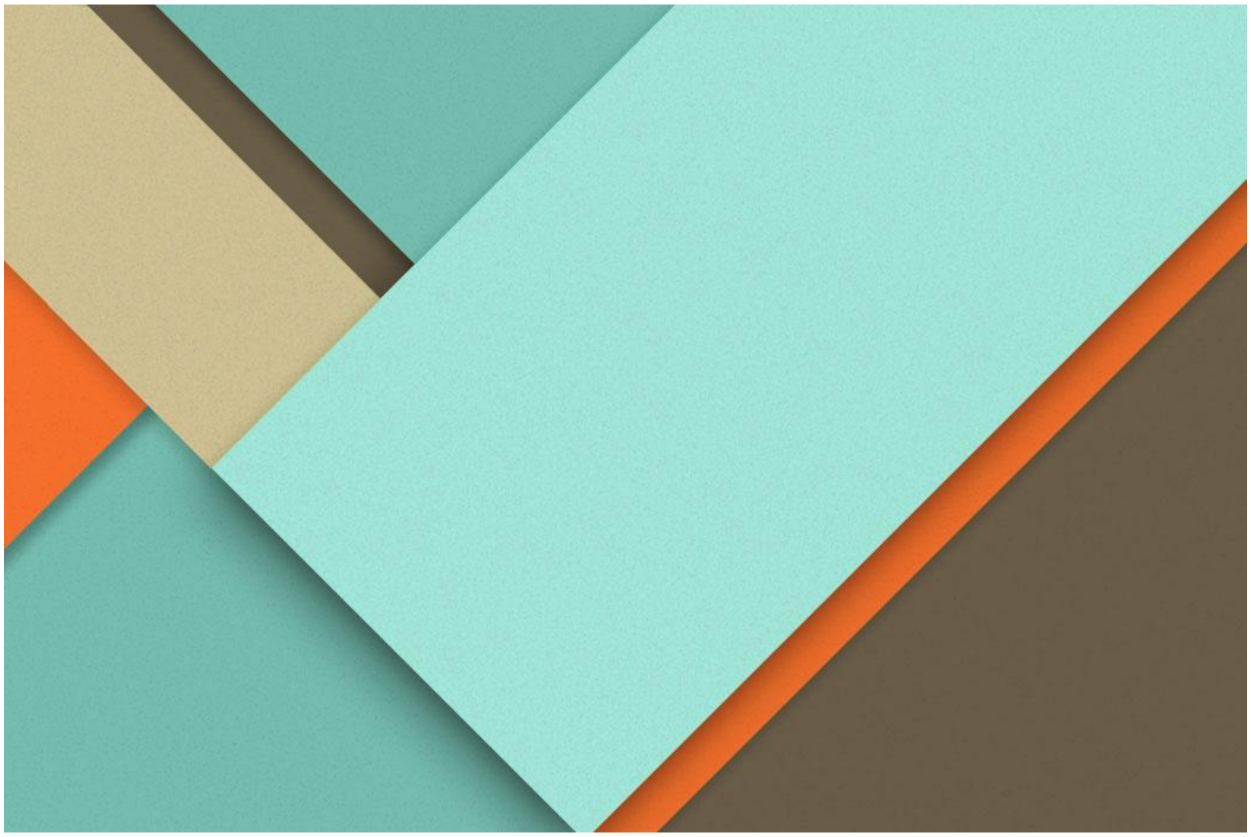


IoTManagementSystem

Technical Documentation

27.01.2023

—

Nauman Shakir
https://NaumanShakir.com
https://3STechLabs.com

Overview

This document is aimed at providing the technical documentation of the IoT Management System

Introduction

IoT Management System is a comprehensive solution for remotely managing and maintaining your Raspberry Pi devices. The platform is comprised of two main components: a client software written in Rust and a web application written in ReactJS.

The client software can be easily installed on your Raspberry Pi by running a single command and then linking the device to the web application by adding its MAC address. Communication between the web application and the client is handled through MQTT, with the broker hosted on the same server as the web application.

The web application provides a variety of features, such as sending files and firmware updates to the client, remotely executing shell commands, and monitoring logs. The web app architecture is made possible by the use of Docker containers for backend, frontend and database, managed by CapRover. "

Overview of the Raspberry Pi Remote Management SaaS app

The web app enables you to easily link your Raspberry Pi devices with the web application by adding their MAC addresses allowing fast and secure communication over MQTT.

The app also allows for remote script execution, over-the-air client firmware updates and multi-user support. Additionally, the app has a logging and reporting feature which can be used for troubleshooting, auditing, and reporting. The platform is designed to be easy to deploy and manage, with minimal setup and configuration required.

The web application also allows for the option of sending only file URLs to the client, allowing the client to download the files directly from the URL. This means you can easily share files and updates with all of your Raspberry Pi devices, no matter where they are located.

The web app also allows for sending the client firmware updates via URL links, such as Github release URLs or S3 links, making it easy to keep your Raspberry Pi devices up-to-date with the latest client firmware.

Overall, the remote Raspberry Pi management platform provides a user-friendly, centralized solution for managing and maintaining your Raspberry Pi devices. Whether you have one device or many, our platform makes it easy to keep them all running smoothly and efficiently.

Technologies Used

The web application components used in IoT Management Systems are

- ReactJS
- MongoDB
- NodeJS

While MQTT is used as a communication protocol and everything is working as a microserver inside a docker container. All of the containers are managed by CapRover.

Basic details of the stack of choice are given below:

1. ReactJS is a popular JavaScript library for building user interfaces and is well-suited for the front end of this web application because it allows for efficient and dynamic updates to the user interface. React's component-based architecture allows for easy maintenance and scalability of the codebase.
2. MongoDB is a popular NoSQL database that is well-suited for storing large amounts of data and is a good choice for this web application because it allows for easy scalability and flexibility in the data model.
3. NodeJS is a popular and efficient JavaScript runtime that is well-suited for the backend of this web application because it allows for fast and concurrent processing of requests.

Using a dockerized approach for each part of the web application is recommended because it allows for easy deployment, scaling and management of the application, and also the application becomes more portable.

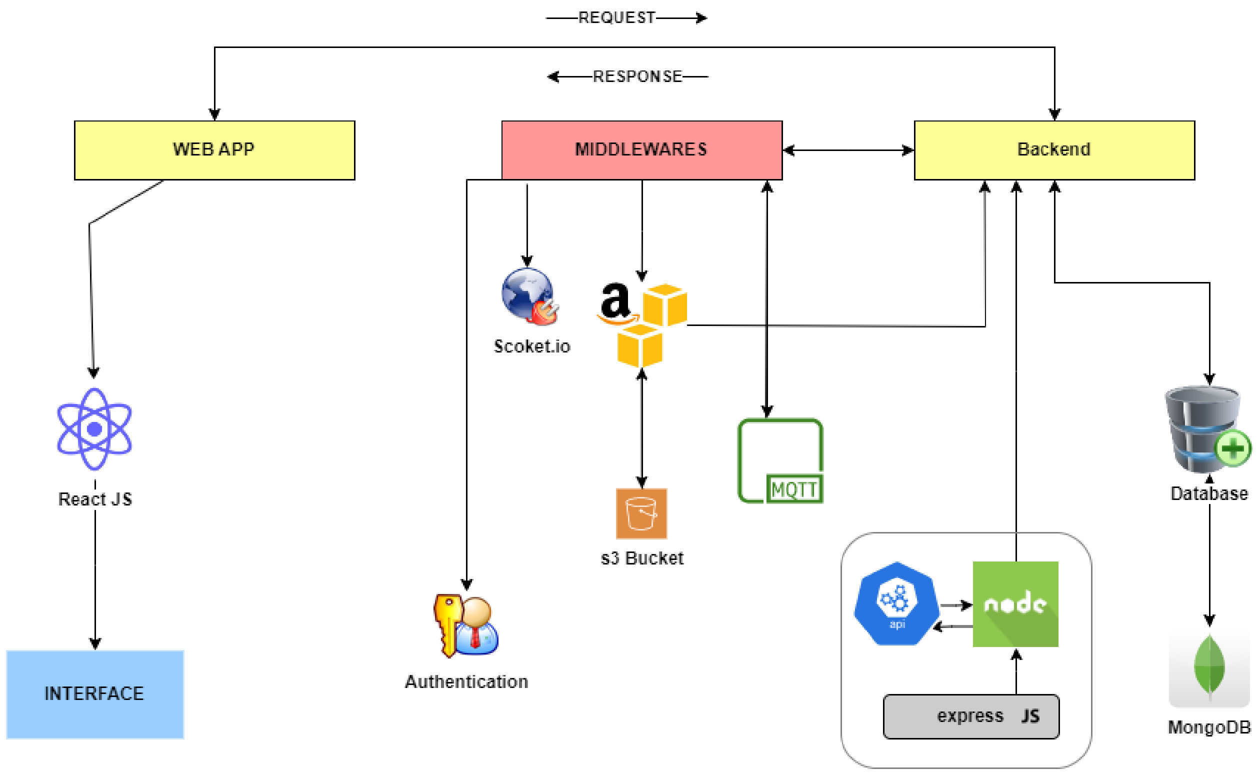
4. CapRover is a good choice for managing the containers because it makes it easy to deploy, scale and manage Docker containers, with a simple and user-friendly web interface.
5. MQTT is a lightweight and efficient messaging protocol that is well-suited for IoT systems, it is designed for low-bandwidth, high-latency networks, making it ideal for communication between IoT devices and the web application. It allows for efficient and reliable communication, with low overhead, and is easy to implement.

System Architecture

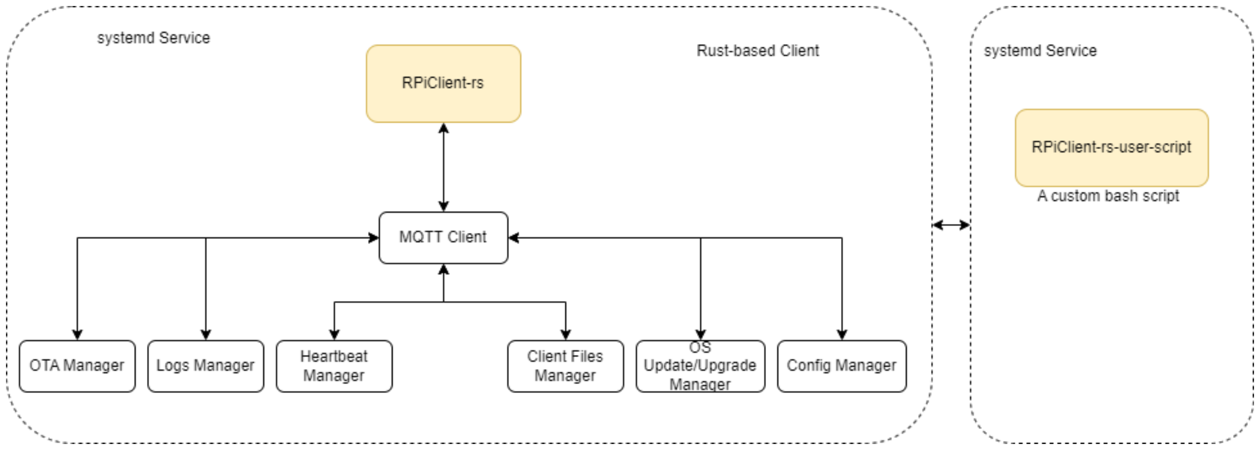
The system architecture for different parts of the system is explained in this section.

WebApp Architecture

The web app architecture diagram is shown below. The major middlewares used are S3, bucket, socket.io and MQTT. S3 is used for file management for client and user files.



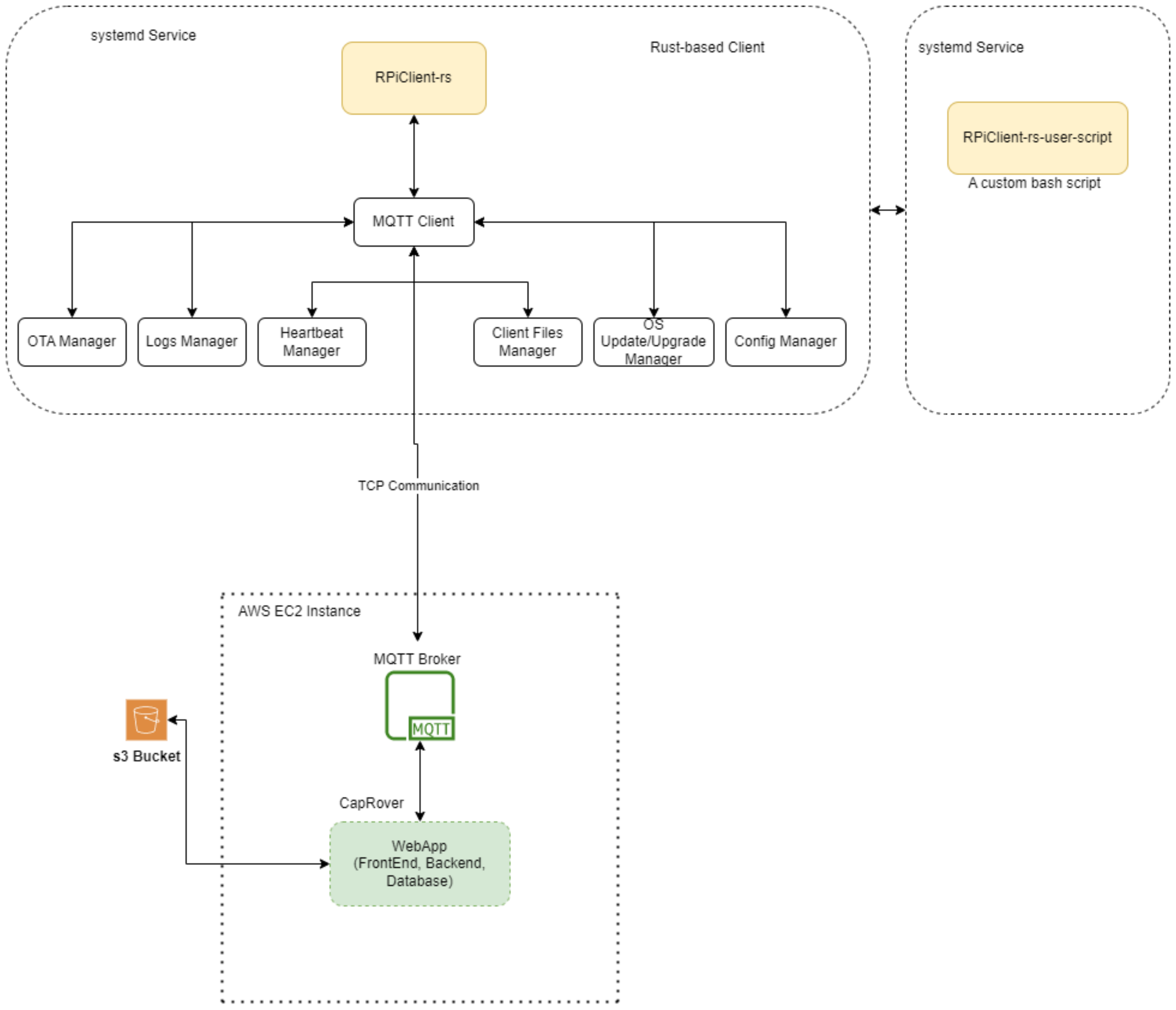
RPiClient Architecture



The RPiClient Architecture diagram is shown above. It is a trust-based client binary and have multiple parts as mentioned in the diagram. All of the parts get or publish data from the MQTT client.

The RPiClient-rs is running as a systemd service along with a RPiCleint-rs-user-script service which is nothing but a simple bash file editable by RPiClient-rs via web app.

System Architecture



The complete architecture of the system, as shown above, comprises two major parts. The first one is the RPiClient-rs client firmware running on the Raspberry Pi while the second part is the WebApp running on the EC2 instance. The communication protocol being used is MQTT. The MQTT broker, in this case, mosquitto is also hosted on the same EC2 instance.

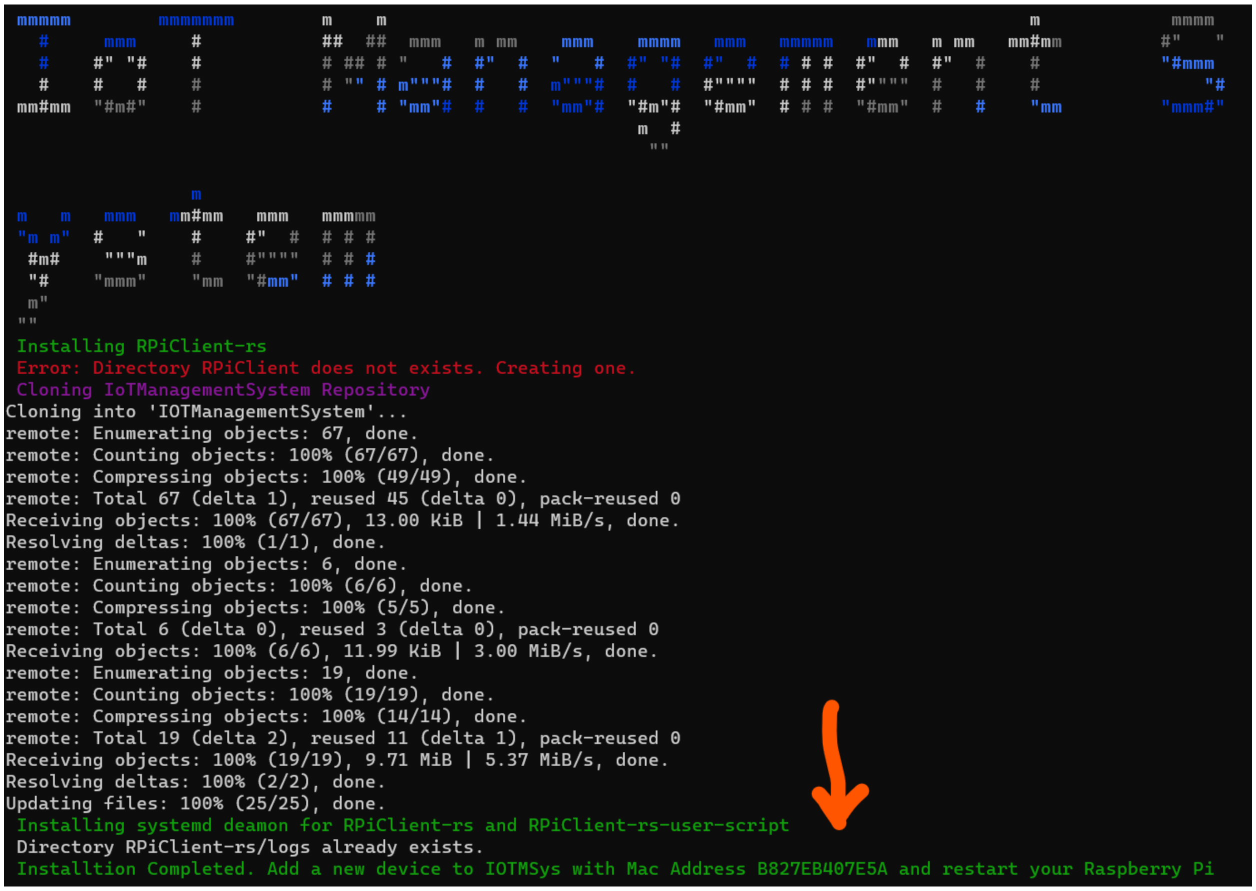
The web app docker containers are managed by CapRover.

System Usage

RPiClient-rs Installation

The system can be used by installing the RPiClient-rs Firmware on the Raspberry Pi using a one-click-installation command. The changes that RPiClient-rs installation script makes, can be fully rolled-back using one-click-uninstall command.

Once the installation script is finished installing the client, it presents the user with the related MAC address of the device as shown below.

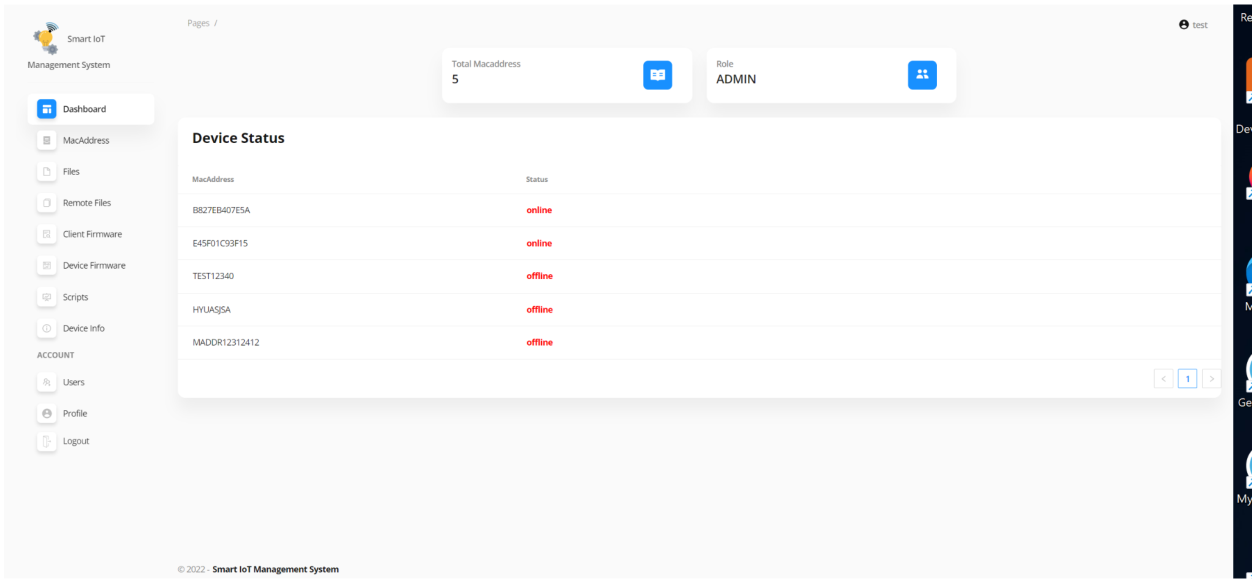


```
 mmmmm        mmmmmmm        m    m                                                            m            mmmm
   #     mmm     #           ##   ##   mmm   m mm    mmm    mmmm   mmm   mmmmm   mmm   m mm   mm#mm          #"    "
   #    #"  "#   #           # ## #  "   #  #"   #  "   #  #"  "# #"   # # # #  #"   # #"   #    #           "#mmm
   #    #    #   #           # "" # m"""#  #    #  m"""#  #    # #""""  # # #  #""""  #    #    #              "#
 mm#mm  "#m#"    #           #    # "mm"#  #    #  "mm"#  "#m"#  "#mm"  # # #  "#mm"  #    #    "mm         "mmm#"
                                                           m  #
                                                            ""

                 m
 m   m   mmm   mm#mm   mmm    mmmmm
 "m m"  #   "    #    #"  #   # # #
  #m#    """m    #    #""""   # # #
  "#    "mmm"    "mm  "#mm"   # # #
  m"
 ""
 Installing RPiClient-rs
 Error: Directory RPiClient does not exists. Creating one.
 Cloning IoTManagementSystem Repository
Cloning into 'IOTManagementSystem'...
remote: Enumerating objects: 67, done.
remote: Counting objects: 100% (67/67), done.
remote: Compressing objects: 100% (49/49), done.
remote: Total 67 (delta 1), reused 45 (delta 0), pack-reused 0
Receiving objects: 100% (67/67), 13.00 KiB | 1.44 MiB/s, done.
Resolving deltas: 100% (1/1), done.
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 6 (delta 0), reused 3 (delta 0), pack-reused 0
Receiving objects: 100% (6/6), 11.99 KiB | 3.00 MiB/s, done.
remote: Enumerating objects: 19, done.
remote: Counting objects: 100% (19/19), done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 19 (delta 2), reused 11 (delta 1), pack-reused 0
Receiving objects: 100% (19/19), 9.71 MiB | 5.37 MiB/s, done.
Resolving deltas: 100% (2/2), done.
Updating files: 100% (25/25), done.
 Installing systemd deamon for RPiClient-rs and RPiClient-rs-user-script
 Directory RPiClient-rs/logs already exists.
 Installtion Completed. Add a new device to IOTMSys with Mac Address B827EB407E5A and restart your Raspberry Pi
```

Dashboard

Once, the client firmware is installed on the Raspberry Pi, the IoTManagement Dashboard automatically displays the added devices and their connectivity status:



Total Macaddress 5

Role ADMIN

Device Status

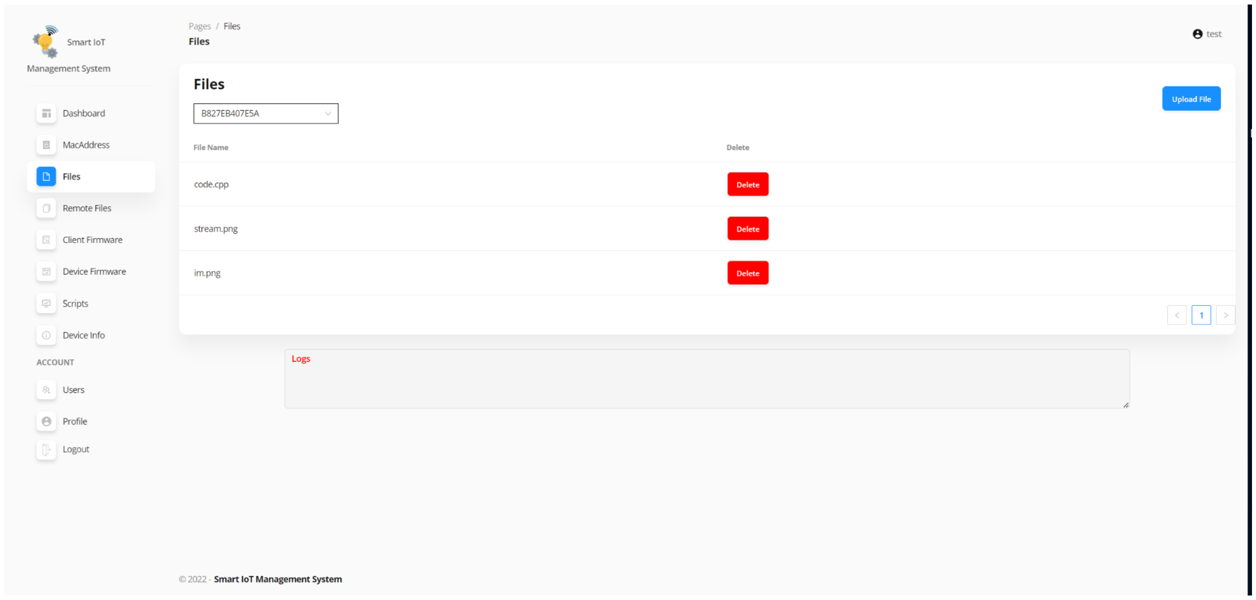
MacAddress	Status
B827EB407E5A	online
E45F01C93F15	online
TEST12340	offline
HYUASJSA	offline
MADDR12312412	offline

© 2022 - **Smart IoT Management System**

Users can link the Raspberry Pi to the dashboard in the MacAddress tab. Only the linked devices can be remotely controlled using the dashboard.

Features

Files Management



Files

B827EB407E5A

File Name	Delete
code.cpp	Delete
stream.png	Delete
im.png	Delete

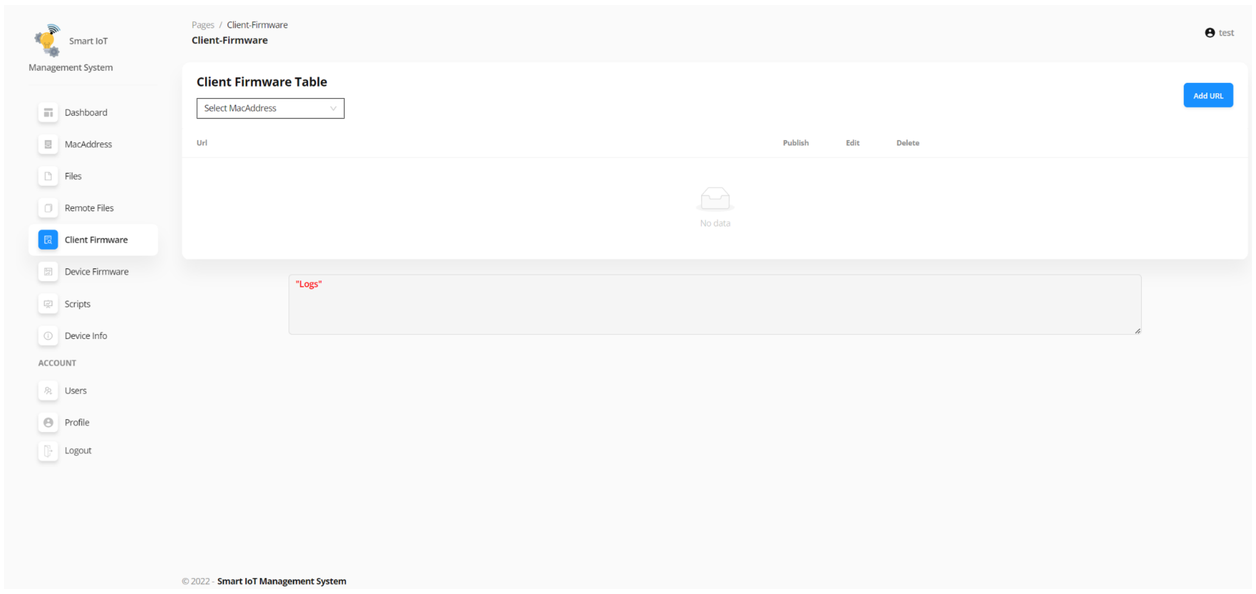
Logs

© 2022 - **Smart IoT Management System**

Files can be sent to the Raspberry Pi over MQTT in the files tab while the files can also be sent to the Raspberry Pi using the file URL in the Remote Files tab.

Client Firmware OTA Update

The client firmware can be updated over the air by just providing the Firmware tar file URL. In this case, it could be a GitHub release URL.



Client Firmware Table

Select MacAddress

Url	Publish	Edit	Delete

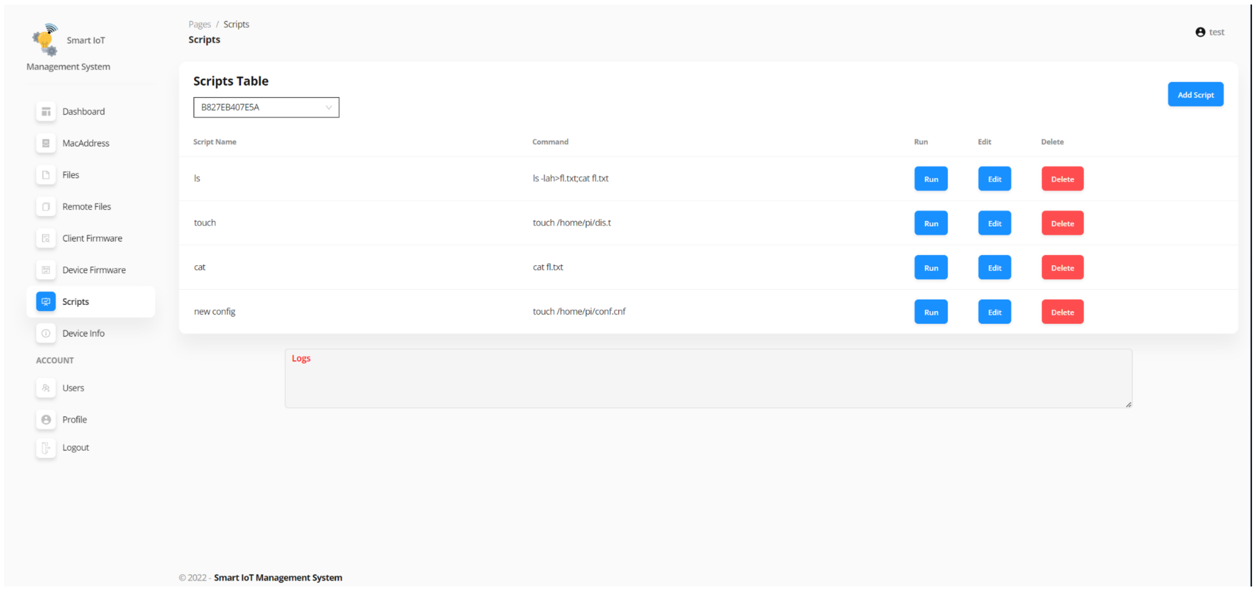
No data

"Logs"

© 2022 - **Smart IoT Management System**

Remote Bash Scripts

Users can add/remove/edit and execute bash scripts using the Scripts tab in the web app.



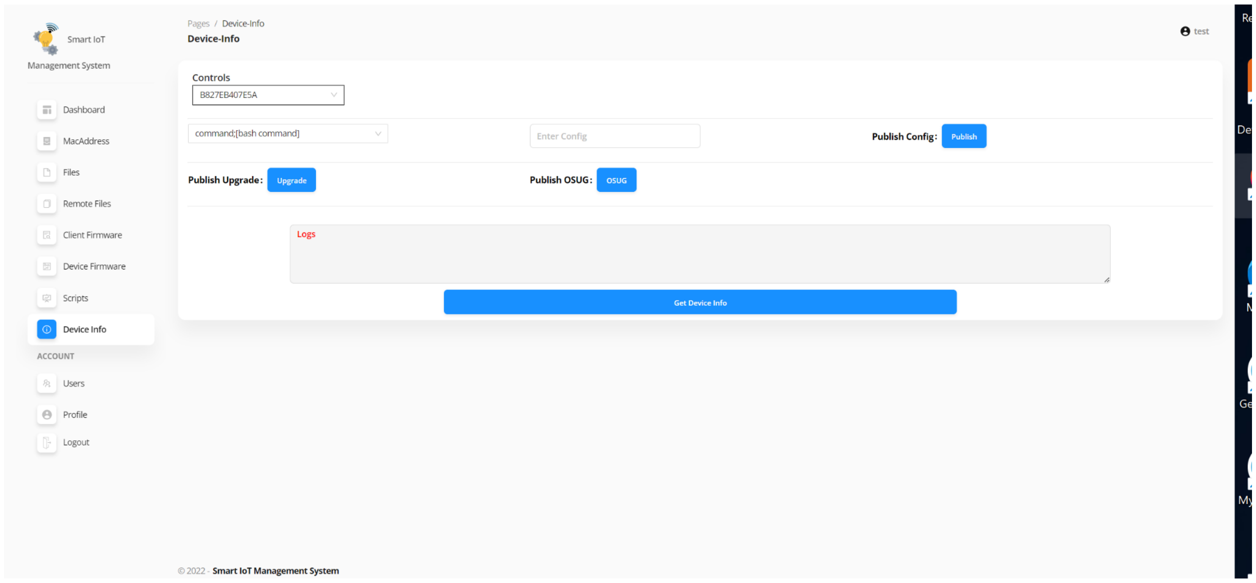
Scripts Table

B827EB407E5A

Script Name	Command	Run	Edit	Delete
ls	ls -lah>fl.txt;cat fl.txt	Run	Edit	Delete
touch	touch /home/pi/dis.t	Run	Edit	Delete
cat	cat fl.txt	Run	Edit	Delete
new config	touch /home/pi/conf.cnf	Run	Edit	Delete

Device Management

The Device Info tab allows running bash commands, updating or upgrading the OS and getting the different logs with different logs level i.e, stdout, and stderr for both RPiClient-rs and RPiClient-rs-user-script.



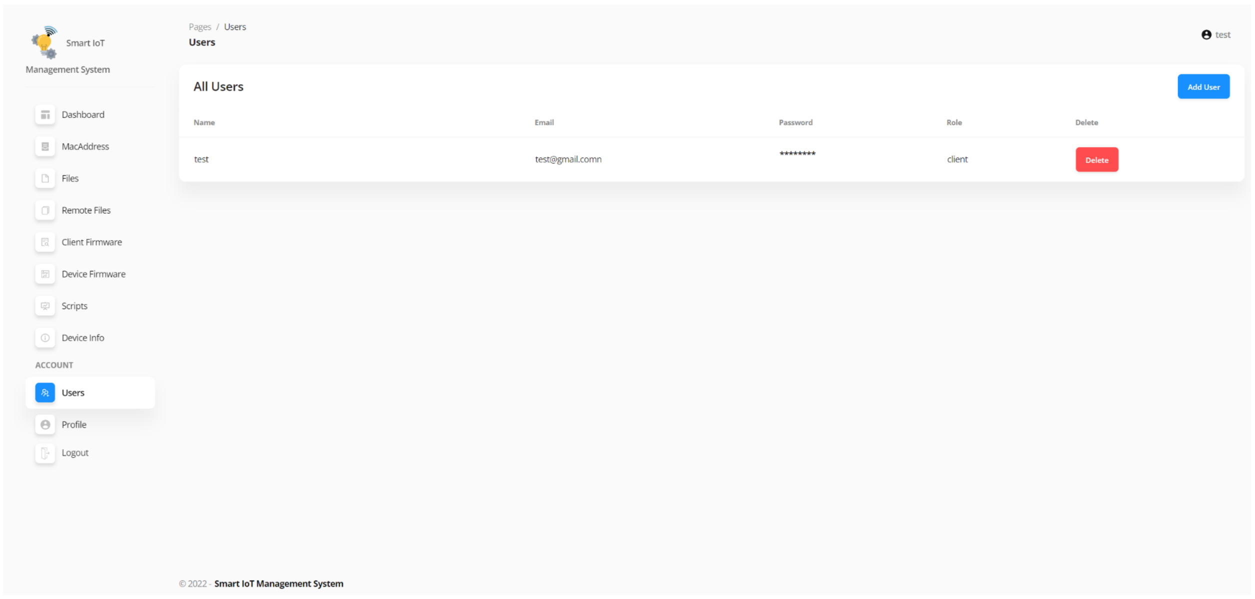
Logs

Each tab shows the Logs field at the bottom and shows the logs of the executed event.

```
"Logs"
```

Users Management

The admin account can invite users to use the IoTManagementSystem web app using the Users tab.



All Users

Name	Email	Password	Role	Delete
test	test@gmail.comn	********	client	Delete

Please note that it is an invite-only system and users can't signup on their own.

Technical Details and Playbook

The technical details/Playbook is present in the GitHub repository of IoTManagement System and it covers multiple things including

- RPiClient Installation
- Server Details
- MQTT Topic Details
- API Details
- Usage
- Test

RPiClient-rs Program

The RPiClient-rs is summarized below.

MQTT Client Connection

This Rust-based RPiClient-rs program is an MQTT client that connects to an MQTT server specified by the first command line argument, or ***"tcp://50.19.43.139:1883"*** if no argument is provided. The client is initialized with a unique ID that is created using the client's MAC address. The program then sets up a connection to the MQTT server using the specified options, including a keep-alive interval, MQTT version, clean session, will message, username, and password.

MQTT Client Topics

Once connected, the program subscribes to various MQTT topics related to the device's OS, firmware, and configuration. It then enters a loop where it waits for incoming messages and prints them to the console. Additionally, it spawns two additional tasks: one that publishes messages to an "info" topic, and another that sends a heartbeat message to the server.

MQTT Client System Topics

When the device receives a message on the topic ***"iotm-sys/device/firmware/file"*** and the topic contains the device's MAC address, it will process the message as a firmware file. It will extract the filename from the message payload, which is in the format of ***"url;filename,"*** and create a new file with that name in the "data" directory. It then uses the "FromBase64Writer" struct to decode the data in the payload and write it to the new file.

When the device receives a message on the topic ***"iotm-sys/device/firmware/file/all,"*** it will process the message similar to the first topic, but it doesn't check for the MAC address in the topic.

When the device receives a message on the topic ***"iotm-sys/device/firmware/script"*** and the topic contains the device's MAC address, it will process the message as a firmware script. It will extract the script from the message payload and create a new file called "user-script.sh" with the script as its content. It then uses the "Command" struct to execute the command to restart the "RPiClient-rs-user-script" service.

After all of this processing, the device will publish a log message containing the output of the commands executed and the MAC address to the topic ***"iotm-sys/device/logs/[MAC address]."***

The topic ***"iotm-sys/device/client/url"*** and ***"iotm-sys/device/client/url/all"*** which are related to updating the device firmware. When a message is received on these topics, it extracts the payload of the message, which is expected to be a URL, and assigns it to the variable "data_link". The variable "flname" is assigned the file path ***"/home/pi/RPiClient-rs/RPiClient-rs.tar"***. The function "perform_ota" is then called with these two variables as arguments. This function is likely responsible for downloading the firmware update from the specified URL and saving it to the specified file path.

After the firmware update is downloaded, a new message is published on the topic ***"iotm-sys/device/logs/{MAC}"*** with the payload "New updates downloaded.". Then the "pwd" command is executed and the output is displayed on the console.

If the topic is ***"iotm-sys/device/client/url/all"*** the same process is followed but this time it will update the firmware for all devices.

MQTT Client Configuration Topics

The last part is handling the case where the message received on the MQTT topic contains the string ***"iotm-sys/device/config/"***, and also contains the MAC address of the device. If the message payload contains the string "command", it is extracting the command and arguments from the payload, executes the command using the Command struct from the std::process module, and capturing the output. It then creates an instance of the Logger struct with the MAC address and output and serializes it to JSON, and publishes it to the ***topic "iotm-sys/device/logs/" + the device's MAC address.***

If the payload contains the string ***"logs=stdout"*** it reads the file located at ***"/home/pi/RPiClient-rs/logs/stdout.log"*** and publishes it to the topic ***"iotm-sys/device/logs/" + the device's MAC address.***

If the payload contains the string ***"logs=stdout-user-script"*** it reads the file located at ***"/home/pi/RPiClient-rs/logs/stdout-uscript.log"*** and publishes it to the topic ***"iotm-sys/device/logs/" + the device's MAC address***.

If the payload contains the string ***"logs=stderr"*** it reads the file located at ***"/home/pi/RPiClient-rs/logs/stderr.log"*** and publishes it to the topic ***"iotm-sys/device/logs/" + the device's MAC address.***

If the payload contains the string ***"logs=stderr-user-script"*** it reads the file located at ***"/home/pi/RPiClient-rs/logs/stderr-uscript.log"*** and publishes it to the topic ***"iotm-sys/device/logs/" + the device's MAC address.***

Technical Decisions

Why Rust for RPiClient-rs?

RPiClient earlier release was based on Python3. While it worked well for the POC but there are certain challenges while using Python3

- The client program slows down at times making the program unresponsive.
- Memory management issues.
- Can't create release binaries as a native language thing.

The performance of a program written in Python compared to the same program written in Rust can vary depending on the specific program and the requirements of the task. However, in general, Rust programs tend to be faster and more efficient than Python programs due to the following reasons:

1. Compilation: Rust programs are compiled to machine code, making them faster to execute than Python programs that are interpreted.
2. Memory management: Rust provides automatic memory management, which results in programs that are more memory-efficient and less prone to memory leaks and data races compared to Python programs that use manual memory management.
3. Type inference: Rust has a strong type system with type inference, making it possible to catch type-related errors at compile-time, which can prevent bugs and improve performance compared to Python's dynamically-typed language.
4. Concurrency: Rust provides built-in support for concurrent programming, making it easy to write efficient and performant code that can take advantage of multiple cores.

In IoTManagementSystem, the key features required were fast execution, good memory management, and concurrent programming.

The obvious choice for RPiClient was C/C++ or Rust. C++ comes with its own set of issues as mentioned below:

1. Memory management: No automatic memory management, prone to memory leaks, and data races
2. Null pointers: No null pointer protection, often leading to crashes and security vulnerabilities
3. Type safety: Weak type system, leading to type mismatches and type-casting issues
4. Object-Oriented Design: Complex OOP model with multiple inheritances, implicit type conversion, and hard-to-understand class hierarchy

Here is a simple example to illustrate the difference in memory management between C++ and Rust:

C++

```
#include <iostream>
using namespace std;

int main() {
  int *ptr = new int[10]; // dynamically allocate an array of 10 integers
  ptr[10] = 42; // this will cause an error, index 10 is out of bounds
  delete[] ptr; // must delete the memory manually
  return 0;
}
```

Rust

```
fn main() {
  let arr = [0; 10]; // statically allocate an array of 10 integers
  let index = 10;
  let _ = arr[index]; // this will cause a compile-time error, index 10 is
out of bounds
}
```

In C++, the code dynamically allocates an array of 10 integers using the new keyword. However, it is possible to access an out-of-bounds index, which can cause a runtime error. The code must manually delete the memory using the delete[] keyword.

In Rust, the code statically allocates an array of 10 integers using the [0; 10] syntax. Rust has a strong type system and will catch the out-of-bounds error at compile-time, making it much harder to introduce runtime errors. Rust also automatically manages memory and frees it when it is no longer in use.

In short, Rust natively overcomes these issues making the program more reliable and a good choice for a system like IoTManagement System.

Rust is a good choice for small single-board computers (SBCs) like the Raspberry Pi for the following reasons:

1. Memory safety: Rust provides automatic memory management and prevents common memory-related errors like null pointers and buffer overflows, ensuring that your program is stable and secure.
2. Performance: Rust is a low-level language and can be used to write efficient and performant code, which is important for small SBCs that have limited resources.
3. Portability: Rust's cross-platform support makes it possible to write code that can run on a variety of platforms, including the Raspberry Pi, without modification.
4. Concurrency: Rust provides built-in support for concurrent programming, making it easy to write programs that can take advantage of multiple cores and run multiple tasks simultaneously. This is important for small SBCs that often run multiple services at once.
5. Ecosystem: Rust has a growing and supportive community that provides a wealth of resources, including libraries and tools, to help developers get started quickly and easily.

Overall, Rust's focus on memory safety, performance, portability, concurrency, and ecosystem make it a good choice for small SBCs like the Raspberry Pi, where security and efficiency are important considerations.

Conclusion

Summary

In conclusion, our remote Raspberry Pi management platform is a powerful and versatile tool for managing and maintaining your Raspberry Pi devices remotely. It is composed of two main components: client software written in Rust and a web application written in ReactJS. The client software is installed on the Raspberry Pi devices and the web application is hosted on a server.

The platform offers a wide range of features such as sending files, firmware updates, and commands, as well as real-time monitoring of the performance and resource usage of your Raspberry Pi devices. It also allows for scheduled tasks, multi-user support, and logging and reporting.

The choice of technologies used in the platform such as ReactJS for frontend, MongoDB for the database, NodeJS for backend, Docker for containerization, and CapRover for container management is well thought out. These technologies are well suited for the task at hand and provide efficient, user-friendly, and reliable solutions. In addition, MQTT is a lightweight and efficient messaging protocol well suited for IoT systems and provide efficient and reliable communication between IoT devices and web application.

Future Enhancements

Future enchantment to the system can include

1. Terminal Access
    a. Full-fledged terminal access to the Raspberry Pi from the dashboard over a secure TCP tunnel.
2. Device Monitoring and Basic Controls
    a. Device Management and Device Info functions on the Dashboard(restart, reboot, network status, free RAM/Diskspace)
3. RPiClient Optimizations
    a. Code readability improvements.
    b. Firmware optimizations.
4. Logs Formatting
    a. Better formatted logs on the dashboard.
5. Dashboard UI Improvements
    a. Organized tabs on the dashboard with tooltips/help on the sidebar.