No part of the candidate's evidence in this exemplar material may be presented in an external assessment for the purpose of gaining an NZQA qualification or award.

Mana Tohu Mātauranga o Aotearoa
New Zealand Qualifications Authority

Scholarship Technology 2024

OUTSTANDING SCHOLARSHIP EXEMPLAR

Cricket Nets Booking Platform

Part 1 - The Problem **2**
- **Context** **2**
- **The problem** **2**
- **The current access control** **4**
- **The current booking system** **6**
- **Stakeholders** **7**
- **Possible solutions** **7**
- **Initial meetings** **8**
- **Initial pathway** **9**
  - User Interface 10
  - Accessibility 11
- Information Architecture 11
- **Social/physical environment** **12**

Part 2 - Initial Thoughts and Ideas **13**
- Version Control 13
- Technical Decisions 14
- Mockup designs 17

Part 3 - Getting Started **20**
- **Software** **20**
- Hardware 31

Part 4 - Improving the MVP **35**

Part 5 - Bringing Everything Together **45**

Part 6 - Conclusion **47**
- Client testing 47
- Future developments 48
- What ifs 51
- Notes for the Assessor 52

Throughout 2024, I have been working on a booking and access system for [redacted]'s cricket nets.

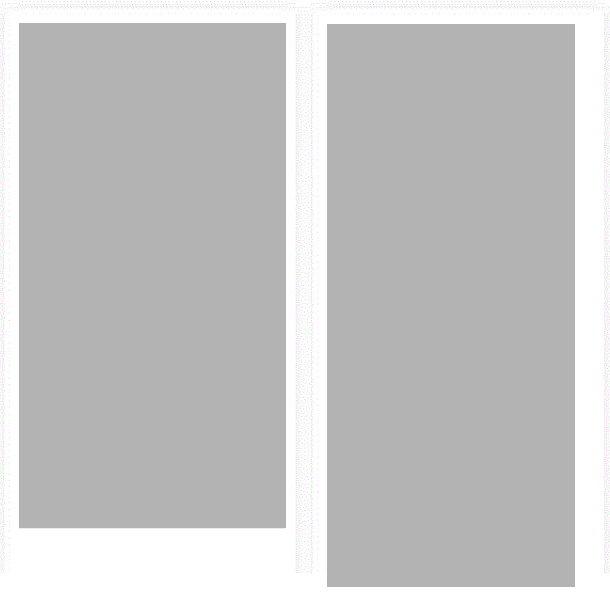
Previously, [redacted]'s cricket nets were only accessible by knowing the 4 digit code to a lockbox that contained a key. This meant that many issues could arise. All it took was for one person to forget to return the key to the lockbox for an entire cricket team to be unable to practise.

The [redacted] Cricket committee investigated possible options, and discovered that solutions to this exact problem did exist, and allowed for bookings to be made by members of the public, but these solutions were very expensive, with recurring subscription costs. They asked me to investigate whether I could create an 'in-house' solution for [redacted].

Part 1 - The Problem

Context

In 2016, [redacted] opened its newly constructed indoor cricket nets. The original purpose for this building was to allow the [redacted] cricket teams to have a high quality area to prepare for games in. The indoor nature of the cricket nets meant that they could be used regardless of whether or not the weather was good. As with any building, over time the building requires maintenance. At a school, any money spent on maintaining a building with the only purpose of sport is money that could have been spent on academics. While [redacted] does invest in sport, it can be difficult to justify spending money on maintenance, when the building is still usable. For this reason, the [redacted] Cricket Committee was investigating whether it could be possible to allow non-[redacted] affiliated people rent out access to the nets, as an income stream for [redacted] Cricket. They saw that [redacted] had a similar scheme with allowing access to their cricket nets, so investigated the issue further. They asked me if I would be able to create a solution for them.



The problem

The problem given to me by the cricket committee was to find out whether an income stream could be generated by renting out the indoor nets, and whether a more convenient way of entering the cricket nets could

be found. While I did investigate off-the-shelf solutions, I found that these were unsuitable as a solution.

This is the pricing scheme for [redacted], a popular all-in-one solution for this type of problem. While the Entry plan may look like it would solve this problem, an added problem is that it doesn't support access control. This would mean that people would be on the honour system to pay for the hours that they use, and would open the facility up to potential vandalism. Therefore the Standard plan would need to be used. $110.40 (including GST)/month does add up, plus the additional "pricing on request" cost of access control.

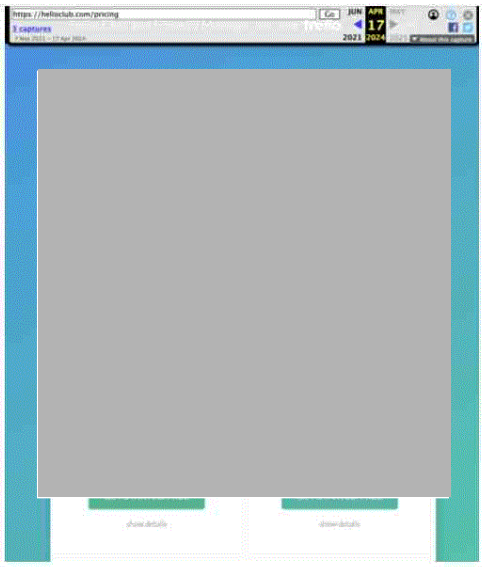
While $110.40/month would be a lot for a consumer to pay for a subscription service, this is actually a reasonable price for large sport facilities.

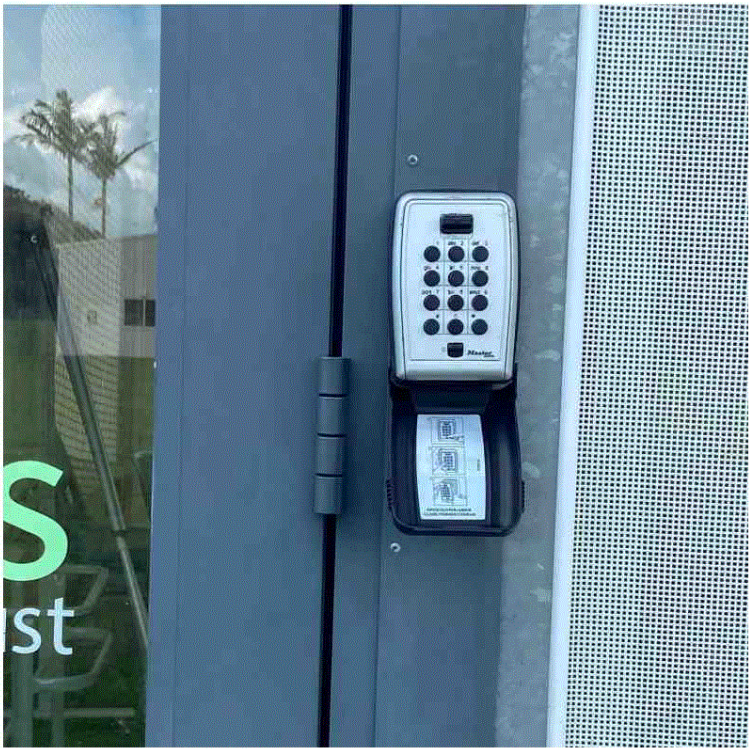
This subscription cost comes at a trade-off.

Pros	Cons
- Stop paying at any time - Quickly switch to another service - A helpline to call if it stops working - Secure payment processing	- Proprietary hardware - Subscription based pricing, which adds up in the long run - The price of the subscription can change at any time

The main problem that the cricket committee sees with this subscription cost is that they are unsure of whether the number of bookings would be able to cover the cost of the monthly fee. While they appreciate that the subscription fee gave them a risk-free way to try out the bookings for a short period of time, they thought it was worth investigating other solutions.

For context on how the price of the subscription can change, I can see through archive.org that the standard plan has increased in cost by 20% compared to what it was 6 months ago.





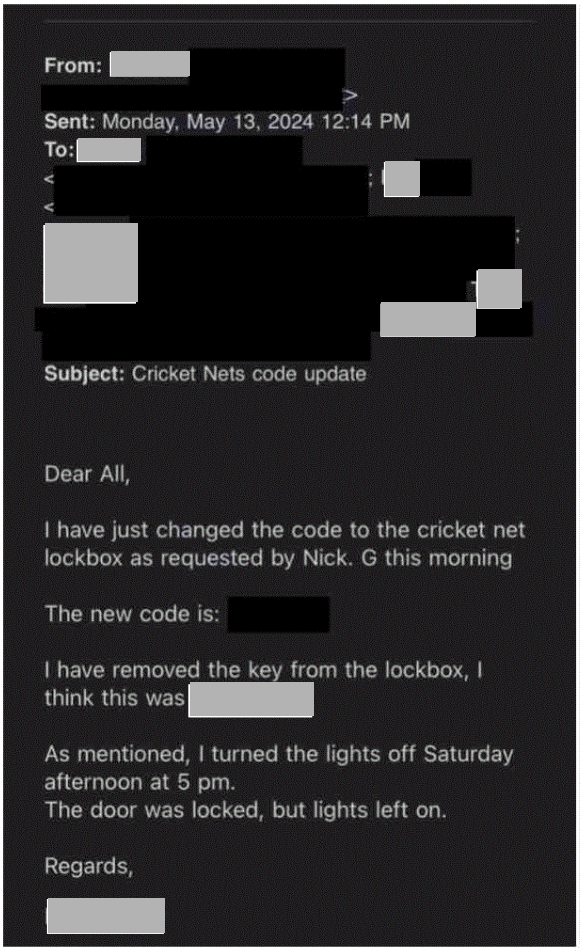
The current access control

The current solution for access control has been in place since the construction of the nets. There are 2 doors to the nets. The main door has a lockbox with a 4-digit code for it that contains the key. This was able to facilitate access to the nets for staff members, but was suboptimal.

If a staff member was in the PE office, and a student asked for the teacher to unlock the nets for them, there were 2 options:

1. Walk over to the cricket nets to put the code in
    - Which requires stopping what you are doing, which is inconvenient.
2. Tell the student the code
    - Which means that not only does that student now have permanent access to the nets if they remember the code, but also this student would likely share the code with their friends.

Because the first option is much more convenient for the teacher to do, it was usually what happened. While it may seem harmless, since the teachers would only give the code to a student they trusted, this code could be spread to anyone else without any traces. Without any traces, any vandalism that occurred inside the cricket nets couldn't be tied back to anyone. In an ideal world, teachers would keep this in mind, and therefore only choose option 1. But we don't live in an ideal world, so this problem needs a different solution if the cricket committee wants any damage to be traceable.



From: [redacted]
>
Sent: Monday, May 13, 2024 12:14 PM
To: [redacted]
[redacted]
Subject: Cricket Nets code update

Dear All,

I have just changed the code to the cricket net lockbox as requested by Nick. G this morning

The new code is: [redacted]

I have removed the key from the lockbox, I think this was [redacted]

As mentioned, I turned the lights off Saturday afternoon at 5 pm.
The door was locked, but lights left on.

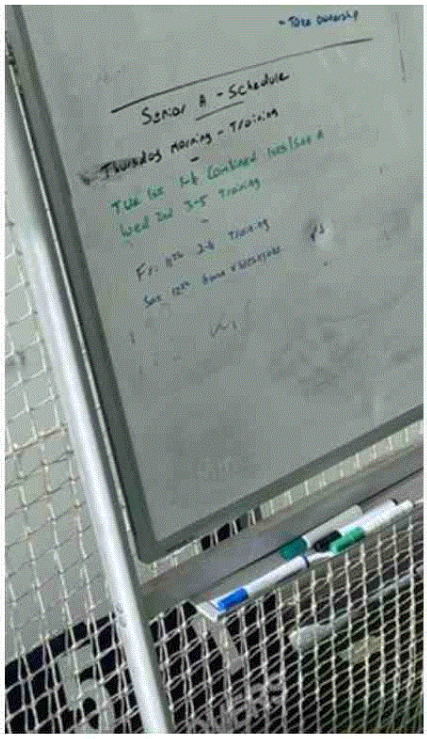
Regards,

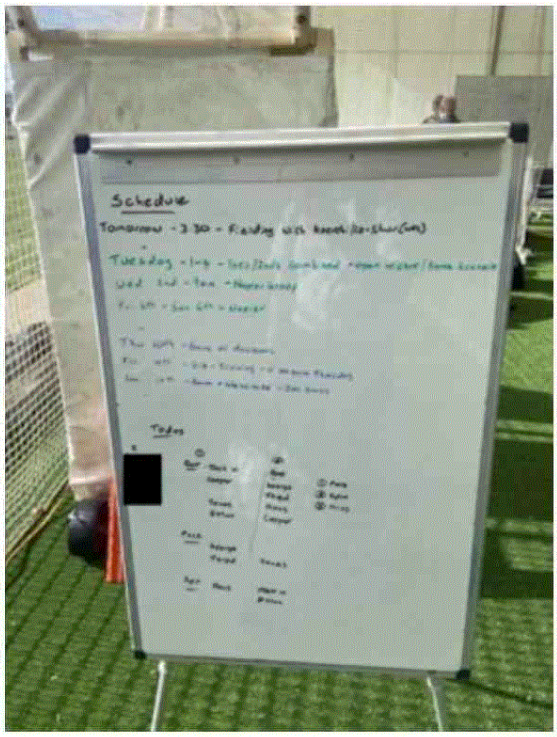
[redacted]

In an attempt to mitigate the problem of everyone knowing the password, the cricket committee began to rotate the code every few weeks, but this just became inconvenient for the staff members who hadn't realised that the code had changed and weren't able to get into the building. It was also inconvenient for the person who had to change the code, because this person had to remember that they needed to change the code. Eventually the committee stopped changing the code, so it has remained the same for months.

While this solution is workable for allowing [redacted] staff into the building, it is inconvenient.

The current booking system





The previous booking system was based on whiteboards inside of the cricket nets, and spoken agreements of certain teams getting certain days of the week at certain times. If there would be any chance of allowing non-staff to book out the facility, these bookings would need to become more formalised. This booking system also meant that the administration had no idea how effectively the facility was being utilised, and any time a staff member wanted to reserve the facility, they would need to be present in-person to update the whiteboard. This worked fine for the regular bookings, where teams would have practices at the same time each week, but caused difficulty during the school holidays where individual players would have different times than during regular time.

To ensure that this was actually a problem, I interviewed multiple cricket staff members. In the past I had experience with developing a solution that slightly inconvenienced some people but improved the workflow greatly for other people. To my surprise, everyone I asked was eager for me to develop a solution, even if that meant that they would need to book timeslots through a website. They said that they would prefer a solution that ensures that there are no scenarios where they believe they have access to the cricket nets but they hadn't fully read one of the whiteboards around the building.

Stakeholders

For any potential solutions that I propose, there will be 4 main groups of stakeholders:

1. [redacted]cricket staff (Including the cricket committee and coaches)
    - I need to make sure that any solution I create does not inconvenience the [redacted]cricket staff. The cricket staff are the main stakeholders with any access system I create because if they do not feel that the outcome is usable, the system won't be used and I will have wasted my time.
2. [redacted]cricket students
    - The cricket committee wants to engage students to ensure that the facility is being used as optimally as possible. Students are the easiest customers for me to reach because of my project's connection with the school.
3. Members of the public
    - Members of the public are stakeholders in this project - If they don't like the outcome then they won't use it, and instead will go to alternative facilities such as [redacted]Cricket Club.
4. [redacted]Property team (which I was not aware would play such a major role at the beginning of the project)
    - If I was going to change who had access to any building, I needed to get permission from the property team, and needed to have any hardware installed by them to ensure that there were no safety issues.

My client was [redacted], who was a member of the Cricket Committee. [redacted] was easily contactable throughout this project.

Possible solutions

After looking at the problem, I realised that there were multiple solutions that could be used. [redacted] was a viable, but expensive solution.

On the other hand, implementing a spreadsheet for a central plan of who would be using the facility at what time would solve the problem of the committee not knowing how many people use the facility and how frequently, but it would require people to accurately record when they used the facility, and wouldn't allow members of the public to use the facility.

However, there was an alternative solution that would be easy to implement: a google booking form. This would allow for people within [redacted] to easily see whether a given time was taken by someone else, by visualising in a Google Sheet. This is similar to the approach that [redacted] took when making their application to book their sports facilities.

It uses a series of checkboxes, date pickers, and long answer sections to create a booking. This would be possible in Google Forms, and would have been easy to implement. However, it provides a pretty abysmal user interface and doesn't let you easily see what times people are available. This user interface is serviceable, but doesn't solve the problem properly.

From my research it was clear that there were a few options available that would maximise the benefit of the outcome.

- Using [redacted] (or a similar alternative) would be expensive, but provide a high quality solution that has been tried and tested
- Using a Google Forms type solution wouldn't allow for members of the public to make bookings
- Asking people to send emails wouldn't attract customers because people expect to instantly be able to make bookings and are sceptical of making transactions over emails. Members of the public would have difficulty confirming that the email is official.
- Creating a custom solution would allow for me to minimise the downsides of each solution while maximising the upsides.

Initial meetings

From my initial meeting with [redacted] (who is part of the cricket committee and was my main client) I made these notes:

- Timeslots should run from ~6AM-9:30PM
- A period of time throughout the day should be set for [redacted] coaches only
- The school does not want me to handle payments directly, therefore I should use a topup code system

- “We want the [redacted], not the [redacted]’, and “We aren’t a professional club, so we don’t need to overdo a solution” ie it is better to have fewer features but have them work well, instead of tons of features that only partly work and wouldn’t get used
- I would need a way for users to report any damage they see
- I should have a meeting with [redacted] (head coach of the 1st 11 team)
- I should investigate who is in charge of security products at [redacted]

I then spoke with [redacted] (head of sport at [redacted], who said that he supported what [redacted] had said, and that it is important that we solve the problem of students being finding a staff member’s permanent access code, but that we also need to keep having a convenient way for staff to enter the facility (therefore we can’t rely on rolling codes).

I then spoke with [redacted] (head coach of the 1st 11 team), who said that I would need to make a way for cricket coaches to be able to quickly book out a timeslot across a whole school term, instead of making one booking at a time. He also said that the system can’t come at the expense of [redacted] staff, so he suggested that if someone had made a booking, but then [redacted] needed the nets at that time (eg if a game was in the afternoon of that day and they needed the practice) an [redacted] staff member should be able to cancel the booking.

I was opposed to this solution, so began thinking of alternate ways to ensure that [redacted] staff have priority access. Cancelling peoples’ bookings that they thought had been secured is a very user-unfriendly practice, which will leave a bad impression for many users. While it does accomplish the goal, it doesn’t make everyone happy. I came up with a compromise that slightly disadvantages members of the public while improving the app for staff where staff members can book as far ahead as they want to, but members of the public can only book up to 3 weeks in advance.

Initial pathway

Following this meeting I realised that the problem with giving staff members a 4 digit code to the facility was that the code was something that the staff members would know - which can’t be taken back from someone once they know it.
There are three generally recognised types of authentication methods (Stewart, 2018):

- Something you know
- Something you have
- Something you are

If I couldn’t use something they would know, I would either need to use something they have or something they are. Something they are (like a fingerprint scanner) would prevent anyone from gaining unauthorised permanent access, but it would be

inconvenient if a student needs access without a teacher being nearby, and would likely lead to someone creating a workaround for entering the building - which defeats the purpose. Therefore I should use something they have. I decided to settle on RFID or NFC cards/tags, because they can be purchased cheaply, easily, and in 2 formats: keychain tags or credit card size.

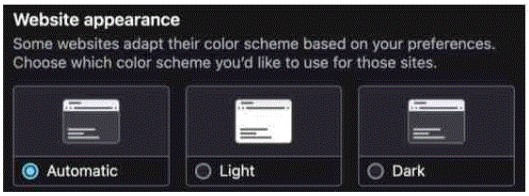
I realised that there was another device that people already have with them that can be read by scanners. The vast majority of mobile phones sold these days have an NFC antenna in them. I decided to keep the idea of using NFC in mobile phones open.

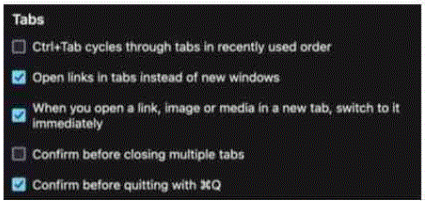
I then planned for how I would implement UE methodologies into the development of my app, which informed the rest of my planning.

User Interface

The user interface methodology focuses on predicting what users will need to do so that the UI is designed around making the important elements that users want easy to find. This methodology is related to multiple of the other methodologies.

To follow this methodology I will follow standard industry practice wherever possible. For example these screenshots are from the Firefox settings page:





These two sections show two different methods of selecting preferences: multiple mutually exclusive options, or a group of options with a similar topic but aren't

mutually exclusive. The round radio button clearly indicates that only one option is selectable for colour scheme, whereas the square checkboxes show that multiple options can be selected. All of this information about how the UI can be used is told to the user without them having to read any words.

I will also have sensible default settings for my users. An example of this is light vs dark mode. Although I personally am a fan, most users will never care enough to change the setting, and light mode has better readability in sunny conditions. To anticipate the user's desire for readability, I will use light mode as the default setting.

Accessibility

It is important that my website is accessible to as many users as possible. While some people may initially think of website accessibility being a low priority, it is an important part of design. Not only are accessibility features useful for people with disabilities (who otherwise wouldn't be able to use certain devices), but they can also be more convenient for everyone. An example of this is keyboard based navigation. This can be used by people who are unable to use a mouse/trackpad, but is also a benefit to other people who are able to because now they can use the app faster. An example likely to come up with my app with at least one user is a language barrier. By minimising the times when reading text is necessary (for example by adding icons and using colours to indicate meanings) the app can be used by more people.

Using colours is an important thing to be mindful of. While for most people they can be used to quickly indicate important information, some users are unable to distinguish between all colours available. Given that colour blindness is experienced in some way by ~8% of men (Sugue, 2024), and [redacted] is an all-boys school this is actually quite likely to come up at some point.

To avoid causing accessibility issues I will:

- Use a secondary method of communication whenever colour is used to communicate (for example having text be red, instead of just having a red somewhere to indicate an error - if the colour can't be distinguished then the text can be read)
- Use a high contrast between my text and background colour. This shows that accessibility isn't just relevant to people with disabilities, because all people can temporarily need accessible apps, for example if their phone screen isn't very bright and they're in the sun.

Information Architecture

Information architecture helps users understand where they are, what they've found and what to expect.

This is definitely relevant to my app. There is no point in me giving my users information if it is laid out in a way that they cannot understand the meaning of or where they are meant to go to find what they are looking for.

How will I implement information architecture into my app?

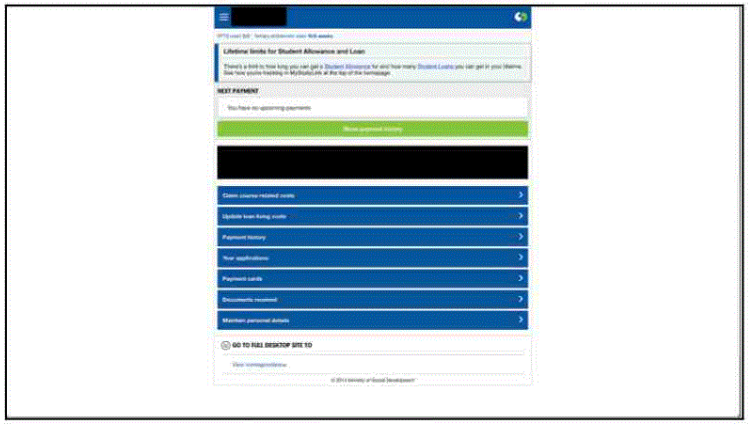
- The home screen will provide easy access to the important locations throughout the app.
- I will make it clear what pages are available to click on by using consistent button styling throughout the app.
- I will use AppBars to show the user where they currently are in the app and what they should do to continue throughout the app.
- I will use colour coding (but not exclusively rely on it) to allow users to quickly differentiate between data.
- I will use an app layout that is found within other apps, so that I am not introducing a new concept to users.

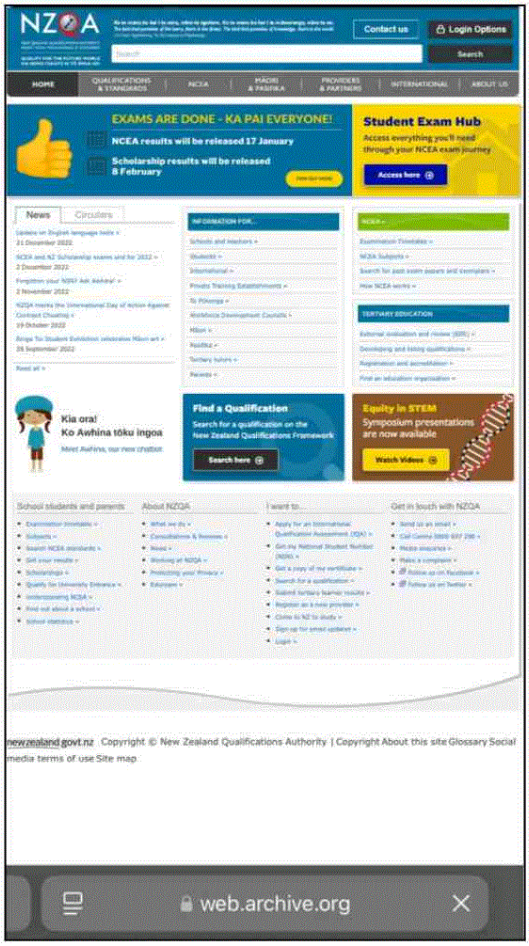
Social/physical environment

This project was aimed to be used by both non- and people. By allowing non- people to have access to the cricket nets, I am opening up the possibility of vandalism, and misuse of the cricket nets. For this reason my key stakeholders and I came to the conclusion that we would need to keep logs of events such as when each booking was for, and keep these logs for after the bookings had concluded, so that if any damage was reported, it could be traced back to a selection of a few possible bookings. My target audience was either school students or parents, so I have assumed that they will not need extra help using the app more than a demographic such as elderly users would need.

According to Cloudflare Radar, mobile browsers make up ~55% of the world's internet users. After seeing this ratio I decided to make my project using a mobile-first, desktop-second approach. My reasoning for this was that mobile is more popular (especially since my project would be sometimes used at the cricket nets, where people almost certainly don't have a desktop computer with them), and also that a mobile-first website can be used to have a mediocre experience on desktop (for example the mobile version of MyStudyLink is

usable because it just adds big borders to the sides), whereas if I try to look at a desktop website on my phone, I can barely read anything (for example the old NZQA website had no mobile version, and was very difficult to use).





As for the physical environment, any outcome I created would need to fit around the cricket nets door. While this area does have power, it does not have any ethernet connectivity. Therefore, my project must be capable of running over wifi. While ethernet would have been more reliable, there isn't much I could do in this regard, and the wifi coverage was quite good due to the building being near the sports office.

Part 2 - Initial Thoughts and Ideas

Version Control

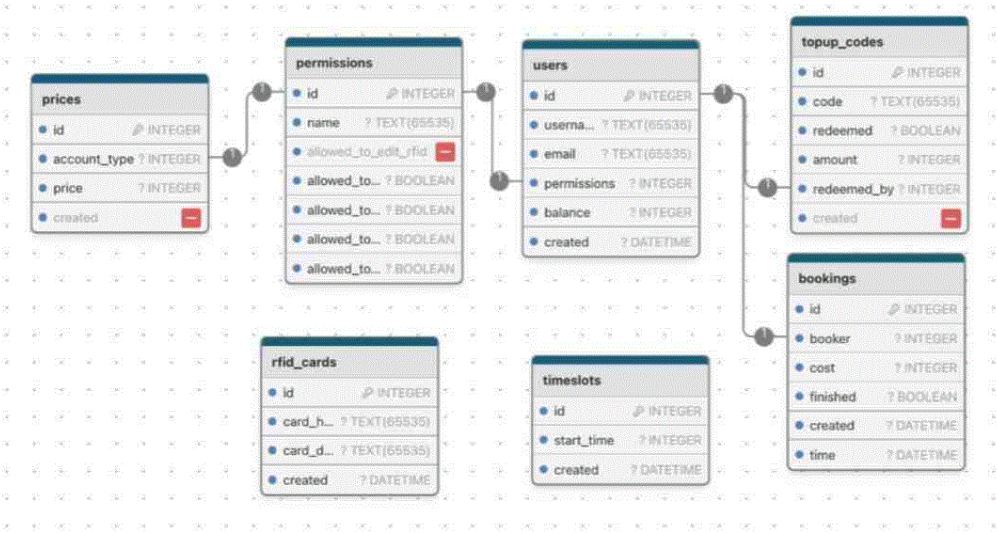
I knew that no matter what tools I used to develop my outcome, I would need to have version control. While there are multiple options for version control (such as SVN or Mercurial), Git is the most popular tool for this job by a wide margin and works well for a small codebase like mine. Because the git commands are built into the VSCode UI, I will be able to easily manage the branches and commits.

To keep an off-site backup I decided to use GitHub, because it can track issues and make pull requests. There are other options, but the chance of my version control hosting website impacting my software outcome is extremely low, so I did not spend a long time deciding on this and instead used the platform that is most popular.

Technical Decisions

At this point I had come to the conclusion that the best solution to this problem would be an app-based or web-browser based application that runs on the users' devices connected to a database on the backend, which a lock on the cricket nets door is able to access.

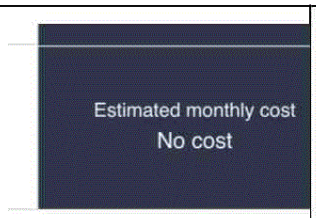
I used DrawDB to mock up what my database will look like.



From previous experience creating a web app with different levels of users, I knew that it would be a lot simpler for me to create the permissions relation field at the beginning of the project. I wasn't sure what permissions I would need from the very start. By splitting the permissions field out, I could make it so that I could create a record in the permissions table that has data like `id=123456 name=Student allowed_to_edit_rfid=False.` While this database layout did mostly work, I did need to make some changes to it throughout the project.

My client had no experience with web app development, so had no input to give me in regards to backend development. I decided to choose the backend as a service that I had experience with and felt would give me the flexibility to do what I needed to get done. I chose PocketBase - a free and open source backend. PocketBase also has some pros and cons. Here I have decided to compare it with Firebase, a popular solution that aims to solve the same problem as Pocketbase.

	PocketBase	Firebase	Building a custom backend

Pros	- I know how it works - The full source code is on GitHub, so I could download the entirety of it right now, and never lose it, even if ganigeorgiev decided to kill it - Always free forever - Extendable using JS - It is impossible to make an infinitely recursive API endpoint that causes a thousands of dollars cloud bill - Can be extended with JS	 - Likely to cost nothing - Has more features - Is more well documented - Hosted by Google	- Fully customisable - Free forever
Cons	- Is still in beta - Is liable to have breaking changes - Needs to be self hosted - Can't infinitely scale	- Uses a proprietary database format, so I would be locked in to it - Could change pricing at any moment - Is made by Google (who are famous for killing successful products. A recent example of this is Google Domains)	- Easy to introduce security risks - Vastly increases the amount of work I would need to do - If any manual DB editing needs to be done it won't have a simple UI

It is worth noting that while I wrote down that a con of making a custom backend is that it is easy to introduce security risks, it decreases the chances in another way. The problem I am referencing here is that PB and Firebase both let you use API rules to control how you can edit the DB, whereas with a custom API you only create the endpoints that you want users to be able to use.

Firebase is notorious for allowing users to have terribly written API rules. A major example of this was the API rule bug that was present in the Arc Browser's database

(which used Firebase). Arc has a feature where you can customise any website with your own JS, and sync these customisations to your Arc account. However, they didn't have a check that when you updated a customisation, you left the userID as your own. Therefore, someone could have created a malicious customisation that would run in the browser of anyone if they knew the userID (Agrawal, 2024).

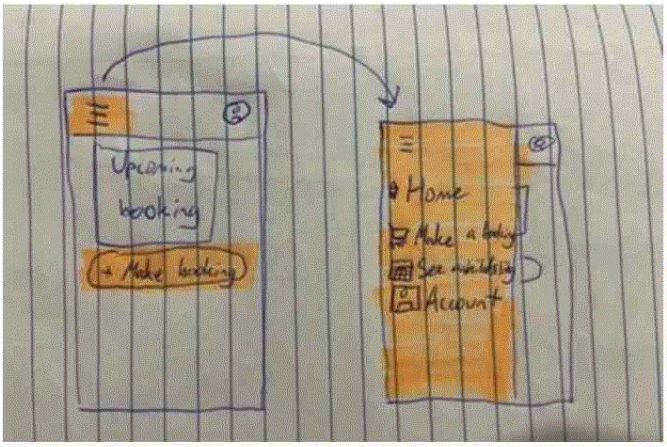
While this security risk can be mitigated by creating a custom backend, I decided that this didn't outweigh the security risks introduced by me not relying on already existing and externally created code. The code is open source, so there is virtually no chance for the creator to use the data in a way that I don't know about or use my app's data to train an AI on. On the other hand, Firebase is proprietary, so I can't see what is happening to my data on Google's servers.

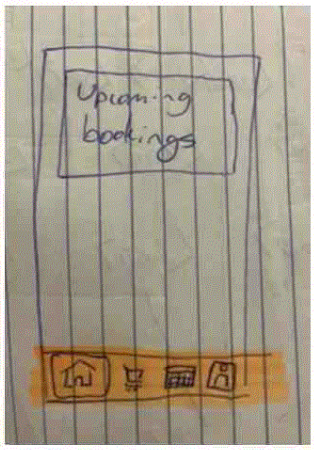
I also needed to decide how I was going to build my frontend UI. Ideally I would be creating an app that runs natively on each platform. However, realistically many users are very hesitant to install an app, and would much rather use their web browser to conserve storage space on their phones. Also, some users may be using obscure platforms where it is difficult to test for native support on (sorry linux), so a web backup would allow them to still use the app. Also, some users may be using laptops setup by their organisation that don't allow new apps to be installed. In theory, I could create a separate website and mobile app, but this would require double the time to be put in any time I made a change and would drastically increase the amount of work I would need to do. Because I am just one person, I decided I would only be running one codebase for all platforms. When taking into account what I was trying to achieve, there were two possible ways of attaining this that I was able to discover: React Native for Web, and Flutter. Both platforms have their advantages, and all were able to support the platforms I was aiming for.

Ultimately this decision came down to the fact that I had more experience in Flutter than with React Native. Both solutions would have worked for this problem. Flutter chooses to be cross-platform as its central pillar, so runs effectively like a game engine where the entire app is rendered through a custom engine bundled with the code that I wrote. This means that there is a very high chance that any code I write will work on all platforms that Flutter supports. End users do not care about these differences, and just want an app that works well. Because I know how to use Flutter already, I decided to choose this for my outcome. Flutter uses the dart language, which is uncommon outside of the Flutter world, but is relatively easy to learn, and generally doesn't have as many strange quirks as JS, so if someone was attempting to maintain the code it may be challenging but doable.

Mockup designs

I then started to mock up designs for the app on paper

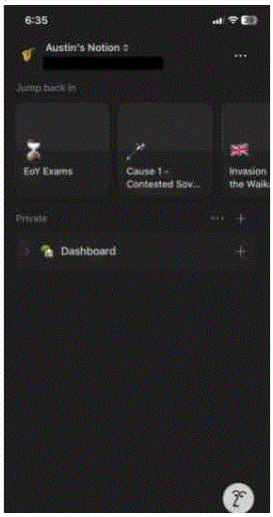


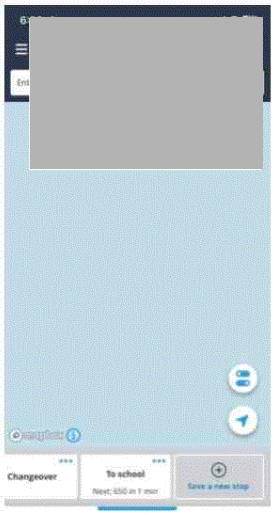


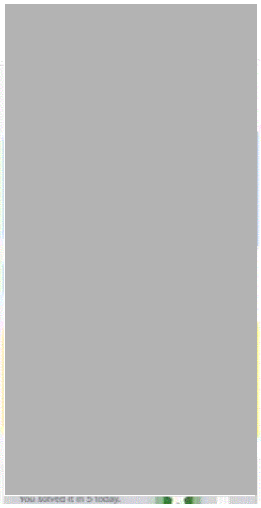
I had two ideas for the main method of navigation around the user-side app. I knew I wanted to be able to have multiple pages easily accessible, but I wasn't sure whether I should go with a slide-out menu, or a bottom navigation bar.

	Slide-out menu	Bottom navigation bar
Pros	- Less ambiguity over what each icon means - Can fit many items	- More simplistic, therefore easier to understand - Used in almost every mobile app
Cons	- Requires pressing the hamburger button to see all the options available	- Only supports up to 4 or 5 icons before it starts to get cramped

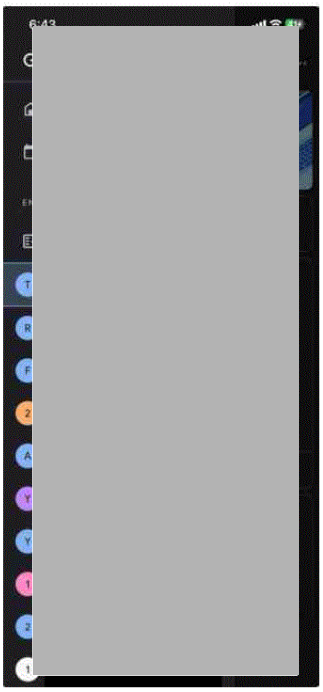
When I asked my client for their opinion, they said that they thought I should go with the bottom navigation bar, because it is very unlikely that I will need a UI with over 4 or 5 main pages. I agreed with this decision because of the pro that it is used in almost every mobile app. Here's three apps that I have on my home screen, all of which use bottom navigation bars





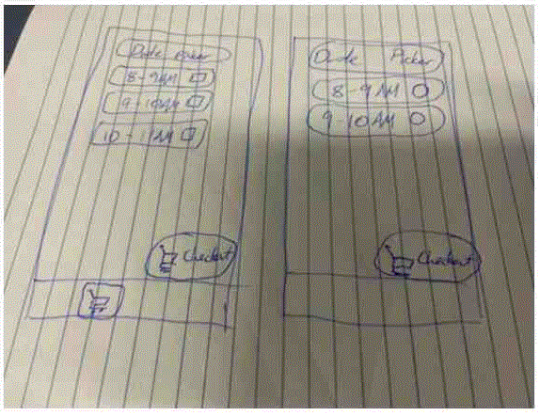


Whereas I can only think of 1 app on my phone that uses a slide-out menu - Google Classroom. Except the Google Classroom also uses a bottom navigation bar once you've selected which class you are looking at. I know that Jakub Nielsen's law says "Users spend most of their time on other sites", so if I want to give users the best chance of understanding how to use my project, I should emulate what other apps look like.

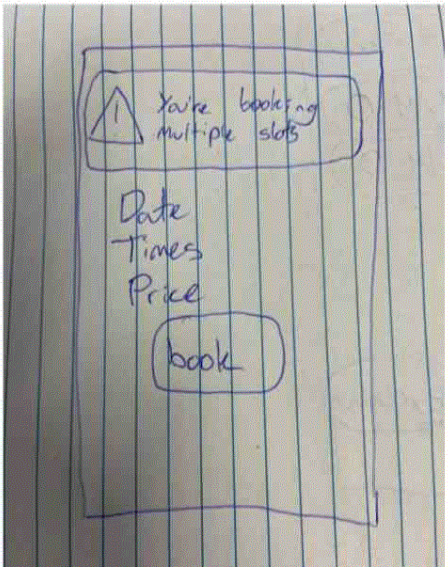


Because the use case for slide-out menus in other apps is when there are either many options to choose from or a customisable number of options, I felt that the bottom navigation bar would be the best choice. I asked my client for their opinion and they agreed. When I began this, I hadn't noticed that every app I have shown also has an AppBar (a title bar at the top of the page with some buttons), which I later decided to integrate into the app.

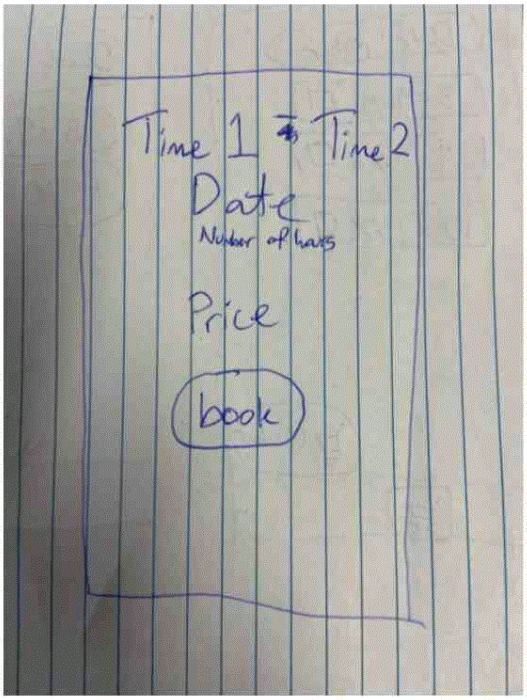
I then knew that I would need some way to let users pick which date and which timeslot they wanted to pick. I started laying out a timeslot selection page, when I realised that I would need to account for if someone wanted to book multiple timeslots in one day. It would be very inconvenient if they had to go through the checkout process multiple times, so I asked my client if they people should be able to select multiple timeslots. They agreed with me that this is likely to come up, so I should allow for multiple slots to be selected.

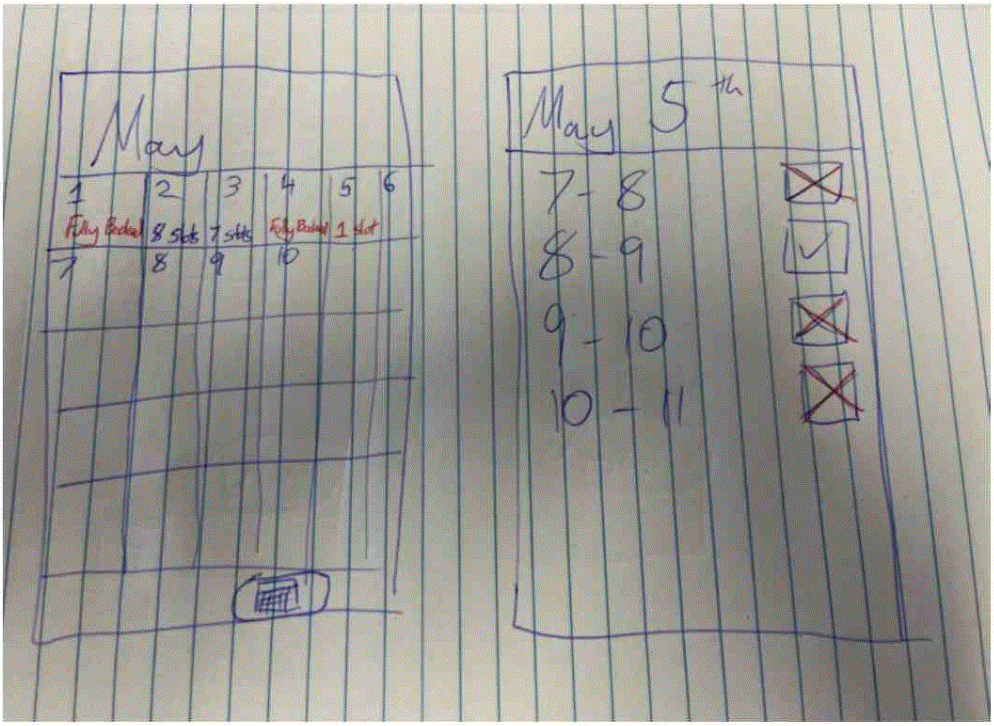


To represent that multiple slots are selectable, I decided to use checkboxes (square) instead of radio buttons (round), because the square buttons show that the options are not mutually exclusive.

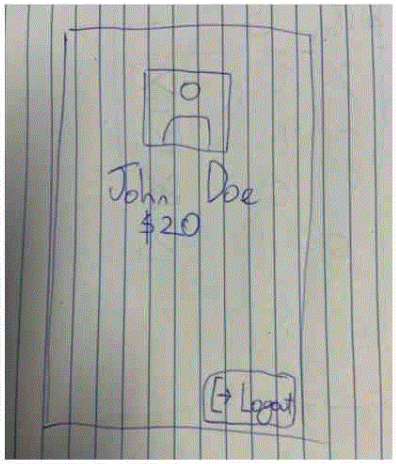


I then created a design for what the checkout page could look like. I asked my client for their feedback on this design, and they said the warning looked too much like an error, implying that you aren't allowed to book multiple slots. To fix this, I decided to create a new layout where if the slots are back to back, the app shows the start of the first to the end of the last times. This layout got approval from my client.

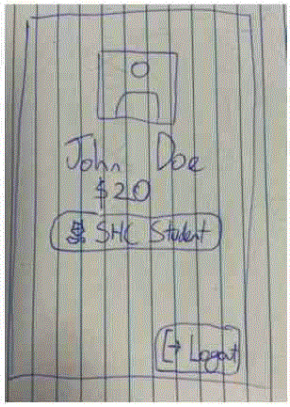




I then drew up a design for the calendar view pages, which would allow people to view which would allow for people to see which timeslots are available on any given day. The use case for this feature is if there is someone who knows that they want to book a timeslot, but aren't sure which day they want to book for, so they want to know which dates have the best availability. Once they clicked on a day, they would be able to see the page on the right, which shows which timeslots are already booked and which ones are available.

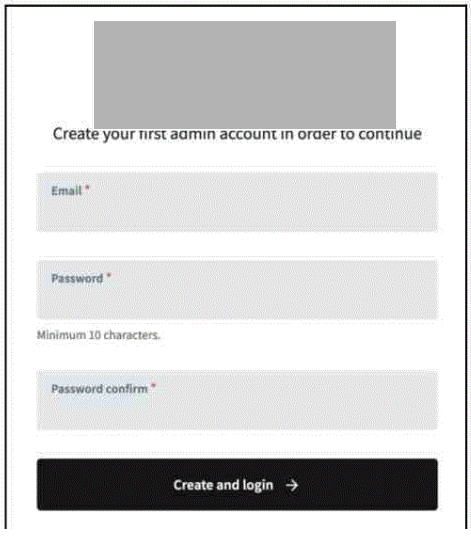


I then made a layout for the account page. I asked my client for feedback and they said there should be some way to tell whether your account has been marked as [redacted] student / staff or just a regular account. I added a card with this information to my design, which got approval from my client.



Part 3 - Getting Started

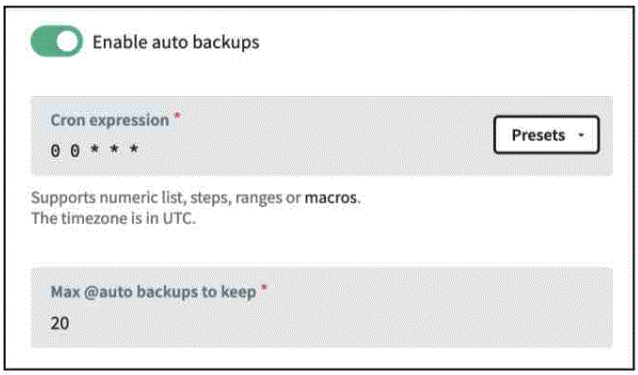
Software



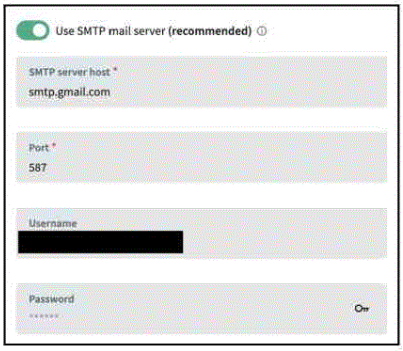
I started by downloading the latest version of PocketBase to my laptop. By the time you are reading this, PocketBase v0.23 will have probably been released, which has some breaking changes in the naming of builtin functions for pb_hooks files. My project currently uses the v0.21 and v0.22 function names. PocketBase can run on windows, macos and linux, and I can

easily transfer the data over to another platform by copying the `data.db` file. I made a new admin account, then in settings I enabled autobackups.

There is a tradeoff when enabling auto-backups. Whenever PocketBase makes a backup, it temporarily goes into read-only mode and any write requests submitted will need to wait until the backup has finished generating. Because my project will likely never have any users at midnight trying to make bookings, I decided that a ~5 second slowdown at midnight every night would be an acceptable tradeoff to make in the name of having backups of the whole database. Because the data saved by my database is limited in scope, I have decided to keep 20 auto backups at a time. The higher this number the better, but I have a limited amount of storage. 20 is a medium between having a lot of backups, and not taking up too much space. I can adjust this number at any time if I find out that I have run out of storage, so I am not concerned about getting the exact number correct at this time.



I then enabled the SMTP server. This was important because it meant that if I ever forgot my admin password, I would be able to easily recover access to the pocketbase DB. There are alternative methods of entering if you edit `data.db`, but being able to reset your password the intended method is the easiest way in.



Once I had PocketBase running, I created a new Flutter app. I started by creating a login page. I decided to test my app using the android emulator, because I found it to be less temperamental than the iOS simulator or running on MacOS.

I started by creating a login page. I had originally considered just using emails and passwords for authentication, which was built into PocketBase. However, I considered that it was important to my outcome that it was secure. No matter how much I try to convince users that they shouldn't reuse a password, it is an unachievable goal for the vast majority of users. Because I wouldn't be able to alter the parameters of

passwords (such as minimum length or special character requirements) without annoying any potential users, I knew I would need to find another solution. After some research I came across the Flutter package passkeys. This package enabled passkeys on web, android and iOS. Use public/private key cryptography to securely authenticate with apps and websites in a way that is simple for the user. I decided to integrate the ability to login with a passkey into my login page. While attempting to integrate this, I discovered that when I was researching how to implement passkeys into Flutter and PocketBase, nobody had done this before (or at least nobody had published it). I did find someone who managed to use passkeys with PocketBase to be used on a JS frontend https://github.com/FlixMa/webauthn-pocketbase. I adjusted my pocketbase to use their modified `main.go` file. This gave me the custom routes that I needed to begin and finish login/signup with a passkey. I then started to implement the frontend code. However, after looking at the usage section of the passkeys package I found that there was less detailed instructions than I had anticipated because the package is created by Corbado, a company that offers their own passkey authentication backend. The instructions assume that you are using the Corbado servers, so it provides the methods necessary to respond to a WebAuthnChallenge, but not how to receive the challenge. At this point I felt that the easiest method would be to try to adapt the frontend code that FlixMa wrote and translate it into Dart.

```
const publicKeyCredentialRequestOptions = await pocketbase.send(`/webauthn-begin-login/${btoa(data.username)}`, {
  method: "POST"
})
console.log("publicKeyCredentialRequestOptions", publicKeyCredentialRequestOptions)

const assertion = await getCredential(parseRequestOptionsFromJSON(publicKeyCredentialRequestOptions))
console.log("finishLogin: send assertion", assertion)

const finalResult = await pocketbase.send(`/webauthn-finish-login/${btoa(data.username)}`, {
  method: "POST",
  //query: data,
  body: assertion
})
console.log("beginRegistrationAction finalResult", finalResult)

pocketbase.authStore.save(finalResult.token, finalResult.user)
```

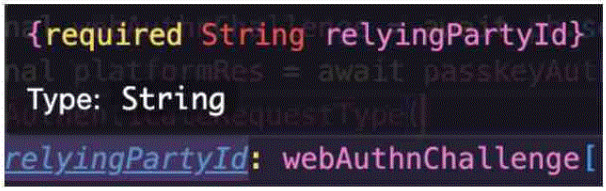
[Not my code]

In Dart, I have access to pocketbase.send(), btoa() [which is called utf8.encode()], getCredential() [handled by the passkeys package, called passkeyAuthenticator.authenticate()]. The difficulty here comes from the fact that Dart has no parseRequestOptionsFromJSON() [which was imported from the webauthn-json/browser-ponyfill module] function.

To resolve this I used VSCode's debugging tools. By pausing the app's execution I could see the values inside the platformRes variable. This mostly helped me, but I also relied on Dart's VSCode extension that showed me the type of each input by hovering over it. This let me know where

I needed to convert between integers and strings, for example.

I then added an extra button to allow for logging in with a passkey. After asking my client for feedback, they said that the UI was a bit confusing for them when they first used it. To simplify it I decided to combine the 'login' and 'login with passkey' button into one 'login' button that either uses a password if one was entered, or uses a passkey if the password box is empty.



My client said that this change made the login screen easier to understand.

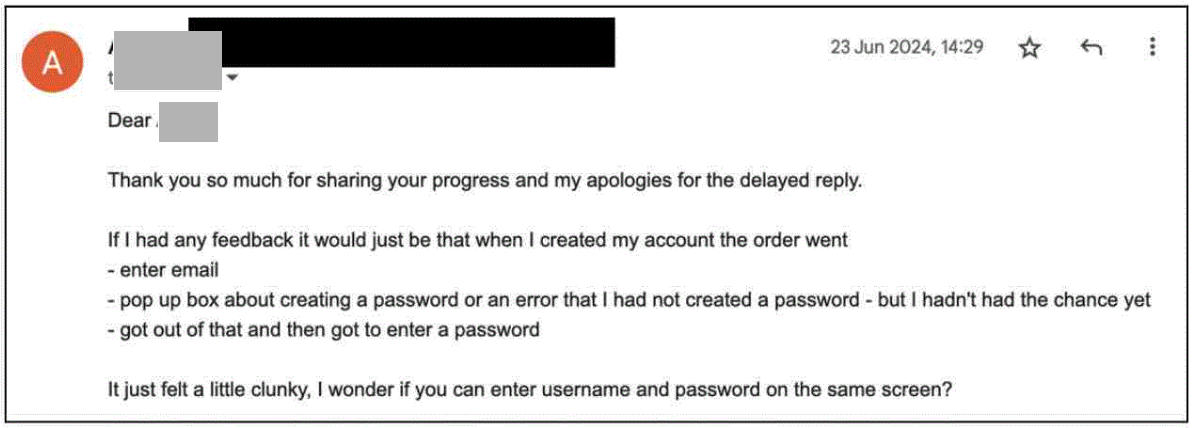
I am fairly certain that most users would find passkeys more convenient to use than a password, and they are definitely more secure than a password. Because of this, I decided to make passkeys the default method of authentication when creating an account.

I set it up so that the flow was

- Go to email entering page
- Get the OS's popup asking to make a passkey
- If the OS says the passkey wasn't made:
    - Show a popup asking if the user wants to try again or just use a password
- Otherwise:
    - Go to the login page with the email autofilled so the user can understand how to use the passkey they just created

I implemented this and asked my client to try logging in without giving her any specific instructions about how to login. She does not keep up with the latest additions in web technology, so had never used passkeys before.

Here is the feedback that she shared with me:



A [redacted] 23 Jun 2024, 14:29

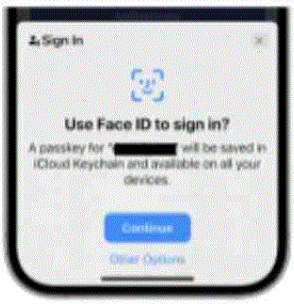
Dear [redacted]

Thank you so much for sharing your progress and my apologies for the delayed reply.

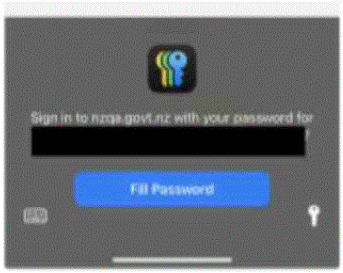
If I had any feedback it would just be that when I created my account the order went
- enter email
- pop up box about creating a password or an error that I had not created a password - but I hadn't had the chance yet
- got out of that and then got to enter a password

It just felt a little clunky, I wonder if you can enter username and password on the same screen?

I was confused after hearing this. I knew that she was using iOS (which I had tested using my phone), and passkeys had been a feature of iOS for over a year. Regardless of how this managed to happen, it was clear to me that I needed to change the messaging around passkeys, because users wouldn't understand what I meant.



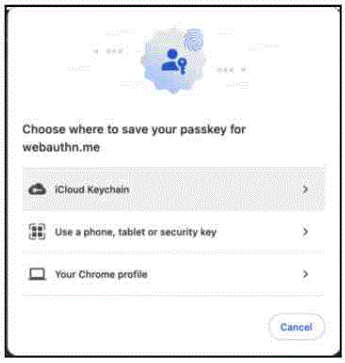
Making a passkey



Filling a password

After some investigation with her, I realised that her phone did support passkeys. Our best guess of what happened was that she saw this message, and assumed that her phone had made a mistake and was trying to autofill a password for a different website, which has a similar UI on iOS to creating a passkey. The confusion may have also come from the UI saying 'sign in' instead of 'signup'. This confusion caused her to press cancel, which was treated by my app as an error during passkey creation, so it showed her a generic error message.

Unfortunately, I don't have any control over how the OS decides to show the passkey UI, so I need to add more notices about what is about to happen when a user creates their passkey.

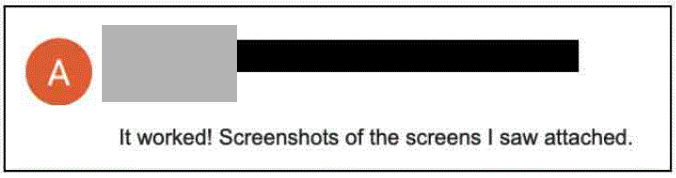


Because of this, I decided to check the wording on other platforms. On Chrome (macOS) the UI is more clear, specifying that a passkey is being saved.

On Chrome (Windows) and Edge the UI is also similar. On Firefox (macOS) the default MacOS passkey UI is used, so it also has the strange wording.

As a result of this I decided to change the flow for creating a new account. I have updated it to:

- Go to email entering page
- Get taken to a page that explains that a passkey is about to be created, with a button that creates the passkey
- Get the OS's popup asking to make a passkey
- Continue with the same flow as before



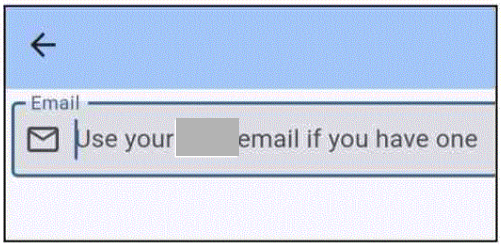
When I asked my client to try again creating a new account, with the only extra instruction being to look out for a message saying "use FaceID to sign in?", she was able to create a passkey.

Once I had the basics of the login/signup page completed on the frontend, I decided to customise the backend code for making a new account. My client had suggested that there should be different price ranges for students, staff, and everyone else, so I would need some way of differentiating the types of people when they made an account. I came to the conclusion that the simplest and most effective way of achieving this with minimal extra work created for cricket staff was to ask students and staff to create their account with their school email address. Because staff emails end with [redacted]`.school.nz` and student emails end with `@students.`[redacted]`.school.nz`, I could differentiate between students and staff, and if the user's email begins with neither of these, they would have a default account.

```
onRecordAfterConfirmVerificationRequest((e) => {
    const e_record = JSON.parse(JSON.stringify(e.record))
    const email = e_record.email
    if (email.endsWith("[redacted].school.nz")) {
        console.log('ends with [redacted]school.nz')
        const record = $app.dao().findRecordById('users', e_record.id)
        record.set("permissions", "0mf1jywqfhzsukh")
        record.set("signup_finished", true)
        record.set("balance", 0)
        $app.dao().saveRecord(record)
    }
    else if (email.endsWith("@students.[redacted]school.nz")) {
        console.log("student email")
        const record = $app.dao().findRecordById("users", e_record.id)
        record.set("permissions", "498dm3tt0vuvgiw")
        record.set("signup_finished", true)
        record.set("balance", 0)
        $app.dao().saveRecord(record)
    }
```

```
    else {
        const record = $app.dao().findRecordById("users", e.record.id)
        record.set("permissions", "gkr18gwzscrz2j0")
        record.set("signup_finished", true)
        record.set("balance", 0)
        $app.dao().saveRecord(record)
    }
},)
```

Here I also set the balance value to 0. In theory, this could have been accomplished using a PocketBase create rule, which I could set to `balance = 0`, so that if any account creation request came in where someone had used their web-browser's dev tools to send a request directly to the server without the UI and set the balance to something other than 0, their request would be denied. However, it is possible that in the future I will add other methods of account creation and forget to implement a similar check, so I decided that the best solution was to check both when creating an account through PB API rules and when verifying the account's email using a JS hook.



My client agreed that automatically determining the account type from the emails was the way to go, but told me that I should definitely make this clear in the frontend when making an account. To do this I set the email TextField's hintText value to reflect that emails are preferred.

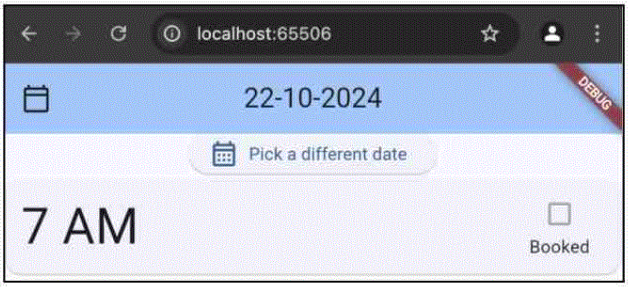
I then moved onto creating the booking page. To do this I knew the frontend would need to know which times are available on each date. I created the API endpoint at /api/shc/gettimeslots/:day/:month/:year.

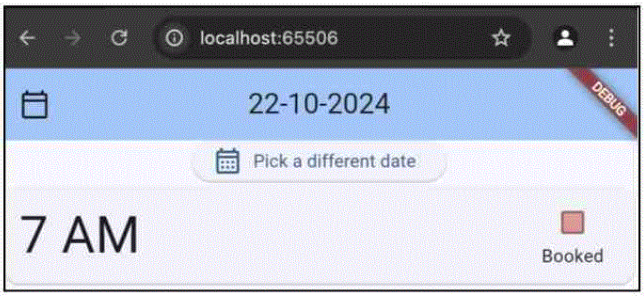
```
const day_booked_slots_db = $app.dao().findRecordsByFilter(
    "bookings",                                    // collection
    `year = ${year} && month = ${monthNum} && day = ${day}`, // filter
    "-created",                                    // sort
    999,                                           // limit
    0                                              // offse
)
```

The endpoint first gets all the bookings that occur on the year, month, and day that is being looked at, and gets all the possible timeslots that could be booked at. It returns a list of objects, which looks like this `[{start_time: 11, booked: false}, {start_time: 13, booked: true}, {start_time: 14, booked: false}, {start_time: 15, booked: true}]`

While I originally considered doing this processing on the client side, I realised that this would be much slower than necessary because of the unnecessary data transfer. It would also require letting the client have access to read all bookings, including being able to see who made the booking. This is clearly much worse than processing it with a PocketBase hook.

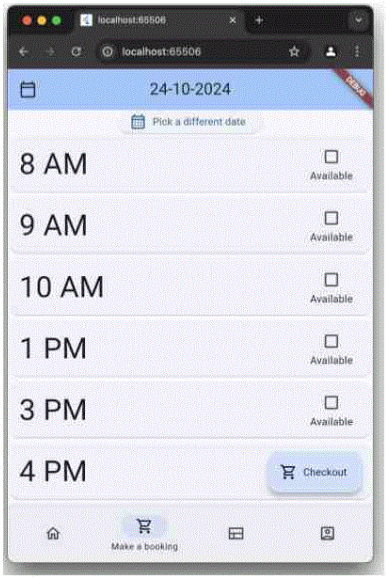
I then created the UI for booking a timeslot. I originally created a list of cards based on the data returned from the API endpoint I mentioned above. I had already decided to allow for multiple slots to be selected during the planning phase, but I hadn't considered how to indicate that a timeslot was booked. My first attempt at this design used the CheckBox's property where onChanged can be set to null, which disables the checkbox, making it not react to any changes and greying it out. Because it was difficult to tell apart the grey from the darker grey that indicates an active checkbox, I decided to add the word 'Booked' underneath the checkbox. I had assumed that this would be acceptable, but during some user testing I discovered that I had overestimated how text in a UI the average user is willing to read.

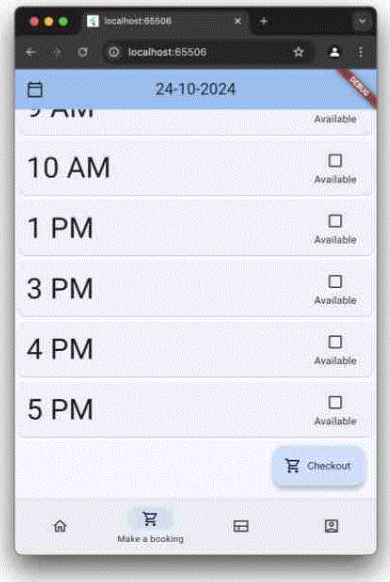


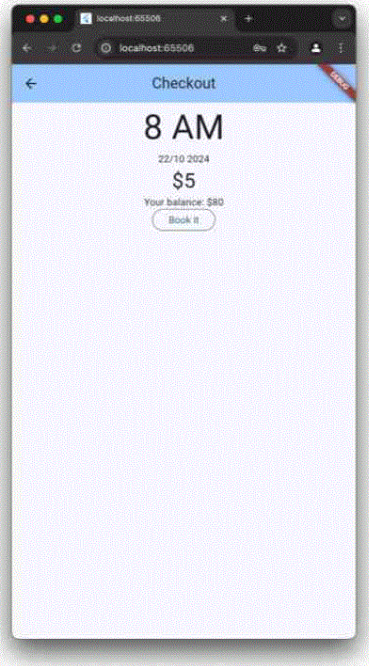


As a result of this user testing, I made the checkbox red when it is unavailable. This is much more easily noticeable to users and requires reading zero words.

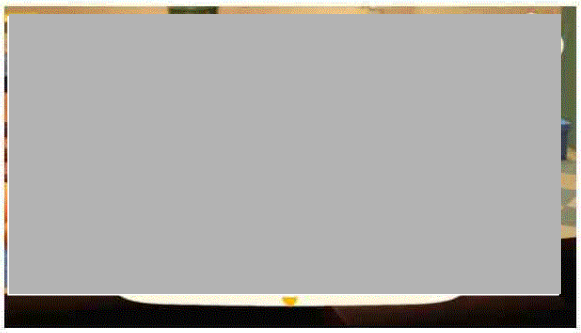
In the creation of this page I accidentally introduced a bug. I had considered that the user might be using a screen that isn't tall enough to see all the options on one page, so had enabled scrolling. What I had forgotten to consider is that my FloatingActionButton to go to the checkout would end up covering one of the checkboxes. This bug was brought to my attention when I asked my client to use the app on their phone to test it out. To solve this problem, I added a SizedBox below the lowest timeslot on the page, so that the page can be scrolled to the point where the checkout button isn't covering anything

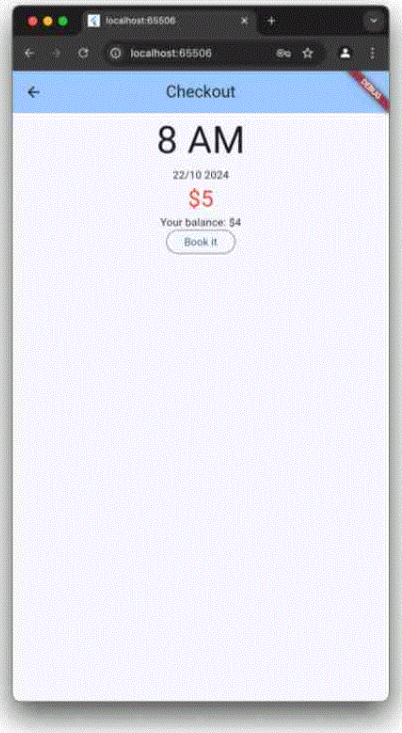






I then moved on to the checkout page. From here I drew inspiration from UIs that I was familiar with. Because most app UIs let you pay with credit cards, they have no way of knowing if your balance will be high enough before making the transaction. However, my only payment method is an account balance, where I will know if the funds will be sufficient before the payment is made. Because of this I can show the current balance, to let the user know if the transaction will go through. This UI design is often present when the software is able to view your account's balances.





When I reflected on this design in later iterations, I realised that the same thing I discovered before (users are much less willing to read than I think) also applied here, so I should also have an alternative way of indicating that the balance is too low. To solve this I decided to make the cost turn red if the balance is too low. This made the checkout page simplistic for the user to understand without needing to read any words.

When I was creating the code for the checkout process, I knew that I would need to handle most of the processing on the server side, especially for account balances. If

I had allowed the account balances to be changed from the frontend, then it would open up easy opportunities for malicious requests to be sent. Originally, I had the frontend send a regular CREATE request to PocketBase's bookings collection, which would then cause a onModelAfterCreate hook to be triggered, which I used to deduct the balance from the user account. While this did work to a basic level, I realised I could optimise the user experience by converting this to a custom API endpoint. The main problem with my first implementation was that if something had gone wrong, there was no way to alert the user in the frontend UI, and they would think their request was successful. While this wasn't a major problem because they could still be sent an email with details, it was still worth it to change to a custom endpoint.

By creating this custom API endpoint, the booking would only be created in the database if the user account that made the request had enough money on it. This also gave me an easy way to send confirmation emails through the PB hooks email service. For the time being, I decided to leave these emails blank, and add the details to them later.

I then moved onto creating the page for entering a topup code. I knew that I would need to keep track of what the value of each code was, whether it had been redeemed (+by who), and the code itself. Originally my topup_codes table looked like this.

☐	je233qj4bj6wpu2	1234	True	20
☐	94qweoz9dwh9ovn	helloworld1234_	True	54

However I realised that there is a significant risk that if the database was compromised, any previously generated topup codes would be accessible. To avoid this problem I decided to store my topup codes as SHA256 hashes.

☐	8vywt4ck0pw4fa2	ee22f477b7caa81de31b0f595f122a5e05911…	False	10

This added significant security, while adding a minimal amount of time delays while checking and generating codes. I had decided to maintain the industry standard of using 16 character alphanumeric codes, with dashes every 4 characters. I decided that in order to avoid confusion with potential typos I would need to have a checksum. In order to solve this, because this checksum isn't security related, I decided to use a simple algorithm that can be easily implemented in both Dart and JS.

```
function checksum(s) {
    var chk = 0x12345678;
      var len = s.length;
  for (var i =0; i <len; i++) {
    chk += (s.charCodeAt(i) * (i+1));
  }
  return (chk & 0xffffffff).toString(16);
```

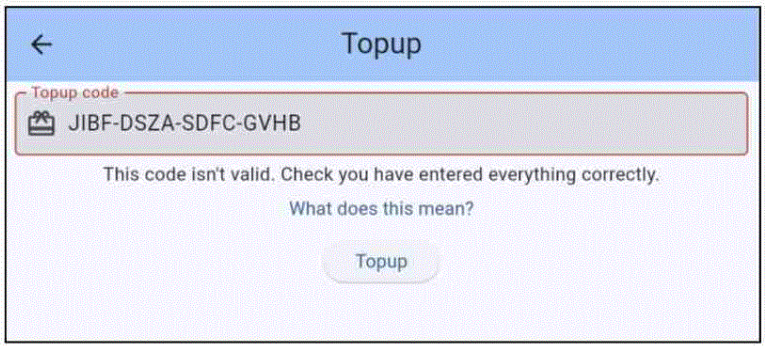
```
    }
```

[Not my code]

The checksum adds together the ASCII values of each character * its location in the string.

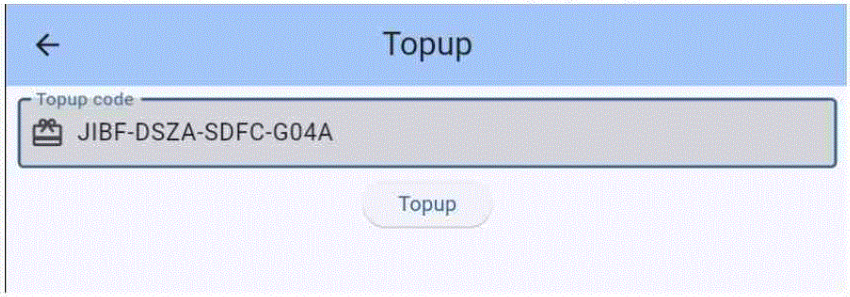
```
const full_code = code + checksum(code).toUpperCase().slice(-3)
```

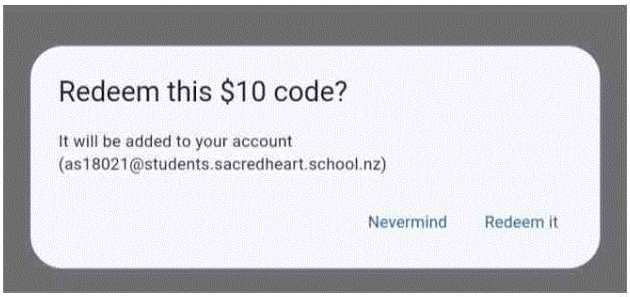
My code uses the last 3 digits of this checksum, combined with the randomly generated part of the code, to create the full code that can be given out.



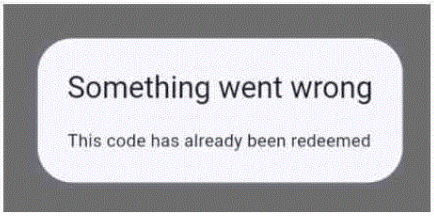
The result of implementing this checksum is that in the UI I can differentiate between someone making a typo/making up random numbers, and someone trying to use a code that has already been redeemed, before the topup button has even been pressed.

Whereas once I change the last 3 digits to the correct checksum, the UI updates and allows the topup request to be sent.

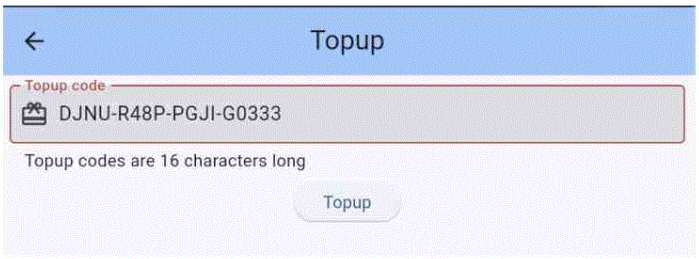




Before my app does the irreversible action of redeeming a topup code, it shows a popup asking to double check, with the email that the topup will be associated with. This is important in case a user had ended up logged into a different account than they intended to.

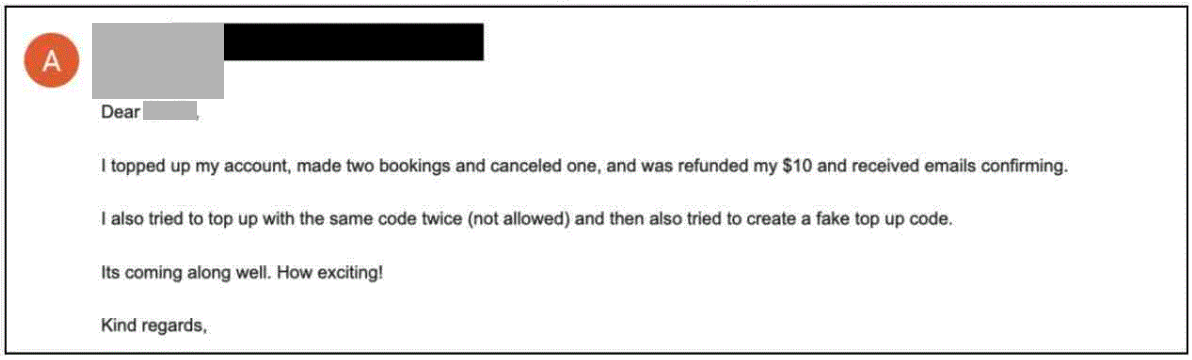


If the user attempts to redeem a topup code that has already been redeemed before, they are shown a specific error message, to avoid confusion.



I added many features to my topup code page that make sure that using it is as error-resistant as possible. An example of one of these features I added is checking that the length of the code is correct.

I then asked my client to test out making a booking, now that the basic components of what is needed for the bare minimum product were present.



She was able to successfully use the new features, and approved of how they had been implemented.

Hardware

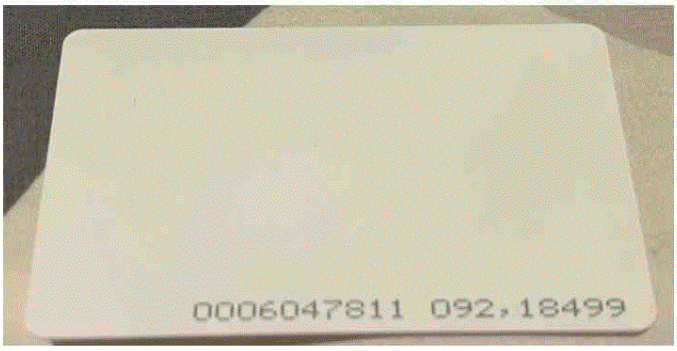
Once my implementation of the software had reached the minimum viable product stage, I began to implement a hardware component. I intended to provide a proof-of-concept for the hardware that could be professionally implemented using my design as a base.

I knew that in order to create this prototype I would need some sort of processor that could use GPIO pins to control other devices. This device would also need to be able to connect to the school's WiFi. I chose the Raspberry Pi Zero 2 W because I had experience with it before, and because of this I knew that it would work with the school's WiFi (which tends to be very finicky with which devices it will allow to connect), and because of its low cost. It could run any type of code that works on linux, but I chose to use python to implement this component. The majority of the code would be run on the server, and this processor's only job was to read the keypad and RFID input, forward them to the server, then open the door if the server says the reading was successful. This meant that I didn't need to worry about choosing the perfect processor for this prototype, because any code I wrote could easily be ported over to another programming language or processor.

I chose Raspberry Pi OS Lite (as opposed to Full) as my operating system because of the limited processing power of the Zero 2 W, so that I wouldn't need to wait minutes for it to boot up into a GUI. My first step was connecting over SSH from my laptop, generating an RSA key-pair, then disabling password authentication. This was far more secure, because it vastly decreases the chances of someone guessing (or brute-forcing) the correct authentication method.

When I purchased the hardware I would need I decided to go with 13.56MHz RFID hardware. I chose this because of its low pricing and easy availability, meaning that any cards needed would be readily available.

However, once I had plugged in all the cables, I was having inconsistent results. Sometimes the reader would not provide any responses, and then after a restart it would work, then after another restart it would break. Because of this I started to investigate other options.



I discovered that there was a similar product with a key difference. RFID 125KHz cards are similar, but they come from the factory with a code that cannot be changed, and is printed on the card.

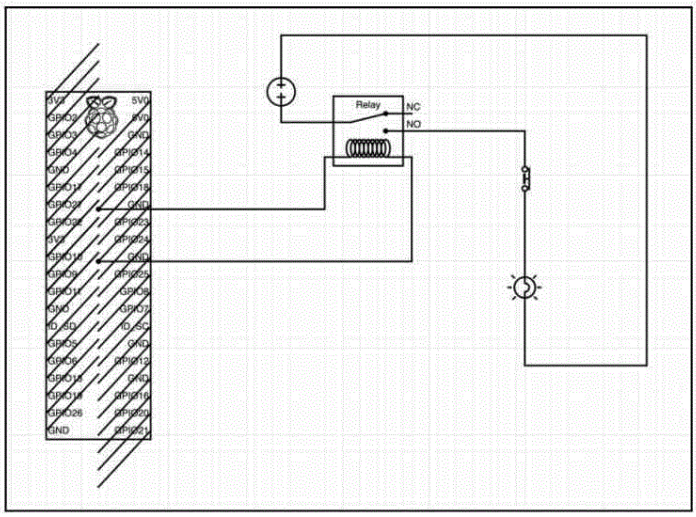
This would solve one of the main problems that I was unsure of how to solve: how would new cards get enrolled onto the system? I knew that the cards would need to be written to, but I wasn't sure how I would be able to implement this in a way that could be easily done without me being present to explain how to write the random value to the card. I investigated using a Flutter app on an Android phone, but I wasn't able to find any phone with the ability to write to RFID cards. If the value was written on the card, then someone would be able to enrol the card using an app that just uses a TextField to ask for the card's value.

I then would need some way of locking the door. I investigated some different kinds of locks. Two main types were options:

	Electromagnetic	Bolt-style
Pros	- If power went out, the door would be unlocked (in this situation this is a pro because the risk of someone	- Some are able to disable themselves if the power goes out - Relatively weak

	being trapped inside is much more important than preventing vandalism/etc) - Strong - If the lock is overpowered the lock remains functional	
Cons	- Takes up space in the doorway (unfortunately the doorway is not particularly tall)	If the lock is overpowered, it breaks

The cricket committee emphasised the importance of not trapping someone inside during a power outage, which is not possible with an electromagnetic lock. Some bolt-style locks in theory wouldn't have this problem, but rely on moving parts, so it is possible (if unlikely) that the lock could get jammed in the locked state. With an electromagnetic lock, there is no way for the lock to get stuck on when it is not receiving any power. Also, the lock not being breakable when it is forced open was seen as a positive. The committee chose to go with an electromagnetic lock.



I then drew up a circuit diagram for how I would control this lock using the raspberry pi (the light bulb is representing the lock). The lock ran at 12V, so I wouldn't be able to power it directly from the Raspberry Pi. To control it, I needed to use a relay. I decided that I would prefer for my proof-of-concept that there was zero possibility of the exit button failing to work, so instead of having it control the relay through software, I decided to have a normally-closed button wired in series with the lock. This means that when the button is pressed, the lock loses power and there are no software bugs that could get in the way of this. Both I and the committee agreed that the number 1 priority of the outcome should be not getting anyone locked inside the cricket nets. This introduces a vulnerability where someone could enter the cricket nets, then use tape to hold the button down and the door would stay unlocked. We decided that this risk was worth it to ensure people's safety. By drawing up a plan, it was much easier to understand what needed to go where once I began plugging in all the components.

I decided that because the lock used a surprisingly small amount of electricity, the best way to power it would be from a USB step up cable. I bought a cable that plugs into any regular 5V usb port, and converts it to 12V. While a 12V DC power supply

would have been usable, it would have been much larger than necessary, and potentially could have introduced a higher level of safety risks than the USB adapter.

I decided that for the prototype lock I wouldn't need a strong one, so I decided to get the cheapest one on TradeMe that would ship in less than a week. Once it arrived at my house, I was surprised at its strength. It could easily hold up its own weight, and I was unable to force it apart.

One important aspect that I hadn't considered is how this lock would be mounted to a door frame. Fortunately, this one had a metal plate that separated from the rest of the lock with screw holes in it. This made it easy to mount because the metal plate could be screwed into the wall, then the lock would be attached to the metal plate using an allen key.

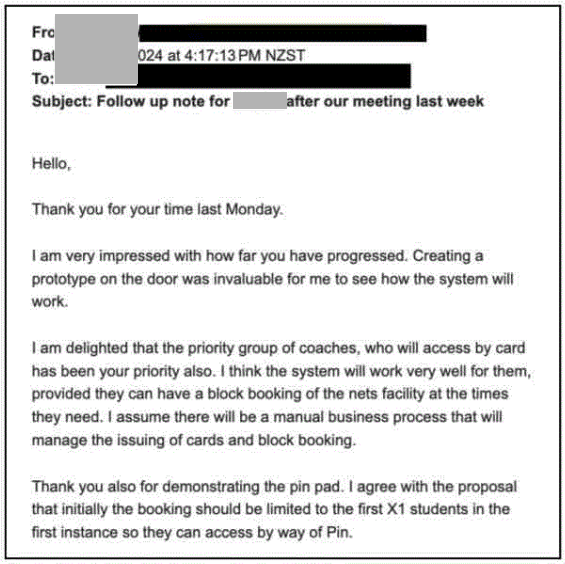
When I wired up the relay (at first it didn't work because I plugged the wires into the wrong pins, but then) it was working. I was able to control the relay with some simple python code.

```
def unlock_door():
    GPIO.output(14, 0) #GPIO 14 set to off
    time.sleep(10)
    GPIO.output(14, 1) #GPIO 14 set to on
```

I then connected a keypad, which also used the GPIO pins. I chose this membrane keypad as a temporary solution to be used while my hardware outcome was still a prototype. However, even in the case of using the outcome as a prototype, I found this keypad to be unreliable, where sometimes it wouldn't be able to read any input, but then later after nothing changed it would be able to. Because this was just a prototype, I decided the best solution to demonstrate the booking system would be to simply switch to using a USB keyboard. This also allowed me to demonstrate the hardware component of my solution in the real world location where it would be used, without needing to damage any walls. By temporarily using a wireless keyboard during the demonstration I could

demonstrate how the solution could allow for the input of numbers, but without needing to have a hole in the wall for a wire to go through. I also had much more reliable inputs with the keyboard, so I decided to switch to it.

Once I realised that I would rely on USB for the keypad, I investigated using USB for the RFID reader. This would have made the input more reliable. While I did switch over to USB for my final prototype outcome, due to an unexpected loss of equipment, I did not have a USB reader on hand in time for recording the demonstration of my outcome, so the video shows a GPIO pin based reader.



Fro [redacted]
Dat 024 at 4:17:13 PM NZST
To: [redacted]
Subject: Follow up note for [redacted] after our meeting last week

Hello,

Thank you for your time last Monday.

I am very impressed with how far you have progressed. Creating a prototype on the door was invaluable for me to see how the system will work.

I am delighted that the priority group of coaches, who will access by card has been your priority also. I think the system will work very well for them, provided they can have a block booking of the nets facility at the times they need. I assume there will be a manual business process that will manage the issuing of cards and block booking.

Thank you also for demonstrating the pin pad. I agree with the proposal that initially the booking should be limited to the first X1 students in the first instance so they can access by way of Pin.

Once I had the RFID reader, keypad, and lock all connected, I showed my client all the components of the hardware working together. She was satisfied with how all the components had come together.

Part 4 - Improving the MVP

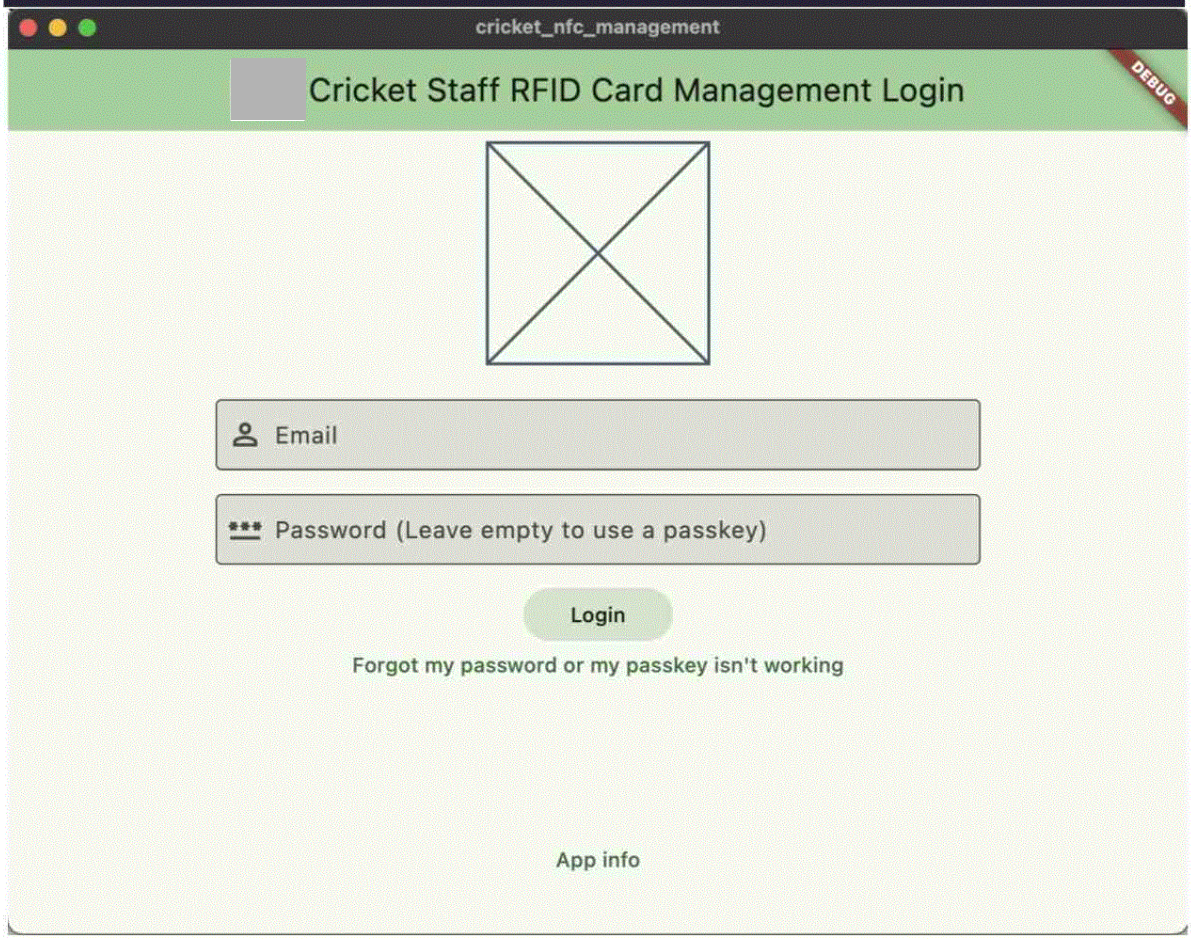
Now that I had the basic features ready, it was time for me to make my minimum viable product into a usable outcome. The problem that I decided to solve first was that there was no way to generate topup codes or activate new RFID cards, except through the PocketBase admin UI, which is doable for me but is intimidating to non-technical users. I decided that I would need to create a simple UI that would allow designated users to activate RFID cards.

I started by making a new Flutter project that connected to the same pocketbase instance. While I had originally considered making the users who were able to edit RFID cards PocketBase admins, I realised that this would be a bad idea because while it would reduce some code that I would need to write, it would also mean that those users have complete control over the PocketBase instance. This very much does not follow the principle of only giving each user as much authority as is necessary.

I decided to create a new project instead of adding a new tab to the already existing app because Flutter Web uses only one main.dart.js file for every page of your application. The app cannot begin rendering until this file has been transferred, and the more pages you have the longer it takes for the app to begin rendering. Because the majority of users will never be able to use the RFID management, it makes no sense for every Web user to have to download the code to run and render these pages.

To differentiate the applications easily I decided to set the colour scheme to

```
colorScheme: ColorScheme.fromSeed(seedColor: Colors.green),
```



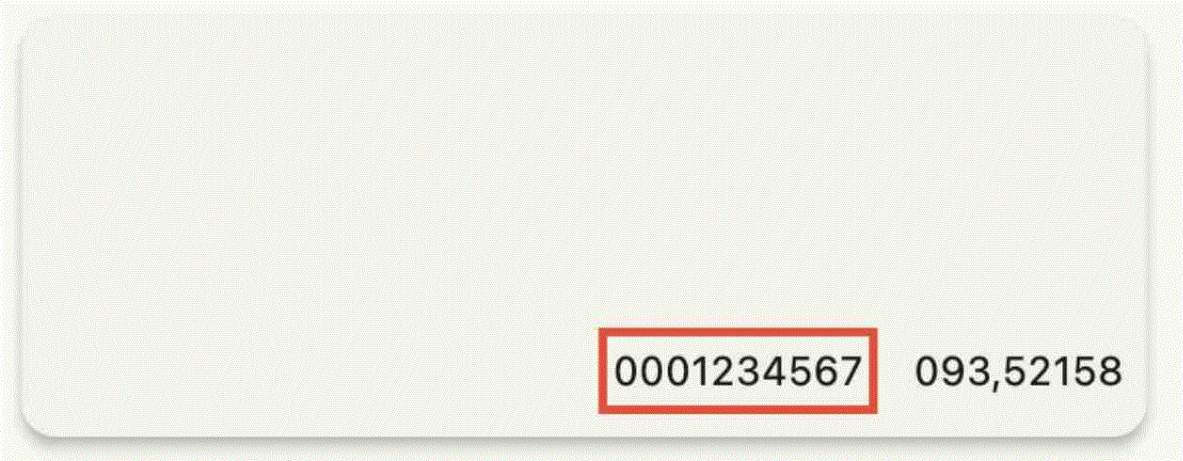
I decided that for this app I would optimise for desktop use, so I decided to use a left side navigation bar. Creating the main app was the first time that I had used a NavigationBar in Flutter, so this time I knew that I would need to create a non-visible widget that handles the layout of the NavigationBar and the selected page. Because there would be an ample amount of screen space, I decided to leave the labels for each tab enabled. Because there would be no need to switch between tabs while saving state, I decided to not use up the unnecessary resources by keeping every page loaded.

Instead of creating custom API endpoints to handle activation and deactivation of cards, I used the default PocketBase create and delete endpoints.

To securely do this I enabled the API for the rfid_cards collection, but restricted it to only accounts with the `allowed_to_edit_rfid` (or read for some rules) permission set to true. This meant that nobody with a regular account would be able to create their own fraudulent rfid card that the system thinks is valid.

I then started work on the UI for this app. I reused the same login page, but because the main.dart file has the colour scheme seed set to green, it has green accent colours.

I then made the code for activating an RFID card. I knew that because the cards have two numbers printed on them, I would need to decrease the ambiguity of the phrase ‘card number’, so I decided to create an example card and surround the number that is important with a red outline. I then made the hint text show the extra instructions.

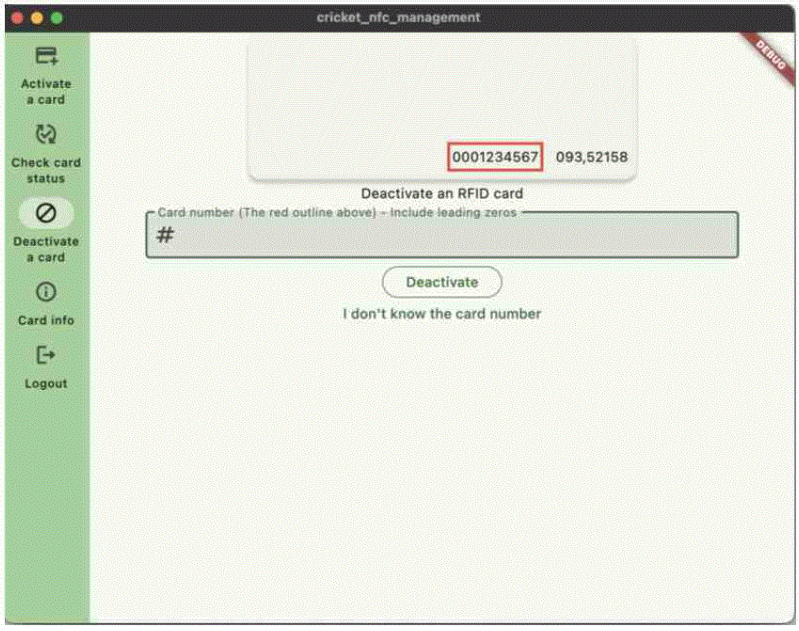



```
if (card_data == "" || card_owner == "") return;
if (loading) return;
setState(() {
 loading = true;
});
try {
```

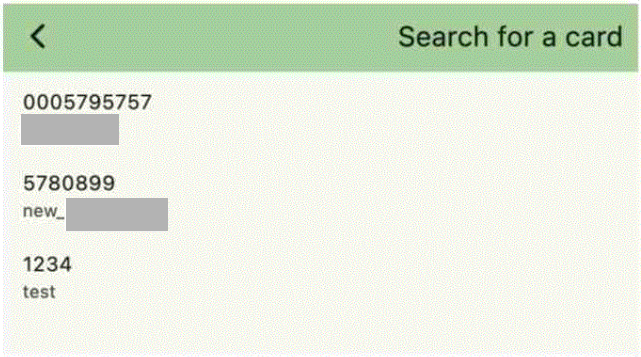
```
await pb.collection('rfid_cards').create(body: {
 "card_holder": card_owner,
 "card_data": card_data
});
ScaffoldMessenger.of(context).showSnackBar(SnackBar(content: Text("Successfully
activated RFID Card")));
Navigator.pushAndRemoveUntil(context, MaterialPageRoute(builder: (context)=>
PagesLayout()), (route) => false);
} catch (e) {
 setState(() {
 loading = false;
});
 showDialog(context: context, builder: (context) => ErrorDialog(error: e));
}
```

I then simply send the card_owner and card_data from the textboxes to pocketbase (assuming both do not ==”” and the button has already been pressed, which would already make loading==true).

I decided to have the card_holder field act as an information field which can be used in case a card needs to be deactivated but the card number isn’t known, so the card_holder’s name can be looked up instead.

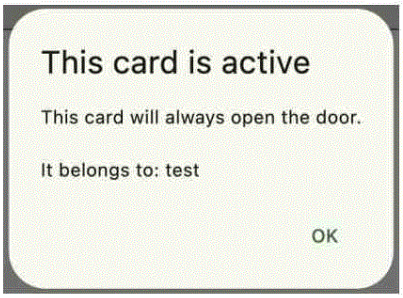


I then made the deactivate page, which either lets someone type in the card number (if they are holding the card they want to deactivate), or if they don’t know this they can press “I don’t know the card number” which will take them to a list of every possible card that has been activated, with the names of each card_holder.



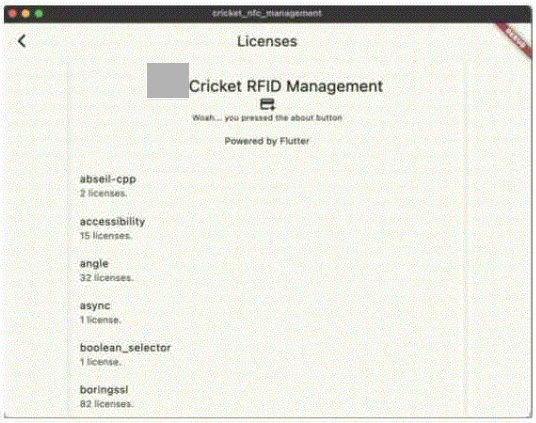
Each of these sections of the scrollable list is a button, which when pressed will return to the previous page, but fill in the text box with the number selected. This then uses the default pocketbase delete API, which

logs the request (including the IP address, which could be used to investigate malicious usage)



I then made the page to check the status of a card. It will either show this formatted response to the input if the button is pressed normally, but later I updated this so that if the button is long pressed, a request is sent to the same API endpoint that the door's Raspberry Pi will use to check if a card is valid. This was extremely useful for debugging whether the door was causing problems or the card was invalid.

Lastly I added the `showAboutDialog()` function to a button on the login page. This is important because for me to be able to use Flutter there were many people that gave their time towards the dependencies of the project who deserve to be credited in the open source licences.



This built-in Flutter function opens a page which shows all the open source licences used to build a flutter app. This page automatically adds all the dependencies my app used. I added this button to all of the separate apps that I developed in Flutter throughout the project.

I then asked the cricket coaches and admins for feedback on what they felt that the app was missing for them. If they didn't feel happy with booking timeslots on the app I was making, the app would be unusable because it wouldn't truly reflect the times that the nets are available or not. Their feedback was the following:

- They need a way of repeatedly booking the same timeslot week after week.
- There needed to be some way of designating an account as [redacted] Staff without my involvement

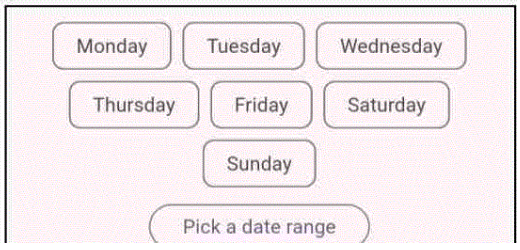
- There should be some way of viewing statistics for who entered the building, and how many times people entered the building
- There should be a better way of viewing the bookings held for people who make many bookings

I decided to implement these suggestions by creating a new Flutter app that would be accessible from a different URL. I began this project by porting over the same login screen (including the code for passkeys - which is important because the same backend - therefore authentication - is shared between UIs, so if I had not allowed passkey login but someone signed up with it, they would have trouble signing in).
I decided to use purple as the base colour for this app because it was distinct from green and blue. While I originally considered merging the RFID card management app with this new one, I decided against it because not everyone who had the ability to use the Staff UI would be able to create new permanently valid RFID cards.

I started with the ability to make topup codes. I had already created an API endpoint which generates a topup code from a random number + the checksum, so I called this API endpoint over HTTPS to ensure that this newly generated code wouldn't be discoverable by an unauthorised person looking at internet traffic. I then displayed this amount in a popup box.

I originally thought that this would be the final endpoint for this page, where it simply asks for an amount for the code to be worth, then shows the new code. However, I had neglected a key usability issue with this page. Because of how Flutter Web works, by default text on the screen is not selectable like with a regular web page. I had been typing out the code manually, but obviously most people would not be typing out their topup codes manually - they will try to select it then copy and paste it. This was an important issue that I found during user testing. The solution was to wrap the Text widget in a SelectableText widget. A simple solution that greatly improved the usability and polish of this page.

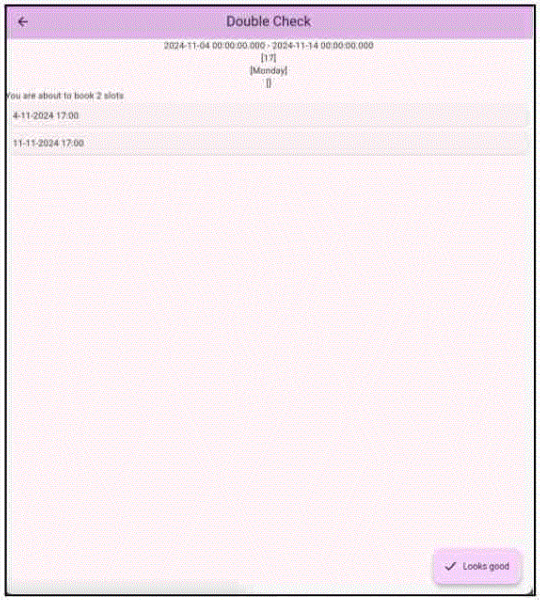
I then began work on solving the problem of recurring bookings. I started with a page that lets users select which timeslots they want to book throughout a week. Because school runs on a weekly basis, I made the assumption that [redacted] Staff would have recurring bookings each week. I checked that this was correct and the Cricket Staff said it was. Since the timeslots available within the app can be changed within PocketBase I needed to wrap this page with a FutureBuilder that first asks PocketBase for some data before allowing the page to be rendered. In this case it asks what is all the data in the `timeslots` collection. I then created some Wrapped



SelectableChips with the weekdays and available timeslots. This meant that the UI would respond to the screen width and take up as much width as is possible before it needs to make a new line. I did this to avoid needing to scroll when there is available space on the screen. It comes at the tradeoff of the separate options being less immediately differentiable when the screen isn't wide enough, but I made this tradeoff in order to have everything visible at the same time on any screen size.

(the 'Add a co-coach button is in these screenshots because I didn't take screenshots of this page before adding the button)

Instead of re-inventing the wheel, I just used Flutter's built in DateRangePicker, which provides a simple UI to pick a date by just clicking on the start date, then the end date, and it shows a visualisation of what you selected.



I then showed a confirmation screen with every timeslot that's about to be booked.

To book these timeslots I had to do something unusual. Because PocketBase doesn't support batch requests (before v0.23) I needed to spawn requests to the database and display their status. To accomplish this, I originally just used some code that looked like this in a for loop that goes through every single timeslot to book:

```
await pb.send(method: "POST",
"/api/shc/admin_make_a_booking/${timeslot.toIso8601String()}/${DateTime(timeslot.year, timeslot.month, timeslot.day,
timeslot.hour+1).toIso8601String()}/$account_id/");
```

This did work on localhost. However once I deployed this to a cloud hosting platform, I started getting 429 Errors (rate limiting). Because of this I knew I would need to artificially slow down the booking process. This shouldn't be a big deal because as long as the UI updates in real time it will feel like the bookings are going fast enough.

I updated my function that books timeslots to first

```
await Future.delayed(Duration(milliseconds: (100 * index).round()));
```

Wait an amount of time that varies according to the number of request this is, then make the same request to the server.

Once the request was successful, it sets the stateList[index] to

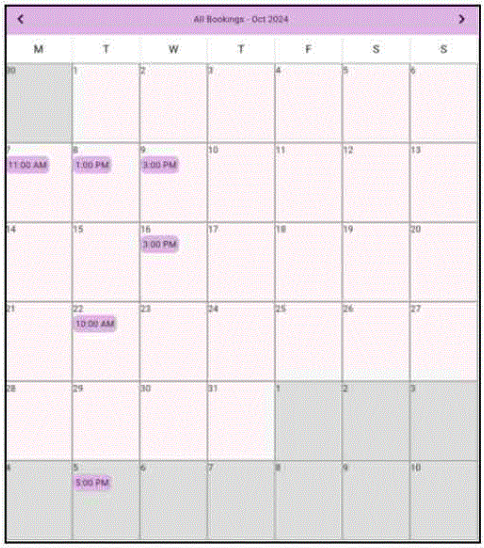
```
{"textColour": Colors.white, "backgroundColour": Colors.green};
```

If the request was successful or

```
{"textColour": Colors.white, "backgroundColour": Colors.red, "error": e};
```

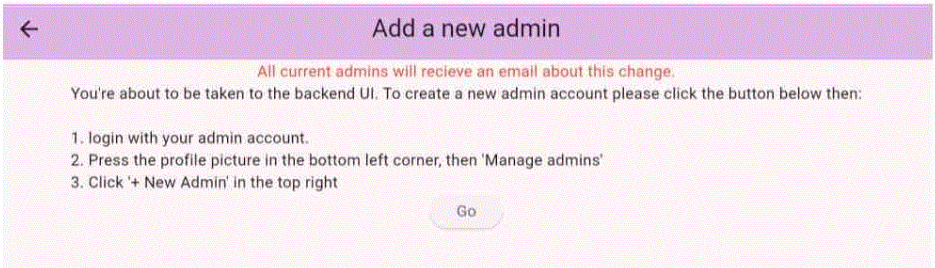
If it was unsuccessful. By updating the stateList I was able to update the UI after each request, making the app feel faster than just having one spinning loading icon.

This would mean that the user can see as soon as it happened if a booking failed (eg if someone else had booked it first). These times start with a grey background then turn to green as they get booked.



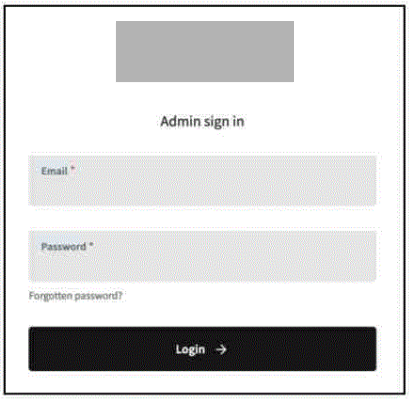
I then addressed the feedback about wanting to see all bookings laid out in a calendar view. I copied the code from the main user app that shows the calendar, but adjusted it so that it shows every timeslot, because I can assume that this user is likely on a laptop/desktop computer with a screen large enough to see all this data. When making this calendar page I decided to avoid one of my personal biggest problems with an app that I use: Kamar for iOS. For some reason, when you first open the Kamar app and look at the calendar view, it will not have finished loading the calendar events, but will tell you an outdated list of events before it has finished loading. This is worse than showing a loading spinner because it showed incorrect data. To avoid this I showed a loading spinner until all the data had been loaded in. While this was less visually appealing, it made the UI feel so much more polished and usable.

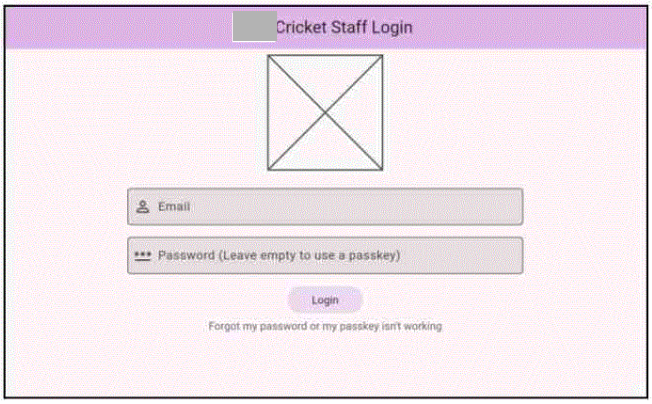
I duplicated this page but set it to only show bookings made by whoever is logged in to address the fourth piece of feedback I received (there should be a way to see all your bookings easily for staff who will make lots of bookings).



I then addressed the feedback that there would need to be a way to add

more accounts designated as [redacted] Staff after I had left. I decided to solve this problem using documentation of the existing PocketBase admin backend. This page has instructions for making a new admin account. I investigated adding this UI into the frontend app, but because creating a new Admin account requires already being an Admin account (which has its own authentication system separate from everything else), creating Admin accounts within the app would be unnecessarily complicated. Users would be easily confused between Admin accounts and Cricket Staff accounts. By having a very visually different login page, it would show that they are different things.





From here it was time to get a full scale prototype ready for my client to see working in the real world.

While I was developing features for my app, I made the mistake of running the command `pocketbase update`. I had to edit the source code of pocketbase to integrate passkeys, but running this command downloads the latest version from the official PocketBase github. Because the official PocketBase doesn't have the code needed for passkeys, this overwrote the working version. To fix this error I added a command to main.go:

```
    app.RootCmd.AddCommand(&cobra.Command{
        Use: "update",
        Run: func(cmd *cobra.Command, args []string) {
            log.Fatal("\033[31mNot updating. Updating from here will download a
generic PB executable, which does not include the code necessary for passkeys.
Please use the custom build PB.\033[0m")
        },
    }
```

This means that if I tried to update using `pocketbase update` it would show a red error, preventing me (or anyone else who comes after me) from accidentally breaking passkeys.

During the development of my project, there was a major security issue that was quickly resolved in the JS ecosystem. Many websites relied on the api found at polyfill dot io to handle their polyfills (which allow you to use new JS features on old

browsers). Because JS was being imported from an external website, that external website had complete control over everything that was being shown on the page. Polyfill dot io's domain was taken over by a malicious actor who began to insert nefarious code that redirected users (Prince et al., 2024). The message that I took away was that I should avoid using any externally hosted code.

This was relevant to my outcome because to support passkeys for Flutter Web, the instructions say to add this line to the index.html file.

```
<script
src="https://github.com/corbado/flutter-passkeys/releases/download/2.0.0-dev.1/bund
le.js" type="application/javascript"></script>
```

I originally had inserted this line, but after witnessing the chaos caused by polyfill dot io's malicious code, I decided to alter this line to:

```
<script src="passkeys.js" type="application/javascript" defer></script>
```

And I downloaded the file at the URL, saving it as passkeys.js alongside index.html. This means that whenever the page loads it will ask my server for the passkeys.js file instead of allowing whatever Corbado uploads to that URL to run on my users' browsers.

At this point I asked my client for feedback again, and she said she would prefer that I implement the logging for each time the door is unlocked. To do this I set up a Google Sheet and a new Google Cloud Platform Project. Now whenever the door is unlocked, this code runs:

```
wks = spreadsheet.worksheet("title", "All access attempts")
all_values = wks.get_all_values(include_tailing_empty_rows=False,
include_tailing_empty=False)
print(all_values)

wks.update_value( (len(all_values) +1, 2), sys.argv[2])
wks.update_value( (len(all_values) +1, 1), sys.argv[1])
wks.insert_rows(len(all_values) +2)
```

	A	B
1	Time	Code/card entered
2	7/10 11:39	"047063"
3	7/10 11:40	"047063"
4	7/10 11:44	"047063"
5	7/10 11:44	"047063"
6	7/10 11:45	"047063"
7	8/10 12:41	"242912"
8	8/10 12:42	"242912"
9	8/10 12:44	"242912"
10	8/10 12:45	"242912"
11	8/10 13:27	"242912"
12	8/10 13:29	"242912"
13	8/10 13:55	"573316"
14	8/10 13:56	"573316"
15	8/10 13:57	"573316"
16	8/10 13:59	"573316"
17		

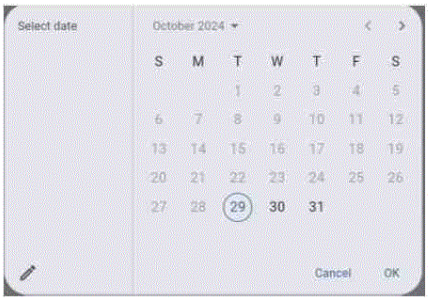
This inserts the time and the code entered when the door was unlocked. This meant that access logs were automatically collected. This is what it looked like after recording my demonstration video. As I had learned with previous projects using Google Sheets, the API will not automatically create new rows, so eventually you will run into errors if you do not create a new row every time you insert data because you will run out of space.

Part 5 - Bringing Everything Together

During this stage of the project I had the most difficult to resolve problem that I hadn't anticipated. Instead of a technical problem, this was a social problem. Installing a lock on the building (even if it isn't permanently installed) would mean that I was changing who had access to the building, and the building owners would need to approve. I had assumed that this would be a simple problem to solve, but I was incorrect. The building management team was unconvinced with my solution, because all they had heard was that a student had developed a project (I don't blame them for not trusting that).

I wanted to show them my prototype in action, but they didn't want to install something until they felt confident in it. We had reached a stalemate. This stalemate lasted for over a month until the HOD for computer science reached an agreement with the building management team: because the lock became completely disabled when power was disconnected, we would install the lock, but leave it unplugged when I wasn't there physically. The old key in a lockbox lock would still be used at all other times. This allowed me to demonstrate my outcome in the physical environment where it would be used.

This was one of my least expected problems that would need to be worked around and lasted for much longer than I expected. However, now that a solution had been reached, I was able to demonstrate my outcome in the physical context where it would be used in the real world.

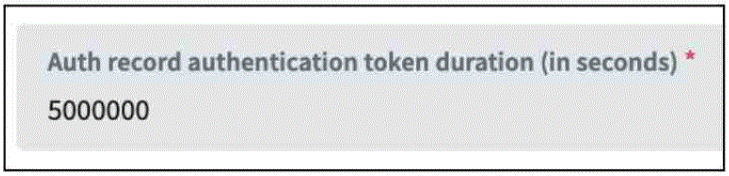


My outcome is fit for purpose as a digital outcome. I have ensured to integrate the accessibility methodology by reducing the amount of text needed to understand the UI wherever possible. I have relied on representations of UIs, such as with the Flutter DatePicker, which mimics a calendar view. Because users are already familiar with how to read a calendar, picking a date is simplistic for the user.

The timeslot system also relies on a UI representation that users are familiar with, especially if they are frequent customers of click and collect groceries.

By integrating passkeys as a login method, I have allowed an innovative security mechanism to be used. Users can login on any device as long as it supports bluetooth (which the majority do), and they then scan a QR code with their phone to connect. I polished this up by allowing for password authentication as a fallback. Because most users will reuse passwords, having passkeys as the default

authentication method adds an increased layer of security at a minimal cost to the convenience of users. I have also considered my users by setting the PocketBase auth token duration to a reasonable amount of time. Because users are likely to use this website infrequently but the website isn't high-security I decided to make the login token last ~60 days. This balances the security required while also ensuring that users don't become complacent with not needing to know their login method so forget what it is by the time they need to log back in. ~60 days is long enough that for the vast majority of bookings will keep the same auth token from when the booking was made to when the booking is used. This shows polish in the user experience.



My solution is unconventional because it is not proprietary in any way. All the parts I used were widely available. This means that if any part breaks in the future, it will be repairable. In the commercial world for access control systems there are fully proprietary systems that must be completely replaced even if only one part has broken.

My passkeys implementation is also innovative because it relies on the browser's implementation of Webauthn. Webauthn is a standard that is often improving, so since my app relies on the OS/browser to handle the specifics, it can have new auth methods added without me needing to specially activate this. An example of this is USB security keys. I had not realised that my outcome already supported using them, and in the future if Webauthn evolves, my outcome will potentially already support the new features implemented.

My outcome solves the problem because I have allowed for a numeric input and RFID card input at the door, which provides multiple methods of authentication at the door.

I have created a UI accessible on any device with a web browser that has been updated in the last 2 years. Firefox 106 (*Supported Platforms | Flutter*, 2024) is the oldest version supported by Flutter , which was released in October 2022. The vast majority of users will own at least one device which supports the app, if their main device doesn't support it. However, because most web browsers automatically update, this is unlikely to be an issue.

My UI allows for users to easily and efficiently create their own bookings, without overcomplicating the experience. Users can securely create their own accounts with

passkeys, and receive updates to their booking (such as a confirmation) to the email that they provide. Because users need to verify their emails, this also gives some level of contactability to each user, for example if there is a report that a user caused damage.

My admin UIs allow for the school staff to edit the data in the system that they need to from their web browser, without needing to interact directly with the admin UI that hasn't been simplified for users. It shows them only what they need to see so that they are not confused.

By integrating Google Sheets, I keep a log of who was given access to the building and when, which can be used to give insights to how the building is being used and when. Google Sheets has powerful graph and data analysis tools, so this will be useful to the cricket admins who need to know how many resources should be invested into maintenance of the cricket nets.

My outcome is fit for purpose because it addresses the requirements that I agreed on with my client at the beginning of the project, on top of the extra design decisions that we added while the project was ongoing.

Part 6 - Conclusion

Client testing

I had continuously asked my client for feedback throughout my development process. This means that once I reached the point where I felt confident in my prototype having reached a 'good enough' stage, my client was already satisfied with my progress. By asking her to use the entirety of the app in front of me, she was able to completely use all of the features. She felt that it was fit for purpose.

I then asked one of the [redacted] Cricket admins to try using their section of the app. While [redacted] was able to evaluate the app for it's understandable-ness, a cricket staff member would be able to evaluate how effective the app would be to use in real life.

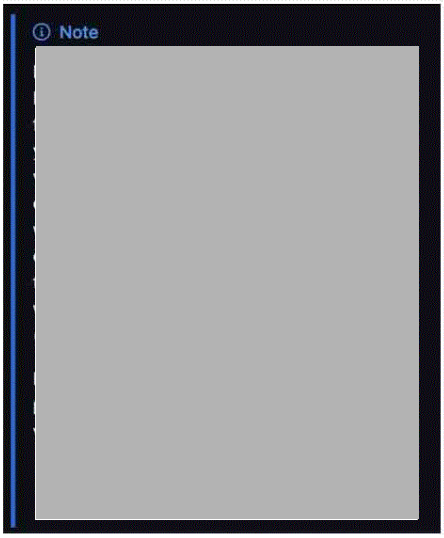
I gave them a login to use, and they were able to book out a section of timeslots and generate topup codes without my involvement. Their feedback was positive of the app, however this didn't mean that they didn't have anything that I should do.

They emphasised that because I am in [redacted] I will need to futureproof as much as possible to ensure that the project can continue to live without my involvement. They said that the first step to this was to create proper documentation. While I agree with them, I did put off creating documentation because to me I know how the app works so its difficult for me to know what parts need to be explained. I decided that I would create a set of manuals for how to use the app. I was glad that the feedback I received was unrelated to any issues or bugs.

To test the outcome with my third stakeholder group ([redacted] students), I asked one of my friends to test out the app. By watching them use the app, I saw that they were concerned that when their phone asked to use FaceID, they thought my app was seeing FaceID. To fix this, I added a message saying that "we cannot see your biometric information" when the passkey is created. Other than that, they were able to create bookings, found the error mitigation on the topup code page useful, and overall thought the app was fit for purpose.

Because there is very little difference between the students and members of the public stakeholder group in terms of how they will use the app, I applied these findings from the students to the members of the public.

Future developments

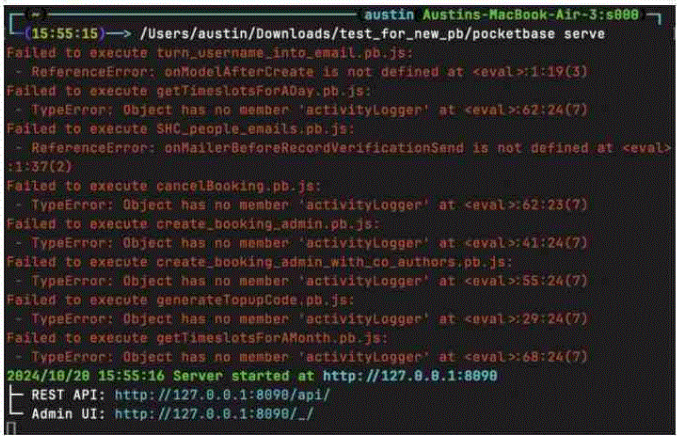


In between when I started developing my outcome and now, PocketBase has introduced the v0.23.0-rc releases. While these add useful new features to PocketBase, they have breaking changes inside the JSVM used to run my JS hooks and custom API endpoints. As this notice from the official PocketBase github says, upgrading isn't necessary just yet because any security vulnerabilities will be backported to v0.22.0 pocketbase until early 2025. Because of this, if I want to keep relying on the security added by PocketBase's continually updating nature I will need to update my code in the JSVM to support the latest version. Currently v0.23

is in prerelease, so while I could update my code to run on the new version, there could be more breaking changes made in the next few days. If I decide to update my pocketbase to v0.23 I will need to update my code to use the new function names for all my custom API endpoints. The database has no changes when upgrading, so all data will be interchangeable between versions. To upgrade my JSVM code to work with the new version, the process should be simple. Very few functions have been deprecated (none that I use), and instead the vast majority of functions have been renamed to remove $app.dao().

`$app.dao().recordQuery(collectionModelOrIdentifier)`	`$app.recordQuery(collectionModelOrIdentifier)`
`$app.dao().findRecordById(collectionNameOrId, recordId, [optFilters...])`	`$app.findRecordById(collectionModelOrIdentifier, recordId, [optFilters...])`
`$app.dao().findRecordsByIds(collectionNameOrId, recordIds, [optFilters...])`	`$app.findRecordsByIds(collectionModelOrIdentifier, recordIds, [optFilters...])`
`$app.dao().findRecordsByExpr(collectionNameOrId, [exprs...])`	`$app.findAllRecords(collectionModelOrIdentifier, [exprs...])`

When I attempted to use my current code with the new PB version, this is the error messages that I saw:



While this looks like quite a few errors, these are all errors arising from names not existing because the names don't exist, because they have been renamed with this version of PB.

The exception to all the functions I relied on just being renamed is `$apis.activityLogger(app)`, because now all requests are added to the logs section by default.

```
$apis.activityLogger(app)
```

```
// No longer needed because it is registered by default for
all routes.
// If you want to disable the log of successful route
execution you can attach the $apis.skipSuccessActivityLog()
middleware.
```

This means that all the error messages that say `TypeError: Object has no member "activityLogger"` will all be solved by simply removing the middleware function that tells PB to log the request.

In the future I will upgrade to the latest version of PocketBase in order to benefit from the security patches implemented by PocketBase, because it is difficult for me to maintain this project with the latest security patches even once I have moved onto different projects.

By using the Material UI when building my frontend, my UI can be easily updated to have the latest UI improvements. By changing just one line in `main.dart` I can make my app revert to Material2, which looks like this. Material 3 embraced using tonal colours, and rounded corners that were previously more square. Because the current state of what looks modern and outdated is constantly changing, when Material 4 is released it is possible that I will be able to update it with just 1 line of code changed (`useMaterial4: true`).

This means that I will be able to keep the UI looking fresh and not outdated with minimal effort required from me.

When I was nearing the end of my project, I was approached by a member of the basketball staff at [redacted] who asked me if it would be possible to adapt my project for the basketball courts. There is nothing stopping this, so in the future if my project is received well in cricket, I may adapt it to work for other facilities at [redacted].

Hardware

While I am satisfied that my hardware outcome demonstrates the functionality of both the hardware and software, it is clear that the hardware is only a prototype, and someone with more experience will need to wire up the hardware in a more permanent manner. In the future, I am hoping that the Cricket Committee is satisfied enough with the prototype that they have seen demonstrated in order to invest in having an electrician install the hardware properly.

Because PocketBase is not platform dependent, it can be run on whatever platform has the cheapest or highest quality server hosting. This means that any cost of running it is determined by whichever platform offers the most competitive rates, or it could even be self-hosted.

What ifs

If I had decided to use a proprietary backend as a service solution, I would have been locked into whatever developments happened on that platform. If the platform decided to shut down, I would have had to rewrite code. However, because I used PocketBase my code will work indefinitely.

If I had decided to go with a Google Forms style outcome, I would have finished much earlier in the year, but my outcome wouldn't have been as fit for purpose as my final outcome because it wouldn't encourage people who are unrelated to [redacted] to book to use the nets.

If I had decided to use something like React to develop my app with a web-first design, I likely would have created a better user experience, however this would have taken me more time and I likely would have introduced more bugs by using a language and framework that I had no prior knowledge in. Flutter Web is usable but not equivalent to something that was built for the web in the first place.

If I had access to an Apple Developer Account I would have been able to list my app on the App Store for people to install. This would also have allowed me to enable access to the NFC reader, which for some reason is only available to paid developers even on test apps. Because I didn't have one of these accounts, I didn't investigate the possibility of people using their phone to authenticate with the door. This could have been even more convenient than an RFID tag for staff members, because staff members are already carrying their phones with them.

If I could restart this project using the knowledge I have gained, I probably would have gone with a microcontroller based hardware solution instead of a microprocessor based one. The OS in the microprocessor added unnecessary complexity and slowed down the operation of my outcome. After power was supplied, it took at least 20 seconds for the OS to boot up to a state where it is able to accept input. While this is good enough for my outcome, not needing to wait for an OS to boot up would be preferable. I may choose to implement this myself as a potential future development.

I will need to find out whose job it will be to distribute topup codes. The school has made it clear to me that they will be handling payments, so I just need to make sure whoever this person is has access to generate new topup codes.

Notes for the Assessor

My GitHub repository can be found at https://github.com/

A demonstration video for my project has been submitted with this file.

References

Agrawal, H. (2024, September 20). *CVE-2024-45489 Response*. Arc Browser. Retrieved October 20, 2024, from https://arc.net/blog/CVE-2024-45489-incident-response

Prince, M., Graham-Cumming, J., & Tremante, M. (2024, June 26). *Automatically replacing polyfill.io links with Cloudflare's mirror for a safer Internet*. The Cloudflare Blog. Retrieved October 29, 2024, from https://blog.cloudflare.com/automatically-replacing-polyfill-io-links-with-cloudflares-mirror-for-a-safer-internet/

Stewart, J. M. (2018, June 26). *The Three Types of Multi-Factor Authentication(MFA)*. Global Knowledge. Retrieved October 14, 2024, from https://www.globalknowledge.com/us-en/resources/resource-library/articles/the-three-types-of-multi-factor-authentication-mfa/

Sugue, M. (2024, July 23). *Prevalence of Color Blindness: Global and Regional Statistics*. Vision Center. Retrieved October 28, 2024, from https://www.visioncenter.org/resources/color-blind-statistics/

Supported platforms | Flutter. (2024, October 21). Flutter Documentation. Retrieved October 29, 2024, from https://docs.flutter.dev/reference/supported-platforms

Outstanding Scholarship

Subject: Technology

Standard: 93601

Total score: 21

	Score	Marker commentary
Synthesis and integration	07	A sophisticated prototype was developed to meet an authentic issue. The requirements and attributes are well researched to establish the parameters of the brief in the broadest sense. The project flows easily, demonstrating excellent design thinking in the technological process. A well thought out and developed prototype is underpinned by complex knowledge and a level of refinement that is the result of highly competent technological practice. The efficient testing and knowledge that has been integrated within the project demonstrated the required elements of synthesis, polish, and optimization that led to a prototype that is seen as a viable solution to the issue. The development of the booking system is innovative and original, enabling the candidate to demonstrate a very high level of skills and creativity.
Justification	07	A combination of very well written evidence and imagery provides visual connections to support the ongoing commentary. A short video clearly demonstrates how the various aspects of the prototype work in practice. Evidence provided throughout the report is always relevant and with limited repetition of content. A high level of technological modelling is evident throughout, which has enabled the candidate's technological practice to be responsive to the booking system's development. The candidate has been discerning with their work to ensure the scholarship criteria has been met. The candidate communicates convincingly and systematically which supports the technological practice of their prototype. This included outlining how complexities and constraints of the developed prototype were resolved and reviewed throughout the project. The result of this allowed the candidate to justify both their technological practice, decision-making, and the effectiveness of the prototype. The candidate has provided a clear and convincing narrative that further enhances our perception of the product.
Critical reflection	07	In this report, it is evident that the candidate has critically reflected on their own technological practice. The candidate selected stakeholders who were genuinely invested in the success of the outcome. The stakeholders were consulted at key moments to provide important feedback.

		They then critically reflected on the information gained to inform and guide their practice, resulting in a high level of skilled execution. There is evidence of how the candidate made their decisions and progressed their ideas, with independence, to ensure the product was fit for purpose.