



TeamSwipe Payment Manager

Project Engineering

Year 4

Liam Nilan

Bachelor of Engineering (Honours) in Software and

Electronic Engineering

G00366621

Galway-Mayo Institute of Technology

2021/2022

Declaration

This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering (Honours) in Software and Electronic Engineering at Galway-Mayo Institute of Technology.

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

Liam Nilan

Acknowledgements

I would like to thank my lecturers Michelle Lynch, Niall O'Keffe, Paul Lennon and my supervisor Brian O'Shea. I am very grateful for all the help and support to complete this project. I would also like to thank JLC PCB for manufacturing the PCB board for my project.

Table of Contents

| | | |
|-----------|----------------------------------|-----------|
| 1 | Summary | 6 |
| 2 | Poster..... | 7 |
| 3 | Introduction | 8 |
| 4 | Background | 12 |
| 5 | Project Architecture..... | 9 |
| 6 | Project Plan | 10 |
| 7 | Project Technologies..... | 11 |
| 7.1 | Frontend..... | 11 |
| 7.2 | Backend | 12 |
| 7.3 | ESP32 | 12 |
| 7.4 | Amazon Web Services | 12 |
| 8 | Frontend Design..... | 12 |
| 8.1 | React..... | 12 |
| 8.2 | NextJs | 13 |
| 9 | Backend Design | 18 |
| 9.1 | Learning NodeJS | 18 |
| 9.2 | NodeJS implementation | 18 |
| 9.3 | NodeJS Authentication..... | 19 |
| 9.4 | NodeJS MQTT | 20 |
| 10 | ESP32 Device | 22 |
| 10.1 | Original concept..... | 22 |
| 10.2 | RFID reader | 22 |
| 10.3 | Manager Menu | 23 |

| | | |
|-----------|----------------------------------|-----------|
| 10.4 | Implementing Firebase | 25 |
| 10.5 | ESP32 PCB | 25 |
| 10.6 | Implementing AWS IoT | 27 |
| 11 | Amazon Web Services | 29 |
| 12 | Ethics | 30 |
| 13 | Conclusion | 31 |
| 14 | Appendix | 32 |
| 15 | References | 33 |

1 Summary

My project is an employee payment management system for businesses. It consists of an online webapp and an RFID clock-in clock-out machine. The goal of this project is to create a functional clock-in clock-out machine that connects to a cloud based database with a webapp to manage that database. It will also follow a MERN architecture.


This project consists of a NextJS frontend, a NodeJS backend and an ESP32 clock-in clock-out machine. The frontend and backend are hosted on an EC2 instance of Amazon web services and the ESP32 connects to it through IoT core and MQTT.

Some of the features this project has are password login and authentication, a pay calculation microservice and separate menus for managers and employees. I also designed a custom PCB for the ESP32 and its components.

This project was developed following an agile philosophy. I used JIRA to manage stories that were connected to four epics. I had an epic for the frontend, backend ESP32 and any other project work that needed to be done. Progress on these epics were tracked on a timeline.

Overall I am happy with how this project turned out. The website worked and ESP32 worked as planned and are able to communicate flawlessly over the cloud. I think with some further development this could become a viable product.


2 Poster



GMTI
INSTITUTE OF TECHNOLOGY
GALWAY MAYNO INSTITUTE OF TECHNOLOGY

TeamSwipe Employee Payment Manager

Course: BENG (H) In Software and Electronic Engineering
By Liam Nilan Year: 2021 - 2022



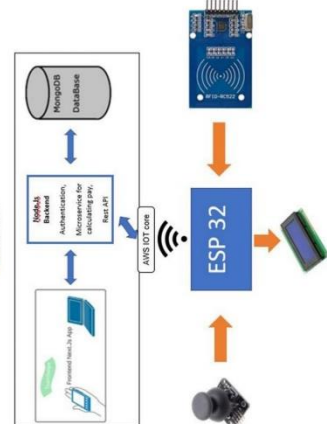
Project Summary

TeamSwipe Employee Payment Manager is a clock-in clock out machine with a website for managing hired employees. Businesses register for the service and add employees through the web app. They then can write the employee's data to a RFID card or key fob so the employee can tap into work.




This is a hardware and software project linked together by the cloud. The hardware side of things is an ESP32 microcontroller on a printed circuit board. On the software side of things there is a Next.js frontend with a Node.js backend connected to a mongoDb database.

I intend to host this project on an EC2 instance of Amazon Web Services and use the AWS IOT core to call and update data from the ESP32.

Architecture Diagram



Technologies

Languages

| | |
|--------------|--------|
| ➤ C | ➤ HTML |
| ➤ JavaScript | ➤ CSS |

Backend Design

The backend for this project consists of three parts. The main server, the authentication server and a mongoDb database. The main server connects to the mongo database and hosts the microservices such as calculating the pay for each employee.

The authentication server manages login and the creation of JWT access tokens and refresh tokens.

Each business has a JSON on the database storing all their data. It contains the email and password and two sub arrays containing their employees and employee presets. Eg. pay rates and roles.

A valid access token is needed with any user data request to fetch data to the frontend.


Conclusion

This project delivers in the goals I set to achieve. Through this project I designed developed and tested front end and back end website design. I Pushed the boundaries of the ESP 32 to successfully demonstrate the clock-in clock-out device in the lab.


All the calls to the backend and database have worked without flaws.

All code for this project can be found with this link: github.com/LiamNilan599/LN_Final_Year_Project

Screenshot of the TeamSwipe Webapp home page



Picture of the TeamSwipe Clock-in clock-out device



3 Introduction

My project is the TeamSwipe payment management system. It is a service that businesses can sign up for where they can hire and fire employees. They are also given a clock-in clock-out device so their employees can sign into work. Management get special RFID tags or cards to use on the device to write and wipe other tags or cards.

The idea of this project was inspired from a restaurant I used to work at. There was a clock-in clock-out machine there for the employees. I thought of a way to use RFID tags to sign in and out. I researched existing devices and found none of them had RFID and they all used some sort of biometric data like facial recognition and fingerprint scanning.

The project could be broken down into three parts. The NextJs frontend, the NodeJS backend and the ESP32. Each of these will have their own sections in this report. The frontend and the backend are hosted on an EC2 instance of Amazon web services and the ESP32 device will connect using the AWS IoT core.

The goal of the project is to have a functional system where a business can open an account and add their employees and then create sign in cards or tags that the employees can use on the ESP32 clock in device. With this project I want to display my ability to write code for software and hardware and cover the skills I have learned over the last four years.

This report will show the technologies I used and their implementation into this project. I will also go over the evolution of the three main parts of this project and how they all came together in the end.

4 Project Architecture

As stated in an earlier section, this project consists of three parts. The frontend, the backend and the ESP32. The NextJS frontend and the NodeJS backend are hosted on an EC2 instance of AWS. MongoDB is also installed so the backend has access to the database. The ESP32 device connects to the EC2 instance through AWS IOT core. This is an MQTT service that the ESP32 will pull the list of employees and their roles from the mongo database through the NodeJS backend.

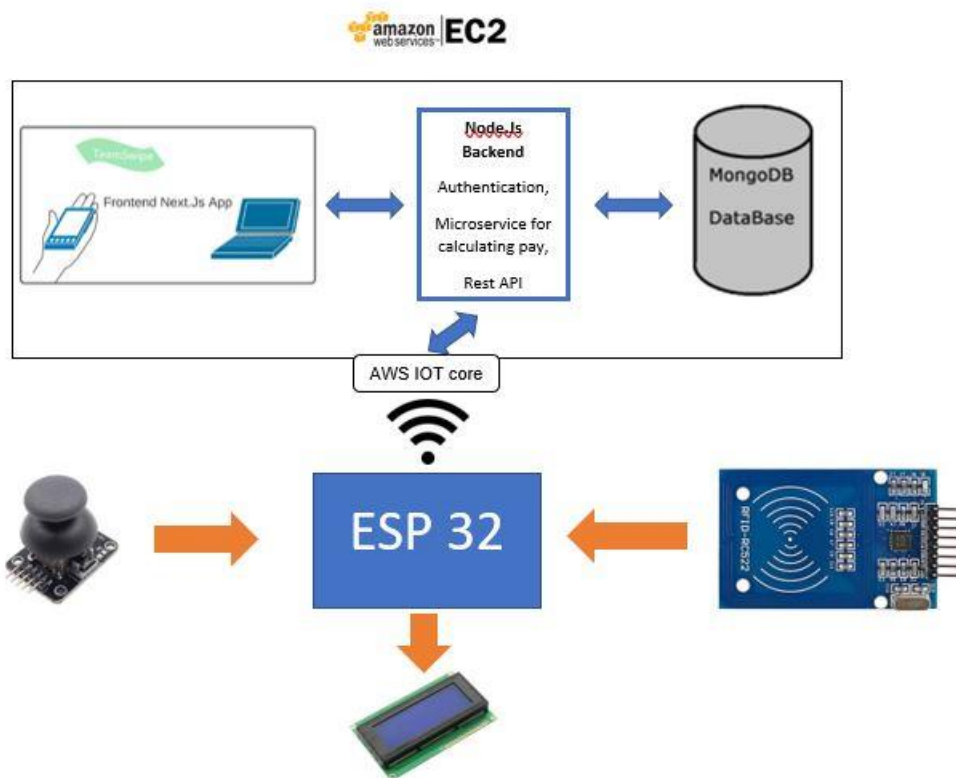


Figure 5-1 Architecture Diagram

5 Project Plan

The project plan was an essential part of the project. Early in the project we were tasked to come up with a timeline for the whole project that we would review after Christmas.

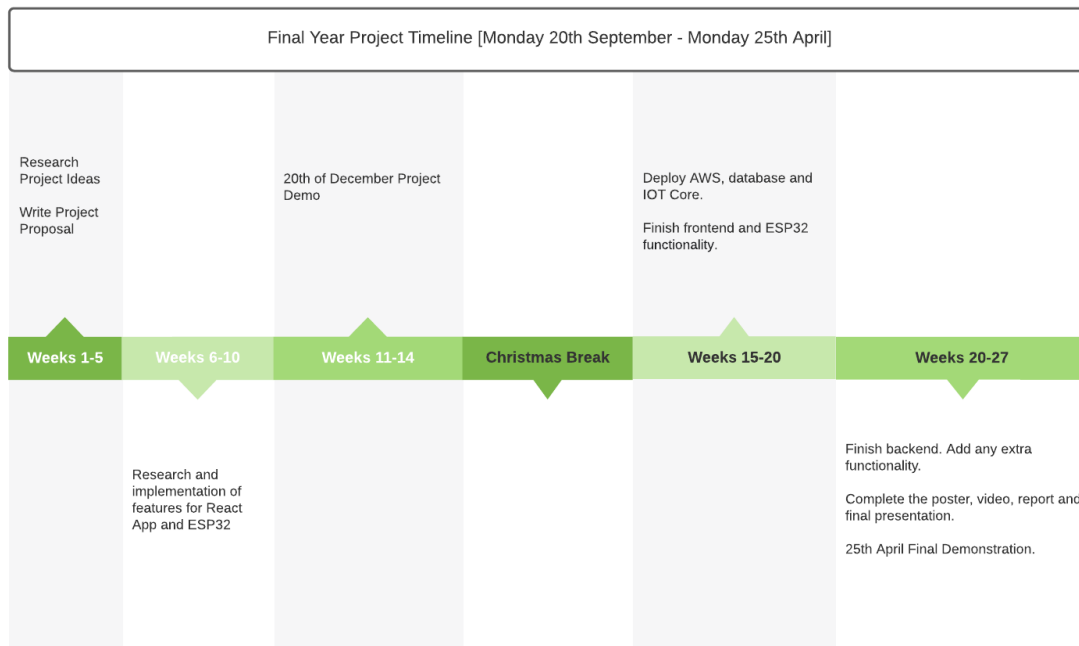


Figure 6-1 Original Project plan

After Christmas I refocused my project planning. I started using a project management software called Trello but I quickly switched to JIRA. On JIRA I created 4 epics for each aspect of the project. One for the frontend, one for the backend one for the ESP32 and one for other project tasks such as the poster, video and project report. I set the starting date for both the frontend epic and ESP32 epic at the 1/1/22 because I had started work on those before Christmas. I set the start point for the backend to the 17th of January because that's when I actually began it.

I would open tickets for each task that needed to be done. For example I opened a ticket to test connecting mongodb to the backend and I estimated the time it would take to finish it.

On the road map page each epic has a line under it indicating how many of its stories are completed. Green represents completed stories and dark blue represents in progress.

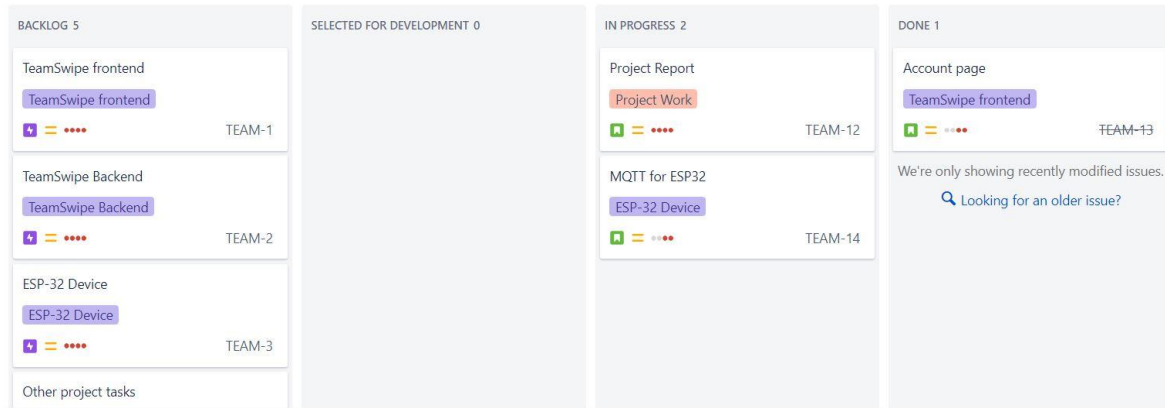


Figure 6-2 TeamSwipe Kanban page

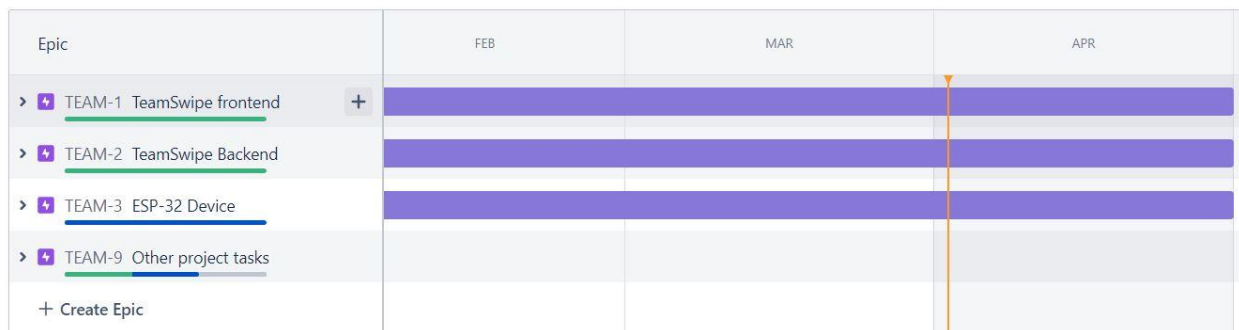


Figure 6-3 TeamSwipe Roadmap

6 Project Technologies

I have used a wide range of technologies in this project. The website uses a MERN stack (Mongodb, Express, ReactJs/NextJS and NodeJS). I will break them down by section.

6.1 Frontend

The technologies I used in my frontend are NextJS and ReactJs. ReactJs is a component based javascript library for developing webapps. The frontend was originally written in react because we were learning it in our cloud computing class. Pages are programmed in javascript, CSS, HTML and JSX. I then upgraded it to NextJS, which is also a javascript library. NextJS has all the features that react has but is more optimised for search engines. Unlike react it also uses file based routing to navigate between pages.

6.2 Backend

The backend of this project uses two NodeJS servers. NodeJS is a javascript runtime environment for backend servers. One server is used for authentication, registration, login and logout. The other handles most data being set to or requested from the Mongodb database using mongoose. On these servers I have express installed for routing requests from the frontend. Most requests use JWT (JSON Web Tokens) tokens to handle requests.

6.3 ESP32

The technologies the clock-in clock-out device is using are MQTT, JSON and AWS IoT core. It uses MQTT to communicate with IoT core. The EC2 instance that the frontend and backend run on is also connected to IoT core and is subscribed to the topics the ESP32 publishes. Through MQTT it receives a JSON document array as a string. This string is parsed so parts of it can be used on the display and write to RFID cards. When a card is tapped data is published to IoT core that signals if the employee is clocked in or out.

6.4 Amazon Web Services

The frontend and the backend are hosted on an EC2 instance. From there port 3001 is accessible to anyone who enters the public IPV4 address with port 3001 added to the end. Amazon Web services also has IoT core which is an MQTT service. This will act as a broker between my ESP32 device and my project's backend.

7 Frontend Design

The frontend of my project is a vital part of my final year project. It is the first thing a user would see if my project was a commercial product. In this section I will go through the design history of the frontend and how it evolved into the final version.

7.1 React

The TeamSwipe frontend began development in the last week of September 2021. I started it by following a youtube tutorial by Maximilian Schwarzmüller on his channel "Academind" [1]. His project was a meetup web app where users could enter a place, an image and a description and then add that to a favourites page through react context. He also used Google Firebase as a makeshift backend. From that project I modified the meetup form to accept name, role, age

etc. I also modified the navigation bar to fit my project. Soon after it was successfully writing employees to the Firebase backend. I added an employees page with a table inside and a button to create a new employee. This button opened a modal with a form on it to enter the data. I followed another youtube tutorial by Chris Blakely on his channel “Chris Blakely” [2] to create the table that had editable fields. I added a verification function on both the form, modal and table to make sure the data entered in each field was correct.

| Name | Age | Role | IDPSN | Hourly Wage | Actions |
|-------------|-----|----------------------|----------|-------------|---------------|
| Harry | 56 | Manager | 55589919 | 200 | Edit Delete |
| Janice | 34 | HR | 55567218 | 80.65 | Edit Delete |
| Bill Hardin | 43 | Accountant | 09878923 | 100 | Edit Delete |
| Oriath | 22 | Secretary | 45612398 | 34.5 | Edit Delete |
| Rory | 44 | Sales Representative | 22468865 | 34.5 | Edit Delete |
| Henry | 22 | Security | 23456543 | 30.3 | Edit Delete |

New Employee

Figure 8.1-1 Employees page with table

I soon began work on the login page. It was split into two columns. One with a slide show and the other with a login form. I followed an online tutorial to use a prebuilt react library for the slideshow [3]. It later got replaced with one that worked with plain react [4].

In late January and early February I had the backend implemented and I had to change how where the fetch requests got data from. I changed the requests to get data from a NodeJS server instead of google Firebase. I had to add headers to the fetch requests so the backend would recognize the data in the requests as JSON objects. At the time I had to add a CORS header to give the request data permission to move between react and NodeJS.

7.2 NextJS

After learning a bit of NextJS in my cloud computing module I decided to upgrade my frontend to use it.

The transfer of all the components and pages from React was easy. I came across two problems when I changed to NextJS. The table in the employees page and login page slideshow broke

because NextJS does not work well with some CSS code. I fixed this by putting that CSS code in a global CSS file used across the project. Once these were fixed I added context API to the project. I needed it to show the number of employees on a badge on the navigation bar. I then began working on authentication.

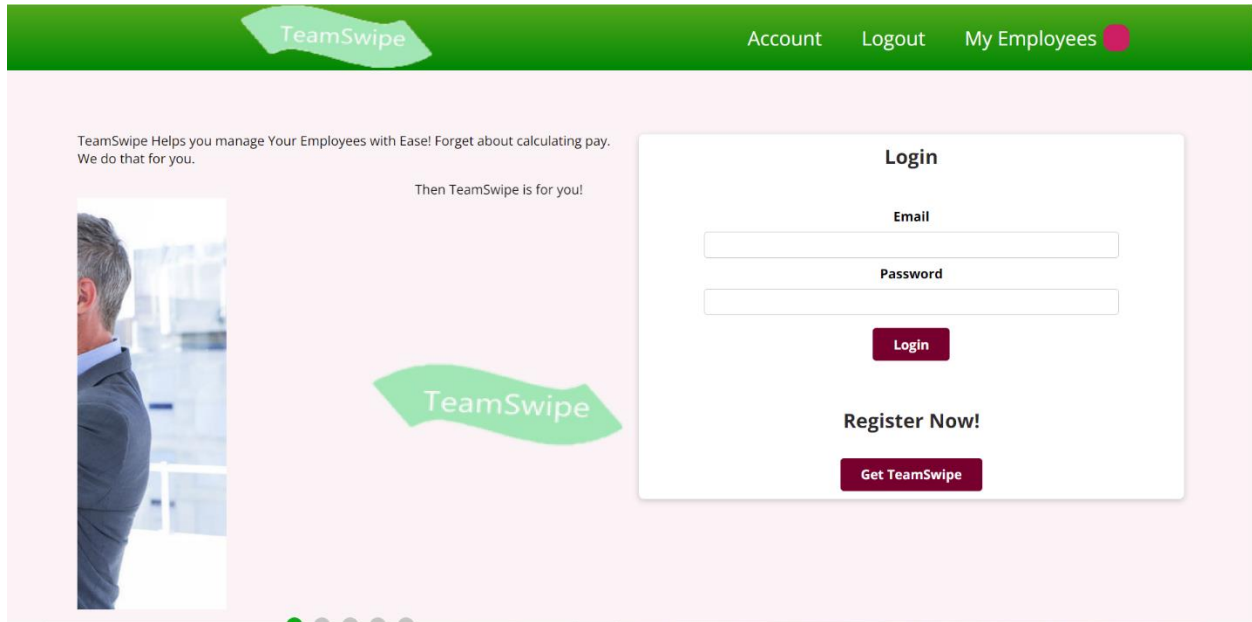


Figure 8.2-1 Home page before NextJs



Figure 8.2-2 Home page with NextJs

While most of the work for this was done on the backend some work was needed to be done on the frontend. The first thing that needed to be done was to make a register page. On this page is a simple form looking for the company email and the details of one manager. At the end of the form is the password. The password gets encrypted in the backend to allow for login.

Register

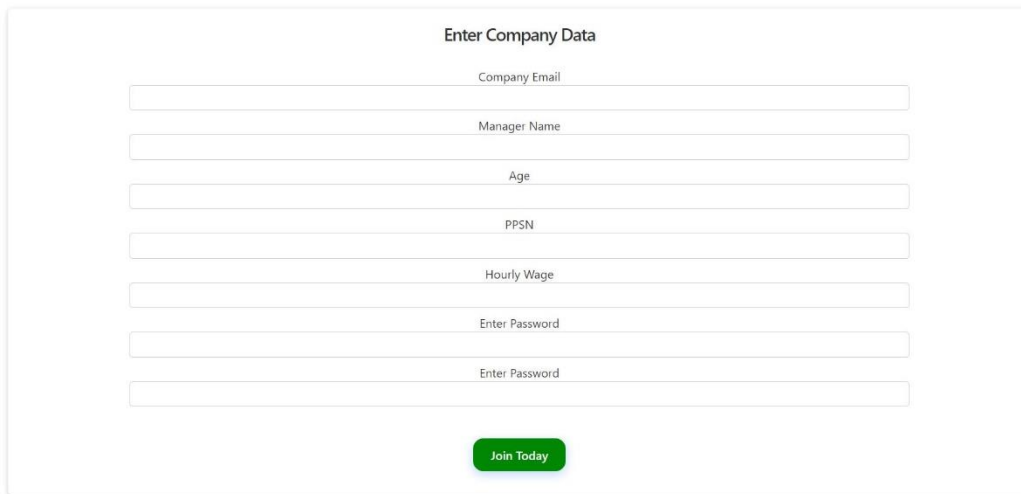
A screenshot of a web form titled "Enter Company Data". The form contains several input fields: "Company Email", "Manager Name", "Age", "PPSN", "Hourly Wage", "Enter Password", and another "Enter Password" field. A green "Join Today" button is at the bottom.

Figure 8.2-3 Register page

After lots of testing on the backend I started attempting sending login and register requests from the frontend. For a while I was getting back errors. Once this was fixed I could log in without any trouble. I had created a context variable to make sure I stayed logged in. This worked until I refreshed and it would be cleared. I decided to save the JWT (Json Web Token) access token to local storage where it would be cleared when the user logged out.

I then began work on the account page. On this page there is a small form for adding preset employee details that you can choose for auto fill in the new employee form. These are shown on a table to the right of the form where they can be deleted. At the bottom of the page is the delete account button. Pressing this opens a modal asking for the password to be entered twice.

Account

Employee Presets

Role:

Wage:

+

Your Presets

| Preset | Wage | |
|----------------------|------|--------|
| Sales Representative | 34.5 | Delete |
| Security | 30.3 | Delete |
| Accountant | 100 | Delete |

Delete Account

Delete

Figure 8.2-4 Account page

Delete Account?

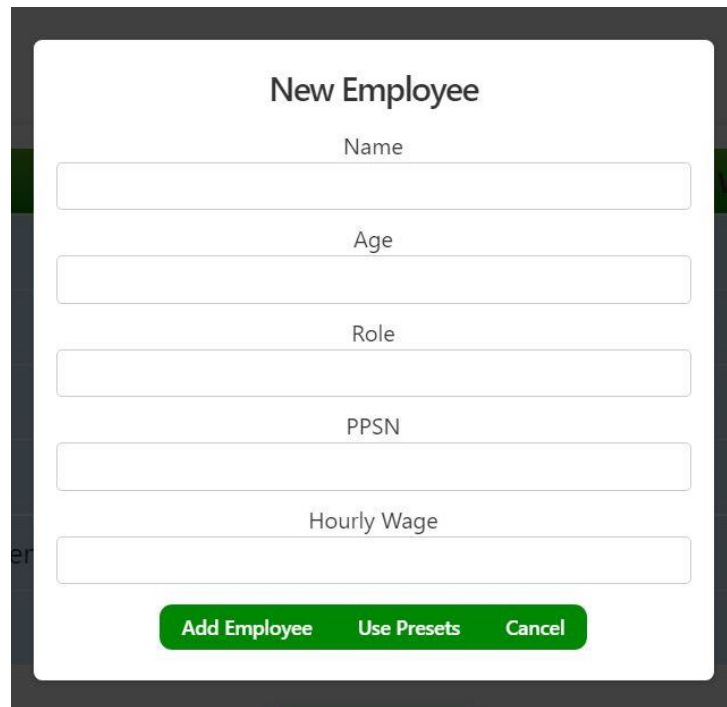
Enter Password

Enter Password again

Confirm Delete Cancel

Figure 8.2-5 Delete account modal

When a preset is created it would be sent to the backend and written to the database and it would be stored in a context array on the frontend. On the add employee modal I added a button that would appear if there were presets in the context array. Pressing this “use presets” button would show a selection of buttons for the user to choose what role preset they wanted.



The image shows a modal window titled "New Employee". Inside the modal, there are five text input fields stacked vertically, each with a label above it: "Name", "Age", "Role", "PPSN", and "Hourly Wage". Below these input fields, there is a green bar containing three buttons: "Add Employee", "Use Presets", and "Cancel". The modal is set against a dark background.

Figure 8.2-6 Employee modal with presets button

Since I was only storing the presets in context the array would reset to empty if you refreshed on any page except the account page. To fix this I needed to implement API routes to the backend. The fetch to get the presets was moved to the context file so it would be called every time context was used in any file. That way the array would be restored after a refresh. I had a good bit of trouble implementing this because I was using fetch commands where it should have been calling the data from the context array. I then had an error where presets were being created with no data inside. This was fixed by adding a "Content-Type: application/json" header.

When all the pages were completed only minor changes were needed to get the frontend working on Amazon web Services. All the fetch commands had to be changed to use API routes instead of calling the backend directly.

8 Backend Design

The backend of my project is the backbone for both the NextJs frontend and the ESP32 device. It handles any requests to the mongodb database made by the ESP32 or the frontend. It has two servers. One that handles login and register and the other that handles almost all of the database calls. In this section I will go through the design history of the backend and how it evolved into the final version.

8.1 Learning NodeJS

In late January 2022 I began researching ways to implement NodeJS into my React project. I followed a YouTube playlist of NodeJS tutorials by Maximilian Schwarzmüller on his channel “Academind” [5] as suggested by my cloud computing lecturer. I watched all the videos relevant to database manipulation and http request handling.

8.2 NodeJS implementation

My NodeJS project is based off one of Maximilian Schwarzmüller’s tutorials. The tutorial contained a basic backend showing the database manipulations you could do to a mongodb database. It used http requests like GET, POST and DELETE to edit the data on the database. It had some handlebars files that made up a makeshift frontend. On the index.hbs file there was a form for entering user data. I edited this project’s frontend to add form fields that were the same as my add employee module. I used this to test if data was going into the database correctly. It wasn’t long before I was successfully getting the list of employees from the NodeJS backend.

When it came to sending POST requests from the frontend the data was not sending. Since I was making cross origin requests I needed to add CORS to my backend. All I needed to do then was add “mode: 'cors' ” to the fetch POST requests on the frontend.

After finishing testing with mongodb I upgraded my NodeJS backend to used Mongoose. This was a major improvement because I could use schemas and models to make sure the data is correct. They made it clearer how data was got from the database and rendered as a JSON document.

```

var Schema = mongoose.Schema;

var EmployeeSchema = new Schema({
  name: String,
  age: Number,
  role: String,
  ppsn: String,
  wage: Number
}, { collection: 'user-data' });

var EmployeeData = mongoose.model('EmployeeData', EmployeeSchema);

router.get('/', function (req, res, next) {
  res.render('index');
});

router.get('/get-data', function (req, res, next) {
  EmployeeData.find().lean().exec(function (err, doc) {
    res.json(doc);
  });
});

```

Figure 9.2-1 Original Mongoose Schema on the Backend

8.3 NodeJS Authentication

In mid-February I started working on authentication for the backend. I followed two YouTube tutorials on authentication and login by Kyle Cook on his channel “Web Dev Simplified” [6] [7]. I used two NodeJS servers for this. I installed a package called nodemon so I could create scripts to run two servers from the same directory. The authentication server runs on port 4000 and manages login, logout and registration.

When a user logs in they use the company email and a password. The company is pulled from the database and the password sent from the request and the encrypted password saved in the database are compared. If the passwords match an access JWT (JSON Web Token) token and a refresh JWT token get generated using the company email and an environment variable. The refresh token gets stored in an array and gets deleted from that array when the user logs out. Registering a new user works in a similar way except it checks the database for a user of the same name. If there is none it creates one and creates an access token and refresh token. The access token lasts 15 minutes before it expires.

The main NodeJS server receives the access token from bearer header in all the http requests. This is then processed by an authentication middleware function. This function parses out the access token and verifies it against the environment variable. From this it gets the company email that was serialized into the access token and returns that to the http request. From there it is used to get data from the database.

To test this I installed an extension to Visual Studio Code called REST Client. This allowed me to make http requests from within Visual Studio Code without having to download an external program like Postman. All I had to do was make a requests.rest file and write my requests in it and run them.

The schema on the backend had to change too. Instead of having a list of employees it instead needed to have a company field and a password field and an employees field containing an array of employee JSON documents.

```

11 var Schema = mongoose.Schema;
12
13 var EmployeeSchema = new Schema({
14   name: String,
15   age: String,
16   role: String,
17   ppsn: String,
18   wage: String
19 });
20
21 var UserSchema = new Schema({
22   company: String,
23   password: String,
24   employees: [EmployeeSchema]
25 }, { collection: 'user-data', versionKey: false });
26
27 var UserData = mongoose.model('EmployeeData', UserSchema);

```

Figure 9.3-1 Schema after authentication

8.4 NodeJS MQTT

The last part of the backend I worked on was implementing MQTT into the backend. This was needed so the ESP32 could get data from the database through the backend and AWS IoT core. I followed a tutorial provided to me by my cloud computing lecturer. The tutorial was by

Ezequiel Miranda [8]. For this I added three new fields to the employee schema. They are two numbers and a Boolean.

On the backend a “device” had to be set up. This function required a private certificate, key and a rootCA.pem file. It also needed the host name to connect to IoT core. After the device is set up a connect function is called where it subscribes to four topics. TeamSwipe_ESP32/request-employee-array, clock-in, clock-out and request-employee-earned.

The next function is the “onMessage” function. When a topic comes in the topic name is checked by four if statements. If the topic name is “TeamSwipe_ESP32/request-employee-array” the payload (which is the company email) is searched in the database and the employee array is sent to the ESP32.

If the topic name is “TeamSwipe_ESP32/clock-in” the Boolean variable “working” is set to true and a date is called set the “startTime” variable. Both of these variables are then updated on the database.

If the topic name is “TeamSwipe_ESP32/clock-out” a new date called “Stop” is created and “startTime” is subtracted from this. The result is multiplied by 1000 to get the time in seconds and is then multiplied by the employees hourly wage in seconds. This result is returned to the ESP32 and the database is updated setting the “working” variable to false and adding the calculated pay to the “totalPay” variable.

If the topic name is "TeamSwipe_ESP32/request-employee-earned" the value of “totalPay” is returned to the ESP32. The last function was the send data function. It takes in a topic name and a payload. The payload is turned into a string and then sent to IoT core.

I had a bit of trouble sending data from the backend to IoT core because I had set up the communication policy wrong. It was set up to only allow the ESP32 to send messages. It was soon changed to receive any messages sent to it but only from devices with the correct certificates and keys.

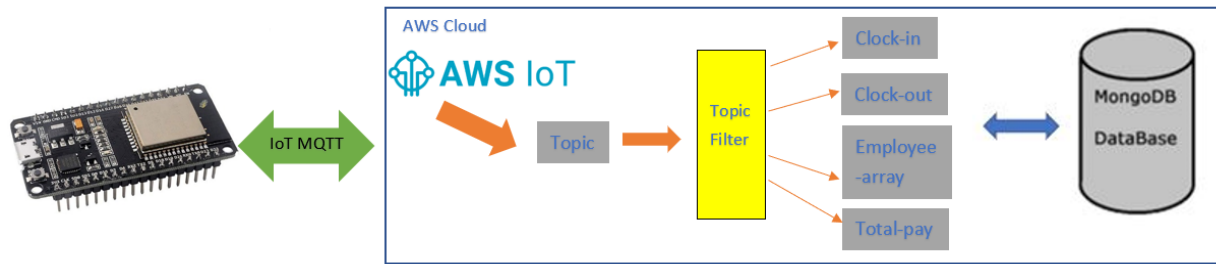


Figure 9.4-1 Backend AWS IoT and MQTT

9 ESP32 Device

The ESP32 device is a prototype of what the TeamSwipe clock-in clock-out machine would be. I originally came up with the idea for it shortly after starting my second year project. The device is intended to be mostly used by the employees of a company or business. A manager can only use it to write new cards or tags and erase them. In this section I will go through the design history of the ESP32 Device and how it evolved into the final version.

9.1 Original concept

The original project idea was that the ESP32 would host the webserver and the payment manager. Much like the final version a manager would be able to see a list of employees on a table and be able to edit them as they wished. In June and July 2021 I started some work on this by creating a two page asynchronous webserver hosted on the ESP32. It had a very basic login with no authentication. The html pages it had were stored in the ESP32 storage using SPIFFS. The work I did on this is on a branch on my github.

9.2 RFID reader

After waiting for a month for all the parts to arrive I began working on the ESP32 code in late October. I started with the RFID reader because I had not worked with one before. When I downloaded the library for it I found that the most recent version was incompatible with the ESP32. After reverting to an older version it worked fine. I based my code off the read personal data example provided with the library. This code read data from sector 4 of the RFID card and parsed it into char arrays. I also modified the write personal data example so I could write the employee name, role and id for testing.

Each sector of the RFID card has 4 blocks that are 16 bytes long. Both the name and the role were under 16 bytes in length but Google Firebase's ids were 20 characters long excluding the null character. The id had to be read into two 18 character char arrays and then copied into a 22 character array so they could be appended together. This led to a lot of stack overflow errors until I got the code right.

To test how I could compare the data on the card to the data on the database I created a char array with the word "Manager" in it. When a card with the role "Manager" was tapped it would compare them using memcmp. If they were the same the console would print "hello" otherwise it would print "unauthorized". I found that I had to reduce the sizes of my char arrays because reading data from the cards was causing a buffer overflow into the one containing "Manager".

After that I created an array of char arrays to store the ids. When a card was tapped the id on the card would be compared to the ones in the array. If the id was not found it would print "unauthorized" and if it matched it would print "hello". Now that I was able to compare the card's id to the array and the card's role to "Manager" I could create a menu for the manager.

9.3 Manager Menu



Figure 10.3-1 Manager menu

The menu had three options. They were “Write New Card”, “Erase a card” and “Quit”. The first option on this menu I implemented was the “Write New Card” option. This function had two versions. The first was a modified version of the “write personal data” example that came with the RFID card reader library. When I wrote data to the card through the serial monitor it appended a \n character onto every input. At first I thought it was a problem with my code but it was actually just an option on the serial terminal.

The next option I implemented was the “Erase a card” option. This was an easy one code because all I had to do was write space characters to the three memory blocks in sector 1 that had data in it.

The menu could be navigated using a joystick module connected to the ESP32. The joystick had a built in button that is used to select an option. The ADC (Analog to Digital Converter) pin that the joystick was connected to needed to be checked every few milliseconds. To achieve this I used a timer interrupt that set a Boolean flag every 25ms. I followed a short tutorial on how to add this [9]. An if statement in the menu loop would check if this flag was set to true and a function would check the value on the ADC pin and move the pointer on the display accordingly

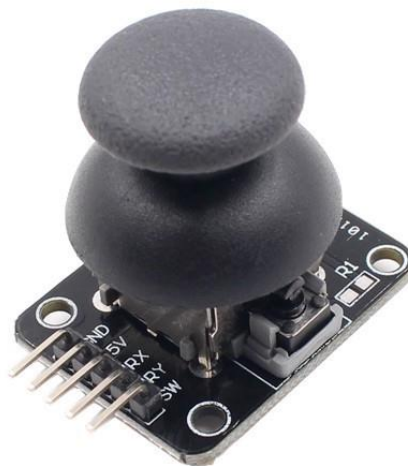


Figure 10.3-2 Joystick module

9.4 Implementing Firebase

As mentioned earlier, the frontend of my project used firebase as a temporary backend and database. To effectively demonstrate the ESP32 side of the project for the Christmas demonstration I needed to access the list of employees stored there. Luckily a library for that existed. I followed another tutorial just to test if I could pull down the data [10]. From this tutorial I was able to pull down all the data as a string but was unable to parse it out as I needed it. At first I tried using the Arduino JSON library but it was not compatible with the Firebase JSONs. After reading the documentation for the Firebase library [11] I found that it had JSON parsing functions specifically made for it. I soon began implementing this code into the main Arduino code.

I started by changing the code I used for comparing to the array of ids to instead compare to the ids taken from Firebase. When this was successful I changed the “Write New Card” option open a menu with a list of all the employee names on the database. Clicking on any of these names writes that employee’s data to the RFID card.

The screen can only show three names at a time so to get around this I used the second axis of the joystick (left and right) to scroll through the employees. The original idea was that they would be displayed in pages of three but I found that if the number of employees was not divisible by three there would be one or two empty spaces on the last page. This was a problem because it would write empty data to the card. To fix this I decided the left-right axis would only scroll by one and would not scroll any further than the last employee in the array.

9.5 ESP32 PCB

In early January 2022 I began working on a printed circuit board to attach the ESP32 and its components to. I originally intended to use Altium designer to design the PCB but I couldn’t access it over eiricom. I instead used a free online PCB software called EasyEDA.

I made footprints for all the components. While designing the board I made two errors on the footprints. The first was that I made wrong measurements for the joystick mounting holes. The second was due to a poor printing on my ESP32. I put a pin marked "CMD" in the same net as the ground pins. When the board arrive and I was testing it this error caused the ESP32 to reset. To solve this I cut the trace on the board.

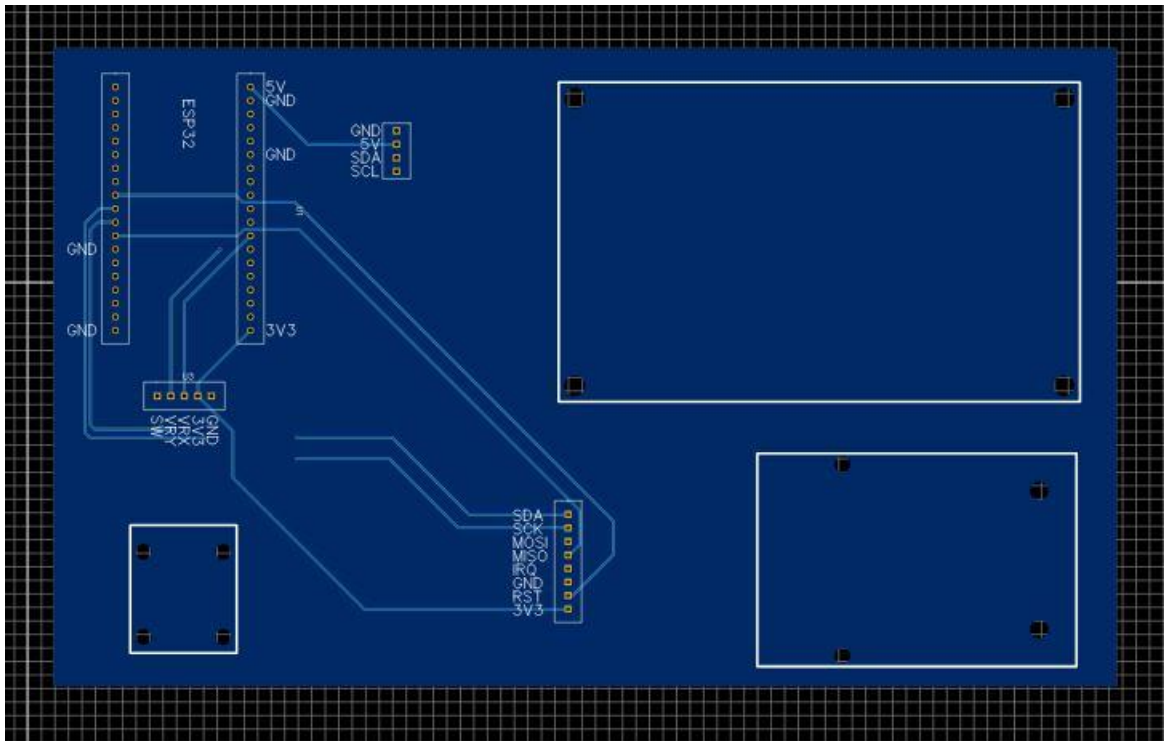


Figure 10.5-1 PCB in EasyEDA design software

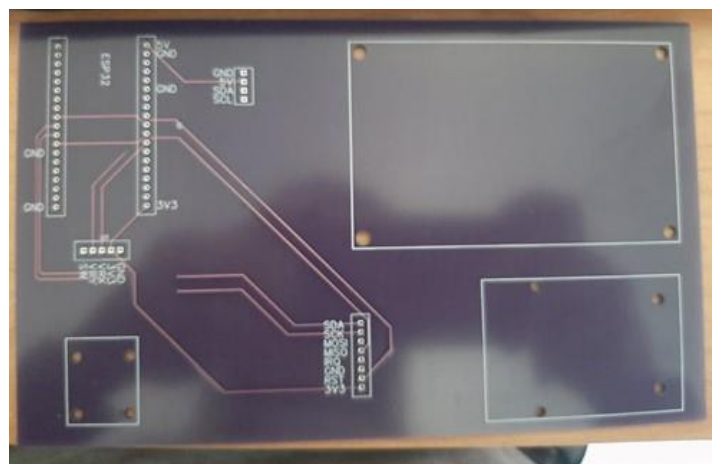


Figure 10.5-2 Physical PCB

9.6 Implementing AWS IoT

In early April I opened a personal account on AWS. I needed this because the student account we were provided by the college did not cover IoT core. To be able to use IoT core I needed to create a “Thing”. This would register allow my ESP32 to send and receive data from IoT core through MQTT. I followed a tutorial on how to set up a thing on IoT core that would be compatible with the ESP32 [12].

For the ESP32 to connect it needed three files. A security certificate, a private key and an Amazon root certificate. The contents of these files had to be added to a header file. This header file had to be put in the gitignore file to prevent its contents from being stolen. In the tutorial that I followed [12] there were three important functions. The first was “connectAWS”. This function took the certificates and keys from the header file and added them to a wifi client. It then began the MQTT connection with IoT core and once it was connected it would watch for messages. The second was the “incomingMessageHandler” function. This waited for a topic and payload to be arrive and would then print it as a string. Finally there was the “publishMessage” function. This function sent the running time of the device and a random number to IoT core. It was called in main alongside “mqtt_client.loop”. “mqtt_client.loop” called the “connectAWS” function if the connction was lost and made sure the “incomingMessageHandler” function was called if a message came in.

To make sure the array of employees I wanted the ESP32 to receive was correctly parsed I modified the “incomingMessageHandler” function. I first put in an if statement checking that the correct topic had arrived and then split the objects in that array. At first I tried converting the payload string as a JSON using a library but it would not convert into a JSON array properly. I instead used string methods to split the array into separate strings using the opening and closing brackets of the JSON objects. From the each string was parsed in a separate function called “dataSorter” where the name, role and ppsn were placed into three string arrays. Using the IoT core test console and a JSON array from my local mongodb database I tested the code.

Subscribe to a topic | **Publish to a topic**

Topic name
The topic name identifies the message. The message payload will be published to this topic with a Quality of Service (QoS) of 0.

Q Enter the topic name

Message payload

```
{
  "message": "Hello from AWS IoT console"
}
```

► Additional configuration

Publish

Figure 10.6-1 IoT core test console

The next step was to redesign the code used to get data from Firebase to instead get it from IoT core. I started by making a copy of the modified tutorial code and adding all the functions from the old version of the ESP32 code that I know would have no effect on MQTT. I then added the “readRFID” function to the main loop. From this I needed to get the data from the RFID card, stringify it and then compare those strings to the ones in the string arrays. I wrote a function that takes in the card data as a byte array, casts it to a string, shortens that string to 16 characters long and then trims off any spaces and returns it. This is done with the card’s name, role and ppsn. These strings are then compared against the string arrays in a while loop until a match is found. If not the lcd screen prints “Unauthorized Card”.

Once that worked I added back the “managerMenu” and got to work on the “writeRFID” function. This function was a lot easier to write this time around since the name, role and ppsn were easier to change into bytes than the Firebase JSONs. When I completed the “managerMenu” I had to begin work on the employee menu. I decided just to modify the “managerMenu” so depending on who logged in it would have different options. I added a Boolean. If it was false the manager options would show up. If it were true the employee options would show up.

Like the manager menu, the employee menu would have three options. They are “Show Pay Earned”, “Show Total Pay” and “Quit”. When the employee clocks in the “Show Pay Earned” option only prints 0.0 because they just started work. When they clock-out the backend returns the pay calculated from the time they clocked in. It is then printed on the LCD when they select

this option. The “Show Total Pay” option gets the pay they have earned over their total time working. Every time they clock-out their pay is added to the total. This is called from the backend every time an employee taps the card so they can see their pay update. When any user exits the menu the “publishMessage” function is called and asks the backend for a new employee array.

The ESP32 sends four topics to IoT core. They are "TeamSwipe_ESP32/request-employee-array", "TeamSwipe_ESP32/clock-in", "TeamSwipe_ESP32/clock-out" and "TeamSwipe_ESP32/request-employee-earned". The “publishMessage” function was modified to take in a topic and employee name. there is an if statement checking if the topic is "TeamSwipe_ESP32/request-employee-array" or not. The topic payload gets serialized into a JSON to be sent to the backend.

The ESP32 is subscribed to three topics. They are "TeamSwipe_ESP32/return-data", "TeamSwipe_ESP32/return-employee-pay" and "TeamSwipe_ESP32/return-employee-earned". “return-data” returns the array of employees to be parsed. “return-employee-pay” returns the calculated pay after an employee clocks out and “return-employee-earned” returns the total pay an employee has earned.

10 Amazon Web Services

All three parts of my project are hosted on or use Amazon web services. The frontend and backend are hosted on it and the ESP32 uses its IoT core service. This section will focus on how I set it up.

After getting a student account as part of my cloud computing class I opened an EC2 instance. I set up port 3001 to be open to any IPV4 address so anyone could access the TeamSwipe website. Once the instance had booted up I entered a list of commands to install NPM, NodeJS and mongodb on it. I then created a full stack development directory (fsd) with a “node” and “next” directory. I downloaded WinScp and followed a tutorial on the WinScp website [13]so I could send my frontend and backend to the instance. I found out that I needed to change the fetch calls on my frontend to call through API routes for it to work.

I had to open a personal account on AWS because IoT core is not available on the student version. I had to follow all the same steps again to set up the instance. For the ESP32 I have to create a “thing” in IoT core. With this I had to create a policy and download certificates and keys. The original policy I created worked fine for the ESP32 but was preventing the backend from receiving or sending data. To solve this I opened the policy to any device that had the correct certification.

11 Ethics

The considerations I took when I started this project are the following:

Privacy

One of the main goals of this project was to create a clock-in clock-out system that did not use biometric data as a way to identify employees. They are instead identified by data that the employer has access to like their PPSN and their role in the company. This prevents less than reputable employers from misusing their employee’s data. As the world becomes increasingly digital and big data is more valuable than ever it is important for people to have control over their data.

Another aspect of privacy shown in my project is a login system using encrypted passwords. This protects the employer’s data and by extension the names and data of all the employees. The password also protects from accidental deletion of the account. The authentication lasts 15 minutes. After that time the user is logged out and the login token is invalidated.

Environmental concerns

To reduce this project’s effect on the environment the frontend and backend are hosted in the cloud. This reduces the carbon footprint that would be caused by running the project on a local server. If I had the project running all day on a local machine it would draw a lot of power from the electricity grid. Comparatively it would only take up a tiny bit of space on the AWS servers. Its power draw would be almost insignificant and would be much more efficient.

12 Conclusion

I am very happy with the outcome of this project. The website and clock-in clock-out machine have both turned out how I imagined them. I am glad I was able to get the website working fully on the AWS cloud. The moving from firebase to NodeJS for the frontend and ESP32 went smoother than expected in both cases. Though it was difficult I kept to my project plan and updated it where I needed. The authentication and login system works how I planned and prevents any crossover of data from other users.

I believe with further development this system could become a viable commercial product someday. I feel it has a very user friendly interface and would be easy for any business to pick up and use. All the code for this project can be found on my Github at this link:

https://github.com/LiamNilan599/LN_Final_Year_Project

13 Appendix

| | |
|--|--|
| IoT, Internet of things..... | Pages 8, 10, 13, 21, 22, 27, 28, 29, 30 |
| RFID, Radio-frequency identification..... | Pages 6, 8, 9, 13, 23, 24, 25, 28 |
| PCB, Printed circuit board | Page 26 |
| MQTT, MQ Telemetry Transport..... | Pages 10, 13, 21, 27 |
| EC2, Elastic compute cloud..... | Pages 8, 10, 13, 30 |
| MERN, Mongodb, Express, React, NodeJS..... | Page 12 |
| AWS, Amazon Web Services..... | Pages 8, 10, 13, 21, 27, 28, 29, 30, 31, 32 |
| JSON, Javascript object notation..... | Pages 13, 14, 16, 18, 19, 20, 21, 26, 28, 29, 30 |
| JWT,JSON Web Token..... | Pages 13, 16, 20 |
| SPIFFS, Serial Peripheral Interface Flash File System..... | Page 23 |
| CORS, Cross origin resource sharing..... | Pages 14, 19 |
| REST, Representational state transfer..... | Page 21 |
| CSS, Cascading style sheets..... | Pages 11, 14 |
| API, Application Programming Interface..... | Pages 14, 17, 29 |

14 References

- [1] M. Schwarzmüller, "react-complete-guide-code," [Online]. Available: <https://github.com/academind/react-complete-guide-code>. [Accessed 22 09 2021].
- [2] C Blakely, "react-creating-a-table," [Online]. Available: <https://github.com/chrisblakely01/react-creating-a-table>. [Accessed 01 10 2021].
- [3] "bestofreactjs.com," [Online]. Available: <https://bestofreactjs.com/repo/express-labs-pure-react-carousel-react-image-gallery>. [Accessed 20 10 2021].
- [4] O. Benseddik, "tinloof.com," 16 11 2020. [Online]. Available: <https://tinloof.com/blog/how-to-build-an-auto-play-slideshow-with-react>. [Accessed 31 12 2021].
- [5] M. Schwarzmüller, "nodejs-basics-tutorial," [Online]. Available: <https://github.com/mschwarzmueller/nodejs-basics-tutorial>. [Accessed 28 01 2022].
- [6] K. Cook, "JWT-Authentication," 19 09 2019. [Online]. Available: <https://github.com/WebDevSimplified/JWT-Authentication>. [Accessed 11 02 2022].
- [7] K. Cook, "Nodejs-User-Authentication," 09 06 2019. [Online]. Available: <https://github.com/WebDevSimplified/Nodejs-User-Authentication>. [Accessed 19 02 2022].
- [8] E. Miranda, "medium.com," 13 08 2021. [Online]. Available: <https://medium.com/dev-jam/getting-started-with-aws-iot-core-and-mqtt-protocol-with-a-node-js-example-ed16bd542704>. [Accessed 06 04 2022].
- [9] "techtutorialsx.com," [Online]. Available: <https://techtutorialsx.com/2017/10/07/esp32-arduino-timer-interrupts/>. [Accessed 07 12 2021].

- [10] R. Santos, "randomnerdtutorials.com," [Online]. Available: <https://randomnerdtutorials.com/esp32-firebase-realtime-database/#esp32-read-data-firebase>. [Accessed 13 12 21].
- [11] K. Suwatchai, "Firebase-ESP32," [Online]. Available: <https://github.com/mobizt/Firebase-ESP32>. [Accessed 13 12 2021].
- [12] Y. Sanghvi, "iotespresso.com/," 20 10 2021. [Online]. Available: <https://iotespresso.com/how-to-connect-esp32-to-aws-iot-core/#:~:text=All%20we%20need%20to%20do,received%20on%20the%20Serial%20Monitor>. [Accessed 26 03 2022].
- [13] "winscp.net," [Online]. Available: https://winscp.net/eng/docs/guide_amazon_ec2#direct_connection. [Accessed 23 03 2022].