookies and in the case of API requests, with Kratos tokens issued by the dashboard after an API login flow.

With a little bit of work, I managed to add a third way, almost identical to the default API one, but this time with the use of Hydra tokens. As they both have a specific amount of time they can be exploited, also known as TTL, one can securely use a Hydra token for authenticated API interaction with the dashboard. When the dashboard detects a Hydra token in an incoming request, it goes on and examines it at the Hydra introspection endpoint. The result, whose structure was actually presented in Figure 2.13, includes the token's validity as well as the embedded unique ID found in the access token, as described in Section 3.3.3. Finally, with this information at hand, the dashboard can provide the appropriate content for this particular user (userID).

Using Hydra tokens to access protected dashboard pages is equally safe to Kratos tokens because in order to acquire a Hydra access token in the first place, a user must firstly prove they are logged in with one of the Kratos' default methods, API token or browser session cookie, so the authentication process must be performed at least once for each Hydra access token. What we achieve with this integration is to skip the constant Kratos authentication process and only use the Hydra's access tokens if a device or external application needs to access one of the dashboard's protected endpoints.

```
location = /someEndpoint {
    auth_request /_check_authentication; }

location = /_check_authentication {
    internal;
    js_content nginx_functions.handleAuthentication; }
```

Figure 3.8: NGINX auth_request configuration

3.4.2 Hydra & Docker

In Subsection 3.1.10 I mentioned Docker and how we used this engine in order to separate the different endpoints while controlling how they influence each other. OAuth 2 Hydra system was similarly installed in its containerized environment and not installed locally on some form of server. Providing it with all its required dependencies and relations with the rest of the containers, Hydra is running independently, thus ensuring even better protection and security to the endpoint responsible for the authorization functionality. Moreover, as this container is given a unique IP and port by the NGINX load balancer, further control can be implemented by rejecting any suspicious external requests targeting the Hydra endpoint.

3.4.3 Hydra & NGINX

Another very significant implementation that can be exploited is delegated authorization access right inside NGINX engine. If implemented correctly, we can perform the introspection process in NGINX and based on the results reject or forward the incoming request. This strategy can be applied on all requests or selectively on different NGINX routes. This works with the usage of `auth_request` (Figure 3.8) and `Subrequest()` (Figure 3.9) [68] functions which allow the system to read the authorization header's bearer token and, if it exists, make a subrequest to the OAuth 2 introspection endpoint in order to examine the token's validity. Then, based on the result, NGINX will allow or reject the incoming request, replying with the appropriate error message and status code in the case of false introspection result.

```
function handleAuthentication(r) {
  r.subrequest("/_hydra_introspect", function (reply) {
    if (reply.status == 200) {
      if (reply.active == true) {
        r.return(204); }} // Token is Valid
    else {
      r.return(403); }});}} // Token Invalid or Expired
```

Figure 3.9: Subrequest function in NGINX javascript file

Chapter 4: Performance Evaluation

This section will be divided into four main subsections. Firstly, there will be a detailed description of the testbed that we developed for this project in University of Bristol along with its requirements and challenges that we faced during the implementation. Then, I will be explaining the need of a device emulator that I designed and how this tool helped us achieve an almost real-like environment of users and devices in order to stress-test our framework and find potential misconfigurations and/or apply further improvements. Continuing, I will be presenting the experiments that were planned and executed on different versions of our framework and setups, by firstly explaining all the different experiment parameters that we had at hand, and finally various results collected will be displayed along with detailed explanations and comparisons.

There will be a large scale of figures and tables that will be constantly referenced in the different following subsections. These are mandatory for this specific chapter as they facilitate the reader's understanding by visualising the results and comparing them side by side wherever is necessary. Apart from the text, all of the content displayed in this section will be accompanied by labels and captions, explaining each presented graph.

4.1 Testbed Details

Developing IoT applications and frameworks is a challenging task in terms of software setup and end-to-end network architecture. In order to be ready to face any challenges and requirements that may be needed, we designed and took advantage of a testbed built specifically for this matter. In order to have a better understating of the platform that we worked on, some goals and requirements of the testbed must be analyzed in this section. Then, the experiment parameters are going to be analyzed along with their effects on the final results which will be also presented with various graphs and figures in one of the last subsections.

4.1.1 Requirements

The main administration requirements [69] will be covered in this subsection so as to have a better understanding of the features we had at hand while building the framework and preparing for the experiments. These requirements are mostly related to development efficiency, configuration ease and future expandability. The University of Bristol owns state-of-the-art equipment and is funded to conduct a lot of research projects similar to this one. Having said that, we had to build a serious and powerful testbed, able to cope with our requirements accordingly. The main requirements that we had set for the testbed are described in the following list.

1. **Maintenance** is a very important requirement as it serves critical concern of admin operations and error management. Its features keep improving throughout the lifetime of the testbed, while offering both physically and remotely access to the building blocks. For this matter, the testbed must make use of a reliable network that allows secure remote connections for any kind of maintenance

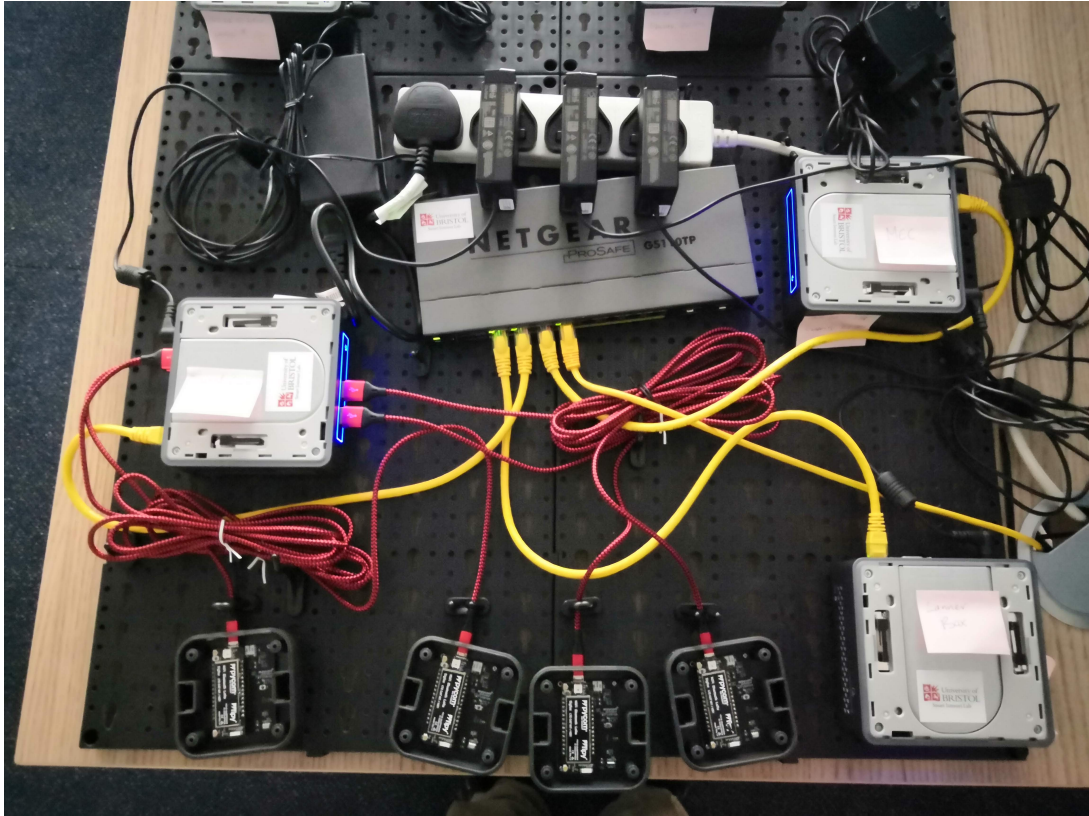


Figure 4.1: Bristol University testbed equipment

and supervision. Additionally, the administration must be able to manage and monitor the service provisioning for the testbed users.

2. **Heterogeneity**, which is also a main characteristic of the IoT, is another strong feature that the testbed should have because of the different locations and configurations that the various experiments are part of. This means that the hardware, wireless technology and software can differ for each particular test. In order to achieve heterogeneity, the testbed must be able to offer a wide range of wireless technologies and communication protocols. In addition, by orchestrating user software to the edge of the network, the devices have the ability to communicate through software agents that are generated by the different experiments.
3. **Extensibility** is probably the most important requirement of all, as it allows for testbed growing, improvement and further development to meet the needs of a specific experiment. This means that the testbed is designed to be future-proof and able to accept changes both physically and virtually, especially with use cases like cities and other public places that the needs are altered based on population and other similar variables. In order to achieve that, the testbed should have space for more sensors, edge servers, wireless APs etc. Furthermore, virtual space and compute power are also mandatory so as to host multiple applications running concurrently. One way to achieve this is by adding more physical nodes and by orchestrating the applications, in order to "recycle" resources.
4. **Concurrent Usage** is obviously required by the testbed in order to run multiple and different experiments at the same time, without throttling or experiencing data collisions. These issues can

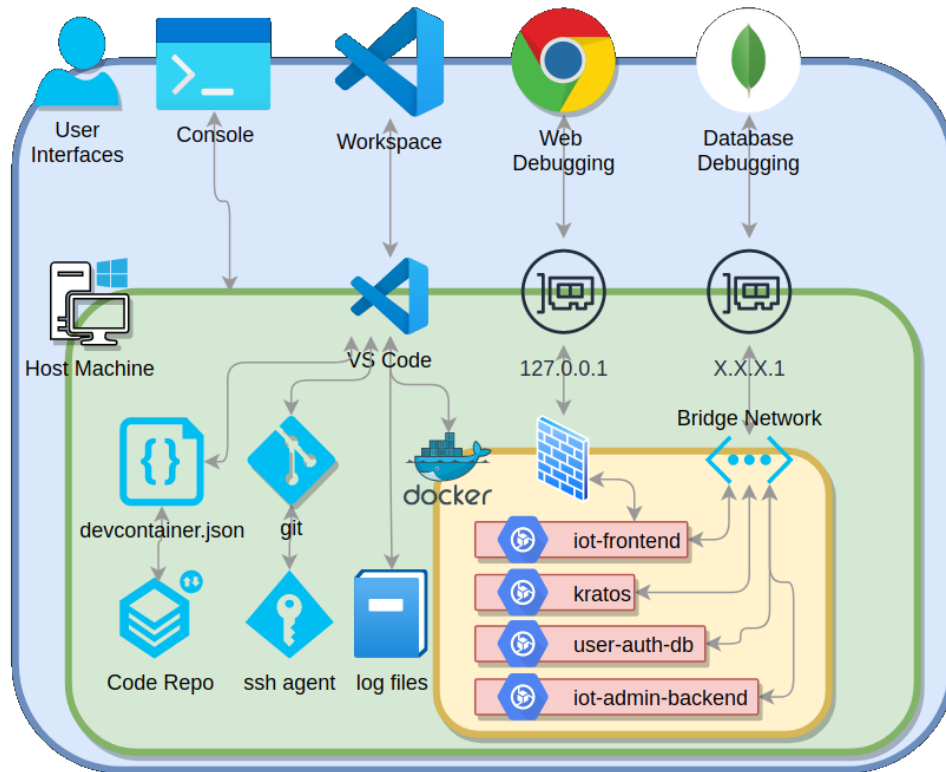


Figure 4.2: Part of the testbed's development environment

be coped with after carefully examining each experiment's requirements and managing the provided resources accordingly. Moreover, the testbed must provide means for verification and cross validation between use case results. Such tools are most of the times front-end control hubs that visualize data and predefined Key Performance Indicators (KPIs).

5. **Interoperability** refers to the basic ability of different computerized products or systems to readily connect and exchange information with one another, in either implementation or access, without restriction. In addition, we needed to create a testbed which would be able to run on different types of environments and CPU architectures, from a compute node with multiple GPUs and configurable CPU Cores to a mobile ARM system. This characteristic enhances scalability because it allows the testbed to allocate any given specifications, which obviously influence the performance but does not prevent the framework from being executed on different types of machines.

4.1.2 Design

What we wanted to achieve with this testbed was to combine different components with different dependencies into one rigid and monolithic framework. This means that although the back-end components might be of a completely different implementation and can be managed separately, the end result will look like a native end-to-end built platform. This way we can easily configure or even change whole parts of the framework by replacing the containers we use and modify their functionality in order to meet the latest standards while still respecting the latest RFCs. This cloud based micro-services schema can be compiled by any OS and CPU architecture (e.g. x86, ARM etc) without any compromises.

As it is built using the Docker engine, the testbed is setup through a `.env` file and various YAML configuration scripts that run when initiating the testbed's operation. Additionally, it can operate without being abstracted by the OS, on any machine with no dependencies, apart from those of a Linux system. This means that the testbed can run natively on Linux and MacOS machines, while Windows requires the Windows Linux Subsystem (WSL) to be installed and configured in order to have access to the Linux kernel. Docker's containers are consisted by 2 parts/processes, the image and their runtime.

1. Image

A Docker image is a file used to execute code in a Docker container. Docker images act as a set of instructions on how to build the container, like a template. Images also act as the starting point when using the Docker engine. An image is comparable to a snapshot in VM environments. Docker images have multiple layers, each one stems from the previous layer but is not identical to it. The layers enhance Docker build performance while increasing reusability and decreasing disk use. Image layers are also read-only files. Once a container is created, a writable layer is added on top of the unchangeable images, allowing a user to make any desired modifications.

A Docker image has everything required in order to run a containerized application, including code, config files, environment variables, libraries and **runtimes**. When the image is deployed to a Docker environment, it can be executed as a Docker container. The `docker run` command creates a container from a specific image. Docker images are deployable on any host. Developers can take the static image layers from one project and use them in another. This saves the user time, because they do not have to recreate an image from scratch.

2. Runtime

Docker runs processes in isolated containers. A container is a process which runs on a host. The host may be running locally or remotely. When an operator executes `docker run`, the container process that runs is isolated, which means it has its own file system, its own networking, and its own isolated process tree separate from the host. By default, a container has no resource constraints and can use as much of a given resource power as the host's kernel has available. Docker provides ways to control how much memory, or CPU a container can use, setting runtime configuration flags of the `docker run` command.

Each endpoint/container is optimised to run with the minimum required dependencies and packages, which are installed and executed when we initiate the testbed's operation. This ensures that the computing power needed is kept reduced, thus providing better performance and less disk space allocation. Moreover, if some equal dependencies are required by different containers, the installation process takes place only once, and the libraries are then shared between these containers. Lastly, this Docker setup of the testbed can be easily transferred and executed on a Kubernetes environment with minimal configuration alterations, which means that the framework will be running across a cluster, while Docker runs on a single node.

4.1.3 Private 5G Networking

The most important feature of the Testbed is its ability to exploit 5G networking. This is mandatory when working with a large amount of IoT devices that must be able to communicate simultaneously without congestion.

The modularity, programmability and flexibility of the 5G system architecture enables realization of 5G networks that support communication applications with heterogeneous requirements. This along with particular requirements of vertical industries, such as strict security and performance requirements, have led the Bristol University to acquire and make use of a private 5G network. In addition, 5G offers an excellent support for machine-to-machine communications and IoT connectivity which is ideal for our use case.



(a) Outdoor 5G antenna



(b) Indoor 5G router

Figure 4.3: Outdoor and indoor 5G network equipment

Furthermore, using a private 5G network ensures optimal communication performance. In contrary to WiFi which can be congested by overlapping channels, 5G cellular networks use a licensed spectrum of communication which is strictly undisturbed. This means that the performance provided by the 5G is predictable and can be trusted for continuous, uncompromised usage. Having said that, these characteristics make the testbed and the whole framework built on it, rigid enough to be used in enterprise level and consumed by real users.

4.2 Device Emulator

Once there was not enough amount of real sensors distributed and installed at the time of my internship and thesis preparation, we had to find a way in order to simulate multiple devices to connect and interact with our API by sending data concurrently. To solve this issue, I took the initiative to implement a tool that will do this exact operation. Motivated from a couple of similar projects, I began constructing the Python Device Emulator (PDE), using scripts and the proprietary Kratos and Hydra SDKs that were

mentioned in Subsections 3.1.3 and 3.1.5. In order to work natively with the rest of the framework and achieve optimal compatibility, I decided that it would be for the the best if I used Python to build this type of project.

The finished project resulted in an emulator that scales on realistic levels and can even replicate inconsistency in terms of packet quality, delay and network jitter. The way that I managed to emulate multiple devices from one single tool, the libraries and technologies used as well as the whole simulator flow and functionality are going to be thoroughly described in the following subsections.

4.2.1 Python Threading

When it come to parallel workloads, Python has a great variety of options depending on the type of work that needs to be done simultaneously. The main "candidates" for the task at hand were threading and multiprocessing. While both of these allow for parallel job operations at the same time, there are some key differences that made me choose one over of the other.

Python threads are units of work that run independently of one another. They don't correspond to hardware threads on the CPU, though—at least not in Python. Threads are controlled by Python's Global Interpreter Lock (GIL), so they run serially. Because only one Python thread runs at a time, they are a useful way to organize tasks that involve some waiting. Python can execute thread A or thread C while thread B is waiting for a reply from an external system, for example.

Python processes are whole instances of the Python interpreter that run independently. Each Python process has its own GIL and its own copy of the data to be worked on. That means that multiple Python processes can run in parallel on separate hardware cores. The trade-off is that a Python process takes longer to spin up than a Python thread.

I hope it can be clearly understood that Python Threading is a way more efficient, scalable and fitting library for our case. With the use of it, we managed to create large numbers of devices at the same time and provide them different types of information in order to function as needed for our experiment purposes.

4.2.2 Configuration

The PDE needed to be versatile and easily configurable so as to act accordingly in all the different ways we wanted it for the various tests that took place. I thought that one the best and straight forward practices to quickly control and change the PDE's behaviour is through a JSON configuration file (config.json) that would contain all the different variable parameters. At the time of writing this thesis, the configuration file consists of the parameters listed in Table 4.1.

There is also another JSON configuration file (names.json), containing random, non-existing users' names and surnames which are used by the emulator in the preparation scripts in order to generate the user emails, usernames and passwords, then register the desired amount of users and and finally log them in and get an access token for each one. The final generated information is stored in a third, separate

```

{
  "users": 10,
  "devices": 50,
  "url": "/api/data",
  "device_name": "iot-device-",
  "delay": 5,
  "minutes_duration": 10,
  "tls": true
}

```

Figure 4.4: PDE's JSON configuration file example

Table 4.1: PDE's configuration parameters

Parameter	Type	Description
Users	Integer	Number of users to be created
Devices	Integer	Number of devices for each user
URL	String	URL of the HTTP target
Prefix	String	Device name prefix
Delay	Float	Dealy between device messages
Duration	Integer	Operating duration in minutes
TLS	Boolean	TLS Session Flag

JSON file (registered.json) which concludes the whole configuration file-system.

4.2.3 Preparation Scripts

Before running the main code of the emulator and create the desired number of devices, we need to prepare our system with the use of three essential scripts in order to generate the required information for the emulation process and most importantly acquire the initial access tokens that are going to be later distributed to the users' devices.

Firstly, we have created a script in order to "reset" the PDE's state and ensure that it is ready to operate without any past information and miss-configuration. The **delete_users.py** script is responsible for deleting a number of previous experiments's data: saved metric records, registered users, DB documents and additional file-systems created during the test. After this cleanup, the following scripts take action and prepare the PDE for the new experiment.

Secondly, the **register_users.py** script is run with its, obvious, task being to register the desired number of users that we have set in the configuration file. After parsing the names.json file and retrieving all the names needed, it goes on an registers the users by utilizing the Kratos admin SDK as mentioned in Subsection 3.1.3, after generating the appropriate emails and usernames by concatenating the name and surname fields. Lastly it creates the registered.json file and stores all the successful registered users' information which will be used by the next script.

Last but definitely not least and potentially the most important script of all, **login_users.py** plays a huge role in this preparation stage. As the final step, the previously registered users need to get logged in and

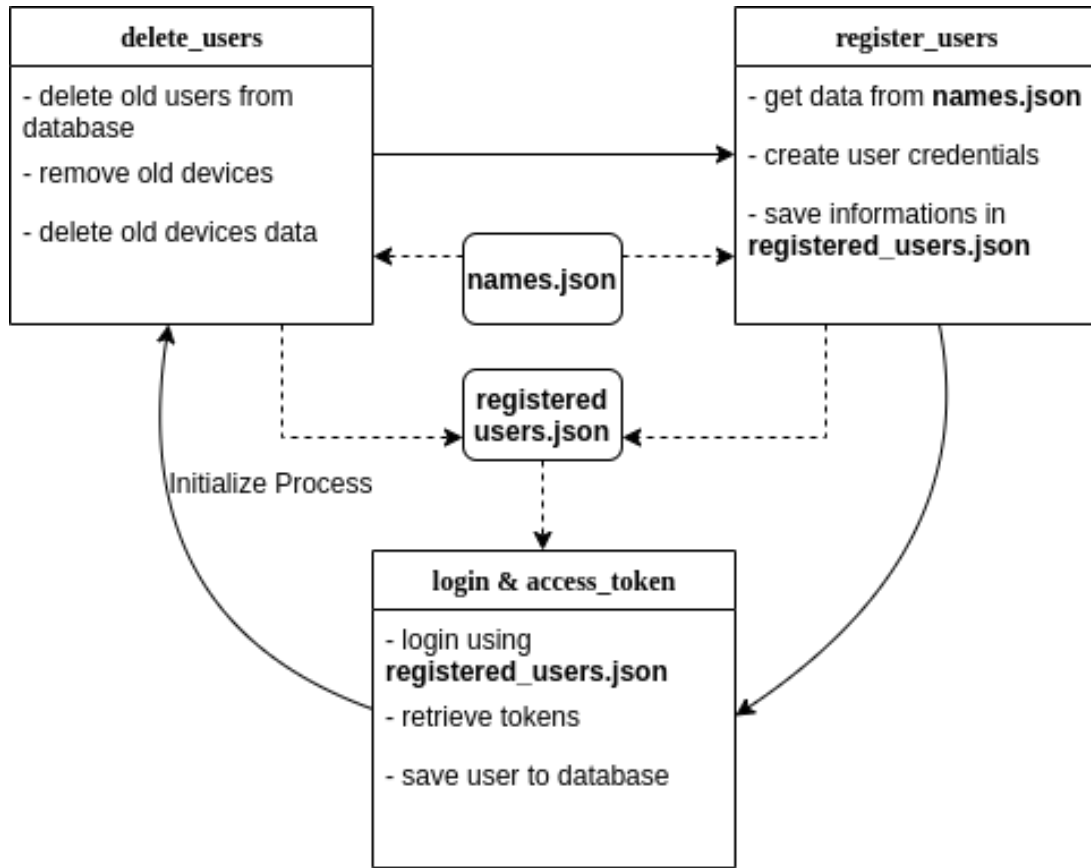


Figure 4.5: PDE's preparation script operations

have an access token assigned for each one of them. Just like with the registration script, we make use of the Kratos and Hydra APIs in order to login the users and initialize the authorization code flow (Figure 2.15) for each one. As a result, we get an access token for each user, with their unique userID encrypted in it. Finally, we save the generated access tokens in the registered.json file along with the rest of the users' personal information.

4.2.4 Supervisor & Device Components

Except for the three scripts mentioned in the previous subsection, the PDE consists of two object classes that are mandatory in order to achieve the desired flow and functionality.

The single-instance **Supervisor** object is the main controller of the PDE. It gets created right after successfully running the preparation scripts and is given the configuration parameters from the config.json file. The supervisor is responsible for creating all the devices and passing the appropriate parameters to each one. In this stage, we can configure the supervisor object to calculate unique message delays for each device, based on a Poisson-like distribution model. After having setup the desired number of devices (threads), it runs until reaching the duration time specified in the configuration file. Another crucial task executed by the supervisor is the download of our self-signed SSL certificate. After saving the certificate locally, the devices are able to locate it and make use of it in order to communicate through HTTPS connections. More details regarding supervisor's object structure, procedure of creating the devices and main functionality can be found in Annex Section A.1.

The multi-instance **Device** object, which is basically a thread object, contains the main functionality of the emulation process, as it represents an actual user's IoT device that would have been deployed somewhere in order to collect and report any kind of data (temperature, humidity, noise levels etc). It makes use of the given configuration parameters acquired by the supervisor and runs indefinitely until its supervisor stops (based on the given experiment duration). Should the TLS parameter be set to true, the thread will take advantage of the TLS session protocol and use the SSL certificate once in order to initiate a secure communication channel. This channel is then repeatedly used and results in the limitation of network handshakes and thus, enhanced performance. TLS handling as well as the rest of the technical details regarding the device object can be found in Annex Section A.2.

4.2.5 Measuring the performance

Another very important feature of the device emulator is that we were able to log each device's performance separately and save it locally after the experiment had been executed successfully. All of the results that are going to be demonstrated in Subsection 4.4 were collected with the use of the device emulator. An easy way to save the device metrics while without using a lot of resources is with the use of Comma-Separated Values (CSV) local files. During an experiment run, when a device sends data for the first time, it also creates a sub-folder for storing its own performance results. For our experiments purposes, we decided that information shown in Figure 4.6 was all we needed in order to supervise the framework and generate all the required figures for the result comparisons.

```

1  EVENT;DURATION;RESPONSE_CODE;TIMESTAMP
2  NT;0.75151;200;2021-09-10 08:51:38.022686
3  NT;0.06862;200;2021-09-10 08:51:41.062944
4  NT;0.01833;200;2021-09-10 08:51:43.826319
5  NT;0.10968;200;2021-09-10 08:51:46.717786

```

Figure 4.6: CSV log file example

The four different pieces of information that we are saving are pretty much self-explanatory. What we really care about is the **DURATION** field because it represents the actual RTT of the specific event. We can filter out each line based on the event codename and status code in order to extract the desired metrics and results. More information related to all the possible event codenames we have created and from which status codes they are caused can be found in Table 4.2.

Table 4.2: Device possible events and status codes

Description	Codename	Status Codes
Normal Transmission	NT	200, 201, 204
Re-Transmission	RT	
Device Recovered	DR	401, 403, 422
Packet Loss	PL	

4.2.6 Other Applications

Before moving to the next sections, another use of the PDE should be referenced. Apart from providing a plethora of emulated devices that report data, a second instance of it can also be used for penetration testing purposes. Without the requirement of the preparation scripts as there will not be any users involved, the PDE can be easily setup to cause traffic on our receiving server and try to reduce the framework's overall performance and liability. Solutions and techniques on how to overcome such situations will be discussed in a following section.

4.3 Experiment Details

Our experiments were conducted in various configurations so we can examine how each of the setups cope with the framework while monitoring the overall performance of our system regarding the RTTs, latency and reliability. We have set two different environments for our tests in terms of networking and location, VLAN and Physical Switch. For the VLAN, we would like to establish the baseline of the expected RTT. This makes sense for small device/user count where the performance of the system is not affected by the presence of the emulator on the same machine.

In order to understand the large scale implications of our framework and further examine our KPIs, we also ran some experiments for longer duration of time and larger device count, therefore we can establish an understanding of the deterministic overhead introduced. This computational overhead, which also translates into active time on the load-balanced services, can be managed by Network Function Virtualization (NFV) orchestration, thus NFV resource allocation and more specifically network slicing and load-balancing can drive the overheads down to meet application specific requirements.

The main goal of the experiments was to measure the performance of the final framework, after having applied various light weight security protocols on it, and study how it is affected by the different experiment parameters that are going to be described in detail in the following subsection. Additionally, we desired to achieve a realistic testing environment which would result in measurements similar to those of enterprise implementations. Lastly, through these experiments we were able to make additional modifications and improvements on our framework in order to achieve better performance metrics and security levels. All of these results along with comparisons between old and new revisions of the project are going to be showcased in Subsection 4.4.

4.3.1 Experiment Parameters

In this subsection I will list all the different experiment parameters, as shown in Table 4.3, which we had the ability to configure and how these affected the final results. This is a very important section if one wants to have a complete understanding of the experiment environment conditions and observe the large impact that some small configuration alterations might have on the final experiment results. Additionally, a few more parameters that are not included in Table 4.3 but are worth mentioning and impact the final results, are going to be discussed.

1. API Config

Table 4.3: Experiment parameters

Experiment	A	B	C	D
API Config	VLAN	VLAN	Physical Switch	Physical Switch
No. of Users	10	10	10	10
Devices per User	50	50	50	50
Authorization	Disabled	Enabled	Enabled	Enabled
Data Rate per Device	Poisson Distribution [6ppm - 24ppm]			
Exp. Duration	10 mins	10 mins	10 mins	16 hours

API Config defines the configuration and location of the server receiving the HTTP POST requests from the devices (or from a PDE in our case), which is basically the whole framework we developed with middleware API being the initial receiving endpoint. As shown in Table 4.3, two different types of API Config were used throughout the four experiments that took place: VLAN and Physical Switch. The difference between these two is the location of the framework and PDE. This means that during both VLAN experiments, the framework and the PDE run under the same server, sharing the same resources whereas in the Physical Switch experiments the whole framework was operating on a separate server from the PDE.

In the VLAN setup, we make use of Docker’s Virtual Switch networking structure. As Dzogovic et. al [70] purposes, one of the shortcomings of the Docker networking structure is the utilization of VXLAN, layer-2 virtual bridges and other network overlays. The effect of this flaw will be observed in the following Subsection 4.4, where experiment B results in a slightly higher RTT average compared to experiment C. Experiments C and D were executed on the Physical Switch setup, which means that the PDE is running on a different server, separated from the framework. Their communication is emulated using a para-virtualized 1Gbit network adapter which makes the two servers act like being located in different LANs from one another.

2. Number of Users and Devices

The number of users and devices influence the experiments directly both during the stage of preparation (Subsection 4.2.3) and during the runtime. Firstly, if more users are required for an experiment, then the process of registration, login and token issuing will take longer time, based on the capabilities of the host system of course. Then, for the number of devices per user, the PDE’s supervisor component is going to take a longer time to generate the threads and start the simulation process if the total number of devices is larger. Lastly, while running the experiment, there is going to be more traffic in the network due to the higher number of devices sending and receiving packets. This will cause the middleware API and whole backend system to work under heavier load, thus resulting in higher RTTs and latency.

3. Authorization

The parameter of authorization, which could be either enabled or disabled, describes whether the experiment is using the OAuth 2 authorization security protocol or not. In experiment A where it is disabled, the whole authorization implementation gets bypassed. The access token distribution, token introspection, user-device-token association and token refresh flows are not being performed. The avoidance of these security interactions make the whole system more vulnerable to attacks and

failure, as the devices POST their data directly to the OCB, without being authenticated or checked for their authorization status. Apart from losing security, the experiment results can, in some cases, be slightly better in terms of performance because of the less operations required for the end-to-end communication, but the security trade-offs are not worth the higher risk just for slightly better metrics.

4. Data Rate per Device

As mentioned in Subsection 4.2.4, the PDE's supervisor can provide the devices a fixed interval as regards to the package report frequency or generate a unique one for each device based on a custom Poisson-like distribution as seen on Figure 4.7. The latter is the method that we used in all of the executed experiments because it represents a more realistic approach to the devices' behavior rather than having them all send their packages simultaneously. In Table 4.3 we are measuring the data rate in Packages Per Minute (ppm), which means that the larger this measurement is, the higher the message frequency is for a specific device. Higher message frequency can cause more traffic on our network and have a negative impact on the overall performance of the framework, that's why we decided to set the lowest and highest ppm possible, using the limit of 6ppm to 24ppm.

```
# Range Possibilities
weights = [0.25, 0.5, 0.25]
# PPM Ranges
ranges = [(6, 11), (12, 20), (21, 24)]
# Pick a range based on given possibilities
r = random.choices(ranges, weights)
# Pick random ppm in selected range, convert to seconds, pass to device
device["delay"] = 1 / (random.randint(ranges[r][0], ranges[r][1]) / 60)
```

Figure 4.7: Custom Poisson-like distribution

5. Experiment Duration

Experiment duration is the parameter that had the smallest impact on the final results. After optimising the framework by making implementations like caching data in order to avoid unnecessary communications and improving the device's code, the framework managed to maintain the same performance no matter how long the experiment run for. For this matter, in Figure 4.11 a comparison between two experiments run for 10 minutes and 16 hours is displayed, showing that the final performance differences are minimal. Both of the experiments were configured identically, except for the duration, and were performed on the Physical Switch API Config.

6. TLS Session

In order to form a secure connection among the client and the other end, whether it is a server, an API application or a browser, a certain protocol should be applied encrypting the transmitted messages. The quoted protocol is TLS commonly known and referred to as SSL which establishes a secure connection between the two ends through a TLS session which is initiated after a TLS handshake [71]. The session is stateful, this signifies that the session can transfer data as long as it is alive without the need of repeating the TLS handshake. A less ideal solution to transfer data securely, would be to establish a new TLS connection in every client request. Even though this

is not erroneous, the proliferation of new sessions in a single interaction can amplify the network traffic and RTTs due to excess overhead data. To desist from this situation, we established a single TLS session in zero RTT mode that was presented in TLS 1.3 [72], allowing session resumption. This way the need for further TLS handshakes is expunged and the overall latency reduced [73], promising better response times and performance for devices that require low latency, such as IoT sensors.

7. Redis Caching

Another parameter that is not listed in Table 4.3 because, technically, it is not a parameter rather than an improvement that was implemented as one of the last modifications we did on the framework, is the Redis caching functionality. However, it deserves to be mentioned for the fact that it provided much better performance results and made the framework even faster and more lightweight because it allowed for less connections with the database endpoint, thus resulting in less backend traffic and faster response times. In the next section, there will be a dedicated subsection in order to showcase the better performance results we managed to achieve with the integration of Redis and comparisons between the framework running with and without the caching functionality.

4.4 Experiment Results

In this section I will be describing in detail the various results we collected based on different experiments that were conducted. Each experiment alterations and modifications are going to be mentioned along with graphs and figures which will visualize the results for a better overall understanding and side to side comparison. The experiments that took place and will be showcased, are targeting in different measurements. There will be results based on the main topic of the thesis, which is the authorization implementation and how it affects the framework's performance, results based on the TLS resumption, comparisons between the different API configurations as mentioned in Subsection 4.3.1, comparison of the metrics with and without the caching functionality and so on and so forth.

All of the experiments were performed on the Bristol University testbed, which was thoroughly analyzed in Subsection 4.1, under the same conditions. There were some modifications on the framework either for testing under different use cases or for implementing improvements in order to achieve better overall performance. These alterations are going to be mentioned as this section is progressing. The motivation for these additional improvements was the results displayed in Figure 4.8. We noticed that after running experiment D for 16 hours, the RTTs were constantly getting higher. This occurred due to extensive load and because of the continuous connections to MongoDB. In order to overcome this problem, we implemented two solutions: TLS session resumption and Redis caching functionality. Both of these features were described earlier in Subsection 4.3.1 and will be also analyzed in following subsection as we will be measuring their performance separately.

4.4.1 Authorization Results

Experiments A and B were conducted specifically for measuring the authorization integration impact on the framework's performance. It must be noted that in these two experiments were executed after having

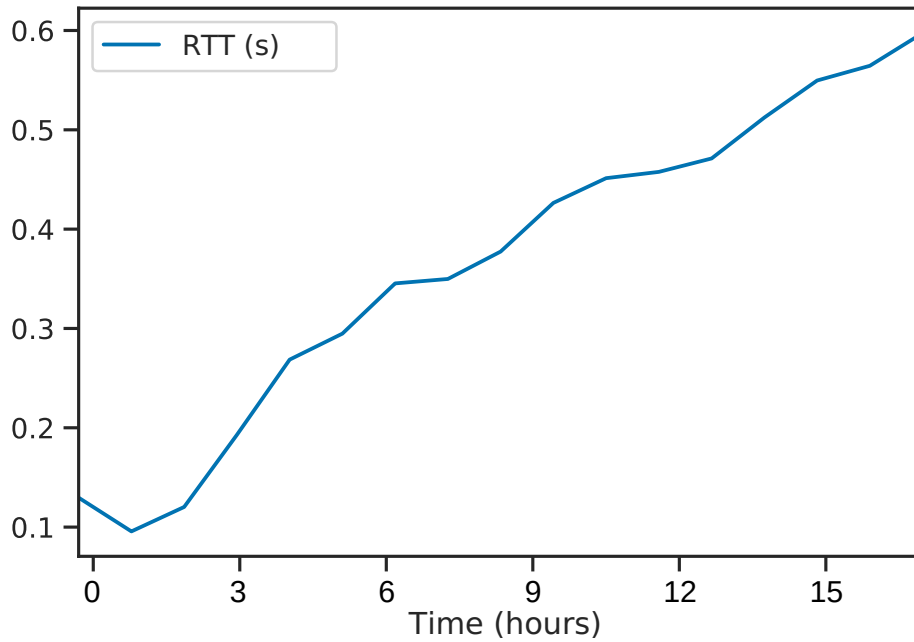


Figure 4.8: Experiment D’s performance without TLS resumption and redis caching

implemented the Redis caching functionality. Its impact will be explained in the following paragraphs and subsections. As noted in Subsection 4.3.1, when the authorization security protocol is enabled, the additional actions that take place consist of: token introspection, user-device-token association validation, token refreshing and new token distribution. In some cases, it is required for all of the above actions to be executed in order to perform the whole authorization flow, ensure the token’s validity and maintain the required security levels for the end-to-end communication. But let’s have a closer look at the experiment results and find out the size of the authorization impact on the framework.

Table 4.4: The impact of authorization

Experiment	Authorization	Lowest RTT	Highest RTT	Median RTT
A	Disabled	0.0314 sec	0.1159 sec	0.0607 sec
B	Enabled	0.0318 sec	0.1016 sec	0.0564 sec

In Figure 4.9 and Table 4.4, we are comparing the RTTs of experiments A and B. As seen in Table 4.3, these two experiments were ran under the same configuration so as to ensure they were not influenced by any other factor. Having disabled the authorization parameter means that the devices communicate directly with the OCB, without having to go through the middleware API in order to have their access token validated and perform the various operations described earlier. While we would assume that skipping all the additional authorization security flows will favor experiment A’s RTTs, taking a closer look at Table 4.4 will prove us wrong. The final results are almost identical, with experiment B achieving an even better median RTT. This was not always the case though.

In some earlier versions of the framework before integrating Redis caching, the authorization communications that had to be executed for every request of every single device, caused 10%-15% additional

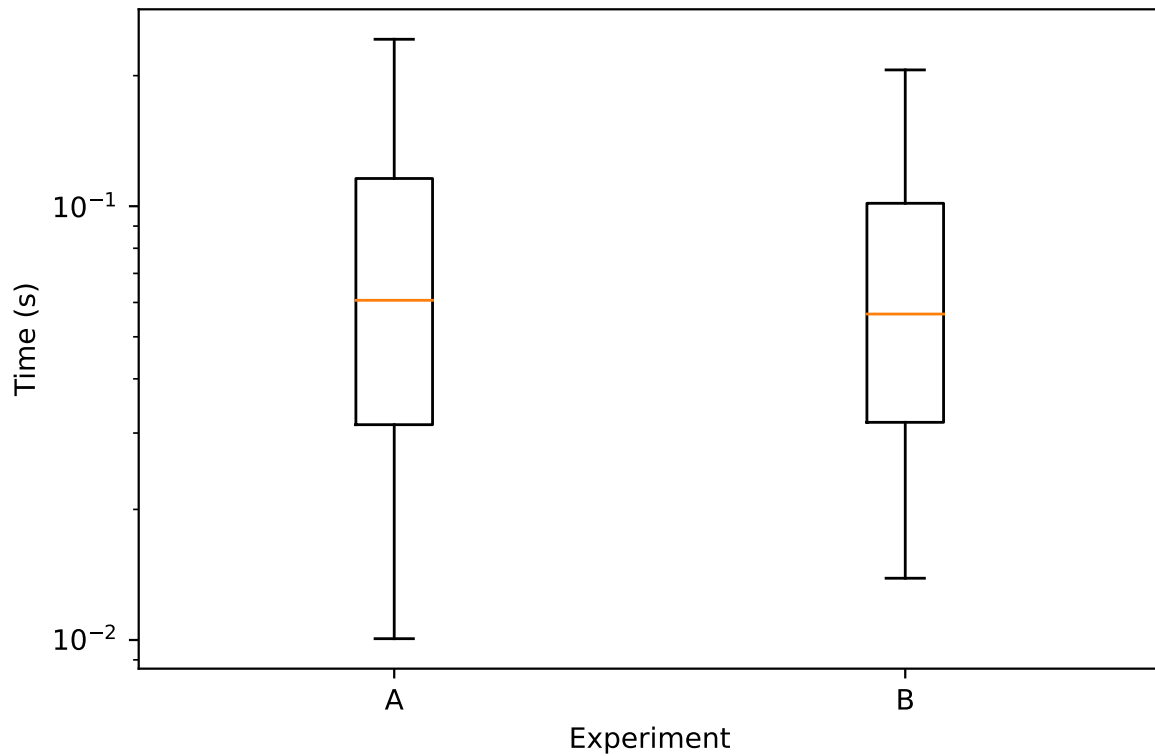


Figure 4.9: Authorization impact: Comparison between experiments A & B

overhead. One can say that the extra security measurements taken are worth the additional cost of time, but we knew there was still room for improvement. So with the use of Redis, we were able to exploit local memory database systems and avoid reaching for MongoDB in every device request. The results were extraordinary as we managed to configure the authorization processes optimally and achieve almost zero additional overhead added to the overall framework performance. These results were one of my thesis' and internship's greatest achievements and surely required a lot of time and effort to accomplish.

Next, with the use of Figure 4.10 and Table 4.5, the way that the RTT gets higher in the case of token expiration will be demonstrated. For this example, we ran Experiment C for 25 minutes with Redis caching enabled, and set the access token's TTL to 2 minutes. As a result, the framework had to perform the refresh token flow every two minutes and we were able to measure the average duration of this process along with the message retransmission times. When a device's token gets expired, the device stalls until the middleware API gets a new token for it and returns it back to the device, as described in Subsection 3.1.7. Then the device retransmits the same package that did not get saved in the database due to token expiration. The time taken for this extra automation, results in higher RTTs that can be clearly identified in Figure 4.10.

Table 4.5: Normal transmission vs re-transmission after token refresh

Transmission	TLS	Redis	Median RTT
Normal	Enabled	Enabled	0,02675 sec
Re-Transmission			0,07973 sec

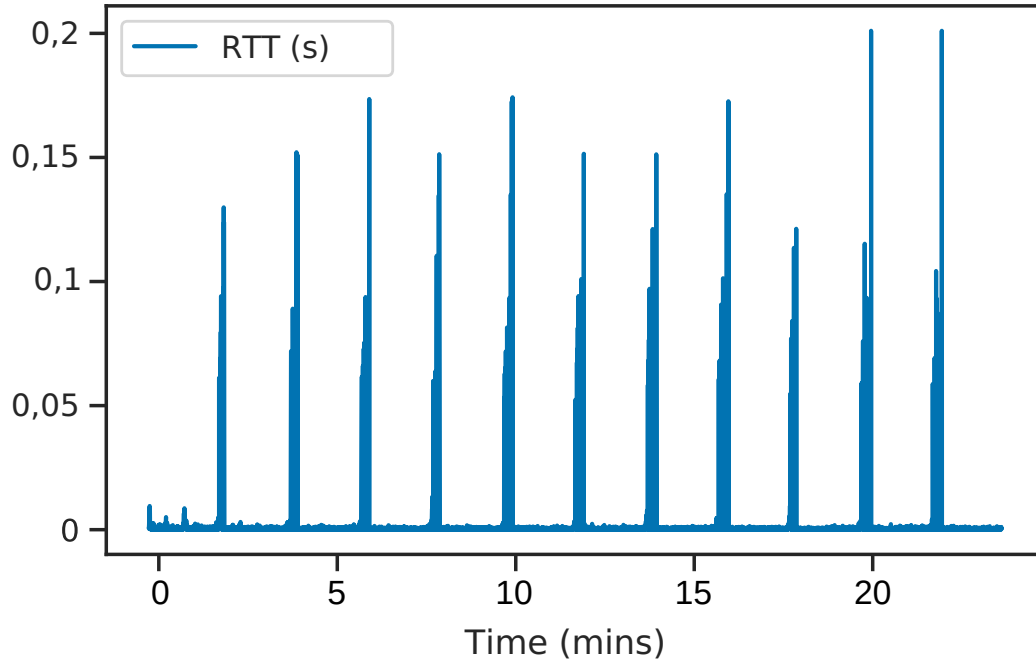


Figure 4.10: Token refresh visualization

Although it seems like a big 'spike' on RTT when the refresh token flow happens, in theory it is not. We could totally make the devices not wait while their token is being refreshed and have 2-3 packet losses due to expired token and rejected authorization. This could in theory work, but it would cause a lot of additional traffic on our network and there would be the risk of archiving the same device data multiple times, rather than once, thus decreasing the framework's overall reliability. So with all these in mind, we decided that it was for the best if we made the device wait while the middleware API gets a new access token for it and then retry the data POST procedure. If, for some reason, the packet fails to reach the middleware API during the retransmission phase, either due to a network problem or token error, only then we log this situation as packet loss.

4.4.2 TLS Session Results

TLS Session did not really get influenced by the different API configurations so there is no need to compare results of it between the two different setups. What needs to be demonstrated are the RTTs of experiments C and D, which had the exact same configuration, with the only difference being their duration, and compare their final results with and without the use of a static session. The concept of TLS Session and how it helps improve communication times by avoiding TLS handshakes, was analyzed in Subsection 4.3.1. Note that the two instances of the experiments C and D that will be compared in the following figure, were ran without the Redis caching functionality.

As it is clearly visualised in Figure 4.11, the results are significantly better when exploiting the TLS Session protocol. The RTTs are more than ten (x10) times or 1000% lower as seen in Table 4.6, which

Table 4.6: Impact of TLS resumption on experiments C & D

Experiment	TLS	Lowest RTT	Highest RTT	Median RTT
C	Enabled	0,0278 sec	0,0734 sec	0,0418 sec
	Disabled	0,2044 sec	0,7701 sec	0,4343 sec
D	Enabled	0,0220 sec	0,0372 sec	0,0282 sec
	Disabled	0,1389 sec	0,5346 sec	0,2946 sec

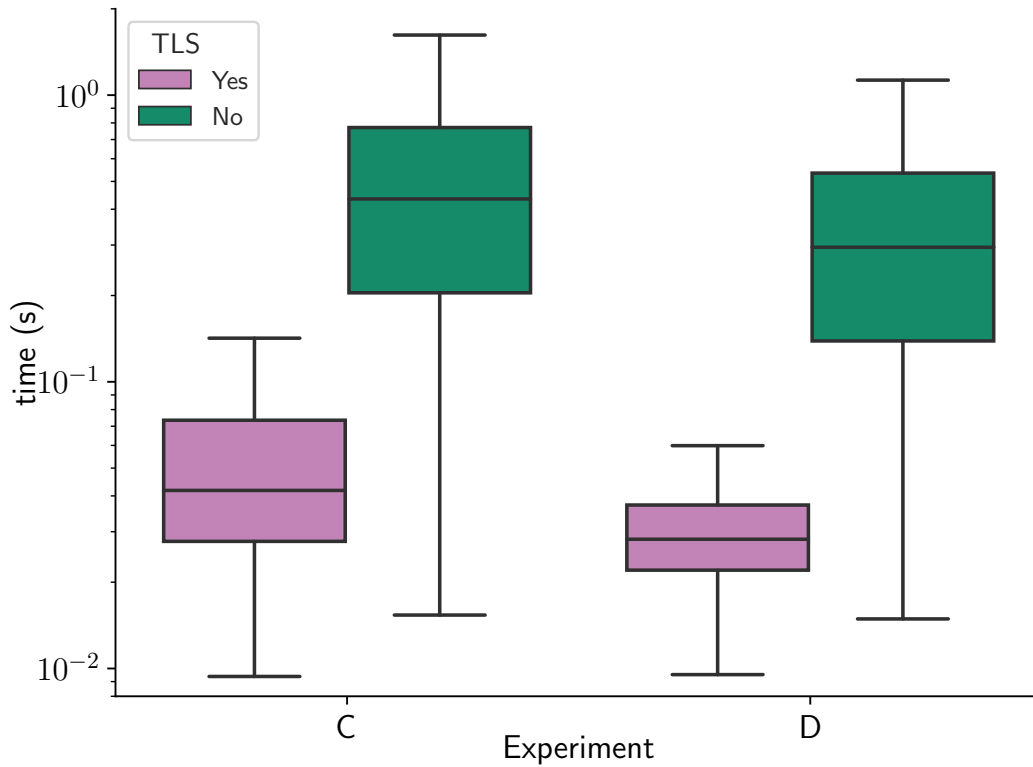


Figure 4.11: RTT measurements with and w/out TLS resumption

means that the devices have a much better response time and the framework gets lighter with faster performance due to the absence of unnecessary TLS handshakes.

4.4.3 Redis Integration Results

As mentioned in the beginning of this section, one of the latest improvements that we implemented on the framework is the addition of Redis caching functionality. We decided to look into this kind of implementation, because of the continuous rise of RTTs that we observed during experiment D, as displayed in Figure 4.8. As large amounts of devices are communicating almost concurrently with our middleware API, this also means that most of our backend services are also being used simultaneously in order to perform all the required operations for each device separately. In order to avoid some of these communications, save some network traffic and achieve better system resource allocation, we decided to integrate Redis, as described in Subsection 4.3.1, with the rest of our framework. This turned out to be a wise move, as we managed to achieve much greater performance and much more acceptable overall results.

Table 4.7: Impact of redis caching on experiment C

Redis	TLS	Lowest RTT	Highest RTT	Median RTT
Disabled	Enabled	0,0278 sec	0,0734 sec	0,0418 sec
Enabled		0,0198 sec	0,0387 sec	0.0272 sec

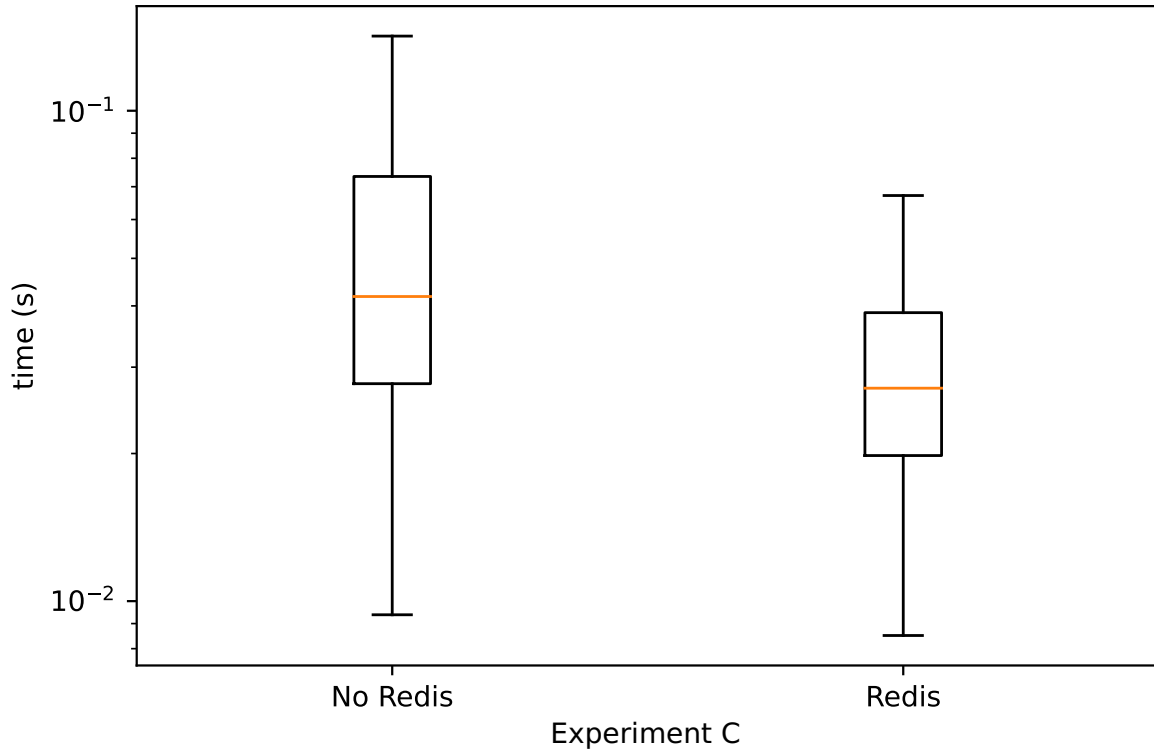


Figure 4.12: Experiment C's RTTs with and w/out redis caching

In memory databases provide a low latency high availability data access that can be load balanced and distributed in the network. Therefore individual Kubernetes pods or equivalent orchestrated services, like Docker in our case, can be benefited for database replication and high availability. Thus, if configured and optimised with the correct parameters, authorization flows and other processes can be executed seamlessly, with almost no additional overhead added to the overall performance of the framework. The results of the caching mechanism are listed in Table 4.7 and displayed in Figure 4.12. We managed to achieve almost double the performance as the final RTTs are almost half of the ones without the Redis caching integration.

4.4.4 Reliability - Packet Loss

Although the framework has reached a very stable form with minimal improvements left to be implemented in future releases, there are some rare cases that the communication between the devices and the framework gets interrupted and causes packet losses and device downtime. This can happen for a number of reasons: Firstly, if there is an error with the authorization endpoint, the Middleware API will not be able to introspect and refresh access tokens causing the failure of the whole flow and framework logic. Secondly, if the database, MongoDB, fails for some reason, the middleware API and OCB will raise a

number of errors because they will be unable to reach the data storage. Thirdly, if the actual middleware API fails, which acts as a connecting link to all the other endpoints, the framework will be totally unable to function.

We have developed, as expected, various fail-safe operations in order to overcome mostly all of the above error situations and minimize packet loss states in order to achieve a seamless and self-controlled functionality of the framework. Most of the packet loss scenarios in our framework were caused because a device's access token was already being refreshed, thus the particular device could not initiate the refresh operation and get the new access token as a reply which, sometimes, results in packet loss. This happens mostly when the refresh token operation takes longer than expected, which causes a timeout while the device is waiting for the new token.

During experiment B, which was conducted on the latest version of the framework with Redis caching enabled, we set the access token's TTL to 1 minute in order to have multiple refresh token operations and count the packet losses. Apparently, there were none! This means that the framework can operate on itself without any information loss and achieve high numbers of reliability. The self-service operation of refreshing the tokens happens seamlessly and re-distributes the new access tokens to the required devices. This concludes to the fact that any packet loss situations that may occur, will be mostly due to a problem of the specific wireless network (e.g. 5G, WiFi, Bluetooth etc) and not the framework's false configuration.

4.4.5 API Configuration Results - Interoperability

While this is definitely not one of the main aspects that we need to measure and showcase in this thesis because it is not closely related to security, noticing the differences in performance that can be caused due to the various API configurations is worth getting into. The main reason of this subsection is to demonstrate how the framework gets affected when running on the same system as the PDE, because of the limited computing resources. Although the Bristol University's servers are quite powerful, there is an still impact on the framework's performance when running under the same node with the PDE. In order to achieve a more realistic approach, along with measuring the framework's capabilities separately, we implemented the Physical Network setup, as described in Subsection 4.3.1.

Running the PDE and the framework on different servers, which means being separated and away from one another, does not necessarily lead to worse results. This happens for a number of reasons: Firstly, the whole framework is not affected by sharing the same resources as the PDE, which means that it can scale better and bigger because of the testbed's interoperability feature mentioned in Subsection 4.1.1. Secondly, The 1Gbit emulated network connecting the two servers allows for exceptionally high communication speeds which finally accomplishes even better results than running on LAN. We can definitely notice a better performance when running on the Physical Switch setup (Figure 4.13) and this is because of the interoperability feature of the framework. Due to larger resource allocation, all of the backend operations get executed even faster and with the combination of the private 5G network, the overall performance is better than the VLAN API Config setup.

With all these said, the main outcome we can take for granted from this subsection is the framework's

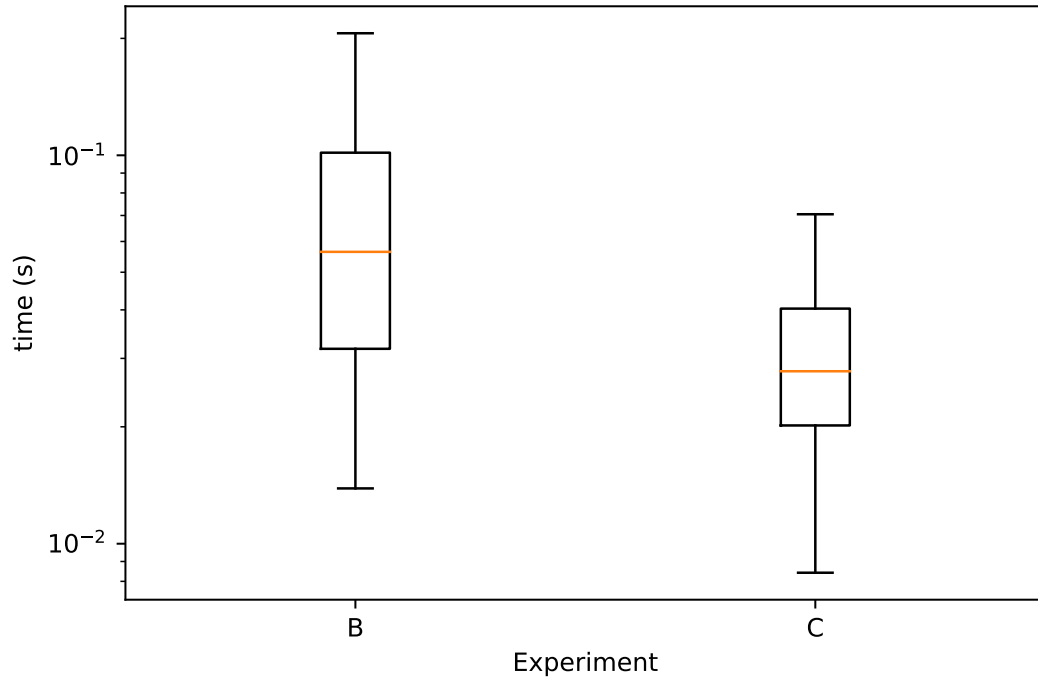


Figure 4.13: API Config impact: Comparing experiments B & C

ability to adjust its performance and adapt on any given system resources and network configuration. Either running on the same machine like experiments A and B, or on separated servers connected through a 5G network like experiments C and D, can result in equally acceptable results, always based on the host system's resources. For real-life usage though, we are only talking about physical switch scenarios and wireless communications as there is no chance for a device to be on the same network as the framework's backend systems.

Chapter 5: Conclusion

As this thesis comes to an end, there ought to be a summary of how the main topic was implemented and improved throughout the thesis, the main aspects covered in terms of literature and software development, an overview of the thesis statement, how it got dealt with and in what extent. The objective was to create a cloud native architecture based on a hardened 5G private network that includes scalability, interoperability, comprehensive privacy, and contemporary light weight security protocols. While there were some already implemented security measures, the major feature that distinguished this initiative is that it is designed to run on a 5G network with all of the libraries used being open-source. This means that security applications were examined and rebuilt to accommodate the 5G network's cellular structure while keeping the whole flow simple and light weight.

Authorization, especially the OAuth 2 system based on RFC 6749, was one of the various security protocols that was required to get implemented on this framework. Because of the model we sought to accomplish for the framework, which comprised people and their devices, cloud-based architecture with secure connections and excellent speed, the implementation was not straightforward. What made this project unique is that it required us to provide authorization protection to a framework running on a private 5G cellular network. This implies the network functions differently from WiFi or other wireless standards because there are no wireless channels existing rather than a deterministic system and licensed bands, which are exploited for the 5G communication.

The structure of the thesis was kept relatively simple, providing only the needed information without a lot of extra details. After the introduction chapter which was needed in order to make the reader familiar with the thesis' main subject, Chapter 2 aimed on presenting all the mandatory literature based on the main topic. There was information regarding MEC logic and its challenges, IoT infrastructure along with its security evolution and then got more specific by focusing on the authorization security protocol. With basic authorization details and its first use cases being already discussed, what followed was a thorough description of the OAuth 2 standard which is, with no doubt, the ideal authorization technique and got implemented in the chapters that followed.

Chapter 3 consisted of more technical details regarding the OAuth 2 implementation and integration on the existing framework that was also described in the same chapter. Going through the different components was very important because we managed to understand how everything depends on the rest of the framework nodes and function as a rigid system, but at the same time separated enough so as to accomplish the desired security levels and confidentiality. That chapter was summed up with some authorization integration ideas and features that got implemented while undertaking project.

Lastly, specific performance metrics were showcased in Chapter 4, after sharing a lot of information about the Bristol University's testbed that was used and the device emulator that needed to be implemented for the sake of testing the framework. The experiments that took place for this thesis, except for measuring the overall framework performance, managed to provide information about the various influencing parameters and indicate ways to improve the performance with features like TLS sessions and in-memory databases (caching). In the following subsection there will be a complete analysis of the results along with some thoughts about the final performance.

5.1 Results Analysis

First and foremost, as regards to authorization, I managed to implement my own custom instance of it with the use of ORY Hydra, as mentioned in Chapter 3. This means that the OAuth 2 flows were slightly altered in order to match the framework's requirements and operate as needed. After a lot of revisions, we managed to add the authorization process in the framework with almost no additional overhead, as described in Subsection 4.4.1. As compared to the performance without executing any authorization action, the final achieved RTTs are not burdened at all with additional overhead. This was accomplished with the combination of TLS session resumption and Redis caching which utilizes in-memory databases thus resulting in less database communications and improved performance. We can safely say that with the right configuration and library integration, the authorization security protocol can be applied on a system with almost zero added overhead to the overall performance.

Secondly, we investigated what happens in the scenario of access token expiration. There was a unique experiment conducted for that matter, which allowed us to visualize the performance alteration while the refreshing of an access token takes place. What was understood in that subsection was that it is preferred, at least for this particular framework, to implement ways of device stalling rather than packet loss. This way the systems benefits from not filling the data collection endpoint with false or duplicated data. The additional overhead of the token refresh operation is persistent, which results in much better reliability levels and confident performance predictability. Based on the results, the average time that a device needs to re-transmit a message after getting a new access token is 0,8 seconds, which is four times larger than a normal transmission RTT but low enough for this kind of application. After the single re-transmission, the device keeps on communicating normally with the lowest possible RTT for each outgoing message.

Thirdly, the impact of Redis caching and TLS session resumption needed to be measured in order to understand their importance and display how they facilitate the whole system. With inspection of Figure 4.11 we were able to have a taste of how much the session resumption improves the RTTs just by maintaining the same connection between the device and the middleware API and by avoiding the execution of unnecessary TLS handshakes for every single outgoing message of every single device. This integration resulted in much more responsive outer communication between the framework and the devices, limiting the delay between their exchanged packets. In order to achieve better inner performance, we had to make use of in-memory data structures. Redis integration provided just what we needed with results that not only made our framework more light weight but also enhanced the overall performance by limiting the communications towards the database endpoint. This was achieved due to local memory information storing, taking advantage of RAM's exceptionally fast read/write times. Redis' impact was displayed in Figure 4.12, showcasing the importance of such library on a project like this.

Lastly, we compared the two different API configurations that were used for the four of the experiments that took place. Although this parameter does not represent a security protocol, it was very interesting to investigate how the different resources impact the framework's performance. We noticed that the framework achieved much better RTTs when running separately from the device emulator rather than running on the same machine and taking advantage of VLAN. This happened because of the testbed's interoperability feature, allowing it to make use of more computing resources whenever possible. Having said that, the framework scaled better when running separately from the device emulator and with the use

of the 1Gbit 5G network adapter the overall performance was much higher as displayed in Figure 4.13, motivating us to make plans for the industrial usage and testing of the project once everything is ready and fail-safe.

5.2 Summary of Contributions

After having done so much research on security protocols and especially on authorization, I should list some of the main contributions that were conducted for the needs of this thesis and project development.

- **Understanding authorization and authentication difference**

Studying authorization made me realize the distinct difference between authorization and authentication. Although it is quite obvious to me now, I believe that anyone working on authorization should be informed about the differences between those two concepts and each one's responsibilities.

- **Implementing a custom OAuth 2 instance**

The particular framework I worked on required a unique configuration of the OAuth 2 authorization mechanism. This made me spend a lot of time with different types of setups until I reached the desired performance and functionality. I discovered new integration approaches and suggested improvements on the initial design of the authorization endpoint so as to accomplish even better performance results.

- **Combining multiple security protocols**

In order to mitigate various vulnerabilities such as those mentioned in Subsection 2.5, I had to combine the authorization protocol with SSL certificates so as to utilize HTTPS connections between the different authorization actors. Additionally, I expanded the user dashboard's security measures but integrating the authorization mechanism in it, allowing it to be accessed through API requests with the use of an access token.

- **Conducting experiments for performance evaluation**

For the needs of this thesis and in order to evaluate the performance of the framework, I learned how to execute experiments and collect metrics. Using these metrics allowed me and my team to investigate system flaws and plan improvements so as to attain a better overall performance. Moreover, we used plotting libraries in order to visualize the final results and have them displayed in Chapter 4.

- **Applying improvements on an existing framework**

Having the experiments at hand allowed us to configure the framework multiple times either on its existing technologies or by adding a few more libraries like Redis and TLS session resumption. After these additions, we conducted the experiments again in order to calculate each library's impact individually, as presented in Section 4.4.

- **Exploiting 5G private networking**

Working at the University of Bristol offered me the chance to have a taste of the structure and functionality of a private 5G network. There are a lot of differences compared to a WiFi network which favors the 5G cellular communications but also makes the whole implementation procedure more challenging and interesting.

- **Verifying literature findings**

Another very interesting contribution was to finally verify and see in action all the different findings that I had collected from the study I made as presented in Chapter 2. I kept going back to this information and I was surprised enough to use a lot of the literature's guiding for the sake of the authorization implementation that I was assigned to conduct.

5.3 Future Work - Recommendations

As the framework had a lot of improvements and revisions until it reached its current state, there is still room for improvement for someone that has the time and effort to give it a chance. In terms of visual representation, as I mentioned in Subsection 3.1.2, the user dashboard can definitely be improved and become more practical. Its current state it is good enough to display the user's information for debugging purposes, but if this project was to be consumed by real world users, a visual overhaul ought to be performed for easier access and better usability. The whole functionality is there, but with some better front-end technologies utilized, the dashboard would be instantly upgraded and polished.

As for the custom SSL certificates, they may be working and achieve what we want them to by providing HTTPS connections and SSL protection, but they require additional configuration in order to make use of them on each device separately. With the use of a domain name and an SSL certificate from an official trusted certificate authority, the TLS protocol would be exploited much easier as there would be no need to install the rootCA on every device that needed to make use of HTTPS communications. This was not a priority improvement for us as the project was built for research purposes and we could automate the certificate authority installation on all of our devices through the device emulator, but in a real world scenario, this should definitely be a prioritized upgrade.

In terms of authorization, one improvement that could make the system even more secure is to use different tokens for each device and even provide multiple tokens to single devices, each one with different use case. As we saw in Chapter 2, there is a specific configuration where one could control where an access token is used and what resource it is allowed to access. In our framework's OAuth 2 implementation I did not make use of this feature. Not only that, but only one token is issued for each user. In a future revision, I would recommend someone to issue different tokens for each device, and limit each token's usage in order to attain even better authorization security integrity.

Another obvious improvement would also be upgrading from Docker engine to a Kubernetes cluster. The system's overall performance could be vastly increased because of Kubernetes running on different nodes. This upgrade would be efficient if the framework was to be publicly used, as it could scale better, withstand the higher traffic that would potentially exist and provide improved availability. If one decides to do this particular upgrade, additional network configuration will be required because of the different architecture between Docker and Kubernetes. Docker operates on a local network on a single

machine whereas Kubernetes makes use of multiple nodes in a cluster, thus requiring more advanced load balancing and network distribution.

BIBLIOGRAPHY

- [1] Q. Jing, A. V. Vasilakos, J. Wan, J. Lu, and D. Qiu, "Security of the Internet of Things: perspectives and challenges," *Wireless Networks*, vol. 20, pp. 2481–2501, 2014.
- [2] H. Shahinzadeh, J. Moradi, G. B. Gharehpetian, H. Nafisi, and M. Abedi, "Iot architecture for smart grids," in *2019 International Conference on Protection and Automation of Power System (IPAPS)*, pp. 22–30, IEEE, 2019.
- [3] S. Muthuramalingam, A. Bharathi, N. Gayathri, R. Sathiyaraj, B. Balamurugan, *et al.*, "Iot based intelligent transportation system (iot-its) for global perspective: A case study," in *Internet of Things and Big Data Analytics for Smart Generation*, pp. 279–300, Springer, 2019.
- [4] R. K. Kodali, V. Jain, S. Bose, and L. Boppana, "Iot based smart security and home automation system," in *2016 international conference on computing, communication and automation (ICCCA)*, pp. 1286–1289, IEEE, 2016.
- [5] A. Gupta, R. Christie, and P. Manjula, "Scalability in internet of things: features, techniques and research challenges," *Int. J. Comput. Intell. Res.*, vol. 13, no. 7, pp. 1617–1627, 2017.
- [6] G. Fortino, C. Savaglio, C. E. Palau, J. S. de Puga, M. Ganzha, M. Paprzycki, M. Montesinos, A. Liotta, and M. Llop, "Towards multi-layer interoperability of heterogeneous iot platforms: The inter-iot approach," in *Integration, interconnection, and interoperability of IoT systems*, pp. 199–232, Springer, 2018.
- [7] D. Kozlov, J. Veijalainen, and Y. Ali, "Security and privacy threats in iot architectures.," in *BO-DYNETS*, pp. 256–262, 2012.
- [8] D. Hardt, "The OAuth 2.0 Authorization Framework." RFC 6749, Oct. 2012.
- [9] V. Beltran and A. F. Skarmeta, "An overview on delegated authorization for CoAP: Authentication and authorization for Constrained Environments (ace)," in *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*, pp. 706–710, 2016.
- [10] H. Tschofenig, "Fixing User Authentication for the Internet of Things (IoT)," *Datenschutz und Datensicherheit*, vol. 40, pp. 222–224, 2016.
- [11] A. K. Yerrapragada, T. Eisman, and B. Kelley, "Physical layer security for beyond 5g: Ultra secure low latency communications," *IEEE Open Journal of the Communications Society*, pp. 1–1, 2021.
- [12] F. Pan, H. Wen, H. Song, T. Jie, and L. Wang, "5g security architecture and light weight security authentication," in *2015 IEEE/CIC International Conference on Communications in China - Workshops (CIC/ICCC)*, pp. 94–98, 2015.
- [13] G. de Koning Gans, J.-H. Hoepman, and F. D. Garcia, "A practical attack on the mifare classic," in *International Conference on Smart Card Research and Advanced Applications*, pp. 267–282, Springer, 2008.

- [14] V. Khatod and A. Manolova, "Effects of man in the middle (mitm) attack on bit error rate of blue-tooth system," in *2020 Joint International Conference on Digital Arts, Media and Technology with ECTI Northern Section Conference on Electrical, Electronics, Computer and Telecommunications Engineering (ECTI DAMT NCON)*, pp. 153–157, 2020.
- [15] K. S. Manosha, M. Matinmikko-Blue, and M. Latva-aho, "Framework for spectrum authorization elements and its application to 5g micro-operators," in *2017 Internet of Things Business Models, Users, and Networks*, pp. 1–8, 2017.
- [16] C.-Y. Li, Y.-D. Lin, Y.-C. Lai, H.-T. Chien, Y.-S. Huang, P.-H. Huang, and H.-Y. Liu, "Transparent aaa security design for low-latency mec-integrated cellular networks," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 3, pp. 3231–3243, 2020.
- [17] A. Rostami, "Private 5g networks for vertical industries: Deployment and operation models," in *2019 IEEE 2nd 5G World Forum (5GWF)*, pp. 433–439, 2019.
- [18] J. Richer, "OAuth 2.0 Token Introspection." RFC 7662, Oct. 2015.
- [19] B.-C. Chifor, I. Bica, V.-V. Patriciu, and F. Pop, "A security authorization scheme for smart home internet of things devices," *Future Generation Computer Systems*, vol. 86, pp. 740–749, 2018.
- [20] Y. Harbi, Z. Aliouat, A. Refoufi, and S. Harous, "Recent security trends in internet of things: A comprehensive survey," *IEEE Access*, vol. 9, pp. 113292–113314, 2021.
- [21] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, 2013. Including Special sections: Cyber-enabled Distributed Computing for Ubiquitous Cloud and Network Services Cloud Computing and Scientific Applications — Big Data, Scalable Analytics, and Beyond.
- [22] S. Hammoudi, Z. Aliouat, and S. Harous, "Challenges and research directions for internet of things," *Telecommunication Systems*, vol. 67, no. 2, pp. 367–385, 2018.
- [23] S. Cirani, M. Picone, P. Gonizzi, L. Veltri, and G. Ferrari, "Iot-oas: An oauth-based authorization service architecture for secure services in iot scenarios," *IEEE Sensors Journal*, vol. 15, no. 2, pp. 1224–1234, 2015.
- [24] A. Arora, A. Kaur, B. Bhushan, and H. Saini, "Security concerns and future trends of internet of things," in *2019 2nd International Conference on Intelligent Computing, Instrumentation and Control Technologies (ICICT)*, vol. 1, pp. 891–896, 2019.
- [25] R. Román-Castro, J. López, and S. Gritzalis, "Evolution and trends in iot security," *Computer*, vol. 51, no. 7, pp. 16–25, 2018.
- [26] Z.-K. Zhang, M. C. Y. Cho, C.-W. Wang, C.-W. Hsu, C.-K. Chen, and S. Shieh, "Iot security: ongoing challenges and research opportunities," in *2014 IEEE 7th international conference on service-oriented computing and applications*, pp. 230–234, IEEE, 2014.

- [27] R. Roman, J. Zhou, and J. Lopez, “On the features and challenges of security and privacy in distributed internet of things,” *Computer Networks*, vol. 57, no. 10, pp. 2266–2279, 2013. Towards a Science of Cyber Security Security and Identity Architecture for the Future Internet.
- [28] J. Manyika, M. Chui, P. Bisson, J. Woetzel, R. Dobbs, J. Bughin, and D. Aharon, “Unlocking the potential of the internet of things,” *McKinsey Global Institute*, vol. 1, 2015.
- [29] I. Ahmad, T. Kumar, M. Liyanage, J. Okwuibe, M. Ylianttila, and A. Gurtov, “Overview of 5g security challenges and solutions,” *IEEE Communications Standards Magazine*, vol. 2, no. 1, pp. 36–43, 2018.
- [30] H. Kim and E. A. Lee, “Authentication and authorization for the internet of things,” *IT Professional*, vol. 19, no. 5, pp. 27–33, 2017.
- [31] M. Liyanage, I. Ahmad, A. B. Abro, A. Gurtov, and M. Ylianttila, *A comprehensive guide to 5G security*. John Wiley & Sons, 2018.
- [32] Z. Kotulski, W. Niewolski, T. W. Nowak, and M. Sepczuk, “New security architecture of access control in 5g mec,” in *Security in Computing and Communications* (S. M. Thampi, G. Wang, D. B. Rawat, R. Ko, and C.-I. Fan, eds.), (Singapore), pp. 77–91, Springer Singapore, 2021.
- [33] R. Masood, M. A. Shibli, Y. Ghazi, A. Kanwal, and A. Ali, “Cloud authorization: exploring techniques and approach towards effective access control framework,” *Frontiers of Computer Science*, vol. 9, no. 2, pp. 297–321, 2015.
- [34] A. Annunziato, “5g vision: Ngmn-5g initiative,” in *2015 IEEE 81st Vehicular Technology Conference (VTC Spring)*, pp. 1–5, IEEE, 2015.
- [35] L. Xiao, X. Wan, C. Dai, X. Du, X. Chen, and M. Guizani, “Security in mobile edge caching with reinforcement learning,” *IEEE Wireless Communications*, vol. 25, no. 3, pp. 116–122, 2018.
- [36] J.-H. Lee and H. Kim, “Security and privacy challenges in the internet of things [security and privacy matters],” *IEEE Consumer Electronics Magazine*, vol. 6, no. 3, pp. 134–136, 2017.
- [37] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, “A survey on mobile edge computing: The communication perspective,” *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.
- [38] P. Ranaweera, A. D. Jurcut, and M. Liyanage, “Realizing multi-access edge computing feasibility: Security perspective,” in *2019 IEEE Conference on Standards for Communications and Networking (CSCN)*, pp. 1–7, 2019.
- [39] R. Roman, J. Lopez, and M. Mambo, “Mobile edge computing, fog et al.: A survey and analysis of security threats and challenges,” *Future Generation Computer Systems*, vol. 78, pp. 680–698, 2018.
- [40] D. Fett, R. Küsters, and G. Schmitz, “A comprehensive formal security analysis of oauth 2.0,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS ’16*, (New York, NY, USA), p. 1204–1215, Association for Computing Machinery, 2016.
- [41] R. Boyd, *Getting started with OAuth 2.0*. ” O’Reilly Media, Inc.”, 2012.

- [42] M. Darwish and A. Ouda, "Evaluation of an oauth 2.0 protocol implementation for web server applications," in *2015 International Conference and Workshop on Computing and Communication (IEMCON)*, pp. 1–4, 2015.
- [43] O. Boehm, J. Caumanns, M. Franke, and O. Pfaff, "Federated authentication and authorization: A case study," in *2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pp. 356–362, IEEE, 2008.
- [44] D. Preuveneers and W. Joosen, "Access control with delegated authorization policy evaluation for data-driven microservice workflows," *Future Internet*, vol. 9, no. 4, p. 58, 2017.
- [45] Q.-h. Bai and Y. Zheng, "Study on the access control model," in *Proceedings of 2011 Cross Strait Quad-Regional Radio Science and Wireless Technology Conference*, vol. 1, pp. 830–834, 2011.
- [46] D. Ferraiolo, J. Cugini, D. R. Kuhn, *et al.*, "Role-based access control (rbac): Features and motivations," in *Proceedings of 11th annual computer security application conference*, pp. 241–48, 1995.
- [47] L. Zhi, W. Jing, C. Xiao-su, and J. Lian-xing, "Research on policy-based access control model," in *2009 International Conference on Networks Security, Wireless Communications and Trusted Computing*, vol. 2, pp. 164–167, 2009.
- [48] V. C. Hu, D. R. Kuhn, D. F. Ferraiolo, and J. Voas, "Attribute-based access control," *Computer*, vol. 48, no. 2, pp. 85–88, 2015.
- [49] E. Hammer-Lahav, "The OAuth 1.0 Protocol." RFC 5849, Apr. 2010.
- [50] S. Emerson, Y.-K. Choi, D.-Y. Hwang, K.-S. Kim, and K.-H. Kim, "An oauth based authentication mechanism for iot networks," in *2015 International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 1072–1074, 2015.
- [51] S. Sciancalepore, G. Piro, D. Caldarola, G. Boggia, and G. Bianchi, "Oauth-iot: An access control framework for the internet of things based on open standards," in *2017 IEEE Symposium on Computers and Communications (ISCC)*, pp. 676–681, 2017.
- [52] G. Brail and S. Ramji, "Oauth-the big picture," 2012.
- [53] M. Jones and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage." RFC 6750, Oct. 2012.
- [54] T. Lodderstedt, S. Dronia, and M. Scurtescu, "OAuth 2.0 Token Revocation." RFC 7009, Aug. 2013.
- [55] B. Leiba, "Oauth web authorization protocol," *IEEE Internet Computing*, vol. 16, no. 1, pp. 74–77, 2012.
- [56] W. Denniss, J. Bradley, M. Jones, and H. Tschofenig, "OAuth 2.0 Device Authorization Grant." RFC 8628, Aug. 2019.
- [57] C.-J. Chae, K.-B. Kim, and H.-J. Cho, "A study on secure user authentication and authorization in oauth protocol," *Cluster Computing*, vol. 22, no. 1, pp. 1991–1999, 2019.

- [58] E. Shernan, H. Carter, D. Tian, P. Traynor, and K. Butler, “More guidelines than rules: CsrF vulnerabilities from noncompliant oAuth 2.0 implementations,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 239–260, Springer, 2015.
- [59] P. Yadav and C. D. Parekh, “A report on csrf security challenges & prevention techniques,” in *2017 international conference on innovations in information, embedded and communication systems (ICIIECS)*, pp. 1–4, IEEE, 2017.
- [60] A. Czeskis, A. Moshchuk, T. Kohno, and H. J. Wang, “Lightweight server support for browser-based csrf protection,” in *Proceedings of the 22nd international conference on World Wide Web*, pp. 273–284, 2013.
- [61] P. Siriwardena, *OAuth 2.0*, pp. 91–132. Berkeley, CA: Apress, 2014.
- [62] K. Hickman and T. Elgamal, “The ssl protocol,” 1995.
- [63] M. View, J. Rydell, M. Pei, and S. Machani, “TOTP: Time-Based One-Time Password Algorithm.” RFC 6238, May 2011.
- [64] M. View, D. M’Raihi, F. Hoornaert, D. Naccache, M. Bellare, and O. Ranen, “HOTP: An HMAC-Based One-Time Password Algorithm.” RFC 4226, Dec. 2005.
- [65] Y. Chu, J. M. Kim, Y. Lee, S. Shim, and J. Huh, “Ss-dpki: Self-signed certificate based decentralized public key infrastructure for secure communication,” in *2020 IEEE International Conference on Consumer Electronics (ICCE)*, pp. 1–6, IEEE, 2020.
- [66] “Docker’s official documentation.” <https://docs.docker.com/get-started/overview>.
- [67] X. Chi, B. Liu, Q. Niu, and Q. Wu, “Web load balance and cache optimization design based nginx under high-concurrency environment,” in *2012 Third International Conference on Digital Manufacturing Automation*, pp. 1029–1032, 2012.
- [68] “NGINX Authentication based on Subrequest.” <https://docs.nginx.com/nginx/admin-guide/security-controls/configuring-subrequest-authentication/>.
- [69] A. Mavromatis and D. Simeonidou, “Experiences from building a multi-access edge computing internet of things testbed,” in *2020 European Conference on Networks and Communications (EuCNC)*, pp. 259–264, 2020.
- [70] B. Dzogovic, B. Santos, V. T. Do, B. Feng, N. Jacot, and T. Van Do, “connecting remote enodeb with containerized 5g c-rans in openstack cloud,” in *2019 6th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/ 2019 5th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)*.
- [71] P. Morrissey, N. P. Smart, and B. Warinschi, “A modular security analysis of the tls handshake protocol,” in *Advances in Cryptology - ASIACRYPT 2008* (J. Pieprzyk, ed.), (Berlin, Heidelberg), pp. 55–73, Springer Berlin Heidelberg, 2008.
- [72] E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.3.” RFC 8446, Aug. 2018.

- [73] N. Aviram, K. Gellert, and T. Jager, “Session resumption protocols and efficient forward security for tls 1.3 0-rtt,” *Journal of Cryptology*, vol. 34, no. 3, pp. 1–57, 2021.
- [74] S. Cirani and M. Picone, “Effective authorization for the web of things,” in *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pp. 316–320, 2015.
- [75] V. A. Siris, D. Dimopoulos, N. Fotiou, S. Voulgaris, and G. C. Polyzos, “Oauth 2.0 meets blockchain for authorization in constrained iot environments,” in *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*, pp. 364–367, 2019.
- [76] S.-R. Oh and Y.-G. Kim, “Development of iot security component for interoperability,” in *2017 13th International Computer Engineering Conference (ICENCO)*, pp. 41–44, 2017.
- [77] B. Ali, M. A. Gregory, and S. Li, “Multi-access edge computing architecture, data security and privacy: A review,” *IEEE Access*, vol. 9, pp. 18706–18721, 2021.

ANNEX: Device Emulator Code

A.1 Supervisor

A.1.1 Object Structure

```
class Supervisor(object):
    def __init__(self, nusers, ndevices, url, device_name, delay, minutes_duration,
        ↪  tls):
        self._nusers = nusers
        self._ndevices = ndevices
        self._device_kwargs = {
            'url': url,
            'access_token': None,
            'device_name': device_name,
            'supervisor': self,
            'delay': delay,
            'tls': tls }
        self._is_running = False
        self._setup_complete= False
        self._devices = []
        self.minutes_duration = minutes_duration

    @property
    def is_running(self):
        return self._is_running

    @property
    def is_setup_complete(self):
        return self._setup_complete

    def _stop(self):
        self._is_running = False
        [d.join() for d in self._devices]
```

A.1.2 Create Devices

```
def _create_devices(self):
    # Parse registered users' information
    with open('config/registered.json', 'r') as infile:
        userCredentials = json.load(infile)
    device_no = 0
    for user_identity in userCredentials:
        user_id = user_identity["user_id"]
        self._device_kwargs["access_token"] = user_identity["user_token"]
        for i in range(self._ndevices):
            # Create Device object, passing it the user's access token
            device = Device(thread_index = device_no, **self._device_kwargs, user_id =
                ↪  user_id)
            device_no += 1
```

```

        device.setDaemon(True)
        device.start()
        self._devices.append(device)
self._setup_complete = True
self._setup_time = time.time()

```

A.1.3 Main Function

```

def start(self):
    self._is_running = True
    try:
        # Retrieve the custom SSL Certificate
        urllib.request.urlretrieve("http://{}/download".format(NGINX_URL), "cert.pem")
        # Create the device spawner thread
        spawner = threading.Thread(target = self._create_devices, name='Spawner')
        spawner.setDaemon(True)
        spawner.start()
        while True:
            alive_devices = len([d for d in self._devices if d.is_alive()])
            if not alive_devices:
                break
            if self._setup_time:
                # Operation duration passed, stop the emulator.
                if time.time() - self._setup_time > self.minutes_duration * 60:
                    break
    except Exception or KeyboardInterrupt:
        self._stop()

```

A.2 Device

A.2.1 Object Structure

```

class Device(threading.Thread):
    def __init__(self, supervisor, url, device_name, thread_index, access_token =
↳ None, delay = 1., tls = True):
        threading.Thread.__init__(self, name='Device_%s' % thread_index)
        # Configuration Parameters
        self._supervisor = supervisor
        self._url = url
        self._device_name = device_name + str(thread_index)
        self._access_token = access_token
        self._delay = delay
        self._downtime_start = None
        self._registered = False
        self._tls = tls

```

A.2.2 Send Message

```
def _send_data(self, req_session:Session):
    if not self._registered:
        body = {"id": self._device_name,
                "type": "Sensor Type",
                "temp": {"type": "Value Type",
                        "value": "Value"}}
        req_endpoint = "/v2/entities"
        req_method = "POST"
    else:
        body = {"id": self._device_name,
                "temp": {"value": "Value"}}
        req_endpoint = "/v2/entities/{}/attrs".format(self._device_name)
        req_method = "PATCH"

    headers = {'Authorization': 'Bearer {}'.format(self._access_token)}
    try:
        if self._tls:
            response = req_session(req_method, self._url, req_endpoint, body, headers)
        else:
            response = request(req_method, self._url, req_endpoint, body, headers,
                               ↪ verify="cert.pem")
        if response.status_code in [200, 201, 204, 422]:
            # Message sent successfully / Device already registered
            self._registered = True
        elif (response.status_code == 403):
            # Token invalid or expired
            if self._tls:
                response = req_session.POST(self._url, "/getNewToken", body, headers)
            else:
                response = requests.POST(self._url, "/getNewToken", body, headers,
                                         ↪ verify="cert.pem")
            if (response.status_code == 200):
                # Replace current token with new one
                old_access_token = self._access_token
                self._access_token = response.text
                headers = {'Authorization': 'Bearer {}'.format(self._access_token)}
                if self._tls:
                    response = req_session(req_method,self._url, req_endpoint, body,
                                         ↪ headers)
                else:
                    response = request(req_method, self._url, req_endpoint, body,
                                         ↪ headers)
            # Packet Loss Cases
        else:
            pass
    else:
        pass
```

A.2.3 Main Function

```
def run(self):
    req_session = Session()
    req_session.verify = 'cert.pem'
    while not self._supervisor.is_setup_complete:
        time.sleep(self._delay)
    try:
        while self._supervisor.is_running:
            self._send_data(req_session)
            time.sleep(self._delay)

    except(Exception):
        logger.info(Exception)
```

ANNEX: Database Samples

B.1 ORY Kratos

B.1.1 Identity Schema

```
{
  "title": "Person",
  "type": "object",
  "properties": {
    "traits": {
      "type": "object",
      "properties": {
        "email": {
          "type": "string",
          "format": "email",
          "title": "E-Mail",
          "minLength": 3,
          "ory.sh/kratos": {
            "credentials": {
              "password": {"identifier": true} },
              "verification": {"via": "email"},
              "recovery": {"via": "email"} }},
        "name": {
          "type": "object",
          "properties": {
            "first": {
              "title": "First Name",
              "type": "string"},
            "last": {
              "title": "Last Name",
              "type": "string" }},
            "username": {
              "type": "string",
              "ory.sh/kratos": {
                "credentials": {
                  "password": {"identifier": true} }}}} },
    "required": ["email"],
    "additionalProperties": false }}
}
```

B.2 OCB

B.2.1 IoT Device Stored Data

```
{
  "_id": {
    "id": "iot_device_1",
    "type": "temperatureSensor",
    "servicePath": "/sector_13"},
  "attrNames": ["temp"],
}
```

```

"attrs": {
  "temp": {
    "value": 11,
    "type": "Number",
    "mdNames": [],
    "creDate": 1629796447.5112798,
    "modDate": 1629796473.010117} },
"creDate": 1625672525.437704,
"modDate": 1625672525.437704,
"lastCorrelator": "e180b000-df39-11eb-9cdc-0242ac18010f" }

```

B.3 Framework's MongoDB

B.3.1 User Token Info

```

{"id": "8902e273-c5dc-4d22-8827-bbedc7aad4a8",
"token": [{
  "access_token": "f0_F9XoLqrHclrM9kaFxCBv2ghDYk7hhDbORSS2yoec...",
  "expires_in": 3600,
  "refresh_token": "QQZQlJd_pnLU0_hS0pr8QmpWhw17q00fDiikSMkNPR8...",
  "scope": "offline_access",
  "token_type": "bearer",
  "expires_at": {"date": "2021-08-05T15:12:10.849Z"} }]}

```

B.3.2 User-Device-Token Association

```

{"lastToken": "f0_F9XoLqrHclrM9kaFxCBv2ghDYk7hhDbORSS2yoec...",
"deviceID": "iot-device-1",
"userID": "8902e273-c5dc-4d22-8827-bbedc7aad4a8" }

```