**Portfolio Points:**

**1a) Architecture of the entire system**

Imagine you are a student with the stress of approaching exams. You want to head into the university to study but you aren’t sure you will be able to find a free chair. The last few times you have been in you have wasted half an hour going from library to library before giving up and heading home. Enter FindADesk! An application that takes away the stress of finding free chairs in university study spaces.

The architecture of the system centers around ‘chair’ objects, with 3 different sub-systems all interacting with the ‘chair’ objects in different ways. These 3 sub-systems communicate with each other, sending information regarding the number and location of chairs, the state that any chairs are currently in and if there is a booking made for the chair. This allows students to see exactly where any chairs are and whether they are available. The communication is carried out by sending JSON objects through MQTT, a lightweight messaging protocol, with the messages automated by the programs in the applications.

**1c) Requirements of key sub-systems (in the form of selected user stories)**

As an administrator, I want to add new study spaces to my institution’s application.

As an administrator, I want to be able to see a visual representation of my institution’s libraries to make the above easier.

As an administrator, I want to be able to receive information from chairs in our study spaces to see information about usage and status and act accordingly.

As an administrator, I want to be able to set up the stack with the appropriate building, room and chair ids on the device directly without re downloading the software each time.

As a student, I want to head into university to study without the stress of not finding a study space.

As a student, I want to be able to see which library spaces are quieter.

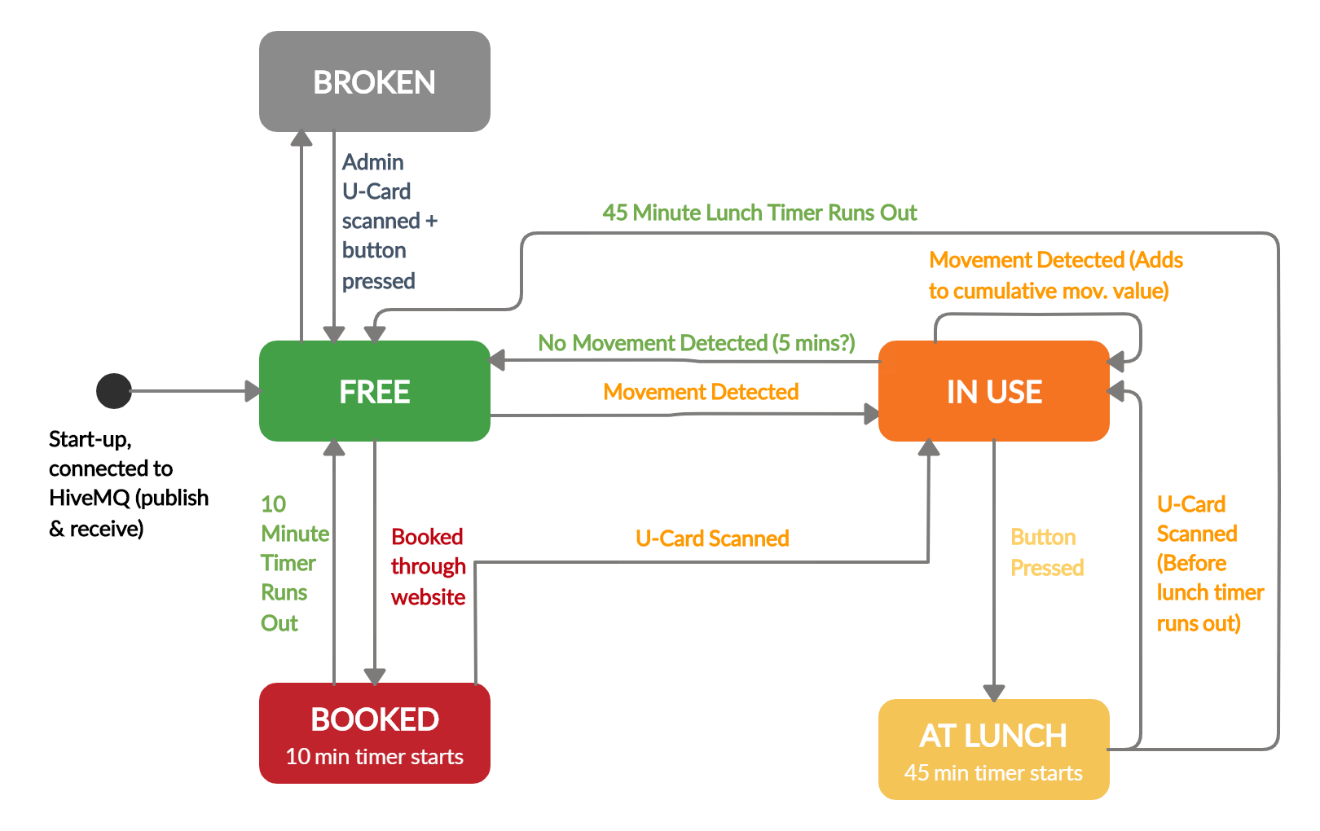
**1b) Object-Oriented design of key sub-systems (e.g. Desktop Application, Web Application etc.)**

The chair object is initially generated by the desktop application – this is the subsystem used by an institution administrator to implement the visual representation of their university’s study spaces. This is achieved through the creation of building, room and desk objects which can be added or removed to reflect the volume and rough visual layout of desks on a room by room basis. Once a room and its desks have been added, the associated data of the room (its parent building id and name, its own name and id as well as the number of tables it contains) is published to the web application via a JSON string. The processing application also receives MQTT messages from the chair if it sets a status that requires action by the administrator (this is primarily if it is broken). This report is generated and is visible by the administrator so that they can take appropriate action.

The Processing application is built around a cascade of objects representing buildings, rooms, desks and chairs. These each contain an Array of objects such that buildings contain rooms which contain desks which have chairs.

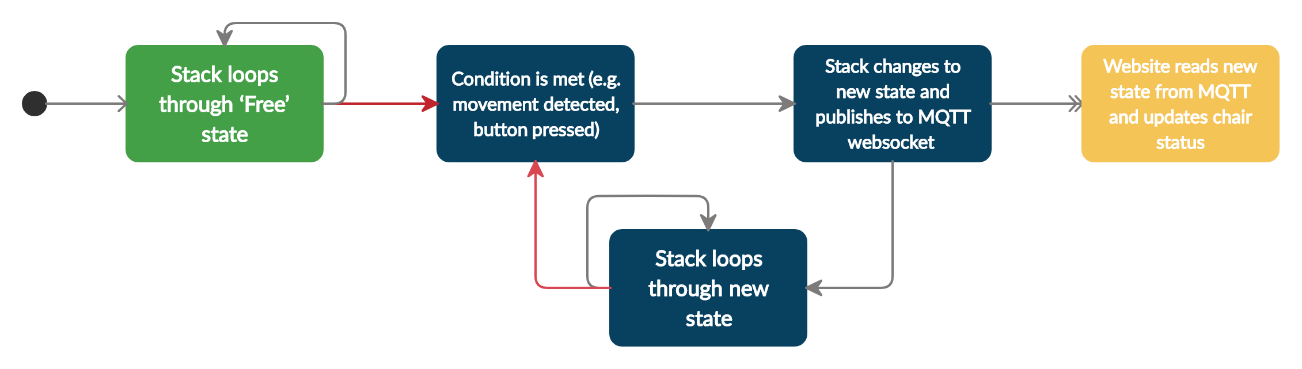
The building and room classes have both name and id attributes, while each subsequent sub-object also stores the id of its parents such that a desk, for example, will also have attributes denoting the ids of the room and building to which it belongs. The chair is somewhat different; its id (instead of being an integer) is a string composed of the cumulative ids of its parents combined with its own id. This id is unique per room. This allows the processing app to easily send chair information to the web app and subsequently receive chair information from the stack. We did not see much value in using inheritance in the application, primarily because the objects function as data structures with little extra functionality gained from using it. In effect, the desktop application uses Object Oriented design to closely mirror the real world layout of a library / study space, and to also act as data storage containers for easy communication between applications. Our desktop application also contains classes representing the main dashboard and one which updates the view based on the user’s interaction. The view class updates the dashboard and also handles the user interactions with the application, including adding rooms and desks. The main dashboard class sets up the dashboard, draws the tables and handles the parsing and publishing of MQTT messages (this will change).

The web application, built using HTML, CSS and node.js, is designed for students who wish to see the current states of the rooms (e.g. which libraries are the busiest or the number of free chairs in each room). It is run using an express server, so it can continuously be checking for messages from the other devices. It subscribes to the desktop application through MQTT so that the website can be updated with the correct number of rooms in each building and the correct number of chairs within each room. This information is saved in a nosql database (MongoDB) so that information can be reloaded (rather than the desktop application continuously sending messages regarding the current room states). The web application also subscribes to the M5Stack so that it can update the web page with the current state of the different chairs (occupied, free, booked or at lunch). Using this information, students are able to see the current occupancy of each room and to make a choice about where they would like to study. Further to this, a student is able to select a chair and make a booking. This will subsequently publish a message to the M5Stack for that particular chair, reserving it for that user. When the chair becomes free again the M5Stack will send a message back to the web application, informing it that it is now free.



The M5 Stack subsystem is responsible for detecting the current state of the chair and displaying it, which is determined by following the above state machine diagram. The stack initialises at the black circle and moves to free, then stays in free until one of the stated conditions is met. It can only then move between the states which are connected by the arrows shown above (e.g. it cannot move from ‘booked’ to ‘at lunch’ directly). The stacks onboard accelerometer is used to measure vibration from the chair, and if it is above a certain threshold the state then moves to ‘in use’. While in use, the leftmost button can be used to start the chair user’s lunch break, initialising a 45-minute timer in which the chair is reserved and cannot leave this state. To exit this state, either the user must scan their U-Card, returning the chair to the ‘in use’ state, or the timer can run out, returning to the ‘free’ state (because evidently the user hasn’t made it back in time from lunch, so their seat is now available).

From ‘free’, the chair can also change to the ‘booked’ state, which means that a user has reserved that seat through the website application. This initialises a 10-minute timer, allowing the user to get to the seat and scan their U-Card, changing the state of the chair to ‘in use’. If they fail to scan their U-Card within 10 minutes, the chair state reverts to ‘free’. A fifth administrator state was added to the state machine, allowing the chair to be declared ‘broken’ if an administrator U-Card is scanned and the leftmost button is pressed. Changing to this state then notifies the processing application that a chair is broken so it can be repaired – the only way out of this state is for an admin to once again scan their U-Card and press a button, thereby preventing the chair from being used when it is broken.



When the stack changes between two states, it publishes its new state to the MQTT WebSocket (which the web client receives and updates on the website and within the database). This can be seen in the diagram above; where a condition is met, the state is changed and published to MQTT, and then the stack loops through its new state until another condition is met and the state changes again.

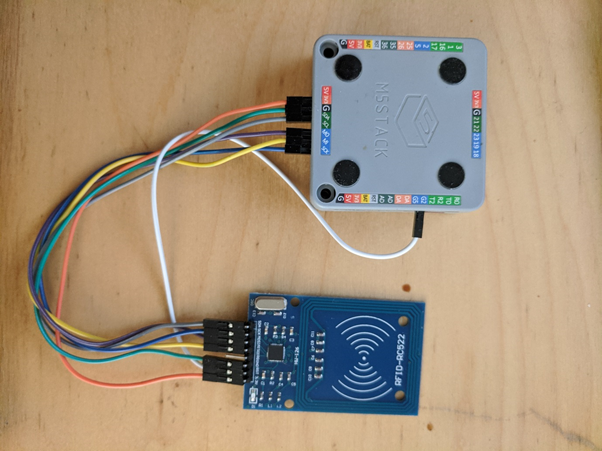
The stack can also receive information from the website and send information to the processing application. Both of these are compartmentalised and it is only able to receive the ‘booked’ command from the website when it is in ‘free’ mode (as dictated by the state machine), and it will only send out information to the processing application when it enters the ‘broken’ state (this is also achieved through the use of MQTT).

In terms of Object-Oriented design, the code for the stack itself is written in C++ in a functional way, but in terms of the entire system the stack in itself is very much a Stack object, which has an id and a state which can be passed around and stored by the processing and web applications.

**Implementing UCard (RFID) scanning functionality:**

In order to confirm the correct individual is using the chair, the student will need to scan their UCard upon arrival. This must be completed within 10 minutes of booking or the chair will be re-released into the pool of available chairs. The UCard acts as an RFID (Radio Frequency Identification Device) and its unique identifier is compared with the ID sent from the web app. If the ID sent from the web app matches the ID scanned, then the student has been successfully authenticated and can begin using the desk.

Reading the ID within the UCard would be achieved using an external RFID reader. The module chosen to achieve this was the MFRC522. The MFRC522 is a highly integrated reader/writer IC for contactless communication at 13.56 MHz. This board is manufactured by NXP but can be purchased from the M5Stack store in an enclosure that connects to the stack’s 4-pin ‘GROVE’ connector. As the delivery times from the online store were large, the standalone MFRC522 board was purchased from eBay, this unit did not include a GROVE interface however. This module was therefore connected to the stack’s I/O bus via jump cables:



**1c) Requirements of key sub-systems (in the form of selected user stories)**

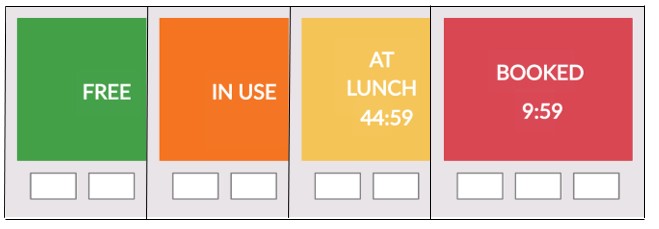
Our intended purpose for the processing app was to have a maintenance/admin application for the owners of the FindADesk system. We wanted the users to receive quick stats about the usage of chairs across the campus, add new desks to rooms, and add new rooms. The idea is that the processing app can be easily used by different institutions to set up desk spaces to be used by students in the web app and to interact with chairs using the stack and stick. The desktop app has several requirements. The first is to provide a graphical representation of the university/institution’s library / study spaces, and that these can be modified by the administrator. The second is to publish this information to the web app so that any modifications made by the administrator are reflected in what end users are able to see. The third is to receive information from the Stack with information about the states of chairs so that reports can be generated / acted upon by the administrators.

Processing

In terms of UI, we first experimented with having a search bar to find buildings. This proved to be both difficult to implement and difficult to use for the user. Based on user feedback, we decided instead to use dropdown lists for users to navigate through buildings and rooms. We tried to do this by adapting the code from the processing code walkthrough, however quickly found out that it only partially fulfilled what we wanted to get done. By using control p5 and by creating functions with the same names as the cp5 controllers we were creating (similar to how you write code in css/js) we were able to extract which items were being pressed in the drop down lists and from that we could start to navigate between classrooms and display them individually (*image 2)*. The fundamental design of our UI revolves around showing the user a graphical representation of the desks and chairs on a room by room basis. Available chairs are shown in green while red chairs signal any other status. Clicking on a drop down menu and selecting a building and room allows users to add a new desk to the room one at a time. The user can also choose to add a new room to the selected building. Users understood the UI intuitively and needed little help navigating it. One user however did comment that when creating a new room, the program should automatically navigate to the new room instead of it having to be selected post-creation. This happened after the user accidentally added more desks to the current room instead of the new one.

In summary, the web application gives students peace of mind when wanting to head into university and use study spaces whilst also helping universities manage their spaces better, ensuring maximum usage.

**1d) The evolution of UI wireframes for key sub-systems**



The initial user interface designs for the M5 Stack were basic, simply indicating the current state with text and a unique colour so that the user could see the state of the chair from a distance. The ‘lunch’ and ‘booked’ states also had timers counting down to indicate the time left in that state – both reverted back to free if the timers expired without any user input. As is included in the state machine, the lunch and booked states are designed for the user to be able to scan their U-Card to access that chair but as a temporary measure for the initial design, the user simply had to press the left-most button to access it.

Starting with a basic paper prototype to imitate the web application, we tested this on volunteers to try and get some early feedback.

Some of the key points include:

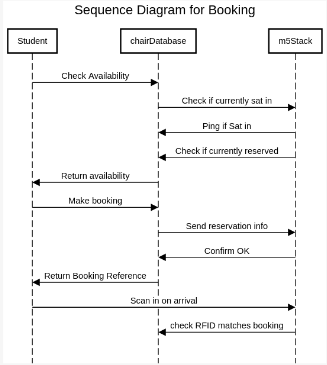
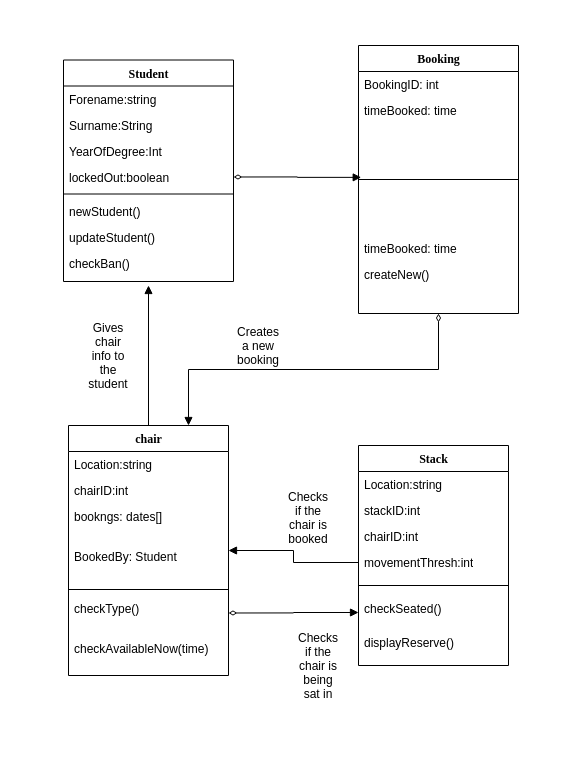
* What is the purpose of the application. What are users supposed to get out of it?
* Oftentimes users had to be guided towards the next page - the flow wasn’t natural to them.
* Colour of chairs not explained (e.g. green = free, red = occupied). A colour key would be beneficial.

Based on this feedback, changes were made to the wireframes before beginning implementation of the website.

The UI layout was reassessed with each subsequent iteration/sprint and the design was changed based on the user feedback. Some of the comments recorded are:

* Wasn’t clear how many free chairs in a room. Did the number refer to total chairs or total free chairs?
* Not clear about booking time. For example, how long do I have to get there?
* How to identify what chair I have booked? Is there an ID/code?

e)

The MFRC522 supports two communication protocols to communicate with the stack, I2C (Inter-Integrated Circuit) and SPO (Serial Peripheral Interface). I2C is sometimes preferred over SPI because it allows for multiple ‘slave’ devices to communicate with multiple ‘masters’ (the stack in this case) using only two lines. One to transmit the data (SDA) and one to carry a synchronising clock signal (SCL). As there are only two lines, I2C requires each slave to be addressed and this is reflected in the MFRC522 object within the source code where the constructor requires an address to be provided when I2C mode is utilised:

*MFRC522(byte chipAddress);*

SPI offers faster data rates compared to I2C however needs at least 4 lines to be utilised, a Clock (SCK), a line to transfer data from the CPU to the peripheral MOSI, a line to transfer data from the peripheral back to the CPU (MISO) and a chip select line (CS) to select the peripheral in question (hence many more lines may be needed for multiple devices).

Unfortunately, we were unable to get the MFRC522 to interface with the stack, this using either of these connection protocols. This may have been due to the fact that eBay bought MFRC522’s can often be temperamental.

MQTT, a lightweight messaging protocol, was used for communication between the different applications. The MQTT transfers information as text and therefore some sort of text structure is required to make the communication simpler and more efficient. It was decided that a JSON type text structure would be able to send information in the most efficient way, allowing for easier parsing when receiving a message.

Our system required that there be multiple communications going in different directions between the different devices. Because of the complicated communication design of our system, we decided to use multiple different subscriptions so that it was clear which messages were being published to which application. These are:

1. *FindADesk\_ProcessingToWeb*

The desktop application needs to publish information to the web application regarding the current state of the university study spaces, including the number of buildings, the building names, number of rooms in each building, room names and the number of tables and chairs in each room.

1. *FindADesk\_WebToProcessing*

Likewise the desktop application is able to make a request to the server to get all the information regarding the buildings, rooms and chairs that the web application has saved in the database.

1. *FindADesk\_StackToWeb*

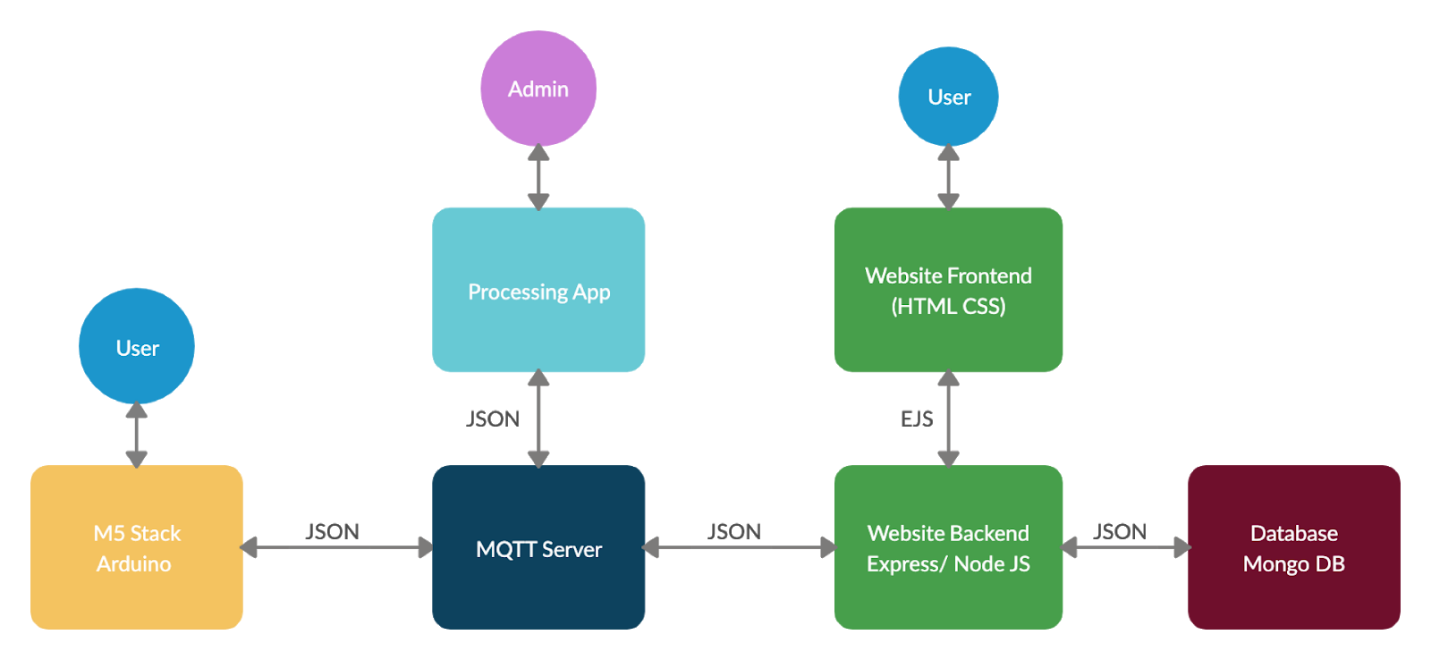
The M5Stack publishes to the web application, sending information regarding the current state of the particular chair it is on (whether this be “free”, “lunch”, “occupied” or “booked”).

1. *FindADesk\_WebToStack*

The web application also needs to publish information to the M5Stack, informing it if a student has made a booking.

1. *FindADesk\_StackToProcessing*

Finally, the M5Stack publishes information to the desktop application regarding any faulty chairs that the admin team will want to respond to.



f) To achieve data persistence, we originally tried using the browser’s local storage to store the data. However this would rely on the other sub-systems continuously sending out messages to ensure the web pages were concurrent with the current state of the rooms and chairs. Furthermore, the user would have to be on the website whilst a message was received from the desktop application or the stack which would result in missed data if the page crashed or was closed. Also if a user was to delete their local storage then they would again lose all the information and could potentially try to make a booking on a chair that was already occupied by someone else.

Following this we considered using an SQL database due to some of the entities having relational features (each building contains many rooms and each room contains many chairs). However, as we were investigating this avenue, we realised that we could use MongoDB which would be much simpler as it would allow us to reuse the same JSON structure that we were already using to communicate, removing the need to design a relational database. The JSON structures that were being used for the different communication paths can be saved directly as a MongoDB Object.

g) In order to build on our original wireframes and get a minimal viable product working, we originally built a simple static website with HTML CSS and Javascript (as mentioned before using the browser local storage for data persistence). As our application developed over subsequent sprints we realised that we would need a server and back-end to process the communication protocols and data-persistence mechanisms.

As we wanted to build a fast scalable application which wouldn’t need to be CPU intensive, we decided to use Node JS as it also provides a lot of features and doesn’t involve using another language for the backend. We also used the framework Express as it provides a lot of tools to quickly and easily build a server-side application with minimal hassle.

As our website would require dynamic html population with information received from the M5 stack and processing, it was decided that a templating language would be needed. We chose EJS as it would allow for easily creating html elements with plain javascript.

MQTT LIBRARY? MONGOOSE? NODEMON? BODY PARSER?

Social and ethical implications

In terms of the ethics of our project, we made sure to follow the ethical guidelines set out by the university. Participants in our study filled out consent forms are were both briefed and debriefed as to the purpose of each of the individual user studies that we carried out. In terms of the data that we are storing, we are not storing any personal data whatsoever, with all the data that we are storing being related to the layout and design of classrooms. We are not aware of any way in which our user testing could cause harm, and we made sure to give participants the ability to withdraw their responses.

Socially there is potentially some implications that can be considered. This product aims to improve student’s ability to find study spaces, but it also has the potential to hinder students who are less willing to interact with or those who are not aware of the product. In attempt to avoid this situation, desks can only be booked for a limited amount of time (20 mins), and therefore we do not believe that this will pose a problem.

Project Evaluation

a) The project overall turned out to be a success? Talk about how the processing part felt like ticking a box and could have actually been done through the website to simplify things with an admin login? Not sure if we have really done OOP for most the project. We accomplished most of what we set out to do.

b) Implementing above features, better UI. More robust system testing. More User testing. Adapt web app for mobile use. Perhaps with some features not as developed as they could be: different user accounts, RFID scanner etc. A lack of testing?. With a databse is there need for MQTT?

c-d) On the whole the group worked together very well. However there were certainly some difficulties we encountered along the way. Originally when we started the project we had an organised structure with weekly meetings, demoing the applications, getting face to face feedback etc. However when we started working remotely it took some time to get used to the changes. For a few weeks our progress was definitely reduced as we tried to develop a system where we could continue working collaboratively on the project. Previously in the web application we had been doing paired programming, with one person writing the code whilst the other watching and giving instructions and guidance. Remotely this proved to be much more difficult, even with screen sharing. In the end it was decided to split the workload differently, with different people working on different sections, but ensuring regular check-ins and commits to the github repository.