BUCHAREST UNIVERSITY OF ECONOMIC STUDIES

Cybernetics, Statistics and Economics Informatics Faculty

Department of Economics Informatics

DISSERTATION

Scientific Coordinator

Assoc. Prof. Cristian-Valeriu TOMA, Ph.D.

Graduate

Andrei-Robert CAZACU

Bucharest

2022

BUCHAREST UNIVERSITY OF ECONOMIC STUDIES

Cybernetics, Statistics and Economics Informatics Faculty

Department of Economics Informatics

Secure IoT solution for office building monitoring

DISSERTATION

Scientific Coordinator

Assoc. Prof. Cristian-Valeriu TOMA, Ph.D.

Graduate

Andrei-Robert CAZACU

Bucharest

2022

TABLE OF CONTENTS

1. INTRODUCTION

Despite the fact that IoT has been in the spotlight for several years, Gartner still predicts a five time increase in the number of devices from 2018 to 2028 [1], up to 1.9 billion units, which further intensifies the need for security among IoT nodes. Through the years, a significant amount of effort has been spent standardizing and improving inter-operability among devices, spawning several lightweight application layer protocols such as CoAP [2], standard formats for data during transit such as IPSO objects [3], or means to encode data that further reduces the size of the payload such as CBOR [4]. While Internet of Things wasn't officially a concept until 1999, one of the first examples of such a device surfaced in the early 1980s, an Internet connected Coca Cola vending machine placed at the Carnegie Mellon University which local programmers modified to report whether a drink was available, and if it was cold or not before making the trip [5]. The phrase "Internet of Things" was coined by Kevin Ashton, then Executive Director of Auto-ID Labs, during a presentation he made for Procter & Gamble.

Recently, great efforts have been poured into securing IoT devices, empowered by the rapid advancements in technology which allows complex cryptographic operations to be executed on device in a reasonable amount of time. While solutions based on existing cryptographic algorithms have recently surfaced, such as Connectivity Standards Alliance's Matter [6], which relies on the Public Key Infrastructure model, new lightweight cryptographic algorithms are being developed in an attempt to make security a thing for even the smallest of devices. This pursuit is fuelled by NIST which initiated a process to solicit, evaluate, and standardize lightweight cryptographic algorithms in August 2018 [7].

The purpose of this paper is to combine existing and emerging technologies into a polished IoT product that can be used for office building monitoring, but still be expandable to suit other needs such as smart cities, or smart homes. This modularity is baked into the architecture of the product, making no assumption about the type of the data sent or received, nor about the type of sensors attached to the node. Another imposed constraint was developing a security model that would be able to ensure the confidentiality of the payload without requiring a gateway, leading to greater computational effort that must be placed upon the IoT node, with the benefit of being able to push the data directly into the cloud and having almost zero configuration needed. While this architecture may not confine to previously agreed standards of having several nodes connected via Intranet to a gateway which then pushed data into a public or private cloud, big names in the cloud computing industry such as Amazon Web Services, Microsoft Azure and Oracle have started shipping solutions that cater to such kind of cloud connected nodes. Solutions have also appeared in the hobbyist market, such as Arduino Cloud, which shows the industry wide trend of IoT edge computing and edge AI.

For clarity, this dissertation has been structured into four chapters:

1. Introduction
2. Technical and Mathematical Concepts, where fundamental concepts of this dissertation are discussed
3. Proposed solution, where the architecture and the implementation are discussed
4. Conclusion, a chapter which discusses the pros and cons of the designed solution

2. TECHNICAL AND MATHEMATICAL CONCEPTS

2. 1. What is Internet of Things?

Internet of Things as a concept represents a system of interconnected devices that are able to transfer data over any medium without requiring human intervention. These devices can take different shapes and form, and their processing power vary widely, ranging from a Zigbee Smart Bulb to a self-driving Car featuring a plethora of sensors and significant processing power.

Commonly, a three-layered architecture is used when designing a IoT solution featuring a perception layer, a network layer, and an application layer.

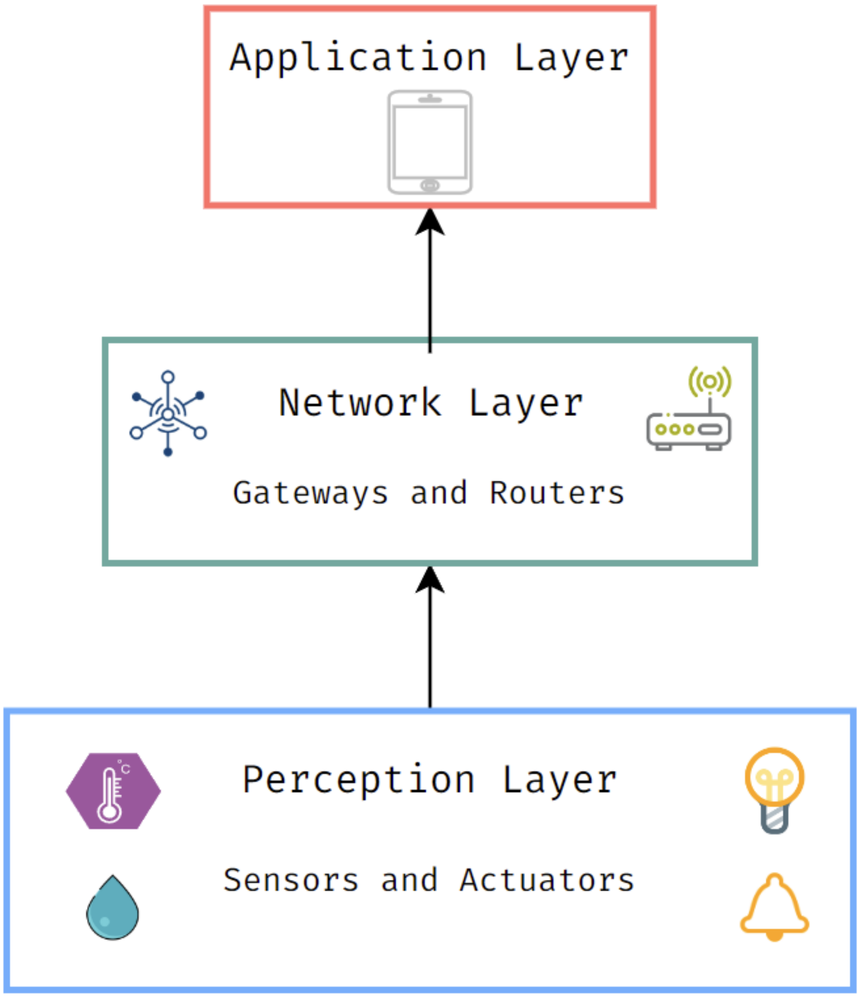


Figure 2-1. IoT architecture

The perception layer represents the physical layer of the solution which contains sensors and actuators. These devices communicate with the external world and often support advanced features such as self-identification or self-diagnosis. At this level, we can find devices such as IoT nodes that represent the building blocks of every solution which can either connect directly to the cloud or push the data to a IoT gateway or another edge device for processing. Here, different technologies might pop up ranging from development boards to different types of sensors.

The next layer of the architecture is represented by the network layer, which has the task of transporting the data provided by the perception layer to the application layer. This should not be confused with the ISO/OSI networking model, which equates the network as a Layer 3, since the IoT equivalent encompasses multiple ISO/OSI layers. At this layer, devices such as the IoT gateway and the network router are placed. An IoT gateway has the role of collecting the data from multiple IoT nodes, process it and sent it further down the chain either for processing or for display. This is also known as fog computing, where multiple low powered nodes are used at the edge of the network for collection of data and other devices are used for intermediary processing before reaching the cloud, in this case the gateway. This architecture greatly reduces power consumption at the edge, allowing cheaper and less powerful devices to be used, but also permitting the use of non-IP protocols for the nodes. The node can also be connected directly to the cloud, skipping the gateway, and allowing end-to-end encryption, which is not possible in the fog computing architecture. In this case, the network layer would be represented by the router, which connects the intranet to the outside world.

The closest layer to the user is the application layer, which stores, aggregates, filters, and processes all the data from the previous layers. Following this process, the data is made available to be used by the user, hiding the heterogeneity of the underlying system. At this level, software solutions such as databases, REST microservices or graphical user interfaces can be found.

2. 2. Cryptography and Blockchain

Cryptography is the study of secure communications techniques that allow only the sender and intended recipient of a message to view its contents [8]. While closely associated with encryption, cryptography can also be used for providing data integrity, authentication, or non-repudiation. Historically, cryptography was used to secure the exchange of messages often with the help of crypto devices. The first recorded occurrence of such a device is the *scytale*, used by the Spartans in 400 BC. It represented an easy to implement transposition cipher, achieved using P-boxes. Later, substitution ciphers, implemented by S-boxes, gained popularity with the apparition of the Caesar

Cipher during the first century BC, later being used as a source for inspiration in more complex ciphers such as Vigenère. Another notable occurrence is the Playfair cipher, used by the British Army during the First World War, or the Enigma Machine used by the German Army during the Second World War with great success. The Enigma Machine is a device with multiple rotors which implemented a complex varying polyalphabetic substitution cipher.

Modern cryptography combines both S-boxes and P-boxes to create a product cipher which is much harder to break. Commonly used cryptographic algorithms include AES block cipher which is used for symmetric encryption and RSA for asymmetric encryption or digital signatures. Modern cryptography is also multi-disciplinary, existing at the intersection of mathematics, computer science and physics. Popular applications include e-commerce, chip-based payment cards, secure communications, and digital currencies.

With the advent of modern cryptography, technologies such as blockchain arrived to the market which are now widely used in all sorts of applications ranging from digital currencies to food safety and supply chain management. It can be seen as a decentralized, distributed, peer-to-peer ledger, which is immutable and used to record information in a secure manner.

Security is assured in a blockchain by:

- Immutability: blocks are write-once append-only; any changes performed to a block will invalidate the ones appended
- Consensus: nodes within the distributed network must agree on the network's state to ensure the validity; this is done by using Proof of Work or Proof of Stake protocols
- Transparency: if a node forges a transaction, all the other nodes can pinpoint the node with the incorrect information

One example of such a blockchain is Ethereum, which encompasses several important concepts, such as nodes, blocks, transactions, and smart contracts.

A node in the Ethereum blockchain refers to a device running a client application. Such an application can verify the transactions in each block, keeping the network secure and the data accurate [9]. All such clients are developed according to the Ethereum Yellow Paper and contain the state of the network together with a transaction pool and an Ethereum Virtual Machine.

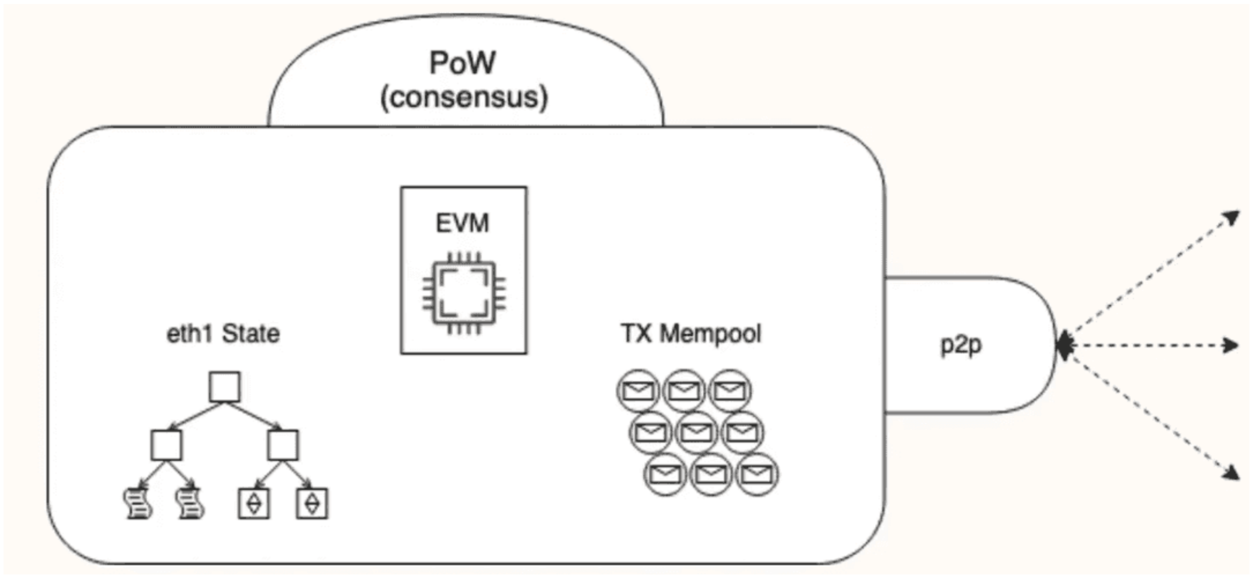


Figure 2-2. Architecture of an Ethereum Client [9]

Another important concept is the block, which is created by bundling multiple transactions together [10], structured in a binary hash tree, also known as Merkle-Patricia Tree. A new block is added only after applying the consensus algorithm, with the block and transactions within being strictly ordered (contains a reference to the parent item). In Ethereum, the Proof of Work protocol is used for consensus, mining nodes having to spend a substantial amount of energy, time, and computational power in order to obtain a "certificate of legitimacy" for each block they propose to the network. When propagating such a block, all nodes must accept the new block as the canonical next block. A block contains multiple information, among which can be found the time the block was mined, a nonce which proves that the block has gone through Proof of Work, and the transactions structured in a binary hash tree.

A transaction within a block is a cryptographically signed instruction initiated to update the state of the network [11]. Once issued, a hash value is computed which can be used the check if the transaction has been validated and altered the network state. Several types of transactions can be observed:

- Regular transactions: transaction from one wallet to another
- Contract deployment transactions: a transaction without the *to* field containing the code of the contract
- Contract execution: a transaction that interacts with a deployed smart contract

For a transaction to be mined, a gas amount must be paid, which represents the cost paid to the miners. This amount is submitted along with the transaction, being determined from fields such as *gasLimit*, *maxPriorityFeePerGas*, and *maxFeePerGas*. If specifying a higher amount of gas than required in *gasLimit* field, the unused gas will be refunded to the wallet of the originator. If a lower amount than required is used, the EVM will use all that amount while attempting to fulfil the transaction, but it will not complete, resulting in a rollback and consumption of gas.

Smart contracts are also an important concept of Ethereum blockchain, which are programs residing at specific addresses [12]. Such contracts can also define rules and enforce them programmatically, such as access control. Smart contracts are Ethereum accounts featuring a balance and internal state, users being able to interact with them by submitting transactions. The transactions must include in the *data* field a value in accordance with the application binary interface of the contract. The first 4 bytes are used to identify the function to call, using the keccak256 digest of the function signature. The rest of the call data is represented by the function arguments encoded accordingly.

2. 3. Application protocol for low-powered devices – MQTT

MQTT (Message Queuing Telemetry Transport) is an OASIS standard messaging protocol originally designed in the late 1990s to deliver data over slow satellite connection, being immune to issues that might arise from high latency and low bandwidth. It is designed as a lightweight publish/subscribe protocol, featuring a small code footprint and minimal network bandwidth requirements [13].

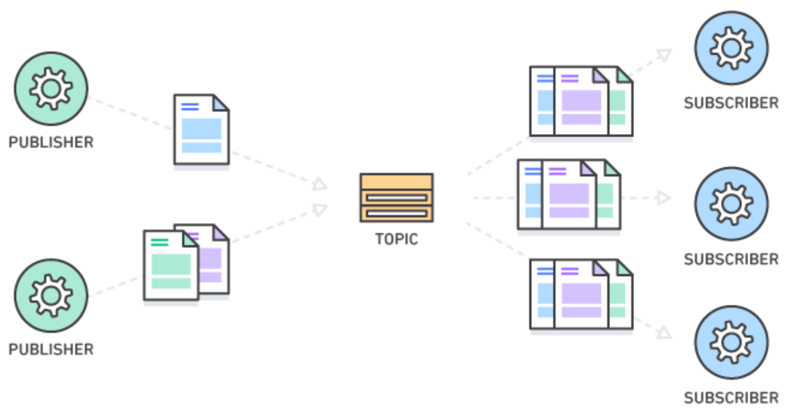


Figure 2-3. Publish-Subscribe model [14]

The publish/subscribe model defines an asynchronous method of communication, where the central point is represented by the broker. Multiple clients can connect to the broker being either publishers, subscribers, or both. Publishers send messages to the broker on certain topics, which are then distributed by the broker to the subscribed clients.

MQTT supports bi-directional communications, allowing both device-to-cloud and cloud-to-device messaging, using as underlying transport protocol TCP, or the security enabled equivalent, TLS. Authentication of clients is also supported with different options available such as username, password combination or even OAuth.

MQTT is designed to support transport over unreliable networks, supporting persistent sessions to reduce the time required to reconnect the client to the broker. It also sports a reliable message delivery mechanism using one of the 3 predefined quality of service levels:

- 0: at most once, where no confirmation receipt is used
- 1: at least once, where confirmation is used
- 2: exactly once, implying a 4-step handshake that ensures no duplicates

The MQTT Broker is tasked with managing active client connections, authentication, message delivery and message storage. It can be seen as a post office, relaying messages to the intended recipients.

A feature-rich open-source message broker is Mosquitto, offered by the Eclipse Foundation. It supports MQTT protocol versions 5.0, 3.1.1 and 3.1, and multiple transport protocols such as TCP, TLS and WebSockets. The lightweight nature of the broker enables deployment on a plethora of devices ranging from low powered single board computers to fully-fledged servers.

The MQTT client allows the connection to an MQTT broker to publish messages, and to subscribe to topics to receive messages.

2. 4. Docker and Cloud Infrastructure

Docker is an open platform for developing, shipping, and running applications [15]. It allows decoupling the developed software from the used infrastructure, allowing faster software delivery and a seamless deployment experience. Applications are packaged into images and ran in containers, which allow an isolated environment that reduces the number of issues expected from running multiple applications on the same host such as port clashing. Docker is also suited for CI/CD workflows, streamlining the development lifecycle by providing homogeneity of environments.

A client-server architecture is used, where the server is represented by the *dockerd* daemon, and the client by the *docker* application.

The Docker daemon is responsible with building, running, and distributing Docker containers, along with orchestrating other daemons in a Docker Swarm environment. It listens for incoming requests via UNIX sockets or network interface and uses a REST API.

The Docker client is used to manage Docker daemons, which can either be hosted on the local machine or on remote machines. It translates commands such as *docker run* to REST API calls, which are then carried out by the daemon.

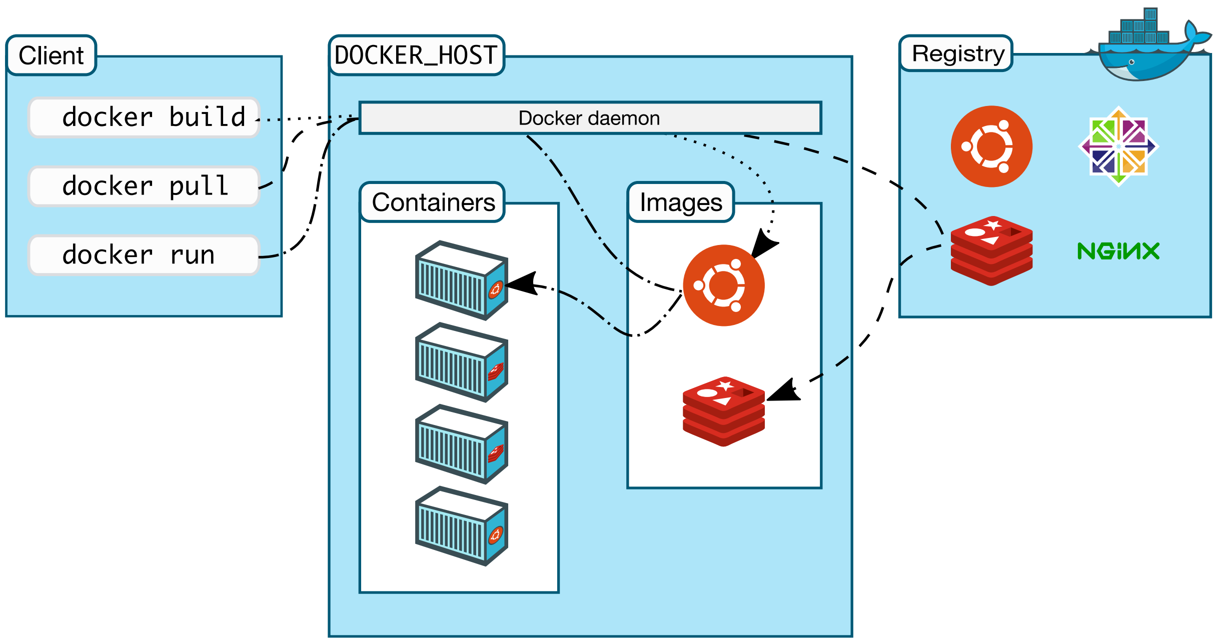


Figure 2-4. Docker architecture [15]

The Docker daemon manages multiple types of objects such as images, containers, networks, and volumes.

An image is read-only template for creating a container, which can be either pulled from a Docker Registry such as Docker Hub or built from scratch using a Dockerfile. When building an image, a layered approach is taken, where a base image is specified then instructions are issued to generate the finite result. Each new instruction equates to a layer, which is then cached to speed up the time needed to rebuild the image in case of any modifications. Building on top of existing images greatly reduces the storage requirements by allowing reutilization.

A container is a runnable instance of an image, which runs by default in isolation from other containers and its host machine. This isolation is achieved by leveraging the *namespaces* functionality of the Linux operation system. The degree of isolation can be configured by exposing ports, connecting the container to a Docker network or even the network of the host machine or mounting local directories. Multiple containers can be orchestrated using Docker Compose, allowing a declarative YAML file to be used for spawning all the required services. This greatly reduces the configuration effort needed while providing an effective way of describing the components of the solution. Also, the processing requirements are greatly reduced compared to a more traditional hypervisor-based virtual machine, providing a more efficient utilization of the available compute capacity. This is achieved by sharing the host operation system along with installed libraries where needed.

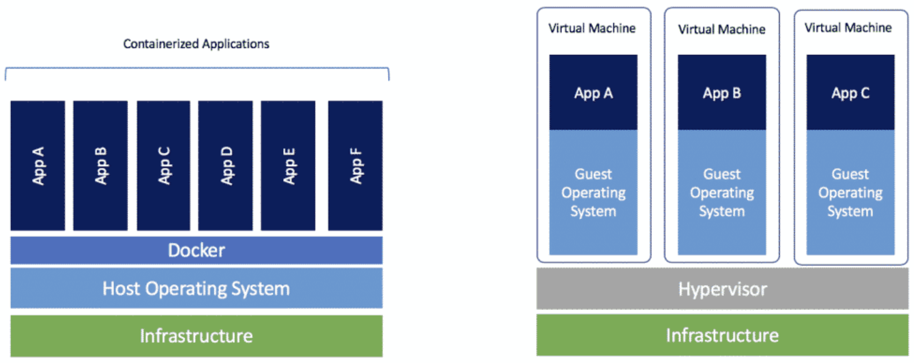
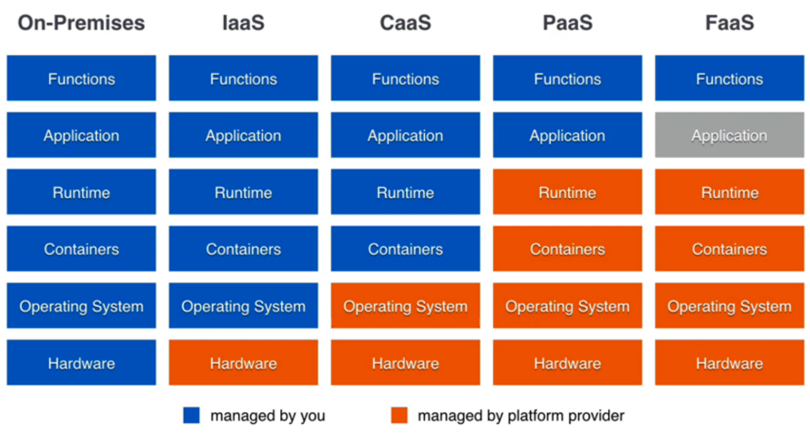


Figure 2-5. Docker compared to traditional virtual machines [16]

This has a significant importance when working in a cloud environment, where reduced resource consumption can be transformed in a reduced cost which is a great incentive for optimizing any solution. Businesses are quickly migrating from on-premise solutions to cloud computing due to the zero upfront cost needed for hosting the application and the ability to quickly adjust the amount of available computing resources. This shifts the responsibility of buying and maintaining IT hardware from the owner of the application to the cloud provider, which in most cases is a favourable solution.

There are multiple types of cloud solutions, ranging from third-party owned public cloud, such as Amazon Web Services or Microsoft Azure, to private cloud, used exclusively by a single business, or hybrid cloud, which combines both types, leveraging already existing infrastructure.



On-Premises	IaaS	CaaS	PaaS	FaaS
🟦 Functions	🟦 Functions	🟦 Functions	🟦 Functions	🟦 Functions
🟦 Application	🟦 Application	🟦 Application	🟦 Application	⬜ Application
🟦 Runtime	🟦 Runtime	🟦 Runtime	🟧 Runtime	🟧 Runtime
🟦 Containers	🟦 Containers	🟦 Containers	🟧 Containers	🟧 Containers
🟦 Operating System	🟦 Operating System	🟧 Operating System	🟧 Operating System	🟧 Operating System
🟦 Hardware	🟧 Hardware	🟧 Hardware	🟧 Hardware	🟧 Hardware

🟦 managed by you 🟧 managed by platform provider

Figure 2-6. Different type of cloud services [17]

There are also multiple services which can be offered by a provider, such as:

- Infrastructure as a Service: provides servers in a pay-as-you-go manner
- Container as a Service: provides a container engine already setup
- Platform as a Service: provides the required platform for hosting
- Function as a Service: suited for serverless architecture

2. 5. Machine Learning

Machine Learning is the science of programming computers so they can learn from data [18]. It has its roots in the 18th century work of Thomas Bayes called "An Essay towards solving a Problem in the Doctrine of Chances", which lays the mathematical foundation of the theory of probability.

In today's world, most applications incorporate machine learning in some form or other. Such an example would be the spam filter, which is an example of a classification task. Such system can be classified in multiple categories, depending on:

- Whether or not are training labels needed
    - Supervised: the system is fed with samples and their respective labels
    - Unsupervised: only samples are provided
    - Semi-supervised: the system is initially trained using samples and labels, then it reuses instances used for predictions to improve itself
    - Reinforcement Learning: special type of learning which is based on positive and negative rewards
- Whether or not they can learn on the fly
    - Online learning: trained incrementally by feeding instances sequentially or in mini batches
    - Batch learning: trained at once using all available data
- Whether they work by comparing new data points to known data points, or instead detect patterns and build a predictive model
    - Model based: build a weighted model adjusted according to the training data
    - Instance based: learns by heart the training data

Most machine learning systems use a model-based approach, ranging from linear regressions for simple tasks to random forest regressors. In order to adjust the weights of the model, either a utility function or a cost function can be used. The utility function measures how good the model fits the data, with the scope of maximizing such a value, while the cost function measures the distance between the model's prediction and the actual label in the training data, with the scope of minimizing such a value. For regressions tasks, cost functions are usually preferred, such as Root Mean Square Error (RMSE), which shows how much error the system typically has in its predictions, or Mean Absolute Error (MAE), when the training dataset contains outliers.

An evolution of classical Machine Learning systems is represented by neural networks. It is a form of machine learning that is usually supervised which attempts to create a rough parallel between with the function of the neurons present in the human brain. The smallest unit in such a network is the *perceptron*, which accepts an input and performs a computation on it. This is then fed to an activation function before passing the value to the next layer. Depending on the use case, several activation functions are available:

- Linear: passes the value without any alteration
- Step function: outputs 0 if the input value is less than 0, otherwise 1
- Rectified Linear Unit (ReLU): outputs 0 if the input value is less than 0, otherwise passes the value without any alteration
- Logistic: used for models that predict a probability since it outputs values between 0 and 1
- Tanh: similar to logistic activation function, but outputs values between -1 and 1

Before making successful predictions, the network first needs to be trained. Let's assume a simple neural network with an input layer, one hidden layer and one output layer. On model creation, the weights will be initialized with random normally distributed values. Training consists of multiple forward loops for calculating the current cost according to the cost function, followed by the process of backpropagation. During this process, the weights are adjusted according to the optimizer's strategy adjusted by the learning rate.

Recently, two new neural networks models that are suited for well-defined tasks have emerged:

- Convolutional Neural Networks (CNNs)
    - Often used for image classification
    - Introduces convolutional layers which provide the system with spatial invariance
- Recurrent Neural Networks (RNNs)
    - Used for natural language processing and time series regressions

3. PROPOSED SOLUTION

3. 1. High Level Solution Description

A single IoT node was built as a proof of concept to demonstrate the feasibility and power of the solution.

The node features an ESP32 which has several sensors attached to it, such as DTH11 temperature and humidity sensor, SW-420 vibration sensor, and MQ-2 gas sensor. While this may not provide an exhaustive list of sensors that can be attached to the node, it serves the purpose of demonstrating the flexibility with which the architecture handles different types of data.

Before pushing any data to the cloud, the device needs to have the Wi-Fi connection configured, but also to have its identity attested by the attestation server. This is done by using public key cryptography. At manufacture time, several files are burnt into the flash storage of the device: the root certificate and device key-pair which includes a public and a private key. Impersonation of the device is avoided by encrypting the flash storage before shipping the device to the end-user, making the key uncompromisable. This is done by leveraging a built-in feature of the ESP32 which transparently handles encryption and decryption of data on the fly, saving the used AES key in eFUSE [19].

Following the attestation of the node, a symmetric key has been established which will then be burnt into the encrypted flash storage to be used the next time the node will be restarted. The integrity of the key is assured by using Elliptic Curve Digital Signature Algorithm (ECDSA), signed with the device's own private key. This key will expire each 24 hours, which ensures forward secrecy if the key is compromised. Following the expiration of the key, it will be removed from non-volatile storage alongside its signature, and the node restarted to perform the attestation process again.

The symmetric key is used in establishing MQTT connection in TLS-PSK mode, which is less computationally expensive than TLS with public key cryptography. This means that the cost of establishing a key is incurred only once each 24 hours when attesting the origin of the device.

The payload will be CBOR encoded and formatted according to the IPSO guidelines and then sent to the MQTT broker in the cloud. From here, the possibilities for data manipulation are endless, enabling functional style programming, subscribing to each topic that is of interest and performing actions when data is received, or persisting the data in a relational or non-relational database for later use.

For this proof of concept, besides the previously stated components, a cloud hosted MySQL setup will be used, enabling fast responses while creating no bottleneck for simultaneous data insertions and retrievals. Two microservices have been developed using the Spring Boot framework. The first one subscribes to all the available topics of devices that have undergone the attestation process and persists the information into the database. The second one performs queries which are exposed via a HTTP REST interface.

All requests are authenticated and authorized in a stateless manner using JSON Web Tokens. This permits scaling with ease, decoupling the login from the machine that first served the request.

3. 2. Bill of Materials

Table 3-1. Bill of materials

COMPONENT	UNIT PRICE(LEI)	QUANTITY	TOTAL PRICE
ESP32 DEV BOARD	42.99	1	42.99
DTH11 TEMP AND HUMIDITY SENSOR	11.99	1	11.99
SW-420 VIBRATION SENSOR	4.89	1	4.89
MQ-2 GAS SENSOR	12.49	1	12.49
BATTERY PACK	109.99	1	109.99
TOTAL		182.35	

1. ESP32 Development Board

ESP32 is a series of low-powered system on a chip microcontrollers featuring Wi-Fi and Bluetooth, developed by Espressif Systems, a Shanghai-based Chinese company, and manufactured by TSMC using their 40nm node.

The particular flavour used here is ESP32S onto an ESP-WROOM-32 derived development board. This features a dual core Tensilica LX6 clocked at either 160 or 240MHz with a 32bit architecture, has an Ultra-Low Power coprocessor, 520KB of RAM and 4MB of flash storage that can be partitioned to the user’s preference. It also features Wi-Fi 802.11 b/g/n and Bluetooth 4.2 with BLE support.

This board is used as an IoT node and performs cryptographic operations, sensor reading and sends data over MQTT to the broker hosted in a cloud instance.

2. DTH11 Temperature and Humidity Sensor

DHT11 is a temperature and humidity sensor that output a digital signal on the data pin. Temperature is measured using a negative coefficient thermistor (NTC) and the relative humidity is measured using a capacitive sensor. The humidity sensing range can be between 20 and 90 % RH and its measurement accuracy is of +/- 5% RH. The temperature measurement range is from 0 to 60◦C and its accuracy is of +/- 2◦C. The supply voltage accepts ranges between 3.3V and 5V.

3. SW-420 Vibration Sensor

The vibration sensor used has a LM393 comparator and uses the SW-420 vibration sensor module that is normally closed, outputting a digital signal on the data pin. Under normal circumstances, the vibration switch is under closed conduction state, outputting digital low. Under strong vibrations, the sensor will output high. It has operating voltage ranging from 3.3V to 5V.

4. MQ-2 Gas Sensor

MQ-2 gas sensor is highly responsive and very sensitive, capable of detecting gases such as: butane, liquefied petroleum gas, methane, and smoke.

It is based on a Metal Oxide Semiconductor type sensor, also known as chemiresistors, as the detection is based upon change of resistance of the sensing material when the gas meets the material.

5. Battery pack

The battery pack provides power to the IoT node, so it can be placed where there are no wall plugs.

3. 3. In-depth explanation of the solution

For clarity, this solution can be split into two parts, the IoT node and the cloud applications.

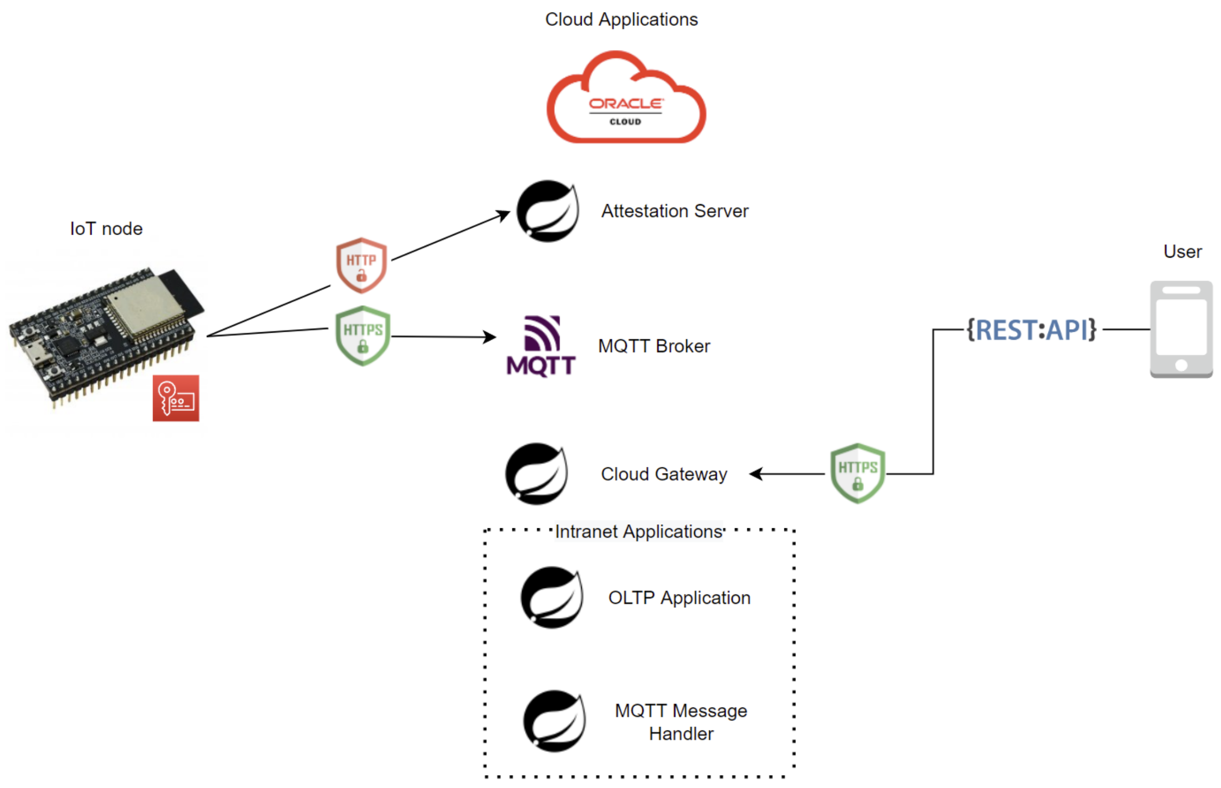


Figure 3-1. High Level Architecture Overview

The IoT node is composed of an ESP32 with several sensors connected to it using a breadboard. When designing the node, careful planning had to be done regarding the processing power, memory capacity and dimensions while still keeping the costs as low as possible in order to provide a marketable solution. For example, using a Raspberry Pi would have rendered faster processing times and easier development, but the footprint of the node would be bigger, and the price would be a real deterrent, ballooning the cost at least 5 times. Since this is a price elastic market [20], such a device would need an attractive price in order to gain market traction.

For the perception layer of the solution, three sensors were used, the DTH11 temperature and humidity sensor, the SW-420 vibration sensor and MQ-2 gas sensor. The ESP32 development board chosen features 30 GPIO pins, of which 18 Analog-to-Digital channels, 3 SPI interfaces, 3 UART interfaces, 2 I2C interfaces, 16 PWM output channels, 2 Digital-to-Analog, 2 I2S interfaces and 10 capacitive sensing GPIOs. According to the datasheet, 10 Analog-to-Digital channels are used by the Wi-Fi card, but the solution’s requirements are still met. Power was used from both the 3.3V pin and the VIN which supplies 5V when connecting the sensors, although preferring the lower voltage when possible. When connecting the temperature and humidity sensor, a 10K Ω pull up resistor was connected on the data pin. This outputs digital signal, and as such was connected to the GPIO 4 pin. The SW-420 vibration sensor was also connected to the 3.3V power source and ground, while the data pin was connected to the GPIO 18 pin in digital mode. The MQ-2 gas sensor requires a burn in period of 24 to 72 hours in order to work correctly and outputs an analogue signal, as such being connected to the GPIO 35 pin. Internally, this pin maps to the Analog-to-Digital converter ADC1, which is still available despite using Wi-Fi.

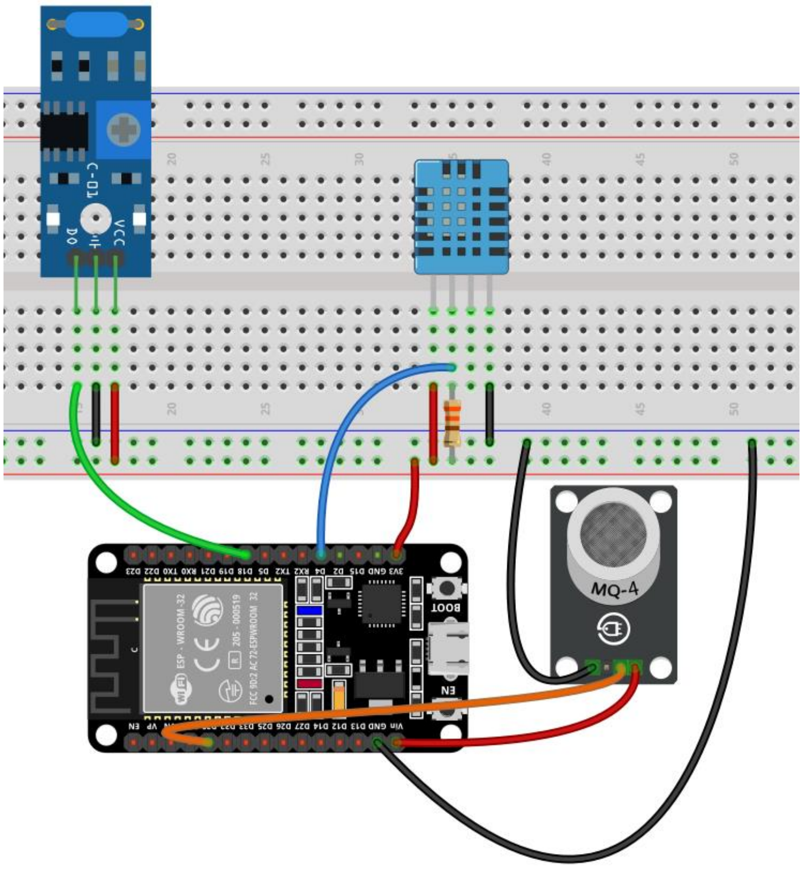


Figure 3-2. Wiring Diagram

After wiring up the node and the sensors, another important decision had to be made regarding the development stack used. While the OS of the device is FreeRTOS, the manufacturer offers two possible abstractions on top: be it Arduino framework, or their own ESP-IDF (Espressif Integrated Development Framework). The difference is significant, Arduino offering an OOP paradigm along with a huge amount of community created libraries, while ESP-IDF is task oriented and has manufacturer developed libraries, along with a tight integration with the hardware and access to low level functions. The flip side is that both use a modified version of the GCC (GNU Compiler Collection), offering some level of portability. This proved to be very important, since back and forth movement was performed during the development process as issues arose.

The solution started as an Arduino project, but the security-sensitive aspect of the solution proved to be impossible to implement without the tight knit integration with the hardware of the ESP-IDF. This is due to the fact that the Arduino abstraction is shipped as a binary distribution, precompiled with the settings the Espressif deemed most relevant while not taking up unnecessary space in the non-volatile memory, while the ESP-IDF allows building from source code using a highly configurable architecture. The configurability is achieved by using a custom build system based partly on CMake, with compile time settings being altered by using the *menuconfig* [21]. This switch allowed free access to all hardware features, such as the transparent runtime encryption of flash memory needed in order to fill in the security puzzle, being able to ensure the confidentiality of the files stored in flash non-volatile memory. Another reason for the switch was the absence of the PSK option from the Arduino binaries, needed to smoothen the connection process to the broker. While this solved most of the issues, drawbacks were present such as the loss of important Arduino libraries that were used. While they were written in C++, and with enough effort any code can be built with any build system, dependencies of the Arduino framework were baked into every third-party library. Even though the dependencies could be removed, be it rewriting the third-party library or writing new code from scratch, time is of the essence, being one of the most valuable resources. This is where the flexibility of the ESP-IDF build system came into play, tying in the whole solution together, allowing the compiling of the Arduino sources and linkage with the other components. Each Arduino library still had to be integrated with the CMake based build system, but it required significantly less effort than writing the required functionality from scratch.

With the underlying technical stack detailed, attention can now be shifted to the functional part of the product. The core functionality of the solution is secure transmission of sensor data directly into the cloud, while still offering an easy to expand and pluggable architecture. Sensor data is handled transparently and collection of it is abstracted in such a manner that new sensor capabilities can be added with little to no effort. The pluggable aspect of the architecture is offered by the possibility to enable or disable optional features at runtime, such as running ML inference on the edge node using a pretrained neural network specialized in anomaly detection or sending blockchain transactions using the sensor data as parameter.

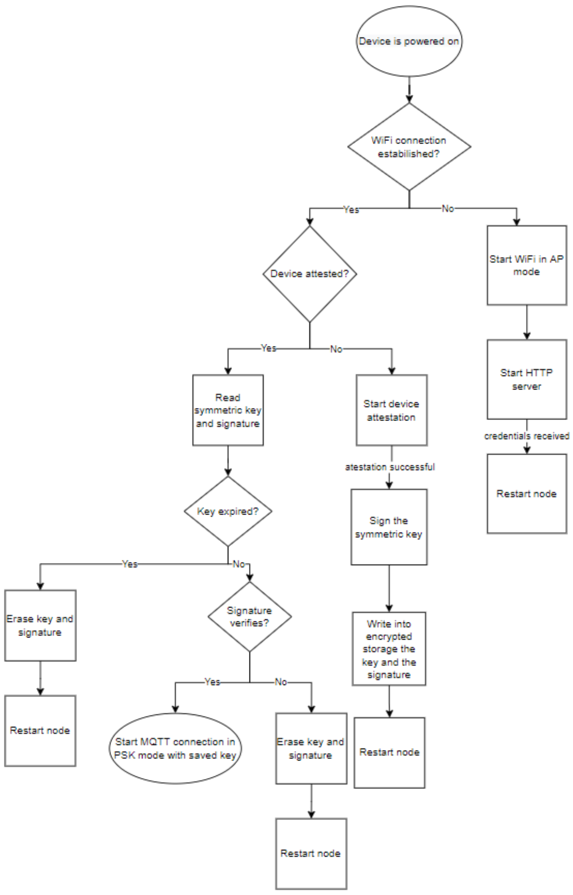
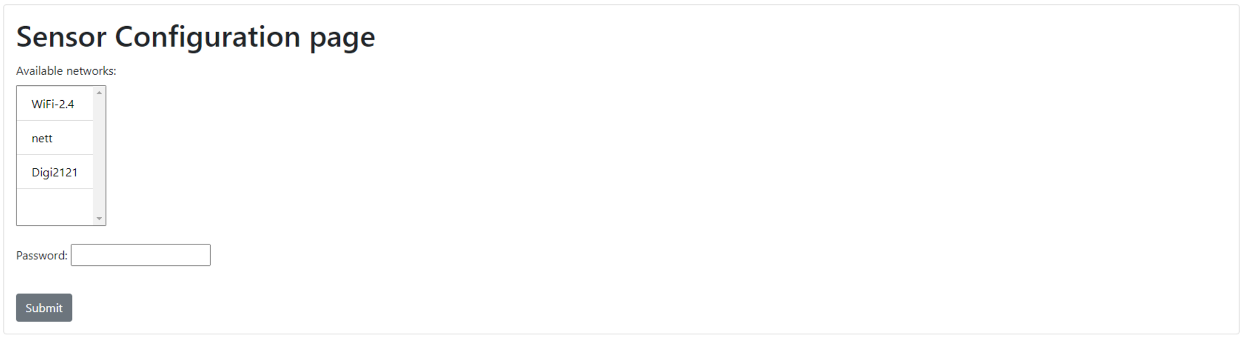


Figure 3-3. Node initialization process

The initialization process of the node as seen in Figure 3-3 starts with the establishment of the Wi-Fi connection. At first boot, no credentials would have been stored inside the ESP32, point at which the node would switch from Wi-Fi station mode into Access Point mode, start an HTTP Server and wait for the user to enter the network credentials.

There is no application level security at this point, relying solely on the ISO/OSI Layer 2 security offered by the network, since the access point is accessible only with a password.

The user is aided into inputting the correct network name by performing a scan of available networks and embedding that information before serving the HTML page to the browser. The Arduino library AsyncWebServer was used to create the HTTP server, with several modifications in order to be able to be built under ESP-IDF environment. The library is fully asynchronous, even supporting advanced features such as serving pages from flash storage and template expansion of strings. The configuration page was created using HTML5 and Bootstrap, the credentials being sent using HTTP POST and persisted into the node. After setting the credentials, the node would restart and either proceed to the attestation process if the connection would be established successfully or switch back to Access Point mode and wait for new credentials to be inputted.


Sensor Configuration page

Available networks:

WiFi-2.4

nett

Digi2121

Password:

Figure 3-4. Sensor Configuration page

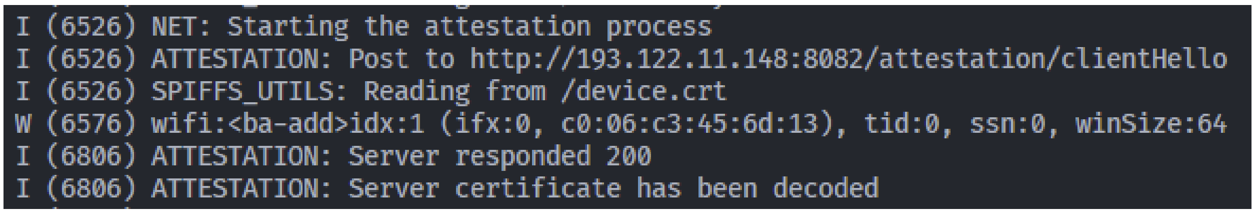
Following successful establishment of network connection, the attestation process is undergone, which implies a multi-step flow involving complex cryptographic operations. The scope of this process is to attest the origin of the node, along with session key establishment used for connecting to the MQTT broker in TLS-PSK mode.

The process involves two parties, the IoT node and the attestation server, using plain HTTP as transport.

This process is initiated by the IoT node and is performed using plain HTTP. On the server side, a Java application was developed using the Spring Boot framework and a custom abstraction layer on top of Java Cryptography Architecture (JCA). Bouncy Castle was also added as a provider in order to supply the needed elliptic curve cryptography operations. On the node side, the HTTP Client provided as part of ESP32's Arduino framework was used for issuing requests, while the cryptographic operations were supplied from a custom built library that uses mbedTLS. Making the JCA and mbedTLS work together deemed quite challenging, seeing that both have their own proprietary representation of keys and parameters. The response to this was unwrapping all the JCA abstraction and accessing the raw fields of the Bouncy Castle implementation in order to provide them in a manner that mbedTLS would accept. Being a library designed to be used in low-powered embedded environments, this often meant that the input is expected to be uncompressed and unencoded.

The attestation is commenced using HTTP Post to */clientHello* endpoint passing the device certificate encoded in Base64. The server then decodes the certificate and attempts to create an JCA object representation of it. If this step fails, the server will respond with HTTP status code 400, also known as bad request, and will halt the attestation process. Otherwise, the certificate is stored in a Hash-Map stored in volatile memory, with the key being the IP of the request. The storage is in-memory since cleaning of unused database records would be a processing intensive task in a high-volume environment, with the owned cost of increased memory consumption. As a response, the X and Y coordinates of the server's public key will be sent, encoded in Base64.

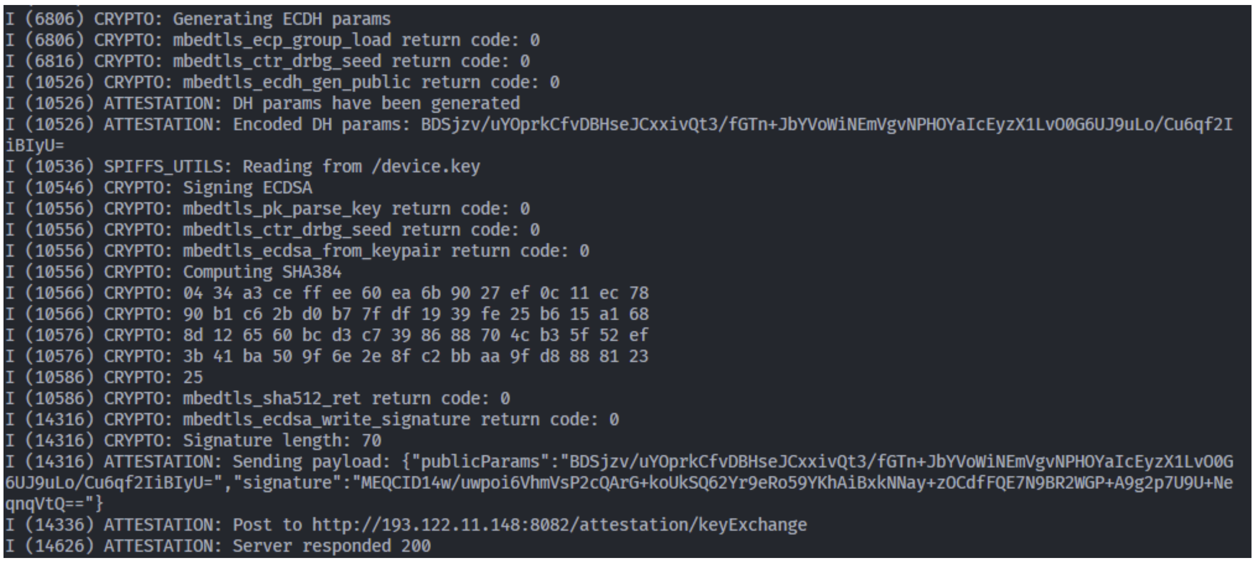
Besides this, the origin of the node can be attested by verifying the certificate's signature, having the requirement to be signed by the same Certification Authority as the attestation server.



```
I (6526) NET: Starting the attestation process
I (6526) ATTESTATION: Post to http://193.122.11.148:8082/attestation/clientHello
I (6526) SPIFFS_UTILS: Reading from /device.crt
W (6576) wifi:<ba-add>idx:1 (ifx:0, c0:06:c3:45:6d:13), tid:0, ssn:0, winSize:64
I (6806) ATTESTATION: Server responded 200
I (6806) ATTESTATION: Server certificate has been decoded
```

Figure 3-5. Attestation /clientHello

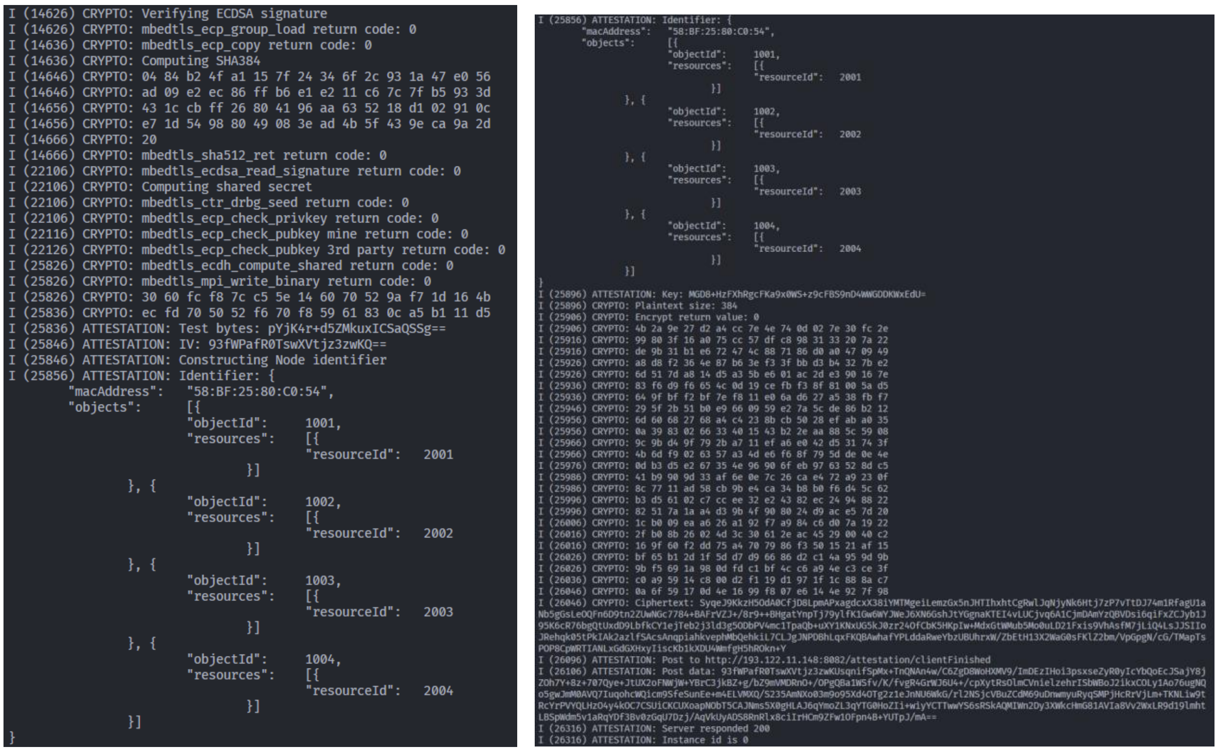
The second step of the attestation process is commenced by issuing a HTTP POST request to */keyExchange*. Elliptic Curve Diffie Hellman parameters are generated on the client side, with which the request body is constructed as the Base64 representation of the concatenated X and Y coordinates of the public point. While this normally ensures the generation of shared secret key, forward secrecy is also provided by making this process ephemeral, new parameters being generated for each request. Non-repudiation is assured by providing a digital signature of the Base64 representation, signed with the device's private key. The useful information is then formatted as a JSON object and sent to the attestation server. On receiving the payload, the server will attempt to retrieve the certificate of the other party by using the IP of the request. If no such certificate is present, the attestation process will halt, sending HTTP status code 400 Bad Request in response. Otherwise, the public Diffie Hellman parameters and the signature will be decoded from Base64. Using the public key from the certificate stored in the previous attestation step, the signature is verified. Since the data exchange was performed over an unsecured transport protocol, assuring the integrity of the data is of utmost importance. If the signature doesn't verify, it means that the payload has been tampered with and the attestation process will stop, returning the same HTTP 400 Bad Request code as before. If the signature is valid, the server generates its own set of Diffie Hellman parameters, packages the public point using the same format as the node, concatenated value of X and Y encoded using Base64, and signs the representation. Along with the aforementioned components, a 16 bytes random sequence is generated in order to ensure that the same session key is generated by both parties. Since the server already has all the ingredients needed to establish the secret key, it will be generated at this step and saved in secure storage. The response is then sent to the IoT node in the form of a JSON object.



```
I (6806) CRYPTO: Generating ECDH params
I (6806) CRYPTO: mbedtls_ecp_group_load return code: 0
I (6816) CRYPTO: mbedtls_ctr_drbg_seed return code: 0
I (10526) CRYPTO: mbedtls_ecdh_gen_public return code: 0
I (10526) ATTESTATION: DH params have been generated
I (10526) ATTESTATION: Encoded DH params: BDSjzv/uYOprkCfvDBHseJCxxivQt3/fGTn+JbYVoWiNEmVgvNPHOYaIcEyzX1LvO0G6UJ9uLo/Cu6qf2I
iBIyU=
I (10536) SPIFFS_UTILS: Reading from /device.key
I (10546) CRYPTO: Signing ECDSA
I (10556) CRYPTO: mbedtls_pk_parse_key return code: 0
I (10556) CRYPTO: mbedtls_ctr_drbg_seed return code: 0
I (10556) CRYPTO: mbedtls_ecdsa_from_keypair return code: 0
I (10556) CRYPTO: Computing SHA384
I (10566) CRYPTO: 04 34 a3 ce ff ee 60 ea 6b 90 27 ef 0c 11 ec 78
I (10566) CRYPTO: 90 b1 c6 2b d0 b7 7f df 19 39 fe 25 b6 15 a1 68
I (10576) CRYPTO: 8d 12 65 60 bc d3 c7 39 86 88 70 4c b3 5f 52 ef
I (10576) CRYPTO: 3b 41 ba 50 9f 6e 2e 8f c2 bb aa 9f d8 88 81 23
I (10586) CRYPTO: 25
I (10586) CRYPTO: mbedtls_sha512_ret return code: 0
I (14316) CRYPTO: mbedtls_ecdsa_write_signature return code: 0
I (14316) CRYPTO: Signature length: 70
I (14316) ATTESTATION: Sending payload: {"publicParams":"BDSjzv/uYOprkCfvDBHseJCxxivQt3/fGTn+JbYVoWiNEmVgvNPHOYaIcEyzX1LvO0G
6UJ9uLo/Cu6qf2IiBIyU=","signature":"MEQCID14w/uwpoi6VhmVsP2cQArG+koUkSQ62Yr9eRo59YKhAiBxkNNay+zOCdfFQE7N9BR2WGP+A9g2p7U9U+Ne
qnqVtQ=="}
I (14336) ATTESTATION: Post to http://193.122.11.148:8082/attestation/keyExchange
I (14626) ATTESTATION: Server responded 200
```

Figure 3-6. Attestation /keyExchange

The third and last step of the attestation process is represented by the */clientFinish* HTTP POST endpoint. Before issuing the request, the node decodes all the parameters from the */keyExchange* from Base64 and verifies the signature of received data. If the signature doesn't verify, an error message is logged, and the node restarted. Otherwise, it instantiates an elliptic curve point based on the server X and Y points and uses it along its own Diffie Hellman parameters to generate the shared secret. Besides generating a shared key and attesting the origin of the node, this process also allows the IoT node to present its capabilities to the attestation server, along with several identifying information. These capabilities should be tamper resistant. Several solutions were candidates, such as using a salted message digest to ensure data integrity, since otherwise if no salt was used an attacker could just recompute the hash of the desired value, digitally signing the capabilities, or using an encryption algorithm. In the case of encryption, the data would be irrecoverable when tampered with, confidentiality being ensured while still providing a mean to determine any tampering attempt. The device identifier consists of the MAC address of the IoT node's Wi-Fi card, a uniquely identifying property of the node since this is burnt into the chip at manufacture time and doesn't allow permanent change, along with the list of capabilities formatted in an IPSO Smart Object friendly way. As such, the capability is identified by an object ID, which uniquely identifies a singular measurement value, e.g. temperature, and a list of resources presented by the object. The resources are identified by an ID, and represent a certain piece of information the object can provide, e.g. maximum temperature, minimum temperature, current value. This identifier will then be appended to the test bytes sent by the server and encrypted in AES/CBC mode using the newly generated shared key, setting a randomly generated 16 bytes sequence as initialization vector. Since the initialization vector is not secret, nor the resulting ciphertext, these are concatenated and encoded in Base64 to be sent in the request body. The instance ID will be returned by the attestation server as a result of several operations.

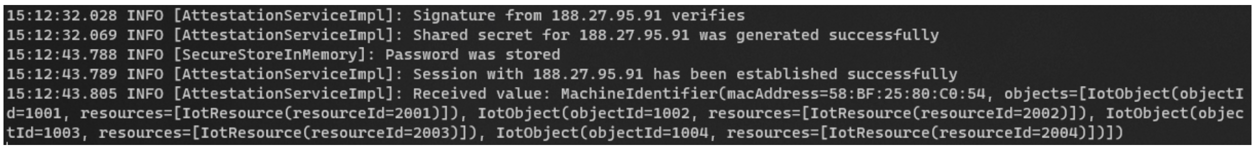


```
I (14626) CRYPTO: Verifying ECDSA signature
I (14626) CRYPTO: mbedtls_ecp_group_load return code: 0
I (14636) CRYPTO: mbedtls_ecp_copy return code: 0
I (14636) CRYPTO: Computing SHA384
I (14646) CRYPTO: 04 84 b2 4f a1 15 7f 24 34 6f 2c 93 1a 47 e0 56
I (14646) CRYPTO: ad 09 e2 ec 86 ff b6 e1 e2 11 c6 7c 7f b5 93 3d
I (14656) CRYPTO: 43 1c cb ff 26 80 41 96 aa 63 52 18 d1 02 91 0c
I (14656) CRYPTO: e7 1d 54 98 80 49 08 3e ad 4b 5f 43 9e ca 9a 2d
I (14666) CRYPTO: 20
I (14666) CRYPTO: mbedtls_sha512_ret return code: 0
I (22106) CRYPTO: mbedtls_ecdsa_read_signature return code: 0
I (22106) CRYPTO: Computing shared secret
I (22106) CRYPTO: mbedtls_ctr_drbg_seed return code: 0
I (22106) CRYPTO: mbedtls_ecp_check_privkey return code: 0
I (22116) CRYPTO: mbedtls_ecp_check_pubkey mine return code: 0
I (22126) CRYPTO: mbedtls_ecp_check_pubkey 3rd party return code: 0
I (25826) CRYPTO: mbedtls_ecdh_compute_shared return code: 0
I (25826) CRYPTO: mbedtls_mpi_write_binary return code: 0
I (25826) CRYPTO: 30 60 fc f8 7c c5 5e 14 60 70 52 9a f7 1d 16 4b
I (25836) CRYPTO: ec fd 70 50 52 f6 70 f8 59 61 83 0c a5 b1 11 d5
I (25836) ATTESTATION: Test bytes: pYjK4r+d5ZMkuxICSaQSSg==
I (25846) ATTESTATION: IV: 93fWPafR0TswXVtjz3zwKQ==
I (25846) ATTESTATION: Constructing Node identifier
I (25856) ATTESTATION: Identifier: {
        "macAddress":   "58:BF:25:80:C0:54",
        "objects":      [{
                        "objectId":     1001,
                        "resources":    [{
                                        "resourceId":   2001
                                }]
                }, {
                        "objectId":     1002,
                        "resources":    [{
                                        "resourceId":   2002
                                }]
                }, {
                        "objectId":     1003,
                        "resources":    [{
                                        "resourceId":   2003
                                }]
                }, {
                        "objectId":     1004,
                        "resources":    [{
                                        "resourceId":   2004
                                }]
                }]
}
```

```
I (25856) ATTESTATION: Identifier: {
        "macAddress":   "58:BF:25:80:C0:54",
        "objects":      [{
                        "objectId":     1001,
                        "resources":    [{
                                        "resourceId":   2001
                                }]
                }, {
                        "objectId":     1002,
                        "resources":    [{
                                        "resourceId":   2002
                                }]
                }, {
                        "objectId":     1003,
                        "resources":    [{
                                        "resourceId":   2003
                                }]
                }, {
                        "objectId":     1004,
                        "resources":    [{
                                        "resourceId":   2004
                                }]
                }]
}
I (25896) ATTESTATION: Key: MGD8+HzFXhRgcFKa9x0WS+z9cFBS9nD4WWGDDKWxEdU=
I (25896) CRYPTO: Plaintext size: 384
I (25906) CRYPTO: Encrypt return value: 0
I (25906) CRYPTO: 4b 2a 9e 27 d2 a4 cc 7e 4e 74 0d 02 7e 30 fc 2e
I (25916) CRYPTO: 99 80 3f 16 a0 75 cc 57 df c8 98 31 33 20 7a 22
I (25916) CRYPTO: de 9b 31 b1 e6 72 47 4c 88 71 86 d0 a0 47 09 49
I (25926) CRYPTO: a8 d8 f2 36 4e 87 b6 3e f3 3f bb d3 b4 32 7b e2
I (25926) CRYPTO: 6d 51 7d a8 14 d5 a3 5b e6 01 ac 2d e3 90 16 7e
I (25936) CRYPTO: 83 f6 d9 f6 65 4c 0d 19 ce fb f3 8f 81 00 5a d5
I (25936) CRYPTO: 64 9f bf f2 bf 7e f8 11 e0 6a d6 27 a5 38 fb f7
I (25946) CRYPTO: 29 5f 2b 51 b0 e9 66 09 59 e2 7a 5c de 86 b2 12
I (25956) CRYPTO: 6d 60 68 27 68 a4 c4 23 8b cb 50 28 ef ab a0 35
I (25956) CRYPTO: 0a 39 83 02 66 33 40 15 43 b2 2e aa 88 5c 59 08
I (25966) CRYPTO: 9c 9b d4 9f 79 2b a7 11 ef a6 e0 42 d5 31 74 3f
I (25966) CRYPTO: 4b 6d f9 02 63 57 a3 4d e6 f6 8f 79 5d de 0e 4e
I (25976) CRYPTO: 0d b3 d5 e2 67 35 4e 96 90 6f eb 97 63 52 8d c5
I (25986) CRYPTO: 41 b9 90 9d 33 af 6e 0e 7c 26 ca e4 72 a9 23 0f
I (25986) CRYPTO: 8c 77 11 ad 58 cb 9b e4 ca 34 b8 b0 f6 d4 5c 62
I (25996) CRYPTO: b3 d5 61 02 c7 cc ee 32 e2 43 82 ec 24 94 88 22
I (25996) CRYPTO: 82 51 7a 1a a4 d3 9b 4f 90 80 24 d9 ac e5 7d 20
I (26006) CRYPTO: 1c b0 09 ea a6 26 a1 92 f7 a9 84 c6 d0 7a 19 22
I (26016) CRYPTO: 2f b0 8b 26 02 4d 3c 30 61 2e ac 45 29 00 40 c2
I (26016) CRYPTO: 16 9f 60 f2 dd 75 a4 70 79 86 f3 50 15 21 af 15
I (26026) CRYPTO: bf 65 b1 2d 1f 5d d7 d9 66 86 d2 c1 4a 95 9d 9b
I (26026) CRYPTO: 9b f5 69 1a 98 0d fd c1 bf 4c c6 a9 4e c3 ce 3f
I (26036) CRYPTO: c0 a9 59 14 c8 00 d2 f1 19 d1 97 1f 1c 88 8a c7
I (26046) CRYPTO: 0a 6f 59 17 0d 4e 16 99 f8 07 e6 14 4e 92 7f 98
I (26046) CRYPTO: Ciphertext: SyqeJ9KkzH5OdA0CfjD8LpmAPxagdcxX38iYMTMgeiLemzGx5nJHTIhxhtCgRwlJqNjyNk6Htj7zP7vTtDJ74m1RfagU1a
Nb5gGsLeOQFn6D9tn2ZUwNGc7784+BAFrVZJ+/8r9++BHgatYnpTj79ylfK1Gw6WYJWeJ6XN6GshJtYGgnaKTEI4vLUCjvq6A1CjmDAmYzQBVDsi6qiFxZCJyb1J
95K6cR76bgQtUxdD9LbfkCY1ejTeb2j3ld3g5ODbPV4mc1TpaQb+uXY1KNxUG5kJ0zr24OfCbK5HKpIw+MdxGtWMub5Mo0uLD21Fxis9VhAsfM7jLiQ4LsJJSIIo
JRehqk05tPkIAk2azlfSAcsAnqpiahkvephMbQehkiL7CLJgJNPDBhLqxFKQBAwhafYPLddaRweYbzUBUhrxW/ZbEtH13X2WaG0sFKlZ2bm/VpGpgN/cG/TMapTs
POP8CpWRTIANLxGdGXHxyIiscKb1kXDU4WmfgH5hROkn+Y
I (26096) ATTESTATION: Post to http://193.122.11.148:8082/attestation/clientFinished
I (26106) ATTESTATION: Post data: 93fWPafR0TswXVtjz3zwKUsqnifSpMx+TnQNAn4w/C6ZgD8WoHXMV9/ImDEzIHoi3psxseZyR0yIcYbQoEcJSajY8j
ZOh7Y+8z+707Qye+JtUX2oFNWjW+YBrC3jkBZ+g/bZ9mVMDRnO+/OPgQBa1WSfv/K/fvgR4GrWJ6U4+/cpXytRsOlmCVnielzehrISbWBoJ2ikxCOLy1Ao76ugNQ
o5gwJmM0AVQ7IuqohcWQicm9SfeSunEe+m4ELVMXQ/S235AmNXo03m9o95Xd4OTg2z1eJnNU6WkG/rl2NSjcVBuZCdM69uDnwmyuRyqSMPjHcRrVjLm+TKNLiw9t
RcYrPVYQLHzO4y4kOC7CSUiCKCUXoapNObT5CAJNms5X0gHLAJ6qYmoZL3qYTG0HoZIi+wiyYCTTwwYS6sRSkAQMIWn2Dy3XWkcHmG81AVIa8Vv2WxLR9d19lmht
LBSpWdm5v1aRqYDf3Bv0zGqU7Dzj/AqVkUyADS8RnRlx8ciIrHCm9ZFw1OFpn4B+YUTpJ/mA==
I (26316) ATTESTATION: Server responded 200
I (26316) ATTESTATION: Instance id is 0
```

Figure 3-7. Attestation /clientFinish

On receiving the request, the server decodes the payload from Base64 to a byte array representation, which is then split into IV and ciphertext arrays. Based on the IP of the request, the previously generated secret key is retrieved from secure storage, removing it in the process. If no such key exists, the attestation process is marked as failed and HTTP status code 400 will be returned. Otherwise, the ciphertext is decrypted and the first 16 bytes of it are checked against the previously sent test bytes sequence. If the bytes don't match, the process is ended as failed and HTTP status code 400 will be returned.



```
15:12:32.028 INFO [AttestationServiceImpl]: Signature from 188.27.95.91 verifies
15:12:32.069 INFO [AttestationServiceImpl]: Shared secret for 188.27.95.91 was generated successfully
15:12:43.788 INFO [SecureStoreInMemory]: Password was stored
15:12:43.789 INFO [AttestationServiceImpl]: Session with 188.27.95.91 has been established successfully
15:12:43.805 INFO [AttestationServiceImpl]: Received value: MachineIdentifier(macAddress=58:BF:25:80:C0:54, objects=[IotObject(objectI
d=1001, resources=[IotResource(resourceId=2001)]), IotObject(objectId=1002, resources=[IotResource(resourceId=2002)]), IotObject(objec
tId=1003, resources=[IotResource(resourceId=2003)]), IotObject(objectId=1004, resources=[IotResource(resourceId=2004)])])
```

Figure 3-8. Attestation server logs

In an attempt to save computational effort, the generated secret key is then used by the node in order to establish MQTT connection in TLS-PSK mode. As such, the attestation server also handles the insertion and removal of PSK keys in the broker configuration file. The MQTT broker expects the file that stores the pre-shared keys to be formatted as *hint:key*, where key is the hex representation of the secret key. On each new modifying operation on the secure storage that handles PSK keys in the attestation server, the whole content of the file is overwritten by the keys saved in the server using the previously specified format. In order for the changes to take effect, a Java ProcessBuilder had to be used which calls a shell script on the host machine. As such, a limitation can be observed, the tight coupling of the attestation server with the MQTT broker.

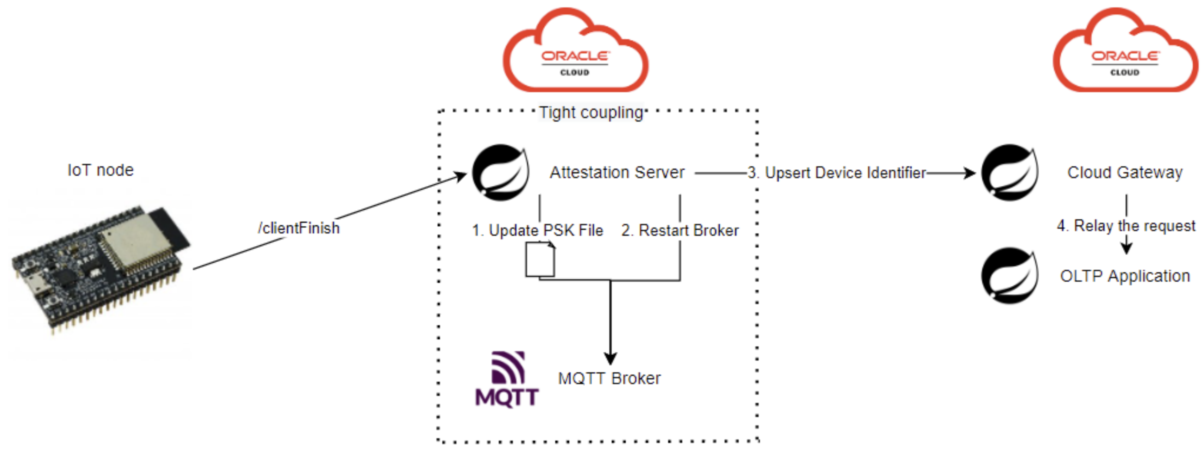
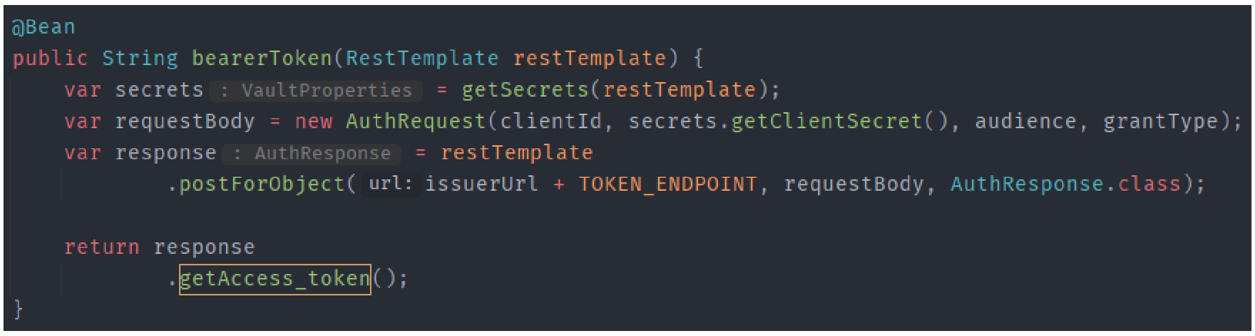


Figure 3-9. Last step of the attestation process

Besides restarting the broker, the server also sends the device identifier to the OLTP microservice in order to have the node information persisted. The request is authenticated using JSON Web Tokens and is performed using HTTPS which offers transport layer security thanks to TLS.

The authentication and authorization of users is delegated to Auth0, the latter issuing and verifying the tokens. In order to obtain an authorization code, an HTTPS POST request is sent to the issuer URL which in turn verifies the client's data and returns a Bearer Token. This token must be bundled with each request that requires authentication, in the Authorization header.

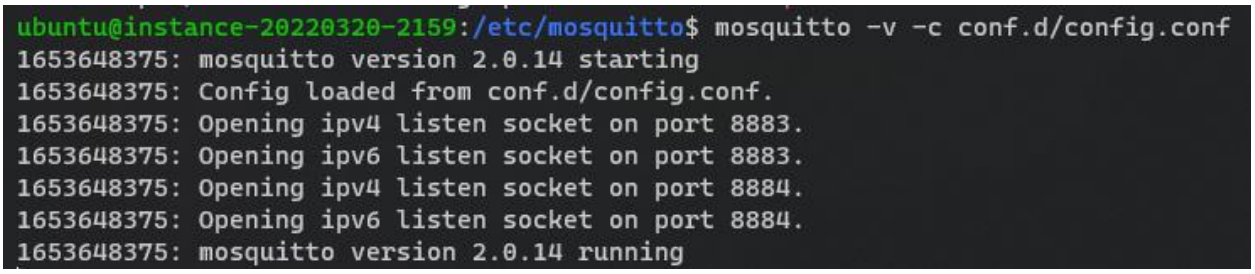


```
@Bean
public String bearerToken(RestTemplate restTemplate) {
    var secrets : VaultProperties = getSecrets(restTemplate);
    var requestBody = new AuthRequest(clientId, secrets.getClientSecret(), audience, grantType);
    var response : AuthResponse = restTemplate
            .postForObject( url: issuerUrl + TOKEN_ENDPOINT, requestBody, AuthResponse.class);

    return response
            .getAccess_token();
}
```

Figure 3-10. Obtaining the Bearer Token

In order to save computational effort, the session key is persisted after a successful attestation process alongside the timestamp generated and the digital signature of aforementioned properties concatenated. Forward secrecy is achieved by expiring the key every 24 hours, undergoing the attestation process again after this period in order to generate a new key. This ensures that a compromised key cannot be used to decrypt past ciphertext.

On the same machine that the attestation server is hosted resides the MQTT Broker, in the form of Eclipse Mosquitto. The broker was setup to allow incoming connections over TLS, be it either with pre-shared keys on port 8883 or with certificates on port 8884.

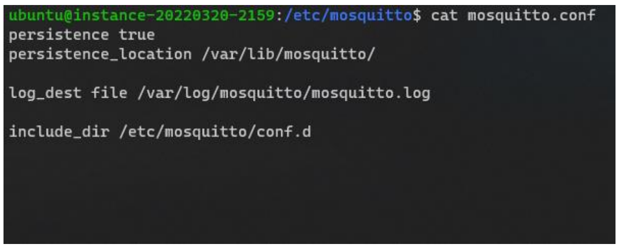


```
ubuntu@instance-20220320-2159:/etc/mosquitto$ mosquitto -v -c conf.d/config.conf
1653648375: mosquitto version 2.0.14 starting
1653648375: Config loaded from conf.d/config.conf.
1653648375: Opening ipv4 listen socket on port 8883.
1653648375: Opening ipv6 listen socket on port 8883.
1653648375: Opening ipv4 listen socket on port 8884.
1653648375: Opening ipv6 listen socket on port 8884.
1653648375: mosquitto version 2.0.14 running
```

Figure 3-11. Mosquitto Start-up

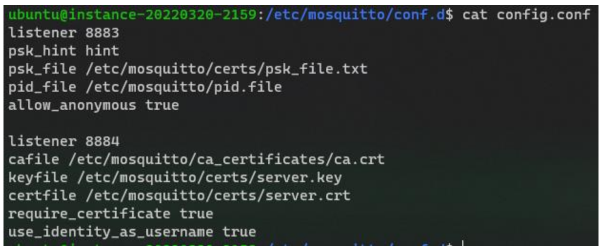
Two listeners were used instead of one in order to provide more flexibility, the certificates endpoint being easier to approach with applications running on more conventional computers with x86_64 or aarch64 architecture since those have more processing power and leverage the already existing Public Key Infrastructure, while the IoT nodes use the pre-shared keys listener since the key is generated during the attestation process. The broker will forward all published messages to the corresponding subscribers, while also allowing persistence of last message for each topic with the purpose of distributing to newly subscribed applications. The delivery guarantee is affected by the Quality-of-Service flag, ranging from 0 to 2; 0 is the *At most once delivery* mode, where no response is sent by the receiver and no retry is performed by the sender, hence no guarantee is offered. 1 is the *At least once delivery* mode, where the packet is acknowledged by the sender with a *PUBACK* packet. Mode 2 offers the highest guarantee, *Exactly once delivery*, for use when neither loss nor duplication of messages are acceptable.

While the broker comes pre-configured with sensible default settings, a custom configuration was created and placed inside the */etc/mosquito/conf.d* directory. This directory is specified as the include directory in the global Mosquitto settings.



```
ubuntu@instance-20220320-2159:/etc/mosquitto$ cat mosquitto.conf
persistence true
persistence_location /var/lib/mosquitto/

log_dest file /var/log/mosquitto/mosquitto.log

include_dir /etc/mosquitto/conf.d
```

Figure 3-12. Global Mosquitto Settings



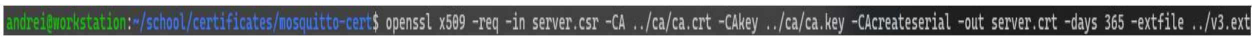
```
ubuntu@instance-20220320-2159:/etc/mosquitto/conf.d$ cat config.conf
listener 8883
psk_hint hint
psk_file /etc/mosquitto/certs/psk_file.txt
pid_file /etc/mosquitto/pid.file
allow_anonymous true

listener 8884
cafile /etc/mosquitto/ca_certificates/ca.crt
keyfile /etc/mosquitto/certs/server.key
certfile /etc/mosquitto/certs/server.crt
require_certificate true
use_identity_as_username true
```

Figure 3-13. Custom Mosquitto Settings

Besides the aforementioned options, some other settings are mandatory depending on the security scheme used. For PSK secured listener, a *psk_file* must be specified which stores the pre-shared keys in the *hint:key* format, where key is the hex representation of it. Since the key is the result of the attestation process, the classic username and password combo for authentication can be skipped by setting the *allow_anonymous* option to *true*.

For the listener secured by using the Public Key infrastructure, the certificate of the CA must be specified, along with the server key pair. The presence of certificates at connection time can be enforced by setting the *require_certificate* option to *true*, while also using the subject of the certificate with *use_identity_as_username* to *true*. To ease development, the certificates were self-generated instead of using a well-known Certification Authority. This practically created my own Public Key Infrastructure, being able to sign an unlimited number of certificates. For all the certificates, OpenSSL was used to create the private keys, certificate signing requests and the certificates.

First of all, the CA key-pair had to be generated. Since this is the root CA, it is self-signed. After this step, the server key-pair can be generated, making sure to specify in the Subject Alternative Name X509 v3 Field the IP address of the broker that is supposed to be installed on. Such a field can be populated using OpenSSL after creating the certificate signing request by specifying the -*extfile* option:

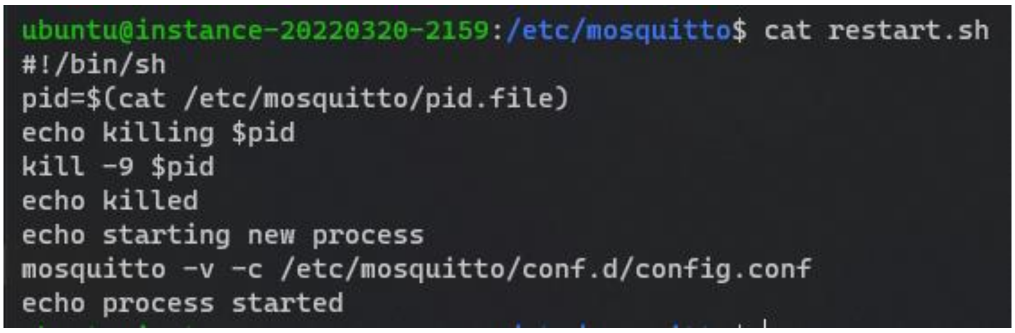


```
andrei@workstation:~/school/certificates/mosquitto-cert$ openssl x509 -req -in server.csr -CA ../ca/ca.crt -CAkey ../ca/ca.key -CAcreateserial -out server.crt -days 365 -extfile ../v3.ext
```

Figure 3-14. Signing server certificate using Extension File

Having generated these certificates, the server part of the setup is done; also, each client that needs MQTT client connection is provisioned with its own key-pair signed by the certification authority.

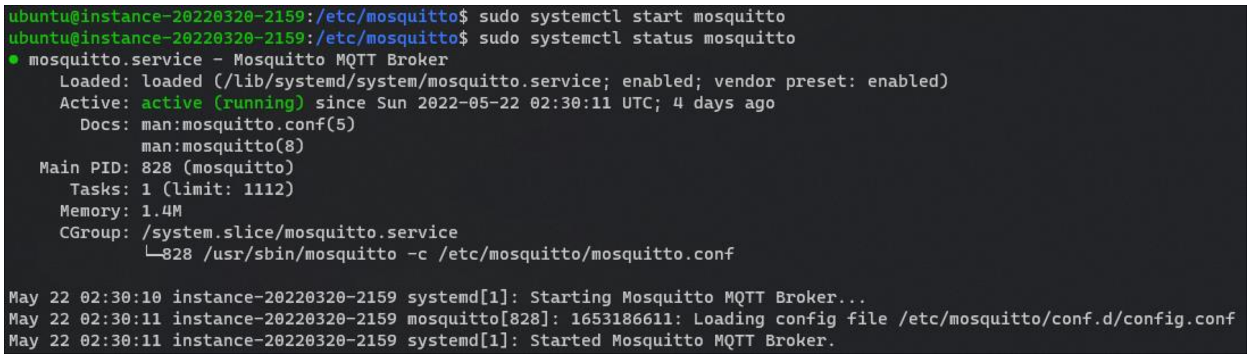
Besides the security options, the *pid_file* option is specified in order to allow the attestation server to restart the broker. This option writes at broker start-up the process ID of Mosquitto. The restart is not done directly, but by using the aforementioned ProcessBuilder that calls the *restart.sh* shell script. This script reads the content of the PID file, kills the process using *SIGKILL* command and starts a new instance of the broker.



```
ubuntu@instance-20220320-2159:/etc/mosquitto$ cat restart.sh
#!/bin/sh
pid=$(cat /etc/mosquitto/pid.file)
echo killing $pid
kill -9 $pid
echo killed
echo starting new process
mosquitto -v -c /etc/mosquitto/conf.d/config.conf
echo process started
```

Figure 3-15. Restart shell script

As it can be seen in the restart shell script, the broker can be started by using *mosquito -v -c <config_file>*, where *v* stands for verbose and *c* points to the configuration file used, or by using *systemctl* if using a Debian based distro.



```
ubuntu@instance-20220320-2159:/etc/mosquitto$ sudo systemctl start mosquitto
ubuntu@instance-20220320-2159:/etc/mosquitto$ sudo systemctl status mosquitto
● mosquitto.service - Mosquitto MQTT Broker
     Loaded: loaded (/lib/systemd/system/mosquitto.service; enabled; vendor preset: enabled)
     Active: active (running) since Sun 2022-05-22 02:30:11 UTC; 4 days ago
       Docs: man:mosquitto.conf(5)
             man:mosquitto(8)
   Main PID: 828 (mosquitto)
      Tasks: 1 (limit: 1112)
     Memory: 1.4M
     CGroup: /system.slice/mosquitto.service
             └─828 /usr/sbin/mosquitto -c /etc/mosquitto/mosquitto.conf

May 22 02:30:10 instance-20220320-2159 systemd[1]: Starting Mosquitto MQTT Broker...
May 22 02:30:11 instance-20220320-2159 mosquitto[828]: 1653186611: Loading config file /etc/mosquitto/conf.d/config.conf
May 22 02:30:11 instance-20220320-2159 systemd[1]: Started Mosquitto MQTT Broker.
```

Figure 3-16. Starting the broker using systemctl

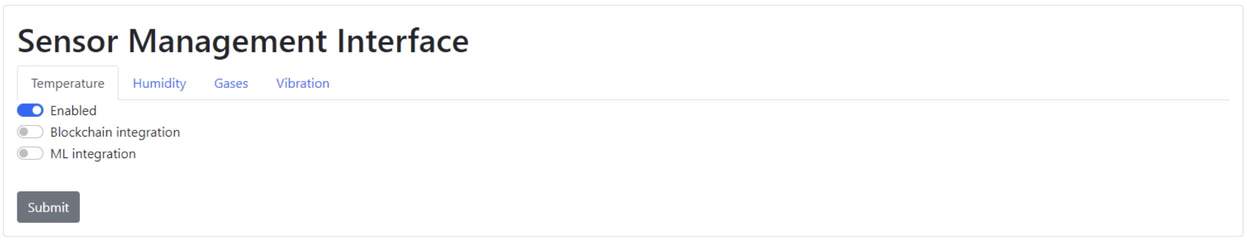
After successfully performing the attestation process, be it either by generating a new key or reusing an existing one that is still valid, the connection to the MQTT broker is established by using the established key. The connection is done using the MQTT library bundled with ESP-IDF which allows an event-driven paradigm to handling communications, and an asynchronous modus operandi leveraging the task-based paradigm of FreeRTOS. The events are submitted to an event loop from which the associated call-backs are called.

Events tackled are:

- *MQTT_EVENT_CONNECTED*: logs a message about successful connection
- *MQTT_EVENT_DISCONNECTED*: triggered when the connection is lost and a retry connection mechanism has been put in place to provide availability and resilience
- *MQTT_EVENT_ERROR*: uses reinterpret_cast to extract the message and logs it
- *MQTT_ERROR_TYPE_CONNECTION_REFUSED*: logs the reason of the refusal

After the connection is established, the sensor is primed and ready to transmit data into the cloud according to the user specified settings. In order to offer a scalable solution, a pluggable architecture was designed. Functionally, this allows the user to choose which data available in the node to transmit to the broker, whether anomaly detection should be performed before sending the data or not and enabling issuing of blockchain transactions directly signed on the node.

These settings can be altered for the management interface hosted by the IoT node. Similarly to the credentials server used when first configuring the node, the static content is served using Arduino library AsyncWebServer which creates a plain HTTP server on port 80 and exposes several endpoints.

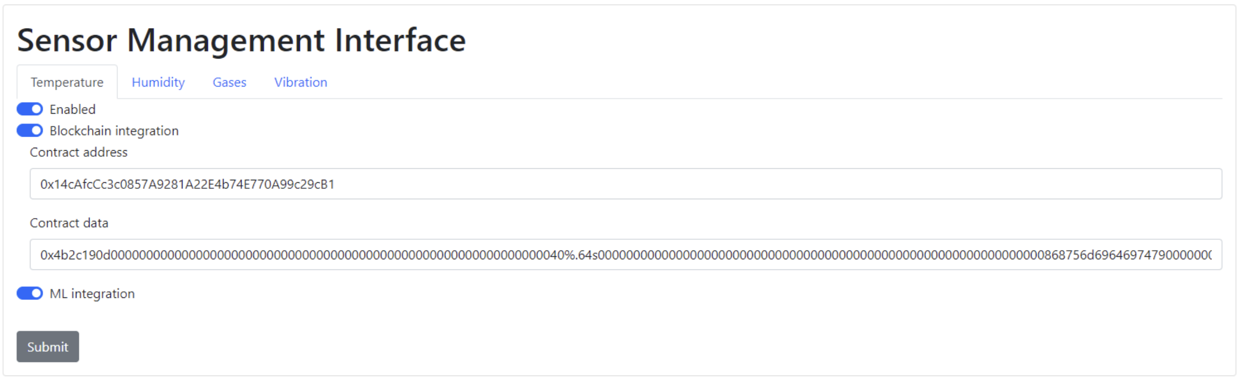


Enabled
Blockchain integration
ML integration

Figure 3-17. Sensor Management Interface

The / endpoint returns the presentation of the page created in HTML5, which allows the user to change settings aided by an easy-to-use graphic interface. For each capability of the node (e.g., temperature), a GET and a POST endpoint are created; the first one allows the display of current configuration in the graphical interface, while the latter one allows the updating of it. In order to persist settings between reboots, a settings file is written into the flash memory of the node using the SPIFFS filesystem. Before starting the management server, the settings are loaded from non-volatile memory and mapped to a *std::map* in memory representation with the key being the name of the capability and the value a structure containing all the related options. The file is structured as a JSON object and parsed using cJSON, which offers great performance even in embedded systems. Each time a setting is updated, the in-memory representation is altered and its content is dumped to the settings file.

Besides enabling and disabling a certain sensor, the user can enable the anomaly detector through ML inference on edge device, or blockchain integration by providing the contract address and the format string the contract payload, or even both.



Enabled
Blockchain integration
Contract address
0x14cAfcCc3c0857A9281A22E4b74E770A99c29cB1
Contract data
0x4b2c190d0000000000000000000000000000000000000000000000000000000000000040%.64s0000000000000000000000000000000000000000000000000000000000000000868756d6964697479000000
ML integration

Figure 3-18. Sensor Management Interface with all options enabled

The very long strings associated with blockchain were took into consideration when designing the system, paying a considerate amount of attention to prevent buffer overflows when copying data from one buffer to another.

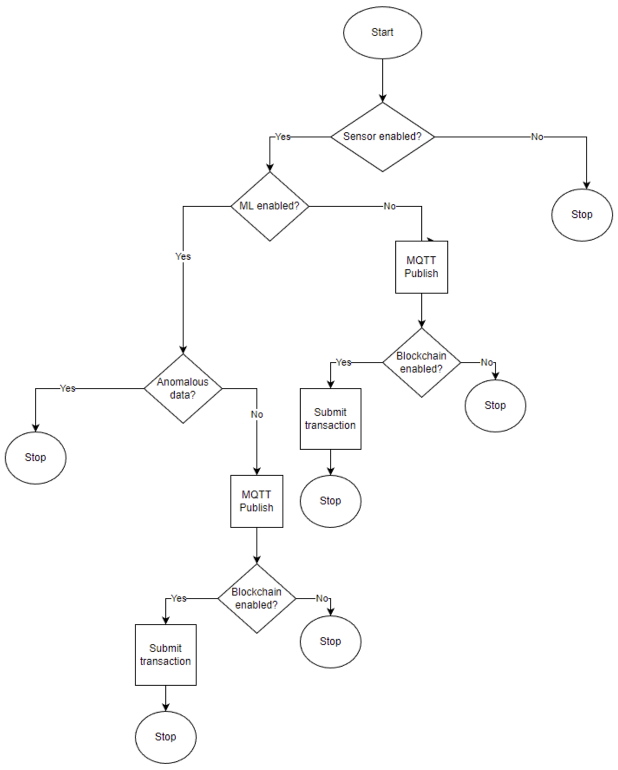
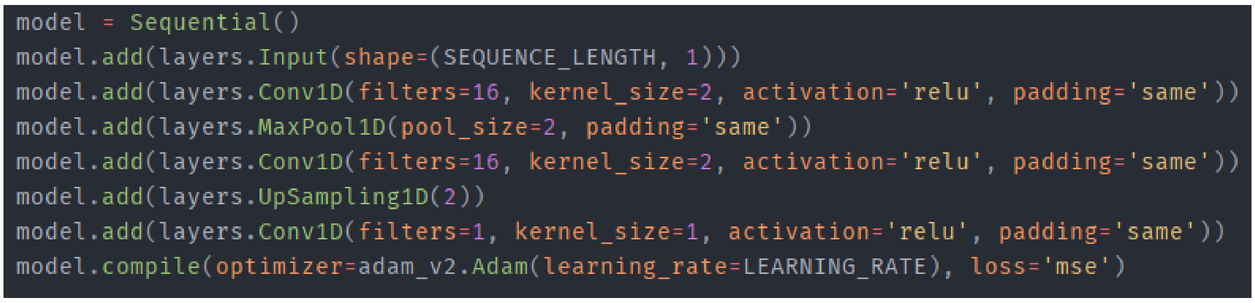


Figure 3-19. Flow of publishing data

Technically, this expandable architecture was achieved by transparently handling the sensor data regardless of their origin, delegating the retrieval of values to a dedicated library residing on the node. After obtaining the useful information, the data is passed through an anomaly detector if ML is enabled by the user, and a delta is calculated between the expected and actual value. If the difference is too high, the data is marked as anomalous and will no longer be sent to the broker.

Anomaly detection is performed by passing the data through a pre-trained neural network, the model being persisted in the non-volatile memory of the node. Even though machine learning often implies high resource consumption, be it processing power or memory footprint, using an optimized model that has quantized weights and the TensorFlow Lite for Microcontrollers platform, edge machine inference has been achieved. The model follows the Auto-Encoder architecture implemented using convolutional layers, which, as the name implies, compresses the data to a latent representation which is then up-sampled to the original size. The network is trained on data that is considered to be in normal working parameters, which makes any anomalous data unable to be reconstructed. Constraints had to be enforced on the size of the model in order to meet size and speed requirements, which led to using a short input sequence of only 4 values and using a relatively small amount of filters for the convolutional layers, 16. Even though time series related regressions tasks are usually associated with recurrent neural networks, Long Short Term Memory (LSTM) layers are currently not supported, and with the inability of the IoT node to run a fully-fledged TensorFlow interpreter, using such a network quickly became out of the realms of current low-cost, low-powered, embedded devices.

The neural network was trained on a machine running Windows 11 featuring a 8 core, 16 thread Ryzen 7 CPU with 16 gigabytes of RAM and a Nvidia GeForce GTX 1070Ti GPU which allowed blazing fast training times thanks to CUDA acceleration. The chosen machine learning platform is TensorFlow 2 using the Keras abstraction layer and written in Python 3 programming language. The usage of such abstraction layers greatly reduces the development time needed, enabling a declarative style of programming by using the Sequential API. The weights are randomly initialized using a normal distribution, which are then adjusted using the process of back propagation using the Adam optimizer, setting the cost function as Mean Squared Error (MSE).



```
model = Sequential()
model.add(layers.Input(shape=(SEQUENCE_LENGTH, 1)))
model.add(layers.Conv1D(filters=16, kernel_size=2, activation='relu', padding='same'))
model.add(layers.MaxPool1D(pool_size=2, padding='same'))
model.add(layers.Conv1D(filters=16, kernel_size=2, activation='relu', padding='same'))
model.add(layers.UpSampling1D(2))
model.add(layers.Conv1D(filters=1, kernel_size=1, activation='relu', padding='same'))
model.compile(optimizer=adam_v2.Adam(learning_rate=LEARNING_RATE), loss='mse')
```

Figure 3-20. Neural Network design

For the training of the model, a dataset was constructed using 500 records that represent normal values. This data was then read from the CSV file containing it and split into training set and testing set in order to prevent the model from overfitting.

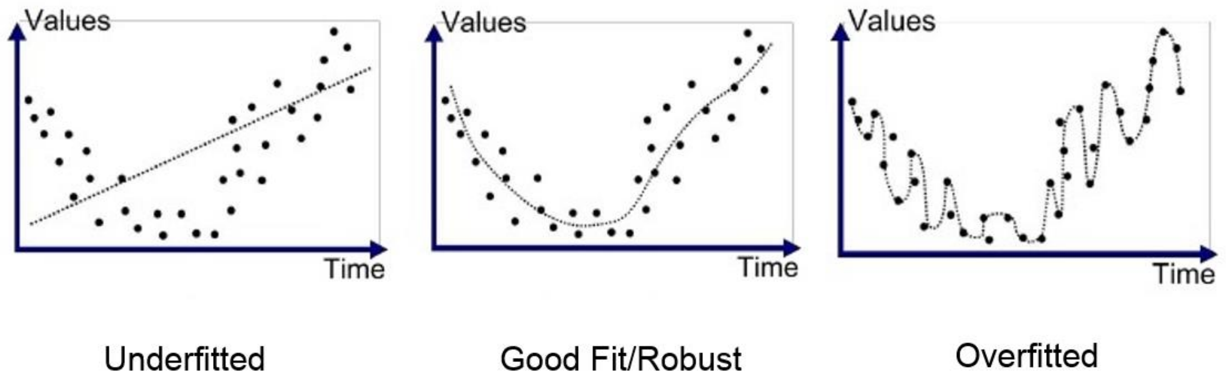
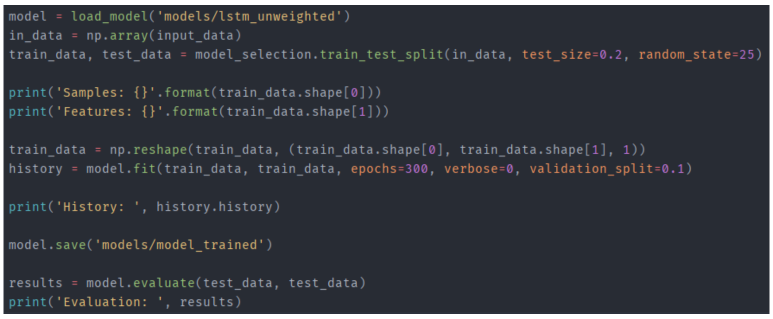
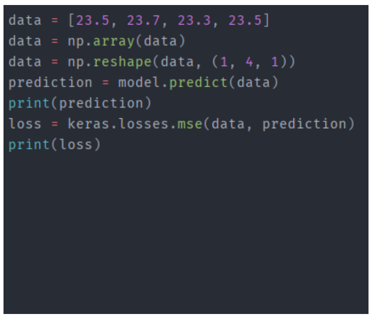


Figure 3-21. Differences between underfit and overfit

The data was then reshaped to the appropriate dimensions expected by the fit method, and the training process ran for 300 epochs. Both the training features and labels are represented by the same dataset, since the Auto-Encoder architecture is characterized by unsupervised learning, of which 10% was used as a validation split. The model results were then evaluated firstly on the test data, and then on a new record.



```python
model = load_model('models/lstm_unweighted')
in_data = np.array(input_data)
train_data, test_data = model_selection.train_test_split(in_data, test_size=0.2, random_state=25)

print('Samples: {}'.format(train_data.shape[0]))
print('Features: {}'.format(train_data.shape[1]))

train_data = np.reshape(train_data, (train_data.shape[0], train_data.shape[1], 1))
history = model.fit(train_data, train_data, epochs=300, verbose=0, validation_split=0.1)

print('History: ', history.history)

model.save('models/model_trained')

results = model.evaluate(test_data, test_data)
print('Evaluation: ', results)
```

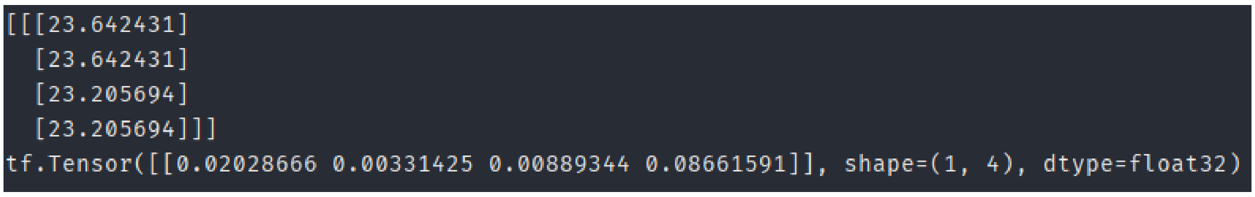


```python
data = [23.5, 23.7, 23.3, 23.5]
data = np.array(data)
data = np.reshape(data, (1, 4, 1))
prediction = model.predict(data)
print(prediction)
loss = keras.losses.mse(data, prediction)
print(loss)
```

Figure 3-22. Training of the model

Figure 3-23. Testing of the model

The network provides a good fit, since prediction based on new information resulted in a lower than 0.1 MSE across the board.

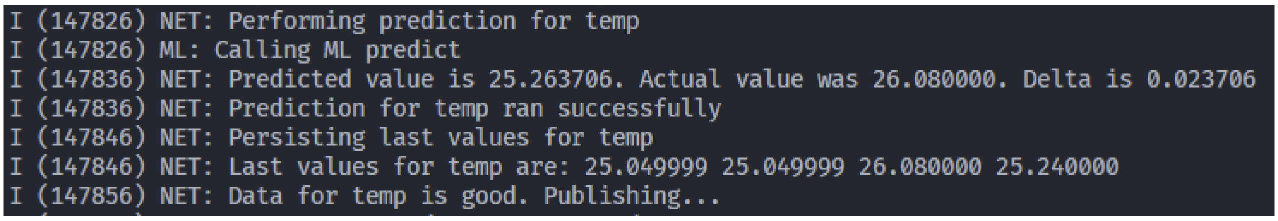


```
[[[23.642431]
  [23.642431]
  [23.205694]
  [23.205694]]]
tf.Tensor([[0.02028666 0.00331425 0.00889344 0.08661591]], shape=(1, 4), dtype=float32)
```

Figure 3-24. Evaluating the model

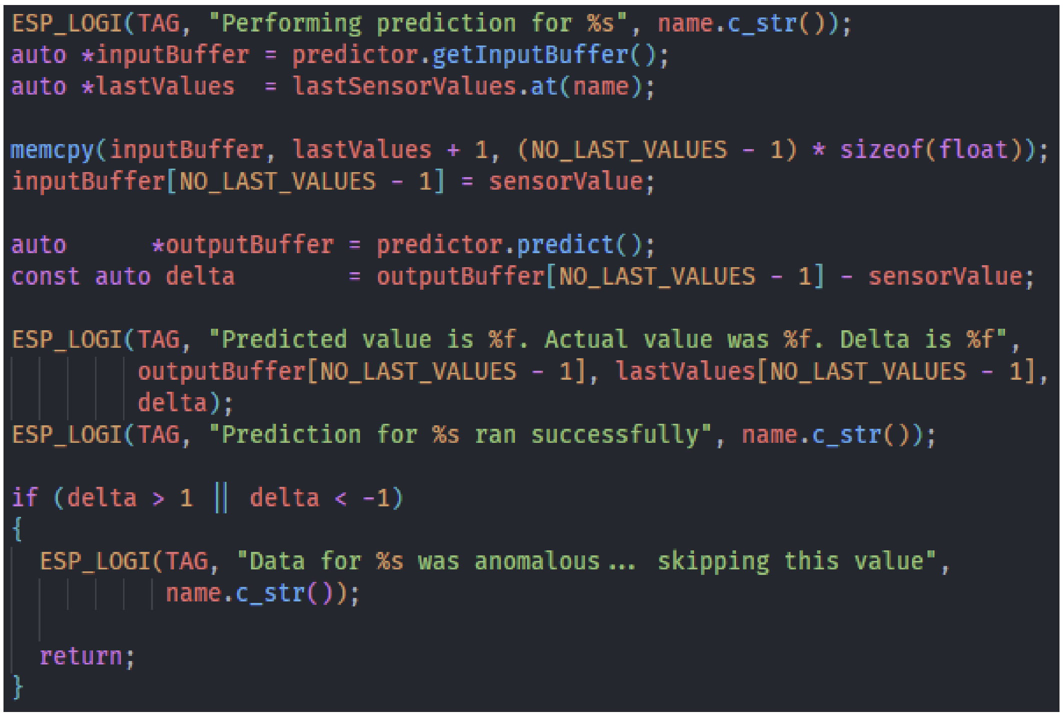
In order to be able to load the model into the IoT node, the trained model has to be converted from TensorFlow to TensorFlow Lite while applying the Default optimization strategy, which reduces size and latency while minimizing the loss in accuracy. The resulted *tflite* file is then converted to a C char array using the Linux *xxd* utility, as such: *xxd -i converted_model.tflite > model_data.cc*, which can then be flashed to the device.

On the node, the usage of TensorFlow Lite has been abstracted into the *MlPredictor* class, which constructs a model instance based on the binary representation of it and exposes methods to set the input tensor and to retrieve the predictions. At instantiation time, the model is deserialized and an interpreter with 4KB is created. For inference, the last three values are retrieved and copied into the input buffer of the interpreter, with the fourth being set as the value to be tested. A delta between the actual and the predicted value is calculated, and if the absolute difference is bigger than 1, the data is marked as anomalous and further actions such as MQTT transmission is halted.



```
I (147826) NET: Performing prediction for temp
I (147826) ML: Calling ML predict
I (147836) NET: Predicted value is 25.263706. Actual value was 26.080000. Delta is 0.023706
I (147836) NET: Prediction for temp ran successfully
I (147846) NET: Persisting last values for temp
I (147846) NET: Last values for temp are: 25.049999 25.049999 26.080000 25.240000
I (147856) NET: Data for temp is good. Publishing...
```

Figure 3-25. Inference results

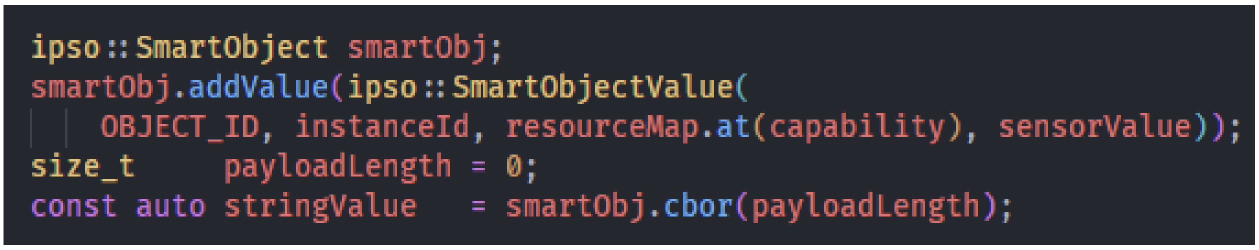


```
ESP_LOGI(TAG, "Performing prediction for %s", name.c_str());
auto *inputBuffer = predictor.getInputBuffer();
auto *lastValues  = lastSensorValues.at(name);

memcpy(inputBuffer, lastValues + 1, (NO_LAST_VALUES - 1) * sizeof(float));
inputBuffer[NO_LAST_VALUES - 1] = sensorValue;

auto       *outputBuffer = predictor.predict();
const auto delta         = outputBuffer[NO_LAST_VALUES - 1] - sensorValue;

ESP_LOGI(TAG, "Predicted value is %f. Actual value was %f. Delta is %f",
         outputBuffer[NO_LAST_VALUES - 1], lastValues[NO_LAST_VALUES - 1],
         delta);
ESP_LOGI(TAG, "Prediction for %s ran successfully", name.c_str());

if (delta > 1 || delta < -1)
{
    ESP_LOGI(TAG, "Data for %s was anomalous... skipping this value",
             name.c_str());

    return;
}
```

Figure 3-26. Business logic running on the node

Otherwise, the usable information is formatted according to the IPSO Smart Object Reference Guide [3] and encoded using CBOR [4]. Since there was no readily available solution for that formatted data according to the IPSO guidelines, a custom library was developed that allows an object-oriented approach to manipulating and packaging sensor data. This library also integrates with the TinyCBOR library develop by Intel in order to provide a seamless operation. The CBOR library is part of standard development kit as shipped by Espressif which made the usage of it effortless.

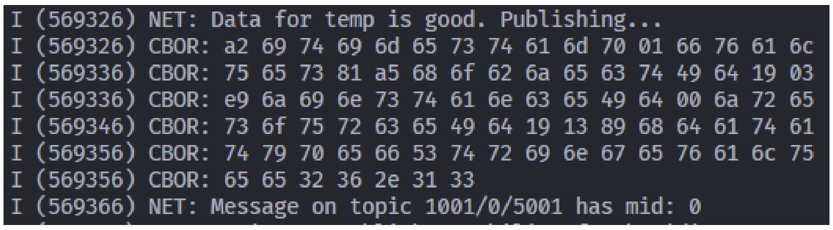
The custom library contains two classes, *SmartObjectValue*, which contains one resource with its value and datatype, uniquely identified by the resource's ID along with ID of the object and the instance on which it was recorded, and *SmartObject*, which contains an array of such values alongside the timestamp the values were sent. As such, the API of this library is quite simple, requiring only the instantiation of a *SmartObject* and then calling the *addValue* method passing a *SmartObjectValue* in order to add a new value to the array. After all the values have been added, the *cbor* method passing a *size_t* variable by reference is called, having a return value of *std::unique_ptr<uint8_t[]>*. The *payloadLength* variable is used since the length of the payload can't be correctly determined using *strlen* because valid CBOR encoded data can contain bytes with value *0x00*, rendering *strlen* unreliable.



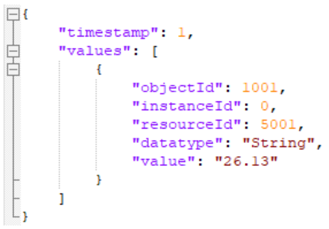
```
ipso::SmartObject smartObj;
smartObj.addValue(ipso::SmartObjectValue(
    OBJECT_ID, instanceId, resourceMap.at(capability), sensorValue));
size_t      payloadLength = 0;
const auto stringValue    = smartObj.cbor(payloadLength);
```

Figure 3-27. Using the SmartObject library

The CBOR encoding was done using TinyCBOR which exposes an easy to use API for encoding and decoding data, and it is used to greatly reduce the payload size. After successfully encoding the IPSO smart object, the encoded bytes are published using MQTT on the topic formatted as such *OBJECT_ID/INSTANCE_ID/RESOURCE_ID*.



```
I (569326) NET: Data for temp is good. Publishing...
I (569326) CBOR: a2 69 74 69 6d 65 73 74 61 6d 70 01 66 76 61 6c
I (569336) CBOR: 75 65 73 81 a5 68 6f 62 6a 65 63 74 49 64 19 03
I (569336) CBOR: e9 6a 69 6e 73 74 61 6e 63 65 49 64 00 6a 72 65
I (569346) CBOR: 73 6f 75 72 63 65 49 64 19 13 89 68 64 61 74 61
I (569356) CBOR: 74 79 70 65 66 53 74 72 69 6e 67 65 76 61 6c 75
I (569356) CBOR: 65 65 32 36 2e 31 33
I (569366) NET: Message on topic 1001/0/5001 has mid: 0
```



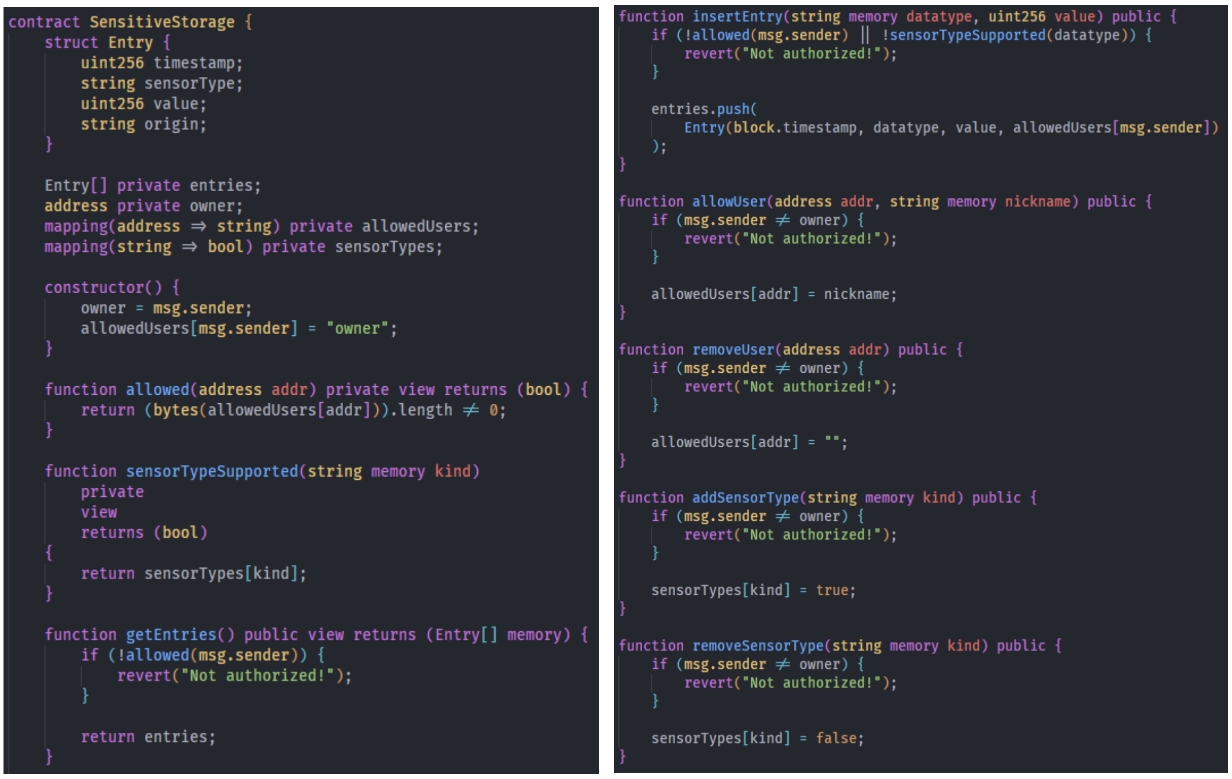
```
{
    "timestamp": 1,
    "values": [
        {
            "objectId": 1001,
            "instanceId": 0,
            "resourceId": 5001,
            "datatype": "String",
            "value": "26.13"
        }
    ]
}
```

Figure 3-28. IPSO Smart Object encoded with CBOR

Figure 3-29. IPSO Smart Object in plaintext

Besides these capabilities, the node is also able to send signed transactions to Ethereum Smart Contracts using Infura Node as a Service provider. Delegating this responsibility to Infura allows the node to only concern about formatting the contract data and signing the transaction in the seckp256k1 curve. The function call is encoded according to Solidity's Contract ABI Specification [22] and is expected to be provided by the user in the settings portal. The sensor data will then be inserted into the format string using *sprintf* and sent to the blockchain.

For this application, a secure storage contract was developed and deployed to the Ropsten Test Network, which has authorization capabilities and exposes several methods.



```
contract SensitiveStorage {
    struct Entry {
        uint256 timestamp;
        string sensorType;
        uint256 value;
        string origin;
    }

    Entry[] private entries;
    address private owner;
    mapping(address ⇒ string) private allowedUsers;
    mapping(string ⇒ bool) private sensorTypes;

    constructor() {
        owner = msg.sender;
        allowedUsers[msg.sender] = "owner";
    }

    function allowed(address addr) private view returns (bool) {
        return (bytes(allowedUsers[addr])).length ≠ 0;
    }

    function sensorTypeSupported(string memory kind)
        private
        view
        returns (bool)
    {
        return sensorTypes[kind];
    }

    function getEntries() public view returns (Entry[] memory) {
        if (!allowed(msg.sender)) {
            revert("Not authorized!");
        }

        return entries;
    }
```

```
function insertEntry(string memory datatype, uint256 value) public {
    if (!allowed(msg.sender) || !sensorTypeSupported(datatype)) {
        revert("Not authorized!");
    }

    entries.push(
        Entry(block.timestamp, datatype, value, allowedUsers[msg.sender])
    );
}

function allowUser(address addr, string memory nickname) public {
    if (msg.sender ≠ owner) {
        revert("Not authorized!");
    }

    allowedUsers[addr] = nickname;
}

function removeUser(address addr) public {
    if (msg.sender ≠ owner) {
        revert("Not authorized!");
    }

    allowedUsers[addr] = "";
}

function addSensorType(string memory kind) public {
    if (msg.sender ≠ owner) {
        revert("Not authorized!");
    }

    sensorTypes[kind] = true;
}

function removeSensorType(string memory kind) public {
    if (msg.sender ≠ owner) {
        revert("Not authorized!");
    }

    sensorTypes[kind] = false;
}
```

Figure 3-30. Ethereum Smart Contract

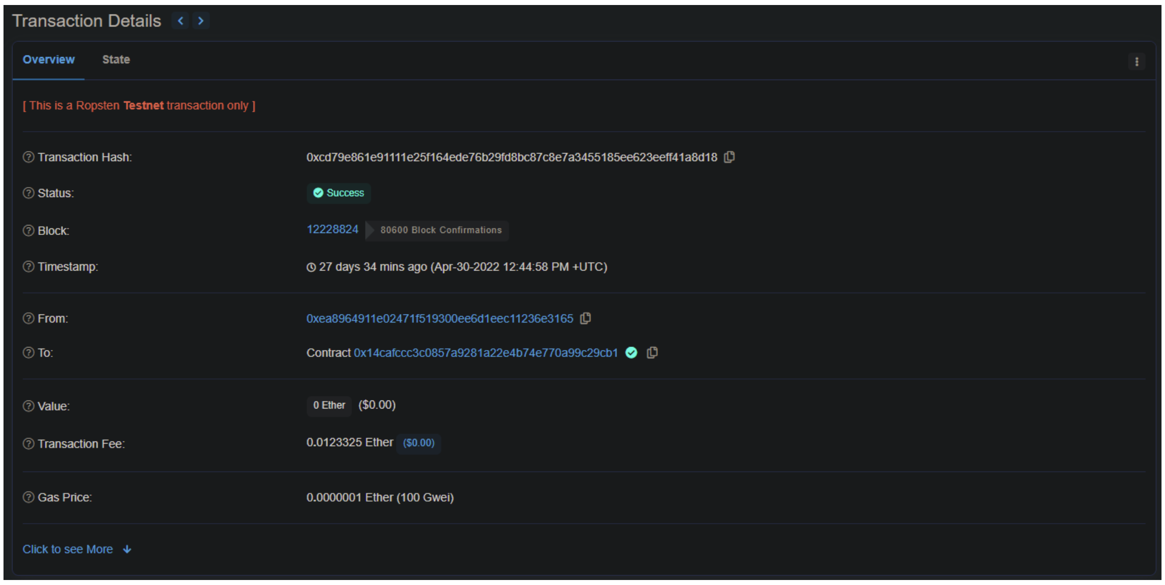
The constructor is called at the moment the contract is created, binding the creator of the owner member to the message sender. Besides that, in order to implement authorization, a map of strings identified by addresses is created. According to Solidity's documentation [23], mappings can be seen as hash tables which are virtually initialized such that every possible key exists and is mapped to a value whose byte-representation is always zero. Having this in mind, a helper function *allowed* was created in order to avoid duplicate code, which checks if a certain address is authorized to perform operations on the contract. Also, only the owner of the contract is allowed to add or remove users to the authorized list. For validation purposes, a similar mechanism was implemented for the sensor types, only allowing types to be created by the owner with the ability to delete unused ones. Besides these management functions, *insertEntry* allows authorized parties to insert records for supported sensor types, storing them according to the *Entry* structure. This structure features the timestamp the record was inserted at, the type of sensors and its value along with the party which requested the insertion. There is also a *getEntries* method which allows authorized users to retrieve all previously saved records.

In this particular case, the function that is called by the node is *insertEntry*, the node being provisioned with an Ethereum wallet, from which the private key is used in order to sign outgoing transactions. To submit such transactions, the Arduino library *Web3* was used in order to speed up development, which only needed little adaptation to work under ESP-IDF with the ESP32. This library supports the latest version of the Infura API, v3, where the Infura host and path are provided as parameters to the constructor of the object used to issue transactions, *Web3*. Besides the aforementioned private key, settings such as maximum gas price value, gas limit, and destination of the transaction are inputted by the user. The payload of the contract is also set by the user as a format string, which is then expanded by the node with the actual value of the sensor using *sprintf* formatting. This payload is formatted according to the specification, where the first 4 bytes are keccak256 digest of the function signature, 32 bytes for each function parameters, where primitives are put in place and dynamic types are identified by the memory location where it can be found, followed by values of dynamic parameters.

```
0x4b2c190d                                                       -> keccak256(insertEntry(string,uint256))
0000000000000000000000000000000000000000000000000000000000000040 -> location of data for 1st argument,   datatype: string
0000000000000000000000000000000000000000000000000000000000000038 -> 56 in hex, value of second argument, value: uint256
0000000000000000000000000000000000000000000000000000000000000008 -> length of first argument, which is a string
68756d6964697479000000000000000000000000000000000000000000000000 -> "humidity" as hex
```

Figure 3-31. Solidity Function Encoding

All these parameters are then used to issue a transaction using Web3's *sendTransaction* method, which transparently handles the signing and sending the request to Infura, which returns the hash of the transaction. This hash can be used on Etherscan to check the status of the corresponding transaction.



[This is a Ropsten **Testnet** transaction only]

Transaction Hash: 0xcd79e861e91111e25f164ede76b29fd8bc87c8e7a3455185ee623eeff41a8d18
Status: Success
Block: 12228824 80600 Block Confirmations
Timestamp: 27 days 34 mins ago (Apr-30-2022 12:44:58 PM +UTC)
From: 0xea8964911e02471f519300ee6d1eec11236e3165
To: Contract 0x14cafccc3c0857a9281a22e4b74e770a99c29cb1
Value: 0 Ether ($0.00)
Transaction Fee: 0.0123325 Ether ($0.00)
Gas Price: 0.0000001 Ether (100 Gwei)

Figure 3-32. Successful transaction checked on Etherscan

Besides the IoT node and the aforementioned Attestation Server, several other Spring Boot microservices were developed in a cloud-native manner. As such, loose coupling is employed throughout the solution, employing technologies such as Service Discovery, or the usage of config servers. All the microservices except the Attestation Server are meant to be run in containers, each of them having a corresponding *Dockerfile*, and the lot orchestrated by Docker Compose. The Attestation Server is designed to be run on the host system since it has tight coupling with the

Mosquitto Broker. The usage of containerization provides increased reliability through automatic restart and can handle fluctuating load levels by scaling up or down the number of replicas needed of each service, achieved by packaging the microservice in a JAR using Maven and then creating an image based on the respective Dockerfile. In order to ease orchestration, a shell script is used where necessary. It pings the container that is dependent of and blocks the execution until the provisioning of it is done.

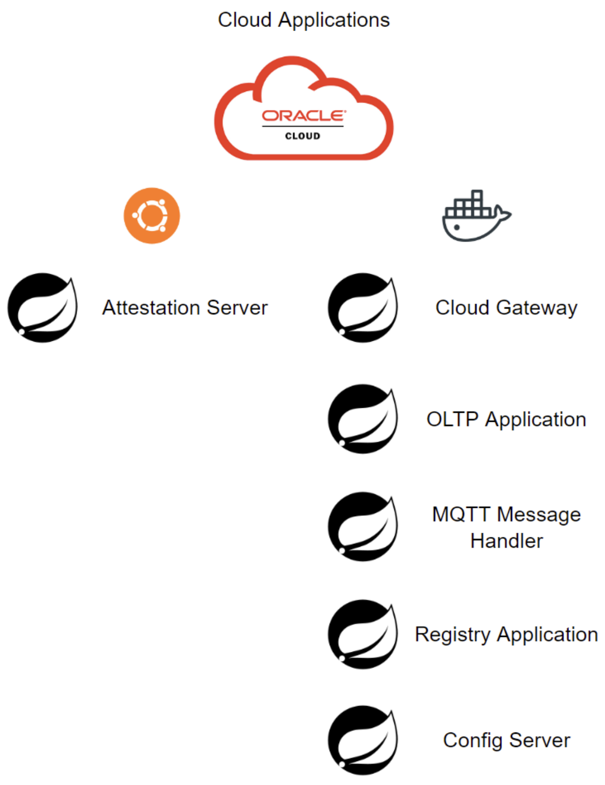


Figure 3-33. Overview of microservices developed

Both the Config Server and the Registry Application server were used to support the infrastructure of the solution.

The Config Server is developed using Spring Boot and the Cloud Config Server starter, which deals with all the implementation details transparently, requiring only the URL of the Git repository from which the configuration files should be resolved at run-time. These are resolved by checking the master branch of the repository for files matching the *<APP-NAME>_<profile>.yml* files. In order to avoid port binding issues, the default Tomcat port has been changed to 8800.

The Registry Application uses the Netflix Eureka Server starter in order to allow service discovery among microservices. Among this starter, the Cloud Config one is used as a client in order to resolve the configuration properties at run-time. Binding issues are avoided by using the 8761 port, and self-registration is avoided by setting the *eureka.client.register-with-eureka* property to false.

Another mentionable aspect that is infrastructure related is the use of Zipkin distributed tracing system, that allows easy visualization and debugging of latency issues, where each microservice involved in resolving requests has both the Spring Cloud Sleuth abstraction and the Zipkin dependencies added and configured. The Zipkin server is also containerized, using the *openzipkin/zipkin* image.

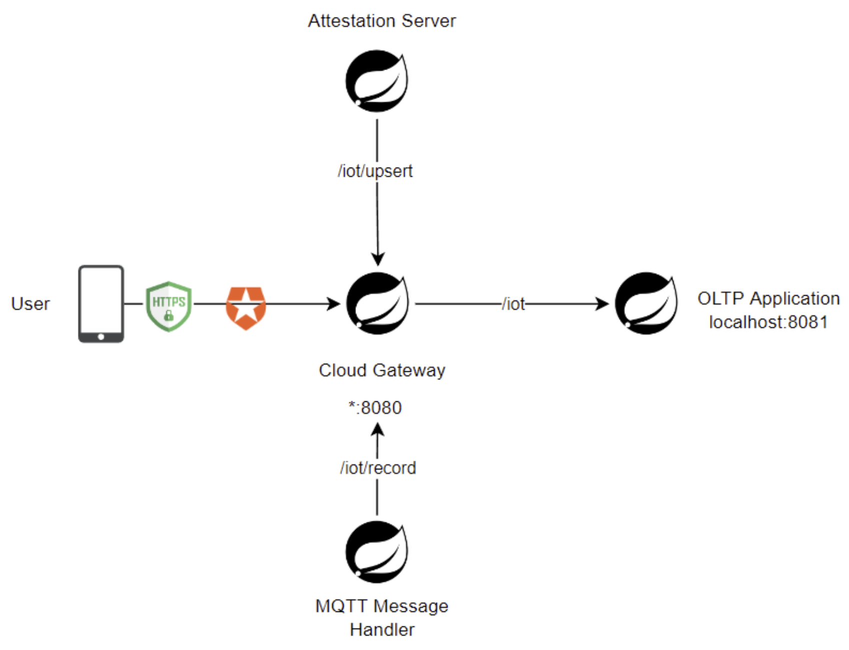


Figure 3-34. Solution flow

The web-facing component of the solution is represented by the Cloud Gateway, developed in Spring Boot using the Cloud Gateway starter. This library provides great integration with Service Discovery platforms like Eureka, having the ability to resolve paths at runtime in a load-balanced approach, and with Circuit Breaker libraries such as Resilience4j. Circuit Breaker is a design pattern that is used when a proxy like a Gateway is used in order to avoid resource exhaustion in the case of a microservice that is down; the principle is similar to that of an electrical circuit breaker, tripping it after a certain number of failures and refusing all requests for a limited amount of time. After this timeout expires, a test period is put in place when the circuit breaker is faster to close if another failure occurs. Resilience4j provides the implementation of such a pattern in Java, allowing the definition of circuit breakers for each route handled. In this case, only one route is defined, the one that forwards the request to the OLTP application if the prefix path is */iot* and resolves the path of the service at run-time using the Eureka client. To improve performance, caching is employed using the Spring abstraction layer and the Caffeine implementation, injecting a *CacheManager* into the context at initialization time. Besides the routing purpose, the gateway also authenticates and authorizes all requests using JSON Web Tokens with the help of OAuth 2.0 Resource Server Starter, that transparently creates a bean into the context using properties from the configuration file such as the Issuer URI, where the configuration file is resolved at run-time using the Config Server. This service binds to the default port 8080.

The OLTP application serves the purpose of persisting information such as node details, available objects, instances, resources and historic records. A MySQL database is used for this together with the Spring Data JPA starter to reduce the amount of boiler-plate code needed to access the data and speed up the development time. Entities are marked by the *@Entity* annotation where their members represent table fields which can have constraints placed on them, e.g., primary key or foreign key. The library follows a domain driven design, data being accessed through the help of repositories that are implemented by extending interfaces such as *CrudRepository* or *PagingAndSortingRepository* which can have custom methods defined using a domain specific language (DSL). Also, transaction support is enabled in order to improve the resiliency of the solution.

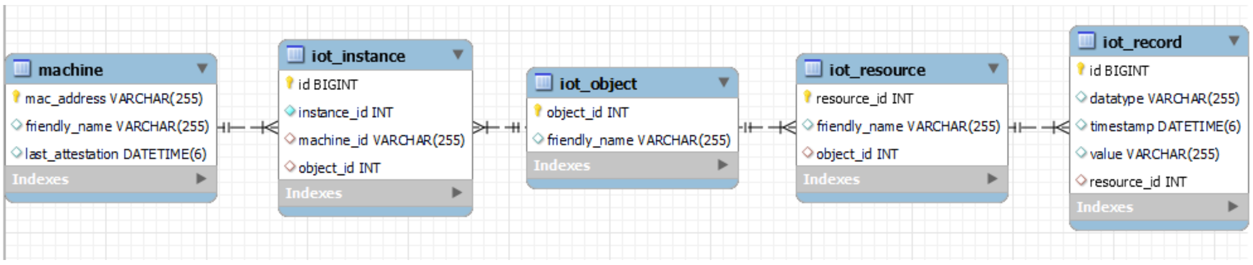
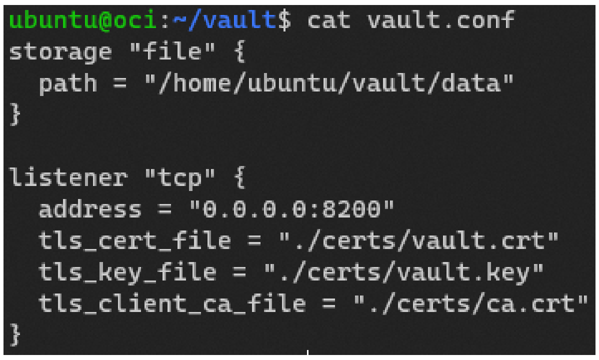


Figure 3-35. Database schema

The MySQL Database is hosted in Docker container, where Hibernate's ability to generate creation scripts is leveraged in order to have the instance run such a script at first boot. This is achieved by mounting the local directory containing the script to */docker-entrypoint-initdb.d*.

Since the password of the database is a secret that must be protected, using only the config server is not enough since the credentials can either be leaked or even sniffed if sent over an unprotected protocol, e.g., HTTP. For this reason, a secret management tool has been employed, HashiCorp Vault, which is used to control access to sensitive data in low-trust mediums.

The Vault server has been setup on the Oracle Cloud instance, running on port 8200 using HTTPS in order to prevent sniffing of secrets. As such, a new key pair was generated and signed by the CA, placing both the private and public key on the server. For this proof of concept, the *file* storage backend was used in order to provide persistency among reboots.

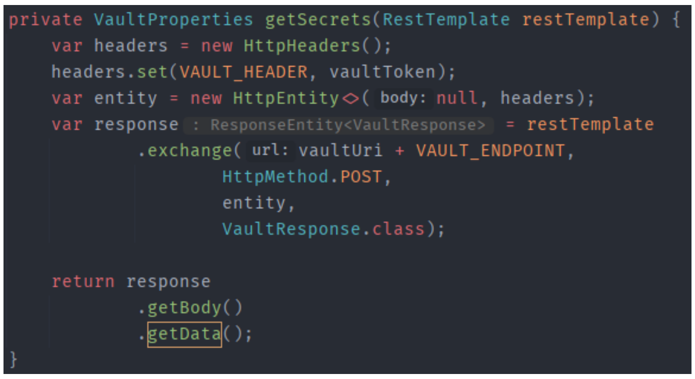


```
ubuntu@oci:~/vault$ cat vault.conf
storage "file" {
  path = "/home/ubuntu/vault/data"
}

listener "tcp" {
  address = "0.0.0.0:8200"
  tls_cert_file = "./certs/vault.crt"
  tls_key_file = "./certs/vault.key"
  tls_client_ca_file = "./certs/ca.crt"
}
```

Figure 3-36. Vault configuration file

After initialization of the server, the Key-Value (kv) secrets engine is enabled at path *secret*, postfixing it with the name of the application used. Even with all these mitigation steps in place, the solution would be as vulnerable as ever without any further actions due to the Secret Zero problem. This is when all the secrets are protected by another secret, in this case the access token that grants read access of such confidential information. Thankfully, Vault has a mechanism in place that solves this issue, Cubbyhole Response Wrapping. This uses the cubbyhole secrets engine to wrap the actual access token into another short-lived single-use token. With proper alert mechanisms in place to detect if such a token is compromised, e.g., the service won't start because the token has already been used means that the token was sniffed and all secrets at that endpoint must be changed, this deems an extremely secure solution to handling secrets since even the operator that generates them doesn't have access to the underlying access token.

This secret management solution is used throughout the application. The Attestation Server uses it to obtain the client secret needed for acquiring the Bearer Token during the client credentials flow of Auth0. The MQTT Message Handler uses it for the same purpose as the Attestation Server, and the OLTP application requires it for storing the database credentials securely. The secrets are retrieved by performing an HTTPS POST to the vault server, placing the wrapping token in the *X-Vault-Token* header and sending no request body. The wrapping token is inputted by an operator who issued it with the following command: *vault kv get -wrap-ttl=120 secret/<app_name>*.

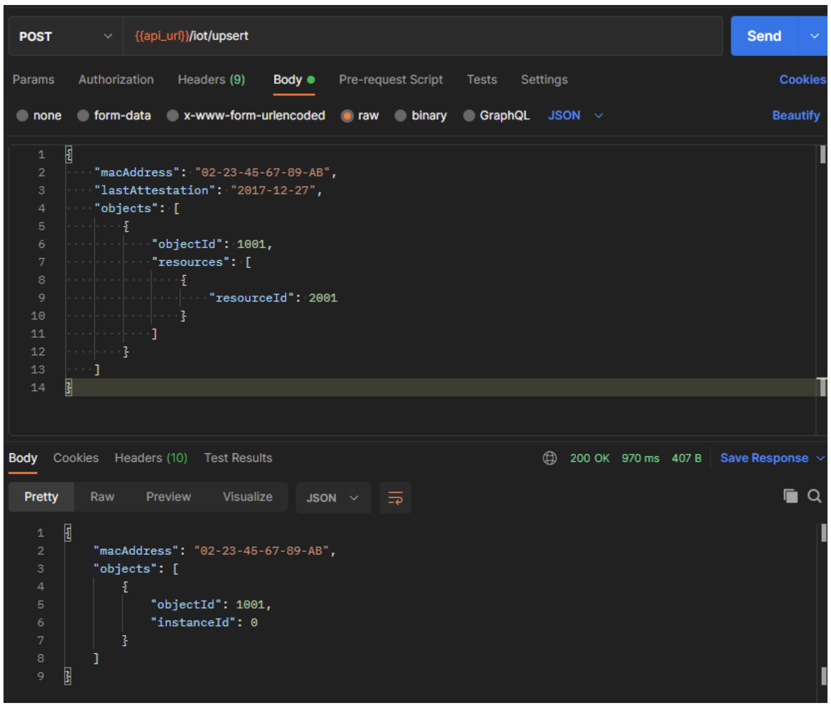


```
private VaultProperties getSecrets(RestTemplate restTemplate) {
    var headers = new HttpHeaders();
    headers.set(VAULT_HEADER, vaultToken);
    var entity = new HttpEntity<>( body: null, headers);
    var response : ResponseEntity<VaultResponse> = restTemplate
            .exchange( url: vaultUri + VAULT_ENDPOINT,
                    HttpMethod.POST,
                    entity,
                    VaultResponse.class);

    return response
            .getBody()
            .getData();
}
```

Figure 3-37. Retrieving the secrets stored in HashiCorp Vault

The microservice is following the REST architectural style, using the Web Starter and adhering to the *Separation of Concerns* principle by splitting the business logic into controllers, services and repositories. Besides that, performance improvements were achieved by using the Spring Boot Cache abstraction layer with the Caffeine implementation. Service registration was done using the Netflix Eureka client, which allows the gateway to resolve the path of the service at runtime. Tight security was employed, authenticating, and authorizing all requests using JWT with the help of the OAuth 2.0 Resource Server Starter.

A rather simple API is exposed, starting with the upsertion of a IoT node at the */iot/upsert* POST endpoint by the Attestation Server following a successful attestation process. The body of such request is of type *MachineRequestDto* composed of the device identifier along with the current date and time in the *lastAttestation* field. At the implementation level, an existence check for such a machine is put in place in order not to lose the friendly name set by the user; if such a machine exists, only the last attestation date is updated, otherwise a new record is inserted. Following this, all the objects exposed by the machine are iterated alongside their resources to be saved or updated, in which step the instance ID is generated. On successive runs, the same instance ID is guaranteed for the same machine.



POST {{api_url}}/iot/upsert

```
{
    "macAddress": "02-23-45-67-89-AB",
    "lastAttestation": "2017-12-27",
    "objects": [
        {
            "objectId": 1001,
            "resources": [
                {
                    "resourceId": 2001
                }
            ]
        }
    ]
}
```

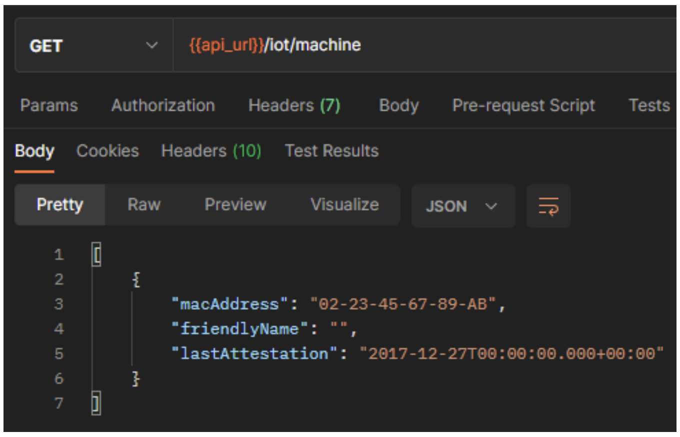
200 OK 970 ms 407 B

```
{
    "macAddress": "02-23-45-67-89-AB",
    "objects": [
        {
            "objectId": 1001,
            "instanceId": 0
        }
    ]
}
```

Figure 3-38. Upsertion of the machine

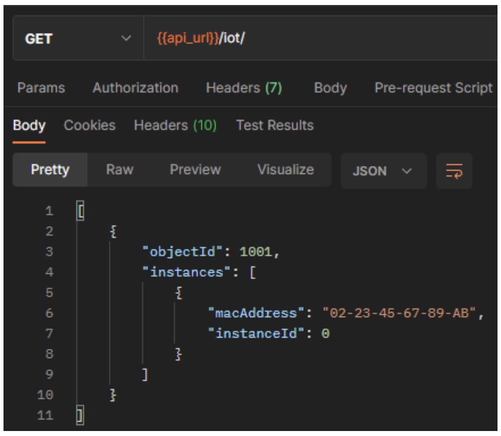
Following a successful upsertion, several GET endpoints can be called to retrieve information about the machines, all the objects and their instances, the resources of an object or records of such resource.

Information about the machines can be retrieved at the */iot/machine* GET endpoint, while information about all the objects can be retrieved at the */iot* GET endpoint.



```
[
    {
        "macAddress": "02-23-45-67-89-AB",
        "friendlyName": "",
        "lastAttestation": "2017-12-27T00:00:00.000+00:00"
    }
]
```

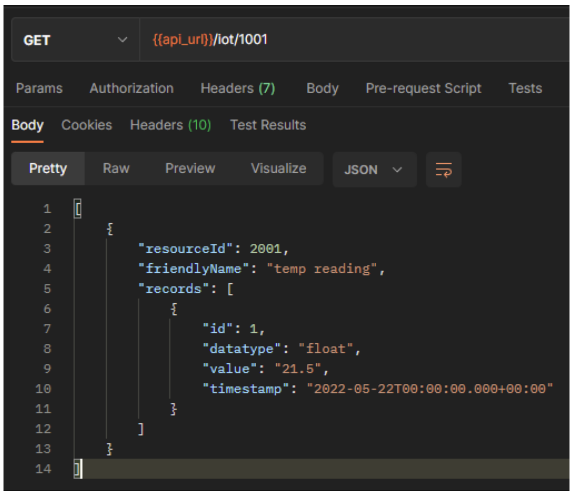
Figure 3-39. Retrieving information about the machines



```
[
    {
        "objectId": 1001,
        "instances": [
            {
                "macAddress": "02-23-45-67-89-AB",
                "instanceId": 0
            }
        ]
    }
]
```

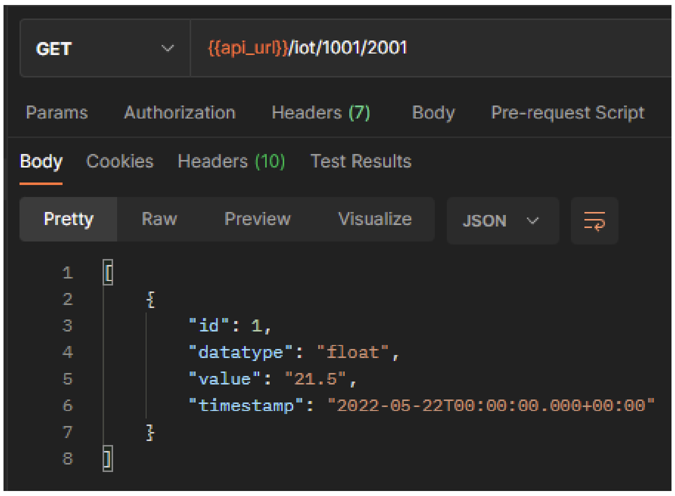
Figure 3-40. Retrieving information about the available objects

Available resources and the last 50 records of it can be queried at the */iot/{objectId}* endpoint, while all the records can be displayed by calling the */iot/{objectId}/{resourceId}* endpoint.



```
[
    {
        "resourceId": 2001,
        "friendlyName": "temp reading",
        "records": [
            {
                "id": 1,
                "datatype": "float",
                "value": "21.5",
                "timestamp": "2022-05-22T00:00:00.000+00:00"
            }
        ]
    }
]
```

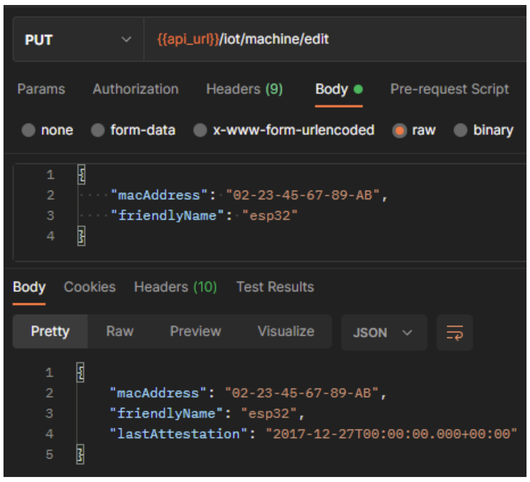
Figure 3-41. Getting available resources



```
[
    {
        "id": 1,
        "datatype": "float",
        "value": "21.5",
        "timestamp": "2022-05-22T00:00:00.000+00:00"
    }
]
```

Figure 3-42. Retrieving records

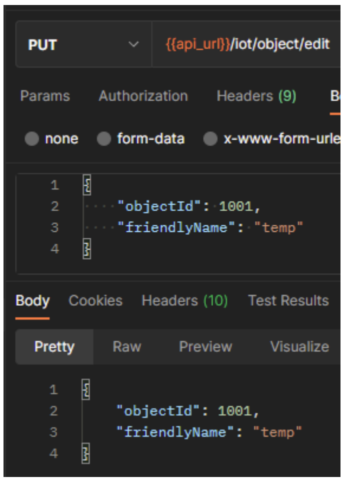
Besides the GET endpoints, three more PUT endpoints are available that allow using more friendly names for machines, resources, or objects. Editing a machine's friendly name can be done by calling the */iot/machine/edit* endpoint, setting the request body to contain the MAC address of the IoT node and the friendly name to be set. In order to edit an object's friendly name, the */iot/object/edit* endpoint can be called using the object ID and the new friendly name in the body of the request. The friendly name of a resource can be altered by calling */iot/resource/edit*, placing the resource ID and the new name in the request body.



```
{
    "macAddress": "02-23-45-67-89-AB",
    "friendlyName": "esp32"
}
```

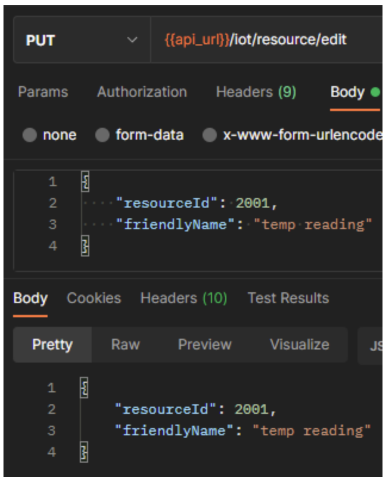
```
{
    "macAddress": "02-23-45-67-89-AB",
    "friendlyName": "esp32",
    "lastAttestation": "2017-12-27T00:00:00.000+00:00"
}
```

Figure 3-43. Editing the friendly name of a machine



```
{
    "objectId": 1001,
    "friendlyName": "temp"
}
```

```
{
    "objectId": 1001,
    "friendlyName": "temp"
}
```

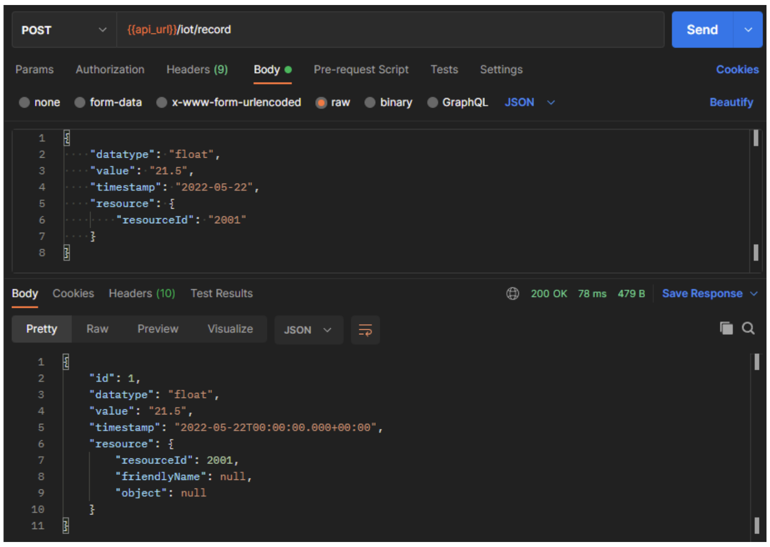


```
{
    "resourceId": 2001,
    "friendlyName": "temp reading"
}
```

```
{
    "resourceId": 2001,
    "friendlyName": "temp reading"
}
```

Figure 3-44. Editing friendly name of an object *Figure 3-45. Editing friendly name of a resource*

Records can be saved by calling the */iot/record* POST endpoint, using a request body that adheres to the IPSO Smart Object guidelines. As such, a record must specify the resource ID for which it was submitted, alongside the datatype, value, and the timestamp.



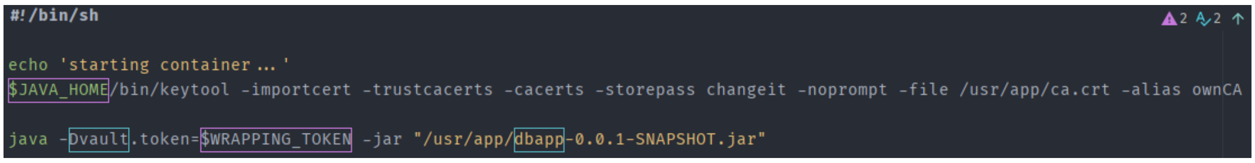
```
{
    "datatype": "float",
    "value": "21.5",
    "timestamp": "2022-05-22",
    "resource": {
        "resourceId": "2001"
    }
}
```

200 OK 78 ms 479 B

```
{
    "id": 1,
    "datatype": "float",
    "value": "21.5",
    "timestamp": "2022-05-22T00:00:00.000+00:00",
    "resource": {
        "resourceId": 2001,
        "friendlyName": null,
        "object": null
    }
}
```

Figure 3-46. Saving a record

Containerizing this service involved an extra step since the Vault's certificate is signed by a custom CA. By default, validation of the server's certificate would fail since the custom certificate wouldn't have been recognized, thus requiring the import of it before starting the application. This is done using a shell script instead of running the Java interpreter directly, calling *keytool* with required parameters first and then using the required start-up command.

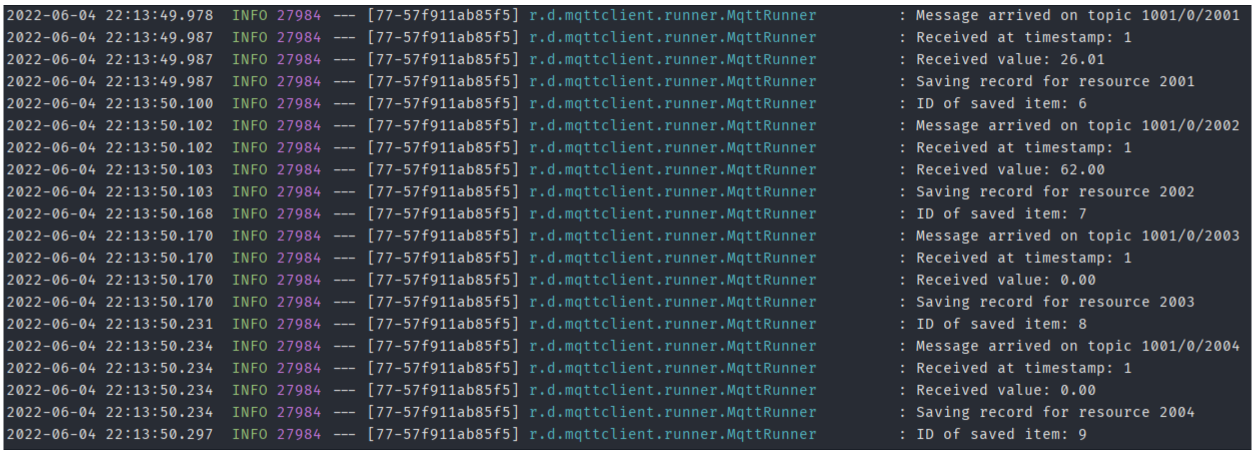


```
#!/bin/sh

echo 'starting container ... '
$JAVA_HOME/bin/keytool -importcert -trustcacerts -cacerts -storepass changeit -noprompt -file /usr/app/ca.crt -alias ownCA

java -Dvault.token=$WRAPPING_TOKEN -jar "/usr/app/dbapp-0.0.1-SNAPSHOT.jar"
```

Figure 3-47. OLTP application start-up shell script

Saving a record by hand would be counter-intuitive since the data is already published to the MQTT broker, and for that a Spring Boot microservice was developed, the MQTT Message Handler. This service uses the Paho MQTT Client originating from the Eclipse Foundation to retrieve published records, subscribing to all available topics by using the # wildcard. The listener on port *8804* is used, which enforces mutual SSL authentication, each party having its own key pair signed by the CA. Since the CA is a self-signed certificate, a custom *SSLSocketFactory* had to be implemented using the Bouncy Castle JCA Provider to pass the certificate check. The MQTT 3.1 protocol version is used, setting a Keep-Alive and Connection-Timeout values of 60 seconds each, injecting the client into the context after successful connection. Since this service publishes data to the Gateway, the Auth0 Client Credentials flow is used, securing the client secret in the HashiCorp Vault. The required configuration properties are served at runtime by the Spring Cloud Config Server. As such, an HTTPS POST request to the vault server is first performed to obtain the client secret, which is then used in a POST request to the issuer to obtain the bearer token which is then injected into the context of the application. In addition, the *jackson-dataformat-cbor* dependency was used which allows decoding of the payload from CBOR to ASCII.

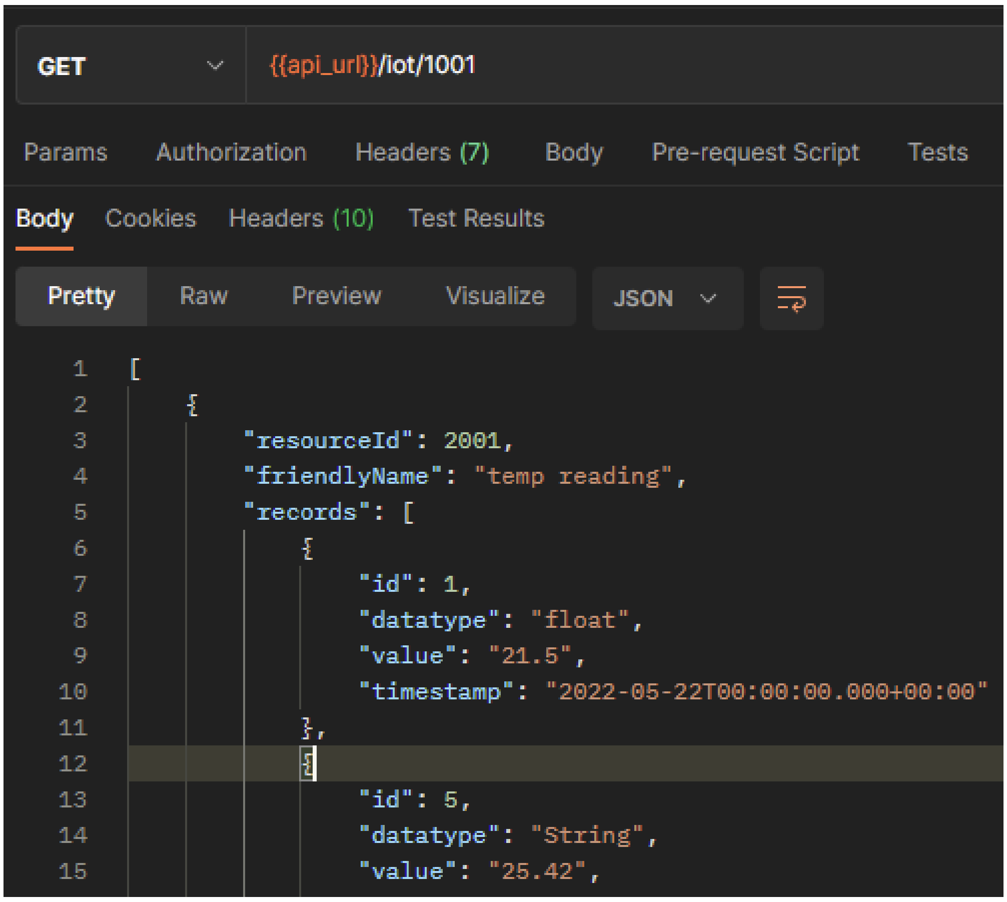
A class that implements *CommandLineRunner* was created, *MqttRunner*, which receives among others the MQTT client which is resolved from the application context. The call-backs of the client are set, logging simple messages in case of lost connection or completion of delivery, or processing any incoming message. The *sleep* method of the *Thread* class is then called in a loop in order to prevent the service from stopping.



```
2022-06-04 22:13:49.978  INFO 27984 --- [77-57f911ab85f5] r.d.mqttclient.runner.MqttRunner         : Message arrived on topic 1001/0/2001
2022-06-04 22:13:49.987  INFO 27984 --- [77-57f911ab85f5] r.d.mqttclient.runner.MqttRunner         : Received at timestamp: 1
2022-06-04 22:13:49.987  INFO 27984 --- [77-57f911ab85f5] r.d.mqttclient.runner.MqttRunner         : Received value: 26.01
2022-06-04 22:13:49.987  INFO 27984 --- [77-57f911ab85f5] r.d.mqttclient.runner.MqttRunner         : Saving record for resource 2001
2022-06-04 22:13:50.100  INFO 27984 --- [77-57f911ab85f5] r.d.mqttclient.runner.MqttRunner         : ID of saved item: 6
2022-06-04 22:13:50.102  INFO 27984 --- [77-57f911ab85f5] r.d.mqttclient.runner.MqttRunner         : Message arrived on topic 1001/0/2002
2022-06-04 22:13:50.102  INFO 27984 --- [77-57f911ab85f5] r.d.mqttclient.runner.MqttRunner         : Received at timestamp: 1
2022-06-04 22:13:50.103  INFO 27984 --- [77-57f911ab85f5] r.d.mqttclient.runner.MqttRunner         : Received value: 62.00
2022-06-04 22:13:50.103  INFO 27984 --- [77-57f911ab85f5] r.d.mqttclient.runner.MqttRunner         : Saving record for resource 2002
2022-06-04 22:13:50.168  INFO 27984 --- [77-57f911ab85f5] r.d.mqttclient.runner.MqttRunner         : ID of saved item: 7
2022-06-04 22:13:50.170  INFO 27984 --- [77-57f911ab85f5] r.d.mqttclient.runner.MqttRunner         : Message arrived on topic 1001/0/2003
2022-06-04 22:13:50.170  INFO 27984 --- [77-57f911ab85f5] r.d.mqttclient.runner.MqttRunner         : Received at timestamp: 1
2022-06-04 22:13:50.170  INFO 27984 --- [77-57f911ab85f5] r.d.mqttclient.runner.MqttRunner         : Received value: 0.00
2022-06-04 22:13:50.170  INFO 27984 --- [77-57f911ab85f5] r.d.mqttclient.runner.MqttRunner         : Saving record for resource 2003
2022-06-04 22:13:50.231  INFO 27984 --- [77-57f911ab85f5] r.d.mqttclient.runner.MqttRunner         : ID of saved item: 8
2022-06-04 22:13:50.234  INFO 27984 --- [77-57f911ab85f5] r.d.mqttclient.runner.MqttRunner         : Message arrived on topic 1001/0/2004
2022-06-04 22:13:50.234  INFO 27984 --- [77-57f911ab85f5] r.d.mqttclient.runner.MqttRunner         : Received at timestamp: 1
2022-06-04 22:13:50.234  INFO 27984 --- [77-57f911ab85f5] r.d.mqttclient.runner.MqttRunner         : Received value: 0.00
2022-06-04 22:13:50.234  INFO 27984 --- [77-57f911ab85f5] r.d.mqttclient.runner.MqttRunner         : Saving record for resource 2004
2022-06-04 22:13:50.297  INFO 27984 --- [77-57f911ab85f5] r.d.mqttclient.runner.MqttRunner         : ID of saved item: 9
```

Figure 3-48. Messages collected by the MQTT Message Handler

After the data has been decoded from CBOR, the values of the IPSO object are iterated and sent to the Cloud Gateway. The implementation details of this operation are abstracted into the *RecordService*, which uses *RestTemplate* to issue requests, placing the Bearer Token into the authorization header and doing format manipulation to present the request body in the required form.

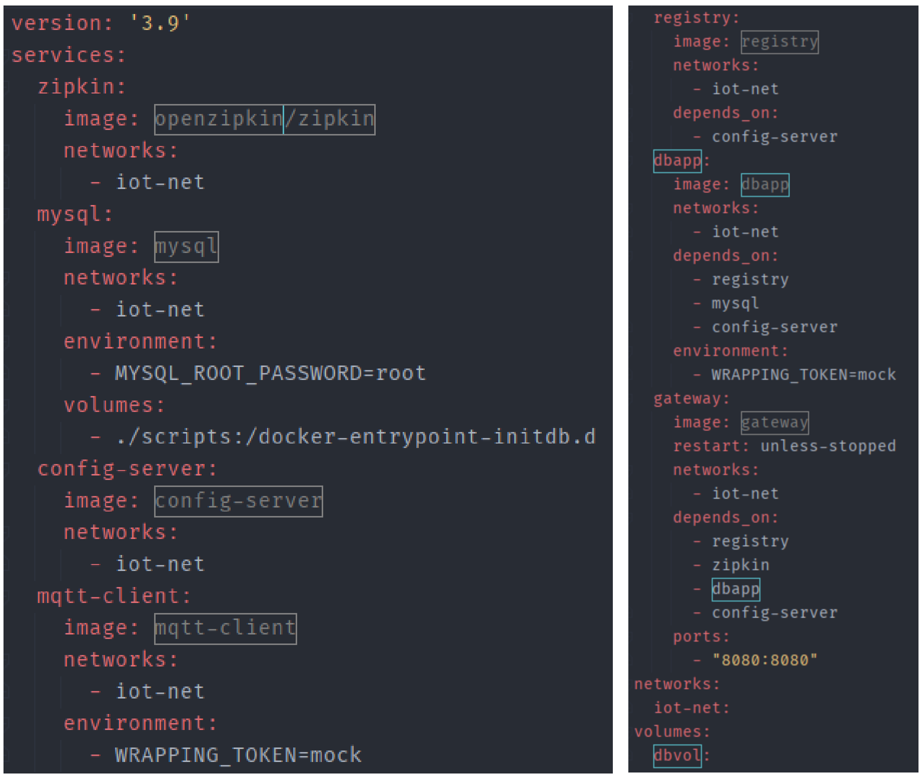


GET {{api_url}}/iot/1001

```
[
    {
        "resourceId": 2001,
        "friendlyName": "temp reading",
        "records": [
            {
                "id": 1,
                "datatype": "float",
                "value": "21.5",
                "timestamp": "2022-05-22T00:00:00.000+00:00"
            },
            {
                "id": 5,
                "datatype": "String",
                "value": "25.42",
```

Figure 3-49. Stored records

Since this solution involves multiple microservices, each hosted into their own container, a *docker-compose.yml* file was created, which allows a declarative approach to starting and managing multiple containers. In places where a wrapping token was required for access to HashiCorp Vault, the operator of the solution is tasked with generating such tokens and placing them in the environment service of each microservice. This is then bootstrapped as an environment variable from which it is retrieved by the start shell script.



```yaml
version: '3.9'
services:
  zipkin:
    image: openzipkin/zipkin
    networks:
      - iot-net
  mysql:
    image: mysql
    networks:
      - iot-net
    environment:
      - MYSQL_ROOT_PASSWORD=root
    volumes:
      - ./scripts:/docker-entrypoint-initdb.d
  config-server:
    image: config-server
    networks:
      - iot-net
  mqtt-client:
    image: mqtt-client
    networks:
      - iot-net
    environment:
      - WRAPPING_TOKEN=mock
```

```yaml
  registry:
    image: registry
    networks:
      - iot-net
    depends_on:
      - config-server
  dbapp:
    image: dbapp
    networks:
      - iot-net
    depends_on:
      - registry
      - mysql
      - config-server
    environment:
      - WRAPPING_TOKEN=mock
  gateway:
    image: gateway
    restart: unless-stopped
    networks:
      - iot-net
    depends_on:
      - registry
      - zipkin
      - dbapp
      - config-server
    ports:
      - "8080:8080"
networks:
  iot-net:
volumes:
  dbvol:
```

Figure 3-50. Docker compose file

The infrastructure for this project consists of an Oracle Cloud Instance featuring a quad-core CPU with an ARM64 architecture and 24 gigabytes of RAM, running a minimal version of Ubuntu 22.04. This hosts the MQTT broker, the Vault Server, and the Docker Engine with great performance, never exceeding the allocated resources.

4. CONCLUSION

In conclusion, a secured end-to-end IoT solution for office building monitoring using a cloud connected sensor is in the realms of possibility with the current processing powered offered by low powered devices, which will ever increase in the future. This solution features an easy to configure IoT node providing graphical interfaces both for Wi-Fi credentials and for a management portal. Additionally, it provides a mean to attest the origin of the node, by undergoing the Attestation Process. A type-agnostic way of sending data to MQTT was designed, fully secured by TLS in pre-shared-key mode and compliant with industry standards such as IPSO Smart Object, whilst still providing a relatively small payload footprint by encoding it to CBOR. Several cloud-native Spring Boot microservices were developed to support the scope of the solution, adhering to industry good practices such as the usage of secret management tools or configuration servers. Records are also persisted in a MySQL database, and all the requests are authenticated and authorized using an Identity Provider in a stateless manner with the help of JSON Web Tokens. Advanced features like anomaly detection using neural networks and blockchain integration support by submitting transactions signed on the edge device are supported in a pluggable manner, whilst still adhering to the constraints of the embedded device.

Whilst the above results are a good starting point, the solution can be further enhanced by adding more machine learning models, adding a battery powered Real-Time clock module to remove the dependency on the NTP server and implementing CoAP support for intranet usage. To ease the manipulation of a large number of records, a data warehouse microservice can also be developed. From the security point of view, whilst being a secure solution at the moment of writing, usage of such a device in the post-quantum world would be highly discouraged due to the heavy use of asymmetric cryptography. Algorithms such as Shor’s should have no issue breaking the encryption, being able to factorize an integer in polynomial time.

REFERENCES

[1] MMH, “Gartner’s Hype Cycle: IoT in “trough” but transformational stage on the way,” 9 September 2020. [Online]. Available: https://www.mmh.com/article/gartners_hype_cycle_iot_in_trough_but_transformational_stage_on_the_way.

[2] IETF, “RFC 7252 - The Constrained Application Protocol (CoAP),” [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7252.

[3] Hewlett Packard Enterprise, “IPSO Object Reference Guide,” [Online]. Available: https://techlibrary.hpe.com/docs/otlink-wo/IPSO-Object-Reference-Guide.html.

[4] IETF, “RFC 7049 - Concise Binary Object Representation (CBOR),” [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7049.

[5] K. D. Foote, “A Brief History of the Internet of Things,” [Online]. Available: https://www.dataversity.net/brief-history-internet-things/#.

[6] Connectivity Standards Alliance, “Build With Matter,” [Online]. Available: https://csa-iot.org/all-solutions/matter/.

[7] NIST, “Lightweight Cryptography,” [Online]. Available: https://csrc.nist.gov/Projects/lightweight-cryptography.

[8] Kaspersky, “Cryptography Definition,” [Online]. Available: https://www.kaspersky.com/resource-center/definitions/what-is-cryptography.

[9] P. Wackerow, “Nodes and Clients,” [Online]. Available: https://ethereum.org/en/developers/docs/nodes-and-clients/.

[10] C. Smith, “Blocks,” [Online]. Available: https://ethereum.org/en/developers/docs/blocks/.

[11] T. Batra, “Transactions,” [Online]. Available: https://ethereum.org/en/developers/docs/transactions/#types-of-transactions.

[12] Joshua, “Introduction to Smart Contracts,” [Online]. Available: https://ethereum.org/en/developers/docs/smart-contracts/.

[13] MQTT, “MQTT,” [Online]. Available: https://mqtt.org/.

[14] Amazon, “Pub/Sub Messaging,” [Online]. Available: https://aws.amazon.com/pub-sub-messaging/.

[15] Docker, “Docker overview,” [Online]. Available: https://docs.docker.com/get-started/overview/.

[16] J. Fong, “Are Containers Replacing Virtual Machines?,” [Online]. Available: https://www.docker.com/blog/containers-replacing-virtual-machines/.

[17] M. v. Neerven, “Cloud Platforms and the Application Development Paradigm Shift,” [Online]. Available: https://www.linkedin.com/pulse/cloud-platforms-application-development-paradigm-marc-van-neerven/.

[18] A. Geron, Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, O'Reilly.

[19] K. Sovani, “Understanding ESP32’s Security Features,” [Online]. Available: https://blog.espressif.com/understanding-esp32s-security-features-14483e465724.

[20] C. Greenberg, “How Tech Manufacturers Can Manage Price Sensitivity Amid Disruption,” 5 July 2020. [Online]. Available: https://www.supplychainbrain.com/blogs/1-think-tank/post/31611-how-tech-manufacturers-can-manage-price-sensitivity-amid-disruption.

[21] ESP-IDF, “Build System,” [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/build-system.html.

[22] Solidity, “Contract ABI Specification,” [Online]. Available: https://docs.soliditylang.org/en/v0.8.13/abi-spec.html.

[23] Solidity, “Types,” [Online]. Available: https://docs.soliditylang.org/en/v0.4.21/types.html.