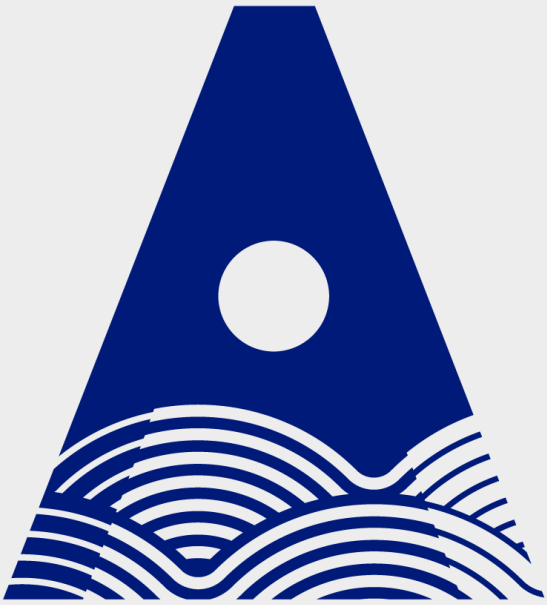


Ollscoil Teicneolaíochta an Atlantaigh

Atlantic Technological University

SCAN & GO

Project Engineering

BEng Software & Electronic Engineering

Year 4

CIAN O DONNELL – G00358872

Declaration

This project is in partial fulfilment of the requirements for the degree of the BEng Software and Electronic Engineering course at the Atlantic Technological University.

This project is my own, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.

_____

Acknowledgements

I would like to Acknowledge my supervisor for this project, Niall O'Keefe, for his guidance and reliability throughout the course of this project. His help was greatly appreciated.

I would also like to thank the remaining project engineering lecturers, Paul Lennon, Michelle Lynch and Brian O'Shea for their help and input throughout the course of this project.

Table of Contents

1 Summary 4
2 Poster 5
3 Introduction 6
4 Technologies 7
4.1 Arduino MKR WiFi 1010 7
4.2 RFID MFRC/RC-522 module 7
4.3 Amazon Web Services (AWS) 7
4.4 MySQL 7
4.5 Node.js 7
4.6 Next.js 8
5 Project Architecture 9
6 Project Plan 10
7 Project and Code Functionality 11
7.1 C++ Code 11
7.2 AWS Services ..... **Error! Bookmark not defined.**
7.3 Next.js Code ..... **Error! Bookmark not defined.**
8 Ethics 26
9 Conclusion 27
10 References 28

1 Summary

Scan & Go is a radio-frequency identification (RFID) door access project that can be implemented in manufacturing, healthcare, and other sectors of the economy. The goal for this project was to show how and where employee data is stored after an employee's RFID badge is scanned. The logged data will include all relative information for an employee as well as their permission status for the door being accessed.

Door access will be granted or denied based on an employee's job role and the job requirements of the room they are accessing. All employee scans will be logged to a database and can be viewed on a web application. New employees can be added and assigned personal RFID cards through the web application. Employee information can also be added, edited or deleted through the web application. The web application is login protected to secure employee information.

A Wi-Fi board with an RFID module connected to it will be the system to read RFID card scans. The card IDs generated from these scans will be sent to a cloud hosted MySQL relational database (RDS) table after the Wi-Fi board publishes the card ID numbers to the cloud. A separate MySQL RDS table will hold all employee information. A card ID can then be assigned to an employee, creating their own personal employee card. The logs for these employee card scans can be securely accessed on the web application.

An Arduino MKR-1010 Wi-Fi board and MRC-522 RFID module are used to write the programme for the card scanning system. Cards at 13.56 MHz frequency are compatible with this module. AWS cloud services are used to retrieve card IDs from the Wi-Fi board and into the cloud hosted database. Next.js is the React framework used to create the web application with the aid of Nodejs for backend development. NodeJS is also used with AWS Lambda function and AWS RDS services.

The project delivered on having successful card readings and having them card readings sent to the required database. Create, Read, Update and Delete (CRUD) functions can all be completed through the web application. Security measures are taken with the web applications login and protecting page routes with the use of authentication. This prevents any employee sensitive data being access or share.

2 Poster

Ollscoil Teicneolaíochta an Atlantaigh

Atlantic Technological University

SCAN & GO

Cian O'Donnell
G00358872
BEng Software & Electronic Engineering

Description

Scan & Go is a radio-frequency identification (RFID) door access project that can be implemented in manufacturing, healthcare, and other sectors of the economy.

Door access is granted or denied based on an employee's job role and the job requirements of the room being accessed.

All employee scans are logged to a database, these logs can be viewed on a web application.

New employees are added to the system through the web app. Employee information is also edited or deleted through the web application.

Technologies

The **MKR1010 Wi-Fi board** handles Wi-Fi connections with the aid of its built-in ESP32 chip. The on-board LED signals access granted (green) or access denied (red) to the user.

The **MFRC-522** RFID card reader is connected to the Wi-Fi board, which will receive card IDs when a card is scanned.

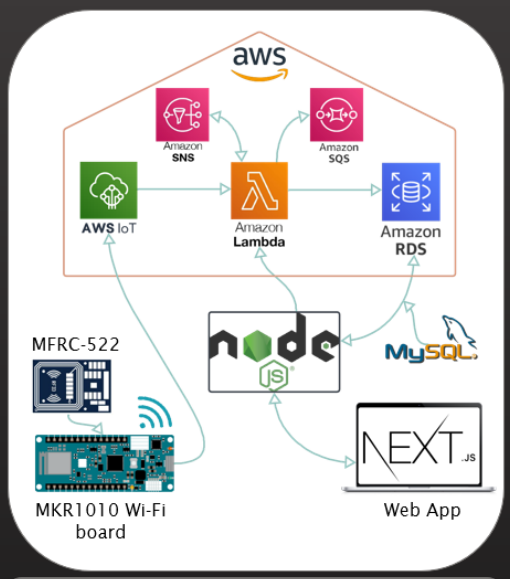
Amazon Web Services (AWS):

- **IoT Core** is used to receive published card data from the Wi-Fi board.
- **Lambda** functions are used to receive card data from IoT Core and send the RFID card data to a relational database (**RDS**).
- **Simple Notification Service** (**SNS**) and **Simple Queue Service** (**SQS**) are used as a way of queuing and pushing all data received in the lambda function, ensuring all data gets sent to the database.
- **RDS** is used as the cloud hosted server for the database.

MySQL is the relational database that holds all employee information and door access tables.

NodeJS is used for the Lambda function, connecting to the RDS database, and writing the backend code for the web app.

Next.js is a React framework that's used to create a full stack web application.



Approach

The approach was to have RFID card IDs read from the card reader and have these card IDs added to a database. The card IDs will then be assigned to employees so they have their own employee badge.

Assigning cards and adding, editing or deleting employee information will be done through the web application.

Employees can then scan their badge and they will be granted or denied access to a room depending on their job role. All card scans will be logged to a database and these logs can be accessed on the web app.

Results

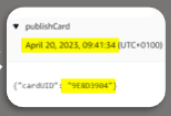
Finding a means to publish data from the Wi-Fi board into cloud services proved difficult throughout this project. Google Cloud was not compatible with the Wi-Fi board. For this reason, AWS was the cloud service chosen.

Their SNS and SQS services were also essential in ensuring that all data received from the IoT core was seamlessly delivered into the database.

This procedure is displayed below:

An RFID card is scanned. This employee will be denied access.

Card ID is published to the IoT core with a timestamp.



Scan has been logged in the database with employee information and door permission for their employee badge.

Conclusion

Scan & Go has delivered on its requirements. A system has been created to read employee card scans when their RFID cards are scanned. Logs of these card scans are saved to a database and can be viewed on the web application.

Employee information can be successfully added, edited or deleted through the Scan & Go web application.

Employee information is able to be edited or deleted through the web app.

3 Introduction

This project considers how records are kept in manufacturing industries. I have spent many years working in various manufacturing companies, I found that the companies that use RFID/Smart cards and had computer software to keep track of all employee's data and task submission were the companies that had the most productivity and success. I got the inspiration for this project from this experience and wanted to see how I could put my own spin on a manufacturing logging system and understand all that goes on behind the scenes.

The approach that I took when creating this project was, I would have RFID cards to control access to certain rooms in a manufacturing building setting. The RFID card will hold all necessary information of the employee, and based on their job title and the room they are attempting to access, they will either be granted or denied access. All card scans will then be stored in a database to log the attempt and time of access. These logs will be available to view by logging in to the designed company web application. The web app would also have a section to perform CRUD operations to add/edit employee information.

RFID card scans will be read by scanning a 13.56 MHz card against the RFID card reader (MFRC-522 module) which will be connected to an Arduino MKR1010 Wi-Fi board. The Card number read from the board will be sent up to Amazon Web Services (AWS) through its IOT core service. The card data published will go through an AWS lambda function to then be sent to a MySQL relational database (RDS) table in AWS RDS. The card number is sent a database table specifically created to hold card IDs. Once a card is read in, that card ID will be joined with the Employee table and assigned to a member of that list. This operation is performed seamlessly through the web app as opposed to doing it through SQL. Once this is completed and an employee scans their RFID card against the card reader for a specific door, they will be either denied or granted access. All scans will be logged and be available on the web app.

JavaScript and CSS were the primary languages used for creating the full stack web application. C++/C was used for writing the programme for the MKR1010 Wi-Fi board for the RFID card readings/publish data, Wi-Fi connections and LED control on the board.

4 Technologies

The technologies section will cover all tools and technologies used for this project. This section will also cover the research that went into these technologies and why they were chosen.

4.1 Arduino MKR WiFi 1010

The Arduino MKR WiFi 1010 board was chosen for the internet of things (IoT) side of the project. The board enables WiFi allowing to connect wirelessly to peripherals, it also features the ECC508 crypto chip for security [1]. The board has a built in LED which is used in the project for displaying access granted (green LED) and access denied (red LED) signals. With these features it was a great choice for the project and is why it was chosen over the ESP32 Board. The board also has ESP32 firmware onboard.

4.2 RFID MFRC/RC-522 module

The RFID MFRC/RC-522 module is a module which is compatible with the MKR WiFi 1010 board and was also cost effective in this project. The MFRC/RC-522 is a highly integrated reader/writer IC for contactless communication at 13.56 MHz [2]. With its connection to the WiFi board, RFID card IDs can be read to the board and deployed to a cloud IoT service of choice.

4.3 Amazon Web Services (AWS)

When researching a cloud platform for this project, the two platforms considered to trial was Google Cloud and AWS. With clearer documentation and better knowledge of google services, I thought Google Cloud would be a much better choice to use with this project. This was not the case and it proved quite difficult with connecting the mkr1010 WiFi board with Google Clouds IoT Core. AWS IoT Core was then used and was successful in receiving publish messages from the WiFi board.

AWS proved more convenient than Google Cloud to set up a relational database on. Its other services such as Lambda, Simple Queue Services (SQS), Simple Notification Services (SNS) and Amplify also played a big part in this project's functionality.

4.4 MySQL

After researching SQL databases, they seemed a much better fit for the style in which this project manages and stores its data. MySQL stores data in tables (rows and columns) and database tables can be joined seamlessly, something NoSQL databases such as MongoDB are not setup to do [3]. This felt like a solid structure for storing employee data and assigning employee RFID cards to these employees through joining tables.

4.5 Node.js

Node.js allows developers to create both frontend and backend applications using JavaScript [4]. Node.js is used throughout this entire project, it is primarily used for the development of the web application. Node.js is also used to write the code for the AWS Lambda function which collects data from IoT Core and delivers it to the RDS database.

4.6 Next.js

Next.js is a React framework which is used to create the full stack web application. Next.js is used in this project to create the web application that allows a user to add, edit or delete employee information with use of its CRUD features in the backend. The web application will display employee tables with all relevant information of the employee. The final feature of the web app is to display the door logs table which the WiFi board which is mimicking a type of room which could be found in a manufacturing building. All these features in the web application will only be accessed after securely logging in.

5 Project Architecture

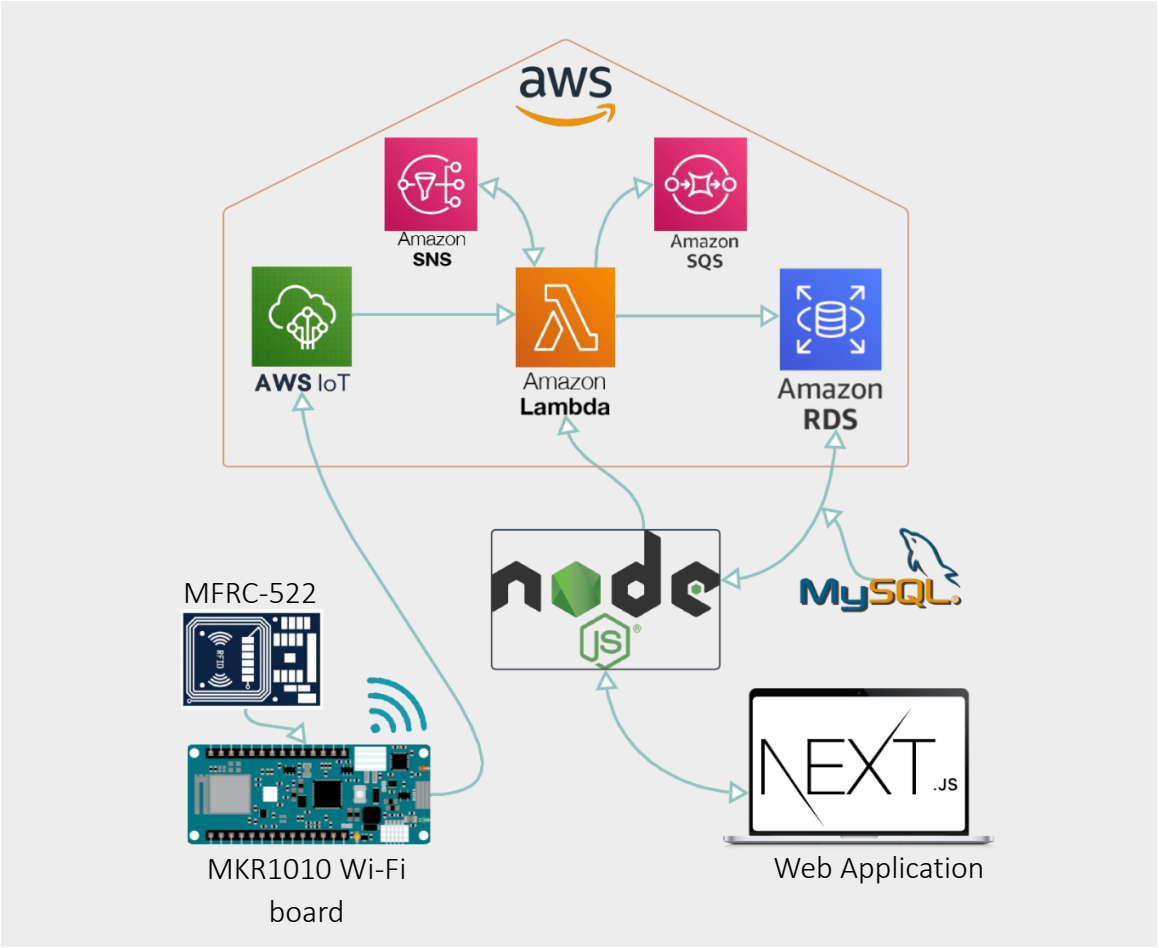


Figure 5-1 Architecture Diagram

6 Project Plan

Jira was the main project management tool that was used for this project. I was able to create an interchangeable timeline with tasks to complete from the start and end date of my project. These tasks were able to have child components added to them to break a big task into smaller pieces for better progress. Sprints were a means of putting child tasks together and setting a start and end date for when these tasks where to be completed. Jira sprints were a big help in project organisation and fulfilling deadlines.

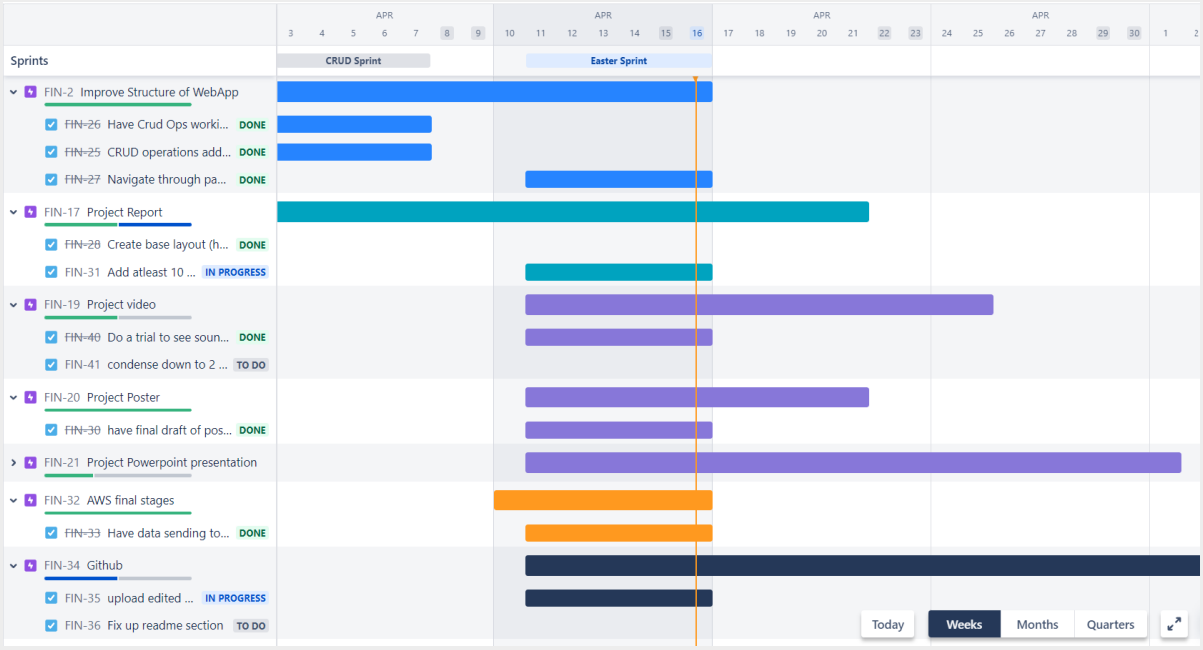


Figure 6-1 Project Plan

Figure 6-2 below will show a well-planned sprint with multiple tasks and were completed on time versus a poorer sprint in figure 6-3, which conveys a sprint that went past its planned end date.

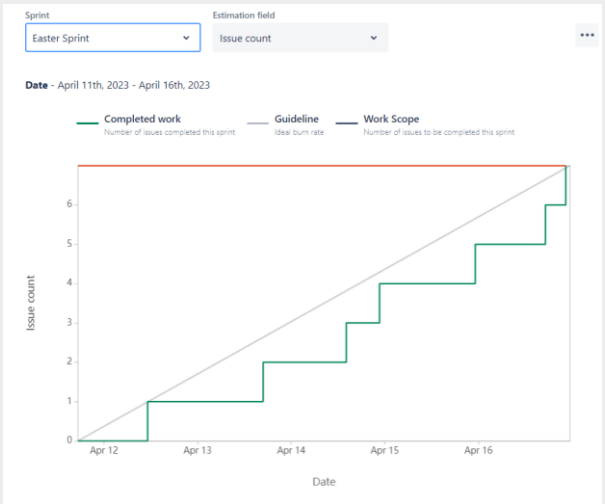


Figure 6-2 Good Sprint

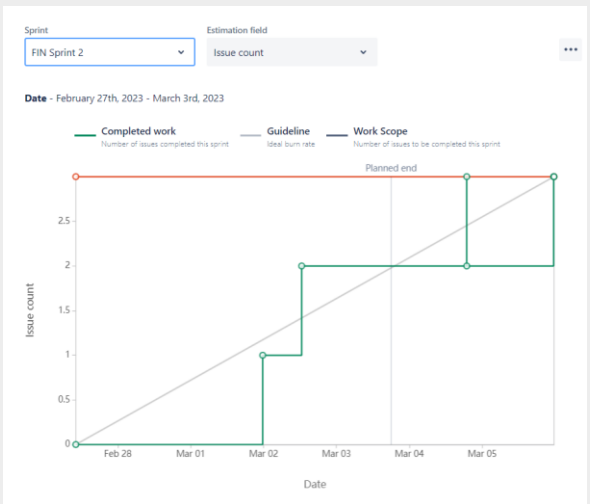


Figure 6-3 Poor sprint

7 Project and Code Functionality

The following section will cover the functionality of the project and the code that went along with it. All code covered in this section can be found on this GitHub repository:

https://github.com/CianODonnellGIT/FYP

7.1 C/C++ Code

The C/C++ code in this project is used for writing the functionality of the MKR WiFi 1010 board and the MFRC/RC-522 RFID module connected to it. This code can be found in the AWS_IoT_WiFi_RFID folder in the GitHub repository. The code in this folder was influenced by Arduino sample libraries [5][6][7].

The WiFi board had to first be encrypted as a security measure for this project and was also necessary to be able to gain access to the AWS IoT Core service [8]. When the board was encrypted, a private Certificate Signing Request (CSR) key is generated. This was achieved using the Arduino 'ECCX08CSR' sample library.

With the board encrypted and the CSR key generated, the code for AWS_IoT_WiFi_RFID could now be written. In the figure below, the necessary included libraries are listed with their role commented by them.

```
#include <SPI.h>      //mfrc522 to mkr1010
#include <MFRC522.h>  //access RFID libraries
#include <WiFiNINA.h> //access wifi libraries
#include <utility/wifi_drv.h> //for colour change on mkr LED
#include <ArduinoBearSSL.h> // encryption
#include <ArduinoECCX08.h> //encryption
#include <ArduinoMqttClient.h> // coonection to mqtt broker
#include "arduino_secrets.h" //WiFi info, CSR key, IoT MQTT Broker
```

Figure 7-1 Included Libraries

The const variables from the arduino_secrets.h were then to be initialized along with the WiFi, SSL and MQTT clients which are used thorughout the code.

```
//Sensitive data in arduino_secrets.h
const char ssid[]        = SECRET_SSID;
const char pass[]        = SECRET_PASS;
const char broker[]      = SECRET_BROKER;
const char* certificate  = SECRET_CERTIFICATE;

WiFiClient    wifiClient;            //Used for the TCP socket connection
BearSSLClient sslClient(wifiClient); //Secure SSL/TLS connection, integrates with ECC508
MqttClient    mqttClient(sslClient); //Mqtt AWS connection
```

Figure 7-2 Initializing Variables

Further initialization and setup of the code will be done in void setup, where all configurations are setup before main execution of the code. The LED, RFID module and checking of board encryption and certificates will be configured here.

```
void setup() {
  Serial.begin(115200);
 // while (!Serial);
  SPI.begin();        // Init SPI bus
  mfrc522.PCD_Init(); // Init MFRC522 card reader

  // set all key bytes to 0xFF
  for (byte i = 0; i < 6; i++) {
    key.keyByte[i] = 0xFF;
  }

  WiFiDrv:: pinMode(25, OUTPUT); //GREEN LED
  WiFiDrv:: pinMode(26, OUTPUT); //RED LED
  WiFiDrv:: pinMode(27, OUTPUT); //BLUE LED

  if (!ECCX08.begin()) {
    Serial.println("No ECCX08 present!");
    while (1);
  }
  // Set a callback to get the current time
  // used to validate the servers certificate
  ArduinoBearSSL.onGetTime(getTime);

  // Set the ECCX08 slot to use for the private key
  // and the accompanying public certificate for it
  sslClient.setEccSlot(0, certificate);
}
```

Figure 7-3 Void Setup

After setup, the void loop function is then run. The void loop checks several functions, firstly for WiFi connection through the connectWiFi function. If there is no connection to the ssid and password provided, it won't continue onto the next step. Connection to the WiFi will be retried every 5 seconds until connected.

```
while (WiFi.begin(ssid, pass) != WL_CONNECTED) {
  Serial.print(".");
  delay(5000);
}
```

Figure 7-4 connectWIFI function Check

Once WiFi connection has been established, if there no MQTT connection, the connectMQTT function is then run. If the MQTT broker is not connected to 'broker' (AWS IoT Core endpoint) at port 8883, then won't continue to the next step of the void loop. Connection to the broker will be attempted every 5 seconds until connected.

```
//endpoint of aws iot core on port 8883
while (!mqttClient.connect(broker, 8883)) {
  Serial.print(".");
  delay(5000);
}
```

Figure 7-5 connectMQTT function Check

After WiFi and MQTT broker connections have been achieved, the next call in the void loop is to poll for MQTT messages. This is used to keep the MQTT broker alive and be available to receive message all the time.

The mfrc_rfid function is then called. This function holds the bulk of the operations that the WiFi board will execute. These operations are receiving RFID card data from the reader, publishing said data to AWS IoT Core and programming correct LED signals for each card.

Once a card is scanned, the first part of the mfrc522_rfid function will read the card ID, the card ID will then be concatenated together into String form which will be used to validate an access granted or access denied card.

```
void mfrc522_rfid(){
  String content = "";

  // Reset the loop If no new card is present.
  if ( ! mfrc522.PICC_IsNewCardPresent())
    return;

  // Select one of the cards
  if ( ! mfrc522.PICC_ReadCardSerial())
    return;

  Serial.print("UID Card:");
  for (byte i = 0; i < mfrc522.uid.size; i++) {
    Serial.print(mfrc522.uid.uidByte[i] < 0x10 ? "0" : "");
    Serial.print(mfrc522.uid.uidByte[i], HEX);
    content.concat(String(mfrc522.uid.uidByte[i] < 0x10 ? "0" : ""));
    content.concat(String(mfrc522.uid.uidByte[i], HEX));
  }
```

Figure 7-6 mfrc522_rfid function Card Check

A selection of the card IDs returned will then be assigned for access granted cards. These will be validated by checking if the selected concatenated ID are scanned, Access granted LED will be signalled, for the other cards access denied will be signalled. The card data will then be sent to IoT core after each signal through the publishMessage function.

```
content.toUpperCase();
if(content == "A15B8E09" || content == "C7823904" ||
   content == "B7C73904" || content == "1E523904" || content == "FEB63904" ){
  Serial.println("Access Granted");
  Serial.println();

  //signal access granted
  //green led displaying for 1 second
  WiFiDrv:: analogWrite(25, 0);
  WiFiDrv:: analogWrite(26, 255);
  WiFiDrv:: analogWrite(27, 0);

  delay(1000);

  WiFiDrv:: analogWrite(25, 0);
  WiFiDrv:: analogWrite(26, 0);
  WiFiDrv:: analogWrite(27, 0);

  publishMessage();
}
```

Figure 7-7 mfrc522_rfid Access Granted

```
else{
  Serial.println("Access Denied");
  Serial.println();

  //signal access denied
  //red led displaying for 1 second
  WiFiDrv:: analogWrite(25, 255);
  WiFiDrv:: analogWrite(26, 0);
  WiFiDrv:: analogWrite(27, 0);

  delay(1000);

  WiFiDrv:: analogWrite(25, 0);
  WiFiDrv:: analogWrite(26, 0);
  WiFiDrv:: analogWrite(27, 0);

  publishMessage();
}
```

Figure 7-8 Access Denied

The publishMessage function at the end of the if/else statements is what grabs the card ID scanned and publishes it up to the AWS IoT Core. The publish message name in mqtt.beginMessage will be the same as what is subscribed to in the IoT Core MQTT broker. The hexadecimal style message will be published in JSON format. Once publishMesssage function is run, this will complete the mfrc522_rfid function.

```
void publishMessage() {
  Serial.println("Publishing message");

  //Start message to be published to IoT
  //publishCard is message name to connect with AWS IoT
  mqttClient.beginMessage("publishCard");

  // The JSON String send up to AWS which will be sent to AWS Lambda
  mqttClient.print("{\"cardUID\": \"");
  for (byte i = 0; i < mfrc522.uid.size; i++) {  //
    mqttClient.print(mfrc522.uid.uidByte[i] < 0x10 ? "0" : "");
    mqttClient.print(mfrc522.uid.uidByte[i], HEX);
  }
  mqttClient.print("\"}\n");
  mqttClient.endMessage();
}
```

Figure 7-9 publishMessage Function

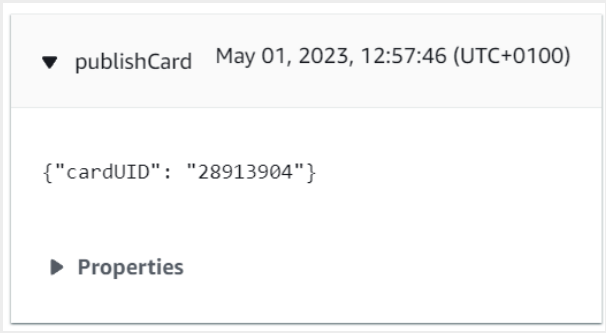


Figure 7-10 AWS IoT Core Publish Message Received

The final function called in the void loop is getTime, which is returning the current time from the WiFi Module. The void loop function will loop continuously in the format shown in the figure below.

```
void loop() {
  if (WiFi.status() != WL_CONNECTED) {
    connectWiFi();
  }

  if (!mqttClient.connected()) {
    // MQTT client is disconnected, connect
    connectMQTT();
  }

  // poll for new MQTT messages & send keep alives
  mqttClient.poll();
  mfrc522_rfid();
  getTime();
}
```

Figure 7-11 Void Loop

7.2 AWS Services

AWS Services had a big part to play in the functionality of this project. The main objective was to receive messages in the IoT Core service, pass the received data through a Lambda Function and then inserted into a MySQL relational database table hosted on RDS service.

IoT Core:

To achieve publishing message to the IoT core Service, a 'thing' had to first be created in the IoT Core console along with the private CSR certificate generated on the MKR WiFi 1010 board. The objective of creating a thing is to connect the WiFi board to the IoT console. My thing was name 'MKR1010RFID'. Once a thing is successfully setup in the console, a policy was then assigned to the thing. The policy for this thing was to allow everything (*). This policy would then be attached the private CSR, after the policy was attached a public CSR key could then be downloaded in a .csr file. This public certificate is what was entered into the SECRET_CERTIFICATE variable in the arduino_secrets.h file (see figure 7-2) [8].

With the MKR1010RFID thing setup correctly, card IDs would then be able to be published to the 'publishCard' MQTT topic (see figure 7-9).

Lambda Function:

AWS Lambda Function named 'rfid_iot_lambda' was used to pull the data from the IoT Core and insert the data into a MySQL table name 'iot'. This table would hold two columns, 'timestamp' and 'cardUID'. The timestamp would be created in the lambda code and the cardUID column would be filled in from the card ID data published to IoT Core. The lambda code connected to the MySQL RDS instance and the inserts the data into the database using SQL Queries.

index.js

```
var mysql = require("mysql");

exports.handler = async (event, context, callback) => {
    let connection = await mysql.createConnection({
        host: "f[redacted].rds.amazonaws.com",
        user: "admin",
        password:"[redacted]",
        port: 3306,
        database: "fypdb"
    });

    connection.connect();

    let str1 = JSON.stringify(event, null, 2);
    let data = JSON.parse(str1);
    let end = new Date();
    end.setHours(end.getHours() + 1);
    let format = { day: 'numeric', month: 'numeric', year: 'numeric', hour: 'numeric', minute: 'numeric', second: 'numeric' };
    let time = end.toLocaleString('en-GB', format);
    let cardId = data["cardUID"];
    let insert = 'INSERT into fypdb.iot (timestamp, cardUID) VALUES (\'' + time + '\', ' + '\'' + cardId + '\'' + ')';

    connection.query(insert, function (error, results, fields){
        if (error) throw error;
    });

    connection.end();
};
```

Figure 7-12 AWS JavaScript Lambda Code with SQL Queries

A 'Rule' had to be setup in the IoT core console for this to be accomplished. The rule had a simple SQL request which was 'SELECT * FROM 'publishCard''. This rule essentially means to select all data from the publishCard topic, this data would then be sent to the Lambda function. The IoT Core rule would have to be setup as a trigger in the Lambda function, meaning every time a card ID was published it would trigger the lambda function and send it the publish topic data.

This method proved effective, although there was a problem as each piece of published data was not being sent to the database. When the Lambda code was run, it failed occasionally and would not send the data into the database. This is where the SNS and SQS services played a big part in the Lambda Function.

An SNS topic was then setup. This SNS topic would be subscribed to the 'rfid_iot_lambda' function as well as being subscribed to an SQS Queue. A SNS trigger was then applied to the lambda function. There was also two destinations setup in the lambda function, a SNS destination on success and a SQS destination on failure.

The SQS was set as a failure destination, meaning if the Lambda code failed it would be sent into the queue. As the SQS service and lambda function were subscribed to the SNS topic, this meant the SNS topic would keep triggering the Lambda function until it successfully entered the data into the RDS database [9]. This is the reason there's a second destination (SNS) on success of a lambda function. With these additions to the Lambda function, the card scans now enter the database seamlessly on every scan.

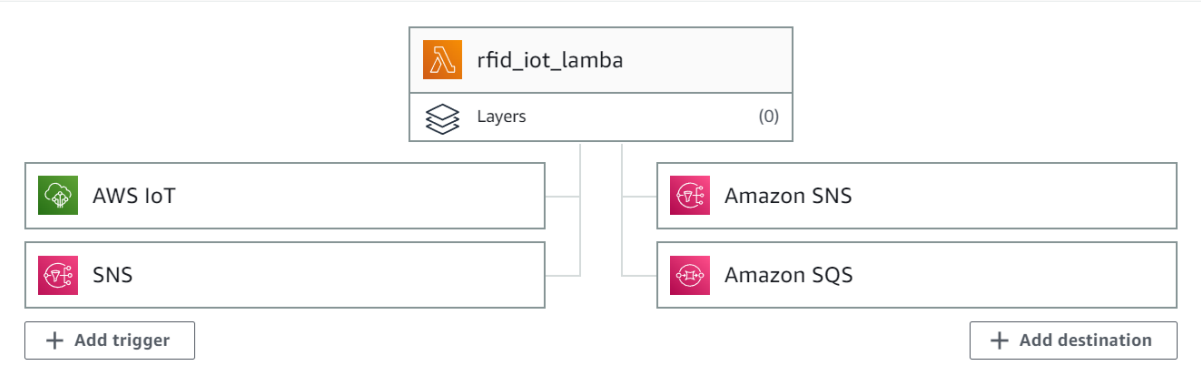


Figure 7-13 AWS Lambda Functions Triggers and Destinations

AWS RDS:

A MySQL instance was setup on the RDS service. The instance created is of size db.t2.micro. The database holds two table, one named 'employees' which stores all relevant employee information and a second name 'iot' which hold the card ID and times that they were scanned. An EC2 inbound security group was setup on port 3306 so that the RDS database could connected to from other services [10].

AWS Amplify:

AWS Amplify was used to deploy the web app for this project. Host the web app on Amplify is secure and prevents from running locally. To deploy the app, amplify will connect to the GitHub repository in which the full stack Next.js code is on and from there Amplify will build and deploy the Scan & Go web app.

7.3 Next.js Code

Next.js is the React framework used to create the full stack web application for this project. A login page will be presented when accessing the web app. Once an authenticated login is complete, a user will be brought to the main page, where the list of employee information can be viewed as well as a records table of door access attempts by the employees. The CRUD functionality of the web app can also be accessed on the web app, where employee information can be added, edited or deleted. The web application is deployed on AWS Amplify.

The main features of the web app can be found in the component's folder of the JavaScript code. All JavaScript code in the component's folder will be accompanied by a '.module.css' file of the same name to styles that component. The first component used will be login which will be called on the home screen of the application. After secure login, a user will be brought to the main ('MainNav.js') page where the 'EmployeeList', 'DoorAccessLog', 'EditEmployee' and 'AddEmp' components functionality can all be used.

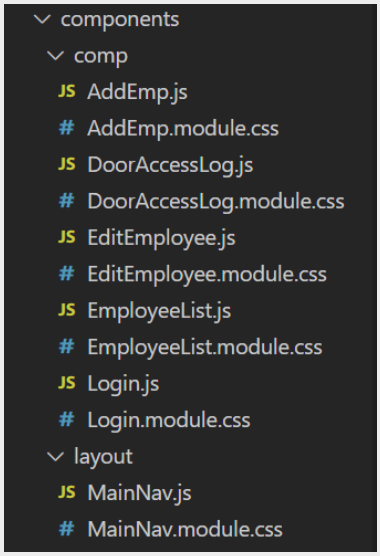


Figure 7-14 Components Folder

Backend:

The pages folder access's two Application Programmable Interfaces (API), one for Auth0 and one for the CRUD operations which will communicate with the MySQL database tables to perform queries. The final page is a server side main page which calls on the MainNav.js component and will execute its code.

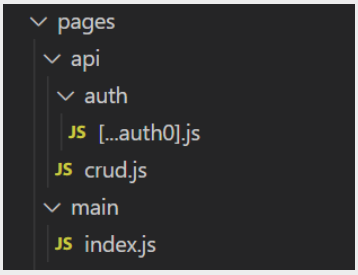


Figure 7-15 Pages Folder

All the Auth0 sensitive connection information will be setup in a '.env.local' file. From there a auth0.js file is created in the 'page/auth' folder. The file is put in square brackets with a spread operator which give access to Auth0's 'auth/login', 'auth/logout' and 'auth/callback' URLs in a single file. The UserProvider import from Auth0 must then be wrapped around the components in app.js to ensure all components are authenticated and protected.

```
pages > JS _app.js > ...
1   import '@/styles/globals.css'
2   import { UserProvider } from '@auth0/nextjs-auth0/client';
3
4   export default function App({ Component, pageProps }) {
5     return (
6       <UserProvider>
7         <Component {...pageProps} />
8       </UserProvider>
9     );
10  }
```

Figure 7-16 Auth0 in app.js

All SQL CRUD queries happen in the crud.js file. A connection with the AWS RDS database must first be completed [11]. This is done in db.js file in the config folder. The query function is asynchronous and waits for a connection to the database before continuing. The variable result is a promise which is returned on successful connection. If there is no connection, an error is thrown.

```
config > JS db.js > ...
1   import mysql from "mysql2/promise";
2
3   export async function query({query, value = []}){
4
5       const mySQLconnect = await mysql.createConnection({
6           host: process.env.SQL_Host,
7           user: process.env.SQL_User,
8           password: process.env.SQL_Pass,
9           port: process.env.SQL_Port,
10          database: process.env.SQL_Database
11      });
12
13      try{
14          const [result] = await mySQLconnect.execute(query, value);
15          mySQLconnect.end();
16          return result;
17      }catch(error) {
18          throw Error(error.message);
19      }
20  }
21
```

Figure 7-17 Connection to RDS instance

The query function is then imported into the crud.js file, this is so the queries executed in the code will be performed on the connected database. The file also imports withApiAuthRequired from Auth0, this import will be wrapped around the code function so that if a user is not signed in and the api route is accessed, access will be denied as the user is not signed in.

The first method requested in the handler function are the two GET requests, which are the read element of the CRUD operations. The first get query, 'employee', selects all the data from the employees table in the SQL instance. The second GET query, 'employee1', selects the employee information from the employees table, selects the card ID and timestamp from the iot table and joins them together. The 'employee' query will be used in EmployeeList.js to display employee information and 'employee1' will be used in DoorAccessLog.js. The two GET queries are then joined

in a combinedQuery variable so that both queries can be responded in JSON format using 'res.status(200).json' and used in the components files.

```
import { query } from "../../config/db";
import { withApiAuthRequired } from "@auth0/nextjs-auth0";

export default withApiAuthRequired(async function handler(req, res) {
  let message;
  if (req.method === "GET") {    // === comparison operator for strict equality between two values

    const employee = await query({
      query: "SELECT * FROM employees",
      value: [],
    })
    const employee1 = await query({
      query: "SELECT E.ID, E.Name, E.cardUID, I.timestamp, permission, Role FROM fypdb.iot I JOIN
      value: [],
    })

    const combinedQuery = {
      employee: employee,
      employee1: employee1
    }
    res.status(200).json(combinedQuery);  //responds json data of the employee table in api/crud
  }
```

Figure 7-18 crud.js Import & GET Requests

The POST request handles the create element of the CRUD operations. This will be used to add new employee name, card ID, job role and access permission for the card being assigned to them. It was important to set the values array of the queries in the same order as the query statement for the create operation to be successful. The 'req.body' in the const variables is requesting the name of the columns as they are written in the database. A success or error message will be given on execution of the SQL INSERT query.

```
if (req.method === "POST") {
  const employeeName = req.body.Name;
  const cardID = req.body.cardUID;
  const permission = req.body.permission;
  const role = req.body.Role;
  const addEmployee = await query({
    query: "INSERT INTO employees (Name, cardUID, permission, Role) VALUES (?, ?, ?, ?)",
    value: [employeeName, cardID, permission, role],
  });
  if (addEmployee.insertId) {
    message = "success";
  }
  else {
    message = "error";
  }
  let employee = {
    id: addEmployee.insertId,
    name: employeeName,
    code: cardID,
    perm: permission,
    role: role,
  };

  res.status(200).json({ response: { message: message, employee: employee } });
}
```

Figure 7-19 crud.js POST Requests

The PUT request is used for the update operation. This request will update the employee information by their employee list ID number, name, card ID and job role. This function holds a lot of the same parameters as the POST. It will return a success or error message on execution of the SQL UPDATE query.

```
if (req.method === "PUT") {
  const employeeID = req.body.ID;  //ID is same as sql column name
  const employeeName = req.body.Name; //Name is same as sql column name
  const cardID = req.body.cardUID;   //cardUID is same as sql column name
  const role = req.body.Role;
  const updateEmployee = await query({
    query: "UPDATE employees SET Name =?, cardUID=?, Role=? WHERE ID = ?;",
    value: [employeeName, cardID, role, employeeID],
  });

  const result = updateEmployee.affectedRows;
  if (result) {
    message = "success";
  } else {
    message = "error";
  }

  let employee = {
    id: employeeID,
    name: employeeName,
    code: cardID,
    role: role,

  };

  res.status(200).json({ response: { employee: employee, message: message } });
}
```

Figure 7-20 crud.js PUT Requests

The DELETE request is the final request. This request will delete an employee by their ID number in the employee list table. It will return a success or error message on execution of the SQL DELETE query.

```
if (req.method === "DELETE") {    // === comparison operator for strict equ

  const employeeID = req.body.ID;  //ID is same as sql column name

  const deleteEmployee = await query({
    query: "DELETE FROM employees WHERE ID = ?",
    value: [employeeID],
  });
  const result = deleteEmployee.affectedRows;
  if (result) {
    message = "success";
  } else {
    message = "error";
  }

  res.status(200).json({ response: { ID: employeeID, message: message } });
}
})
```

Figure 7-21 crud.js DELETE Requests

The final page, ‘main’, is a server side rendered page. This page executes the functions inside of the MainNav.js file, which will be the elements of the crud operations described above. This file imports withPageAuthRequired from Auth0, this import will wrap the code so that if a user is not signed in and the page route is accessed, a user will be prompted to login to continue.

```
import MainNav from '../../components/layout/MainNav'
import { withPageAuthRequired } from "@auth0/nextjs-auth0";

function mainNavPage() {

    function mainNavHandler(enteredCrudpData) {

    }

    return <MainNav onMain={mainNavHandler} />

}

export default mainNavPage
export const getServerSideProps = withPageAuthRequired();
```

Figure 7-22 Main Page

Frontend:

When the web app is first accessed, the user is met by a home page (index.js) with a page title and the project Scan & Go logo linked in the header, the Login component is then called (<Login />).

```
export default function Home() {

  return (
    <>
    <div className = {styles.container}>
      <Head>
        <title>Employee Logs</title>
        <link rel="icon" href="/Logo1.png" sizes='1002x1000' />
      </Head>
      <Login/>
      </div>
    </>
  )
}
```

Figure 7-23 Home Page – index.js

In the Login component, using the Auth0 API, the useUser hook provided by Auth0 allows for the Login page to check if the user is singed in, if there is an error or is the page is loading [12]. If the user is signed in, they will be welcomed and then be brought to the main page by clicking the 'Continue to Main' button.

If a user has not signed in yet, a button to login will be on screen and they will be redirected to the authentication page to securely login.

```
import { useUser } from "@auth0/nextjs-auth0/client";
import styles from './Login.module.css'


export default function Login() {
  const { user, error, isLoading } = useUser();


  if (isLoading)
    return <div className={styles.head}>....Loading</div>

  if (error)
    return <div className={styles.head}>{error.message}</div>

  if (user) {
    return (
      <>
        <div className={styles.container}>
          <div className={styles.main}>
            <h1 className={styles.head}>Welcome</h1>
            <h3 className={styles.success}>Sign in successful</h3>
            <a className={styles.button} href="/main">Continue to Main</a>
          </div>
        </div>
      </>
    )
  }
```

Figure 7-24 Login Page – useUser Hook functionality

```
return (
  <>
    <div className={styles.container}>
      <div className={styles.main}>
        <h1 className={styles.head}>Employee Logs</h1>
        <h3>Sign In</h3>
        <a className={styles.button} href="/api/auth/login">Login</a>
      </div>
    </div>
  </>
)
}
```

Figure 7-25 Login Page – Login prompt

Navigation to the main will happen after login. On the main page, there are four tabs, a logout button and a Scan & Go Systems header displayed on the screen. The four tabs open and on close using 'useState' hooks from React. In the example below, the four useStates used are shown and a function for showTableHandle. This handle sets the 3 of the 4 tabs to false which means they wont be shown and the setShowTable is set to '!showtable', here the ! operator shows the opossite value of showTable. So if SetShowTable is false it will be set to true and vice versa.

This function will be sent into an onClick handler in a button component. Once clicked it will open the DoorAccessLog.js file and display all the contents of that file until the button component is clicked again or if another one of the 4 buttons are clicked on. This process to display the 'DoorAccessLog' file is rinsed and repeated to display the 'EmployeeList', 'EditEmployee' and 'AddEmp' files from there corresponding display button. When the Scan & Go Systems header is clicked, any tabs opened will be closed. The logout link is setup with the Auth0 API and when it is pressed, the user will be securely logged out and directed back to the login page.

```
const [showTable, setShowTable] = useState(false);
const [showEmpList, setEmpList] = useState(false);
const [showAddEmp, setAddEmp] = useState(false);
const [showEditEmp, setEditEmp] = useState(false);

const showTableHandle = () => {
 setShowTable(!showTable);
 setEmpList(false);
 setAddEmp(false);
 setEditEmp(false);
}
```

Figure 7-26 MainNav – useStates and DoorAccessLog Display Function

```
function goHome(){
  router.push('/main')
  setShowTable(false);
  setEmpList(false);
  setAddEmp(false);
  setEditEmp(false);
}
```

Figure 7-27 MainNav – Function When Header is Clicked

```
return (
  <>
  <section className={styles.container}>

  <header className={styles.header}>
    <div className={styles.logo} onClick = {() => goHome()} >Scan & Go Systems</div>
    <nav>
      <ul>
        <li>
          <Link href='/api/auth/logout'>Logout</Link>
        </li>
      </ul>
    </nav>
  </header>

  <div className= {styles.selection}>
    <button className= {styles.database} onClick={showTableHandle}>Door Logs</button>
    <div>{showTable && <DoorAccessLog />}</div>
    <button className= {styles.database} onClick={empListHandle} >Employee List</button>
    <div>{showEmpList && <EmpList />}</div>
    <button className= {styles.database} onClick={addEmpHandle} >Add Employee</button>
    <div>{showAddEmp && <AddEmp />}</div>
    <button className= {styles.database} onClick={editEmpHandle} >Edit Employee</button>
    <div>{showEditEmp && <EditEmp />}</div>
  </div>
```

Figure 7-28 Main Page – Return of MainNav.js

In EmpList file, the employee list will be displayed here. To do so we need the async function which fetch the GET method from the 'employee' query in the crud API. When the query is fetched, it will be entered in the setEmployee useState array. The response from the query will be mapped in the return function and setup in a list form.

The DELETE method will also be fetched and using react icons, the delete operation can be completed by assigning its operations to the icon. This will delete the Employee by the ID its next to.

The same functionality was performed for the DoorAccessLog file, although it uses the GET method from 'employee1' query instead.

```
import {CiTrash} from 'react-icons/ci'
import styles from '../comp/EmployeeList.module.css'
import { useState, useEffect } from 'react'

export default EmpList;

function EmpList(){

  const [employee, setEmployee] = useState([]);
  const [deleted, setDelete] = useState(false);

  async function getEmployee(){
    const getEmpdata = {
      method: "GET",
      headers: {
        "Content-Type": "application/json",
      },
    };
    const res = await fetch('https://main.d2xu1i4qh95c6u.amplifyapp.com/api/crud',
    getEmpdata
    );
    const response = await res.json();
    setEmployee(response.employee);
  }
```

Figure 7-29 EmpList – Fetch GET

```
async function deleteEmployee(id){
  if(!id) return;
  const deleteEmpdata = {
    method: "DELETE",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify({
      ID: id,
    })
  };
  const res = await fetch('https://main.d2xu1i4qh95c6u.amplifyapp.com/api/crud',
  deleteEmpdata
  );
  const response = await res.json();
  if(response.response.message !== "success") return;
  setDelete(true);
  const removeId = parseFloat(response.response.ID);
  setEmployee(employee.filter((d) => d.ID !== removeId));
}

useEffect(() => {
  getEmployee();
}, []);
```

Figure 7-29 EmpList – Fetch DELETE

```
return (
  <>
    <div className={styles.container}>
      <section>
        <div className={styles.read}>
          <h2>Employee Information</h2>
          <div className={styles.list}>
            {employee.map((item, index) => {
              return (
                <div key={item.ID} className={styles.empList}>
                  <span> Employee ID: </span> {item.ID} |
                  <span> Name: </span>{item.Name} |
                  <span> Role: </span>{item.Role} |
                  <span> Card ID: </span>{item.cardUID} |
                  <span> </span><span> </span>
                  <CiTrash className={styles.icon}
                    onClick={() => deleteEmployee(item.ID)}
                  />
                </div>
              );
            })}
            {!employee.length ? <>No Products found</> : null}
          </div>
        </div>
      </section>
    </div>
  </>
```

Figure 7-30 EmpList – Return Mapping List

The AddEmp file has useRef hooks to reference a field when completing an add employee form in the web application. The POST request will be fetched from the crud API and will add a client once the necessary fields are filled out in the form. If the form is filled correctly, the create employee useState hook will be set to true, a ‘success!’ response will be given and the new employee will be added to the employee list. The same functionality will be used for the update method, but instead it will be the PUT method being fetched.

```
  const postEmpdata = {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify({
      Name: employeeName,   //Name is name of column in sql database
      cardUID: employeeCardId,
      permission: empPermission,
      Role: empRole
    }),
  };
  const res = await fetch('https://main.d2xu1i4qh95c6u.amplifyapp.com/api/crud',
    postEmpdata
  );
  const response = await res.json();
  if (response.response.message != "success") return;

  const newEmployee = response.response.employee;
  setEmployee([   //update data right away
    ...employee,
    {
      ID: newEmployee.ID,
      Name: newEmployee.Name,
      cardUID: newEmployee.cardUID,
      Role: newEmployee.Role
    },
  ]);
  setCreate(true);
}
```

Figure 7-31 AddEmp – fetch

```
return (
  <>
    <div className={styles.container}>
      <section>
        <div className={styles.create}>
          <h2>Add New Employee</h2>
          <div className={styles.input}>
            <h3>Name:</h3>
            <input className={styles.label} type='text' ref={nameRef} />
            <h3>Employee Role:</h3>
            <input className={styles.label} type='text' ref={roleRef} />
            <h3>Card ID:</h3>
            <input className={styles.label} type='text' ref={cardIdRef} />
            <h3>Card Permission:</h3>
            <select className={styles.label} type='text' ref={permissionRef}>
              <option></option>
              <option>Accepted</option>
              <option>Denied</option>
            </select>
          </div>
          <div>
            <input
              className={styles.button}
              value='Save'
              type='button'
              onClick={addEmployee}
            />
          </div>
          {create ? <div className={styles.success}>Success!</div> : null}
        </div>
      </section>
    </div>
  </>
)
```

Figure 7-32 EmpList – Return

8 Ethics

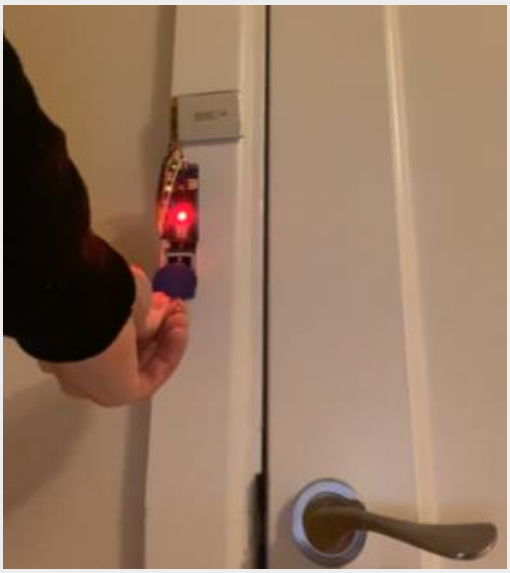
With regards to ethics in this project, the main goal was to have employee's sensitive data protected. The goal was to also not discriminate between race or gender of employees, which therefore is why they were not included in the employee information. The name, card ID, job role and door permissions seemed the most necessary pieces of information.

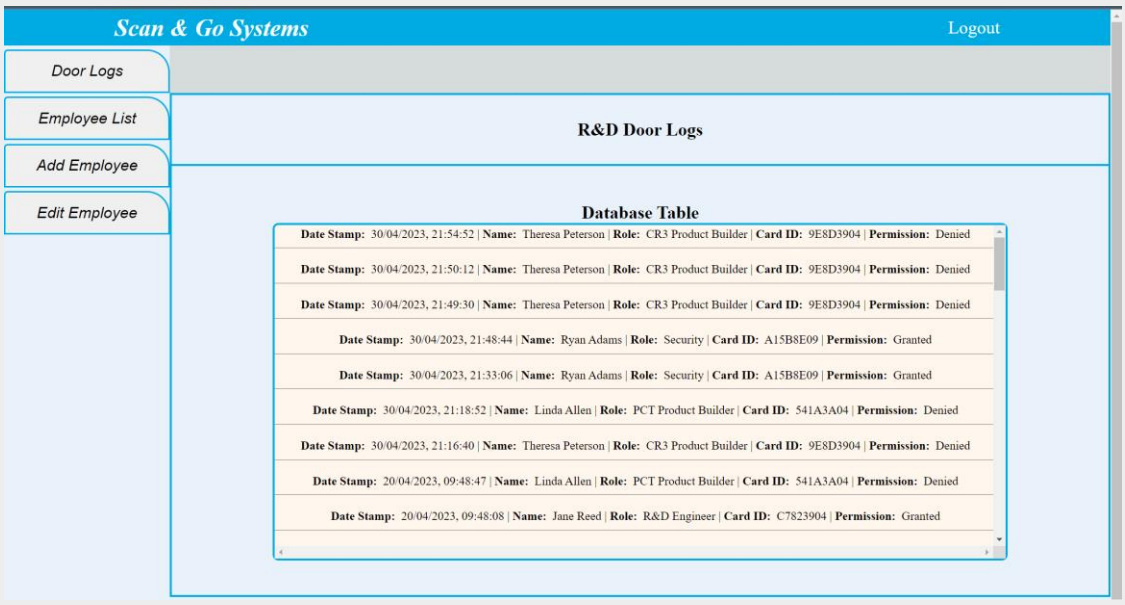
The authentication was also a big part of the ethics of this project. It would be unethical and highly dangerous to not have authorization over the web app, resulting in a massive breach of employee's sensitive information. The sign in features and route protection help to protect against this.

9 Conclusion

In conclusion, the project has delivered on its requirements. There is a system to read a card ID, the card ID can be sent securely from an encrypted board to the cloud and that card data can be sent into its necessary database table securely. All the CRUD operations perform as should, and the full stack web application is highly secure with its use of authentication.

This project was highly challenging. I have learned a lot of technical skills throughout this year as well as project management and teamwork skills. I would like to thank you for taking the time to read my Scan & Go final year project report.





10 References

[1] Arduino, "MKR WiFi 1010 Arduino Domunetation," PythonProgramming, [Online]. Available: https://docs.arduino.cc/hardware/mkr-wifi-1010.

[2] NXP, "MFRC522 Standard performance MIFARE and NTAG frontend," PythonProgramming, [Online]. Available: https://www.nxp.com/docs/en/data-sheet/MFRC522.pdf.

[3] Guru99, "MongoDB vs MySQL" [Online]. Available: https://www.guru99.com/mongodb-vs-mysql.html.

[4] FreeCodeCamp, "What Exactly is Node.js" [Online]. Available: https://www.freecodecamp.org/news/what-is-node-js/.

[5] GitHub, "arduino-libraries/ArduinoECCX08" [Online]. Available: https://github.com/arduino-libraries/ArduinoECCX08.

[6] GitHub, "rfid/examples/ReadAndWrite" [Online]. Available: https://github.com/miguelbalboa/rfid/tree/master/examples/ReadAndWrite.

[7] GitHub, "ArduinoCloudProviderExamples" [Online]. Available: https://github.com/arduino/ArduinoCloudProviderExamples/tree/master/examples/AWS%20IoT/AWS_IoT_WiFi.

[8] Ardunio Documentation, "Securely Connecting an Arduino MKR WiFi 1010 to AWS Iot Core" [Online]. Available: https://docs.arduino.cc/tutorials/mkr-wifi-1010/securely-connecting-an-arduino-mkr-wifi-1010-to-aws-iot-core.

[9] Plain English, "AWS SNS Events: SNS to Lambda vs SNS to SQS to Lambda" [Online]. Available: https://plainenglish.io/blog/aws-sns-to-lambda-vs-sns-to-sqs-to-lambda-788d4cc96f34.

[10] AWS re:Post, "Configure a Lambda function to connect to an RDS instance" [Online]. Available: https://repost.aws/knowledge-center/connect-lambda-to-an-rds-instance.

[11] GitHub, "olbega/nextjs-crud-mysql" [Online]. Available: https://github.com/oelbaga/nextjs-crud-mysql.

[1 GitHub, "auth0/nextjs-auth0: Next.js SDK for signing in with Auth0" [Online]. Available:
2] https://github.com/auth0/nextjs-auth0.