

Table of Contents

Analysis	3
Problem Identification	3
Stakeholders.....	3
Computation Solution	3
Problem Recognition.....	3
Problem decomposition.....	3
Research.....	4
Similar Solutions	4
Features	4
Initial Concept	4
Limitations of the solution	5
Requirements.....	6
Software & Hardware	6
Software Requirements	6
Design	6
Functionality	6
Success Criteria	7
Design	8
User Interface	8
Website Interface.....	8
Device Interface	10
Hardware Setup	13
Hardware Listening mode.....	10
Website dashboard	11
Algorithms.....	12
Configuration	13
Registration	15
Handling server commands	14
Usage of libraries	15
Data Structures	17
Classes.....	17
Testing Method	18
Iterative Development	18

Development and testing.....	19
Arduino Firmware	19
Stage 1: Configuring the device into a default state.....	19
Bibliography	18

1. Analysis

1.1. Problem Identification

Currently there are few canteen payment systems that are easily accessible without needing to pay a large amount for a bespoke design. These systems benefit in a multitude of ways, such as allowing you to easily keep track of how much you are spending at lunch as well as letting you restrict which items your child could buy in a school setting to help with their diet. In addition to this, it can help reduce school / workplace workload by providing revenue reports that can be exported. It also mitigates the need to bring cash into your workplace/school. Your card can be topped up from a dashboard, has spending limit alerts and an auto-top up feature.

1.2. Stakeholders

The clients for this software and hardware could be schools or any small-medium-sized company which offers beverages and/or food on-site. It is particularly good for such companies who don't have the budget to spend on bespoke solutions, as the device comes at an affordable price point.

1.3. Computation Solution

1.3.1. Problem Recognition

This problem clearly lends itself to a computational solution as it automates the payment process. In a simple form, the card reader captures the ID of the card then finds it in a database (DB) and can charge it. Please note that this is highly abstracted, and many processes happen in between the fetch from the DB. This is a repetitive and mundane process and prone to human error; by using a computational solution, one can remove such errors and provide improved security and consistency of application process. There are many advantages to using a computation solution, for example an administrator can now easily search up all transactions, monitor revenue and remotely top up users' cards.

1.3.2. Problem decomposition

The process is easily broken down into individual steps:

1. User connects Arduino to Wi-Fi.
2. Device sends first-time online status to the backend server, which receives a code and displays the code and the domain/URL on the LCD screen allowing user to create an account.
3. User registers the device to the corresponding account using the displayed code, device is ready to start normal operation.
4. Wait for client (e.g. a cashier) to request payment, user then taps their RFID card to a reader.
5. Send the RFID card UID to the backend API (Application Programming Interface) with the amount to be charged.
6. Update the amount in the database.

1.4. Research

1.4.1. Similar Solutions

Tucasi

(Tucasi, n.d.)

Overview:

Tucasi is an online payment system designed for schools. It has many different modules that assist in payment related tasks. The one that links itself to the solution is the dinner money module. It implements many of the features such as diet management, balance alerts and the ability to pre-order your meal choices by providing an online food menu.

Features that I can apply to the solution:

Tucasi is like the solution however it also comes with many other modules that the client may not be interested in and therefore needs to pay extra for to just gain access to the single module. One feature that I liked about it was the ability for parents to choose which food their children can purchase which can encourage healthy eating, this could be implemented in a post-release update as it is not essential to provide a working solution.

Canteen System Concept (Hawas Design)

(Hawas Design, n.d.)

Overview:

This is a design concept which shares many features with the solution. It includes a dashboard for user management and allows users to top-up their children's cards and restrict their diet choices.

Features that I can apply to the solution:

Many of the features are already in the solution, allowing the users to easily top-up and send balance alerts as well as diet restrictions. The key features I will implement from this is user management and topping up children's cards.

1.5. Features

1.5.1. Initial Concept

The solution will involve using an Arduino for the hardware side of the project to communicate with RFID readers, a LED screen to display information to the user and a cryptographic module in which all the respective operations happen. For the front end of the project, I will create a web dashboard to easily add and manage these devices. It also allows individual users to top-up their cards and set up notifications for balance management. I also wanted to focus on security and thus implemented numerous security features throughout the program.

As a part of any application involving payments, security is paramount. As such I will make use of the ATECC508A crypto IC (integrated circuit) which allows for data to be securely signed externally on an isolated IC made for this purpose. The device itself prevents private key exposure and has specific hardware which accelerates the heavy mathematical operations involved in cryptography. For its use in respect to this project is to ensure the client (Arduino) can identify itself to the server with authenticity so the server can fully trust the clients it communicates with.

1.5.2. Limitations of the solution

While I have made the best possible effort to mitigate some known security vulnerabilities there are some which are unable to fully lessen. These include the RFID cards. The cryptography algorithm used by the RFID cards was broken in 2008 (Nohl, et al., 2008), so it is easy to clone a card. The reason for not using the updated, secure version of RFID technology is that it is under NDA (non-disclosure agreement) and not available for individuals/students however it is assured that in any commercial implementation the more secure version could easily be used.

1.6. Requirements

1.6.1. Software & Hardware

Hardware

- **ESP-12E Wi-Fi Module** – This is the main component of the device, the “computer” in which all other components interact to get their orders from.
- **MFRC522 RFID Module** – This module interacts with the RFID cards and accesses their internal storage.
- **TFT 2.4-inch screen** – This is simply a colour display that shows information to the user during the payment process.
- **ATECC508A** – This Integrated Circuit (IC) is where all the cryptographic operations happen as well as secure key storage.
- **Wi-Fi Enabled PC** – This is for accessing the web dashboard for the end-user to interact with

Software

- **Any operating system with a browser** – To access the web dashboard.
- **Arduino IDE** – Used for the development of the Arduino programs.
- **Visual Studio Code** – Used for developing the backend API and frontend dashboard.
- **Typescript** – A programming language that includes static typing that then transpiles (converts) to JavaScript.

1.7. Software Requirements

1.7.1. Design

Requirement	Explanation
Simple device and web interface	Needs to be easy to use with minimal training so anyone of any age can use it
Connect to Wi-Fi easily	The solution makes use of a Wi-Fi captive portal to capture a networks credentials and guides the user through the process, increasing ease of use.
Device is cheap to produce	Device can be bought by anyone, only costing £15 pounds to buy all the parts.
Simple setup process	Much of the setup is automated, the only manual parts are connecting to Wi-Fi and linking the device to your account.

1.7.2. Functionality

Requirement	Explanation
RFID reader reads card in <1s	The device must be responsive, or users can get easily frustrated
A users card's balance can be remotely controlled	Allows administrators to set a balance to use for the day as well as allowing parents to top up remotely
Secure transmission and storage of data	In a system involving payments, security must be paramount.

1.7.3. Success Criteria

Criteria	How to evidence
Device connects to any Wi-Fi network in good conditions without error and can maintain the connection	Show device connecting with the backend API over an internet connection.
Device can capture RFID card UID and data in a short time (10 ms)	Hold the card over the device until a beep is heard, this shows a good read / write.
Device communicates to server in a secure manner	Request body contains signatures to allow the server to verify incoming data.
Device has understandable UI interface	Pictures of the UI and explanation
Device has an easy initial setup flow	Screenshots showing the setup flow
Web backend charges the RFID card in the database	Screenshots before / after database
Web dashboard allows users to manage the device remotely	Screenshot showing management flow
Web dashboard is easy to navigate	Screenshot showing web dashboard design
Web dashboard allows users to set daily balance	Screenshot showing card management screen
Web dashboard has an authentication system that allow allows authenticated users to manage their devices	Screenshots showing login screen & authentication code

2. Design

2.1. User Interface

Using a mock user interface creator, I made the following interfaces to showcase how the dashboard would look.

2.1.1. Website Interface

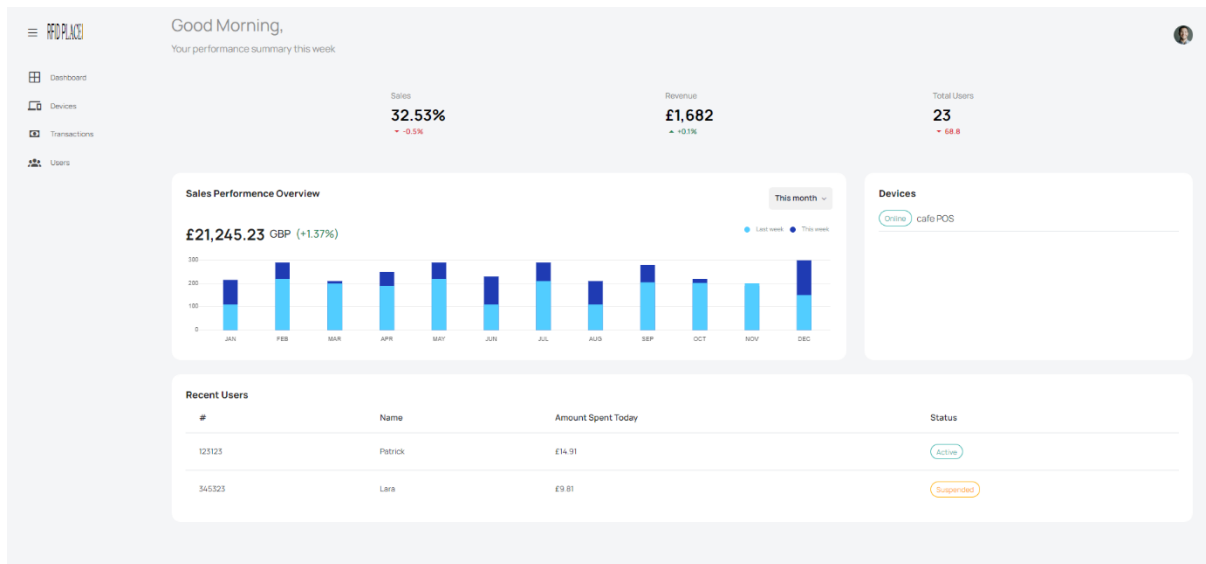


Fig 1, Main Dashboard

In Fig 1 all the main information is shown briefly to allow the user to quickly get a picture of what is going on in their system. On the right, it shows the devices and their status, this allows the user to diagnose any problems. The bottom table shows users who recently made purchases. Finally, the small graph gives an overview of their sales performance.

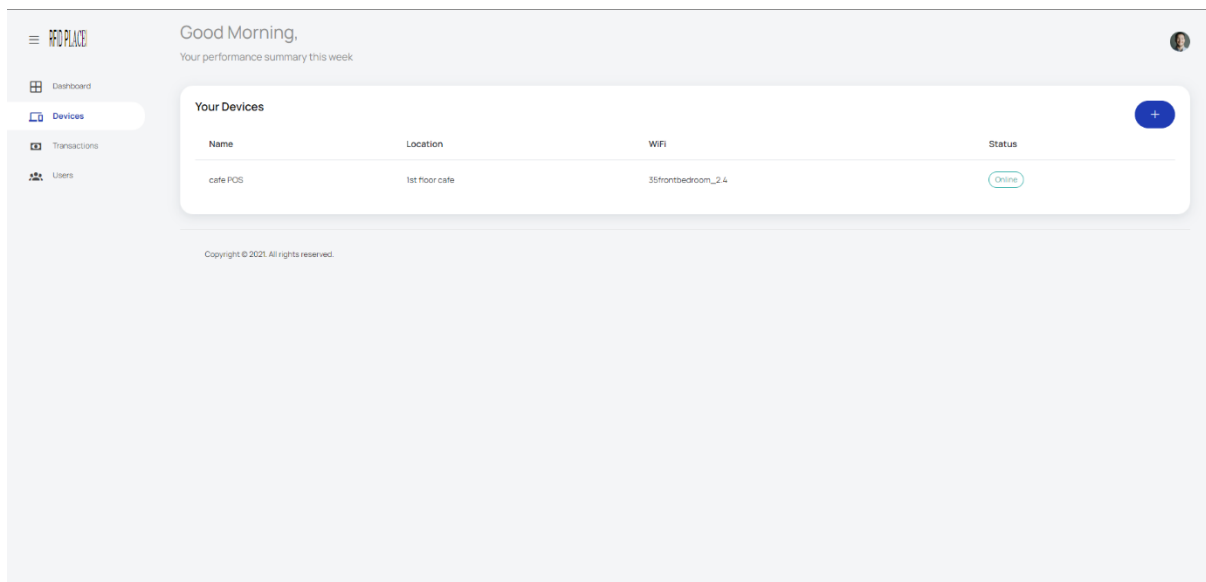


Fig 2, Devices Module

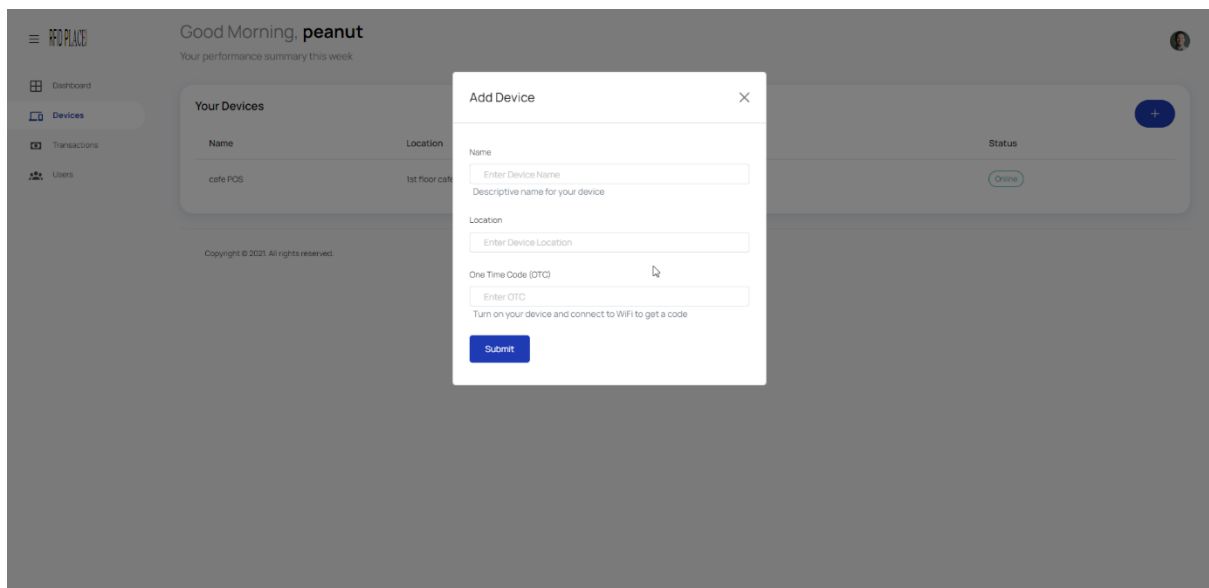


Fig 3, Add Device Modal

Fig 2 shows the devices module which allows users to fully manage their devices, showing its status, connected Wi-Fi, location, and unique name. Upon pressing the plus button, a form will appear which allows users to enter a code given to them via the card reader to register the device to their account, as shown in Fig 3.

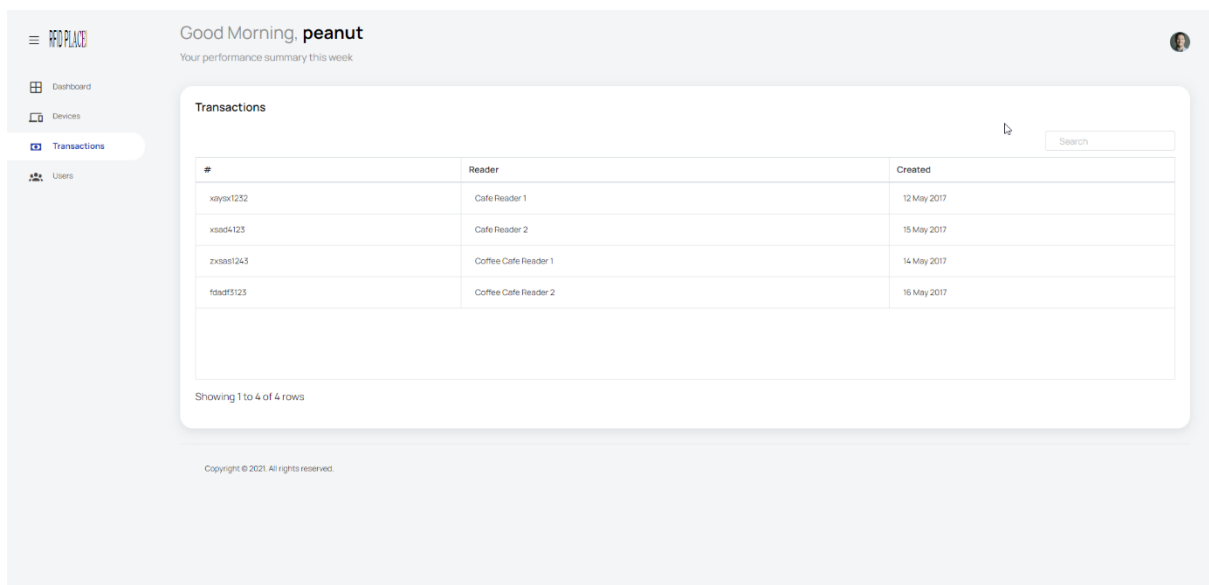


Fig 4, Transactions Module

In Fig 3, it shows all the transactions that has happened and the associated card reader, this will be extended.

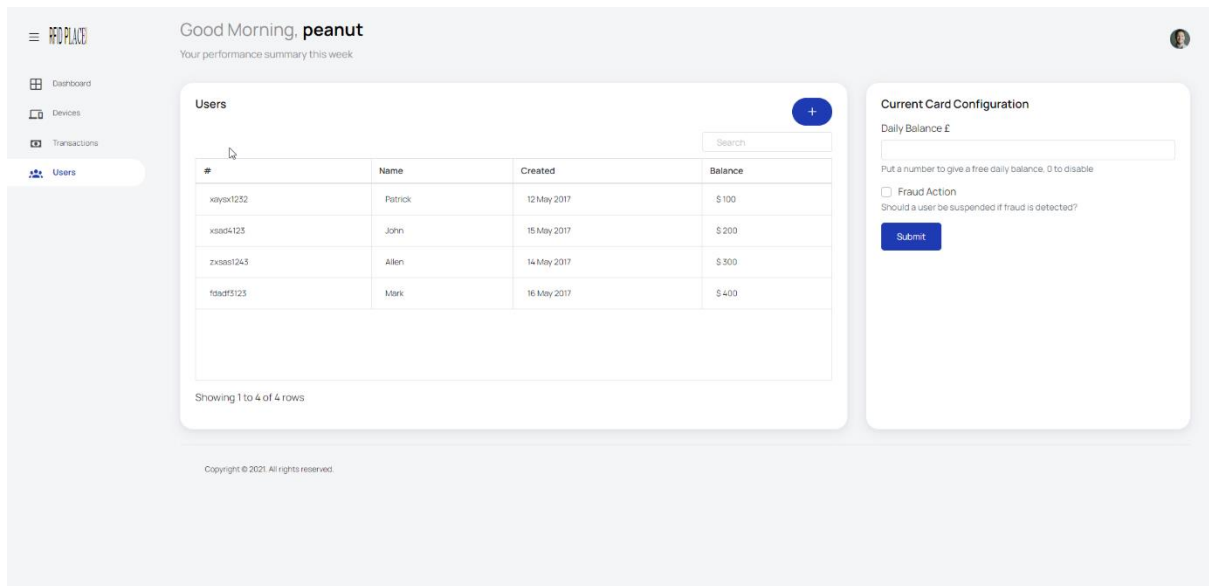


Fig 5, Users module

In Fig 4, an administrator can create users, set their balances, and create a free daily allowance.

2.1.2. Device Interface

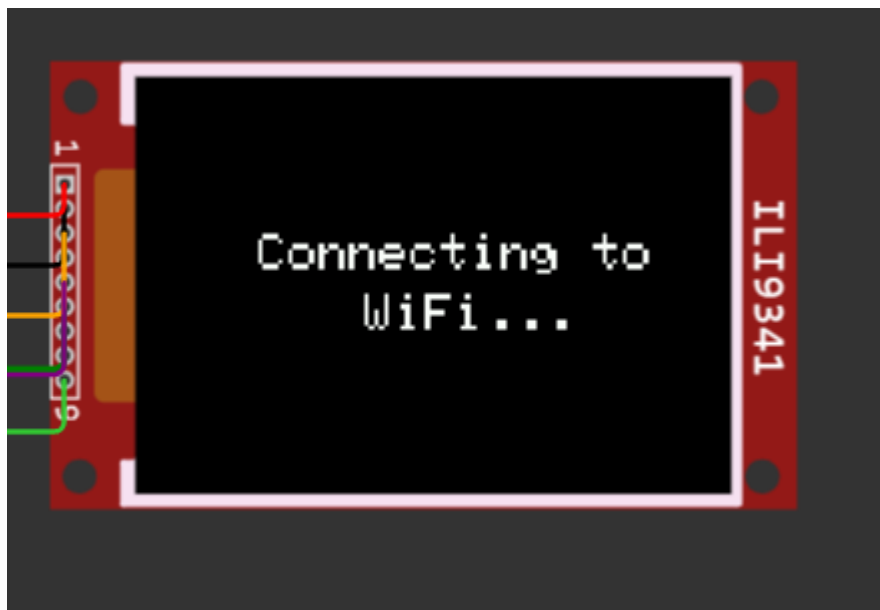


Fig 6, Connecting to Wi-Fi Interface



Fig 7, Prompting the user to connect to the device's Wi-Fi to continue setup



Fig 8, Prompting the user to visit the web dashboard to register the device to their account

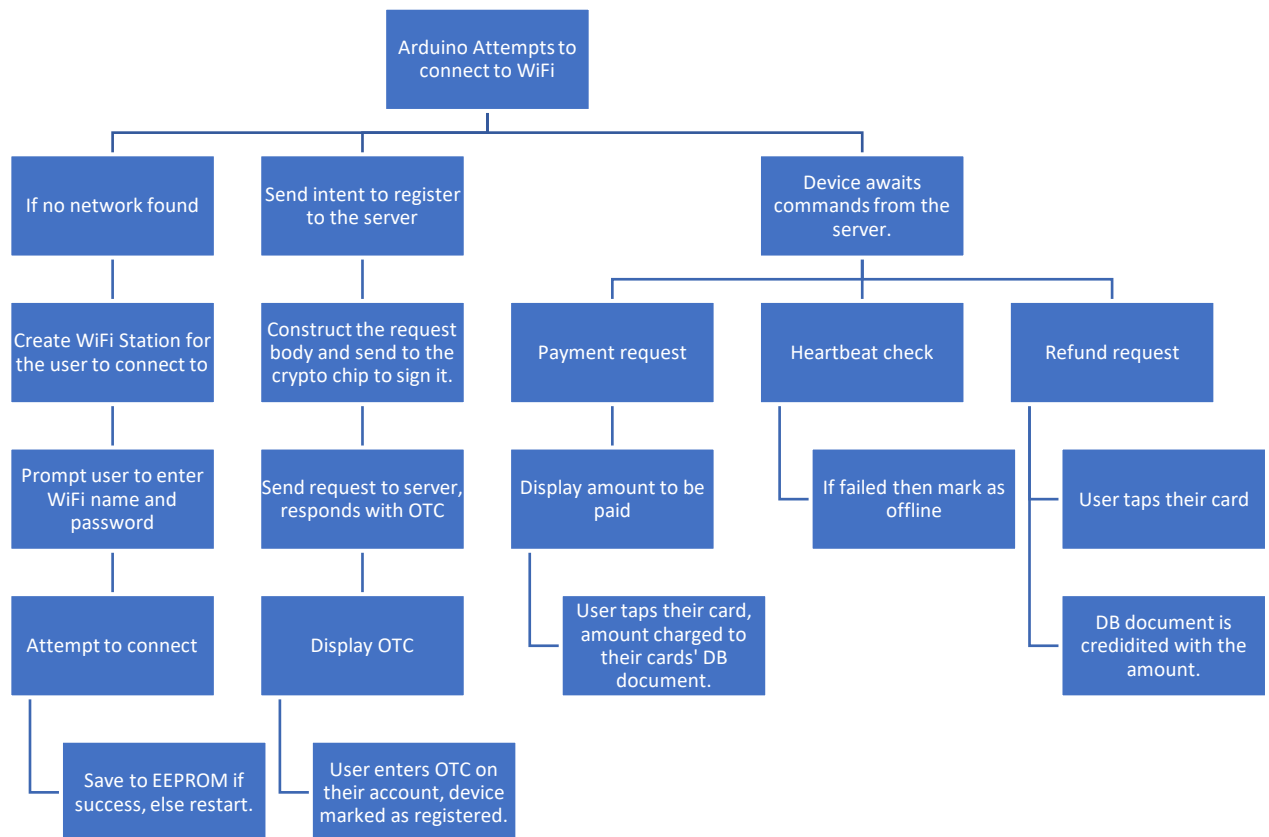


Fig 9, Example error message

2.2. Algorithms

The main challenge in implementing this solution is ensuring authenticity between the client and server.

Send intent to register to server → Show one time code → User connects the device to their account → Device awaits commands from the server.



The process is split into three main sections, connecting to the Wi-Fi, registering to the server and how it manages commands from the server.

Configuration overview

- Happens before being sent to the end-user e.g., in factory.
- Write configuration to the ATECC508A cryptography IC (integrated circuit).
- Write PSK (pre-shared key) to the ATECC508A EEPROM
- Lock the data zone and configuration zone of the EEPROM.
- Create a document for the device in the database with a serial number and the pre-shared key.

Registration overview

- Connect device to Wi-Fi.
- Construct body for the intent to register to be sent to the server.
- The request body is sent off to the crypto chip to be signed.
- Send request to the server, receive OTC.
- Display OTC, user enters the code into the dashboard.

- Device is registered, awaits further commands.

Device handling server commands overview

- Connect to the server via WebSocket.
- Send device UID signed with timestamp + nonce (or just get a token when the first registration, use TLS if this.)
- Monitoring for server commands.

2.2.1. Hardware Setup

Initialize core components

Firstly, the two most important components are initialized: the crypto IC and TFT LCD screen. The former allows secure communication while the latter outputs essential information to the user during the setup process.

Connecting the device to the Wi-Fi

Firstly, the device will attempt to connect using any credentials saved in the EEPROM, if none are found then it will create an access point that the user can connect to via any Wi-Fi enabled device. Upon connecting, a captive portal will pop up; this is simply a webpage that the user is forced to stay on otherwise they are automatically disconnected from the network. The captive portal asks the user to select a Wi-Fi network to connect to from a list and the associated password. The device then attempts to connect and if successful will store the access point ID and password.

Registering the device to a user

After Wi-Fi connectivity is verified, the device makes a request to the backend API and first requests its registration status. If it is not registered, then it will make a second request for a one-time code which is then displayed to the user. This request is also signed using a PSK (pre-shared key) which ensures authenticity of the device. The user then visits the dashboard where it can add a device using the code as shown in Fig 3, Website Interface. **This topic is covered in depth at [section 2.2.3](#).**

Final Setup

In this stage, some hardware components that only require minimal setup are initialized such as the RFID module and buzzer.

2.2.2. Configuration of the crypto IC

Purpose of the crypto IC

As mentioned in the first concept, security in this project is of paramount importance as such the project makes use of a chip specially made for cryptographic operations. It includes secure storage, hardware acceleration and anti-tampering mechanisms. The security issue I faced was that when the firmware was uploaded to the Arduino the code on it can still be decompiled (converted from compiled code back into source code) and a private key stored in the code could easily be compromised by a malicious actor. To solve this problem the project makes use of the secure storage element on the IC, which is never revealed to Arduino, mitigating any risk of key exposure. Additionally, speed is important as users can get easily frustrated with slowdowns so making use of a chip that accelerates cryptographic operations helps the project.

Common attack types

This solution attempts to repel a few different common attacks as best as it can. I will provide a definition of each attack.

- **Replay attacks**
These attacks attempt to replay the state of the device in the past. For example, if some data was used to get a successful authentication, a hacker would replay the data to get that correct authentication.
- **Man-in-the-middle attack**
This attack would alter information being sent to the server, we fixed this by adding the hashing signature system with the ATECC508A to ensure authenticity.
- **Rainbow tables**
A table of precomputed hashes relating to a hacked password. A attacker that gains access to the database would try to see if any hashes in the database matches his rainbow table. If there are any matches he can check what the actual value of the hash is in his table and get the unhashed value. We repel this by using dynamic information e.g. time and get the ATECC508A itself to generate some random bytes which is added onto the hash.

Structure of the crypto IC

The datasheet describes in detail how to configure the IC for your usage. For this project, the IC will be used to store private keys and signing of data. In the IC's data zone, it is split up into sixteen slots and each slot being thirty-six bytes. The slots are further divided into blocks, for example, one slot would take up 2 blocks with the first block having 32 bytes then the remaining 4 bytes being put in the second block. This structure is also similarly reflected in the configuration zone however it differs in that it is made up of one slot and 4 blocks with each block still holding thirty-two byte each.

Configuration of the crypto IC

In the configuration zone, we describe how each slot in the data zone should be treated by the device. As said above, the project uses the IC for storage of secret keys and for signing data, therefore the first slot in the data zone is set as holding a private key and is allowed to be used for cryptographic operations. We also set the read/write permissions, this slot is secret so it should never be able to be read or overwritten, so we disable both read/write.

Locking the IC and general usage

Once the configuration is complete, we lock the configuration zone so no further changes can be made. We then write the PSK to the data zone in the slot we configured, slot one. After this the data zone is locked as well. The IC is now ready to be used. The authentication algorithm we will be using is HMAC as it is lightweight and as such is fast, this is perfect for embedded devices. To sign some data, we will send an initial command to the IC with the following parameters (the 0x represents it is in hexadecimal e.g., 0x0F = 15 = F): opcode for hashing (0x47), mode HMAC (0x04) and the slot for which secret key to use (slot 1). Once the IC responds that it has acknowledged the first command, we then send the data we would like to sign. Once finished we send another command this time with the mode HMAC end (0x05), this makes the IC respond with the hash of our data.

Creating database entry

Before being sent to the end user, we will create a database document entry of the serial number taken from the crypto IC which is guaranteed to be unique as well as the pre-shared key saved in the crypto IC. This is used for the server identifying the device for when the end-user is registering it.

2.2.3. Registration

Connecting the device to Wi-Fi

Arduino will attempt to connect to the last known Wi-Fi credentials stored in the EEPROM. If it is unable to connect, it will create a Wi-Fi network for the user to connect to where a popup will appear and prompt the user to select a Wi-Fi network and enter its password. Once successful, the credentials are then stored in the EEPROM for future use.

Constructing intent to be registered

The device then checks its registration status with the server, if not registered it requests an OTC (one-time code) for the user to enter on their dashboard and connect it to their account. The way it does this is as follows: construct the body of the request, send it off to the ATECC508A crypto chip to be signed using a PSK (pre-shared key) that only the device and server know and then send it off to the server. This ensures that the request is coming from an authentic device and not a bad actor. The same process is used for any request that is privileged (e.g., requesting a payment).

Authenticating the user to the dashboard

When a user signs up, the web server captures their username, email, and password. The password is hashed using bcrypt which automatically appends a salt to the end of the hash. This protects it against attacks like those using [rainbow tables](#) as the salt adds randomness to the hash. When a user then tries to login in, we take their password and retrieve the hash from the database. We can then recalculate the hash and if it matches then we know they have entered the correct password. The user's cookie session is then authenticated.

2.2.4. Charging the users RFID Cards

Handling commands from the server

Using Websockets we connect to the server, we are using Websockets as the connection as it will be an extended one. We authenticate ourselves using the signing algorithm. There are three commands that the device expects: PAY_REQ and REFUND_REQ. The names are self-descriptive, and both just involve deducting or incrementing the amount of the respective RFID card in the database.

Charging the user's card

Using the method described above to communicate with the server, when a PAY_REQ command is received we will debit the user's entry in the database. Each card has a unique identifier that allows us to create an entry in the database and access it when a user taps their card. Their balance in the database would be deducted by the amount of the purchase.

Mitigating the insecurities of the Mifare Classic cards

Due to the lack of security in the Mifare classic cards, the solution attempts to mitigate the effect of card cloning. The way the algorithm works is as such: each card holds its UID and a hash of the last transaction in the card's EEPROM. The idea is that there should only ever be a single card so if a card attempts to pay for a transaction and the last hash doesn't match, we know that at some point there must have been two separate cards. The backend would then suspend that card UID, blocking it from making transactions.

2.2.5. Usage of libraries

On device usage

In the device, the solution requires the use of libraries to simplify communication between components. Each component has its own library which is created by the Arduino community and is used in this project. It has occurred however where a certain library did not fulfil all the needs of the solution, so we expanded the library to fill the use case.

On backend server usage

The solution uses a few libraries only where absolutely needed. For example, express.js is used as the web app framework while Mongo DB and their official library is used for the database.

2.3. Data Structures

Structure	Explanation
Byte array	For communication with the peripheral devices e.g., RFID reader. They communicate in binary
Char array	Holding text data
JSON Objects	Format stored in database as well as structure used for communication with the API

2.4. Classes

Class Name	Explanation
Wi-Fi	Has all the methods and properties associated with Wi-Fi
Adafruit_ILI9341	Has all the methods and properties associated the TFT LCD Display
MFRC522	Has all the methods and properties associated the RFID Reader
Device	Has all the properties for defining a single device on the backend server
Payment	Has all the properties for defining a single payment on the backend server
User	Has all the properties for defining a single user on the backend server

2.5. Testing Method

In development, each module created should be independently assessed to ensure that the functions of it output their expected values. Due to the nature of the application, there is no 'out of range' data simply correct or incorrect. The structure of the incorrect data is important, we will evaluate malformed data and data which has the correct form but the data itself is not valid. Using the IDE will allow us to check for syntactical errors and allow us to leverage debugging tools to fix logical errors.

2.6. Iterative Development

The developmental cycle used for the project will be iterative development. Each problem will be broken down into smaller sub problems and will make up a module of the entire project. These can then be combined to form a complete solution. The algorithms section already lays out the general flow of the program and the separate modules.

3. Development and testing

The development of the solution is made up of two separate applications: the Arduino firmware and the backend webserver. The former manages capturing RFID data and communicating it to the server. While the latter manages storing that data in the database and the web dashboard where users configure their device remotely.

Through this iterative development process, the first iteration focuses on the Arduino firmware, each section being broke into modules and iterated through. The modules are broken into initializing the device, connecting it to the server and authentication and displaying information to the user.

The second iteration is about the backend and web dashboard and focuses on providing the logic to the Arduino firmware modules as well as logic for making the web dashboard operate.

3.1. Arduino Firmware

3.1.1. Configuring the device into a default state.

We first need to initialize certain components for the first time as they require pre-configuring before being deployed to the consumer.

Configuring ATECC508A

The ATECC508A is configured as defined in the [configuration of the crypto IC \(2.2.2\)](#).

```
bool configDevice() {
    // Arrays manually constructed using the datasheet pg 13
    uint8_t slotConfig[] = { 0x83, 0x20, 0x83, 0x20 };
    uint8_t keyConfig[] = { 0x1C, 0x00, 0x1C, 0x00 };

    // Writing slotConfig to the config zone at address 0x0005
    if (!atecc.write(0x00, 0x0005, slotConfig, 4)) {
        printOutputBuffer();
        return false;
    }

    delay(10);
    Serial.println("Set SlotConfig Successfully.");

    // Writing keyConfig to the config zone at address 0x0018
    if (!atecc.write(0x00, 0x0018, keyConfig, 4)) {
        printOutputBuffer();
        return false;
    }

    delay(10);
    Serial.println("Set KeyConfig Successfully.");
    return true
}
}
```

Fig 1: configDevice() function, configATECC508A.ino

In Fig 1, this function writes the two-byte arrays that configure the crypto IC. The config and keyConfig uint8_t byte arrays both hold four bytes. They have been constructed in accordance with the ATECC508A's datasheet (Microchip, 2017).

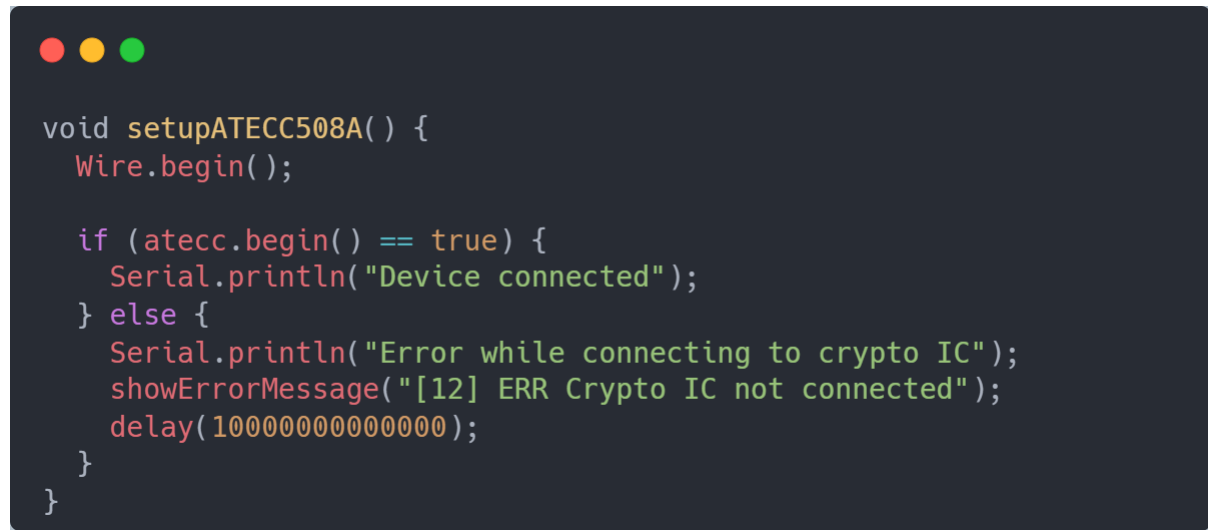
3.1.2. Initializing the device for the first time

The user now has the device and turns it on for the first time. Some initial steps need to take place as defined in the [hardware setup section \(2.2.1.\)](#).

Initialize core components

The ATECC508A crypto IC as well as the TFT LCD screen are initialized.

ATECC508A Initialization


A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains the following C++ code:

```
void setupATECC508A() {  
    Wire.begin();  
  
    if (atecc.begin() == true) {  
        Serial.println("Device connected");  
    } else {  
        Serial.println("Error while connecting to crypto IC");  
        showErrorMessage("[12] ERR Crypto IC not connected");  
        delay(1000000000000000);  
    }  
}
```

Fig 1: setupATECC508A() procedure, configDevice.cpp

The procedure simply starts transmission between the Arduino and ATECC508A. The atecc token is a class for the crypto IC's library that provides methods for communication.

TFT LCD Screen Initialization

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains the following C++ code:

```
void setupDisplay() {  
    tft.begin();  
    tft.setRotation(1);  
}
```


Fig 1.1: setupDisplay() procedure, DisplayFunctions.cpp

This procedure again starts transmission between the TFT LCD screen and the Arduino and the tft token is again a class for communication with the TFT LCD screen provided by a library.

Connecting the device to Wi-Fi

All the core components have now been initialized so Wi-Fi setup begins.

Wi-Fi Setup Entry procedure



```
void setupWifi() {
    loadFromEEPROM();
    displayStartupConnection();

    bool connected = false;

    // loop tries 3 times to connect with stored name and pass, if
    // fails then falls through and makes user connect again
    for (int tries = 0; tries < 3; tries++) {
        WiFi.begin(ssid, password);
        int result = WiFi.waitForConnectResult();
        if (result == WL_CONNECTED) {
            connected = true;
            break;
        }
    }

    if (!connected) {
        configWifi();
    }

    Serial.print("Connected, IP address: ");
    Serial.println(WiFi.localIP());
}
```

Fig 1.3: setupWifi() procedure, WIFIFunctions.cpp

This procedure is the entry into the Wi-Fi setup algorithm. It tries to connect using a stored Wi-Fi name and password from a previous setup and if not present, makes the user connect to one via the configWifi() procedure.



```
void configWifi() {
    WiFi.mode(WIFI_AP_STA);
    WiFi.softAPConfig(apIP, apIP, IPAddress(255, 255, 255, 0));
    if (WiFi.softAP("ESP8226")) {
        Serial.println("Ready");
        Serial.println(WiFi.softAPIP());
    }

    dnsServer.start(DNS_PORT, "*", apIP);

    // serves the html page that forces the user to enter Wi-Fi
credentials
    serveCaptivePortal();

    // displays a message on the tft lcd screen prompting the user
to connect to the device's wifi on their phone
    setupWifiMessage();
    while (serverOn) {
        // handle next request
        dnsServer.processNextRequest();
        webServer.handleClient();
    }

    // disable AP here and turn off webServer.
    WiFi.softAPdisconnect(true);
    webServer.close();
    dnsServer.stop();
}
```

Fig 1.4 configWifi() procedure, WIFIFunctions.cpp

This procedure creates a Wi-Fi access point, waits for the user to connect then serves a captive portal which forces the user to enter Wi-Fi credentials.

```

String wifiConnect() {
    Serial.println(ssid);
    Serial.println(password);
    WiFi.disconnect();
    WiFi.begin(ssid, password);
    int connRes = WiFi.waitForConnectResult();
    String responseResultHTML = ""
                                "<!DOCTYPE html><html lang='en'><head>"
                                "<meta name='viewport' content='width=device-
width'>"
                                "<title>CaptivePortal</title></head><body>";

    switch (connRes) {
        case (WL_CONNECTED):
            // save to eeprom logic
            saveToEEPROM();
            serverOn = false;
            return responseResultHTML += "<h1>Connected to " + String(ssid) + "!</h1>
</br><h3>You can close this now</h3></body></html>";
            break;
        case (WL_CONNECT_FAILED):
        case (WL_WRONG_PASSWORD):
            return responseResultHTML += "<h1>Password Inncorrect</h1></br><a
href=\"/\>Click here to return</a></body></html>";
            break;
        default:
            return responseResultHTML += "<h1>Error " + String(connRes) + "</h1><a
href=\"/\>Click here to return</a></body></html></body></html>";
    }
}

```

Fig 1.4, wifiConnect() function, returns success message to user on the captive portal

```

void saveToEEPROM() {
    EEPROM.begin(512);
    EEPROM.put(1, ssid);
    EEPROM.put(1 + sizeof(ssid), password);
    char ok[2 + 1] = "OK";
    EEPROM.put(1 + sizeof(ssid) + sizeof(password), ok);
    EEPROM.commit();
    EEPROM.end();
}

void loadFromEEPROM() {
    EEPROM.begin(512);
    EEPROM.get(1, ssid);
    EEPROM.get(1 + sizeof(ssid), password);
    char ok[2 + 1];
    EEPROM.get(1 + sizeof(ssid) + sizeof(password), ok);
    EEPROM.end();
    if (String(ok) != String("OK")) {
        ssid[0] = 0;
        password[0] = 0;
    }
}

```

Fig 1.5, saveToEEPROM() & loadFromEEPROM() procedure, used for writing the Wi-Fi credentials to memory to use in future startups

3.1.3. Connecting to the server and authenticating the device

Once a Wi-Fi connection is established, we can move onto connecting to the backend server and verifying the devices' authenticity. These steps are described in section [2.22](#) and [2.23](#)

Entry function for authentication

```
void setupDevice() {  
  
    WiFiClient client;  
    ESP8266WiFiSTAClass WiFi;  
    HTTPClient http;  
  
    // state variables  
    int status = 0;  
    int tries = 0;  
    bool failed = false;  
    bool registered = false;  
    int otc = 0;  
  
    // loop that tries 5 times to connect to the server, otherwise shows error  
    while (status != 200) {  
        if (tries == 5) {  
            failed = true;  
            break;  
        }  
        displayRegisterServer(tries);  
        delay(1000);  
  
        http.begin(client, BASE_URL + "v1/devices/requestotc");  
  
        http.addHeader("Content-Type", "application/json");  
  
        int nonce = atecc.getRandomInt();  
        String reqBody = buildBody(WiFi, getTime(), nonce);  
  
        status = http.POST(reqBody);  
  
        if (checkRegStatus(1000)) {  
            break;  
            registered = true;  
        }  
        if (status != 200) {  
            Serial.printf("[HTTP] POST... failed, error: %s\n", http.errorToString(status).c_str());  
            http.end();  
            tries = tries + 1;  
            continue;  
        }  
  
        DynamicJsonDocument payload(1024);  
  
        deserializeJson(payload, http.getString());  
  
        otc = payload["otc"];  
  
        http.end();  
    }  
  
    if (registered) { return; }  
  
    if (!failed) {  
        registerMessage(otc);  
        if (!checkRegStatus(600000)) {  
            setupDevice();  
        }  
        return;  
    }  
  
    // HTTP_REG_FAIL is a constant defined in DisplayFunctions.h, its value is 1 and is for internal use to identify the type  
    // of error.  
    showErrorMessage(HTTP_REG_FAIL);  
}
```

Fig 1.6, setupDevice() procedure, entry function for starting the authentication process.

The main purpose of the code in fig 1.6 is to establish a connection with the server and authenticate itself. The solution achieves this by looping over a request that it sent to the server five times, if it

fails then we show an error message. Once a connection is established the client (Arduino) will send a request as formatted in section [2.23: Constructing intent to be registered](#).

Checking registration status

```
bool checkRegStatus(int timeout) {
    WiFiClient client;
    ESP8266WiFiSTAClass WiFi;
    HTTPClient http;

    bool registered = false;
    unsigned long intialMillis = millis();
    int tries = 0;

    while (!registered && (millis() - intialMillis < timeout)) {
        http.begin(client, BASE_URL + "v1/devices/regStatus?uid=" +
        WiFi.macAddress());
        http.GET();

        if (tries > 5) { break; }
        if (status != 200) {
            tries++;
            continue;
        }

        DynamicJsonDocument regStatBody(1024);

        deserializeJson(regStatBody, http.getString());

        if (regStatBody["registered"]) {
            registered = true;
            break;
        }

        http.end();
        delay(1000);
    }

    if (!registered) {
        return false;
    }

    return true;
}
```

Fig 1.7, checkRegStatus() function, utility function that checks the current registration status of the device on startup, if true we can skip the remaining setup in Fig 1.6.

```

String buildBody(ESP8266WiFiSTAClass WiFi, unsigned long timestamp, int nonce)
{ DynamicJsonDocument doc(1024);

  unsigned long epochTime = timestamp; // change

  JsonObject dataObj = doc.createNestedObject("data");
  dataObj["uid"] = WiFi.macAddress();
  dataObj["ssid"] = WiFi.SSID();
  dataObj["timestamp"] = epochTime;
  dataObj["nonce"] = nonce;
  String jsonDoc;
  serializeJson(dataObj, jsonDoc);

  uint8_t jsonDocBytes[jsonDoc.length()];
  jsonDoc.getBytes(jsonDocBytes, jsonDoc.length() + 1);

  uint8_t jsonInternalHash[32];
  atecc.hmac(jsonDocBytes, sizeof(jsonDocBytes), 0x0000, jsonInternalHash);

  String signature = "";
  for (int i = 0; i < sizeof(jsonInternalHash); i++) {
    if ((jsonInternalHash[i] >> 4) == 0) {
      signature += "0"; // append preceeding high nibble if it's zero
    }
    signature += String(jsonInternalHash[i], HEX);
  }

  doc["signature"] = signature;

  String finalDoc;
  serializeJson(doc, finalDoc);

  return finalDoc;
}

```


Fig 1.8, buildBody() function, returns a string of the formatted data in JSON formatting to be sent to the server.

Most of the code in fig 1.8 is repeating as we add multiple properties to a single object. The important part is when the jsonDocBytes is initialised. Here, we create an ASCII hexadecimal byte array of the data in the JSON object (unique id, Wi-Fi ssid, timestamp and nonce) then we pass it into the crypto IC's hmac() function which writes the output to jsonInternalHash. This is also given as an ASCII hexadecimal byte array so we iterate over each byte and append it to a string. We can then attach the result and the full JSON object is made.

3.1.4. Displaying information to the user

Displaying custom messages


These functions are used to display information on the TFT LCD display.



```
void showDataOnDisplay(String uid, uint8_t size)
{
    tft.fillScreen(ILI9341_BLACK);
    tft.setCursor(0, 0);
    tft.setTextColor(ILI9341_WHITE);
    tft.setTextSize(size);
    tft.println(uid);
}
```

Fig 1.9, showDataOnDisplay() procedure, this procedure displays a custom message on the TFT LCD screen.

Displaying error messages



```
void showErrorMessage(int err) {
    tft.fillScreen(ILI9341_RED);
    tft.setTextColor(ILI9341_WHITE);

    tft.setCursor(0, (240-(8))/2);
    tft.setTextSize(2);
    switch(err) {
        case CRYPTO_IC_DC:
            tft.print("Error [");
            tft.print(CRYPTO_IC_DC);
            tft.print("]");
            tft.println(" Failed to connect to crypto IC. Restart.");
            break;
        case HTTP_REG_FAIL:
            tft.print("Error [");
            tft.print(HTTP_REG_FAIL);
            tft.print("]");
            tft.println(" Failed to connect to web server. Check Wi-Fi.");
            break;
    }
    while(true); // infinite loop
}
```

Fig 2.0, showErrorMessage() procedure, this procedure uses the constants defined in DisplayFunctions.h to write an error message to the display.

Displaying register status



```
void displayRegisterServer(int attempt)
{ tft.fillScreen(ILI9341_BLACK);
  tft.setTextColor(ILI9341_WHITE);

  tft.setCursor(16,90);
  tft.setTextSize(3);
  tft.println("Registering with");
  tft.setCursor(79, 126);
  tft.println("server...");
  tft.setCursor(80, 170);
  tft.print("Attempt: ");
  tft.print(attempt);
  delay(1000);
}
```

Fig 2.1, diplayRegisterServer() procedure, this procedure displays a specific message telling the user the device is registering with the server and the current attempt.

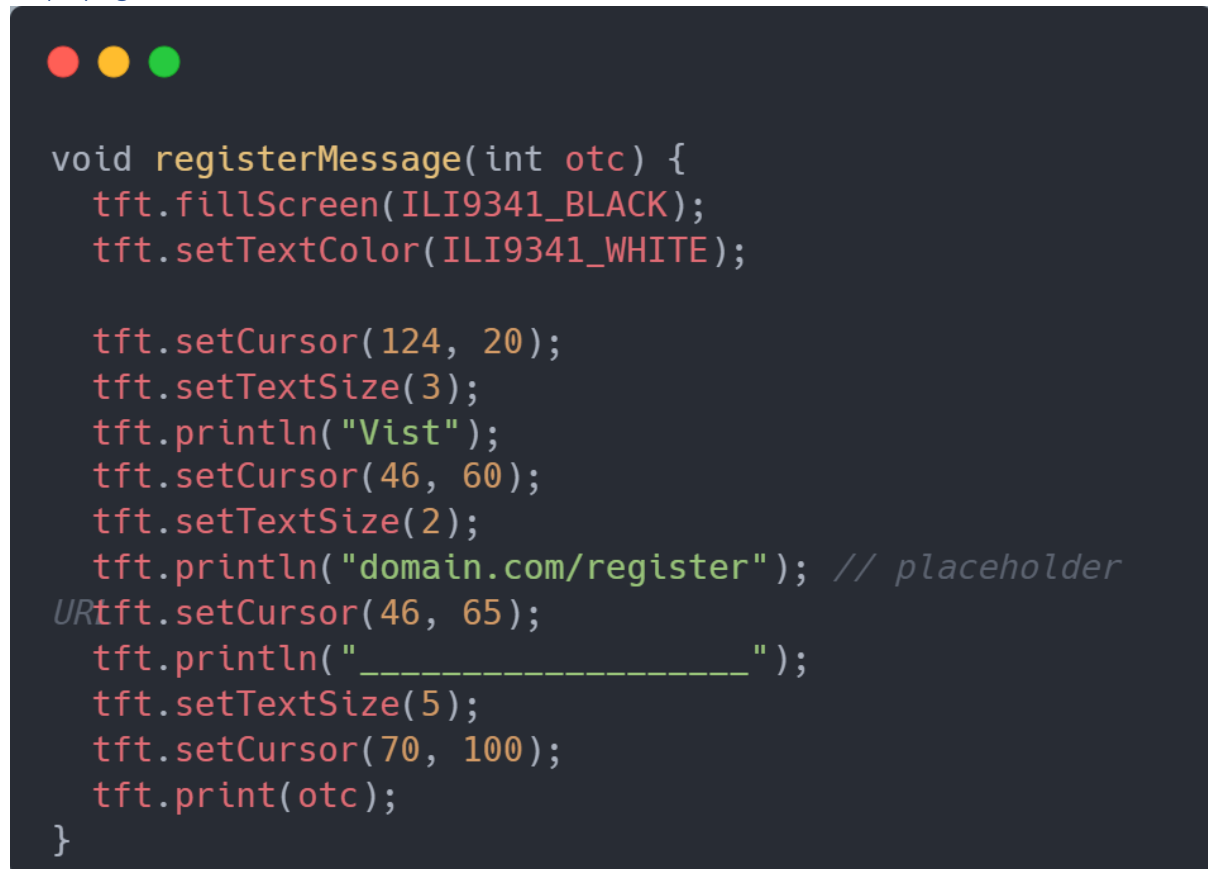


Fig 2.2, registerMessage() procedure, this procedure displays the OTC (one-time code) and the URL to register the device at.

3.1.5. Testing at each stage

Connecting to Wi-Fi

In section 3.1.2 during connecting the device to the Wi-Fi a bug was found where if the Wi-Fi name had spaces in it the device wouldn't be able to connect to it.

Wi-Fi Name	Connection successful?
Testwifi	True
Test_wifi	True
Test wifi	false

After investigating, I found that the algorithm mistakenly thought a space was the end of the string and the bug was rectified by changing the logic behind where the end of a string actually is.

Displaying information to the user

This bug came in two parts as because of the display failing it effected the crypto IC as well which caused unexpected behaviour and hard to pin down the problem. The display and crypto IC wouldn't respond to certain commands or output garbage responses. The problem was that on the breadboard the wiring was misconfigured and some pins from the LCD and crypto IC weren't going into their proper slots. After rewiring, both components now worked flawlessly.

3.1.6. Arduino Firmware Summary

In this section, we have described the method of communication between the Arduino and the backend server. We have securely authenticated the device so that the server recognises it as genuine and can interact with it in payment transactions.

In initial prototypes, the crypto IC was absent however after research and seeing how easily a device could be compromised without it, I decided to add it to the solution.

3.2. Web server

3.2.1. Device registration workflow backend

In this section we describe the steps that happen on the backend when the workflow in [section 3.1.3](#) is executed. This section follows the algorithms defined in [section 2.2](#).

Checking registration status

```
async function getRegStatus(req: Request, res: Response) {
  const deviceId = req.query.uid;
  try {
    req.session.destroy(() => {})
    const db = await getDB();
    const devices = db.collection<deviceSchema>("devices");

    // fetch the device in db for some basic checks
    let regCheckDevice = await devices.findOne<deviceSchema>({
      deviceId: deviceId
    });

    if(!regCheckDevice){
      throw new ApiError(HttpStatus.NOT_FOUND, "NOT_FOUND")
    }

    return res.json({
      success: true,
      registered: regCheckDevice.registered
    })

  } catch (e: any) {
    if(e instanceof ApiError) {
      return res.status(e.statusCode).json({
        success: false,
        error: e.message
      })
    }
    return res.status(HttpStatus.INTERNAL_SERVER_ERROR).json(
      {
        success: false,
        error: HttpStatus[500]
      }
    );
  }
}
```

Fig 3.1, getRegStatus() function, checks the given device UID's registration status which allows the device to skip it if it is already registered.

Requesting one-time code

```
async function requestOTC(req: Request, res: Response) {
  const signature: string = req.body.signature;
  const nonce: string = req.body.data.nonce;
  const clientTimestamp: number = req.body.data.timestamp;
  const ssid: string = req.body.data.ssid;
  // data has to be submitted in the same order everytime btw
  try {
    if (!req.body.data.uid) { throw new ApiError(httpStatus.BAD_REQUEST, "NULL_PARAM"); };
    // expand this to make it clear which error

    const db = await getDB();
    const devices = db.collection<deviceSchema>("devices");

    // fetch the device in db for some basic checks
    let deviceForConfirmation = await devices.findOne<deviceSchema>({
      deviceUID: req.body.data.uid
    });

    const serverTimestamp = Math.round(Date.now() / 1000);
    if (!deviceForConfirmation) { throw new ApiError(httpStatus.NOT_FOUND, "NO_RECORD"); };
    // check it actually exists
    if (!isValidNonce(nonce)) { throw new ApiError(httpStatus.BAD_REQUEST, "INVLD_NONCE") };
    // check if valid nonce

    console.log("recieved body:\n"+JSON.stringify(req.body))
    console.log("recieved signature:\n"+req.body.signature)
    console.log("expeceted signature\n"+hmac(deviceForConfirmation.psk,
    JSON.stringify(req.body.data)))
    if ((serverTimestamp - clientTimestamp) > 10) { throw new
    ApiError(httpStatus.REQUEST_TIMEOUT, "TIMEOUT"); }; // only accept 5s delay
    if (hmac(deviceForConfirmation.psk, JSON.stringify(req.body.data)) !== signature) {
    throw new ApiError(httpStatus.BAD_REQUEST, "BAD_SIG"); }; // verify signature

    const otc = await generateOTC(devices);

    await devices.updateOne({ deviceUID: req.body.data.uid }, {
      $set: {
        ssid: ssid,
        publicIP: req.ip,
        otc: otc
      }
    });

    // why i do this is to let the ssid & ip be updated IF they have switched wifi's
    if (deviceForConfirmation.registered) { throw new
    ApiError(httpStatus.CONFLICT, 'ALR_REG'); }; // checks if registerd

    return res.json({
      success: true,
      error: null,
      otc: otc
    });
  } catch (e: any) {
    if (e instanceof ApiError) {
      console.log(`[!] Device ${req.body.data.uid || "NO_UID"} failed request for OTC
      because ${e.message}`);
      return res.status(e.statusCode).json({
        success: false,
        error: e.message
      });
    }
    console.log(`[!] Device ${req.body.data.uid || "NO_UID"} failed request for OTC because
    ${e.message}`);
    return res.status(httpStatus.INTERNAL_SERVER_ERROR).json({
      success: false,
      error: httpStatus[500]
    });
  }
}
```

Fig 3.2, requestOTC() function, returns the one-time code for device registration.

Registering device to server

```
async function registerDevice(req: Request, res: Response) {
  const registerBody: registerBody = req.body

  try {
    if(registerBody.authKey) {
      // do api key stuff
    } else if (!req.session.auth){
      throw new ApiError(httpStatus.UNAUTHORIZED,"NO_AUTH")
    } else if (!req.session.data){
      throw new ApiError(httpStatus.BAD_REQUEST,"MAL_SESS")
    }

    // now auth
    const db = getDB()
    const users = db.collection<userSchema>('users');
    const devices = db.collection<deviceSchema>('devices');

    const device = await devices.findOne({
      otc: registerBody.data.otc
    })

    // check if data.location and data.name are present, if not throw ApiError
    if(!registerBody.data.name || !registerBody.data.location) {
      throw new ApiError(httpStatus.BAD_REQUEST,"NULL_PARAMS");
    }

    if(!device) {
      throw new ApiError(httpStatus.BAD_REQUEST,"BAD_CODE");
    } else if (device.registered){
      throw new ApiError(httpStatus.CONFLICT,"ALR_REG")
    }

    await devices.updateOne({otc: registerBody.data.otc},
      {$set: {registered: true,registeredUserID: req.session.data?.uid, location:
registerBody.data.location, name: registerBody.data.name}
    })

    users.updateOne(
      {uid: req.session.data?.uid},
      {$push: {deviceUIDs: device.deviceUID}
    })

    return res.json({
      success: true,
      error: null
    })
  } catch (e: any) {
    if(e instanceof ApiError) {
      return res.status(e.statusCode).json({
        success: false,
        error: e.message
      })
    }
    return res.status(httpStatus.INTERNAL_SERVER_ERROR).json({
      success: false,
      error: httpStatus[500]
    });
  }
}
```

Fig 3.3, registerDevice() function, checks the challenge created by the device then sets the device as registered in the database as well as attaching additional useful information.

3.2.2. Web Dashboard

This section builds the website dashboard to allow users to manage their devices, control certain parameters and users on their system.

Creating users

```
async function createUser(req: Request, res: Response) {
  const body: authUserBody = req.body;
  console.log(body)
  try {
    if (req.session.auth){
      throw new ApiError(HttpStatus.FORBIDDEN,"LOGGED_IN")
    }

    if(!body.data?.email) {
      throw new ApiError(HttpStatus.BAD_REQUEST,"NULL_PARAMS")
    }

    const db = await getDB()
    const users = db.collection<userSchema>('users')

    const user = await users.findOne({
      email: body.data.email
    })

    if(user) {
      throw new ApiError(HttpStatus.CONFLICT,"ALREADY_EXISTS")
    }

    const uid = uuidv4();
    const regDate = new Date();
    const newUser = await users.insertOne({
      email: body.data.email,
      uid: uid,
      deviceUIDs: [],
      password: await bcrypt.hash(body.data.password,config.app.saltRounds),
      registered: regDate,
      username: body.data.username
    })

    req.session.auth = true
    req.session.data = {
      uid: uid,
      email: body.data.email,
      deviceUIDs: [],
      registered: regDate,
      username: body.data.username
    }

    return res.json({
      success: true,
      error: null,
      data: {
        newUser
      }
    })
  } catch (e: any) {
    if(e instanceof ApiError) {
      return res.status(e.statusCode).json({
        success: false,
        error: e.message
      })
    }
    return res.status(HttpStatus.INTERNAL_SERVER_ERROR).json({
      success: false,
      error: HttpStatus[500]
    });
  }
}
```

Logging in user

```
async function loginUser(req: Request, res: Response) {
  const loginUserBody: authUserBody = req.body

  try {
    if (req.session.auth){
      throw new ApiError(HttpStatus.FORBIDDEN,"LOGGED_IN")
    }

    const db = getDB();
    const users = db.collection<userSchema>('users');
    const user = await users.findOne({email: loginUserBody.data.email})

    if(!user) { throw new ApiError(HttpStatus.NOT_FOUND,"USER_NOT_FOUND"); }

    const compareResult = await bcrypt.compare(loginUserBody.data.password,user.password)

    if(!compareResult) { throw new ApiError(HttpStatus.UNAUTHORIZED,"BAD_PASS"); }

    req.session.auth = true
    req.session.data = _.pick(user,['uid','deviceUIDs','email','registered','username'])
    return res.json({
      success: true,
      error: null
    })
  } catch (e: any) {
    if(e instanceof ApiError) {
      return res.status(e.statusCode).json({
        success: false,
        error: e.message
      })
    }
    return res.status(HttpStatus.INTERNAL_SERVER_ERROR).json({
      success: false,
      error: HttpStatus[500]
    });
  }
}
```

Fig 3.2, loginUser() function, authenticates a given user with their username and password.

Logging out users

```
function logoutUser(req: Request, res: Response) {
  req.session.destroy(() => {
    res.redirect('/login')
  })
}
```

Fig 3.3, logoutUser() function, logs out a user.

Rendering dashboard pages

For the rendering technique I decided to use SSR (server sided rendering). We populate all the data on the server then send the HTML page to the client.

```
async function renderDashboard(req: Request, res: Response) {
  if(!req.session.auth) return res.redirect('/login')

  let userDevices: object[] = []
  try{
    const db = getDB();
    const devices = db.collection<deviceSchema>('devices');

    const userDevicesCursor = devices.find<deviceSchema>({
      registeredUserID: req.session.data?.uid
    })

    for await (const doc of userDevicesCursor) {
      userDevices.push(_.omit(doc,['_id']));
    }
  } catch(e) {
    userDevices.push({err:true})
  }

  res.render('index',{session:req.session, devices: userDevices})
}
```

Fig 3.4, renderDashboard() function, renders the HTML by fetching DB data then substituting it into variables in fig 3.5.

SSR Rendering example

```
<div class="list-wrapper">
  <ul id="devicesList">
    <%-
      devices.map(function(device) {
        return `
          <li class="d-block">
            <span class="badge badge-success">Online</span>
            <div class="p-2">
              ${device.name}
            </div>
          </li>
        `
      }).join('')
    <%>
  </ul>
</div>
```

Fig 3.5, snippet of html code describing how data gets rendered. By using EJS, we can declare variables in our HTML code which when rendered like in Fig 3.5 on the last line, lets us substitute the data in.

3.2.3. Web server summary

In this section, we discussed how the backend web server handles the data sent to it by the Arduino as well as covering the front-end dashboard. For the dashboard the main testing focus was ensuring that it looked consistent across different devices. I made use of chrome's developer tools which allows you to adjust screen size of your browser to emulate devices environments (e.g. phone, tablets).

3.2.4. Testing at each stage

For the backend testing, correct data would be fed into the signature checking algorithm and if the expected response wasn't received, we would know the test failed.

```
{
  "data":{
    "nonce":"123413",
    "timestamp":"231233131",
    "ssid":"wifi_name",
    "uid":"device_uid"
  },
  "signature":"sha256_sig"
}
```

Example input data format; constant data would be input to the data object. We then test that the same signature is generated on the client and the server.

There were many iterations of this algorithm due to a multitude of problems. Here is a table summarising some of the errors found and how I fixed them:

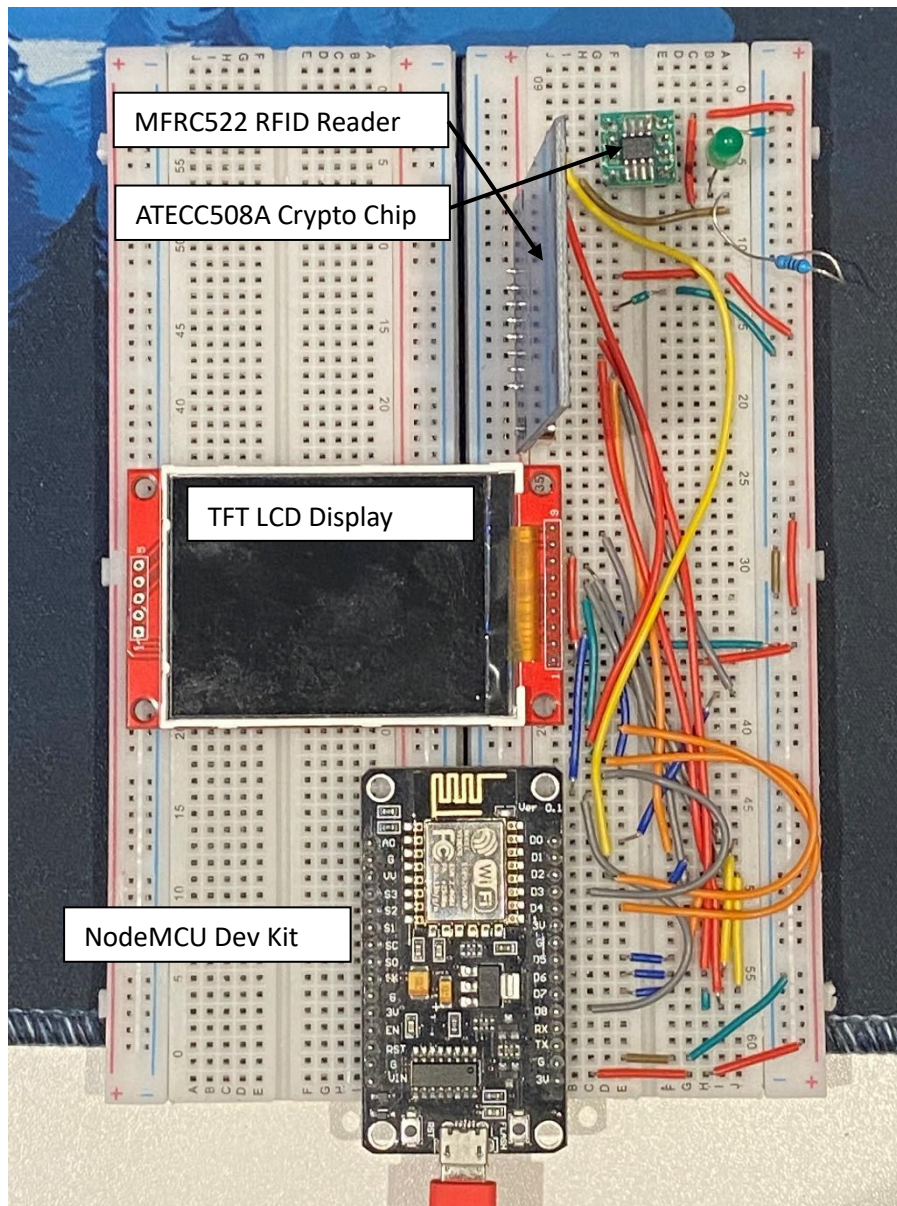
Problem found	Solution
Signature generated not matching expected signature	This was both a hardware and software problem. Firstly, some of the pins were plugged into the wrong I/O slots on the Arduino and the data wasn't being encoded into bytes correctly
JSON malformed when sending client data to server	This was simply a case of the JSON not being in the correct format.
Replay attack was possible if you resent the data within the 5 second timeframe	The "nonce" property in the data object was added. Before this a hacker could resend the data as many times in that 5 second window before the request became too old.

Usage of validation in the backend

Much of the validation is internally done in the libraries (e.g. validating the JSON to ensure it is the correct format). There is also validation to when we recalculate the signature on the server and ensure it matches the one the device sent.

4. Evaluation

4.1. Final physical device iteration



4.2. Black box testing

For this type of testing, we have some data to input and have prepared an expected output, if the expected output does not match the function output, then we know there is an issue with that function.

Testing was carried out at each iteration during the development and testing chapter, as documented in sections 3.1.5 and 3.2.4. In the first instance, a Wi-Fi issue was identified, which would have led to a failure of success criteria one. Subsequently, there was a problem with the display, causing it to inaccurately present information and thereby violating success criteria four and five.

4.3. Usability testing

In this testing, I have ran through the entire device setup flow which brings together the dashboard, website and the device itself.

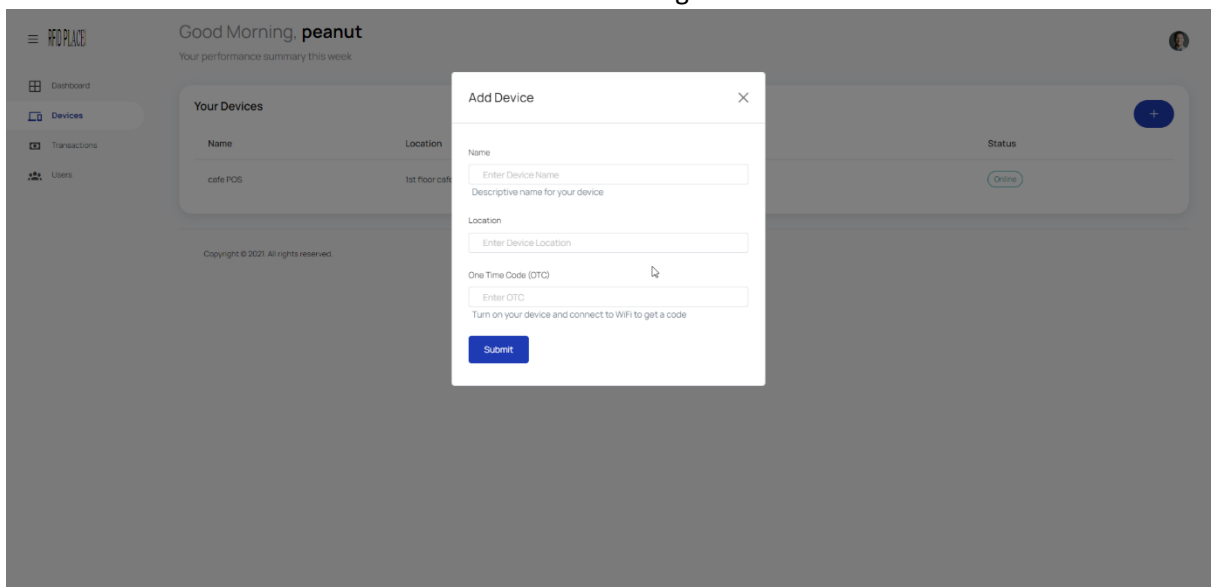
1. Turned on device, it starts searching for Wi-Fi networks then prompts the user to connect to its own Wi-Fi network so the user can exchange which Wi-Fi it wants the device to connect to.



2. In the Wi-Fi captive portal, the user can choose a Wi-Fi to connect to and enter the password if required.
3. Once connected the device will then display a onetime code to be used on the website for registration. It shows a clear domain and code which is very understandable for the user on what to do next.



4. Now on the dashboard the user can enter the code to register the device to their account.



5. Once the device is added it allows the user to manage them, see their status and send payment requests to it.
6. Now on the device, it is in a registered state and is ready to receive instructions.

4.4. Criterion Fulfilment

Criteria	Was it met
Device connects to any Wi-Fi network in good conditions without error and can maintain the connection	Fully met
Device can capture RFID card UID and data in a short time (10 ms)	Fully met
Device communicates to server in a secure manner	Fully met
Device RFID cards cannot be cloned	Not met
Device has understandable UI interface	Fully met
Device has an easy initial setup flow	Fully met
Device LCD shows each stage of the setup process	Fully met
Device has clear error messages that are easy to diagnose	Fully met
Low memory and power usage for the device	Fully met
Web backend charges the RFID card in the database	Partially met
Web dashboard allows users to manage the device remotely	Fully met
Web dashboard is easy to navigate	Fully met
Web dashboard allows users to set daily balance	Partially met
Web dashboard has an authentication system	Fully met

Table 1, original from 1.7.3 Success Criteria

4.4.1. Evidence for fully met criteria

1. Evidence for device Wi-Fi connectivity.

```
_id: ObjectId('64f20a995ced4c678b459738')
deviceUID: "BC:FF:4D:5F:EB:75"
lastHash: ""
otc: "203101"
psk: "2555885553752669D66245EBE549BCDE"
publicIP: "::ffff:192.168.178.96"
registered: true
registeredUserUID: "9f478602-7bf8-4985-b4d0-ac7c1087ab8a"
ssid: "35frontbedroom_2.4"
location: "Home"
name: "test device"
```

Wi-Fi name captured by the backend.

Fig 1, Database entry for a device

In figure 1, we see that the program has automatically captured the SSID (Wi-Fi name) of the device.

2. Device communicates to server in a secure manner

On the breadboard there is an ATECC508A which facilitates all cryptographic operations. This also holds the PSK (pre-shared key) in its EEPROM which protects it from attackers attempting to access it. All of this together makes it extremely difficult for an attacker to send duplicate information or fraudulent information.

3. Device has understandable UI interface & Device has an easy initial setup flow

The evidence for the UI interface is on the previous page. I believe that it is clear in its message and only provides the user with important information in the moment. The result is a minimal user interface that is both clean and efficient.

4. Device LCD shows each stage of the setup process

This has been achieved, evidence on previous page.

5. Device has clear error messages that are easy to diagnose

I decided to give a unique number to each known error. The error display system is found in section 3.1.4: Displaying error messages. The manual or website would then provide a table for information on these errors' codes with troubleshooting information for easy resolutions.

6. Web dashboard allows users to manage the device remotely

In figure 1, two important properties allow us to facilitate this. Firstly, the registeredUserUID property links a specific device to a user, allowing us to display it in their dashboard for further configuration and management. Configurable properties include location which can allow a user to pinpoint a device for easy physical access as well as a name property which can help in making it even more recognisable.

7. Web dashboard has an authentication system

```
_id: ObjectId('64f1f9fb652619ba73ff5c15')
email: 
uid: "9f478602-7bf8-4985-b4d0-ac7c1087ab8a"
deviceUIDs: Array (1)
  0: "BC:FF:4D:5F:EB:75"
password: "$2b$10$60TNwIfd3ISYTDJxT9cki.nUiuXspcEbj5XQwT4UvvhVi/Oy.4fXa"
registered: 2023-09-01T14:49:31.045+00:00
username: "peanut"
```

Fig 1.1, database entry for a user

Here we see a specific user's entry in the database. For the authentication we use an username and password. We can then use the password hash to recalculate the inputted password.

4.4.2. Justification for not met & partially met criteria

1. **Device RFID cards cannot be cloned**

As discussed in the analysis section and after deeper investigation during development it was found that the Mifare classic cards have a broken encryption implementation. This allows them to be easily cloned and allows replay attacks from that angle. The solution is clear: use the updated Mifare DESFire cards that include AES encryption, the problem however is that the documentation for these cards is under NDA, so I am unable to understand them and program them into the solution.

In a commercial solution we would have a genuine need for access to company proprietary information so that could easily be fixed.

2. **Web backend charges the RFID card in the database**

The logic on the backend for this has been implemented by the device implementation wasn't able to be fully completed to a fully functional standard. This could be achieved with more development time.

3. **Web dashboard allows users to set daily balance**

Again, this has been implemented in the dashboard as seen in Fig 5 in section 2.1.2 but the device implementation has not been completed. This could also be achieved with more development time.

4.5. Limitations in the final solution

As described above are the most critical limitations and reasons for not meeting particular criteria. Here I will go more in depth as to why certain important criteria weren't met. For the first one, it is self-explanatory. I was unable to access certain documentation and the cards themselves weren't readily available to individuals to buy in singles. As for the second and third, they both link to the actual payment part of the solution. These criteria were partially completed on the backend as mentioned, however as they were placed at the last criteria to be completed in the implementation, the timeline didn't work out how I expected. I preferred to focus on securing the device and try to minimize the effect of a card or device being compromised. It did add significant development time to the solution due to amount of research done and wanting to take full advantage of the security features it enabled. This has been achieved, the solution attempts to mitigate the security issues of the Mifare classic cards as mentioned in [section 2.2.4](#). In addition to this, on the device the usage of the ATECC508A and a unique PSK for each device means that even if a single device gets compromised it would not affect the security of other devices on the system.

4.6. Overcoming limitations

The first limitation could only be fixed by becoming a genuine company with a real product. Only then could I gain access to the documentation which is under NDA. The second would be easily completed with more development time. The implementation for this part of the solution is detailed in section 2.2.4.

4.7. Maintenance and improvements

The main concern of the solution is about the security of it. In any area where payments are involved security should always be a top priority. I would do continuous penetration testing to identify and fix any security issues before deployment to production as well as during it. Another improvement would be designing a custom PCB for all the components to sit in a compact package, ready for

commercial usage. Additionally, certain routes on the backend API could be exposed for public use to allow them to implement their own features and improve their workflows in using the device.

Features being added would be based on user feedback as the people using the solution for an extended time can provide great insight into improvements.

References

Hawas Design, n.d. [Online]

Available at: <https://hawasdesign.com/cashless-canteen-system/>

Microchip, 2017. *Microchip ATECC508A Datasheet*. [Online]

Available at: <https://docs.rs-online.com/eba3/0900766b8166f69b.pdf>

Nohl, K., Evans, D., S. & Plotz, H., 2008. *Reverse Engineering Mifare Classic cards*. [Online]

Available at: https://www.usenix.org/legacy/event/sec08/tech/full_papers/nohl/nohl.pdf

Tucasi, n.d. [Online]

Available at: <https://www.tucasi.com/dinner-money/>