TOP SCHOLAR EXEMPLAR

Scholarship 2022

Technology

SUBlime Substitution Manager

Video: https://www.youtube.com/watch?v=HVfE5bsWX5A

Download: Available on App Apple Store and Google Play Store

Introduction

The benefits of children playing sport are manyfold. One of the key recommendations of the World Health Organization's Commission on Ending Childhood Obesity was the promotion of physical activity[1]. Physical activity prevents early onset of childhood obesity and related negative health outcomes. Sport is also protective of mental health; being an equal part of an inclusive team helps build resilience and esteem in a child.

Identified Problem

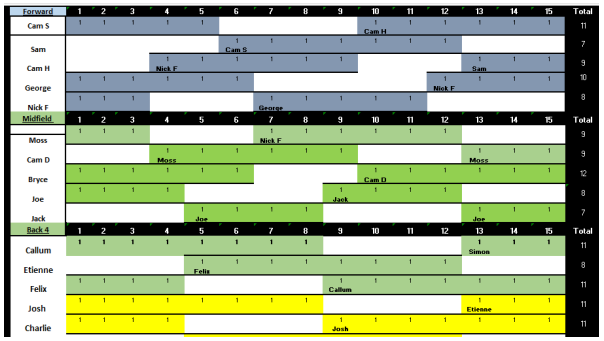
Unequal gametime in junior sports is a major issue that continues to fail to be addressed. Young players who get less gametime than their teammates are made to consider their worth within the team, breeding feelings of insecurity and inadequacy. This may evoke feelings of disinterest and boredom as they begin to associate a game of sport with sitting on the bench for most of the match.

Unequal gametime also erodes team culture. Players may lack confidence when they play as they fear one minor stuff up could get them benched for the rest of the match. Feelings of discontent also develop between players and coaches, as players believe that the coach doesn't value them as a member of the team. These feelings are extrapolated by parents who rightfully support their kid and question the coach's judgement, worsening team culture as parents and coaches become further divided. This makes parents less likely to encourage their kids to play sport as they feel their time and financial commitment of having their child in a sports team is not being valued. If a coach aims to pursue fair and equal gametime this requires so much micromanaging, that it is detrimental to their ability to coach and develop skills of young players.

The end consequence of this is kids simply don't enjoy playing sport. This is reflected by 75% of kids dropping out of sport by age 13[2]. The ramifications of this are major. Children stop playing sports into the future and miss out on a wide range of benefits. Physically, they participate in less exercise resulting in higher chances of diabetes and other related diseases. Socially and mentally, they miss out on making lifelong friends and being valued as a part of a team. Also, without young players continuing to play sport, sport breaks down at a grassroot level as teams can no longer be fielded, to the detriment of wider communities.

Solution to Identified Problem

The idea for a solution to this problem came to me at the start of this year. I was reaching out to friends and family for potential ideas for digital technology projects. An idea was suggested by my dad for a digitalized solution to the 'subsheets' he used for the hockey team he coached. He explained how every week he would create a subsheet on Excel – that would show substitution orders of his game - then print it out and put it on the dugout wall for players to look at. This process of creating a subsheet took a very long time and therefore he believed he could benefit from having it digitalized, allowing him to make them and display them on his iPad.



Excel made subsheets. The subsheet shows when each player is on the pitch and what position their playing at. Total gametime is on the right. side

This discussion served as the starting point for the development of this project. I saw the potential in having digitalized subsheets but believed their best application would not lie in competitive sports grades but instead in junior and social sports grades. This gave birth to the idea of *SUBlime*. *SUBlime* would be a substitution management mobile app that allowed coaches to create digitalized subsheets. The allocation of time to players on these subsheets would be collated into gametime allocation statistics for both individual games and across seasons. These statistics would be presented in a way to be easily screenshotted so they could be sent out to parents, serving a dual purpose of forcing coaches to allocate gametime equally and reducing parent-coach tension as they know that their kids are getting equal gametime. Alongside this, a game overview feature would also exist in this app. This feature would be used during the game having a

[1] https://apps.who.int/iris/bitstream/handle/10665/204176/9789241510066_eng.pdf
[2] https://www.youthsportspsychology.com/youth_sports_psychology_blog/when-kids-drop-out-of-sports-because-of-little-playing-time/

countdown to upcoming subs. This would provide an additional benefit alongside encouraging equal gametime, in the form of the promoting self-management in junior sports players as they could now see when their sub is occurring allowing them to carry it out independently of the coach. This would also allow benched players to feel involved without being on the pitch as they need to actively check for when they next sub on. Additionally, this also allows coaches to devote their focus towards coaching as they don't have to micromanage subs.

Preproduction

With a clear problem identified and a practical solution found I now moved to pre-production. Pre-production is the planning phase of a project. I viewed it as essential to put aside substantial time towards preproduction, so I could seamlessly move into the development of *SUBlime* with a clear creative vision and milestones laid out.

I broke preproduction into three phases

1. Refine and finalize the design of the project – through market research, stakeholder consultation and creation of conceptual designs.
2. Identify the needs of the project and select development tools that address these needs.
3. Learn and develop my skillset with any development tools I hadn't used before to prevent time being wasted early in the project due to my incompetence.

Project Timeline

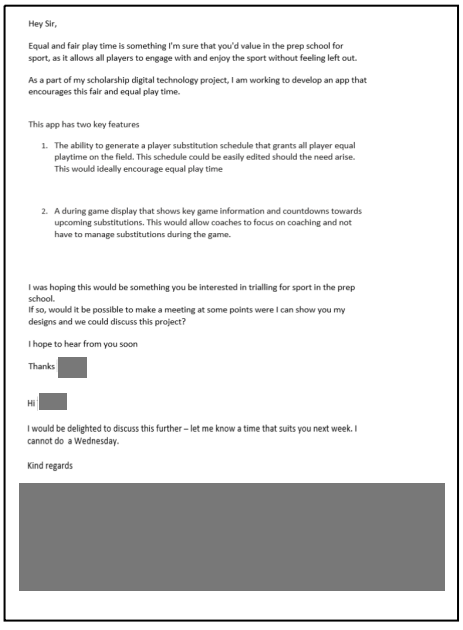
The first task I completed in preproduction was the creation of a project timeline. This timeline would contain key milestones of the project and dates for me to work towards.

Date	Goal
Term 1 Week 3	Begin the preproduction phase of development.
Term 1 Week 8	Finish the design/tool selection process and begin developing my skills with the selected tools.
Term 2 Week 1	Finish Preproduction and begin development of a minimal viable product.
Term 2 Week 8	Finish minimal viable product and carry out first technical test.
Term 3 Week 1	Begin widespread testing with StAC sports teams and if time permits carry out testing with local sports governing bodies.
Term 3 Week 6	Finish the implementation of any feedback as must leave adequate time for the write up.
Term 3 Holidays	Implement any final feedback and submit the app to Google Play Store and Apple App Store

First Meeting with A K

As I saw immense potential in the idea of *SUBlime* I wanted to get as many stakeholders as possible on board to refine and test out the idea. However, I recognized that groups such as local sports governing bodies (Christchurch Netball, Mainland Football etc..) that I wanted to work alongside, would likely only want to try out and discuss the app if it was semi complete. Therefore, I decided to first keep the stakeholder base small then expand it out as development progressed.

The principal stakeholder I decided to work alongside was A K. A K is the Preparatory School sports coordinator at my school. A K oversees the running of all sports codes in the Prep School and therefore, valued the importance of getting juniors engaged in sport. I sent an email to A K with the core idea of *SUBlime* asking to organize a time to meet and discuss. He stated he liked the idea and was keen to meet up and discuss it. In our discussions A K stressed how much he liked the idea as both a parent, coach and sports coordinator who has to constantly deal with parents' complaints about unfair gametime allocation. He stated one of the Prep Schools' sports values was inclusivity and this app would go a long way in ensuring that. He said he'd be willing to arrange testing with coaches from a wide range of sports. Finally, he helped me finalize the idea of the minimum viable product and then suggested we meet up again later in the term when I had some designs completed.



Hey Sir,

Equal and fair play time is something I'm sure that you'd value in the prep school for sport, as it allows all players to engage with and enjoy the sport without feeling left out.

As a part of my scholarship digital technology project, I am working to develop an app that encourages this fair and equal play time.

This app has two key features

1. The ability to generate a player substitution schedule that grants all player equal playtime on the field. This schedule could be easily edited should the need arise. This would ideally encourage equal play time
2. A during game display that shows key game information and countdowns towards upcoming substitutions. This would allow coaches to focus on coaching and not have to manage substitutions during the game.

I was hoping this would be something you be interested in trialling for sport in the prep school.
If so, would it be possible to make a meeting at some points were I can show you my designs and we could discuss this project?

I hope to hear from you soon

Thanks

Hi

I would be delighted to discuss this further – let me know a time that suits you next week. I cannot do a Wednesday.

Kind regards

Finalization of Features in Minimum Viable Product (MVP)

In the past, development projects I have worked on have suffered from scoping issues where I try and add too many features into an app resulting in it never being finished. To combat this for *SUBlime* I decided that from the start of the project I would target a strict MVP and then only when I achieved this add new features.

The four base features of the app I decided on based on with my consultation with A K were:

1) The ability to select a formation for players on the field.
2) The ability to add players to your team and assign them positions.
3) A digitalized subsheet that players can be assigned to. This would also show the gametime per player.
4) A game overview screen that would manage substitution automatically showing a countdown to all the upcoming substitutions.

Further to this, initially I would only develop the app for hockey but once development progressed, I would contact stakeholders of other sports to help me setup the app for that sport.

Market Research on Similar Apps

With an MVP defined I wanted to next carry out market research before continuing with designing. I focused my research of them on two things. The first was whether any apps successfully fulfilled the desired purpose of *SUBlime* – mainly the ability to create subsheets, view allocation of gametime and manage substitutions automatically. The second was whether these apps had any unique and helpful features that I hadn't considered and therefore may consider implementing into my designs. After searching on the App Store, I found a few apps that marketed themselves as sport substitution apps. Out of these sports substitution apps I looked at the three most highly ranked apps on the App Store. These apps were called *SoccerSubstitution*, *CoachAny* and *TeamCoach*.

App One – SoccerSubstitution

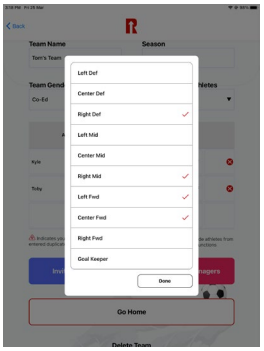
This app was rated the 89th highest on the App Store sports charts, far higher than any of the other sub management apps so I went in with high expectations. The first thing I noted about this app was that it was online only meaning that coaches would not be able to use this app pitch side where they don't have Wi-Fi – this made me consider the importance of *SUBlime* working offline. More importantly in terms of features, this app lacked features I viewed as crucial to my app such as the ability to create game subsheets and overview gametime allocation. However, it had an extremely easy to use and understand player creation system that made it easy to add and remove positions from players. I planned to use this player creation system as the basis of my player creation system. This app also had a game overview screen, which was very ugly, and I couldn't get the timer to start. Additionally, substitution had to be done manually through the coach's actions during the match as opposed to doing them automatically based on a premade subsheet.

App Two – CoachAny

I found CoachAny to be a very stripped back and minimalistic version of what I wanted my app to be. The app had an extensive settings screen which really gave me control as the user to setup the game how I wished. Unfortunately, that was where the positives of the app ended. It lacked the four features I viewed as key to my app. Alongside this, you were unable to assign positions to players in this app resulting in a complete lack of control being provided to the user when it came to who was subbing on for who. The final issue was that the game overview screen was really confusing as to who was subbing on for who and coaches had to manually handle these substitutions during the match as opposed to – how I planned where - the substitutions would be carried out automatically based on a premade subsheet.

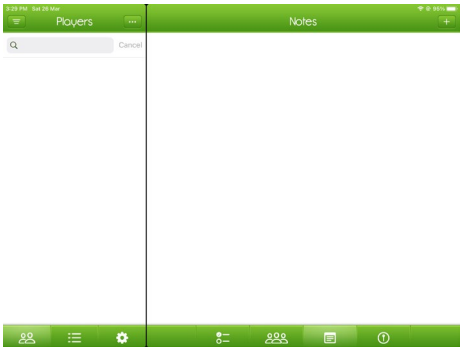
App Three - TeamCoach

To put it lightly this app was awful. The user interface was confusing, and it took me five minutes to figure out how to make a team. In addition to this, the app didn't have the capability to manage substitutions in a game and only provided a visual display of the starting line-up.



App 1 – Assigning position portion of the player creation system I liked.

App 2 – Unclear in the game overview screen that is hard to interpret who is subbing for who. For example, both Toby and Xavier are going on and Kyle and Corin off. But it isn't clear what position they're subbing at and whose swapping.



App 3 – Home screen off app. It was very unclear what to do on this page I tapped around on it for five minutes and still couldn't find anything

Conclusions From Market Research

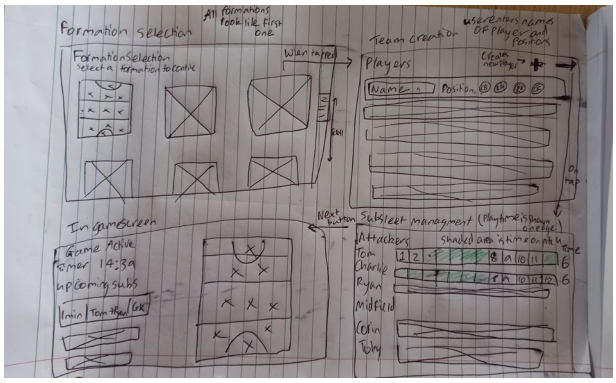
At the end of conducting market research, I was feeling inspired. None of the existing apps resolve the issue I wanted to address. None of them provided the capability to create subsheets, view allocation of gametime and manage substitutions automatically. This showed me that *SUBlime* could stand as unique from others on the app store and resolve an issue that has yet to be addressed by technology. A further weakness I noted in all these apps was that their UX was not very good, meaning I could establish a further point of difference with these apps by having good UX. Other things I noted from my market research was that I preferred apps that gave the user more options and ability to customise their teams, alongside apps that worked without internet.

Conceptual Design Process

I know had a clear vision of what the features in *SUBlime* would be, therefore it was now necessary to transform these ideas into a creative reality. To do this, I created a conceptual design. Creating a conceptual design served two purposes. Firstly, it creates a blueprint that I can work towards replicating when I begin the development of my app. This will save me time later as I can focus on coding and not worry about the design process as its already done. Secondly, by creating visualizations of *SUBlime* I can identify potential issues with the design of the app early on, giving me greater time to address them.

Wireframes

The first step of the design process was the creation of basic wireframe diagrams. Wireframes are blueprints of an app which layout content and functionality. They provide a basic structure of what each app page should look like. In my wire frames I also worked to include descriptions of core functionality. I created a wire frame for all four of the features of the MVP.



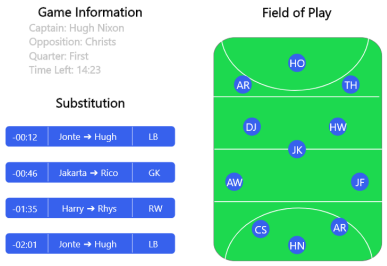
Wireframes of all four features of MVP. Top Left – formation selection. Top Right – Team management . Bottom Right – Subsheet Creation. Bottom Left – game overview screen

Adobe XD designs

I next had to add more detail to these wire frames and create a design that I would aim to replicate in my final product. For this use of a higher end UI/UX design tool was necessary. Adobe XD was my tool of choice for doing this, as the school provided a free licence for me to use it. The other benefits XD provided was that it had a large premade component library making the designs easier to make.

The first page I designed was the game overview screen. This was the screen that was displayed to users during a game. On this screen I would show upcoming subs to users whilst also showing key game information and the layout of the field. On the left-hand side of the page, I decided to display game information alongside the upcoming subs. The game info included who was playing and how long the game had been going for. Below this info I displayed upcoming subs. Each upcoming sub had a countdown until the sub occurred, who was involved in the substitution and at what position this substitution was occurring at. Then on the right of the app I displayed the field of play. This shows a game pitch with all the initials of all the players on the field in their positions.

First iteration of Adobe XD design for game overview screen. Upcoming subs on left game overview on right.



Second iteration of Adobe XD design for game overview. Changes made is adding game info such as a game clock

Slider shown for each player showing their assigned time. On left is total time for each player.

The next page of *SUBlime* I designed was the subsheet creation screen. On the subsheet creation page I wanted to have a slider for each player. On these sliders would be tile for each minute in the game interval. A position could then be added to one of these tiles to signify the player playing this position at that minute. Drag functionality of the slider would then allow the user to increase or decrease their time at a position. Following the bar was a number showing how many minutes the player had been assigned. I showed these designs to my digitech teacher asking if it met good UX conventions. He noted a few design issues.

- Players could only be placed in one category of forward, midfield and defence. There would likely be cases were a player played in two of these categories.
- The design doesn't allow specification of exact positions. For example, Rhys, Adam and Rico start as forward but no description is provided as to what sub role they play as a forward ie left wing, centre forward and right wing.
- It isn't clear who is subbing on for who. Rhys and Adam both sub off at the same time. Likewise, Corin and Harry sub on at the same time. It isn't clear who is subbing for who in this situation.

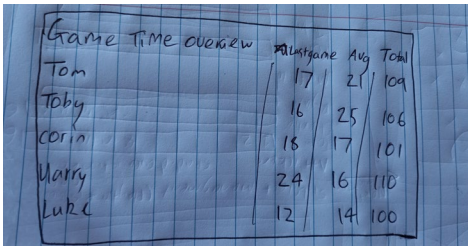
New design for Subsheet Creation Page

Based on these issues I remade the design. This new design shifting from displaying the subsheet in terms of players to position. The subsheet shows each of the positions that the team uses. Each of these positions has a slider that players can be added to and then dragged to give more or less time at that position. This addressed all of the design issues.

This was the final feature I did an Adobe XD design for as they were taking a large amount of time to do, and I was satisfied with the wire frame designs I had for the other two features of the MVP.

Design Meeting with A[redacted] K[redacted]

Now with a physical design complete I organized a second meeting with A[redacted] K[redacted] to discuss the designs and collect his feedback on them. A[redacted] K[redacted] liked the designs expressing how it was nice to have something conceptually complete. He agreed with my reasonings for redesigning the subsheet creation page but rightfully pointed out that by redesigning it I had removed the gametime per player. He stressed since this was a key feature of *SUBlime*, and it would provide very little unique value without tracking gametime, that I should implement a separate page that showed game time stats for all players. Apart from that he was very happy with progress made and suggested we met in a terms time once I had developed an MVP for him to test out.



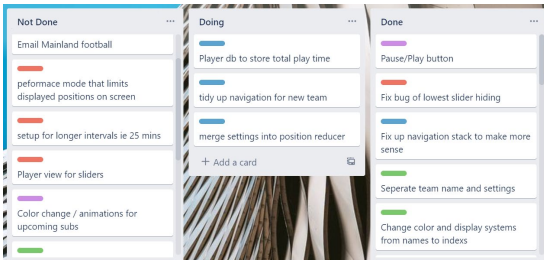
Wireframe design I made of time overview page following meeting. Shows time played in last game, average gametime and total gametime for each player. I sent this A[redacted] K[redacted], and he said it was what he had in mind.

Project Management Setup

A common issues developers face when creating an app is that they run out of time, due to missed deadlines, feature creep causing the project to become out of their control, or poorly managed codebases causing the project to become unmanageable. As this was a project on relatively tight deadlines, I viewed it as necessary to make use of a range of project management tools.

Trello

Trello is a project management tool that allows you to break complex projects down into simpler parts. I had used Trello in the past with other digitech school projects and therefore could appreciate its usefulness and decided to setup a project board. On this board I created three sections: Not started, Doing and Done. I then started creating tasks and adding them to the not started section. Each created task was assigned a tag for the part of the project it made up. For example, I had tags for subsheet creation, team creation and stakeholders etc. Examples of tasks I made was for team creation I added a task for a dropdown to add positions to players. With this set up I was able to have an overview of what tasks I had to do and what I had completed. Throughout the project I moved tasks between the different sections.



Trello board in use when I was overhauling the back end to add teams.

GitHub

GitHub is almost a requirement for any software development project. It offers a range of functionality within project development, but I only made use of it for basic version control. By maintaining version control I would be able to easily rollback any changes to my code base that I regretted. This would allow me to save development time as I could rapidly prototype and easily undo all my changes if I regretted them. Github would also provide a backup in the unlikely event of a computer breakage or file deletion. I created a repository for this project, then at the end of every development session I planned to commit to this GitHub repo.

Word Document Journal

The final tool used was a Word document journal of sorts. On this document I kept a project calendar that I would frequently check back on to make sure I was meeting project deadlines. Also, on this document I also wrote a brief overview of every feature of *SUBlime* that I would come back to over time and refine. This document also served as the basis of this written report as I would constantly take note of any key decisions I made.

Development Tool Selection

A common adage is that you must have the right tools for the job. This adage is particularly true in software development, were making an incorrect decision with the selection of a tool could spiral a whole project. If I selected an overly complex tool, the project wouldn't progress as I would constantly battle to understand how everything works. If I selected an overly simple tool, it may lack functionality that I would need to develop my app. Based on this, these decisions were key.

Mobile App v Website v PC Application

From the start of the project, it was clear that *SUBlime* would only work on mobile devices. This is because you are required to bring the device *SUBlime* is installed on to the sports game. It is not practical to bring a laptop or PC to a sports game therefore a device such as an iPad or large screen phone that could be put in a protective case would be the best choice. This ruled out developing a PC application. Development of a website was also ruled out as I wanted the app to be offline only. This meant the only viable option was a mobile app. As I knew I wanted to develop an app I began looking at cross platform development frameworks.

Possible Development Frameworks

When deciding on a possible app development framework I made a few key considerations when narrowing down my choices. The considerations were as follows.

- The framework must support cross platform development i.e. I develop a single codebase that builds to both Android and iOS. This is to increase accessibility of the app.
  - This would allow a uniform app to be brought to both solutions in half the time of developing both natively.
  - This ruled out using frameworks such as Swift or Android Studio with Java
- The framework must be widely used and therefore have a large number of available documentations and libraries.
  - This would make it easier to learn the language and save time in development as I could make use of pre-existing libraries as opposed to developing my own
- The framework should be open source and be free to use.
  - This reduces the initial cost of the project and makes it easier for me to go under the hood if necessary due to it being open source.
- The framework should be in a language that I have some experience (HTML / JS / C# / Python)
  - This eliminated all Java or C++ based frameworks
- The framework must be able to interpret user gesture input. This was because the subsheet creation page required users to carry out drag motions to allocate time.
  - This ruled out the use of PWAs as they do not have good support for native gesture input.

Based on these considerations I ruled out quite a few development platforms and ended up with three possible frameworks that fulfilled all these criteria with these being Xamarin, React Native and Angular. In addition to this I would also need to select a database to store my data, however I put this off for now and decided to only make this decision later in development when it became clear to me what type of data I would have to store.

Consultation with Tech Industry Veteran B[redacted] L[redacted]

With three possible frameworks decided on I reached out to B[redacted] L[redacted]. B[redacted] L[redacted] is a tech industry veteran and has multiple years of experience working on numerous development projects across sectors of agriculture, finance, and gambling. I had worked with B[redacted] L[redacted] in the past on projects and was always left in awe with his depth of knowledge on various development frameworks. I sent B[redacted] an email asking him what he would recommend, and I received an in-depth reply. B[redacted] recommended Xamarin but also provided information and insight into how I would go about a more web-based solutions such as React Native and Angular.

Hi [redacted]

If I was building your application and it had to run locally would start with downloading the free version of Visual Studio

https://visualstudio.microsoft.com/vs/community/

or Visual Studio code

(https://code.visualstudio.com/)

Then look at C# + Xamarin (there are other options but mobile phone dev is not something I do) for iOS & Android support.

https://visualstudio.microsoft.com/xamarin/

If a webby application worked for you scenario...

I would select a backend hosting environment (Azure, AWS, GCP, Digital ocean, Redhat...) then the development environment (ASP.MVC, LAMP, firebase etc.) then build a webby application with Angular or similar has functionality like "automagic" nice rendering on different vendors devices, aspect ratios etc.

If you go the webby way you should ensure that the application is secure. I would suggest you use something like AzureAD, AWS Directory service etc. as DIY of A&A can end badly.

[redacted] email of advice

Evaluation of Possible Frameworks

With all this information I now needed to decide on a final framework to use. To do this I created a table that evaluated the pros and cons of all these frameworks. X = best in category

Framework	Bryn's Endorsements	Community support/ resources	Performance	Development Environment	Developed by a morally dubious company
Xamarin Forms	X		X	X	X
React Native		X			X
Angular					X

Based on this my final decision for a framework was Xamarin with React Native second. I downloaded Visual Studio and started playing around with Xamarin Forms with the intent of learning how to use it.

Visual Studio IDE

The final development tool I selected for use was the Visual Studio IDE. Ultimately there is very little difference between development IDEs, but I have used Visual Studio Code in the past and therefore selected it out of the familiarity it provided. Also, within Visual Studio Code I downloaded a range of extensions such as Intellicode (that provides autocomplete suggestions), Prettier (to tidy up my messy codebase) and Bracket Pair Colorizers (that makes it easier to determine which bracket matches up with which). Although these extensions may seem superfluous the adoption of them makes small compounding differences which, makes the development process a lot easier and stream-lined over time.

Change in Development Framework

About a week into my use of Xamarin forms I was starting to face some problems. Firstly, I was unable to test my app on iOS. This was because I didn't have an Apple device which was a requirement to build an app for testing as you needed to run it through XCode. Although I could run an iOS emulator on my laptop this would take about 15 minutes to compile due to how slow my laptop was. This was problematic as by continuing to use Xamarin I would not be able to do any quality testing on iOS. Secondly, I found the learning curve of Xamarin Forms to be quite steep and at times it seemed overly complex. Although I was very apprehensive about changing framework, I believed these two reasons as enough to justify changing the development framework. Additionally at this point in the project I was only a week into development and if I was to make a change it would be best to make it now – whilst I had a lot of time remaining - as opposed to later in the project. The framework I decided to swap to was React Native as based on the table I believe it was the next best alternative.

React Native in simple terms is a cross platform app development app that allows users to create native apps for iOS and Android. React Native is based on the React development framework which is a UI framework built on ES6 Javascript. As I move through this report, I will provide explanations of all React Native features on their first use in code.

Learning React Native

Like with Xamarin I put a week aside to learn React Native. I deliberately did this to ensure that I had at the very least a basic understanding of React Native so I didn't make any disastrous programming decisions early on that would have negative repercussions later. Coincidentally, I also had Covid this week, so I was able to really focus for a week on learning the React Native fundamentals. I followed Programming with Mash's 5-hour video tutorial.[3] This tutorial taught me all the React Native basics whilst also providing an introduction to higher level React Native concepts such as Redux, async storage and navigation.

Testing Methodology Throughout Development

Testing with stakeholders would be unfeasible and unpractical until the MVP was complete. I had agreed this with A[redacted] K[redacted] at our second meeting as we both noted that it would require a large degree of effort and coordination to organize a live test with a team for every newly added feature. We both believed the quality of the feedback would not meet this required effort. Therefore, during development, I instead decided to use my classmates for testing features as I developed them. I would carry out testing with a variety of classmates some of whom play sports or coach teams as they would be able to provide insight as potential future users of the app and others who are more technically minded who woukd try and find bugs in the app. I would then send a fortnightly email to A[redacted] K[redacted] detailing development process and asking for his feedback. Once the MVP is complete, I would then begin to carry out live tests with teams.

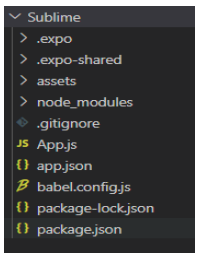
[3] https://www.youtube.com/watch?v=ANdSdIlgsEw

Development of Minimum Viable Product

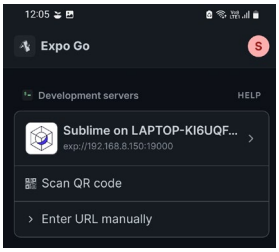
Setting up Project File

When setting up a React Native project you can choose between two development environments. The first option is Expo Go which is an SDK that provides a wide array of development APIs to use, Expo Go also carries out project management and can manage dependencies. Expo Go also offers the ability to do across the cloud testing without having to using Android Development studio or XCode. The second option is using React Native CLI. React Native CLI lacks most of the SDK features provided by Expo Go and requires the use of XCode and Android development studio to test the app. However, to make up for this React Native CLI allows users to interact with the native code of the app. Although React Native CLI provided a more in-depth range of features, it is a lot harder for beginners to use due to it lacking the SDK features of Expo Go. I decided to trade off the functionality of React Native CLI for the ease-of-use Expo Go provided.

The install process of Expo Go is well documented on their forums.[4] First I installed Node JS. Node JS has multiple application but the main reason I made use of it was to manage the installation and maintenance of packages with its NPM (Node package manager) feature. With this setup I then ran an install of Expo. Whilst this install was happening, I created an Expo account. After the install was complete, I ran **npx expo login**. Next, I ran **npx create-expo-app SUBlime**. This command generated all my project dependencies and files into one single folder – essentially it completed the whole project setup process for me. With this setup I then had to download the Expo Go app onto my phone. With the app installed I ran **expo start** which began a development server. I could then select this development server on the expo app to open my app and test it on my phone. Under the hood Expo is running a packager called Metro that bundles the app and then sends it to my device to emulate on the Expo app.



Project file generated from running create-expo-app



Expo App on my phone. The development server appeared after running expo start. I could then tap on the server to run my app.

PlayerTab Component

The first component I decided to develop was the PlayerTab component. The PlayerTab component would make up the team management screen. A PlayerTab component would exist for each player and on this PlayerTab a player's data could be entered such as their name and positions. In comparison to what I had planned in the future this component seemed relatively easy to make therefore I decided to develop it first to allow me to hone my skills on an easier task. The design constraints of this component were that it had to be easy for coaches to add players data. This is so setup time for a team is low, and users aren't turned away from the app by a high initial setup time.

Introduction to Basic React Native Paradigms

Before explaining my development of this component, it is necessary to describe the key concepts of React and to the extension of that React Native. The first main concept is JSX. In simple terms JSX is the combination of HTML (markup) and Javascript (Logic) into a single language. It was created on the belief that rendering logic is inherently coupled with UI logic. JSX is used to create components in React Native. Components are functions that accept inputs (called 'props') and return React Native elements that should be rendered to the screen. With this basic understanding established when developing the PlayerTab component I would need to have props for the player's name and the positions the player has assigned so it could be displayed on the PlayerTab.

```
const element = <h1>Hello, world!</h1>;
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}
```
Example of JSX and React Native Functional Component

[4] https://docs.expo.dev/get-started/installation/

Development of the PlayerTab Component

When starting with the development of the Playertab component I first created a JS file for the component, added the associated boilerplate code to define the function and then added in a View component.[5] React Native has a large library of pre-made components available such as the View component. Wherever a component existed for what I wanted to achieve I opted to use the component to save time. The View component is functionality identical to a HTML div tag and acts as a container for content. To this View component I applied a style prop. The style prop refers to a created Stylesheet[6] and assigns associated styling to the component. The styling I applied made each of the PlayerTab components span the width of the screen, have a distinct background and nice rounded borders.

```
//Container view tag for the playerTab
<View style = {styles.playerTab}>
//Defined styles very CSSesqe in styling
playerTab : {
    backgroundColor: 'white',
    width: '98%',
    height: 50,
    marginLeft: '1%',
    flexDirection: 'row',
    borderColor: 'black',
    justifyContent: 'center',
    alignItems: 'center',
  },
```

Example view tag and styling used on PlayerTab

Next within this styled View component I started adding the features of the PlayerTab. I first added a TextInput[7] component. This component was another prebuilt React Native component and allows users to enter their name into a text field. I added a prop on this TextInput component for placeholder text that made it clear to the user that this is where they needed to tap to enter their name. Next to this TextInput I added a RNPickerSelect[8] component. This component took two required props the first items which contains an array of objects. Each object has a label and value field that determines what items are rendered in the list. On this list I hardcoded an object for all selectable positions the player can choose in a hockey match. The second component was an onValueChange prop that calls a function when a new value is selected from the list. I also added a range of others props to it that handled the placeholder and styling of the component.

```
//Implemented Text Input component
<TextInput
    style = {styles.playerTextInput}
    placeholder='Player Name'
    placeholderTextColor="grey"
/>
//Implemented RNPIckerSelect
<RNPickerSelect
    onValueChange={(value) => setSelectedPos(value)}
    placeholder={{ label: 'Add positions', value: null }}
    style = {pickerSelectStyles}
    items={[
          { label: 'Left Foward', value: 'LF'},
          { label: 'Center Foward', value: 'CF' },
          { label: 'Right Foward', value: 'RF' },
          { label: 'Center Midfield', value: 'CM'},
          { label: 'Left Half', value: 'LH' },
          { label: 'Right Half', value: 'RH'},
          { label: 'Left Back', value: 'LB' },
          { label: 'Right Back', value: 'RB'},
          { label: 'Goal Keeper', value: 'GK' },
     ]}
/>
```

Setup Textinput for player name and RNPickerSelect to choose a position

With the addition of the RNPickerSelect I was now faced with a new issue in the form of how to store positions for each player. Initially I solved this by using Hooks, React Natives solution to managing state (I scrapped this later, but I will use this as an opportunity to explain Hooks). Hooks are a type of function that lets you 'hook into' React features. There is a multitude of Hooks available to hook into a range of features but in this case, I made use of the useState hook that allows state to be added to components. useState hooks are defined in the general form off const var_name = [<getter>, <setter>] = useState(<initialValue>). When called, the useState function returns two functions the first a getter which when called returns the value of the state and the second a setter which when called updates the state. I set up a hook state variable for playerPositions - which was an array - then created a function called addPosition. This function first checked if a player had a position and if not added the position to the player. I attached this addPosition function to the onValueChange prop of the RNPickerSelect were I parsed the data of the selected position. Also state hooks are immutable objects and therefore a new list of positions must be created then passed into the hooks setter function to not violate immutability.

```
//Defined state variable hook
const [playerPositions, setPlayerPositions] = useState([]);

function addPosition(value)
{
  //Check if added position isn't already in list
  if (!playerPositons.includes(value))
  {
    //Create new list with added items. New list must be created
in order to not violate immutability
    const newList = list.concat({ position: `${selectedPos}`});
    setPlayerPositions (newList);
  }
}
```

State hook is defined for players position as an array. Add position function adds a new position if that position wasn't already added

[5] https://reactnative.dev/docs/view
[6] https://reactnative.dev/docs/stylesheet
[7] https://reactnative.dev/docs/textinput
[8] https://github.com/lawnstarter/react-native-picker-select

With a way to add positions to the players I now had to display these positions. For this I made use of the React Native Paper Chip[9] component. A Chip is a pressable component that when pressed is dismissed. I made these chips into a renderPositionChip component which took a prop for the position name and then rendered a Chip with that position name. With these Chips I had to render one of them for each position. Therefore, I had to make use of the Flatlist component[10] to render multiple of the same component. The mandatory props of a Flatlist are a data prop which I parsed the playerPosition list into. The second prop is the renderItem which I passed the renderPositionChip component. The final prop is keyExtractor which gets the id for each rendered item, for this prop I used a hacky solution to pass the id of each rendered item as their position in the list. With this set up I now had the ability to add a name to the PlayerTab then assign positions to that player.

```
//Render a position chip for each position in the list
const renderPositionChips = ({ item }) => (
   <Chip
      style = {styles.positions}
      onPress = {() => {deletePosition(item)}}
      onClose = {() => {}} >{item.position}
    </Chip>
);

//Flatlist component that renders a position chip for
each item in the playerPostions list
<FlatList
    data={playerPositions}
    renderItem={renderPositionChips}
    horizontal
    keyExtractor={item => list.indexOf(item)}/>
```

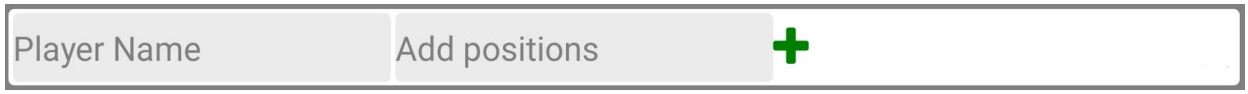
Flatlist that renders a chip for each position.

First iteration of PlayerTab. Users can enter a name, then add a position. Added positions are then displayed as chips.

I next applied a series of refinements to this PlayerTab. The first was the ability to remove positions from the player. To do this I added an onPress prop to the position chip that called a deletePosition function. When called this function filters the deleted position out of list then updates the position state hook. Next, I added a pressable to the PlayerTab to confirm whether the user wished to add a position to a player. Originally when the user tapped a position for a player from the RNPickerSelect it would add the position instantly. This would result in positions being accidentally added. To address this, I added a Pressable[11] component to the PlayerTab that the user would have to tap to confirm the addition of the positions. Within this Pressable component I put an icon. To add an icon, I made use of the Vector Icon component[12]. This Vector Icon component had access to a wide array of icon libraries such as FontAwesome. When I add an icon, I can search these libraries[13] then reference the icon I want in the icon component, and it would appear. The Pressable component called the addPosition function I made earlier when pressed. However, the addPosition function would not know what the currently selected position was. To address this, I created a state hook that stored the currently selected position. This was updated when a position was selected in RNPickerSelect then when the addPosition function is called – on the press of the plus icon - that position is added to the player. The final change I made was styling touch ups to make the PlayerTab look nicer.

```
//Pressable component with plus icon inside
<Pressable
    style = {styles.positions}
    onPress = {addPosition}>
        <Icon
            name='plus'
            size = {30}
            color = 'green'
         />
</Pressable>
//Delete position fuction
function deletePosition(chip) {
    //Make a new position list with out the removed chip
    const positionList = list.filter((item) => (item) !== chip);
    //Reset the list variable with a new value
    setPositionList(positionList);
}
```

Pressable that when pressed adds new position. Delete position function is called when a chip is pressed to delete that chip

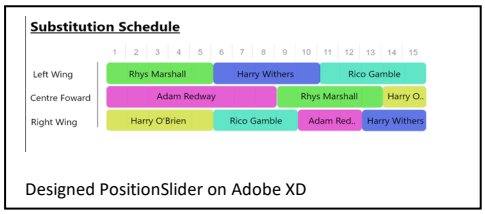


Refined and more stylized PlayerTab Component

[9] https://callstack.github.io/react-native-paper/chip.html
[10] https://reactnative.dev/docs/flatlist
[11] https://reactnative.dev/docs/pressable
[12] https://github.com/oblador/react-native-vector-icons
[13] https://oblador.github.io/react-native-vector-icons/

PositionSlider Component

The second custom component I developed was the PositionSlider. This component was going to make up the subsheet creation page, with a PositionSlider existing for each position. This component would allow a user to assign a player gametime over a given time period. Drag functionality would then be available for the user to allocate the player more or less gametime. This drag functionality had to be easy to use so subsheets are quick and easy to setup. Creating this component was going to be very difficult as I would have to build it entirely from the ground up as no components, that achieved the same functionality already existed. When designing this component, it went through multiple phases of designs as I attempted a range of differing solutions all with varying degrees of success.



Designed PositionSlider on Adobe XD

First Iteration of the PositionSlider Component

The first solution I attempted was making use of Pressable components. I planned to render the Pressable components in a Flatlist such that there was one Pressable component for every minute that could be allocated. I planned for users to be able to drag over these Pressable's to allocate more or less time to player. However, this solution was short lived as I soon realised the Pressable component lacked key functionality such as being able to detect whether a long press is occurring and whether a drag motion has been carried out. This meant that the basic components provided by React Native would not provide me with an in-depth enough gesture handling.

Second Iteration of the PositionSlider Component

As the provided React Native components were unable to handle raw gesture data the way I wished, I would have to find a way to access users raw gesture data. After a bit of searching, I found a React Native Gesture Handler library[14]. This library transforms native gesture data into interactable data. It provides a range of components that you can use based on the type of gesture you wish to interpret. In the case of the PositionSlider, I wanted to know the side to side drag of the user's finger therefore I opted to make use of the PanGestureHandler[15]. When setting this up in code I first had to wrap my component inside of a GestureHandlerRootView. Inside of this I put a PanGestureHandler. The PanGestureHandler had props for gestureStart, gestureActive and gestureEnd. In these props a function can be placed that would be called when those events happen. Additionally the PanGestureHandler provides event data to these functions such as finger x, finger y position of drag, initial x,y of drag that I could then use in my calculations. With this data I could determine how far a player had been dragged and therefore how much time to assign them.

The second iteration made use of the React Native Gesture Handler Library. To begin with I created positionData state hook that contained an object for each added player to that position. The object would contain the player's name, colour to display for player, start x position of player on the PositionSlider and end x position of player on the PositionSlider. To this data I ran a formatting function that made it nicer to display. This data was formatted under a process I called SNAP – FILL – CUT. Although this may sound like a torture method it couldn't be further from that (although adding this process may have tortured me mentally). The first step of SNAP rounded all the start x and end x values such that they all started and ended exactly at the edge of one of the minutes. The value I rounded to was, the multiples of was the expected width of each minute tile that was determined based on the number of minutes in the interval and the width of the screen. Next came FILL, in this step I filled in gaps in the PositionSlider. For example, there may only be two players assigned for a single position meaning there may be unassigned minutes. The FILL step ensures that these gaps are filled with an empty object. This was necessary due to me not using absolute styling. The final step is CUT where unassigned time objects in the PositionSlider that span more than one minute are broken into smaller unassigned time objects that only span one minute. This was done so that each unassigned minute can have a RNPickerSelect on it allowing users to assign a player to that unassigned time.

The table below shows values going through the SNAP-FILL-CUT process. Note that in this case the screen width is 500 and the interval is 5 minutes long. This means every minute has a width of 100.

[14] https://docs.swmansion.com/react-native-gesture-handler/
[15] https://docs.swmansion.com/react-native-gesture-handler/docs/gesture-handlers/api/pan-gh

Step Applied	Initial	SNAP	FILL	CUT
Data values	Tom, startx:100, endx: 160 Riley, startx: 380, endx:490	Tom, startx:100, endx: 200 Riley, startx: 400, endx:500	Unassigned, startx: 0, endx: 100 Tom, startx:100, endx: 200 Unassigned, startx: 200, endx: 400 Riley, startx: 400, endx:500	Unassigned, startx: 0, endx: 100 Tom, startx:100, endx: 200 Unassigned, startx: 200, endx: 300 Unassigned, startx: 300, endx: 400 Riley, startx: 400, endx:500
Comment	These initial values don't precisely sit within a minute. This could be from a drag that recently finished which changed these vales.	All the values are rounded to the nearest 100	Two gaps occur and both of them are filled with unassigned objects. These unassigned objects can have players added to them.	These unassigneds are then cut so they take up one minute each

The next step was visualizing each object in the PositionSlider. When displaying the PositionSlider I opted to make use of the .map() method to render a component for each object as opposed to a FlatList due to the data being rendered not requiring lazy loading which is the main draw of Flatlist. For each rendered component in the PositionSlider, I wrapped it in a PanGestureHandler to detect the drag starting and ending. Next, I defined the width of each object as the difference between the end and start x then set that equal to its width. I also assigned the colour of the view to be equal to the colour of the player. Finally, I used conditional rendering that rendered based on whether the minute had a player allocated to it. If it did an empty view would render if not a RNPickerSelect would render allowing a new player to be selected for that place.

```
{/*Render each item in the positionData list by mapping it to a component. */}
{positionData.map((prop,index) => {
    return (
        //Panguesturehandler controls the drag functionality. Functions are called based on current drag event
        <PanGestureHandler key = {index}
             onGestureEvent = {(drag) => dragActive(drag,prop)}
             onActivated = {(drag) => dragStart(drag,prop)}
             onEnded = {(drag) => dragEnd(drag,prop)}>
             {/*Set the width, color and name of the item based on what the object is at that point in the list.*/}
             <View  style={{...styles.tagSection ,  backgroundColor:prop.color, width: (prop.end-prop.start)} }}>
             {/*If object is empty render RNPickerSelect to allow user to add player at that minute. If not empty render the
players name at that position*/}
                 {(prop.name != 'Empty') ?
                     <Text numberOfLines={1}  style = {{...styles.tagSectionText} }>{prop.name}</Text>:
                     <RNPickerSelect
                         onValueChange={(value) => {addNewPlayer}}
                         placeholder={{ label: '+', value: null }}
                         style = {pickerSelectStyles}
                         items={[{ label: 'Alex Ying', value: 'Alex ying'},
                                 { label: 'Jerry Chang', value: 'Jerry Chang' },
                                 { label: 'Callum Lockhart', value: 'callum lockhart' }]/>
             </View>
        </PanGestureHandler>
    )}
```

Following this I had to deal with the users drag input and adjust the minutes allocated based on that. As shown above the PanGestureHandler has props that refer to a function when a certain event occurs. The first of these events called is onActivated or when the drag begins. When the drag begins, we want to determine the direction of the drag. This is because one end of the dragged player object must remain fixed during the drag and that end would be opposite to the drag's direction. To determine the drag direction, I compared the x position of the drag start with the midpoint of the player object. If the x position was greater than halfway of the players width the user was dragging to the right if its less, they are dragging to the left.

The next event called is onGestureEvent which is called at every render cycle when the user is dragging. As the user moves their finger the width of the players object must change to reflect this drag. The way this is updated is dependent on the direction of the drag because if the player is moving their finger right you want to increase the size of the players objects and reduce the size of the player or unassigned time object to the right. Likewise, if you drag to the left you want to decrease the size of the player or unassigned time object to the left. With the dragged player bar now moving two cases of movement must be considered.

```
//If drag direction is to the right decreasing size of object to right
updatePlayerData[prop.index].end = drag.nativeEvent.absoluteX-sliderBarMargin
updatePlayerData[prop.index+1+amountDeleted].start = (drag.nativeEvent.absoluteX-sliderBarMargin)

//If drag direction is to the left decreasing size of object to left
updatePlayerData[prop.index].end = drag.nativeEvent.absoluteX-sliderBarMargin
updatePlayerData[prop.index-1-amountDeleted].start = (drag.nativeEvent.absoluteX-sliderBarMargin)
```

Firstly, the player increases the size of the player object and overlaps an adjacent object. When this occurs the adjacent object no longer renders as it has a width value of 0. Although it is no longer rendered it still exists in our positionData list. This is an issue as the adjacent object no longer has an index difference of 1

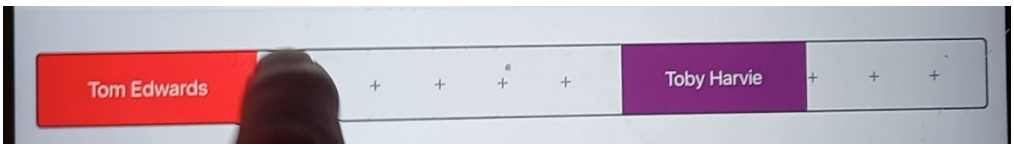
instead it has an index difference of 2 as the previously adjacent object is now covered up. However, our code doesn't know this meaning that it doesn't decrease the size of the next object after the overlapped one. The obvious solution to this would be to delete the overlapped object from the positionData list however this causes issues when deleting an object to the left of the selected one as it causes the index of the current object to decrease cancelling the drag as the .map() method to render the objects becomes confused. Therefore, an index offset must be applied based on how many objects are overlapped. This offset is added to the code the reduces the width of the adjacent object.

```
//As we cant delete the covered object as it would cause an indexing error we must instead 'skip' over it by increasing the
amountDeleted index that we then add to all the code. If the end of an object is before its
if ((updatePlayerData[prop.index+1+amountDeleted].end - updatePlayerData[prop.index+1+amountDeleted].start) <= 0)
{
  setAmountDeleted(amountDeleted+1)
}
```

The second case is when the player decreases the size of their object. As they decrease the size of their player object a new object must be created in that area. If the area isn't filled all content progressively moves towards the direction of the drag due to absolute styling not being used. To counteract this a state hook is created that keeps track of the original position of the end of the dragged player object. This value is taken in the dragStart event. If the drag position is behind the stored value in the state hook a new unassigned object is created to fill this 'deleted' space from the dragged object. In addition to this the index offset must be decreased by one to account for the addition of a new object to the list.

```
if (drag.nativeEvent.absoluteX-sliderBarMargin <= enterPos)
{
  //Splicing a list creates a new version of that list with an item inserted at a given index. As a new version of the list
is created immutability isn't violated.
  updatePlayerData.splice(prop.index+1+amountDeleted,0,{
    name: 'Empty',
    color: 'white',
    index: 0,
    start: drag.nativeEvent.absoluteX-sliderBarMargin,
    end: enterPos
  })
  //We dont want this code to fire again therefore we set the enter pos to 0 which is the
  //bottom boundry of the PositionSlider and therefore will never be dragged across
  setEnterPos(0)
  //Indexing is adjusted by -1
  setAmountDeleted(amountDeleted-1)
}
```

Finally at the end of the drag the data formatting function is called that formats the data through the SNAP – FILL – CUT process. This is done so that all the objects are formatted and displayed correctly.



I managed to lose all the screen shots of this PositionSlider iteration as a I deleted them when I moved onto the next iteration of the PositionSlider. I only had left a video of me dragging a player object, so this screenshot is from that video. If a user pressed a plus they would be able to add a new player there. Also note visual glitches of the + not being correctly aligned. I talk more on this later.

Linking the PlayerTab and PositionSlider Components

With a basic implementation of both the PlayerTab and PositionSlider components my next goal was to link them up to allow created players to be added to the PositionSlider and then assigned gametime. The implementation of this was complex as I had to setup navigation between pages and then find a way to pass data between these pages.

Navigation Between Pages

The first problem I tackled was adding navigation between pages. For this I made use of the React Navigation library[16]. The React Navigation library supports a wide range of Navigation paradigms such as stack navigation (how navigation works on webpages), tab navigation (navigation buttons at bottom) and drawer navigator (navigation buttons that can be pulled out at the side). Out of these three options I decided to use stack navigation as it was the simplest to use. Should I need to make use of another navigation paradigm I could set that up in the future when I had improved my skillset with this library. Setting up the stack navigator requires adding a lot of boiler plate code within the App.js file which is the index page of the app. All pages within the stack navigator must be defined within this App.js file.

```
//Creates the native stack navigator object. This gives the navigation a
native feel
const Stack = createNativeStackNavigator()

const App = () => {
  return(
    // Container that all navigators must be defined within
    <NavigationContainer>
      {/* Within the stack navigator all screens in the Stack must be
defined. This is done by refrencing their component */}
      <Stack.Navigator>
        <Stack.Screen name= 'TeamManagment' component={TeamManagment}/>
        <Stack.Screen name= SubsheetCreation'component={SubsheetCreation}/>
      </Stack.Navigator>
    </NavigationContainer>
  )
}
```

Stack Navigator setup with a screen defined for every page that is navigated to

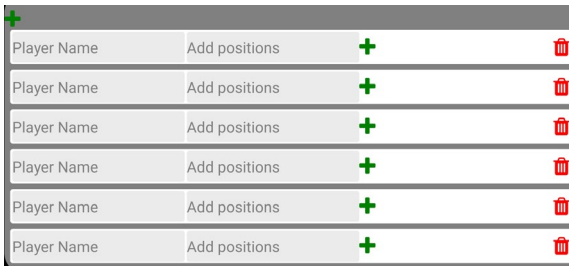
With this boilerplate setup the next step was implementing the code that allowed the navigation between pges. To do this I setup a Pressable that had an onPress prop. When this onPress prop was called a navigation function was run that had the name of the page – as defined on the index page - to move too. Above is the setup pressable to move between the team management and PositionSlider page.

```
<Pressable
  style = {styles.nextPage}
  onPress =
{()=>navigation.navigate('SubsheetCreation')}>
  <Icon
    name='check'
    size = {30}
    color = 'green'/>
</Pressable>
```

Use of Pressable to call the navigate function to move the user between team management and the subsheet overview page

Scaling up the PlayerTab Component to make the Team Management Page

With the ability to move users between pages the next step was to allow data to move between pages. However before doing this, I needed to have meaningful data to move between the pages. I had yet to scale up the PlayerTab system to allow multiple players to be created. Therefore, I had to scale up this system to allow multiple players data to be passed to the next page. To do this I created a playersData state hook that was an array that contained all the player objects. Next, I made a createPlayer function that added a new empty player object to the playersData array. After implementing this I realised that all player objects must have a unique id. To address this, I created a playerId state variable that incremented by one for every player created. This meant that all players had a unique id. The final step was to render this all the players. To do this I made use of a Flatlist where the render item was the PlayerTab and the data used was the playersData.



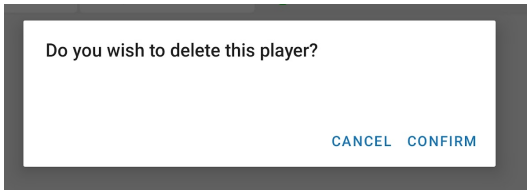
Scaled up player team management page. Displayed are multiple empty Player Tab components. The plus in the top left would create more.

You may also note that in the above screenshot there is a trash icon for every player. This trash icon is universally recognized in UI/UX as a symbol for deleting something. In this case when its pressed it will delete the player. When setting up a deletion system I created a Pressable with the trash icon inside of it. I then set the Pressable onPress prop equal to the deletePlayer function. Inside this function is an alert component[17]. An alert is a dismissible static pop up. They are generally used as warnings or to provide app users with information. In this case the alert appears as a confirmation – confirming whether the user wishes to delete this player. If the user does select the option to delete the player another function is run to remove the player from the playersData array. The alert was deliberately used to prevent users from accidentally deleting players that they spent time setting up. With the scale up of the PlayerTab now complete I have achieved a complete team management page. Any further reference to this page and its component in the report will be under the name of the team management page.

[16] https://reactnavigation.org/docs/navigating
[17] https://reactnative.dev/docs/alert

```
const deletePlayer = (playerId) => {
    //Create alert to show to player
    Alert.alert(
    "Do you wish to delete this player?",
    '',
    //Selectable options are defined as a list of objects
    [{text: "Cancel", style: "cancel"},
        {text: "Confirm",onPress: () => removePlayer(playerId)}])
}
```

Delete player function. Options for users are defined as objects in list.



Pop up alert that confirms whether the user wishes to delete the player

Data Management Between Pages

Now I was able to begin passing the data between pages. First, I looked at using React Navigations route parameters[18]. Route parameters allows variables to be passed in the navigation method and then accessed on the following page. It is very similar to the $_GET method in PHP. Although this solution appealed due its simplicity it was not a scalable solution. This is because later in development there may be 10+ unique variables that need to be passed between each page and this data must be passed to multiple pages. This becomes worse when multiple components are updating the state at the same time. This would cause a massive headache and therefore I opted out of this system. What I was looking for was a data management system that stored data globally in a scope that encompasses all pages so data can be passed downward to both pages as opposed to being passed between the pages. This would also allow any component in the project to update and access state which wouldn't be previously possible. Based on this I decided to make use of Redux[19] which is a global state container.

Redux Explained

The Redux data management paradigm is initially quite complex to wrap your head around, but I will do my best to provide a succinct explanation. The first concept to understand is **Actions**. Actions are an event that describes something that happened in the application. Actions have two fields. The first is type which contains a descriptive name of what the action does. The second field is payload which can be used to pass additional information when an action is called such as the value of a variable. Actions must contain the dispatch method in order for it to reach a reducer when called.

```
//Action being defined in the action.js file. As you can see the action is defined as an object that contains a descriptive
title of what the action does and has a payload that allows the newly created player data to be passed through
export const CREATE_PLAYER = 'CREATE_PLAYER';

export const create_player = new_player_data => dispatch => {
  dispatch({
    type: CREATE_PLAYER,
    payload: new_player_data,
  });
};

//Dispatch method being defined in the team creation page and wrapped around the create player action.
const dispatch = useDispatch()
const createPlayer = player_data => dispatch(create_player(player_data))

//Create player being called inside a player creation fuction. As you can see an empty player object is being passed as the
payload for this aciton
createPlayer({
  id: newPlayerId,
  name: '',
  positions: [],
})
```

The second concept is **Reducers**. Reducers are functions that receive the current state of an application alongside an action. The reducers then use the action to decide how the state of the application should be updated. Reducers are like the JS concept of event listeners. Additionally, a few rules must always be applied to Reducers. These being they must not modify the existing state and instead must use an immutable update.

[18] https://reactnavigation.org/docs/params
[19] https://redux.js.org/

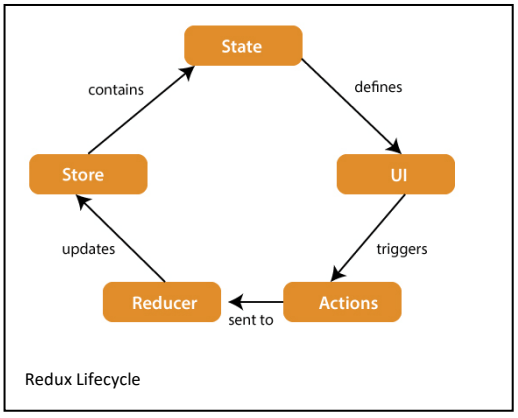
```
//Player reducer function inside the reducer.js file.
function playerReducer(state = initalState, action)
{
  //The reducer first takes the passed action type to determine what update to the state is being called
  switch (action.type)
  {
      //As you can see when the all updates are done immutability and action.payload is reference when making updates
      case CREATE_PLAYER:
          return {...state,player_data: [...state.player_data, action.payload]};
      case REMOVE_PLAYER:
          return {...state,player_data: state.player_data.filter(item => item.id !== action.payload)};
      default:
          return state;
  }}
```

The third concept is **Stores**. The state of a redux application lives in an object called a store. The store is created by passing in a reducer. The state of the store can be accessed through out the application by using selectors which allow a specific part of the state to be referenced.

```
//Creation of the store in store.js
//As the project progress I will create more reducers therefore I implemented the combinereducer for future proofing
const rootReducer = combineReducers({
    playerReducer,
});
export const store = createStore(rootReducer);

//Accessing of the player data within the PositionSlider component
const playersData = useSelector(state => state.playerReducer);
```

To the right is a useful diagram that explains the relationship between these concepts and how data is passed around the app. As you can see from just the code above only having a CREATE_PLAYER action requires a lot of boiler plate. Therefore, later in development when I could have upwards of 50 actions the codebase could get very bloated. To combat this, I setup three js files for each of the concepts, these being actions.js, reducers.js and store.js. I exclusively put the associated concepts on these pages. Initially I setup global redux stores for player_data and wrote reducers to add new players and change the data of players.



Redux Lifecycle

Third Iteration of the PositionSlider Component

With the player data now being accessible on the PositionSlider page I could now add the ability for created players to be assigned time on the PositionSlider. Before that I would first need to scale up the PositionSliders such that there was one for each position on the field. When I went to complete this scale up, I faced performance issues. This was because of design issues with the PositionSlider alongside this the PositionSlider had two other issues resulting in me having three main issues with the current implementation of it.

1. Poor performance. This was because the function that was called whilst the drag is active had a large amount of logic in it. Every render cycle that the drag was happening this logic ran, tanking the performance of the PositionSlider. This caused the drag to appear ‘laggy’ as the user’s input was only rendered a few seconds after their motion.
2. Glitches related to the indexing of list. As I constantly deleted and added new objects to the list during a drag if an object had been overlapped or pulled back on. This resulted in bugs related to indexing as the items in the list invariably changed.
3. Visual glitches / bugs from rounding code. When new objects are created their width is found by dividing the screen width by the amount of minutes in an interval. Rounding is applied to this resulting in inconsistent values being obtained. This causes visual glitches as rendered players shift in width dependent on how they round. This cased a minor ‘glitchy’ effect.

Based on these issues for the project to continue with a decent degree of success I would need to redesign the PositionSlider component. This would be the third iteration of the component and therefore I could not stuff it up again if I wished to finish the project before the end of the winter sports season. Therefore, I spent a large portion of time planning out this new iteration. The first step I took in planning was looking at the problems of the old component and determining how I would them in the next iteration. The first issue with

lag caused by logic being run every render cycle could be addressed by limiting the amount of logic being run when a drag was occurring. The indexing related issues could be fixed by having the data structure storing the information of fixed length with no objects added or deleted from this list. The rounding related issues could be fixed by applying consistent rounding such as flooring as opposed to using the randomness of rounding.

By the end of the design process, I decided to use a multi-layered component with three separate layers. The bottom layer would be a component layer that contains the RNPickerSelects for each minute allowing players to be assigned. The second layer would be a visual layer that displays the data for what player is playing when by having a player blob that would cover up the minutes that player has been assigned. The final layer would be the drag layer that handles the drag. I would render these layers on top of each other by styling them absolutely.

The Component Layer of the PostionSlider

I first started by setting up the component layer. First, I setup a new global state variable within the Redux store. This state was for position_data. I defined position_data as having two fields. The first field was for the position name, in this case it was 'CF' and then the position_timeline. The position_timeline is a list that is the length of a game's interval (in this case its hardcoded at 15 for hockey). The index of the list corresponds to the minute of play and the value at that index is an object that determines who is playing in that minute and what colour to display them as. The position_timeline is also of fixed length meaning it will not encounter indexing issues like the previous iteration.

```
//Added a new field to the state
const initalState = {
  player_data: [],
  position_data: [{position_name: 'CF', position_timeline: new Array(15).fill({name:null, color:null})}]
}
```

With that setup I next rendered the component layer. For each value of position_timeline I rendered a View component. Each view component was styled with a flex value of 1 meaning that each of them took up equal space. I then applied conditional rendering to determine whether the RNPickerSelect component should be rendered. I checked if that value of the position_timeline was null and if so, no player was allocated that minute and therefore the RNPickerSelect should be rendered. I got the selectable values for the RNPickerSelect by mapping the player_data which I retrieved from the redux store into the format of label and value for the RNPickerSelect. This would allow users to allocate any of their created players that had that position the unassigned minute. I also set the placeholder values of the RNPickerSelect to equal the one above its index value, so it represents its minute of play.

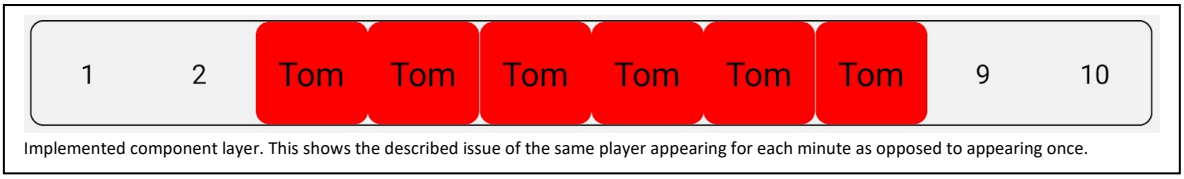
```
//Creation of pickerSelectData by mapping the player data in the form of
selectable pickerselect options
const pickerSelectData = globalState.player_data.map(item => ({label:
item.name,value:item.name}))

//Rendering of the component layer
globalState.position_data[0].position_timeline.map((prop,index) => {
    return(
      <View key = {index} style = {{flex:1,borderColor:'black'}}>
        {
          (prop.name == null)  ?
            <View style = {{alignItems:'center',justifyContent:'center'}}>
              <RNPickerSelect
                onValueChange={(value)=>{updatePosition([index,value,'CF'])}}
                placeholder={{ label: (index+1).toString(), value: null }}
                style = {pickerSelectStyles}
                items={pickerSelectData}
        /></View> : <View/>
        }
      </View>
  )
}
```

The final step was adding the code for when a player was selected. For this I wrote a reducer for updating this position that took the selected players name, position minute and position name. I added this reducer to the onValueChange of the RNPickerSelect such that when a player is selected at a minute the position data is updated to reflect this change.

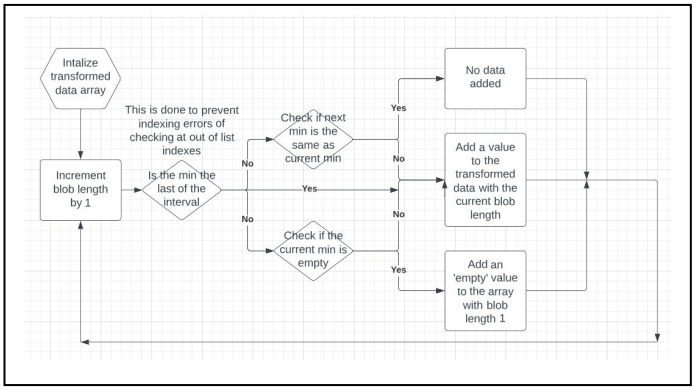
```
//Update position reducer that highlights the pain of immutable updating. First it finds the corresponding position. Then it finds the
corresponding minute of play. Finally it updates the name at that value to reflect the selected player.
case UPDATE_POSITION:
          return{...state, position_data: state.position_data.map(
            (content, i) => content.position_name === action.payload[2] ?
              {...content,
                position_timeline: state.position_data[i].position_timeline.map((content,i)=> i===action.payload[0] ?
                  {...content, name: action.payload[1], color: 'red'}  : content)}
          : content)}
```

The component layer was now setup and users could select a player to add at a certain minute. The next step was reflecting this change visually by making the players name appear over the minutes they had been assigned. My initial solution to this was using the conditional rendering of the component layer. Should the RNPickerSelect not render because the minute had been assigned, I instead rendered a view component that had the player's name inside of it. This worked but it was ugly because if a player had been assigned 10 minutes their name appeared 10 times for each minute. It would have been a lot visually nicer if their name only appeared once and was centred in the middle of their 10 assigned minutes.



Implemented component layer. This shows the described issue of the same player appearing for each minute as opposed to appearing once.

The Visual Layer of the PositionSlider

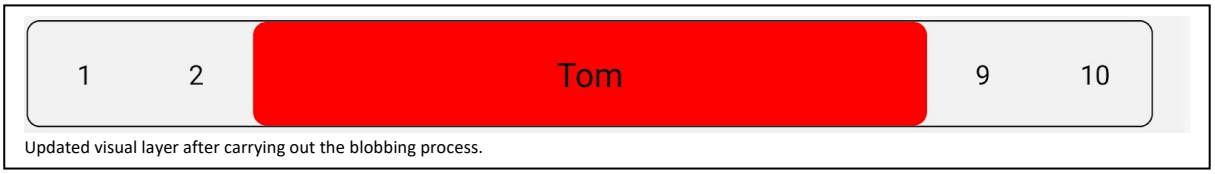


To implement this, I created a visual layer. The view container of the visual layer was styled with absolute positioning meaning that it rendered on top of the component layer. In the visual layer I wanted to carry out a process I referred to as 'blobbing'. In blobbing, single minutes players play in a row are blobbed into a single view, as opposed to existing as single one-minute views. This allows the players name to only appear once for the blob and be centred at the middle of the blob. This would cause a player playing from minutes 5-10 to have one single view that spanned five minutes as opposed to five separate views that spanned one minute. The implementation of this blobbing function is shown in the logic tree above.

```
const blob_data_for_visual_display = () =>
{
  //Initalize an empty array that the blobbed data will be put into
  let blobbed_data = []
  let blob_length = 0;
  let interval_length = globalState.position_data[0].position_timeline.length
  //loop through whole list
  for(let i = 0; i < interval_length; i++)
  {
    //Incerement blob length
    blob_length += 1
    //Check if the current minute is not assigned to a player
    let isMinEmpty = globalState.position_data[0].position_timeline[i].name == null;
    //Check if the minute to the right of the current minute has the same player assigned.
    let isNextMinSame = globalState.position_data[0].position_timeline[i].name ==
globalState.position_data[0].position_timeline[ i+1].name;
    //Blob has ended if next minute doesn't have same player in it or next minute is unassgined
    if (isMinEmpty || (!isNextMinSame && !isMinEmpty) || i == interval_length-1)
    {
      //Add the blob to the blobbed data and reset the length of the blob
      blobbed_data.push({name: globalState.position_data[0].position_timeline[i].name, length: blob_length,color:
globalState.position_data[0].position_timeline[i].color})
      blob_length = 0
    }
  }
  return blobbed _data
}
```

With the blobbing function able to return blobbed data I now I had to map this data to be rendered. These rendered components are stylized view components with their width equal to their length value times the minute width constant that I calculated by getting the PositionSlider width and dividing it by the number of minutes displayed. I also set the colour of the view to transparent if no player was selected at this minute to prevent the RNPickerSelect being covered up.

```
<View style = {{position:'absolute',flexDirection:'row'}}>
  {blob_data_for_visual_display().map((prop,index) => {
    return(
      <View key = {index}  style = {{...styles.sliderBox, width: interval_width*prop.length, backgroundColor:(prop.name ==
null? 'transparent':prop.color)}}>
        <Text style = {styles.sliderText}>{prop.name}</Text>
      </View>
  )})}
</View>
```



Updated visual layer after carrying out the blobbing process.

The Drag Layer of the PostionSlider

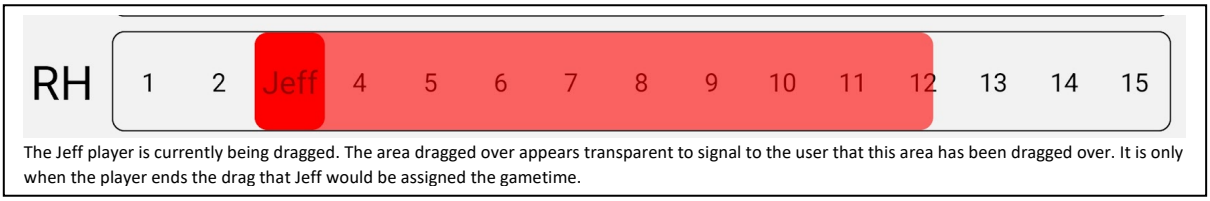
The final layer to add was the drag layer. Currently minutes could only be assigned through selecting players on the RNPickerSelect and not by dragging. With the drag layer I had to keep all logic when the drag was active to a minimum to overcome the lag related issues of the last iteration. Therefore, I decided to make the drag layer purely visual in the sense that it existed as an overlay to show the user where they had dragged, and only after the drag had finished would I carry out the logic of updating the minutes assigned. The first step in doing this was setting up the visual part of the drag layer. First, I setup a dragBar state variable that was a list of three objects. All objects had a start and end field representing the x coordinates of the position of the bar. This dragBar variable was defined in dragStart, updated in dragActive and set to null in dragEnd. When rendering this dragBar I first used conditional rendering to render only when the dragBar is active. I then rendered three views with the middle representing the current drag happening. Also note that in the styling of the middle view that represented the drag the background colour was the same as that of the player blob, but it had a reduced opacity to give it an overlay feel.

```
{/* Checks if drag bar is active and if so draw three views one for the dragbar and the other two for either side */}
{(dragBar != null)?
  <View style = {{position:'absolute',flexDirection:'row'}}>
      <View style = {{ width: (dragBar[0].end-dragBar[0].start),opacity:0,height:100}}/>
      <View style = {{...styles.dragBar,backgroundColor: globalState.position_data[0].position_timeline[startMinute].color,
width: dragBar[1].end-dragBar[1].start}}/>
      <View style = {{width: dragBar[2].end-dragBar[2].start,height:100}}/>
  </View>: null}
```

With the frontend setup I then dealt with the back-end implementation. First with the dragStart function. Like the previous iteration of the PositionSlider for the drag to work we need to know what direction the drag is going. The difference is that in the previous iteration we knew the length of the player objects, and it was, therefore, easy to determine the halfway point of the object. However, in this iteration we don't know the length of each player object as they exist across multiple minutes. Therefore, we must find how many minutes that player spanned then find the middle of those minutes. We could then find the x coordinate of this minute and determine whether the drag started on the right or the left of that value. The start minute of the drag is also stored on the dragStart function so the visual overlay of the drag can be set to the colour of that minute.

Next on the dragActive. I fully cut back on the logic in this in comparison to the previous iteration. The only code was updating the dragBar variable so that the drag bar reflects the movement of the user's finger.

```
const dragActive = (drag) =>
    {
      //Move direction is to the right so right end moves while left end is fixed
      if (moveDir == 'right')
      {
          setDragBar([{start: 0, end: dragBar[0].end},
                      {start: dragBar[1].start, end: drag.nativeEvent.x},
                      {start: drag.nativeEvent.x, end:screen_width}])
      }
      //Move direction is to the left so left end moves while right end is fixed
      else if (moveDir == 'left')
      {
          setDragBar([{start: 0, end: drag.nativeEvent.x},
                      {start: drag.nativeEvent.x, end: dragBar[1].end},
                      {start: dragBar[2].start , end:screen_width}])
      }
    }
```



The Jeff player is currently being dragged. The area dragged over appears transparent to signal to the user that this area has been dragged over. It is only when the player ends the drag that Jeff would be assigned the gametime.

Finally, on dragEnd. It was only when the drag is completed that I updated the position_timeline to reflect the changes in times allocated to a player. First, I found the index of the minute the drag ended on by rounding the user's finger to the nearest minute. Next, I had to consider the direction of the drag this was because if I dragged to the right, I would iterate through the list forwards from the start minute to the end minute changing values but if I dragged to the left, I'd want to iterate from the end minute to the start minute to update values. Within the directions I also considered the cases of whether the drag overlaps adjacent minutes – in which it changes the value of these minutes to the same as its own – or if the drag pulled back – in which I would set the minutes pulled back to null to be empty.

```
//This code is only for if the moveDir was to the right. There is
similar code for moving to the left

const endMinute = Math.round(drag.nativeEvent.x / interval_width)

if (moveDir == 'right')
{
    if(endMinute <= startMinute)
    {
        for (let min = endMinute; min <= startMinute; min ++)
        {
            updatePosition([min,null,'CF'])
        }
     }
     else
     {
        for (let min = startMinute; min < endMinute; min ++)
        {
            updatePosition([min,globalState.position_data[0].posi
tion_timeline[startMinute].name,'CF',globalState.position_data[0]
.position_timeline[startMinute].color])
         }
     }
}
```

Final Visual Touch-Ups to Position Slider

The PositionSlider was now functionally complete, and I could add as many players as I wished and drag them forwards and backwards to give them more time. It was at this point I made a stylistic change to allow users to differentiate between players. As you can see in all code excerpts the colour of Views are set by the value of the colour field of the object at the given minute in the position_timeline. I initially always set this colour field to be red, but I recognized as time progressed the need for each player on the PositionSlider to have a unique colour. To implement this, I made use of a script to generate a random hex value then assigned it as the player's object colour. I then wrote a function that retrieved the colour of a player given its name. This assignColor function was then called every time a player was added to the position_timeline to set the colour field of that minute to be the same as their colour. Note I did realize that I shouldn't be joining on a name and instead should join on a common key and will discuss fixing this later.

```
const assignColor = (name) =>
    {
      //Null check prevents unassigned time from being given a color
      if (name != null)
      {
        //Join on the name of the two lists and return the color
        let join = globalState.player_data.find(player => player.name == name)
        return join.color
      }
    }
```

For this position George, Bob and Kyle have been assigned gametime. Their randomly generated colour allows us to distinguish between them

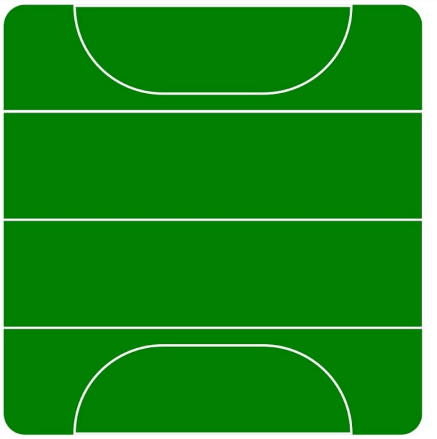
Development of Formation Selection Screen

Different coaches make use of different strategies in a sports match. What underpins these strategies is the formation of players on the pitch. To cater to these different coaching styles, it is necessary to implement a way for users to have choice in which formation they use. My solution to this is to implement a formation selection screen where users can choose from a range of predefined formations. By implementing this I would increase the accessibility and use of the app as coaches should be able to find a formation that caters to their coaching style. Additionally, on the subsheet creation screen all the available positions that appeared were arbitrary. By allowing users to select a formation the positions of that formation can then appear on the subsheet allowing users to assign time to a player at a certain position. The implementation of a formation selection screen had two steps. Firstly, the visual implementation followed by the back-end implementation.

Creating a Visual Game Pitch

First, I wanted to create a visual representation of a hockey pitch that would be the background for the formations. To do this I made use of View components with flex styling to create an image of a hockey pitch. I used this method as opposed to a static image because flex styling allows the hockey pitch to resize to a range of screen sizes and still resemble a hockey pitch. I also set the position of it to be absolute to allow the positions to be overlayed on top.

```
{/* INLINE STYLING IS DELIBERATE TO SHOW HOW FLEX STYLING WORKS */}
<View style = {styles.gamePitch}>
    {/* Top quarter */}
    <View style = {{flex: 1,flexDirection:'row', backgroundColor:
'green',borderBottomWidth:2,borderColor:'white'}}>
        <View style = {{flex:1, backgroundColor:'green'}}></View>
        <View style = {{flex:4}}>
            <View style =
{{flex:17,borderRightWidth:2,borderLeftWidth:2,borderBottomWidth:2,border
Color:'white', borderBottomEndRadius:150,borderBottomStartRadius:150}}/>
            <View style ={{flex:3}}/>
        </View>
        <View style = {{flex:1, backgroundColor:'green'}}></View>
    </View>
    {/* Middle half of field */}
    <View style = {{flex: 1, backgroundColor:
'green',borderBottomWidth:2,borderColor:'white'}}/>
    <View style = {{flex: 1, backgroundColor:
'green',borderBottomWidth:2,borderColor:'white'}}/>
    {/* Bottom quarter */}
    <View style = {{flex: 1, backgroundColor:
'green',flexDirection:'row'}}>
        <View style = {{flex:1, backgroundColor:'green'}}/>
        <View style = {{flex:4}}>
            <View style = {{flex:3}}/>
            <View style =
{{flex:17,borderRightWidth:2,borderLeftWidth:2,borderTopWidth:2,borderCol
or:'white', borderTopLeftRadius:150,borderTopRightRadius:150}}/>
        </View>
        <View style = {{flex:1, backgroundColor:'green'}}/>
    </View>
</View>
```



Rendered game pitch from code to the left

Implementation of Positional Grid System

With a pitch rendering we know wanted to render the positions on top of it. What I wanted was to have a circular icon for each position. These icons would have the initials of the position and be placed in the position of that position on the hockey field. For example, a circular icon with 'GK' on it is by the goal. This could be done by using absolute styling to set each of the position icons at a fixed x and y value, however this would not effectively scale to different screen sizes. Therefore, I decided to make use of a responsive grid formation. I defined the hockey pitch as a 7x7 grid which I represented with a 2d array. I then set the array value where a position was to the position initials and all other values to 0 to represent it being empty.

```
[[0,0,0,'CF',0,0,0],
[0,0,'LF',0,'RF',0,0],
[0,'RI',0,0,0,'LI',0],
[0,0,0,'CH',0,0,0],
['LH',0,0,0,0,0,'RH'],
[0,0,'CB',0,'CB',0,0],
[0,0,0,'GK',0,0,0]]
```

Array for 3-2-3-2 formation

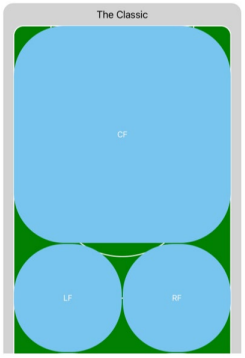
The next step was rendering this grid as icons. To do this I used a double map method. Firstly, I mapped the grid so that I had access to each row of data. I then mapped each of these rows of data to View component, causing a View to be rendered for every item in the 2d array. I then used conditional styling to make values of 0 in the array not appear by setting their opacity to 0. For values that contained a position I applied styling

to its View component to make it appear as a circular icon. Additionally, all the View components – irrespective of being transparent or not - had a flex value of 1. This made all the icons the same size.

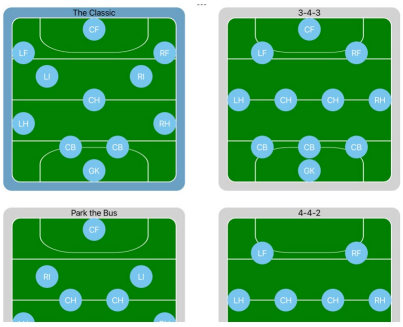
```
{layoutData.map((prop,index) => {
    return(
        // Row data is remapped
        <View key = {index} style = {{flex: 1,flexDirection:'row'}}>
            {prop.map((prop,index) => {
                //All view component need to be rendered due to how the positioning of them is dependent on flex styling.
However we don't want all icons to appear so a transparency is applied if they should not appear
                let opacity_ = 1
                if (prop == 0)  opacity_ = 0
                return(
                    <View key = {index} style ={{...styles.iconBodyStyle, opacity: opacity_}}>
                        <Text style = {styles.iconText}>{prop != 0 ? prop[1]:''}</Text>
                    </View>
                )
            })}
        </View>
    )
})}
```

I next needed to scale this up to allow for the rendering of multiple formations at once to allow the users to select a formation. To do this I setup a list for all possible formations. I then made formation objects for each formation with fields of formation name, formation layout (that being the 2d array) and formation id. With this setup I then used a Flatlist to render the whole list of formations. I consulted with my Dad (he used to select the Black Sticks so he probably knows a thing or two about hockey) about basic hockey formations and he gave me a range formation to add such as 3-2-1-2, 3-4-3, 1-2-4-3 and 4-4-2.

Formation that is rendered from passing in the array above



What happens to formation if unrendered positions don't have a flex value of 1.



Scaled up formation selection. The currently selected formation is highlighted in blue

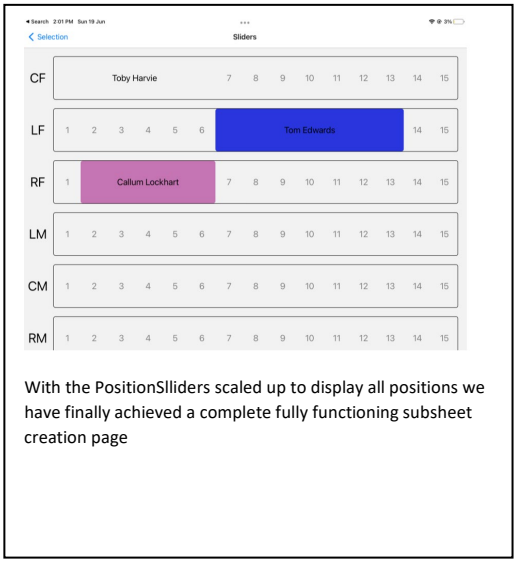
With all the formations displayed I now needed to add the ability for users to select a formation. To implement this, I created a state hook that stored the id of the currently selected formation. I then wrapped all the formation components inside of a Pressable. When a Pressable was tapped I updated the state hook to be equal to the id of the tapped formation. I then used conditional styling to make the formation that had an id the same as the selected one appear with a blue background, indicating has been selected.

Connecting Formation Selection Page with the Rest of the App

The final task was to connect the formation selection page up with the rest of the app. I started by adding it to the stack navigator and set it as the default page as I wanted users to first select a formation before adding positions to their players. I then added a Pressable to the formation selection page that signified the completion of a selection. When this Pressable was pressed multiple things happened. Firstly, I transformed the data from the formation into data to appear as PositionSliders on the subsheet creation page. To do this I iterated through the formation data and created an object for each of the positions in the formation. All these position objects were added to a list. I then made a reducer to update the position_data to be the same as this list. Finally, I called a navigation event to move the user to the team managment page. With this implemented, when a user gets to the subsheet creation there is a PositionSlider for each of the positions in the formation they selected. I also rendered the initials of each position next to their respective PositionSlider to clearly indicate to users which position was which on the subsheet. With this complete, the subsheet

creation page was now finished. Additionally, I also implemented an alert that would prevent the users from progressing if no formation was selected.

```
///positionData is the formatted data. Formation data is the 2d array for the
currently selected formation
let positionData = []
let formationData = formationDataAll[selectedFormation].formation_data
let index = 0;
//Iterate through all values of the 2d formationData
for(let rows = 0; rows < formationData.length; rows++)
{
    for(let columns = 0; columns < formationData[rows].length; columns++ )
    {
        //Check if a position exists at that place
        if(formationData[rows][columns] != 0)
        {
            //Create a new position object for that position
            positionData.push({
                position_id: index,
                position_inititals: formationData[rows][columns][1],
                position_timeline: new Array(15).fill({name:null, color:null})
            })
            //Increment the index to ensure all positons have unique id
            index += 1
        }
    }
}
//Reducer that is called to update redux store of positonData
updatePositionData(positionData)
```



With the PositionSliders scaled up to display all positions we have finally achieved a complete fully functioning subsheet creation page

The final change I made was to the team management page. I no longer needed to arbitrarily set the selectable positions that could be added to a player. I could instead iterate through the position_data list adding a selectable position for each of the positions in the selected formation.

```
let positionSelectionData = []
//Iterate through the position_data
for(let i = 0; i < position_data.length; i++)
{
    //Format the data in the accepted form of the RNPickerSelect
    let formattedData = {label: position_data[i].position_name, value: position_data[i].position_inititals}
    //Check if the position is not already in the list as some formations have for example two CB
    if(!positionSelectionData.some(formattedData => formattedData.label == position_data[i].position_name ))
    {
        positionSelectionData.push(formattedData)
    }
}
```

Development of the Game Overview Screen

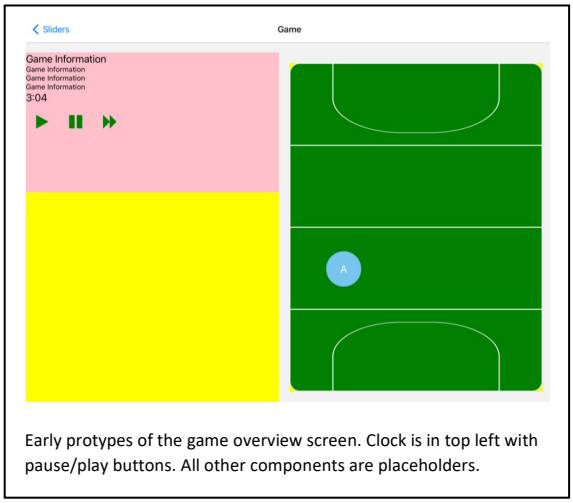
The game overview screen was the final major part of the planned MVP I agreed upon with A[redacted] K[redacted]. On this page it would show upcoming subs and the active game situation. By players being able to view the upcoming subs during the match and when they occur they are able to carry out their substitutions without the help of a coach. This allows young players to develop self-management skills and the quality of coaching to increase as the coach no longer needs to micromanage subs.

Implementing a Game Clock

The functionality of being able to see upcoming substitutions and the active game situation is reliant on knowing the current time on the games clock. No clock existed therefore I had to create my own. The first step of creating the game clock was creating state hooks for current minute, current second and whether the game was paused (I added a pressable to toggle the pause variable). Next, I began researching how to setup a clock system. What I found was that it was quite easy to cause a memory leak by causing an infinite render cycle therefore I had to be quite careful in how I approached this. What I decided to use was setInterval. setInterval takes two arguments, the first a function that you want to run every interval and the second the interval length in ms. I next put in a clearInterval function that was necessary to clear the interval every render cycle to prevent a memory leak happening. However, when I went to run this code a memory leak occurred, and the timer increased at an inconsistent rate. This initially perplexed me, but I soon realised that this was occurring because every render cycle an interval was being set. When this interval finished the app re-rendered causing a new interval to be set. This caused after a period an infinite number of intervals to be active. To prevent this, I had to put the setInterval inside of a useEffect hook. A useEffect hook is traditionally used to siderun code every render cycle, however it can take a second argument called a dependencies array. In this dependency array are variables that must change for the useEffect to be called. By setting the

dependency array to have values of timerActive (that changes every time the user unpaused or paused the game), the interval is only set when this happens preventing the memory leak from infinite render cycles occurring. Once I had updating time variables, I formatted these using Regex to make it appear nice and then I rendered it a text component.

```
useEffect(() => {
    if(timerActive)
    {
        const interval = setInterval(() => {
            //Update time related variables
            let updateMin = false
            setSecond(seconds => {
                if(seconds == 59)
                {
                    updateMin = true
                    return 0
                }
                else
                {
                    return seconds+1
                }
            })
            if(updateMin)
            {
                setMinute(mins => mins+1)
                //Set timer to false if end of interval has been reached
                if((minute+1) == intervalLength)
                {
                    setTimerActive(()=> false)
                }
            }
        }, 1000);
        //Interval must be cleared to prevent memory leak
        return () => clearInterval(interval);
    }
//Dependecies array means useEffect is ran everytime timerActive
}, [timerActive]);
```



Early protypes of the game overview screen. Clock is in top left with pause/play buttons. All other components are placeholders.

Connecting the Subsheet Creation Page with the Game Overview Page

With the timer setup I could now add the two pieces of functionality that are dependent on the timer. That being the substitutions countdowns and the active game situation. For this to work we needed to get relevant data from the completed subsheet. For the sub countdown we would need the minute each substitution occurs and who is subbing on for who. For the active game situation, we would need the same data plus the coordinates of each position in the 2d formation array so it could be overlayed on the game pitch like on the formation screen. To get this information I wrote a function that is called when a user finishes their subsheet. The function creates sub_data by iterating through all the position_data to find occurrences of where the player in a position changes. These substitutions and their relevant data are then saved as objects in a list. This subdata is then passed through a reducer to be saved to the global store variable sub_data so it can be accessed on the game overview screen.

```
let subData = []
let subId = 0
//Loop through all of the positions
for(let position =0; position< globalState.position_data.length; position++)
{
    //Set up constants to improve readability of code
    const positionTimeline = globalState.position_data[position].position_timeline
    const positionInitials = globalState.position_data[position].position_inititals
    const positionCoordinates = globalState.position_data[position].position_cordinates
    let priorPerson = positionTimeline[0].name
    for(let min = 0; min < positionTimeline.length; min++)
    {
        //Check wether player has changed if so sub has occured
        if(priorPerson != positionTimeline[min].id)
        {
            subData.push({subId: subId, subMin: min,subPlayerOn:priorPerson,subPlayerOff:
positionTimeline[min].name,subPos:positionInitials,subCords: positionCoordinates})
            //Sub id used so all subs have unique id
            subId ++
        }
        //Set the id of prior person to the current person so on the next ieration its the prior person
        priorPerson = positionTimeline[min].id
    }
}

//Save game data through reducer
createSubData(subData)
```

Substitution Countdown Component

On the substitution countdown component, I needed to communicate who's currently on the field, who's subbing onto the field, the time in the match this substitution occurs and the position it is occurring at. I created a component that displayed the position of the sub and who's subbing. I next had to find how long until the sub occurs. To do this I found the difference in minutes and seconds between the time of the substitution and the current time then formatted that time. I then set up a FlatList with this component as the renderItem and the data as the sub_data as a result it rendered all the upcoming subs in a list.

```
//Determine the diffrence in time between the game time and
the given time for the sub
let minToSub = subMin - minute
let secToSub = 60-second

//If the secs are 60 make it equal to 0 if not minus 1 from
min to sub as you are cutting down
if (secToSub == 60)
{
    secToSub = 0
}
else
{
    minToSub -=1
}
const formattedSubTime = '-
'+(minToSub+':'+secToSub.toString().padStart(2,'0'))
```

Above is the subsheet used. Below are the upcoming subs based on this data. At this point the game clock was at 12 seconds complete.

Active Game Situation

Next was to show the current state of the pitch and who was currently on the field. To do this I wrote a function that would return a 2d formation array – which could then be overlayed on the game pitch – given a minute of the game. The function took a minute as a parameter and then iterated through the data of all positions at that given minute. It would then retrieve the coordinates value of the position on the pitch and the name of the player at that time. With this data it would set the value of a pitch 2d array to the players name of the player on the pitch at those cordinates value. Once it had iterated through all these values it would return a complete pitch 2d array. I called this function every time the minute was incremented by the clock and saved its return variable in a gamePitch state hook. I would then pass the data in the gamePitch hook into the GamePitch component I used for formation selection. This would result in a game pitch appearing with the players' names in the positions they are in.

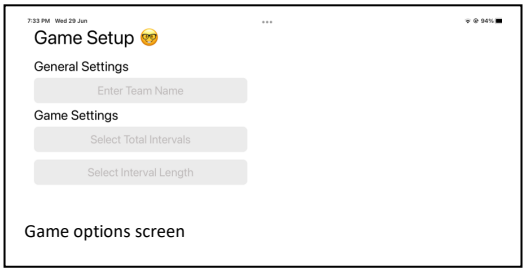
Screenshot of complete game overview screen from the first technical test of the app

Implementation of the Final Minor Features of the MVP

At this point the app now resembled the MVP I planned with A[redacted] K[redacted]. However, there were a few rough edges I had to smooth out before the first technical test, such as adding an options screen and the ability to save subsheet.

Option Selection Screen

Currently the number of minutes in each subsheet was hardcoded at a value of 15. This worked fine for testing the apps features internally. However, as I began to move towards carrying out tests with teams, I would have to consider the difference in game lengths between teams dependent on their sports and grades ie 7 aside hockey plays 45 minutes games, senior football plays 90 minute games. A second consideration I made was whether it would be feasible to fit a full 90 minutes onto a subsheet. I tested this out and if the minutes on a subsheet became too large it became impossible to assign a player anytime as the RNPickerSelect for each minute became too small to tap. Based on this I planned to implement a way to display only one interval at a time on a subsheet and allow users to toggle



Game options screen

between these intervals. This would mean only 45 minutes at a time would be displayed for a 90-minute football match making the minute RNPickerSelects easier to tap.

To account for these considerations, I decided to make an options screen that appears as the first page of the app that allows users to select the number of intervals (2 – halfs, 3 – thrids and 4 - quarters) and length of the intervals in their game. I used a RNPickerSelect to allow users to select the number of intervals and interval length. When they selected a value, I called a reducer I wrote to save the number of intervals and length of intervals to global redux state variables. In addition to this I added a TextInput that allowed users to enter their team's name.

Implementing Multiple Intervals on the Subsheet

The next step was accounting for the existence of multiple intervals in my code. Currently if a user created a 4 interval 15-minute game, the subsheet screen would display a 60-minute long subsheet as opposed to 4 individuals 15 minute subsheets that could be toggled between. The first step in making this transition was setting up a toggle that allows users to select an interval to display. I first setup a displayed interval state hook, then created a toggle that allows users to change this displayed interval. To make this toggle I mapped an array of the length of total intervals and rendered a pressable for each item in the array. I then updated the displayed interval when these Pressable's were pressed to their respective interval before finally applying conditional styling to make it obvious what the currently displayed interval was.

```
// Renders an item for each item in the list. In this case the length of the list is the same as ammount of intervals
{[...Array(gameData.total_intervals)].map((prop,i) => {
    //i+1 is used as intervals dont at 0
    const interval = i+1
    //Conditonal styling for if selecetd
    const color = ((interval == gameData.current_interval)? '#95b7ed' : 'transparent')
    const textColor = ((interval == gameData.current_interval)? 'black' : 'white')
    return(
      <Pressable key = {i} onPress={()=>{updateCurrentInterval(interval)}} style = {{...styles.intervalButton,
backgroundColor:color}} >
        <Text  style = {{...styles.intervalText,color:textColor}}>{interval}</Text>
      </Pressable>
    )
})}
```

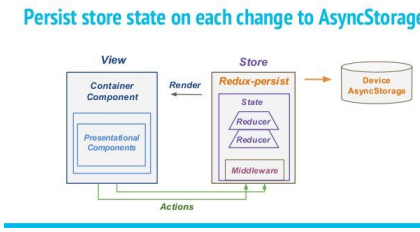
Interval toggle for game with thirds

Now with the ability to toggle between intervals to display, I had to reflect this change visually. To do this I had to implement offsets in the PositionSlider such that the part of the position_timeline that was being rendered reflected the current displayed interval. After doing this I set up intervals on the game overview page by adding a state hook for played intervals. At the end of every interval the clock would pause, and the time variables would reset, signifying the end of the interval.

```
//EXAMPLES OF CHANGES MADE IN CODE TO ACCOUNT FOR MULTIPLE INTERVALS
//Condtional rendering on the component layer to only render the current itnerval to the screen
{positionTimeline.map((prop,i) => {
    if(i >= (currentInterval-1)*intervalLength && i < ((currentInterval)*intervalLength) )

//Start tile of drag being offset by the current interval
const dragStartTile = Math.floor(drag.nativeEvent.x / positionIntervalWidth) + intervalLength*(currentInterval-1)
```

Save System and Selection of Persistent Database



Data flow with Redux Persist

The next thing to set up was a save system. It was necessary that users would be able to save their subsheet so they could: a) use them again in the future, meaning they only must set up a subsheet once, and b) set subsheets up before a match. I first had to decide on how I was going to store this data. Two constraints guided my decision-making process on a database. The first was that the database had to have offline only support. This was for two reasons. Firstly, most people would be using the app out on a sports pitch where Wifi was inaccessible. Secondly, online databases are a lot more work to setup and would require me to make use of external APIs and tools such as AWS. The second constraint was that the database had to have object support or be JSON based. This was because I didn't want to have to setup a serialization process to change my data stored in objects to another form. These two constraints eliminated basically all databases apart from Redux Persist. Redux persist takes your redux state object and saves it to a

persisted storage. Then, on app launch it retrieves this persisted state and saves it back to your Redux store. Redux Persist is commonly used in combination with an online database as a failsafe in case the user can't access Internet. However, it works as well as an offline-only database. The major plus of Redux Persist is that it only would take me at most five lines of code to implement as I already had Redux setup.

```
//Setup redux persist
const persistConfig = {key: 'root', storage: AsyncStorage, whitelist: [] };
const pReducer = persistReducer(persistConfig, rootReducer);
const middleware = applyMiddleware(thunk);
const store = createStore(pReducer, middleware);
const persistor = persistStore(store);
```

With this set up I had to whitelist a specific reducer in the persistConfig to persist the state off. I decided that I wouldn't preserve the state of the whole app. Instead, I would make a new redux state variable called save_data. I would save all subsheets and players to this as an object then persist this state so users can access their saved subsheet later. The first thing to setup was saving the data. An autosave system that saved periodically would be too memory intensive as it would have to do constant comparison between the state to save and saved states to determine whether a save would be necessary. Instead, I made saving occur at two points. Firstly, I added a save button to the subsheet management screen, that when pressed the current subsheet and players were saved. Secondly, when the match begins, I would save the subsheet and players used in that match. When the save was triggered, I would call a reducer to add a new item to the global save_data list.

Loading the Saved Subsheets

With the save data now being stored, I now had to implement a way for users to load their saves. To do this I first set up a home page. On this home page there were two options for users. The first was to create a new subsheet that took users to the subsheet option page to begin their subsheet creation and the second was a button that took users to a load save page. On this load save page I created a SaveView component which displayed the details about a save. I then created a Flatlist that rendered a save view for each of the saved subsheets.

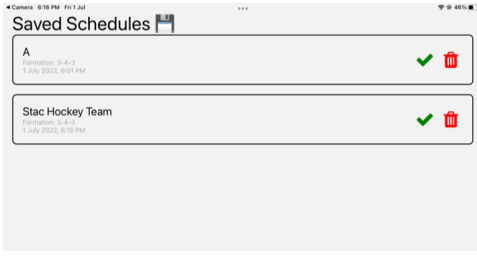
```
//Saveview component displays the name of the subsheet formation used and time it was saved.
<View style = {styles.saveView}>
    <Text style = {styles.titleText}>{item.save_name}</Text>
    <Text style = {styles.subText} >Formation: {item.save_positionsData.formation_name}</Text>
    <Text style = {styles.subText} >{format_time()}</Text>
</View>
```

Next, I added two Pressable's to the SaveView component. The first had a tick icon that when pressed loaded the subsheet. To load the subsheet I wrote reducers that overrode the current player and position data (the displayed subsheet) with the loaded data. After that it navigated users to the subsheet creation page with all the save data loaded. The second pressable had a trash can and when pressed called a reducer that deleted the save data.

```
//Adjusted index is set to prevent indexing errors that may occur due to
subsheets being deleted
const adjusted_index = savedState.save_data.findIndex(item => item.save_id == i)
//Update the player,position and game  data
uploadPlayerData(savedState.save_data[adjusted_index].save_playerData)
uploadSubsheet(savedState.save_data[adjusted_index].save_positionsData)

//Load the page that the sliders is on
navigation.replace('SubsheetCreation')

//Reducer that deletes the save data at a given index
case DELETE_SAVE_DATA:
    return{...state,save_data: state.save_data.filter(item => item.save_id !==
action.payload)};
```

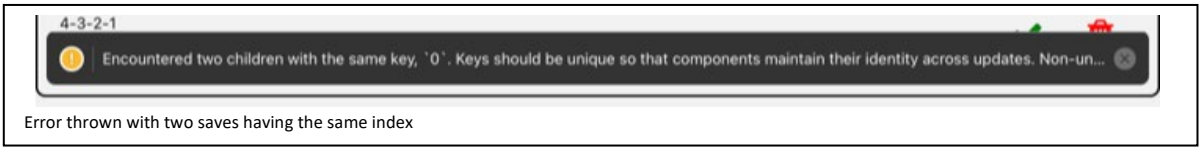


Multiple SaveViews displayed on the save schedule

Bugs Caused by Asynchronous Logic

The second bug was that if a user rapidly created players or saved the game, multiple objects with the same id would be saved to their respective data structures as the indexing key would not increment in between taps. This would cause indexing errors to be thrown and glitches to occur due to them all having the same id. After a lot of thinking and debugging I found the issue to be in the asynchronous nature of state hooks. The indexing key I was incrementing for the id of the player objects was a state hook and when I incremented it, it updated asynchronously. This meant that there would be a delay between the update being called and the

value changing due to the nature of asynchronous logic. This meant when tapping rapidly you would use the non-updated version of the index key resulting in multiple objects in the list having the same id. To address this, I first set up a state variable for whether the button was tapped and then a useEffect hook. I put the code of adding the object to the list as the function of this hook and the state variable in the dependency array. The use of the useEffect meant that only once per render cycle was the code ran after canAddPlayer was update and therefore multiple objects couldn't be added with the same index.



Error thrown with two saves having the same index

```
useEffect(() => {
    //We check if canaddplayer is true as after adding a player it is set to false causing the useffect hook to be called a
second time unwanted
    if(canAddPlayer)
    {
        createPlayer({id: playerState.player_index,name: '',positions: [],selectedPos: null,
                      color: '#' + Math.floor(Math.random()*16777215).toString(16)})
        //Increment the player index
        incrementPlayerIndex(1)
        setCanAddPlayer(false)
    }
//CanAddPlayer is set to true when the plus button to add a player is pressed, this code then runs at the first render cycle
following that
},[canAddPlayer]);
```

Refinement Process

It was now the last weekend of term two. I had achieved my planned minimal viable product and had a few weeks until the start of term three when I planned to roll out *SUBlime* to a large testing base. During the refinement process I planned to carry out a technical test to determine if the app would work with widespread testing without crashing, get in contact with more stakeholders to widen testing base and address any glaring design and technical issues and finally add some outstanding features such as the gametime overview, which I had put aside earlier in favour of completing the MVP so a technical test could be carried out before the end of the term.

First technical test

Goal of first technical test

With this technical test I wanted to try the app out in a low stakes game environment with the purpose of determining whether the app would work technically across a whole game and would therefore be ready for wider scale testing. For this testing I reached out to Jakarta Klebert, a friend of mine, captain of the 1st XI hockey team at [redacted] college and a junior hockey team coach. As this team was junior and very social Jakarta said he wouldn't mind if the app didn't work.

Beyond the technical aspects this test had a few other goals. Firstly, I wanted to observe how intuitive and easy the app was to use by handing Jakarta the app and letting him try and set it up without me giving him pointers. This could reveal any UX issue. Secondly, I wanted to collect feedback from Jakarta and get his suggestions for improvement. And finally, I wanted to see how the players in the team interacted with the app.

Setting up App for 7 Aside hockey

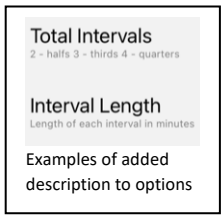
Prior to the testing on 2nd of July I had preliminary discussions with Jakarta surrounding game formations in 7-aside. Prior to this I had no formations setup in the app for 7-aside. Through discussion he introduced me to several 7-aside structures that he had used or seen being used in competition. All the formations were added to the app's list of formations.

Results of Technical Test

Technically the testing of the app was largely a success. The app did not crash, and Jakarta was able to create a subsheet and play it out without any issues. Two minor technical issues did arise. The first was around the iPad screen timing out causing the screen to go to black. When this happened the app clock no longer progressed causing the time on the app to become desynchronized from the timer on the scoreboard. This

was a frustrating bug to fix as I was unable to replicate it in my Expo testing environments and it only arose in live environments. The solution I found to this was using the Keep Awake library[20]. This library allowed me to make use of a keepAwake state variable that I set to true whilst the game was running. This caused the screen to stay awake and the desync to not occur. The second technical issue was to do with the performance of the PositionSliders, I will touch on my solutions to this later in the report.

Observationally I noted a few things when Jakarta used the app. The first was that Jakarta found some of the settings in the game setup confusing, reflected by him asking me what they meant, so I remedied this adding a brief description to all setting categories that could be considered vague. The second thing I observed was that the PositionSliders were quite unresponsive to Jakarta's touch, and he had to press them multiple times before they allowed him to drag. This was something I made note of to improve when I inevitably overhauled the PositionSliders for the third time. The feedback I received explicitly from Jakarta largely focused on how long the game subsheet took to setup. Jakarta suggested a solution that a new setting should be added that allowed all the game intervals to be the identical in when substitutions occur. He thought this would be a good idea as coaches often have the same substitution patterns across intervals. This meant a coach only had to set up the subs for a single interval as opposed to three or four. I also made note of this for my PositionSlider overhaul.

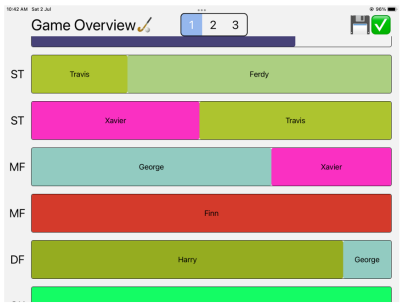


Examples of added description to options

The interaction of the players in the team with the app exceeded my expectations and was the highlight of the first test. Throughout the game the currently substituted players would go and check the iPad every 30 seconds to see how long it was until they were subbed on (even though their substitution may be four minutes away). It also led to some awkward questions being asked about Jakarta's substitution practices when players realised a player was barely getting subbed in comparison to everyone else. A few players also helpfully pointed out a bug on the game screen that caused the names to be displayed the wrong way when a sub was occurring which I quickly changed.


Expensive iPad I am borrowing resting above a metre drop onto concrete


Players checking out the iPad during the game



The setup subsheet for the game

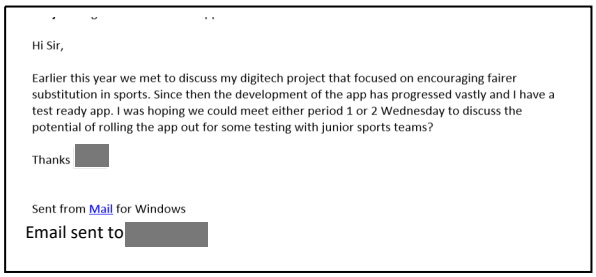
Firewall related issues at school

The momentum of the first successful test did not last long as the school IT department unintendedly massively impede the development of my project. On my return to school on Monday I was met with an update to the school firewall. This update prevented Expo from working meaning I couldn't test my app at school. I went up to the schools IT department to see if they could provide a work around but unfortunately, they were unable to help. Therefore, I changed how I managed my time spent on the project. While I school, I worked on the write up and started getting in contact with more stakeholders for testing. At home where there was no firewall, I focused on developing the app.

[20] https://docs.expo.dev/versions/latest/sdk/keep-awake/

Third meeting with A K

This meeting with A K came in the final week of Term 2. It had been a while since our last meeting therefore I first focused our discussion on what he thought of the state of *SUBlime*. He stated he was happy with the functionality provided by all the features and found the process of setting up a subsheet for a game intuitive. He noted a lack of polish when it came to the UX and design of the app which I largely agreed with. The largest criticism came against not what existed in the app but what was missing. He noted the lack of ability to see the total time for each player in a game a feature he had stressed the importance of since our first meeting. I agreed with him on this criticism, and I explained that I had pushed this feature aside to prioritize getting the features of the MVP done for a technical test.



Hi Sir,

Earlier this year we met to discuss my digitech project that focused on encouraging fairer substitution in sports. Since then the development of the app has progressed vastly and I have a test ready app. I was hoping we could meet either period 1 or 2 Wednesday to discuss the potential of rolling the app out for some testing with junior sports teams?

Thanks [redacted]

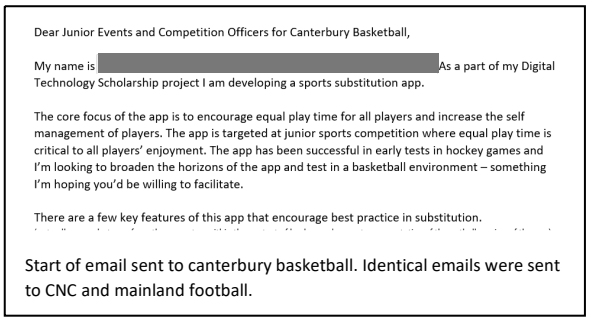
Sent from Mail for Windows

Email sent to [redacted]

The second part of the meeting focused on testing. A K wanted to hold out on testing until the player time overview feature was implemented. This was frustrating as I was hoping to get the testing underway as soon as possible. However, this frustration quickly faded as I realised there was no sport for the next two weeks due to the school holidays. As far as testing went A K was willing to support a testing model of sending out the app to coaches to download alongside a survey link for feedback. This was extremely helpful to me as it meant I could gain access to a large testing base. Additionally, he was happy to send out the app to a wide range of coaches from all of the core winter sports (Football, Hockey, Rugby and Netball). Finally, we planned to meet again on the first week back at school for term 3 to plan to roll out testing.

Contacting Regional Sports Governing Bodies

Whilst being in discussions with A K about rolling out testing with StAC prep school sport teams I also got in contact with Netball Christchurch, Mainland Football and Canterbury Basketball through email. I had held off contacting them until now, as they would likely only be willing to consider discussing the app if it was functionally complete and had prior success in testing. I had now reached this point. The purpose of working alongside them was to help me adapt my app to their respective sports code (through their knowledge of that sport) and distribute the app to the many junior sports teams they oversaw. These groups each oversaw upwards of 100+ junior teams respectively, so this could be a huge testing opportunity. I sent out a basic email with a description of the app asking for a time to meet and discuss.



Dear Junior Events and Competition Officers for Canterbury Basketball,

My name is [redacted] As a part of my Digital Technology Scholarship project I am developing a sports substitution app.

The core focus of the app is to encourage equal play time for all players and increase the self management of players. The app is targeted at junior sports competition where equal play time is critical to all players' enjoyment. The app has been successful in early tests in hockey games and I'm looking to broaden the horizons of the app and test in a basketball environment – something I'm hoping you'd be willing to facilitate.

There are a few key features of this app that encourage best practice in substitution.

Start of email sent to canterbury basketball. Identical emails were sent to CNC and mainland football.

Further Iteration on the PositionSlider Component

Although the PositionSlider had already been through multiple iterations, testing within my class and with Jakarta showed that there was still need for further improvement.

Optimizations to Improve PositionSlider Performance

In my technical test with Jakarta the performance of the PositionSliders was still not the best. Although this lag had improved over prior iterations of the sliders system the drag was still quite laggy. This lag slowed the process of creating a subsheet which created a barrier towards users using the app. To analyse these performance issues, I installed the React Native debugger and ran it whilst I carried out a drag motion to determine what was using a high amount of memory. What I found was that all the RNPickerSelect's were being re-rendered every render cycle during the drag. The re-renders were being caused by the dragBar state hook being updated forcing a re-render. This large number of re-renders especially in subsheets with longer intervals was tanking the performance. When addressing this issue, I was limited in what I could do as I was unable to change the fact that the dragBar value changes, because without it changing no drag would be visible. Therefore, I had to stop the re-renders occurring on unchanged components. To do this I made use of the useMemo hook. The useMemo hook memoizes a function, caching it such that it only re-renders when the component state has changed. A dependency array can be used in which no re-renders occur without

the value in the array changing. I set the useMemo up in my code in the Flatlist renderComponent that renders the AddPlayer component which contains the RNPickerSelect for a minute. I put the item value – which is the data passed to it from the Flatlist - in the dependency array. This change massively improved performance as the RNPickerSelects now only re-rendered when the data passed into it changed.

```
renderItem={(item)=>useMemo(()=> AddPlayer(item),[item])}
```

Overlap Warning When Creating Subsheets

An issue identified with the design of the PositionSlider whilst getting a friend to try it out was that they didn't notice that they had assigned a player to two positions at once. Resultingly, they started the game with a player playing both CF and LH. Originally, I implemented a modal based solution to this, in which a modal appeared whenever an overlap occurred. This modal approach added length to the subsheet creation process as it required users to close modals and it increases the total amount of drags, they must make. Therefore, I opted out of this approach as it went against feedback from Jakarta that the setup process of a subsheet needed to be shortened.

Instead, I decided to add a visual cue that highlights overlapped players making it obvious to users they need to address the overlap. To add this visual cue, I added a fourth layer to the PositionSlider - the overlap layer. This layer went above the visual layer and below the component layer. To create this layer, I first collected an array of all occurrences of a player playing two positions at a given minute. To get this array I iterated through the positon_timeline for a position and carried out a check to determine if the player assigned at a given minute is also playing at a different position at that minute. The value returned is a 1 if there is an overlap and 0 if there isn't. This list then goes through the blobbing process to combine the adjacent overlaps which is then then rendered through a map method. This is very similar to how the visual layer is handled and a more in-depth description of that is above. The visual cue I added to indicate an overlap was a transparent diagonal cross. This clearly shows to the user that there is something they need to address.

The overlapped area is indicated with a visual cue

Drag Buffer Zones

Behaviourally I noticed that when people tried out the drag functionality of the PositionSlider they placed their finger slightly to the side of the player they wished to drag when starting their drag. Consequently, the drag wouldn't start as the code required the user's finger to be precisely on the player when dragging. This would result in them becoming frustrated as they presume the app hasn't registered their gesture. To address this, I decided to add a drag buffer zone that allowed users to have their finger slightly to the side of the player and still have the drag registered. This buffer zone was the adjacent tile, only if the adjacent tile is unassigned. The index of startMinue must then be adjusted so it is the minute that has the player in it. This made the drag a lot easier to carry out.

Mirrored Subsheet Across Intervals

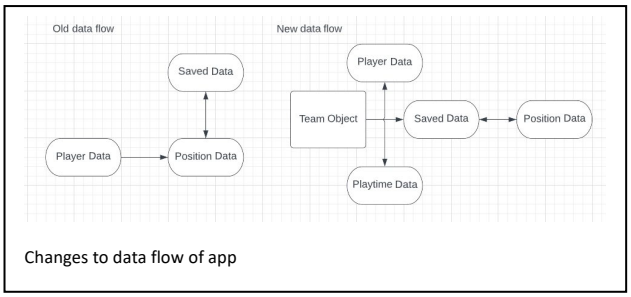
As this was feature that not all users would likely want to use, I began by adding an option for users to select whether they wish to have mirrored intervals on the option screen. Next, I rewrote the reducer for updating position_data to allow all the intervals to be updated the same if mirroring intervals is selected. With this implemented it would quarter the setup time for the app for any coaches using it in a 4-interval game.

```
//0 minute 1 player id 2 position id 3 mirror interval 4 interval length
case UPDATE_POSITION:
    return{...state, position_data: state.position_data.map(
        (content, i) => content.position_id === action.payload[2] ? //Only relavent position is changed
            {...content,
            //Updates position in two cases 1)not mirroing interval but correct minute 2)mirroing intervals and same minute as
changed minute but in diffrent interval ie 2, 17, 32, 47 in 15 min intervals
            position_timeline: state.position_data[i].position_timeline.map((content,i)=>
            ((i===action.payload[0] && action.payload[3]== false) ||
            (i%action.payload[4]===action.payload[0]%action.payload[4] && action.payload[3])) ?
                action.payload[1]  : content)}: content)}
```

Implementation of Teams

Following the meeting with A K my main goal was to implement the gametime statistics overview. A prerequisite of this system was storing data of all minutes played so this could be compiled into a single display. To do this I decided to setup a game_data Redux state variable that stores objects of the minutes each player played in a game. However, after implementing this, I realised a major design issue. That being that this stored game_data would not be specific to a team. For example, two users may be using the same device for different teams, or a keen parent could be coaching both of their kids' teams. Currently I had no way to differentiate between multiple teams on a single device meaning that this seasonal gametime data would be useless as it would contain a mix of players from both teams. To address this design issue, I decided I would rewrite my backend code to implement teams.

The first challenge of doing this was deciding what data each team object would store. Out of the three Redux state variables I had setup so far – player_data, position_data and saved_data – I decided to set up a separate player data and saved data for every team object but keep the position data separate. My rational for this was that saved subsheets and the players in a team are generally unique to every team but the position_data - which is displayed as a subsheet - was only a temporary data store that was previously saved to or loaded from saved_data. Therefore, the position_data would serve as a temporary store, that saved_data from a team object could be loaded into. Likewise, user created position_data could be saved into saved_data. I also planned to add a third field to the team object called game_data which was a list of objects. One object existed for each game a team has played, and the object contains a list of how many minutes each players played in that game.



Changes to data flow of app

Making this change wasn't as much technically difficult as it was a grind. Throughout my code base I had to firstly rewrite reducers so that they accounted for the team-based system. For example, the PLAYER CREATE reducer would before, add a new player to the player_data redux global variable, but now to carry out the same task but the reducer would have to first find the team that is currently in use, then add a player to the player data field of that team object. Alongside making changes to how data was updated I also had to change how data was referenced. For example, instead of referencing the player_data array to get a player's name I now had to first reference the team the player was on before getting the player data array from that teams object to get their name from.

```
//THESE ARE EXAMPLES OF THE MULTITUIDE OF SIMILALR CHANGES I MADE IN MY CODE TO ADAPT TO THIS NEW SYSTEM
//Updated create player reducer to create the player within the currently selected team
case CREATE_PLAYER:
    return {...state,team_data:
        state.team_data.map(
        (content,i) => content.team_id === action.payload[0] ?
            {...content, team_player_data: {team_player_index:
content.team_player_data.team_player_index+1,team_players:[...state.team_data[i].team_player_data.team_players,action.payload[1]]}}
        : content
    )};

//Previously I could just refrence player_data now i had to refrence the player_data of the currently selected team
const player_data = teamState.team_data[current_team_index].team_player_data
```

The final changes made were to the home screen. As opposed to creating or loading a subsheet from the home screen users now created or loaded a team. I then had to add an options screen for when creating a team so users could set the name of their team and what sport it was. After creating a team, it would take users to the old home screen where they could choose between loading a subsheet or creating a new subsheet. When adding the functionality for loading a team I reused most of the code for loading subsheets, reusing the SaveView component but changing the data passed into it. When a user loaded a team, I set the id of that team to a redux global variable called current_team_index. Then when team data is used throughout the app, the index of the team data array referenced was current_team_index.

Implementing Better Data Management Practises

Whilst rewriting all of this backend code I also took this as an opportunity to update and remove some of my more egregious data management practices. The main one was in position_timeline. For every minute on

the position_timeline the value was either null or a player's name. I would then join that name value to the associated player object to get that players colour. This works until there are multiple players with the same name. This is because when the join is called it can only join to one of the many player objects with the same name. This resulted in wrong values being returned. To address this, I moved to a common key system where instead of putting the name of the player I put the id of the player at a minute in position_timeline. This value then acted as a common key that was unique to a player preventing a double up when a join is called. In doing this I had to implement a second step for getting the data of a player. To do this I wrote a function called get player data. When called this function joined the id passed into the function with a player's id then returned all the data of that player.

```
//Assign color gets the relavent information from the player data strucutre and assigns that color to the slider
export default function getPlayerData (player_id,playerData)
{
    if (player_id != null)
    {
        let join = playerData.find(player => player.id == player_id)
        //If join is undefined player no longer exists
        if (join != undefined)
        {
            //0 for name, 1 for color
            return [join.name,join.color]
        }
        else
        {
            return ['Deleted Player', 'red']
        }
    }
}
```

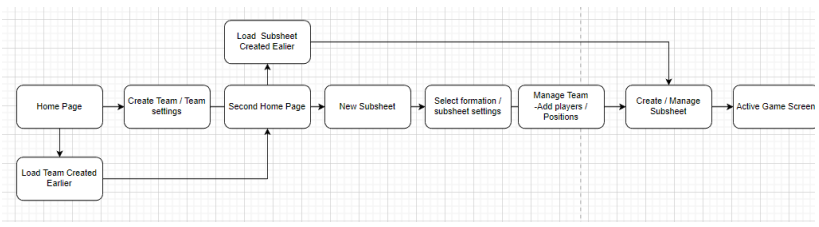
It was also whilst setting this up that I realised that should a player be deleted but still exist on a subsheet the app would crash. This is because the join function would return -1 as opposed to a player object resulting in the code attempting to reference the name and colour of an object that doesn't exist. To counteract this if the join function returned -1, I would make the get player data function return the player name as deleted and the player colour as red. This would make it clear to the user that there is a deleted player in their subsheet that they need to remove it.

How a deleted player appears on a Position Slider

Reconfiguring Navigation System

Problems with Old Navigation System

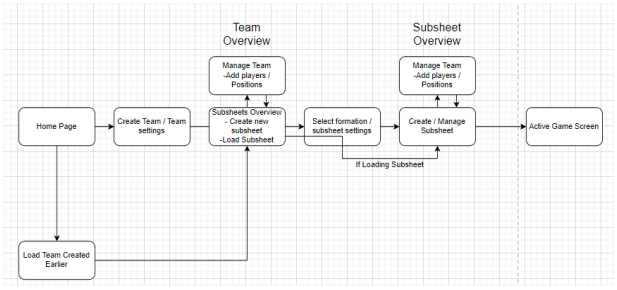
From the start of the development, I never really gave much consideration to how the navigation system worked. My approach was just to stack pages on top of each other. This worked when *SUBlime* was small but now *SUBlime* had a wide range of features and the navigation system was not keeping up. The first problem was that navigation through the app was too linear. This linearity wasn't an issue when users moved forward through the app, but it became an issue when they tried to move backward through it. This was because due to the apps linear nature if the user wanted to go back to an earlier page, they would firstly have to press the back button multiple times and secondly undo all of their progress in the app as they move backwards. This would commonly happen when testers would get to the subsheet creation page, start allocating players time than realise they didn't add all their players. They would then go back to add this player resetting the subsheet. The second problem was that the navigation system struggled with the introduction of the team system; I now had two home pages at different points in the app, which was not ideal.



Overview of current navigation system

Use of tab navigator / stack navigation combo

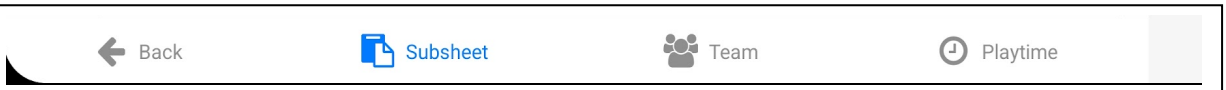
My solution to these issues was to make use of a tab navigator / stack navigation combo. The purpose of combining these navigators was to break up the linearity of the navigation and make it easier for users to move between important pages. A tab navigator is a component that controls navigation through buttons at the bottom of a screen. These buttons when pressed navigated to their respective pages. I setup a tab navigator at two points in the app. The first was after the user either created or loaded a team. This tab navigator was called the Team Overview and had buttons for two pages those being a page to load / create subsheets and the team management page where users could create subsheets. This tab navigator eliminated the two-home screen issue. The second tab navigator was called the Subsheet Overview and contained the subsheet creation page and a second team management page. By including the team management page in both tab navigators, it addressed the issue of linearity in the navigation. Although navigation through the app was still largely linear, by placing the team management page on both tab navigators' users didn't have to travel back pages to reach it. So, although it didn't address the linearity it made it a non-issue. The stack navigation part of the combo came in the form of allowing users to move between these tab navigators through stack navigation.



Refined navigation system

The implementation of these tab navigators was very boiler plate heavy. For all pages on a tab navigator, I had to define a screen for each page. Below is the tab navigator for team overview.

```
const TeamOverview = () => {
  return(
      <Tab.Navigator
        screenOptions={({route})=>({
        headerShown: false})}>
        <Tab.Screen name="Subsheets" component = {SelectSchedule}  />
        <Tab.Screen name="Team" component = {PlayerView}  />
        <Tab.Screen name="Season Playtime" component = {TimeOverview}  />
        <Tab.Screen name="Game History" component = {GameHistory}  />
      </Tab.Navigator>
  )
}
```



Tab navigator for subsheet overview. Note the addition of the gametime option is discussed below

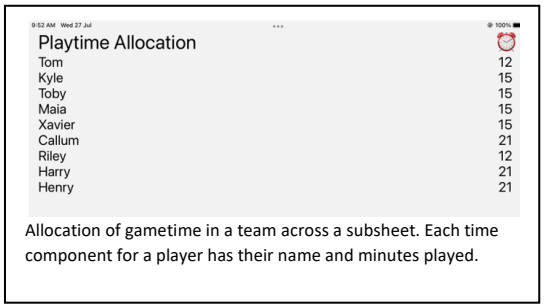
Gametime Overview Page

With the background work of setting up a team-based system and reconfiguring the navigation I was now ready to implement game time overviews. The gametime overview is the defining feature of *SUBlime* as it is used to ensure equal gametime distribution. It is also what A[redacted] K[redacted] identified as the most important feature. I wanted to show users the gametime distribution at two points: firstly, on the Subsheet Overview tab navigator when they are creating a subsheet. This would allow them to make use of the tab navigator to swap between the gametime distribution page and the subsheet they are creating. Based on the distribution of gametime in the subsheet they could then make changes to their subsheet to make the distribution of gametime more equal. They could then check back to the distribution of gametime page to ensure that these changes resulted in a more equal gametime distribution. The second place was when they are on the Team Overview tab navigator to allow them to see the seasonal overview of gametime distribution. On this page it would be clear if any player was consistently getting less gametime, allowing coaches to give more gametime to that player in the future. A final design consideration of the gametime overview page was to ensure that all the gametime distribution information fit within in a single screen allowing it to be screenshotted and sent to parents.

Calculating Gametime of Each Player

I decided to first do the subsheet gametime distribution as it would be the easier of the two. First, I setup a new page to show this time data and added it to the Subsheet Overview tab navigator so users can tap between the subsheet and the time overview, allowing them to adjust the subsheet based on what the gametime distribution is to make a more equal subsheet. To collect the time data over a subsheet I created a list. In the list there was a further list for each player. This list was of length 2 with the first item being the id of the player and the second minutes played of the player– this was initially set to 0. I then iterated through all assigned minute of gametime in the position_data array getting the id of the player assigned at that time. I then found the index of the item in the time list that had the matching player id and added the minutes to the item. I then used a FlatList to render a component for each player that showed their name and minutes played. I deliberately didn't make the time component for each player large. This was done so all the players time components in a team could fit onto the gametime allocation screen without having to scroll to reach any of them. As all players and their gametime was on a single screen, this screen could then be screenshotted by coaches and sent out to parents. Therefore, making it easy to have transparency within gametime.

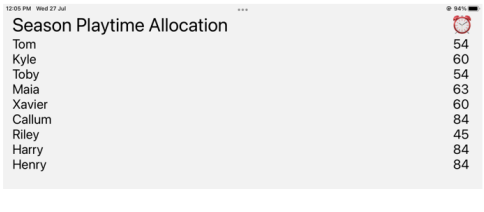
```
//Create list for all players index 0 id index 1 minutes played
let timeData = []
for(let players = 0; players < team_data.length; k++ )
{
    timeData.push([team_data[players].id,0])
}
//Loop through all assigned minutes
for(let position = 0; position < positionsData.length; position ++)
{
    for(let minute = 0; minute < positionsData[position].position_timeline.length;minute++)
    {
        let player = positionsData[position].position_timeline[minute]
        if (player != null)
        {
            //Find the index of timeData where the players have the same id
            let indexToAddTime = timeData.findIndex(player => player[0] == player)
            if (indexToAddTime != -1)
            {
                timeData[player][1] +=1
            }
        }
    }
}
```



Allocation of gametime in a team across a subsheet. Each time component for a player has their name and minutes played.

Displaying Gametime Distribution Across a Season

With gametime distribution being viewable per game, I next added a seasonal breakdown of gametime. To do this I added a new field to the team object that was a list that stored game_data. Then when the player began a game, I would run the function above to collect the time breakdown for each player before saving it to this game_data array. I now had the data and just had to present it visually. I set up a new page in the Team Overview tab navigator for seasonal gametime. In here I wrote a function like the last one to calculate total time for each player. This function iterated through all the saved game_data gametime objects summing the total time each player played across all the games. I also implemented a check to determine if a player still existed. For example, a player may have played the first few games of a season before quitting and hence was deleted from the app. The app would throw an error if I tried to display time for a non-existent player, so I had to account for this. Similarly, to the time overview of a subsheet I then rendered this data using a Flatlist and made it so all players time data could be on a single screen so it could be screenshotted and sent to parents.



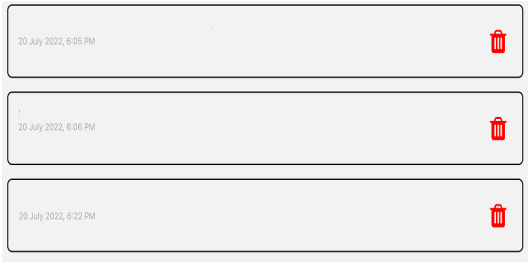
Gametime allocation across a season

Game History

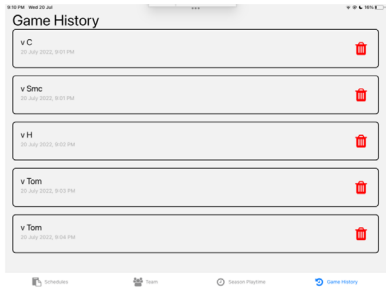
This system would work for an ideal user, but most users are not ideal. For example, users may accidentally start a game early then cancel it because they started it too early then start it again when the game begins properly. This would result in the time data being added twice to the seasonal count, with no way to remove this time data resulting in inaccurate time data across a season. This was reflected by testing carried out in my class were testers claimed there was a bug where the time was added twice to the seasonal total but they had instead started the game twice. My solution for this was to set up a game history page that would be on the Team Overview tab navigator. On this page users would be able to see played games and delete games they accidentally started. To do this I set up a FlatList with a component for each game played and added a

pressable to delete that game from game history. As this game_data was the same data used to calculate total seasonal gametime by deleting a game it removed it from the seasonal total gametime. Once I implemented this, I realised it was impossible to distinguish what game was which. All that was displayed was a formatted time value which wasn't very descriptive. Therefore, I needed to add a way to distinguish which game was which. To address this, I decided I would ask the user who the opponent was in a game when they started it. This opponents name would then be displayed on the game history page, allowing users to distinguish between matches.

```
<Modal
    animationType="slide"
    visible={modalVisible}
    onRequestClose={() => { //Required prop that calls a function when
a hardware back button is pressed such as android back button
        setModalVisible(!modalVisible);}}>
    <View style={styles.centeredView}>
        <View style={styles.modalView}>
            <Text style={styles.textStyleTitle}>Final Steps</Text>
            <TextInput
                style = {styles.textInputStyle}
                placeholder='Enter Opponents Name'
                //Saves the other teams name
                onChangeText={(value)=>{setOtherTeamName(value)}}/>
            <View style = {{flexDirection:'row'}}>
                <Pressable
                    style={[styles.button, styles.buttonClose]}
                    onPress={() => setModalVisible(!modalVisible)}>
                    <Text style={styles.textStyle}>Go Back</Text>
                </Pressable>
                <Pressable
                    style={[styles.button, styles.buttonClose]}
                    onPress={() => {if(otherTeamName!= '')
{setModalVisible(!modalVisible); setupGame()}}}>
                <Text style={styles.textStyle}>Begin Match</Text>
                </Pressable>
            </View>
        </View>
    </View>
</Modal>
```



Original game history display. The only shown field is time of game making it hard to distinguish between matches



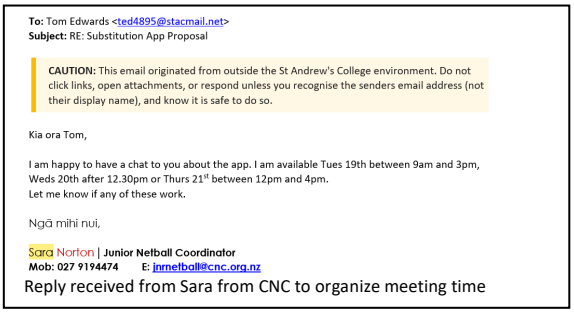
Updated game history page. Shows who the match was against.

To get from the user who the game was against, I made use of the Modal component.[21] A modal is a pop up like the alert component but differs in allowing what is displayed to be fully customizable. On this modal I added a basic TextInput for users to enter the opponent's name and a pressable to submit the modal. Modals have a prop called visible that determines whether it displays or not. I setup a state hook called displayModal and set that equal to this prop. This value was then set to true when the user went to confirm their subsheet and was set to false once users entered a name and pressed the pressable. I then added a new field to the saved game object for the opponent's name to store this variable.

Wide Spreading Testing and Expansion of Stakeholders

Meeting with Sara Norton (Junior Netball Coordinator for Christchurch Netball)

After the first technical test I got in contact with regional sporting bodies to discuss the potential of working alongside them. Out of these groups Netball Christchurch was the only group I heard back from within a short time frame. Their junior netball coordinator S[redacted] N[redacted] got in touch with me midway through Term 2 holidays and offered to discuss the app and potentially test it with their Future Fern teams. This was great news as the Future Ferns (year 3-6) grade was the target demographic for this app as equal gametime is imperative at this age. After initial discussion over email with S[redacted] we made a time to meet at the start of term 3.



To: Tom Edwards <ted4895@stacmail.net>
Subject: RE: Substitution App Proposal

CAUTION: This email originated from outside the St Andrew's College environment. Do not click links, open attachments, or respond unless you recognise the senders email address (not their display name), and know it is safe to do so.

Kia ora Tom,

I am happy to have a chat to you about the app. I am available Tues 19th between 9am and 3pm, Weds 20th after 12.30pm or Thurs 21st between 12pm and 4pm.
Let me know if any of these work.

Ngā mihi nui,

Sara Norton | **Junior Netball Coordinator**
Mob: 027 9194474 **E:** jnrnetball@cnc.org.nz

Reply received from Sara from CNC to organize meeting time

Prior to my meeting with S[redacted], I went through the process of setting up my app on Expo Go on the testing channel as a published build. This was simple to setup and just needed to run the command expo publish:testing. It was necessary to do this as when I met with S[redacted] at the [redacted] Netball Centre, I would

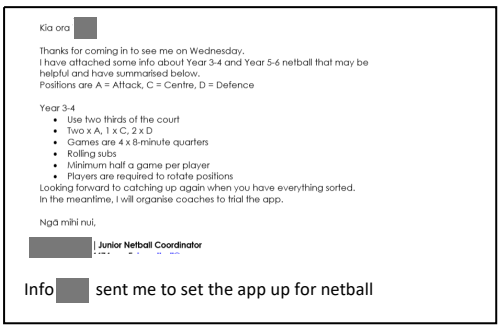
[21] https://reactnative.dev/docs/modal

not have access to Internet and therefore couldn't use the expo testing network. By publishing I could access the build offline allowing me to show her the app.

I had carried out this process before when showing the app to A K last term and when doing the technical test, but it only became a relevant topic to discuss now. This was because when attempting to set this up I kept on getting a bug where the app would white-screen when adding a new player. I was left perplexed by the fact the bug only occurred on the published build. I frantically tried to fix this, but it was 10pm and to add to the stress alongside it being my meeting with S tomorrow it was also the first day back at school for term 3. Debugging this white screen was near impossible for two reasons. Firstly, I couldn't replicate the bug in my test build, and it would only appear on the published build. When the bug occurred on the published build there was no error message making it extremely hard to isolate the issue. Additionally, as it was a published build, I couldn't make use of any debugging tools to gain error logs. Secondly, it would take two minutes to publish the new build, followed by a two-minute download onto my phone. This meant the process of debugging was extremely slow. I attempted to fix this error for about four hours before eventually giving up for the night. Luckily, I had a published build from a month ago when I last met A K that I could show S. This would do but was not ideal as the app had progressed a lot since then.

Despite the mishaps the night prior trying to get Expo distribution working the meeting was surprisingly successful. S was very supportive of the idea of the app as she believed there was a clear need for it. She explained to me about how they had a substitution management app for non-junior netball and said prior to my email that she was in search of a substitution management app to use in junior netball. She was very happy with all the features and believed there was a real demand for its use. My discussions with her also revealed a helpful application the app had. Sara talked a lot about how she often has to deal with parents complaining about their children not getting enough gametime resulting in heightened parent-coach tension. She believed that the gametime statistics was a clear remedy to this as it could be used to easily and transparently communicate to parents the distribution of gametime. Because of being able to send this data to parents, she believed it would remove this tension allowing a greater team spirit and coaches being able to coach without having a target on their back from the parents.

S was also extremely helpful in explaining how I should adapt the app to netball, explaining how positions and subs work in netball. The greatest success of the meeting was S's willingness to carry out testing. She said that she overlooked 160 Junior Netball teams and was willing to distribute the app to all of the coaches alongside actively encouraging its use at manager meetings. This was a huge success as it opened the door to massive amounts of testing and feedback. The final topic of discussion was the time frame to carry out the testing. She explained there was four weeks left to the netball season and was willing to carry out testing in the coming weeks games. Although tempting I opted to organize the testing for the last two weeks of the season, giving me a two-week grace period to get the app sorted for mass testing.



Kia ora

Thanks for coming in to see me on Wednesday.
I have attached some info about Year 3-4 and Year 5-6 netball that may be helpful and have summarised below.
Positions are A = Attack, C = Centre, D = Defence

Year 3-4

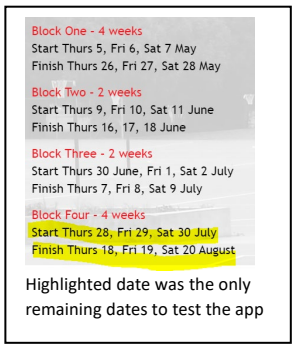
- Use two thirds of the court
- Two x A, 1 x C, 2 x D
- Games are 4 x 8-minute quarters
- Rolling subs
- Minimum half a game per player
- Players are required to rotate positions

Looking forward to catching up again when you have everything sorted.
In the meantime, I will organise coaches to trial the app.

Ngā mihi nui,

Junior Netball Coordinator

Info sent me to set the app up for netball



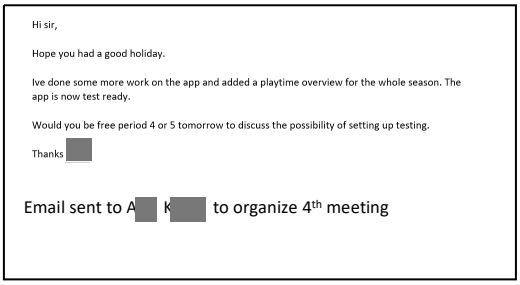
Block One - 4 weeks
Start Thurs 5, Fri 6, Sat 7 May
Finish Thurs 26, Fri 27, Sat 28 May

Block Two - 2 weeks
Start Thurs 9, Fri 10, Sat 11 June
Finish Thurs 16, 17, 18 June

Block Three - 2 weeks
Start Thurs 30 June, Fri 1, Sat 2 July
Finish Thurs 7, Fri 8, Sat 9 July

Block Four - 4 weeks
Start Thurs 28, Fri 29, Sat 30 July
Finish Thurs 18, Fri 19, Sat 20 August

Highlighted date was the only remaining dates to test the app



Hi sir,

Hope you had a good holiday.

Ive done some more work on the app and added a playtime overview for the whole season. The app is now test ready.

Would you be free period 4 or 5 tomorrow to discuss the possibility of setting up testing.

Thanks

Email sent to A K to organize 4th meeting

Fourth meeting with A K

At the start of term 3 the winter sport season was in its twilight, so it was necessary to get testing with StAC teams under way. I organized a meeting with A K for the second day back of the term. As it was the day following the meeting with S, the white screen error still persisted, so I was unable to show A K the work I had done over the holidays adding the team system he had requested. Luckily, I had a few screenshots saved that I could show him. Going into the meeting the agenda was largely the same as the netball meeting.

A K was impressed with the progress of the app and believed the app was test ready. However, we both agreed on two further steps before the app could be sent out to StAC teams for testing. First, I would have to adapt the app to different sports environments, because at this point I only had it set up for hockey and the relevant information to setup it up for netball. I had also lost hope in Mainland football and Canterbury Basketball replying to my email – within a reasonable time frame - so I had to get the information to adapt the app to other sports in another way. A K gave me a list of contacts of Heads of Sports at College. The second request he had before testing was for me to compile a document that explained the setting up of the app for testing on Expo, so coaches could easily install and use the app.

Beyond this A K also suggested the implementation of a feature that allowed changes to be made to a subsheet midgame. He believed this would be worth implementing to account for the possibilities of injuries. Currently if a player is injured the subsheet will become desynced from the game situation as the app will try and substitute on this injured player. By allowing changes to be made to the subsheet mid game this injured player can be removed from the subsheet, addressing this issue.

Meeting with College Heads of Sport

After the meeting with A K, I made contact with the relevant heads of sport at over email. Out of the list provided I decided to contact the Heads of Sports for winter sports – as they were mid-way through a sports season and could therefore provide testing.

The people I contacted were as follows

- J C – Head of Football at
- B E – Head of Basketball at
- M J – Head of Rugby at

All of them replied stating their excitement and interest in the project and organized a time to meet up to discuss.

Hi Sir,

As apart of my digital technology project I am working on an app that manages substitutions in sport. I have completed successful trials of the app in a hockey and netball environment but I am looking to broaden the scope of the app to include rugby as one of the options. In order to do this I need information on stuff such as formations, positions and rules of substitution in rugby.

As you are in charge of rugby at St Andrews I was hoping you would be willing to give me like 5-10 minutes of your time at some point so we could discuss the implementation of rugby into the app.

If this is possible what times work for you to discuss this?

Thanks from

Email sent to to organize testing for rugby.
Similar emails were sent to

Meeting with J C (Head of Football at)

The first of the three heads of sports I met with was J C. J was the head of football at alongside working as club director for the local sports club also has a wealth of experience as a former football player, playing for pro teams in America, Guatemala, Australia, and New Zealand. In addition to this J has done work in the past as a sports analyst for a football team. J therefore has immense insight as both a football player and a coach.

The meeting started with me showing J the app. He was impressed with the app and explained that he already makes use of a substitution management app but pointed out that my app uniquely had two features beyond his currently used app. Firstly, a system that counts total minutes played across the each game and season and second, a game overview screen that showed the current field of play and counted down to upcoming subs. J recognized these two features as immensely useful and believed they were lacking from his current app.

Next J went through a logical thought process of a coach using this app and what concerns / issues they may have with the app. The first thing he questioned was, what would happen if a player got injured during the game. This further reaffirmed A K's feedback about adding a feature to allow changes to a subsheet midgame. The second thing he pointed out was to do with seasonal gametime. J emphasized a lot how much he liked this feature, but he noted that inequality in gametime across a season could be simply due to a player missing a game because they're on holiday / injured resulting in them not getting any minutes for a game. He said it would be crucial to add some sort of system to account for the missing of games in gametime distrubtion.

The next part of our discussion was about setting the app up for a football environment. I had the pleasure of a passionate lecture from J C about the pros and cons of a range of different formations in football. The conclusion from this lecture was that there was a lot of variation of formations in football, more so than hockey. This meant that it would be hard to account for all formations like I had done in hockey and netball. He gave me a few standardized structures such as 4-3-3, 4-4-2, 4-4-2 diamond and 3-4-3 which I added. In

order to account for this large variation in structures he suggested I add the ability for users to create their own formations. J also talked about how in junior football there is 7-aside and 9-aside so there would be need to also add these options. Beyond that, substitution rules in hockey and football – at a junior level – were largely structured the same, so not much change was needed.

The final point of discussion was about testing. J offered three forms of testing. Firstly, he was keen to test it with the College First XI girls' team he coached. This was unique testing environment as all other organized tests thus far were with junior teams in a low stake environment; this was a competitive high-stake environment. Secondly, he said he was happy to send out the app to football teams he overlooked to test. Finally, he suggested that we meet again another time over summer to set the app up for, and carry out testing with Fustal teams.

Meeting with B E s (Head of Basketball)

Following the meeting with J C I met up with B E later that day. B is the coordinator of basketball at College. In addition to this he is an accomplished basketball coach and player himself. The meeting began with me showing B the app. He similarly expressed that he was impressed with the app and believed it had immense promise. B then showed me the current programme the school used for basketball. The programme was called Hudl[22] and it managed statistics / total minutes played for teams. Although Hudl was a professional and complete product he pointed out a couple advantages of my app. The first was the ability to view game time across a season; Hudl only allows you to view minutes played for each game. The second was that my app had a visual display during the game, that tells people when to sub; Hudl only showed statistics after the game and could not be applied in this way. Finally, my app could be used to create subsheets a feature Hudl lacked.

In terms of adapting the app to Basketball we broke the discussion into two parts. First to do with competitive Basketball. B believed that at a competitive level the app didn't provide enough flexibility to make on the fly changes. He explained how basketball coaches make subs based on how many fouls a player has made, how players are playing and their fitness level. These changing variables mean that basketball coaches at high levels often go off their prescribed subbing plan making the app not very useful. However he noted if I implemented the ability to make changes to the subsheet midgame – like A K and J suggested – it would remedy this issue as coaches would be able to adapt their subsheet to the game situation midgame. This further highlighted the need to add this feature as it would expand the user base of the app to competitive play alongside social sports.

Team A Player Stats

#	Athlete	Pts	FG	3FG	FT	+/-	MINS	OREB	DREB	AST	DEFL	STL	BLK
13	Brad Hilligoss	0	0/0	0/0	0/0	-	-	0	0	1	0	0	0
21	Eric Brouillette	3	1/3	1/3	0/0	- 8	21	0	0	1	1	0	0
2	Brandon Arnold	0	0/0	0/0	0/0	+ 3	1	0	0	1	0	0	0
23	Alex Ward	7	3/6	1/1	0/0	- 16	32	1	6	3	0	1	1
17	Mark Kolarik	6	2/5	2/3	0/0	+ 1	10	1	0	2	0	0	0
20	Andrew Abraham	16	4/12	2/8	6/6	- 15	26	0	1	0	1	0	0
6	Doug McClure	5	2/4	0/1	1/4	- 10	27	0	2	4	1	0	1

Hudl stat page

The conversation then shifted to social / junior basketball. He believed that the best use of the app would be in social / junior basketball where enjoyment in these grades was determined by getting time on court as opposed to winning. He noted that substitution rules aren't as strict in social / junior basketball and therefore would not have to make the considerations made in competitive.

The final thing discussed was to do with testing. Social Basketball provided a unique testing opportunity. This is because there were six social basketball games at every Friday night and all of the games were overseen by a coordinator M W. B E suggested that I organised a time to meet with M W and teach him how to use the app. He then said M could pass it on to basketball coaches on the Friday evening and teach them how to use the app. This was advantageous as it would eliminate the need for basketball coaches to install the app on their own devices and instead could make use of a pre-setup device. This would make them more likely to try out the app and provide feedback.

[22] https://www.hudl.com/

Meeting with M J (Head of Rugby)

The last meeting, I had was with Mr J. This conversation was by far the briefest as the app has a limited application to Rugby. This was because Rugby makes use of archaic substitution rules that only allows substitutions to occur at fixed points in the game. Because of that the app had very little application for managing subs during the game as it was easier for coaches to announce their subs at these fixed time as opposed to setting up the app beforehand. However, Mr J did express his liking for the seasonal gametime overview as he viewed equal gametime as a key pillar of junior sport.

In the end we decided to not carry out any testing with rugby as my effort setting up testing would be better suited for other sports. Instead, we added a formation for rugby into the app so that the option could be available for rugby coaches to use the app further down the app. Mr J provided me a rugby formation; luckily rugby has fixed positions with little variation so only adding one formation was nesscary.

Preparations for widespread testing

I now had a two-week window before I planned to roll out the app. I broke these tasks down that I wanted to complete in these two weeks.

1. Fix critical bugs that prevent the app from working
2. Test out on different devices and make bug fixes were appropriate to allow users on all devices to use the app
3. Noob-proof the app; I.e. making the app easier to use and prevent the user harming themselves
4. Implement the new sports to the app
5. Add requested features from meetings with stakeholders
6. Set up a distribution method for testing.

Bug Fixing

The first task I carried out was fixing bugs. It was crucial that I had a bug free app when the app was sent out for testing. This was because crashes and major bugs would prevent users from using the app. Also, if the app was buggy all the feedback from testers would focus on bugs as opposed to features.

The first critical bug I fixed was the white screen bug. As explained above this was extremely hard to debug due to it only appearing in published builds. To identify what the issue could be I removed code from the problematic PlayerTab component (A PlayerTab component was created for every new player, as the app crashed when a player was created, we know the PlayerTab is at fault) systematically and re-published the project until it ran. This would hopefully isolate the problematic code. This made no change to the white screen appearing. Eventually I had removed all the code and instead put in a normal text component. The white screen persisted meaning that the code inside the component was not at fault.

I then realised that I was still importing data and functions into the component, meaning that these imports could potentially be the issue. This was confirmed when I imported nothing, and the white screen didn't appear. Now I had to identify which of the imports was the issue. Like the process of slowly removing code part by part I slowly removed the imports from the component. As I removed the imports, I noticed a trend; there was no white screen when only importing data but when a function was imported the app white screened. My solution to this was moving the PlayerTab component inside the scope of the TeamManagment component. As the functions being imported were defined in the TeamManagment scope, the PlayerTab component no longer had to import these functions as they now inherited them from the TeamManagment scope.

Code ran	Results

`const PlayerTab = ({item},playerData,currentTeamIndex,updatePlayerName,addPositionToPlayer,removePositionFromPlayer,deletePlayer) => {`
`  return(`
`  <Text>Test</Text>`
`  )}`
	Importing: Functions and Data
	Outcome: White Screen
	Show: White screen not caused by code as only plain text component returned.

`const PlayerTab = () => {`
`  return(`
`  <Text>Test</Text>`
`  )}`
	Importing: nothing
	Outcome: No white screen. Text appears.
	Shows: Imports are at fault for white screen

`const PlayerTab = ({item},playerData,currentTeamIndex/*,updatePlayerName,addPositionToPlayer,removePositionFromPlayer,deletePlayer*/) => {`
`  return(`
`  <Text>Test</Text>`
`  )}`
	Importing: Only data.
	Outcome: No white screen. Text appears.
	Shows: Data imports not at fault for white screen

`const PlayerTab = ({item},/*playerData,currentTeamIndex,*/updatePlayerName,addPositionToPlayer,removePositionFromPlayer,deletePlayer) => {`
`  return(`
`  <Text>Test</Text>`
`  )}`
	Importing: Only functions.
	Outcome: White Screen
	Shows: Function imports are at fault for white screen

The second critical bug I addressed was the multitude of indexing errors being caused by a team being deleted. As explained above in the setting up of the team reducer, I store a value of the currently loaded team's id and that value is used as the index when retrieving data for the selected team from team_data. This worked initially but upon adding the ability to delete teams the variables for a team's id and their index became desynced. For example, a list of five teams were deleted apart from the last one. The last team has an id of 4 but its index in the list is 0. This means when the current selected teams id was used to access the data it would check at index 4 but no value existed there causing an index error. I fixed this by adding a variable for adjusted team index which is retrieved by getting the index of the team that has an id matching up with the current_team_id.

```
//Gets the id of the currently selected team. Then finds the index that matches up with that id in team data.
const team_id = generalData.current_team_id
const adjusted_team_index = teamData.team_data.findIndex(item => item.team_id == team_id)
```

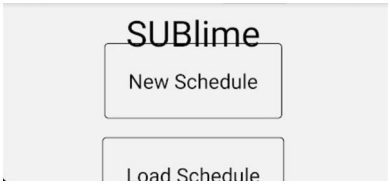
Other minor bugs fixed was the extending of the implementation of the overlap warning system to do a check for a player overlap before starting a match to prevent users starting a game with a player in multiple positions in a given minute. Additionally, more minor bugs were fixed such as applying a filter to the seasonal gametime overview to prevent deleted players from causing indexing errors.

Device Specific Glitches

Most of the development of my app up to this point had been done on a single iPad. Consequently, I had at times, styled parts of the app to fit an iPad screen size. This resulted in devices without large screens such as smaller mobile devices having visual glitches, where content wouldn't display. To find these glitches I used my Mums old phone which was an iPhone SE. The iPhone SE has one of the smallest screen sizes on the market so I assumed if content would display correctly on that device, then it should on all.

Home screen without responsive design. Appears fine on large device

Upon trying the app out on the iPhone SE I found the visual glitches to be quite common due to my design being solely around an iPad prior to this and most of my styling having fixed dimensions irrespective of devices. The fixes to these glitches were the same in all cases and required me to implement responsive syling. I addressed this by using responsive design and wrapping components in views with flex styling, so they resize dependent on device screen size. I also made use of max-height and max-width styling for larger devices to prevent components being too large. Below is the responsive styling of the home screen.



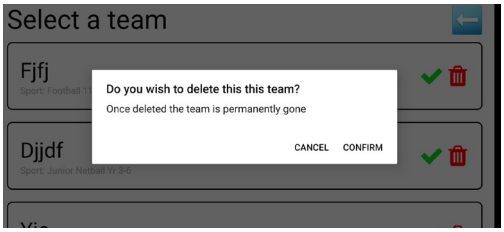
Home screen without responsive design on small devices. Content overflows and overlaps.

```
//Inline styling is deliberate to demonstrate responsive design
<View style = {{flex:7, alignItems:'center',justifyContent:'center'}}>
    <Pressable onPress ={()=> setDisplaySetup(true)} style
=   {{borderWidth:2,borderRadius:9,width:240,alignItems:'center',marginBottom:10,flex:3,justifyContent:'center',maxHeight:120}
}>
        <Text style = {{fontSize:30}}>New Team</Text>
    </Pressable>
    <Pressable onPress ={()=> navigation.navigate('LoadSave')} style =
{{borderWidth:2,borderRadius:9,width:240,alignItems:'center',flex:3,justifyContent:'center',maxHeight:120}}>
        <Text style = {{fontSize:30}}>Load Team</Text>
    </Pressable>
    <View style= {{flex:1}}/>
</View>
```

Noob Proofing the App

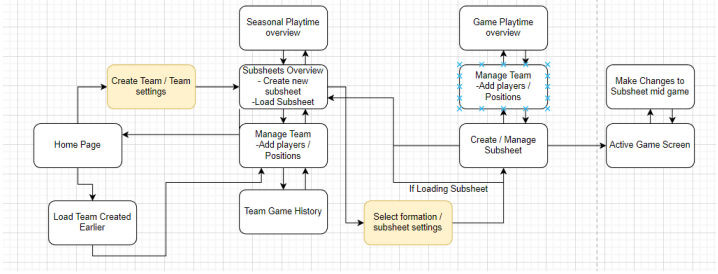
With a stable product achieved the next step was noob proofing. Making the app easy to use was crucial for widespread testing, as I couldn't physically be there to guide users through the app. It was crucial I made the app easy to use so users didn't become frustrated and give up on the app during testing. Additionally, I also aimed to make changes here to protect users from themselves. I.e. making it harder to accidentally delete teams. First I improved the navigation system by adding a back button to both tab navigators. This was done by adding a new component to the tab navigator that when pressed navigated to the prior page. This provided a easy way for users to navigate backwards through the app, as previously users had to do a back swipe motion which may not be clear to less tech-savvy users.

The next change was adding an alert notification that confirmed whether the users wished to delete a subsheet / team. This would prevent accidental deletions. I used the alert component for this. The next change of safeguarding users against themselves was adding a pop up that reminds users to save their subsheets. There is not an auto save system for subsheets, therefore it is nesscary to remind users to save before exiting. I addressed this by making an alert appear the first time the user reached the subsheet page, reminding them to save their subsheets.



Pop up alerts that make users confirm whether they wish to delete a team. This prevents them accidentally doing it.

Finally, I added safeguards to prevent the user from accidentally overriding their team and subsheets. Users could create a team, add players to it and then assign positions to all their players. They could then accidentally press the back button taking them to the team creation settings page as it was the screen prior in the navigation stack. They would then press the button to create a new team assuming it would take them back to their team. Instead of doing this the app would override the old team deleting their progress. To fix this I changed the settings menu from a page into a modal. Whilst this may seem unintuitive, it was done to remove the setting pages from the navigation stack as they are now modals. Now if a user swiped backwards, it would take them back to the home page as opposed to the team creation settings page – as it's removed from the navigation stack as a modal - meaning they couldn't accidentally override their team. I also turned the subsheet settings page into a modal as users could also override created subsheets in a similar way.



Modal options pages are shown in yellow. As shown back arrows skip these pages preventing overrides

Implementation of Additional Sports

Throughout the meetings with the various sports coordinators, I had collected information related to their sports. This info contained basic formations for the sport, what the game pitch looked like and specific subs rules. Luckily none of the sports had subbing rules that would require a redesign of the app.

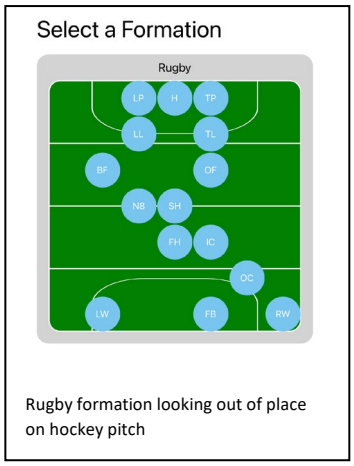
There were two places in the app were content would have to be displayed differently dependent on sport. The first was in what formations were to be displayed. When working alongside stakeholders for each sport I got a basic list of formations to add. The implementation of these formations was a twostep process. The first step was hardcoding these new formations into the formation data array. I explain above how this data is turned into formations. I added a new value to each formation object in the form of formation_sport which contained the sport that formations are used for. The second step was applying a filter to this array, so it only displays formation data, for formations that have the same sport as the sport selected. I applied a similar process to selectable positions that meant that only positions in the selected sport were able to be added to a player.

```
//Example of one of many newly added formations for 9 aside football
{
    formationId:19,
    formationName: '3-2-3',
    formationSport: '9F',
    formationData: [
        [0,0,0,['Centre Foward','CF'],0,0,0],
        [0,['Left Wing','LW'],0,0,0,['Right Wing','RW'],0],
        [0,0,0,0,0,0,0],
        [0,0,['Left Midfield','LM'],0,['Right Midfield','RM'],0,0],
        [0,['Left Back','LB'],0,0,0,['Right Back','RB'],0],
        [0,0,0,['Centre Back','CB'],0,0,0],
        [0,0,0,['Goal Keeper','GK'],0,0,0]]
},
//Filter applied that removes all formations that aren't for that sport
const formationData = formationDataAll.filter(item => item.formationSport == teamData.team_data[current_team_index].team_sport)
```

With this implemented functionally the code worked. A user could select their sport then chose a formation specific to that sport. However, it didn't look quite right visually because all formations were displayed in front of a hockey pitch. To counter this I made new pitch components for all newly added sports using the same view/flex method I used to make the original hockey pitch. With these new pitch designs added I

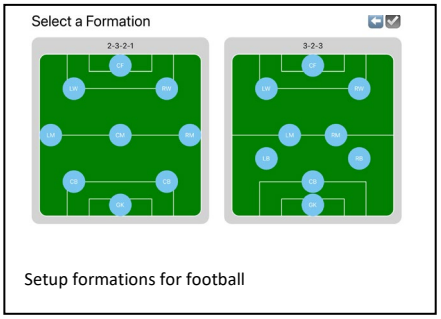
implemented a switch statement that returns a different pitch design to render based on what the teams sport is.

A final minor change I made was specific to junior netball. In junior netball there are only three positions and Sara explained to me how a big part of junior netball is everyone rotating through positions. As this was the case, I auto added all three of the netball positions to a netball player on their creation, thus saving setup time.

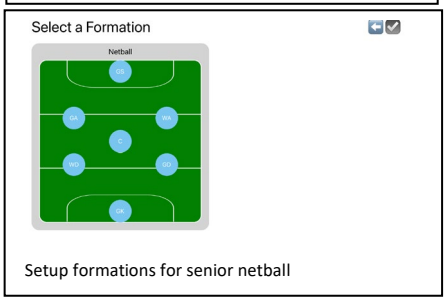


Rugby formation looking out of place on hockey pitch

```
const RenderedPitch = () =>
{
    switch(props.sport)
    {
        case '7H':
            return <HockeyPitch/>
        case '11H':
            return <HockeyPitch/>
        case 'N':
            return <NetballPitch/>
        case 'NS':
            return <NetballPitch/>
        case 'B':
            return <BasketballPitch/>
        case '11F':
            return <FootballPitch/>
        case '7F':
            return <FootballPitch/>
        case '9F':
            return <FootballPitch/>
        case 'R':
            return <RugbyPitch/>
        default:
            return <HockeyPitch/>
    }
}
```



Setup formations for football



Setup formations for senior netball

Addition of Stakeholder Requested Features

Mid Game Subsheet Changes

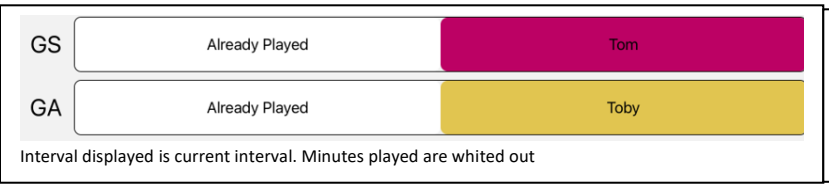
Making changes to the subsheet during the game is a task I had viewed as quite difficult throughout the projects development and therefore had put it off. However, due to many stakeholders stressing the need of the feature due to the fact a subsheet could easily be ruined if a player becomes injured or fouled out mid game, I decided to implement it. When implementing it I planned to reuse the subsheet creation page but make the change that users couldn't edit the subsheet in already played game time. Additionally, any changes made would be reflected instantly on the game overview screen.

Before implementing this feature two steps of setup were required. First, I had to add the option to navigate to the subsheet change page from the game overview page. I did this by creating a Match Overview tab navigator and adding the game overview and the new subsheet change page to it. Second, I had to move time related variables out of the game overview screen to the Match Overview tab navigator. This was done to allow time variables to be inherited by both the subsheet change page and the game overview page, as opposed to previously only existing in the game overview page. I also moved the code that handled the changing of the clocks time to the Match Overview tab navigator.

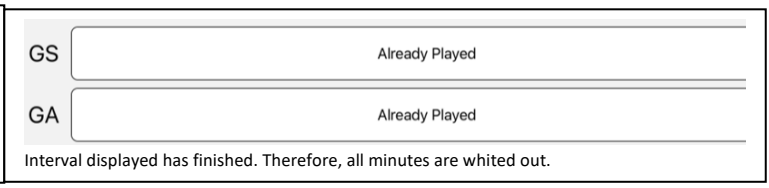
Now with the time data being passed into the subsheet change page I had to transform it visually to display what time has been played. To do this I planned to overlay a white area on already played time. To set this up I made changes to the visual layer of the PositionSlider. I needed to consider two cases when whiting out played time. The first was when the interval being displayed had already been played. In that case I overrode the running of code that created the visual layer for that interval and instead added a single blob that spanned the whole interval with name 'Already Played' and colour white. The second was when the interval was being currently played. In this case I applied a modulus operation to determine how many minutes had been played that interval then set the length of the already played blob to that. I then ran the code to create the visual layer for the un-played minutes of that interval.

```
//If interval already played - no other visual layer creation code runs after this
blobbed_data.push({name:'Already Played',length:intervalLengthProper,color:'white',overlap:[]})

//If current interval is being displayed. Visual layer creation code runs for the unplayed minutes after this
blobbed_data.push({name:'Already Played',length:minutesPlayed%intervalLengthProper,color:'white',overlap:[]})
```



Interval displayed is current interval. Minutes played are whited out



Interval displayed has finished. Therefore, all minutes are whited out.

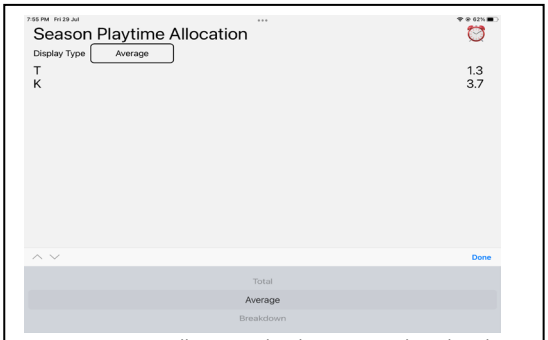
With the frontend code working I next had to do make the backend of the code consistent with these changes. The first was to prevent users allocating already played time to players. To address this, I added checks to the drag end function that prevents already played minutes from being reallocated. The second and far more complex change was handling the saving and updating of the subsheet data during the game. To implement this, I first had to add a new prop to the PositionSlider component called gameActive. This prop was used to distinguish whether the user was saving the subsheet midgame or not. It was necessary to add this prop as I would have to handle the saving differently between when the user creates the subsheet before the game and when they update it midgame. I used this prop to remove certain code from running when the game was active and the subsheet was saved. The code I removed was navigation code that was specific to the user moving between creating the subsheet and loading the game overview screen. I also removed the calling of the save game data reducer as I didn't want duplicate games to appear in game history when users save the subsheet mid game. I also removed the updating of the current interval to 1 that was specific to starting the game as it would reset the user's game by going back to the first interval. With this setup users could now make changes to their subsheet midgame, and those changes would be reflected on the game overview page.

Analytical Gametime Break Down

In my discussion with J[redacted], he believed a greater degree of information was needed to be provided in gametime distribution stats. He said inequality in gametime would arise from people missing games due to them being sick, being away or being injured. Therefore, they would have a lower seasonal total gametime which was not reflective of how much gametime they got in the games they attended. Based on this I decided to add two new view types for gametime data alongside the already present total gametime.

The first was average gametime. The average gametime accounted for games that players missed, and removed any games were a player played 0 minutes. This allowed for an understanding of gametime distribution irrespective of games missed. To implement this, I created a list for each player that had their time data for each game. I then filtered this list removing any games where 0 minutes were played before finding the average of the list. These values are then rendered for each player using a Flatlist.

```
//Loop through every players time data and average it out
for(let player = 0; player < timeData.length; player++)
{
    //Apply a filter to remove any games were the player didnt get any time
    timeData[player][1] = timeData[player][1].filter(time => time != 0)
    //Check if player has played no games and if so set time to 0. This is
to prevent division by 0 error
    if(timeData[player][1].length == 0)
    {
        timeData[player][1] = 0
    }
    else
    {
        //Average the players time of the games they have played
        timeData[player][1] = (timeData[player][1].reduce((a, b) => a + b,
0) / timeData[player][1].length).toFixed(1)
    }
}
```



Average gametime allocation. Also shown is RNPickerSelect that allows users to change view.

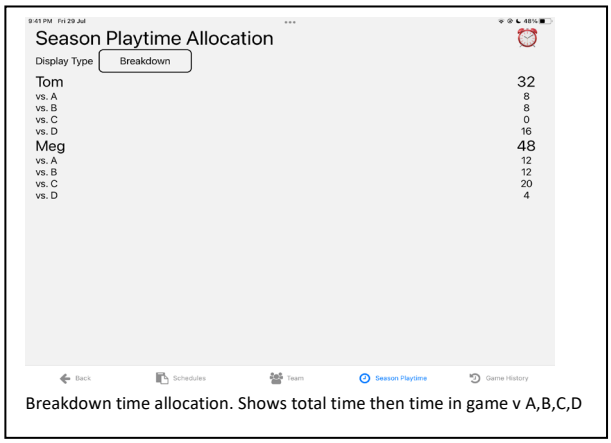
The second was breakdown gametime view. The idea with this gametime view was to display the total time played for each player, then show how many minutes they played in each game below that. To display this, I made use of a new component I hadn't used before called SectionList[23]. Functionally it was very similar to

[23] https://reactnative.dev/docs/sectionlist

FlatList which I had used throughout my project, but the difference SectionList provided was the ability to display headers and subsections of data below these headers. This was exactly what I needed as I could set the header to the player's name and total time played, then the subsection data to their individual games played. To set this up I had to make a couple of changes. The first was that when creating the data for the SectionList I formatted it as an object for each player with three fields, namely id of the player, total minutes played across the season and an array that had the minutes a player played for each game. With this data I was able to render a header that had the players name and total minutes that player played across all games. I set the section data equal to the array that stored the minutes played in each game for a player. This then rendered below each player header all the games the player had played and how many minutes they got in each game. Also note the use of the gameFromIndex function which retrieved the name of the team that the game was against allowing it to be displayed. Finally, I implemented a new prop to TimeTab – the custom component I made to display gametime - called isHeader that allowed conditional styling to be applied to make the header large than the section data.

```
<SectionList
    sections={timeData}
    keyExtractor = {(item,index) => item+index}
    renderSectionHeader={({ section: { id,total } }) => (
        <TimeTab
        name = {nameFromIndex(id)}
        total = {total}
        isHeader = {true}/>)}
    renderItem = {({item})=>
        <TimeTab
            name = {gameFromIndex(item[0])}
            total = {item[1]}
            isHeader = {false}/>}
/>

//Function used to retrieve who the match was against to display
function gameFromIndex(i)
{
    return 'vs. ' + gameData[gameData.findIndex(game =>
game.game_id == i)].game_opponent
}
```



Breakdown time allocation. Shows total time then time in game v A,B,C,D

The final thing I did was implement conditional rendering of these different gametime views. I put a RNPickerSelect at the top of the page that allows users to move between these views.

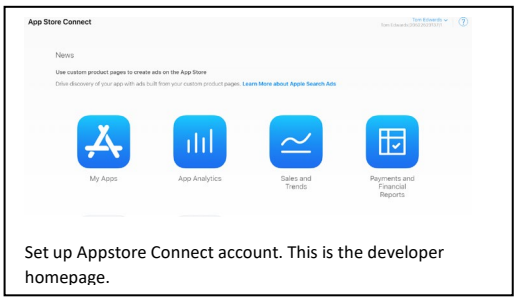
Setting up distribution of app for testing

Failed Expo Distribution method

Initially I planned to distribute my app through Expo Go using the publish method I made use of to show the app to stakeholders offline. This publish method generates a QR code that could then be scanned for the app to be emulated on the Expo Go app on a user's device. However, this failed as I operated under the false assumption that people would only need to install the Expo Go app then scan a QR code to access my app. What I didn't realise is that people would need to create their own Expo account and then I would have to invite their account to my project in order for them to be able successfully scan the QR code. While I could distribute the app this way I decided not to as I believe it would add unnecessary red tape and confusion to the installation process that may push people away from using the app. Following this disappointment, I carried out some research into other distribution services of beta apps. In the end I decided to use Testflight for iOS as it was *literally* the only way to do widespread testing for iOS. For Android I decided to put my app on the Android Play Console Beta Programme and distribute it that way.

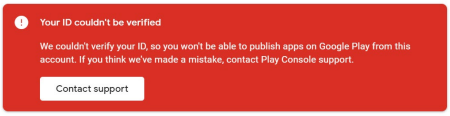
Apple Account Setup / Testflight

Testflight is Apple's official beta testing platform, and it serves as a precursor to the App Store. As a developer I can upload my builds to Testflight. The builds then go through a verification process to make sure the app isn't malware or shovelware before its accepted onto Testflight. This verification process usually takes a day. From there users on Apple devices can download the Testflight app then download my app through it. The first step to setting this up was creating an Apple developer account. This process was relatively straightforward apart from a hefty developer fee I had to cop in order to set up my account.



Set up Appstore Connect account. This is the developer homepage.

Google Play Console Account and Change to .apk Distribution Model

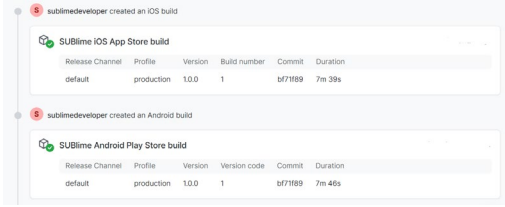


Google denying my id verification

Google Play Console Beta Programme is essentially the Android version of Testflight. The process to set it up is largely the same apart from the fact that an identity verification is required. Google continually rejected my identity verification even when I tried a range of documents such as passports, driver's licences and school ids. In the end I gave up on using this distribution model and searched for a new method to distribute the app on Android. What I decided on was instead building an apk version of my app then getting testers to sideload it onto their device. To do this I would have to reconfigure the Android build option to change the build type from .aab to .apk. A final disadvantage of side loading apk's is that it sends virus warnings to users, on installation as the app is not from the Play Store. This was not ideal as I feared it may scare away some users who mistakingly view my app as malware. Although these disadvantages exist, I would have to put up with them as I couldn't distribute the app any other way on android.

Setting up Published Build

When it came to configuring and publishing builds of my app I made use of Expo Application Service (EAS). EAS is a cloud service that compiles and signs Android/iOS apps with native code in the cloud. In layman terms EAS takes my React Native code, compiles it into native code for each platform, packages it nicely and handles all the signing and certification. This is often a complex process that most developers dread made into a simple chain of console commands. This build process began by configuring my builds by entering **`eas build:configure.`** This command ensured that my code was build ready and no errors would appear. After this I ran **`eas build --platform all`**. This compiled my builds for both platforms. This command also guided me through the process of signing the app and creating certificates for distribution.



First EAS builds appearing on the cloud console

Once this had finished my builds appeared on the Expo developer cloud console. From there I could download the .apk versions of my app and send it to Android devices or run a further command **`eas submit -p ios`**. This command prompted me to sign into my Apple Developer account then submitted my build to Testflight. Once EAS had submitted my app onto Testflight I then had to log onto the Apple developer console to submit my build for Testflight verification. The Testflight verification process then takes roughly 18 hours before my app is returned for me to test.

First build accepted on Testflight for testing

On Wednesday 3rd August my first build was approved for Testflight rollout. Initially this was a massive relief as it gave me a lot of time to send out the app before the first round of testing on Thursday 11th to Saturday 13th. This relief went to stress quickly as when I went to test the app, the app would crash when selecting a sport of your team. To make matters worse this bug couldn't be replicated on any test build. My time frame for fixing this was by tomorrow, as I was heading away over the weekend, and I was required to have the app sent to sports coordinators by the Tuesday 9th and therefore fixing it on Monday 8th would be too late. Therefore, I had one day to identify and isolate the bug then make changes and republish the build. To make matters worse I found a StackExchange post that had the same issue, but it had no comments on it.

Meeting with Nigel Pitts to Discuss Technical Issues

Due to how important it was that I correctly identified this bug and fixed it in a short time frame I organized a meeting with a family friend – and software developer - Nigel Pitts to help me identify the bug. I organized the meeting for the night of Wednesday 3rd allowing me the following day to implement a fix and then resubmit it to Testflight. Nigel has worked in the tech industry his whole life and has experience with other front end JS frameworks such a View therefore he would hopefully be of help.

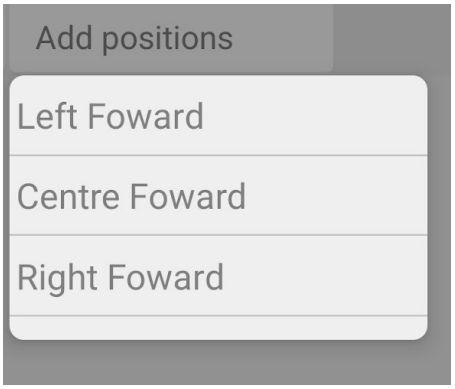
Initially he was quite perplexed by the bug as I showed him how it crashed on a published build but not a test build. First, we tried nuking all the project dependencies then rebuilding them – as when a build is compiled by EAS all the dependencies are rebuilt - but that had no impact. This was confusing as it was not an issue

with dependencies on the published build which we initially assumed. After a bit more thinking we both concluded that the app only crashed when interacting with a component RNPickerSelect. Therefore, he suggested I find a replacement Drop down select component and change it with RNPickerSelect component and rebuild it and hope it works.

Resolution of Technical Issues

The component I found to replace it was SelectDropdown[24]. When selecting a replacement there were a few options but I decided to pick this option as it was easiest to setup, and I was able to adapt all my old code to it with only minor changes required. The RNPickerSelect took a list of items as a prop. Each item in this list was an object with two fields value and label. An option in the RNPickerSelect was rendered for all objects. I was able to reuse this list of options for the SelectDropDown by making the text to appear for each option be the label field of its object. I also had to change the onValueChange prop to the – functionality identical but just has a different name for this component - onSelect prop. The final additional bonus of this new component was its added styling capability. It allowed styling to every part of the component in comparison to the very limited styling allowed on RNPickerSelect. I swapped all occurrences of RNPickerSelect in my code base with SelectDropdown then rebuilt the app before submitting it to Testflight. After another tense 18 hours wait the build was accepted and the crashes were resolved.

```
//Old RNPickerSelect code from earlier in devel
<RNPickerSelect
    onValueChange={(value) => {
        if (!playerPositions.includes(value) && value != null)
        {
            addPositionToPlayer([team_id,playerId, value])
        }
    }
    placeholder={{ label: 'Add positions', value: null }}
    style = {pickerSelectStyles}
    items = {positionSelectionData}
    useNativeAndroidPickerStyle={false}/>
//New picker select code
<SelectDropdown
    data={positionSelectionData}
    onSelect={(selectedItem) => {
        //Check if player isn't already assigned that position and if so add that position
to the player
        if (!playerPositions.includes(selectedItem.value) && selectedItem.value != null)
        {
            addPositionToPlayer([team_id,playerId,selectedItem.value])
        }
    }}
    buttonTextAfterSelection={(selectedItem) => {
        //Handles rendering of the text on the button
        return selectedItem.label
    }}
    rowTextForSelection={(item) => {
        //Handles rendering of text for each item in list
        return item.label
    }}
    defaultButtonText={'Add positions'}
    buttonStyle = {styles.dropDownButton}
    buttonTextStyle={styles.dropDownText}
    rowTextStyle={styles.dropDownText}
    dropdownStyle={styles.dropDownStyle}/>
```



New Select drop down in use. Users have to scroll to see all the options

Social Basketball Testing and Meeting with M W

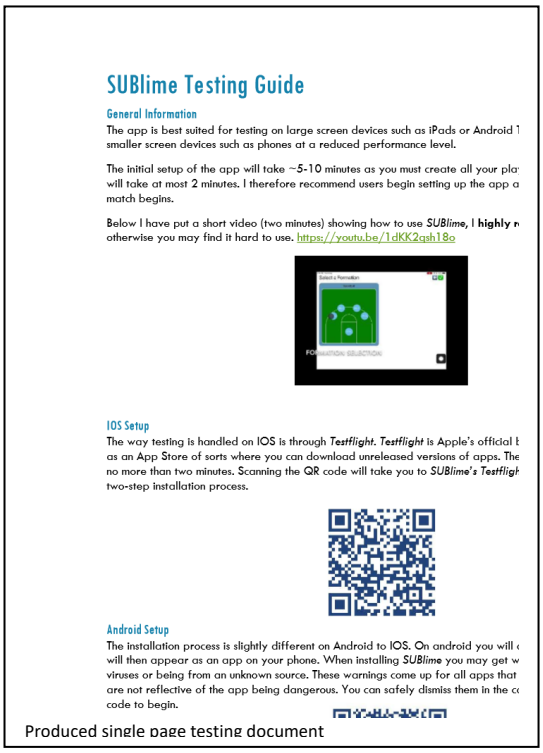
In the hecticness of this week I also met with M W . Social basketball is on every Friday night and M W is responsible for running it at . B E had discussed with M about testing the app out for social basketball. When I met with him, I gave him a tutorial of the app then discussed the best method to approach testing. He suggested that a single iPad should be used throughout the night with the app loaded onto it which he would pass between teams. This idea was a huge success as it made it very easy for testers to pick up and use the app as they did not have to download anything.

Curation of Testing Information

The critical bugs were now fixed, and I had the app both verified on Testflight and available as a downloadable apk. My next challenge was how I was going to send out information about the app and installation to testers. From discussions earlier with A K we both decided the best way to handle the distribution of the app was for me to collate a single page document that would have a brief explanation of the app, installation information, a link to a feedback form and contact information. A K said he would then send this document out to coaches for testing. Other stakeholders I talked to such as S and J agreed with approach.

[24] https://www.npmjs.com/package/react-native-select-dropdown#License

I first wrote a brief description of the app that focused on why testers should use it. I next made a short 90 second video running users through using the app. This would serve as a basic tutorial to less tech savvy users In this video I ran users through making their first team and subsheet. In this video I included subtitles at the bottom to add increased clarity to my actions. The final thing I did in this video was to include a tap gesture that made a highlighted area wherever I tapped or dragged. This was to show potentially confused users what buttons to press to progress. I opted to include this video as opposed to written instructions on how to use the app as I believed instructions are better communicated through showing as opposed to telling.



SUBlime Testing Guide

General Information

The app is best suited for testing on large screen devices such as iPads or Android 1 smaller screen devices such as phones at a reduced performance level.

The initial setup of the app will take ~5-10 minutes as you must create all your pla will take at most 2 minutes. I therefore recommend users begin setting up the app a match begins.

Below I have put a short video (two minutes) showing how to use *SUBlime*, I highly r otherwise you may find it hard to use. https://youtu.be/1dKK2qsh18o

IOS Setup

The way testing is handled on IOS is through *Testflight*. *Testflight* is Apple's official t as an App Store of sorts where you can download unreleased versions of apps. The no more than two minutes. Scanning the QR code will take you to *SUBlime's Testfligh* two-step installation process.

Android Setup

The installation process is slightly different on Android to IOS. On android you will will then appear as an app on your phone. When installing *SUBlime* you may get w viruses or being from an unknown source. These warnings come up for all apps that are not reflective of the app being dangerous. You can safely dismiss them in the c code to begin.

Produced single page testing document

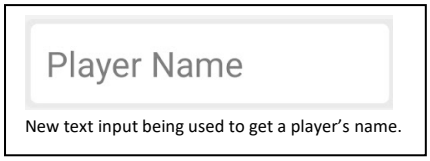
In terms of installation information, I created a testing group on Testflight that had a public link anyone could join. For the Android apk I created a Google Drive and created a public link for that. I deliberately decided to share the app on Google Drive as most Android phones come with Google Drive preinstalled. Therefore, when they scanned the QR code they'll download it into their Google Drive where they can find it easily. Whereas if it was a Dropbox link or other alternative file sharing platform, they would download the .apk into their file system resulting in the app being harder to find.

With links for both platforms, I created a QR code that would take users to the download link. I put in a QR code as opposed to a web link, as a QR code forces users to take out their phone and scan the link on their phone ensuring the link is opened on their phone. Whereas with a link people may click on it and open it on their laptop and get confused when it doesn't install. Initially I also had additional instructions to walk users through the installation process but decided against it as a) the installation process was very simple and self-explanatory – evident by my mum even being able to download it and b) I felt it would scare people away from trying out the app if the installation instructions are long.

The final thing I put on the document was a survey form. The survey asked if the user had any technical issues and asked what device they used, as technical issues may be specific to a device. I also asked a range of questions such as how easy the app was to use, if they like the looked of the app and whether they would use it again. I ended the survey with open-ended questions asking if they have any suggestions to add new features in. I also emphasised the incentive I had added for completing the feedback form which was $100 gift card from Winnie Bagoes.

Last Second Bug testing / UX changes

With the distribution document setup, I passed it around my digitech class to see if people were able to install my app based on those instructions. There were no issues with the installation process; everyone was able to scan the code and download the app onto their phone. However, when people started using the app a few critical bugs and UX issues were revealed. The bugs which were Android exclusive stopped people from progressing through the app and UX issues resulted in users consistently getting confused and failing to make a subsheet. As these issues were critical, I made a decision to delay sending the app out to coordinators by a day and instead send it out on the 10th of August. This was not ideal as it meant the app would get to coaches later than I wanted and give them less time to be aware of the app and set it up for their team for testing prior to their game. However, this was a necessary decision to make if I wanted users to be able to use the app.



New text input being used to get a player's name.

The critical bug was that the text input would instantly close when opened on Android devices resulting in users being unable to input data such as players name. Like the described issue with RNPickerSelect above this could not be recreated in a test environment and was therefore very hard to debug. I found a chain

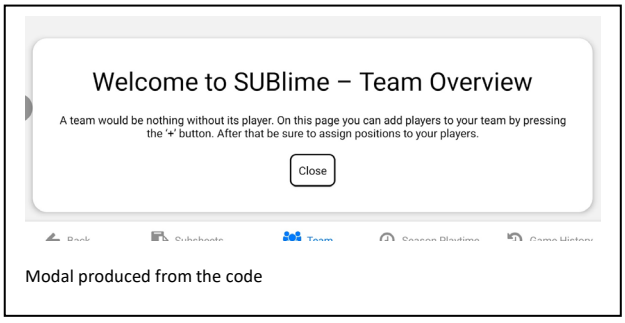
of posts on StackExchange of people who also had this issue[25]. The solution provided was quite complex and included playing around with dependencies version, so I just opted to install another Textinput package and use that instead due to time constraints. The package I used was called react-native-element-textinput[26]. The new Textinputs implementation into code is identical to the default Textinput component I was using prior, and it was only diffrent in appearance. Therefore, all I had to do was uninstall the old Textinput package and install the new and didn't have to worry about changing / updating props.

The next and more problematic issues was with UX. Users consistently got stuck on the subsheet creation page. What would happen is that when they created their team the app would prompt them to create their first subsheet. This would lead them to creating their first subsheet before creating any players. As a consequence, they would have no players to add to their subsheet and would become confused when nothing happened. I came up with two solutions for this. The first was changing the navigation route through the app. I decided to change the route to navigate users to the team management page before they created a subsheet. This way the user added players into their team before creating their first subsheet. This meant they wouldn't become confused when creating this subsheet as they now had players to add to it.

The second and more general fix I implemented to users having a bad UX was a tutorial system. The plan for this tutorial system was a pop-up that appears every time a user reaches a page for the first time. This popup would explain to the user how to use the apps feature on that page. On a user's subsequent visits to a page the popup wouldn't appear. To add this, I first added a new value to each team object called team_tutorial. This was an array that contained Boolean values for whether each tutorial had been viewed. Each index in the array corresponds to a page ie 0 – create subsheet, 1 – manage players & 2 – time overview etc. I next added a modal to each of the pages were there was a tutorial prompt. The visible prop of the modal was set to the respective Boolean value for whether that modal had been viewed. When the modal was dismissed the Boolean value for that tutorial was set to false, meaning that it would not be displayed in the future.

```
 //New value that is added to the team object. It is an array that contains 5 true
values initially each corresponding to a tutorial. Once a tutorial is dismissed the
value is set to false and then it isn't displayed again.
team_tutorial: [true,true,true,true,true]

//Modal that displays the tutorial a separate modal is used for each tutorial on
each page
<Modal
      {/*Modals visibility is determined by weather the team tutorial value for
that tutorial is true or false*/}
      visible={teamState.team_data[adjusted_team_index].team_tutorial[1]}>
      <View style={styles.centeredView}>
        <View style={styles.modalView}>
          <Text style{styles.modalText}>>Welcome to SUBlime - Team Overview</Text>
          <Text style = {{textAlign:'center'}}>{'A team would be nothing without
its player. On this page you can add players to your team by pressing the '+'
button. You can select as many positions as you like per player.\n\nOnce you have
added some players press the 'Subsheets' button to continue\n'}</Text>
          <View style = {{flexDirection:'row'}}>
            <Pressable
              style={[styles.button, styles.buttonClose]}
              {/*Calls the reducer that sets the value in team_tutorial at the
index of the tutorial to false*/}
              onPress={() => {updateTeamTutorial([team_id,1])}}>
              <Text style={styles.textStyle}>Close</Text>
            </Pressable>
          </View>
        </View>
      </View>
</Modal>
```



Modal produced from the code

After I made these changes, I remade the tutorial video as the way users moved through the app had changed and therefore it was necessary to update it to reflect this change and avoid confusion.

First Round of Testing

In the first testing round the app was sent out to the following groups for testing

Sport	Total Testers
Social Basketball	Setup for the whole Friday night and intended to be passed between 5 teams with the help of M[redacted] Westrupp
Canterbury Netball	Sent out to 160 Junior Netball Coaches to test
Preparatory School	Sent out to ~20 hockey teams, ~20 netball teams, ~10 basketball teams and ~10 football teams
J[redacted]s Football teams	Sent out to ~12 teams. These team ranged in skill level and age level.

[25] https://github.com/facebook/react-native/issues/33164

[26] https://www.npmjs.com/package/react-native-element-textinput

Changes Made Following Feedback from First Round of Testing

The feedback received largely varied. Some feedback was positive other negative and then some at times aggressive. The feedback stemmed around a few things: first UX, second bugs and issues specific to devices and third new features. Fortunately, no feedback was received surrounding crashes – which was a huge relief. I had limited time to address bugs and make changes due to the two days turnaround from feedback to resubmitting meaning I focused on the most pressing issues. First to do with UX.

Feedback: *'after a while of adding players to the subsheet it is hard to see the time'*

To fix this I added a minute timeline across the top of the subsheet. The timeline was created by generating an array that contained a value for each minute in the interval. This array was then mapped to text components where a number was rendered for each minute.

```
Array.from({length:interval_length},(interval_length*(current_interval-
1),(interval_length*current_interval))=>(interval_length*current_interval+1)).map((prop) => {
    return(
       <View key = {prop} style =
{{flex:1,justifyContent:'center',alignItems:'center',paddingTop:10,borderRadius:1,borderColor:'red'}}>
          <Text style = {{fontSize:20}}>{prop}</Text>
        </View>
    )})
```

Feedback: *'i had to rewatch the video a couple times to understand how to add players to the subsheet '*

This feedback showed people were struggling to figure out how to add players to the PositionSlider. To make it more obvious I changed the numbers on each minute of a PositionSlider to a '+'. People intuitively press plus making it likely that if they don't understand what to do, they'll press the '+' hoping it will do something then it'll prompt them to fill out the PositonSlider and complete their subsheet. Also, with time across the top of the subsheet it was no longer necessary to show minute values on the PositionSlider.

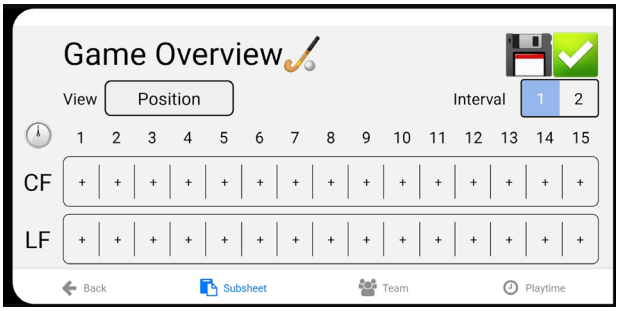
The next chunk of feedback was to do with device specific issues. I received constant feedback about the PositionSlider that I found to be device specific.

Feedback: *'when I try and give players more time on the subsheet, the drag freezes'*

This feedback initially confused me as I couldn't replicate this in my testing on iOS devices. However, I noted a trend with this feedback; it only came from Android users. I then tested the app on my phone (I had experienced this issue on my phone before but wrote it off to my phone being old and slow). What I noticed was that the scroll of the FlatList was conflicting with the users drag to allocate more time to a player on a PositionSlider. Mid-drag the user may accidentally slightly scroll up or down on the FlatList causing the drag to cancel as the Flatlist scroll now takes priority. This would cause the drag to appear frozen, hence the complaints. To address this, I would need to disable the scrollEnabled prop on the FlatList when the user was dragging a player. By disabling the FlatList from scrolling this conflict no longer occurs. To check whether the user was dragging, I checked if the moveDir state variable was set to null, as it was defined null when no drag was taking place.

```
scrollEnabled={moveDir==null}
```

Other feedback specific to devices was that on phones you could only view one position at a time on the subsheet. I addressed this by making the PositionSliders smaller by reducing their height style prop. I played around with the height value for a bit until I reached a height where it was small enough to allow multiple positions to appear on smaller screens but large enough for the touch not to be finnicky.



Revised subsheet creation page based on three pieces of feedback received. Added numbers across top, added pluses and made PositionSliders smaller.

Finally, to do with new features. The main feature requested was the ability to create your own formations. The inability to do this led to passive aggressive hatemail from one tester. I didn't have time

to implement this before the next round of testing, so I added in some new default formations as a temporary fix. For example, I added 3-5-2 and 4-5-1 to football as new defaults.

Second Round of Testing

After all these changes were made and the new build was approved onto TestFlight I updated my testing documents, adding in a new QR code to install the new build. I then sent this updated document to the sports coordinators who sent it out to their teams. The testing groups for this week were the same as last week apart from a discontinuation of the social basketball testing. This was because the development time I would lose by having to give the iPad I used for testing, to M over the weekend for social basketball, did not met the benefit of the feedback from the five teams that would use it, when I already had a massive testing base.

Sport	Total Testers
Canterbury Netball	Sent out to 160 Junior Netball Coaches to test
Preparatory School	Sent out to ~20 hockey teams, ~20 netball teams, ~10 basketball teams and ~10 football teams
Juans Football teams	Sent out to ~12 teams. These team ranged in skill level and age level.

Meeting with Mainland Football


Kia Ora

My name is I am the Community Development Officer at Mainland Football.

kindly passed your email on to me, great work on your project it looks very interesting. Would you be free sometime next week to have a zoom meeting with me so I can get some further information and detail about the app?

I am happy to fit a zoom meeting around your schedule if you let me know a time and day that would work for you.

Ngā mihi | Kind regards

- Community Development Officer

Email received from Mainland Football

Earlier in development sent out emails to three major sports governing bodies in Canterbury: Christchurch Netball (who I had rolled out testing for), Canterbury Basketball (who ghosted me for now) and finally Mainland Football. Mainland Football is the regional governing body of football in Canterbury, West Coast, Nelson and Malbrough. They are responsible for the overseeing and organization of 51 sports clubs. My initial email to Mainland Football earnt a reply from their CEO M F D, who applauded the idea. He told me he had passed the idea onto his technical team. After this email I heard no follow up for a month leading me to assume they were no longer interested and therefore I contacted J C to discuss setting up the app for Football.

However, after sending out the testing information for the second round of testing, I received an email from Mainland Football's Community Development Officer A K asking to setup up a time for a Zoom meeting about my app. In preparation for the meeting, I prepared a PowerPoint to screenshare with A that had QR codes to install the app then various screenshots of it in use. I created this as I was unable to show A the app physically on an iPad but still wanted him to try out the app.

The meeting with A began by him installing the app then us discussing the features. He really liked how the app could be used to address an unequal gametime distribution. He stated how in his role as Community Development Officer he commonly had to mediate complaints from parents about their kids not getting enough gametime. He explained how it is often hard to resolve these complaints as there is very little concrete evidence to support sides' case. He believed that *SUBlime* would be useful in resolving these complaints due to it providing records of gametime across a season. Adam did identify a minor design issue on smaller screen phones (like the one he used) that when playing a game with 45-minute intervals – which is standard in football – all of the buttons for each minute on the PositionSlider would become very small – as 45 minutes were being displayed - making them hard to tap. A work around he suggested was implementing subintervals that broke longer intervals into two parts when displaying them. Additionally A noted that he would like to see the implementation of a system that allows users to create and customize their own formations beyond the default options.

The final topic discussed was testing. The junior football season was basically over – with the last game being this week and A stating it would be to late at this point to roll it out - and therefore I was unable to carry out testing in the short term. However, they did offer me the opportunity to carry out testing at junior football tournaments that were going to be carried out later in the year. This was a great opportunity as it provided access to a large volume of testing in a short period of time. This testing would be done during the term 3 holidays and therefore feedback wouldn't come in until the latter half of the holidays. At this point it

would be too late to make changes and implement feedback as I would have shifted my focus to report writing and the publishing of *SUBlime* to app stores.

Changes made because of Second Round of Feedback

The feedback was generally more positive in the second round of testing, but minor complaints still existed.

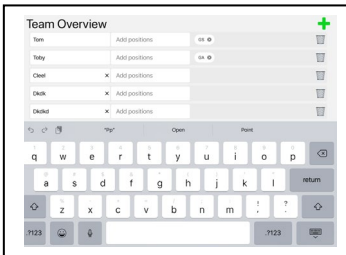
Feedback: *'I made the same feedback last week. It takes too long to add positions to players'*

The first pieces of feedback were to do with it taking a long period of time to assign positions to players. This was a change I had to address due to the setting up of positions of players in a team being one of the first things users did. If this took too long, it could potentially turn away future users. I implemented two changes based on this feedback. The first was removing the need to press the plus on the PlayerTab to assign the selected position to a player. Instead, I made it that when the user selected a position from the dropdown it would assign it to the player instantly without them having to press the plus. This may have saved one – two seconds per position assigned which may not seem like much but if a coach was assigning 30+ positions across a team it could save upwards of a minute.

The second change was to add a 'positionless' option for when users created a team. If a team was positionless, every player was assigned all the positions meaning that no time had to spent assigning positions to players. This change was mostly targeted at junior teams were coaches are encouraged to play players at a range of positions. Upon implementing this, I came to the realization that this change would cause crashes with users who had used the app prior to this update. This was because these users would have setup teams without a defined positionless value. What this meant was the app would crash when checking the positionless value as it didn't exist. To counteract this, I wrote a reducer that added the positionless field to any old teams and set it to false.

```
//Position array is initally defined as empty. If positonless is true the aarray is filled with all positions then assigned to the player
let position = []
if(positionless == true) position = positionSelectionData.map(item=> item.value)
createPlayer([team_id,{ id: current_player_index, name: '', positions: position, color:, randomColor(), selectedPos: null}])
```

Feedback: *'not a bug more an annoyance. When I open the keyboard it covers up the text input and I cant see what I have written'*



Keyboard blocking user from seeing currently typed name.

The next feedback was to do with a bug that when you went to enter the name of a player the keyboard would cover up the Textinput making it impossible for you to see the name that you were typing. This made it harder to know if you were spelling a player's name right. To counteract this, I used a KeyboardAvoidView [27] component that I wrapped around the PlayerTabs. Consequently, the PlayerTabs shifted when a keyboard was opened to not be covered.

Feedback: *'app crashes on subsheet select page for no reason. Very annoying'*

The bug was a crash caused when a user went onto the select subsheet page. The bug would only occur on subsequent uses of the app and not when they used it the first time. I played around with this for a bit and managed to replicate the crash by doing the same steps. I was very confused as to what may be causing this crash so part by part, I removed components from the select subsheet page to identify what the issue could be. In the end what I found was that the crash was being caused by the date the subsheet was saved at being rendered in the SaveView. I did a bit of research online and found people with similar issues. What I eventually concluded was that it was something to do with the way Redux Persist was storing my date time values. This was because the date time values only caused crashes when being displayed after being saved. This led me to look at how Redux Persist stored data and what I found was that Redux Persist serializes your data to store it. Date time strings are non-serializable therefore when being deserialized no value returns. This explains the crash as no date time values are returned when they are expected to be. My work around for this was to store the date time in a serialization friendly format. This format was on object of fields of

[27] https://github.com/APSL/react-native-keyboard-aware-scroll-view

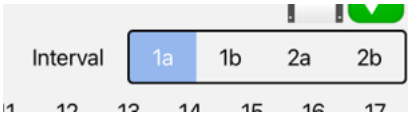
year, month, hour and minute. This data was able to be serialized then deserialized. I then had to go through an extra step of returning this date time object back into string form to be displayed.

```
//The saved date is stored as an object then saved to the save_data ds
const current_date = new Date()
const savedDate = {year:current_date.getFullYear(),month:current_date.getMonth(),day: current_date.getDate(), hour:
current_date.getHours(),minute: current_date.getMinutes()}

//The format time function then turns this object into a date time and formats it
function format_time()
{
    const time_data = item.schedule_date
    const time = new Date(time_data.year,time_data.month,time_data.day,time_data.hour,time_data.minute)
    const options = { hour:'numeric',minute:'numeric', year: 'numeric', month: 'long', day: 'numeric' };
    return time.toLocaleDateString('en-NZ',options)
}
```

Feedback: *'when I set up a subsheet for my football team the plus are too small to tap'*

The next change was made based on feedback from users on small screen devices and stakeholder consultation with Mainland Football. This change was the implementation of subintervals. The issue was that some sports such as football have intervals as long as 45 minutes. Consequently, this resulted in two issues. Firstly, on small screen devices each of the minute tiles became so small they were untappable due to 45 of them being on screen for each position. The second issue was with lag, as discussed above the PositionSlider still faced minor performance issues that became a lot worse when 45-minute tiles were rendered per PositionSlider. This issue was pointed out in both feedback from J[redacted] C[redacted] and other football coaches alongside A[redacted] when he tried out the app in our meeting. My solution to this was to implement sub intervals were any interval longer than 30 minutes would be split into sub interval a and subinterval b. Each half the length of the original interval.



New position toggle with sub intervals selectable.

The first change I made was setting up the new interval selector to allow users to pick sub intervals. Previously to make the interval selector I had mapped a list of length of total intervals making a button appear for each selectable interval. In this new interval selector I planned to use similar approach with a list that contains all the selectable intervals. To create this list, I first determined if the interval was longer than 30 minutes and if it was therefore nesscary to have sub intervals. If sub intervals were necessary, each interval longer than 30 minutes was broken into a lower section and upper section each containing half of the minutes of the original interval. If the interval had an odd number of minutes the upper section would take the extra minute. For example, an interval of 45 minutes is broken into sub intervals with a lower section of 22 and upper section of 23. I then created an object for each of the sub intervals with fields of tag (which was name of the interval to display a for lower section b for upper section), upperintervaloffset (how large the upper interval was, the lower interval could then be determined by subtracting this off interval length), intervalValue (which was the current interval that the sub interval was within) and lower (a boolean value that was true for lower section and false for upper section). This list of objects was then mapped and displayed with a selectable value like the last toggle interval.

```
//Create the interval selector data
let interval_selector = []
if(intervalW >= 30)
{
    //If its greater than 30 the interval is split into two parts. The lower section is a and upper section is b. We need to
find the offset for the upper section
    const upper_interval_offset = Math.ceil(intervalW / 2)
    //Now add the data for each interval adding in the lower and upper section
    for(let interval = 0; interval < intervals; interval ++)
    {
        interval_selector.push({intervalTag: (interval+1)+'a',upperIntervalOffset:
upper_interval_offset,intervalValue:interval+1,lower:true},{intervalTag: (interval+1)+'b',upperIntervalOffset:
upper_interval_offset,intervalValue:interval+1,lower:false})
    }
}
else
{
    //Otherwise just create an interval selector without sub tags
    for(let interval = 0; interval < intervals; interval ++)
    {
        interval_selector.push({intervalTag: (interval+1),upperIntervalOffset: intervalW,intervalValue:interval+1,lower:true})
    }
}
```

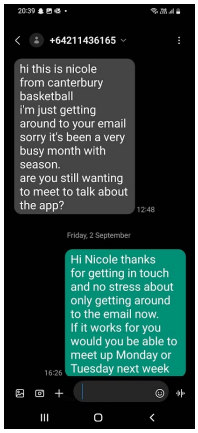
The next step was dealing with the constraints of sub intervals in the rendering and logic of the PositionSliders. As these constraints had to be applied to a lot of features within the PositionSlider such as what minutes to render, what minutes to allocate to a player based on user drag and what the drag direction is, etc. I decided to create two variables for the lower and upper bounds to apply rendering and logic to. These variables were setup as ternary such that they have different value if the upper or lower section is being displayed. I then applied these variables to my code.

```
//Lower and upper bounds of minutes to display
const intervalLowerBound = (lower? intervalLength*(currentInterval-1):intervalLength*(currentInterval-1)+upperIntervalOffset)
const intervalUperBound = (lower? intervalLength*(currentInterval-1)+upperIntervalOffset:intervalLength*(currentInterval))

//Examples of these bounds being applied. First is to render minutes only between those bounds. Second is only applying logic
between bounds
if(i >= intervalLowerBound && i < intervalUperBound )
if(minutesPlayed >= intervalLowerBound && activeGameInterval == currentInterval)
```

Meeting with Canterbury Basketball

In the twilight of the projects development, I got a message from N G . N is the junior events and competition officer for Canterbury Basketball. She reached out to me over text showing interest in the app and wanting to make a time to discuss it. In our discussion I showed N the app and she expressed how invaluable it would be to have such a tool in junior sports. Unfortunately, due to the fact it was now the last week of term 3 there was no chance to test the app out with junior basketball teams. However, N stated the potential to test the app out next year.



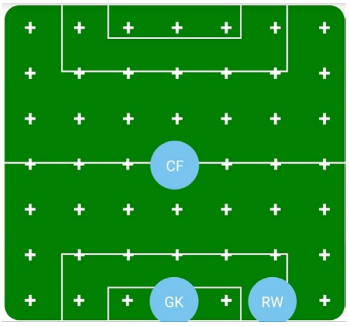
Messages with Nicole

Formation Selection

The final major feature I wanted to implement before shipping the app was the ability for users to create their own formations. Currently in the app all the provided formations were hardcoded in, and users had no ability to add or edit their own formations. As coaches make use of a wide range of unique formations this was not ideal and could turn away potential users as the app doesn't have the formation they wish. This was reflected heavily in feedback with users asking a lot for the ability to add their own formations and stakeholders such as J and Mainland Football who wanted this feature added.

First, I had to overhaul the backend of how formations were stored. Previously they were hardcoded as a list of objects for each formation and then these formation objects were rendered on the formation select screen. As these formations could now be added, deleted, and edited I had to make this list editable. To do this I added a new field for each team object called team_formations. This field would contain the list of formations. I added the already existing formations to this list as default formations. After this I wrote new reducers to allow the addition and deletion of formations from this list. I also had to add in a check when the user loads a team to determine if the team had the team_formation field and if not add it in. This was necessary to allow for backwards compatibility of team's objects created prior to this change.

Next, I could move onto the front end. First, I added a new page to the team overview tab navigator for formation management. On this page I reused the formation selection component allowing all the formations to be displayed. Next, I wanted to add the ability to create formations, to do this I added Pressable with a plus icon. When pressed it would open a modal that would allow users to create a formation. My plan for this formation creation page would be to display a pitch. Users could tap anywhere on this pitch to add a specific position at that point. Due to the limited time frame with setting this up (It was now term 3 holidays) I attempted to make use of as much previously existing code as possible to save development time. Therefore, I decided to reuse the grid-based position rendering system I had used earlier to display positions on formations. The change I would make would be that if no position was assigned at a place on the field a SelectDropDown would be rendered instead allowing users to add a position at that place. The first step in setting this up was creating a formation 2d array to pass into the rendering code. I set this up as a state hook and set all values in this 2d array to -1. By being set to -1 it

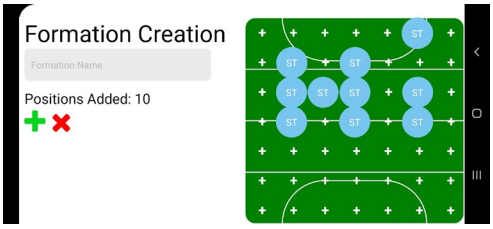


Example pitch. -1 in 2d array are the + icon. Some positions have been added.

signifies no position was assigned at that place in the formation. I then applied conditional rendering to render a SelectDropDown allowing a position to be selected if the position value was -1. Next, I added the functionality of adding a position to the formation. When a user selected a position from the SelectDropDown I changed the value in the 2d array to that associated position.

On the left-hand side of this formation creation modal, I added in Textinputs for the user to enter the name of the formation. I also added a counter of the current positions in the formation by iterating though the 2d array and incrementing a counter by 1 for all non -1 values. I deliberately added this so coaches don't have to constantly recount how many positions are in their formation. The final piece of functionality I added was the ability to delete positions from a formation. To do this I wrapped all the formation icons in a Pressable component. When pressed an alert would appear confirming whether the user wished to delete that position. When pressed I would update the state hook and set that value to -1 therefore making the position disappear and a plus appear in its place to add a new position.

With the ability to create a formation I now had to make it able to be saved. I added a plus Pressable that when pressed would create a new formation object with the formation 2d array inside of it and a unique id. This would then be saved to formation_data. When the formation was saved the modal closed and the new formation appeared on the formation management page.

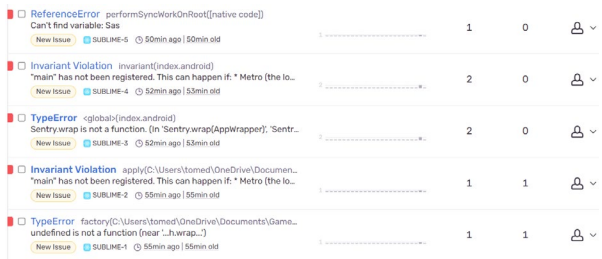


Complete formation creation page

The final consideration I made was whether to allow users to create a formation if they didn't have the required number of positions for their sport in it ie 11 football, 15 rugby and 7 netball. I decided to not have this requirement in order to increase accessibility and flexibility. For example, in another country they may play 6 aside football as opposed to 7 aside football at a junior level. Currently the app doesn't cater to 6 aside football locking these users out of using the app. However, by having no restrictions on formation sizes a coach could create a 6 aside formation thus allowing them to use the app. Therefore, by placing no restrictions on formation size the accessibility of the app is massively increased.

Setting *SUBlime* up for a Postlaunch Environment

Post launch of the app, I will become disconnected from the users of my app. Previously, all testers of the app had access to my contact details and a feedback form in which they could report any bugs or glitches they found. Once the app has launched, I will hopefully have users from all over the world. These users have no way to submit feedback forms or contact me if they find any issues. Therefore, it was necessary for me to set up a crash logging tool that records all crashes that occur of the app, allowing me to address these crashes without being able to contact users.



Sentry Dashboard for SUBlime showing crash details of recent crashes. Ironically, the first few crash logs were crashes caused by Sentry not being set up correctly.

I looked at a range of tools that collect crash logs post launch such as Firebase, Datadog and Sentry. In the end I decided on Sentry as it was free of charge and can be setup easily in the Expo development ecosystem. The process of setting up Sentry was very boilerplate heavy. I first had to create a Sentry Account then generate various API keys before linking the Sentry account to my code. Now when the app runs it is connected to Sentry. Any crashes that occur while the app is in use is logged to my Sentry dashboard. A full stack trace of the crash is provided allowing me to specifically identify what aspects of my code caused the crash. Other data such as version model and operating system of the device is also noted. This would be useful in attempting to address platform specific glitches. With this setup I now have confidence moving beyond the launch of *SUBlime* that I will be able to address any bugs and glitches.

In addition to this I intend to remain close with the regional sports bodies I worked with – Canterbury Basketball, Christchurch Netball and Mainland football – to collect qualitative feedback. All of these groups were happy to support the use of the app into the future.

Submission of App to Google Play Store and Apple App Store

With the development of the app now complete. I moved onto the next task of releasing the app on the Google Play store and Apple App Store. Although it would be easier to only pursue releasing the app on Google Play Store due to its less strict regulations, I wanted to launch cross platform to increase accessibility.

Earlier in development I setup my Apple developer account but was consistently declined when attempting to make a Google Play Console account. Luckily, when I tried to make an account again this time it worked (this was presumably due to an 18-year-old age requirement to make an account that I didn't meet earlier). Next, I had to setup the app project on Google Play Console. To do this I had to setup a Google Cloud Services account. Create a cloud project, generate a private key then link that to a created project on Google Play Console[28].

Next, I needed to gather the relevant information and details to create a listing for the app on the play store and app store. For this I was required to have an app name, app icon, app feature graphic, app description, a website for the app, screenshots of the app in use on different platforms and other pieces of minor information. First, I setup the name of the app. Unfortunately, the app name *SUBlime* was taken already so I changed the official app name to a variation of it: *SUBlime Substitution Manager*.

Original App Icon

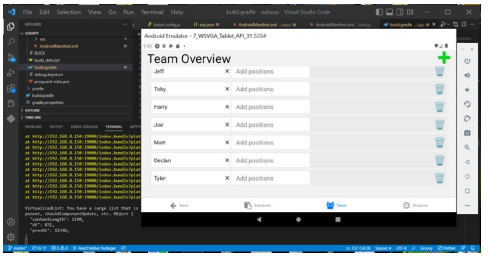
Revised App Icon

The next task was creating an app icon. Both Android[29] and Apple[30] provide very in-depth design specifications for creating an app icon. I began by reading these and took note of design conventions and what sizes the icons would need to be. I also noted the importance of having a good logo as it was the first thing a prospective user saw therefore it must effectively capture what the app does. To help me come up with the design I organized a meeting with A K to discuss what the key elements of the logo should be. We both agreed on three key elements for it. Firstly, some sort of sports iconography to indicate to users this is a sports app. For this we decided to use a football as it's the most universal sports symbol. Secondly, arrows that are symbolic of substitution or swapping. Thirdly, the app name. I then outsourced the design of the logo to my younger sister who is a lot a better at graphic design than me. After a while she returned to me with the design. I then sent the design to A K for feedback, and he suggested I turn one of the footballs into a netball, so the app looks more like a general sports substitution app as opposed to a football substitution app. After the changes were made, I then added the new icon into the apps files and rebuilt the Android and iOS dependencies to reflect the change in icon.

Next was the app description. For this I rewrote app blurbs I had used in the past when sending the app out to stakeholders. I began the app description by explaining how and why *SUBlime* uniquely solves the issue of unequal gametime and then bullet pointed all of the app's features. Following this I created the app's website. On this website I had to include a privacy policy for the app and contact details. As I was limited for time at this point in development, I decided against building a website from the ground up and instead made a free Wix site[31]. On this site I made a home page with the app logo, a contact page with my contact details and a contact form, and a page with a privacy policy. To make this privacy policy I made use of an online privacy policy generator as a lawyer was not in the budget for this project. This generator asked me questions

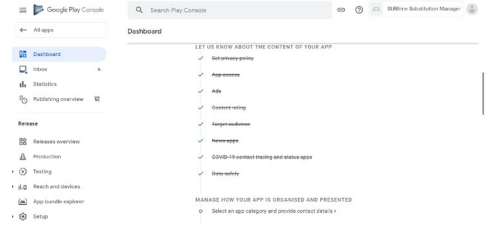
[28] https://pagepro.co/blog/publishing-expo-react-native-app-to-ios-and-android/
[29] https://developer.android.com/distribute/google-play/resources/icon-design-specifications
[30] https://developer.apple.com/design/human-interface-guidelines/foundations/app-icons/
[31] https:// .wixsite.com/

ranging from what age uses my app to how I store data and what data is stored[32]. It then generated a privacy policy which I put on this webpage.



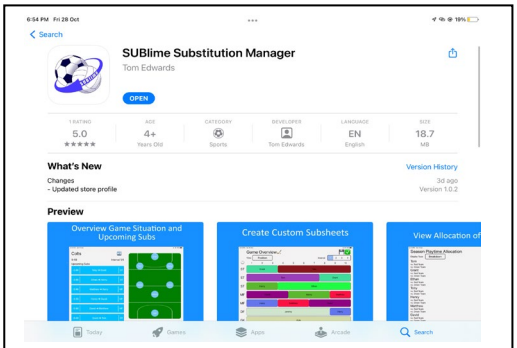
Launched emulator of a 7 inch android tablet

The final and by far the most painful step of creating the app listing was creating screenshots of the app. Both Android and Apple are very specific about their app screenshots and require a range of screenshots from a range of device sizes. To begin the design process, I looked at app listings on app stores to determine what other apps did. I noted two different groups of designs. The first was just straight screenshots of the app and the second was a screenshot of the app slightly zoomed out allowing text to be placed around it. I preferred the second design as I it allowed more information to be communicated about the app in a single image. Various Appstore screenshot generation tools exist that convert app screenshots into screenshots to display on an app listing. However, I didn't use these tools for two reasons, firstly most of them cost money and secondly none of them had landscape screenshot support and my app was landscape only. This meant I would have to create these app screenshots myself. To create these app screenshots, I would first have to get screenshots of the app in use on the 7 different screen sizes that are required across the Android and Apple stores. I don't own all these 7 different devices therefore I had to make use of an emulator of the device on my laptop. As I was using an emulator to get my screenshots it took a long period of time as my laptop is poorly speced and struggled to run the emulator. Additionally, the drag functionality of the PositionSlider was hard to do on the track pad. I got four screenshots on each device I emulated. These showed the subsheet creation page, the game overview page, game time stats page and team managed page. I then edited the screenshots on GIMP to put them in front of a background with text above explaining the feature. I played around with positioning, color and text size for a bit before landing on screenshots I was happy with.



Exercise in checking boxes part of the app submission

Alongside screenshots videos could be provided for the App Store listing. I opted to not add videos as I would have to take them on the emulator. Therefore, the videos would be laggy which is not representative of the app. More minor tasks I carried out was filling out various forms so the publishers could determine the age rating of the app, the permissions used, and whether data was stored correctly. Next, I had to decide whether I wanted to charge for the app. Development costs up to this point were $275 ($150 Apple developer account, $100 Winnie Bagoes voucher that I gave to testers and a $25 Google Play Console account). As this cost was quite high I considered charging 50 cents for the app and using the money from sales to offset development costs. However, I decided against this as I believed 50 cents – although small – was a large enough barrier of entry to prevent users from trying out the app and therefore missing out on the social benefit it provided. Additionally, it seemed in bad faith to charge stakeholders and testers to continue to using the app, considering their generosity. After this I was able to submit the app to both the Apple Appstore and Google Play Store. After a couple day wait *SUBlime Substitution Manager* was released onto the Apple App Store and a week later it was released onto the Google Play Store.



SUBlimes app store listing

[32] https://www.privacy.org.nz/tools/privacy-statement-generator/

Conclusion

Next steps

Although I am extremely content with the final outcome of this project, I did identify some next steps to complete into the future.

- Create localizations of the app to other major languages such as French, Spanish and Chinese. To do this I will make use of localization tools. This would allow the app to become more accessible as the language barrier no longer exists.
- Add support for new sports. Currently planned is Futsal, Water Polo, Handball and Ultimate Frisbee. This would require consultation with an experienced player or coach from this sport, then for me to hard code in these new options.
- Further improving the tutorial system by adding a video tutorial system to guide users.
- Implement changes based on feedback received from the Mainland Football testing. This ranged from suggestions for a setting page, to the ability to customize the colour for each player on the subsheet.

Final Remarks

The development and success of *SUBlime Substitution Manager* has far exceeded any of my expectations. Over a year I have gone from no experience in mobile app development – and very little with JavaScript - to developing a cross platform app with React Native and Expo that has been published on both the Apple App Store and Google Play Store globally. Currently on the date of publishing this report, the app sits at 167 downloads on the App Store 7 days after release (No download statistics exist for Google Play Store as the app only released on it the day of publishing this report). The initial popularity of *SUBlime Substitution Manager* has earnt it the place of the third app to come up when users search 'Sports Substitution' on the App Store.

Whilst developing *SUBlime Substitution Manager* I worked alongside a wide base of sports' stakeholders within my school: A K, J C, B E, M J, M W and J K, as well as major sporting bodies in the Canterbury region such as Christchurch Netball, Mainland Football and Canterbury Basketball. I also worked alongside technologically minded stakeholders such as B L, N P and P A – however my biggest regret of this project was not finding a technological stakeholder who had prior experience with React Native. This was because at time I had questions specific to React Native and I was unable to ask anyone them.

The functionality of *SUBlime Substitution Manager* has far exceeded that of the minimum viable product. Users can create their team, add players, give them positions, create custom game formations, create subsheets, have a dynamic overview of ongoing games and view the allocation of gametime across a game and season. By providing this functionality I have addressed the problem I laid out at the start of this report. *SUBlime Substitution Manager* actively encourages equal gametime to make young athletes feel valued and part of a team. It will break down the negative association that some players have between playing sport and sitting on the bench. It will play a key role in diffusing parent – coach tensions by providing a way for coaches to transparently display the gametime of each player. Beyond this original goal, *SUBlime Substitution Manager* will help develop the self-management skills of young sports players as they can now actively manage their own substitutions. I hope as a result of this, *SUBlime Substitution Manager* will play a large part in increasing junior retention in sport, allowing for better physical / mental health and social outcomes for youth globally.

Photo sent in by testers of *SUBlime* in use