Ecole supérieure en Informitique

Sidi Bel Abbes

Option : ISI

Conception of a Microservice based IoT System

Authors
Khebchi Abdallah
Laroui Mohammed
Bendaoud Mohammed Amine

Advisors
Pr. Rahmoun
Dr. Bensenane

September 9, 2021

Abstract

Due to large-scale IoT systems and connected devices, enterprises face difficulty in developing, deploying, integrating and scaling the applications. Microservices and containerization enable efficient and faster development by breaking down IoT functionalities into small, modular and independent units that work in isolation without affecting the overall performance of the IoT ecosystem.

With the advent of IoT and DevOps 2.0, the technology and architecture on which the IoT platforms and solutions are built will be one of the driving factors, determining the long-term sustainability and success for any organization. If we time travel from the times of mainframe, client/server, and web to now in the cloud era, the adoption of microservices and containerization becomes imperative in the IoT world.

Contents

1 Introduction

IoT has a huge deployment ecosystem containing multiple servers, applications, sensors, and protocols. It also has many end-points like firmware, web, mobile, and QA, which require heavy integration between device, data, and applications. This increases the development time and efforts even in an agile setup. Gartner predicts that through 2018, half the cost of implementing IoT solutions will be spent in integrating various IoT components with each other and back-end systems. With microservices, the functionality is broken down into the lowest level components as small, modular, independently deployable and loosely-coupled services, which reduces the integration complexity faced with monolithic architecture.

1.1 Objectives

The objective of this project is to build and maintain an IoT platform based on Microservice architecture with the adoption of all it aspects from design to deployment making the use of DevOps tools such as Docker for containerization our services hosted in Raspberry pi 3 b+ server.

2 Background

In this section, several related concepts will be explained for better understanding the process of project.

2.1 Internet of Things

The Internet of Things, or IoT, refers to the billions of physical devices around the world that are now connected to the internet, all collecting and sharing data. Thanks to the arrival of super-cheap computer chips and the ubiquity of wireless networks, it's possible to turn anything, from something as small as a pill to something as big as an aeroplane, into a part of the IoT. Connecting up all these different objects and adding sensors to them adds a level of digital intelligence to devices that would be otherwise dumb, enabling them to communicate real-time data without involving a human being. The Internet of Things is making the fabric of the world around us more smarter and more responsive, merging the digital and physical universes.

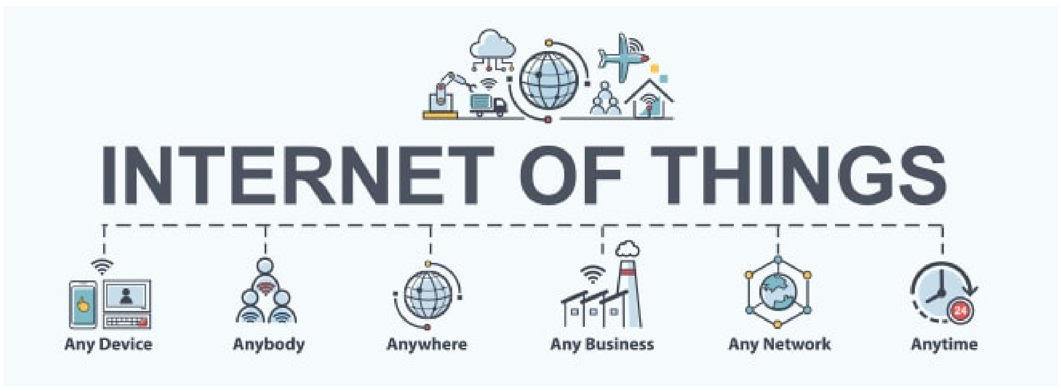


Figure 2.1: IOT

2.2 Microservices

2.2.1 What is Microservices

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

- Highly maintainable and testable
- Loosely coupled
- Independently deployable
- Organized around business capabilities
- Owned by a small team

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

2.2.2 Why using Microservices

With microservices, multiple teams work on independent services, enabling you to deploy more quickly — and pivot more easily when you need to. Development time is reduced, and your teams' code will be more reusable. By decoupling services, you won't have to operate on expensive machines. Basic x86 machines will do. The increased efficiency of microservices not only reduces infrastructure costs, it also minimizes downtime.

2.3 Containerization

2.3.1 What is Containerization

Containerization is the packaging of software code with just the operating system (OS) libraries and dependencies required to run the code to create a single lightweight executable—called a container—that runs consistently on any infrastructure. More portable and resource-efficient than virtual machines (VMs), containers have become the de facto compute units of modern cloud-native applications.

2.3.2 Why using Containers

Containers give developers the ability to create predictable environments that are isolated from other applications. Containers can also include software dependencies needed by the application, such as specific versions of programming language runtimes and other software libraries

2.3.3 Docker

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications.

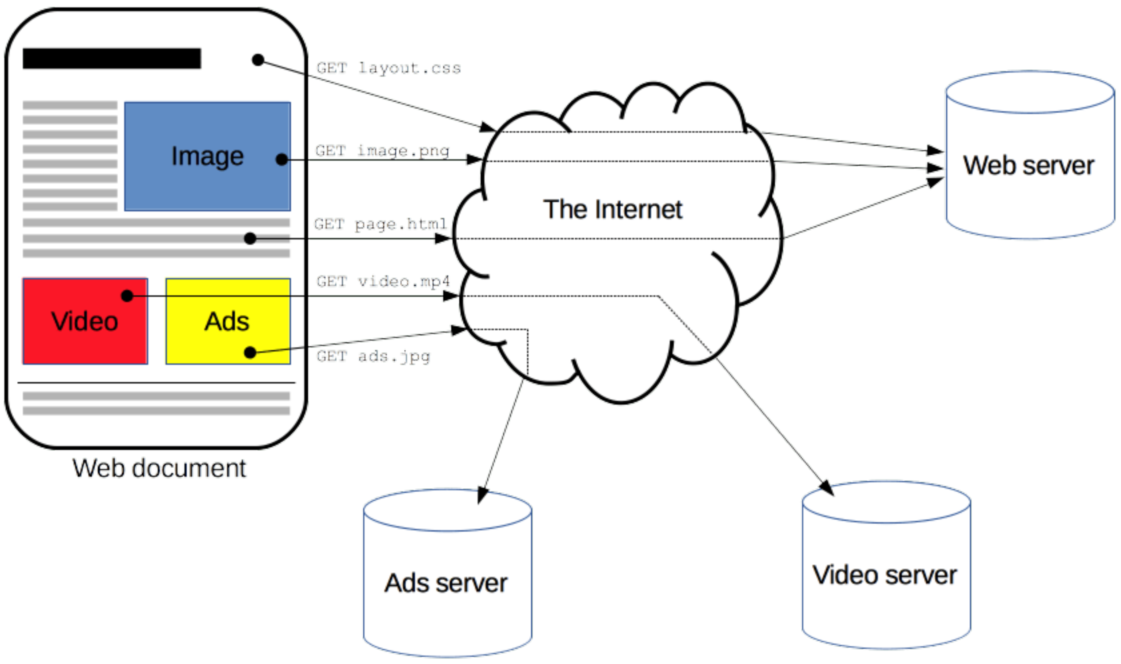
2.3.4 Docker Compose

Docker Compose is a tool that was developed to help define and share multi-container applications. With Compose, you can create a YAML file to define the services and with a single command, can spin everything up or tear it all down.

2.4 Telecommunication protocols

2.4.1 HTTP

HTTP is a protocol which allows the fetching of resources, such as HTML documents. It is the foundation of any data exchange on the Web and it is a client-server protocol, which means requests are initiated by the recipient, usually the Web browser. A complete document is reconstructed from the different sub-documents fetched, for instance text, layout description, images, videos, scripts, and more.



Clients and servers communicate by exchanging individual messages (as opposed to a stream of data). The messages sent by the client, usually a Web browser, are called requests and the messages sent by the server as an answer are called responses.

2.4.2 TCP

TCP stands for Transmission Control Protocol a communications standard that enables application programs and computing devices to exchange messages over a network. It is designed to send packets across the internet and ensure the successful delivery of data and messages over networks. As a result, high-level protocols that need to transmit data all use TCP Protocol. Examples include peer-to-peer sharing methods like File Transfer Protocol (FTP), Secure Shell (SSH), and Telnet. It is also used to send and receive email through Internet Message Access Protocol (IMAP), Post Office Protocol (POP), and Simple Mail Transfer Protocol (SMTP), and for web access through the Hypertext Transfer Protocol (HTTP).

2.4.3 MQTT

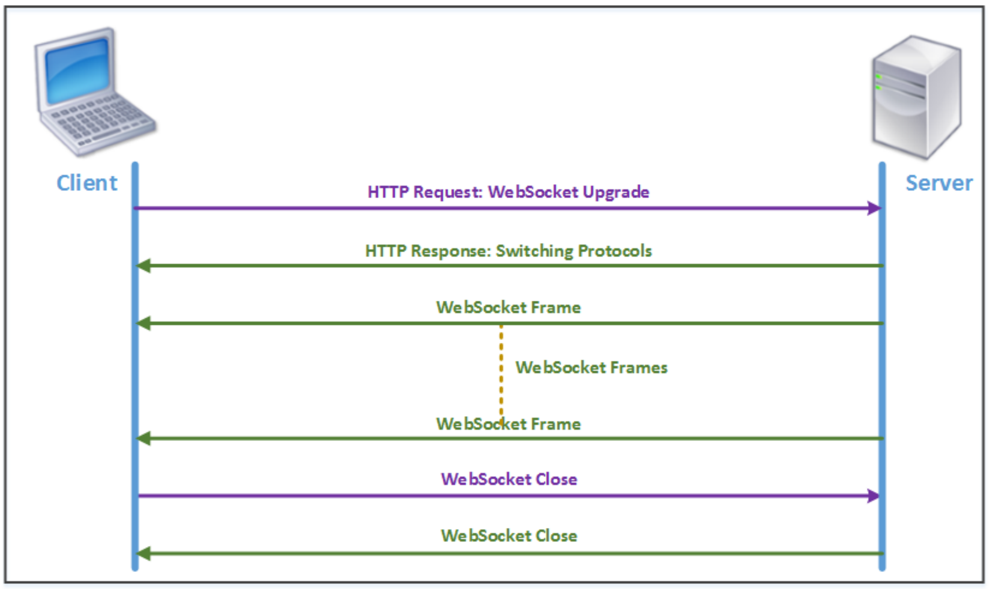
MQTT is a publish/subscribe protocol that allows edge-of-network devices to publish to a broker. Clients connect to this broker, which then mediates communication between the two devices. ... When another client publishes a message on a subscribed topic, the broker forwards the message to any client that has subscribed.

2.4.4 SSH

SSH, or Secure Shell, is a remote administration protocol that allows users to control and modify their remote servers over the Internet.It provides a mechanism for authenticating a remote user, transferring inputs from the client to the host, and relaying the output back to the client

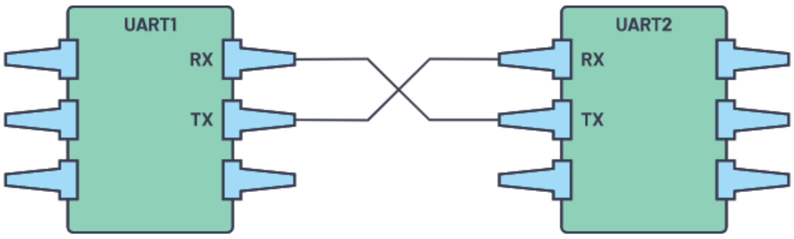
2.4.5 Websocket

A WebSocket is a persistent connection between a client and server. WebSockets provide a bidirectional, full-duplex communications channel that operates over HTTP through a single TCP/IP socket connection. At its core, the WebSocket protocol facilitates message passing between a client and server



2.4.6 UART

By definition, UART is a hardware communication protocol that uses asynchronous serial communication with configurable speed. Asynchronous means there is no clock signal to synchronize the output bits from the transmitting device going to the receiving end.



Two UARTs directly communicate with each other

2.4.7 I2C

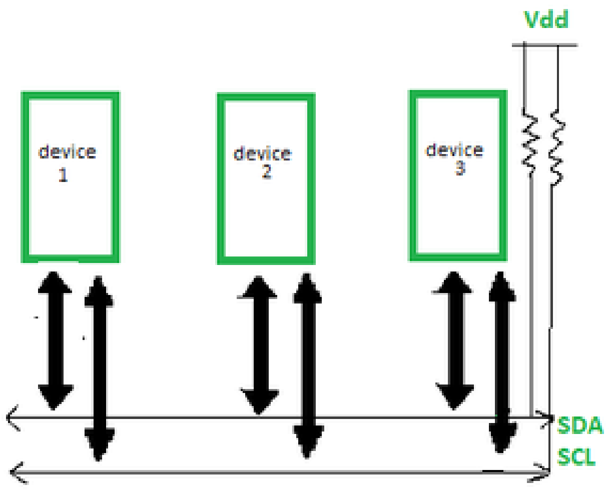
I2C stands for Inter-Integrated Circuit. It is a bus interface connection protocol incorporated into devices for serial communication. It was originally designed by Philips Semiconductor in 1982. Recently, it is a widely used protocol for short-distance communication. Working of I2C Communication Protocol : It uses only 2 bi-directional open-drain lines for data communication called SDA and SCL. Both these lines are pulled high.

Serial Data (SDA) – Transfer of data takes place through this pin. Serial Clock (SCL) – It carries the clock signal.

I2C operates in 2 modes –

- Master mode
- Slave mode

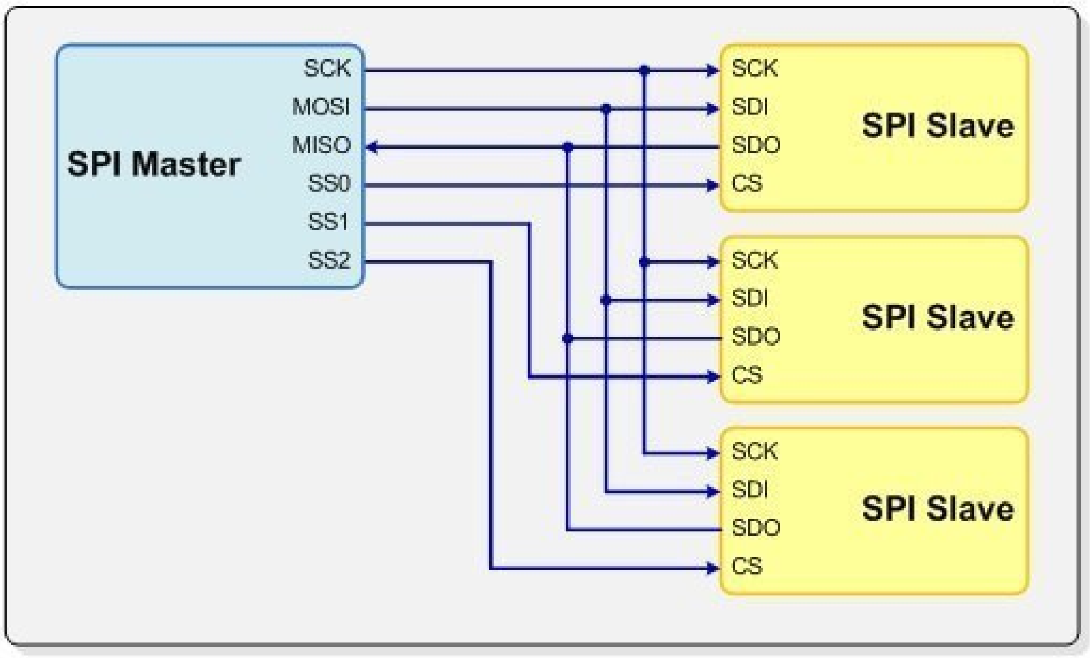
Each data bit transferred on SDA line is synchronized by a high to the low pulse of each clock on the SCL line.



I2C protocol

2.4.8 SPI

The Serial Peripheral Interface (SPI) bus is a synchronous serial communication interface specification used for short distance communication, primarily in embedded systems. ... SPI devices communicate in full duplex mode using a master-slave architecture with a single master.



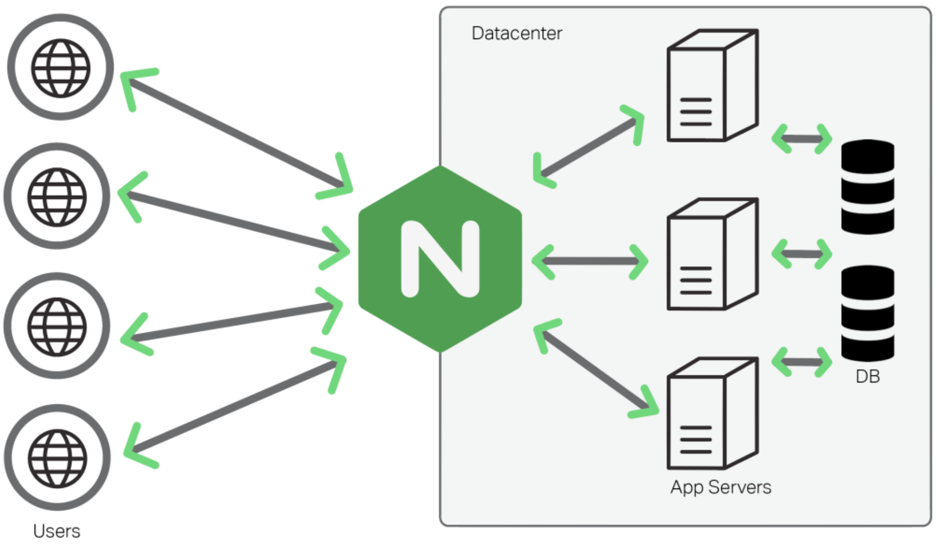
wire SPI bus configuration with multiple slaves

2.4.9 USB

The USB protocol, also known as Universal Serial Bus, was first created and introduced in 1996 as a way to institutionalize a more widespread, uniform cable and connector that could be used across a multitude of different devices. With the increase in technological devices during this time, having a universal cable would help reduce the confusion and inconvenience of having a collection of cables needed for each individual device.

2.5 Web servers

2.5.1 NGINX



NGINX is open source software for web serving, reverse proxying, caching, load balancing, media streaming, and more. It started out as a web server designed for maximum performance and stability.

How Does Nginx Work? Nginx is built to offer low memory usage and high concurrency. Rather than creating new processes for each web request, Nginx uses an asynchronous, event-driven approach where requests are handled in a single thread. With Nginx, one master process can control multiple worker processes

2.6 Databases

A database is an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by a database management system (DBMS). ... The data can then be easily accessed, managed, modified, updated, controlled, and organized.

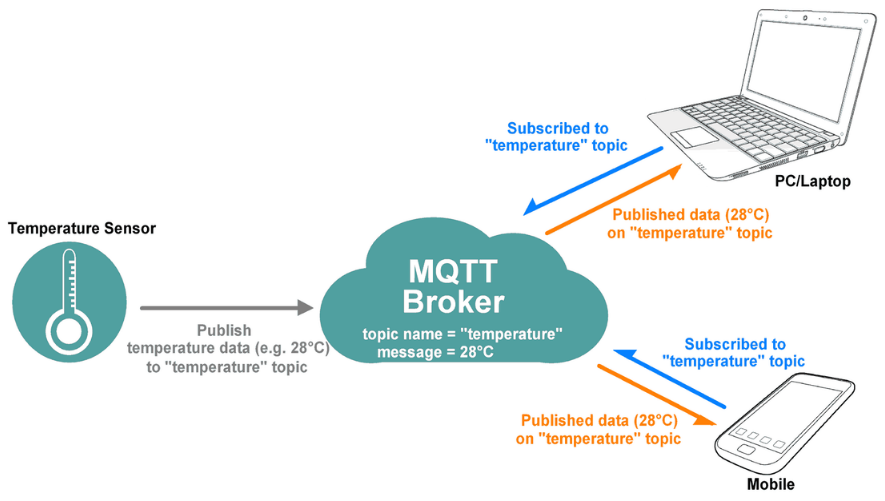
2.6.1 Postgresql

PostgreSQL is a powerful, open source object-relational database system that uses and extends the SQL language combined with many features that safely store and scale the most complicated data workloads.

2.7 Message Brokers

Message brokers are often used to manage communications between on-premises systems and cloud components in hybrid cloud environments. Using a message broker gives increased control over interservice communications, ensuring that data is sent securely, reliably, and efficiently between the components of an application

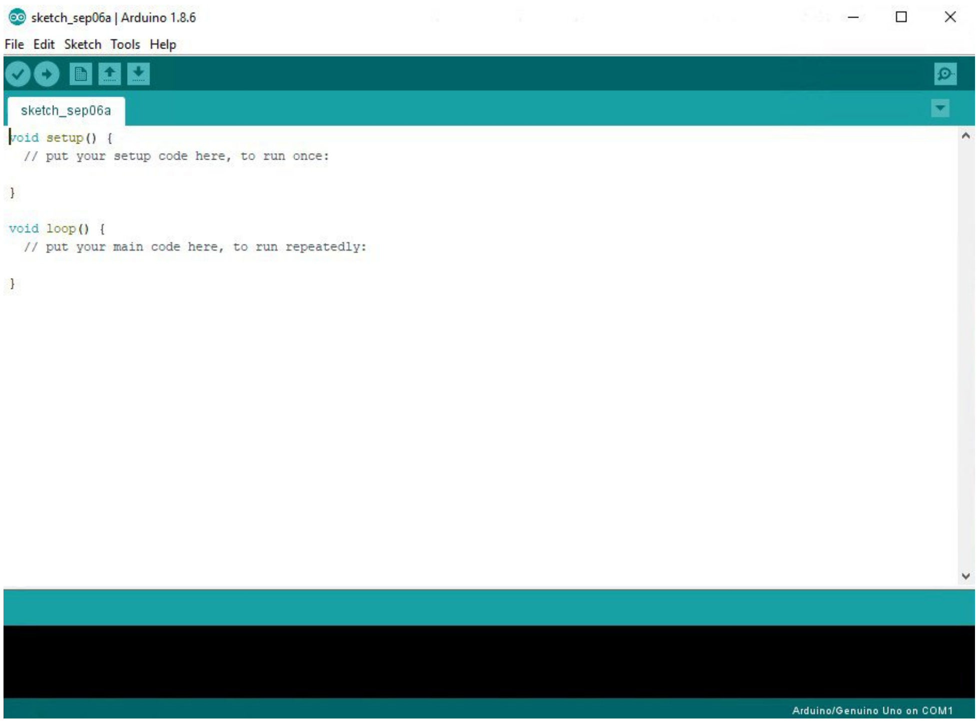
2.7.1 Mosquitto


Subscribed to "temperature" topic
PC/Laptop
Published data (28°C) on "temperature" topic
Temperature Sensor
MQTT Broker
topic name = "temperature"
message = 28°C
Publish temperature data (e.g. 28°C) to "temperature" topic
Subscribed to "temperature" topic
Published data (28°C) on "temperature" topic
Mobile

Mosquitto is a lightweight open source message broker that Implements MQTT versions 3.1.0, 3.1.1 and version 5.0. It is written in C by Roger Light, and is available as a free download for Windows and Linux and is an Eclipse project.

2.8 Ide's and Programming Languages

2.8.1 Arduino IDE


sketch_sep06a | Arduino 1.8.6
File Edit Sketch Tools Help
sketch_sep06a
Arduino/Genuino Uno on COM1

```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

The Arduino Integrated Development Environment - or Arduino Software (IDE) - contains a text editor for writing code, a message area, a text console, a toolbar with buttons for common functions and a series of menus. It connects to the Arduino and Genuino hardware to upload programs and communicate with them.

2.8.2 Goland IDE

GoLand is a cross-platform integrated development environment (IDE) for Go developers. GoLand includes such features as context-dependent code completion and refactoring, debugging, profiling, type and declaration navigation, and error analysis. In addition to tools for core Go development, supports JavaScript, TypeScript, Node.js, SQL, Databases, Docker, Kubernetes, and Terraform. You can always extend current functionality by installing additional plugins from the plugin repository.

2.8.3 C++ Language

C++ is a cross-platform language that can be used to create high-performance applications. C++ was developed by Bjarne Stroustrup, as an extension to the C language. C++ gives programmers a high level of control over system resources and memory.

Why using C++ in embedded programming

By design, C++ lends itself to embedded development because the language sits in between higher-level software and hardware, allowing you to access and control hardware directly without sacrificing the benefits of a high-level language.. It's particularly effective for hardware that will need to be around for a while, as programs written in C++ can operate for decades at a time due to the language's high stability.

C++ also gives developers the ability to efficiently use abstractions without too much cost to infrastructure. The data structure of C++, like C, is algorithm-based, which makes it a great choice for solving all the little puzzles you encounter during embedded development.

2.8.4 Go Language

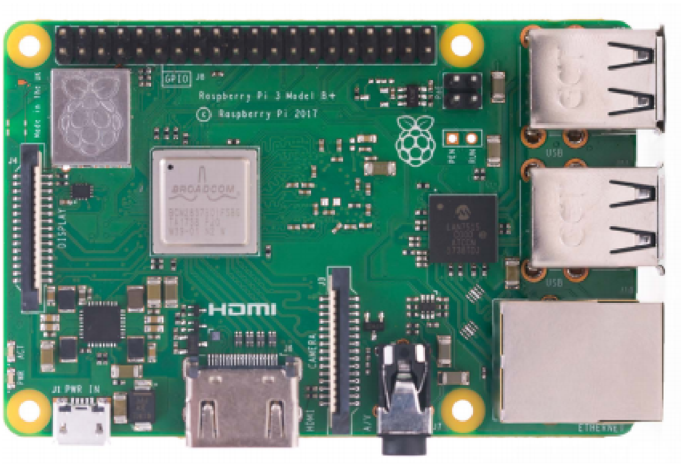
Go was originally designed at Google in 2007. At the time, Google was growing quickly, and code being used to manage their infrastructure was also growing quickly in both size and complexity. Some Google engineers began to feel that this large and complex codebase was slowing them down. So they decided that they needed a new programming language focused on simplicity and quick performance. Robert Griesemer, Rob Pike, and Ken Thompson designed Go.

Why using Go in microservices

Although go is a new language compared to others, it has many advantages. Programs coded in golang are more robust. They can withstand heavy loads that allow applications to build with loaded services. Golang is more suited for multiprocessor systems and web apps. Moreover, it integrates with GitHub easily to manage the distributed packages. The most use of microservice architecture is done when the application needs to be scalable. And if there is one language that can perfectly fit the criteria, then it is – Golang, the reason that it does is because of its inheritance from C-family programming languages, the components written in golang are easier to combine with components coded in other languages that reside in the same family.

3 Hardware

3.1 Raspberry Pi 3 B+



3.1.1 Definition

The Raspberry Pi 3 Model B+ is the latest product in the Raspberry Pi 3 range, boasting a 64-bit quad core processor running at 1.4GHz, dual-band 2.4GHz and 5GHz wireless LAN, Bluetooth 4.2/BLE, faster Ethernet, and PoE capability via a separate PoE HAT The dual-band wireless LAN comes with modular compliance certification, allowing the board to be designed into end products with significantly reduced wireless LAN compliance testing, improving both cost and time to market. The Raspberry Pi 3 Model B+ maintains the same mechanical footprint as both the Raspberry Pi 2 Model B and the Raspberry Pi 3 Model B.

3.1.2 Specifications

- Processor : Broadcom BCM2837B0, Cortex-A53 64-bit SoC @ 1.4GHz
- Memory : 1GB LPDDR2 SDRAM
- Connectivity:

  - 2.4GHz and 5GHz IEEE 802.11.b/g/n/ac wireless LAN, Bluetooth 4.2, BLE
  - Gigabit Ethernet over USB 2.0 (maximum throughput 300Mbps)
  - 4 × USB 2.0 ports

- Access : Extended 40-pin GPIO header
- Video and Sound :
  - 1 × full size HDMI
  - MIPI DSI display port
  - MIPI CSI camera port
  - 4 pole stereo output and composite video port
- Multimedia: H.264, MPEG-4 decode (1080p30); H.264 encode (1080p30); OpenGL ES 1.1, 2.0 graphics
- SD card support: Micro SD format for loading operating system and data storage
- Input power:
  - 5V/2.5A DC via micro USB connector
  - 5V DC via GPIO header
  - Power over Ethernet (PoE)–enabled (requires separate PoE HAT)

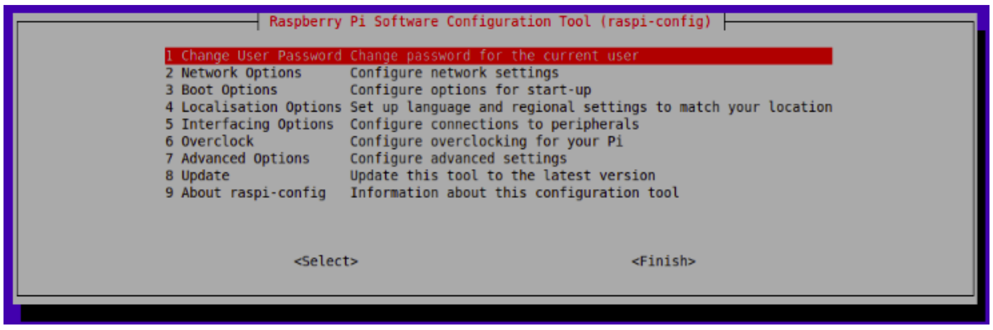
3.1.3 Configuration

raspin installation

Raspberry Pi recommend the use of Raspberry Pi Imager to install an operating system on your SD card. You will need another computer with an SD card reader to install the image.
full video : https://youtu.be/ntaXWS8Lk34

Config

The raspi-config raspi-config is the Raspberry Pi configuration tool originally written by Alex Bradbury. You will be shown raspi-config on first booting into Raspberry Pi OS. To open the configuration tool after this,a simply run the following from the command line: sudo raspi-config



```
Raspberry Pi Software Configuration Tool (raspi-config)

1 Change User Password Change password for the current user
2 Network Options      Configure network settings
3 Boot Options         Configure options for start-up
4 Localisation Options Set up language and regional settings to match your location
5 Interfacing Options  Configure connections to peripherals
6 Overclock            Configure overclocking for your Pi
7 Advanced Options     Configure advanced settings
8 Update               Update this tool to the latest version
9 About raspi-config   Information about this configuration tool

                <Select>                                  <Finish>
```

3.1.4 Network Configuration

You will need to define a Wpa.application.conf file
A wpa.supplicant.conf file example:

```
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
country=<Insert 2 letter ISO 3166-1 country code here>
update_config=1

network={
 ssid="<Name of your wireless LAN>"
 psk="<Password for your wireless LAN>"
}
```

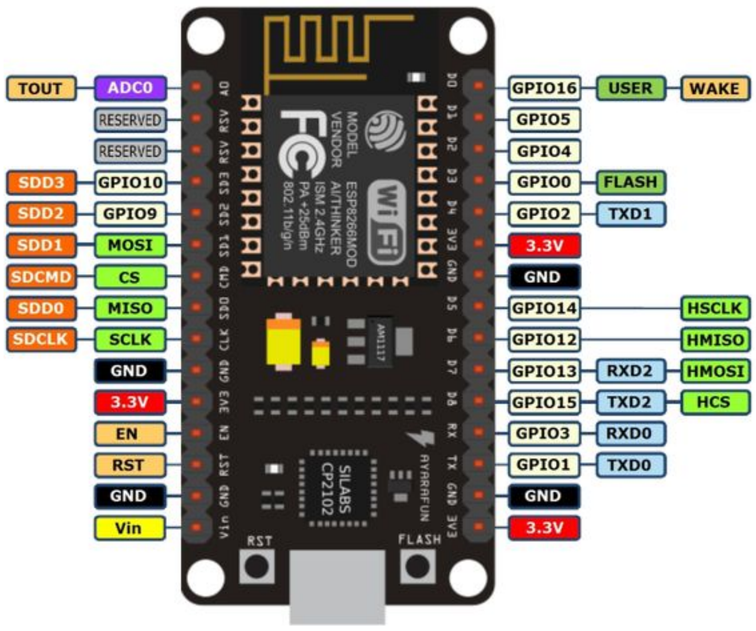
3.1.5 Static IP Addresses

If you wish to disable automatic configuration for an interface and instead configure it statically, add the details to /etc/dhcpcd.conf. For example:

```
interface eth0
static ip_address=192.168.0.4/24
static routers=192.168.0.254
static domain_name_servers=192.168.0.254 8.8.8.8
```

3.2 ESP8266

3.2.1 Definition



ESP8266 is a wifi SOC (system on a chip) produced by Espressif Systems . It is an highly integrated chip designed to provide full internet connectivity in a small package. ESP8266 can be used as an external Wifi module, using the standard AT Command set Firmware by connecting it to any microcontroller using the serial UART, or directly serve as a Wifi-enabled micro controller, by programming a new firmware using the provided SDK.

The GPIO pins allow Analog and Digital IO, plus PWM, SPI, I2C, etc.

3.2.2 Specification

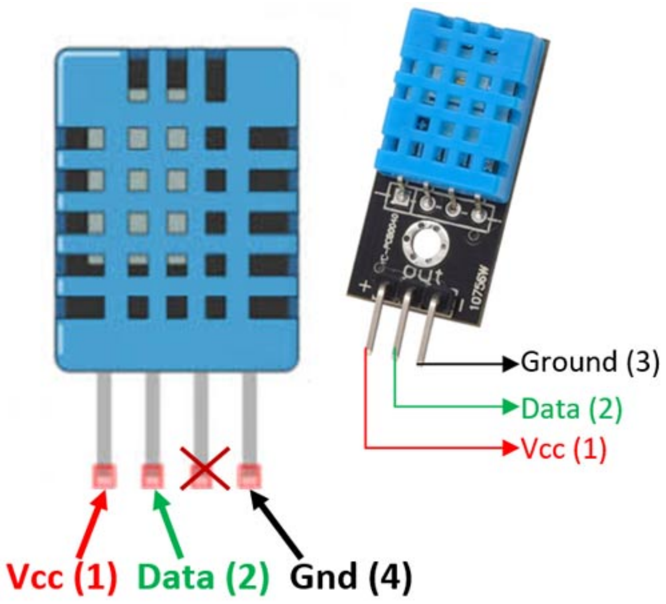
- Processor: L106 32-bit RISC microprocessor core based on the Tensilica Xtensa Diamond Standard 106Micro running at 80 MHz
- Memory:
  - 32 KiB instruction RAM
  - 32 KiB instruction cache RAM
  - 80 KiB user-data RAM
  - 16 KiB ETS system-data RAM

- External QSPI flash: up to 16 MiB is supported (512 KiB to 4 MiB typically included)
- IEEE 802.11 b/g/n Wi-Fi:
  - Integrated TR switch, balun, LNA, power amplifier and matching network
  - WEP or WPA/WPA2 authentication, or open networks
- 17 GPIO pins
- SPI
- I2C (software implementation)
- I2S interfaces with DMA (sharing pins with GPIO)
- UART on dedicated pins, plus a transmit-only UART can be enabled on GPIO2
- 10-bit ADC (successive approximation ADC)

3.3 Sensors

The main purpose of sensors is to collect data from the surrounding environment. Sensors, or 'things' of the IoT system, form the front end. These are connected directly or indirectly to IoT networks after signal conversion and processin

3.3.1 DHT11

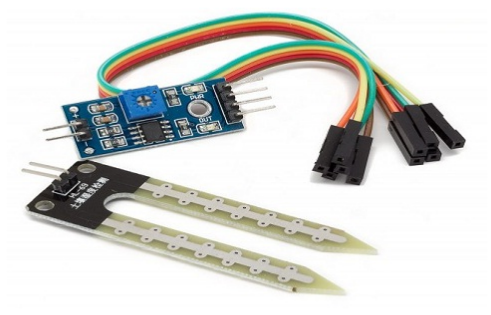


The DHT11 is a commonly used Temperature and humidity sensor that comes with a dedicated NTC to measure temperature and an 8-bit microcontroller to output the values of temperature and humidity as serial data.

Specifications

- Operating Voltage: 3.5V to 5.5V
- Operating current: 0.3mA (measuring) 60uA (standby)
- Output: Serial data
- Temperature Range: 0°C to 50°C
- Humidity Range: 20
- Resolution: Temperature and Humidity both are 16-bit
- Accuracy: ±1°C and ±1

3.3.2 Soil Moisture Sensor



The soil moisture sensor is one kind of sensor used to gauge the volumetric content of water within the soil. As the straight gravimetric dimension of soil moisture needs eliminating, drying, as well as sample weighting. These sensors measure the volumetric water content not directly with the help of some other rules of soil like dielectric constant, electrical resistance, otherwise interaction with neutrons, and replacement of the moisture content.

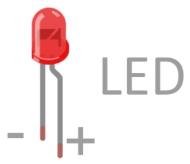
The relation among the calculated property as well as moisture of soil should be adjusted and may change based on ecological factors like temperature, type of soil, otherwise electric conductivity. The microwave emission which is reflected can be influenced by the moisture of soil as well as mainly used in agriculture and remote sensing within hydrology.

Specifications

- The required voltage for working is 5V
- The required current for working is ¡20mA
- Type of interface is analog
- The required working temperature of this sensor is 10°C 30°C

3.4 Actors

3.4.1 LED-pin



LED stands for light-emitting diode, which means that much like their diode cousins, they're polarized. There are a handful of identifiers for finding the positive and negative pins on an LED. You can try to find the longer leg, which should indicate the positive, anode pin.

4 Software

4.1 Requirements

There are three primary categories of system requirements: functional, nonfunctional, and architectural. Functional requirements are “what” the system should do. Nonfunctional requirements are “how well” the system should perform in one or more areas. Architectural requirements are more descriptive of “connections” between the subsystems to form the final system.

Functional versus nonfunctional requirements Functional requirements specify what the system has to do. They are traceable to a specific source, often to Use Cases or Business Rules. They are often called “product features.”

4.1.1 Functional Requirements

The user can :

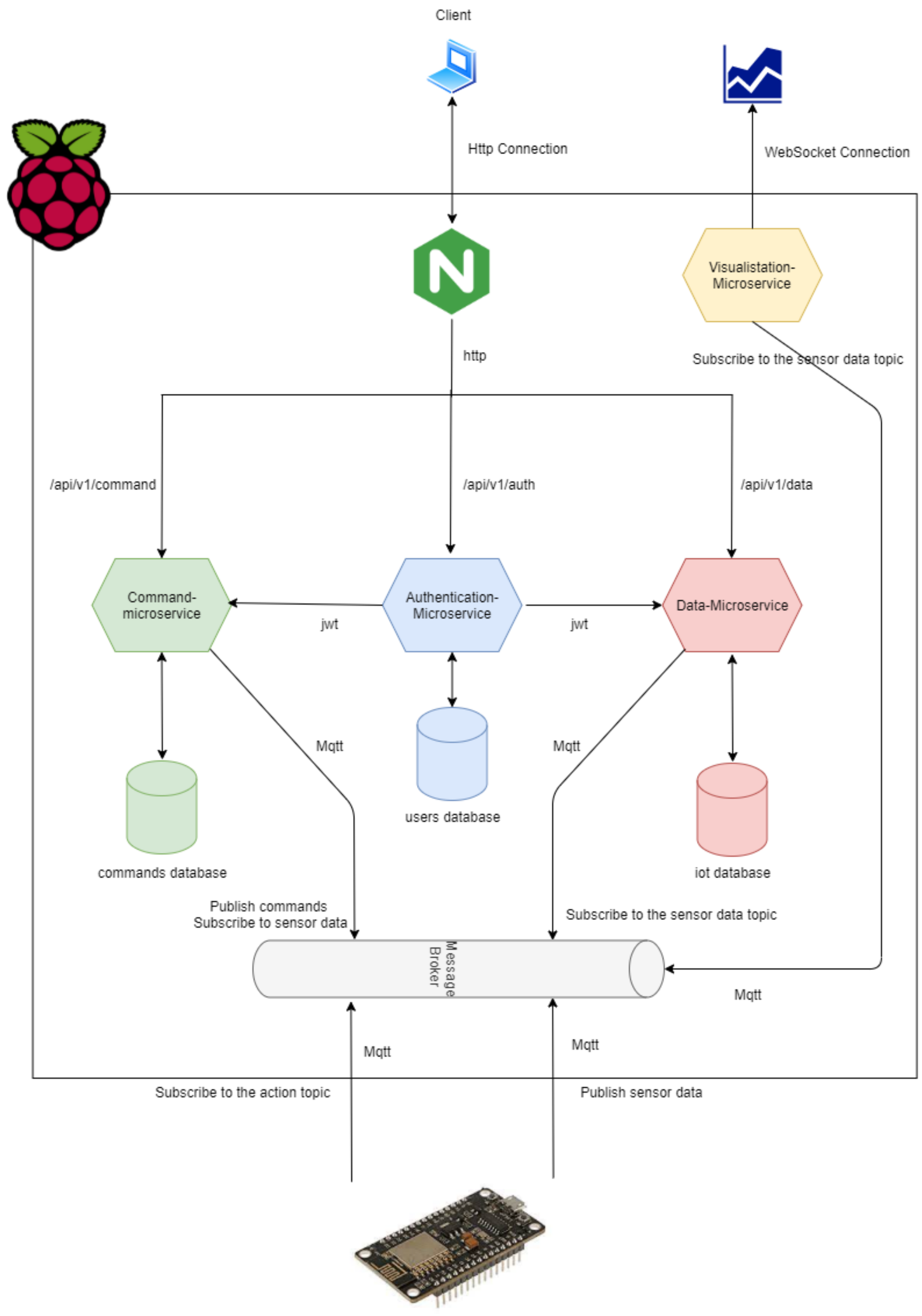
- Signup and login into the application
- Visualise the collected data in form of graph
- Query the collected data :
  - Query all the data stored
  - Query the last N-measurement
  - Query the collected data by date
- Perform a command to actors (ON——OFF)
- Add timed command for some duration
- Setup a trigger/alert on a specific measurement and the command when the alert happens
- Receive emails from the app when the alert is triggered.

4.1.2 Non Functional Requirements

Nonfunctional requirements are mostly quality-related requirements which include the areas of performance, availability, reliability, usability, flexibility, configurability, integration, maintainability, portability, and testability. This category may also include implicit requirements for modification and upgrades, reusability, and interoperability.

4.2 Conception

4.2.1 Global Architecture



4.2.2 Authentication Service

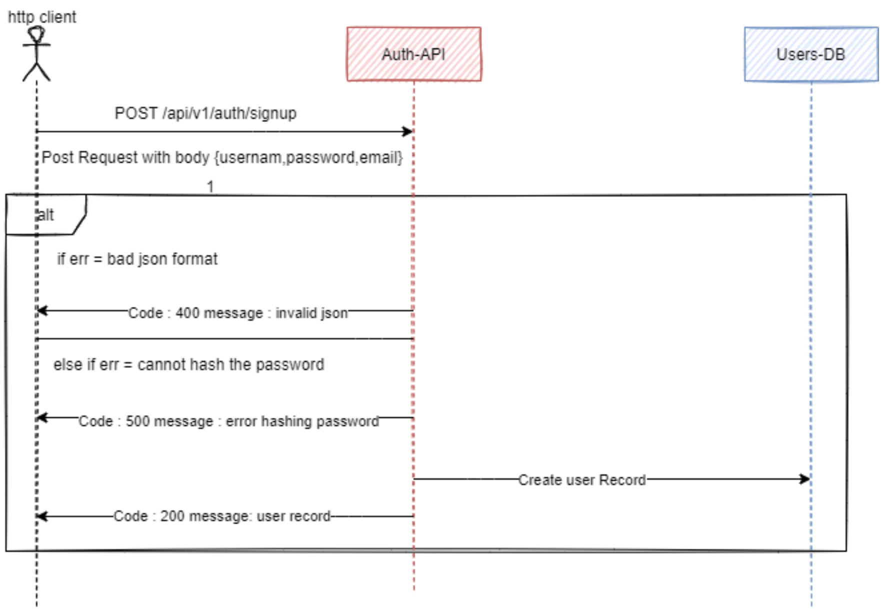
Role

Mainly the auth-service is responsible for securing user infos and protecting API routes around the services .
This service handle 3 requests :

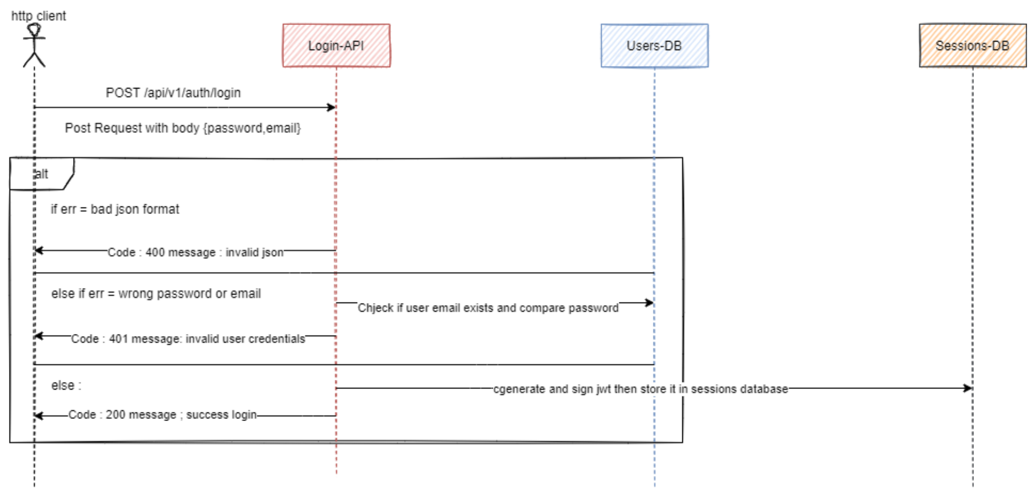
- POST /api/v1/auth/signup
- POST /api/v1/auth/login
- GET /api/v1/auth/logout

Service Workflow

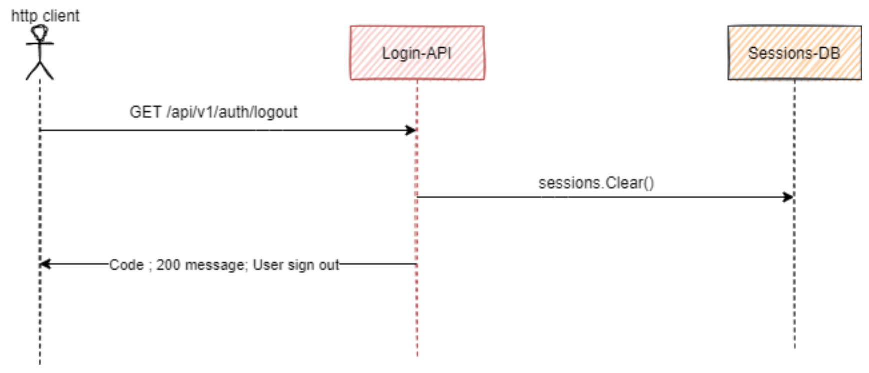
Signup sequence diagram


http client
Auth-API
Users-DB
POST /api/v1/auth/signup
Post Request with body {usernam,password,email}
1
alt
if err = bad json format
Code : 400 message : invalid json
else if err = cannot hash the password
Code : 500 message : error hashing password
Create user Record
Code : 200 message: user record

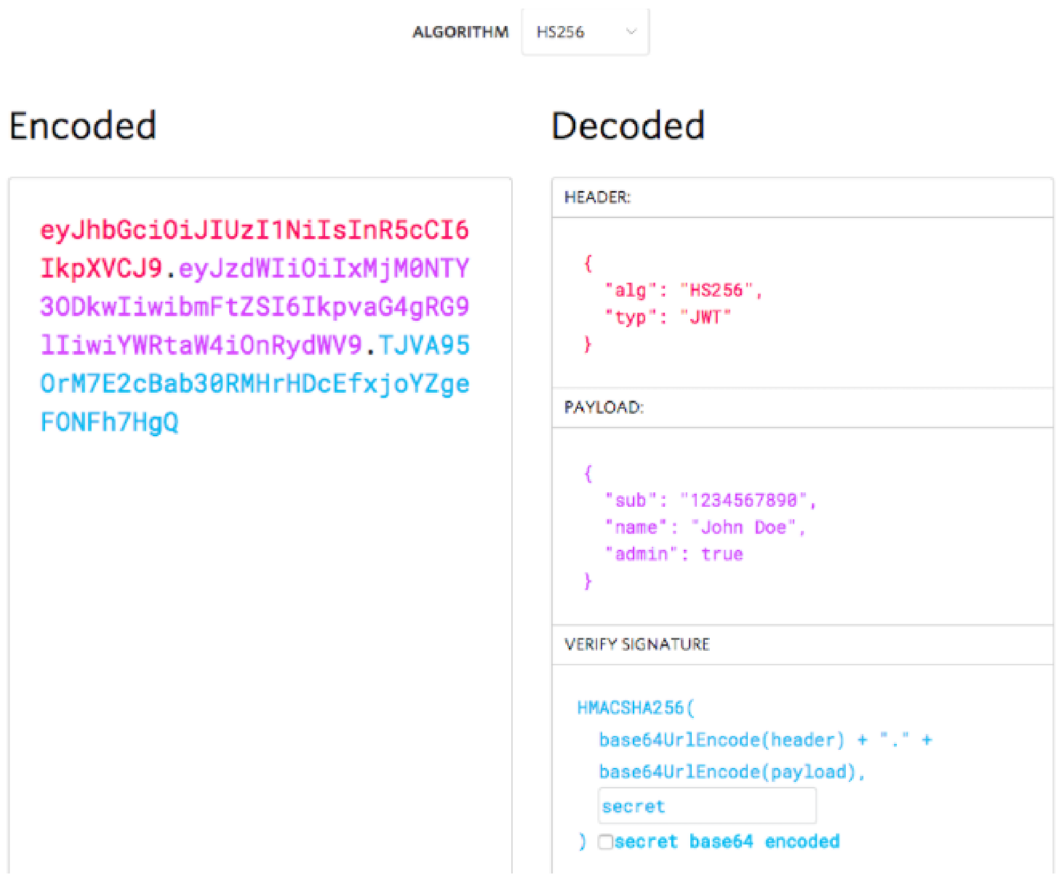
Login sequence diagram


http client
Login-API
Users-DB
Sessions-DB
POST /api/v1/auth/login
Post Request with body {password,email}
alt
if err = bad json format
Code : 400 message : invalid json
else if err = wrong password or email
Chjeck if user email exists and compare password
Code : 401 message: invalid user credentials
else :
cgenerate and sign jwt then store it in sessions database
Code : 200 message ; success login

Logout sequence diagram



JWT



ALGORITHM HS256

Encoded

```
eyJhbGciOiJIUzI1NiIsInR5cCI6
IkpXVCJ9.eyJzdWIiOiIxMjM0NTY
3ODkwIiwibmFtZSI6IkpvaG4gRG9
lIiwiYWRtaW4iOnRydWV9.TJVA95
OrM7E2cBab30RMHrHDcEfxjoYZge
FONFh7HgQ
```

Decoded

HEADER:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret
) ☐secret base64 encoded
```

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA.

Although JWTs can be encrypted to also provide secrecy between parties, we will focus on signed tokens. Signed tokens can verify the integrity of the claims contained within it, while encrypted tokens hide those claims from other parties. When tokens are signed using public/private key pairs, the signature also certifies that only the party holding the private key is the one that signed it. JSON Web Token structure :

- Header: The header typically consists of two parts: the type of the token, which is JWT, and the signing algorithm being used, such as HMAC SHA256 or RSA.
- Payload: The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional data. There are three types of claims: registered, public, and private claims.

- Signature: To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.

Sessions and Redis

When a user login into the app this service store a session containing the signed jwt in the redis database with a specific life-time , other services use it to check the authentication of the user using authentication middleware.

Database model

The sevice communicates with 2 databases (users database and sessions in memory database)
the users database contains the user table (username,hashed password and user's email)
the session database contains the user jwt as a token and a life-time for that specific token (expire time)

4.2.3 Data Service

Role

The main objective of the data-service is storing collected data from sensors into a database .
Data service also responsible for :

- Subscribing to the mqtt topic viq an mqtt client
- Parse the mqtt message into an object
- Store the object in database
- Providing an API that handle some query requests :
  - GET /api/v1/data/all
  - GET /api/v1/data/last
  - GET /api/v1/data/byDate

Database model

The data service store all incoming telemetry data in the iot database containing one table (Event table) with fields such as time and date and measurement data (Temperature,Humidity,Soil moisture)

4.2.4 Command Service

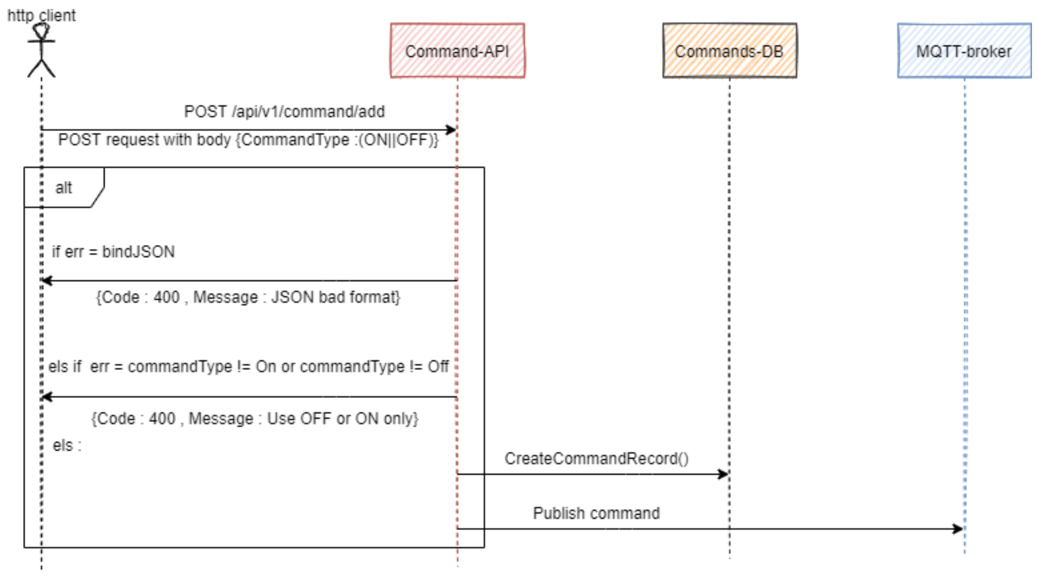
Role

The command service provides an API for performing commands on actors in the nodMCU IC and query all commands that has been done.
The command service also provides an endpoint to set thresholds on specific measurement , the service notify the user by email when the threshold triggered.
The command service API :

- POST /api/v1/command/add
- GET /api/v1/command/all
- POST /api/v1/command/trigger
- POST /api/v1/command/timed

Service Workflow

Add command diagram


http client
Command-API
Commands-DB
MQTT-broker
POST /api/v1/command/add
POST request with body {CommandType :(ON||OFF)}
alt
if err = bindJSON
{Code : 400 , Message : JSON bad format}
els if err = commandType != On or commandType != Off
{Code : 400 , Message : Use OFF or ON only}
els :
CreateCommandRecord()
Publish command

Database model

The command service store data in database called commands it contains 2 tables (Command(id,time,Email,CommandType(On/Off)) and the Threshold table (id,time,email,value,SensorType,Comm

4.2.5 Visualisation Service

Role

The visualisation service make use of both MQTT and Websocket protocols to perform real time data visualisation. The visualisation service expose one endpoint for handling websocket connection with the browser.
The visualisation API : GET /ws

WebSocket

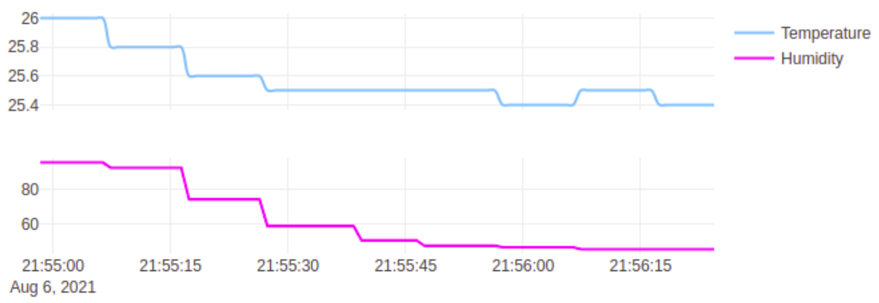
In the Background Chapter we mentioned the websocket protocol now we will dive in the challenge of how to send mqtt messages over websocket protocol ?
Using golang channels and goroutines can solve this by considering subscribing to topic and handling websocket connections two separate processes and passing between them the message from mqtt using channels.
code snippet :

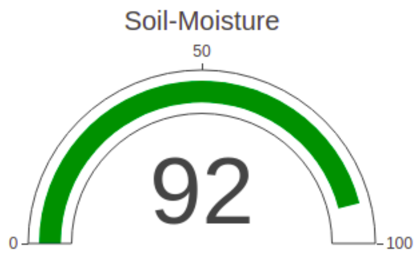
```
var eventChannel = make(chan model.Event)
func onMessageReceived() func(client mqtt.Client ,msg mqtt.Message) {
  return func(client mqtt.Client , msg mqtt.Message){
   switch msg.Topic() {
     case "esp/sensor" :
     event:= parseMessage(msg)
     // sending the event to the WsSocketEndPoint function over eventCh
     go func(){eventChannel <- event}()
       }
    }
 }
func WebSocketEndPoint(x *gin.Context) {
  socket , err := UpgradeConnection(x.Writer,x.Request)
  if err != nil {
    log.Println(err)
  }
  for {
     select {
     // when the eventChannel received the server write it to the
     // browser over websocket connection
     case event := <- eventChannel ;
          err = socket.WriteJSON(event)
          if err != nil {
          log.Println(err)
          }}}}
```

Using Plotly js

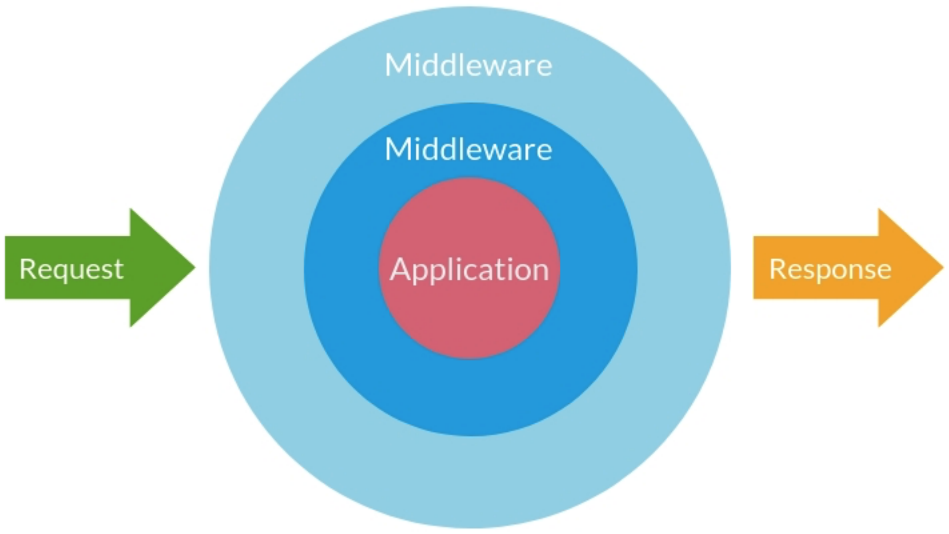
Plotly JavaScript Open Source Graphing Library Built on top of d3.js and stack.gl, Plotly.js is a high-level, declarative charting library. plotly.js ships with over 40 chart types, including 3D charts, statistical graphs, and SVG maps.

Results





4.3 Middlewares



Middleware is software which lies between an operating system and the applications running on it. Common middleware examples include database middleware, application server middleware, message-oriented middleware, web middleware and transaction-processing monitors.

4.3.1 Authentication Middleware

In each service we have to implement an authentication middleware for checking the user sesssion in the redis database and validating the jwt token .
example code for auth-middleware :

```
func AuthMiddleware() Http.HandlerFunction {
    return func(context Context) {
        s := sessions.Deafault(context)
        token := s.GET("token")
        if token == "" {
        context.JSON(StatusUnauthorized,"not auth")
        context.Abort()
        return
        }
        ...
        // if the token is valid the request will be
```

```
            // handled to the service handler
            // else will be an "no auth" message
            context.Next()
      }
}
```

4.3.2 MQTT client

Using a service that sub/pub to an MQTT broker needs an mqtt client we used Eclipse Paho MQTT Go client for establishing the connection between services and the message broker / publishing a message into message broker / and subscribing to a specefic topic. example for creating an mqtt client :

```
func NewMqttClient(host,port string) mqtt.Client {
  options := mqtt.NewClientOptions()
  options.AddBroker("%s:%s",host ,port)
  options.AutoReconnect = true
  client := mqtt.NewClient(options)
  if token := client.Connect(); token.Wait() && token.Err!=nil{
   panic(token.Err)
  }
  return client
}
```

4.4 Setup NGINX server as an API GateWay

4.4.1 NGINX/NGINX+

NGINX Multitude of use cases

One NGINX instance can be :

- Web server
- Reverse proxy
- Load balancer
- Cache
- API Gateway and more .

Supporting all cloud platforms and can run on bare metal like raspberry pi 3 b+ .

API GateWay vs API Management

API Management :

- Policy management .
- Analytics and Monitoring
- API Documentation

API Gateway ;

- Request Routing
- Authentication
- Rate Limiting
- Exception Handling

API Gateway Essential Functions

- TLS Termination
- Client Authentication
- Access-control
- Request Routing
- Rate Limiting
- Load Balancing

4.4.2 NGINX config

Install and run

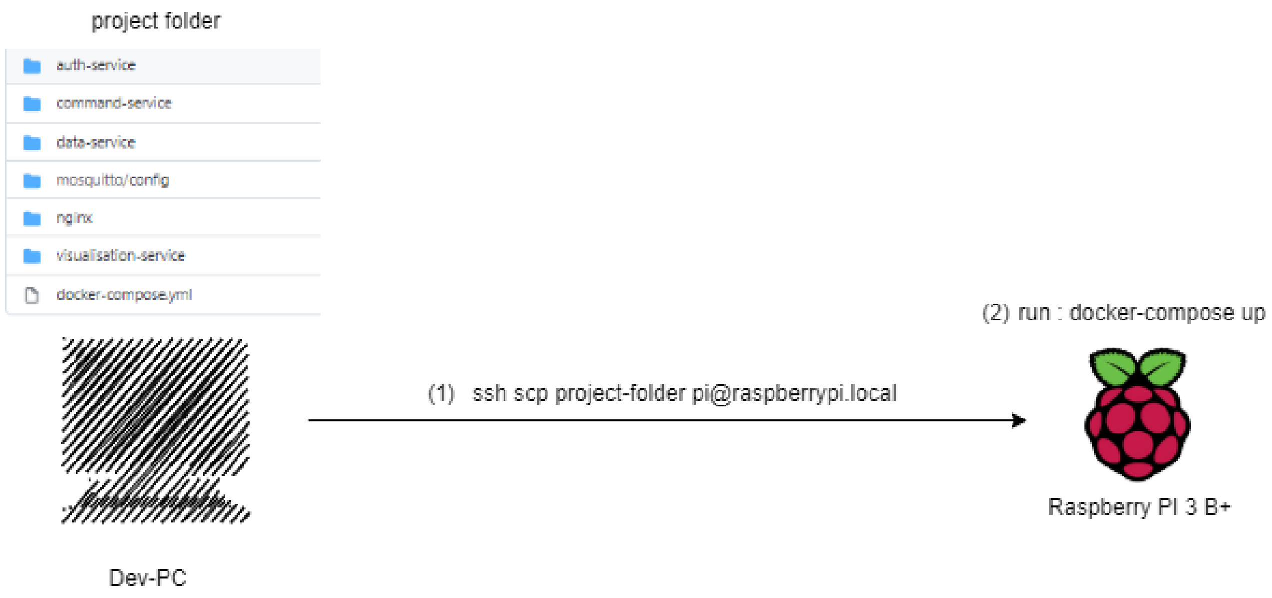
- Getting the official nginx image :
  - docker pull arm32v7/nginx
- Run nginx image :
  - docker run –net=host –name ngx -p 80:80 nginx
- Change the default config :
  - docker exec -it ngx bash
  - nano /etc/nginx/conf.d/default.conf

```
     error_page    500 502 503 504   /50x.html;
location = /50x.html {
    root    /usr/share/nginx/html;
}
location /graph/ {
proxy_pass http://localhost:5555/api/v1/visual/ ;
proxy_set_header     Host                $host;
proxy_set_header     X-Real-IP           $remote_addr;
proxy_set_header     X-Forwarded-For     $proxy_add_x_forwarded_for;
          . . .
}
```

5 Deployment

5.1 Deployment strategy

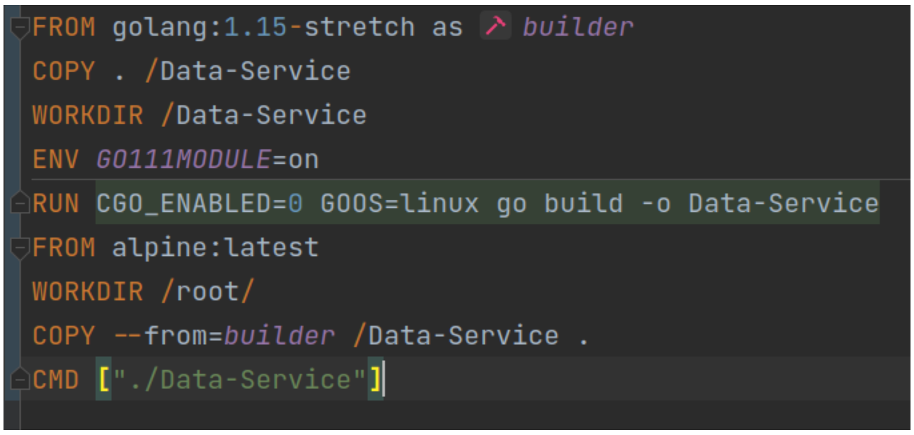
Using Docker and DockerCompose to run our services on the device.



5.2 Dockerizing our Services

5.2.1 Example :

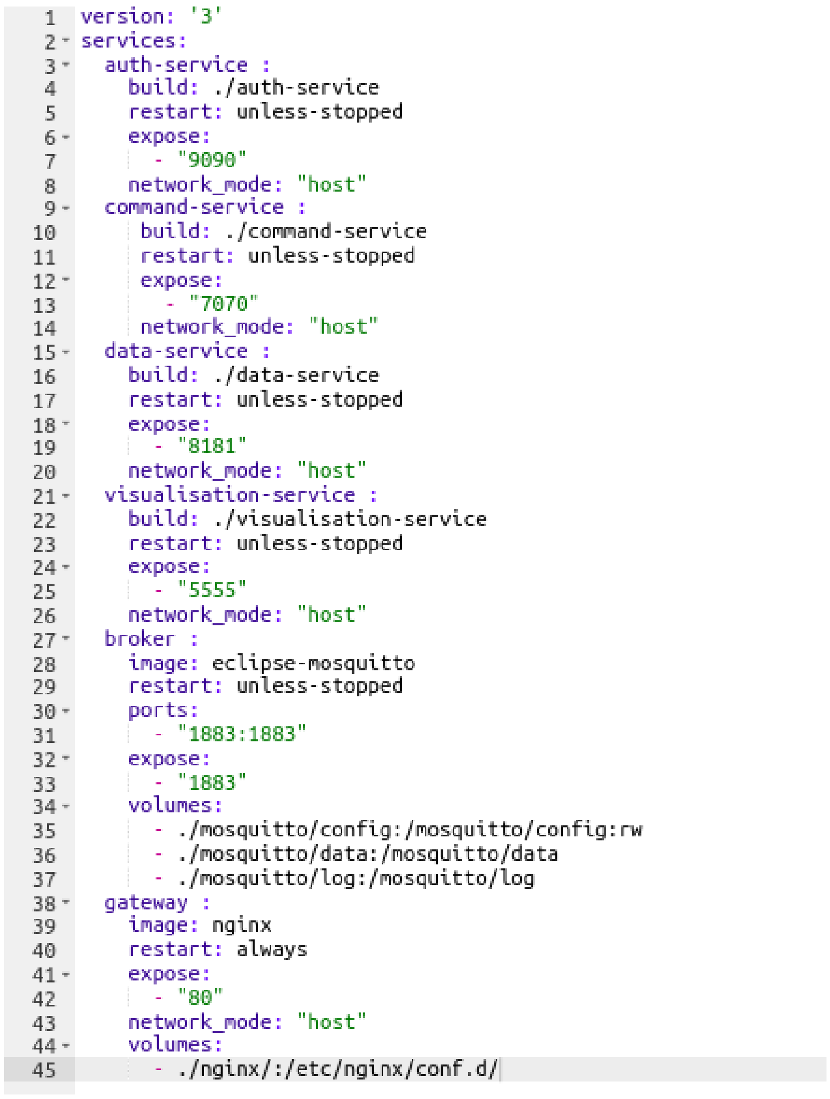
Building docker image using multi-stage build for optimizing our containers.



```
FROM golang:1.15-stretch as builder
COPY . /Data-Service
WORKDIR /Data-Service
ENV GO111MODULE=on
RUN CGO_ENABLED=0 GOOS=linux go build -o Data-Service
FROM alpine:latest
WORKDIR /root/
COPY --from=builder /Data-Service .
CMD ["./Data-Service"]
```

5.3 Using Docker-Compose to manage Service Containers

5.3.1 docker-compose.yaml :



```yaml
version: '3'
services:
  auth-service :
    build: ./auth-service
    restart: unless-stopped
    expose:
      - "9090"
    network_mode: "host"
  command-service :
     build: ./command-service
     restart: unless-stopped
     expose:
       - "7070"
     network_mode: "host"
  data-service :
    build: ./data-service
    restart: unless-stopped
    expose:
      - "8181"
    network_mode: "host"
  visualisation-service :
    build: ./visualisation-service
    restart: unless-stopped
    expose:
      - "5555"
    network_mode: "host"
  broker :
    image: eclipse-mosquitto
    restart: unless-stopped
    ports:
      - "1883:1883"
    expose:
      - "1883"
    volumes:
      - ./mosquitto/config:/mosquitto/config:rw
      - ./mosquitto/data:/mosquitto/data
      - ./mosquitto/log:/mosquitto/log
  gateway :
    image: nginx
    restart: always
    expose:
      - "80"
    network_mode: "host"
    volumes:
      - ./nginx/:/etc/nginx/conf.d/
```

5.4 Conclusion

Throughout this project we built a solid knowledge about iot/microservices/deployment and various protocols such as mqtt/ssh/websocket, but still faced some technical problems mainly in configuring the raspberry and connecting the nodemcu to our local network in addition of that security is one of the subject that we did not tackle in the project duo to the fact that using the TLS protocol needs a signed certificate from a legit authority in the internet and must of the internet browsers or clients like postman do not accept self signed certificates.

THE END