MERLIN MINDER

How to Develop a **Fitness App**

From Empty Lines to a Running Application

KBW, Supervised by Kaspar Jost

2022.12

let Abstract =

This Matura Project combines a practical project and the correlating documentation thereof. As illustrated in the title, the project consists of a fitness app I programmed and designed from scratch. The resulting code and designs are open-sourced and available on GitHub and Figma, respectively. The documentation includes blog posts displaying the process from planning to programming, debugging, and testing. In conjunction with other parts of the Matura Work, such as this report, I published it on a self-made website, which I developed to support this project.

const Table_of_Contents = [

];

```
const Preface = () => {
```

It is essential to know what to do in the gym. One way to make this easier is by using an app that shows your plan on the phone. However, every application has its weaknesses. It either does not suit your needs, does not represent your training style, or includes a paid version that unlocks the essential features. A way around this predicament is to create one yourself.

I have been learning to code ever since the first lockdown in 2020. As I was not following any courses or lectures, I never got into a project that lasted more than two weeks. This also led me to switch between multiple interests and explore new languages. Toward the end of 2021, I encountered mobile app development as part of my interest in JavaScript frameworks, which sparked an idea.

I could combine the need for a proper workout app with launching my first more extensive project. This was also around the time of gathering ideas for the Matura Project, which would begin in spring 2022. On top of all that, I debated pursuing programming as a career. It would be beneficial to build a portfolio of projects such as the one I had in mind for future job applications. With all these thoughts and motives pointing in the same direction, this incentivized me to commit to starting this project as my Matura work.

I also wanted to be able to document my development process on an additional website so that I could further expand my portfolio. The Matura project also required procedure documentation, so the supplement was well-aligned.

The website should be viewed on a computer for proper display and can be found under this link:

https://matura-website.web.app

```
};
```

// Acknowledgments

Since this project is software, at its core, I must mention the open-source community. As most people know, the internet is built on free, open-to-use software, and our technological advancement is closely coupled with open-sourced software. The same applies to my project, as I used multiple publicly accessible code libraries. I am very grateful for all the work that people and developers before me chose to release free of charge. That's why I'm open-sourcing my project for anyone to see and take inspiration from. The main programs I used for my project were: VSCode, Expo, React Native, Android Studio, Figma, GitHub, Firebase, Vuejs, and TinyMCE.

One of the main inspirations for the project was William Candillon, and I must give special thanks to him. Thanks to his YouTube channel, I got into using React Native in the first place. On top of that, he developed a graphics library called react-native-skia right before I started my project. Therefore, I was lucky enough to have access to it, and all the graphics in my app are attributed to the existence of his library.

The links to his works are here:

GitHub - https://github.com/wcandillon

YouTube - https://www.youtube.com/@wcandillon

Of course, I would also like to thank Kaspar Jost for supporting me. In my opinion, it was valuable to have somebody to help you with the general structure of the project, such as keeping track of time so you could focus on the work itself.

Also, huge thanks go out to my friends (Eric, Lea, Gian, and Noelle) and family (Helen, Peter, and Leila), who were by my side while working on this project. Be it just working together, answering questions, or even reading a paragraph to check for errors. They gave me some much-needed support.

Finally, thanks to the help of word-checking programs, I didn't have to pay anybody to read through or correct my paper. Writing AIs has become more than just a gimmick in this modern age. If you know how to use them, they can immensely reduce your time on rephrasing or checking grammar. The programs I used were: Grammarly, Ludwig, Wordtune, msword.blogspot (cover).

if (!Introduction) return;

Leading Question

What does the process of programming a fitness app look like? I chose to work out this question to allow for more freedom. This leading question gave me the certainty that even if the app failed, I would still have answered the question. It doesn't require a finished product but focuses on the process and collection of skills along the way. When dissecting the sentence, the process describes the documentation, and the programming represents the product.

As mentioned before, even with an unfinished or mediocre app, the project would have still been complete thanks to the posts I wrote about the process. To help with the writing and posting of real-time updates, I wrote a separate website. Ultimately, my project is technically split 50-50 between product and process. Therefore, the website should not be disregarded.

Goal of Project

I planned to develop a fitness app and document the process. The main goal was to answer the leading question to complete my task. However, other motivations also kept me working on the project. One of them was to use the app while working out. This encouraged me to stick to the vision I created initially and create something I would use personally. Another one was to show the project to future job employers. With that in mind, it kept me engaged and willing to work almost daily for at least an hour. Finally, if I were to mention one last motivation, it would be my drive to improve my programming skills. The best way to accomplish this is to learn and improve by completing projects and documenting the process for self-reflection.

Structure of Report

The report is structured to separate the process and the product. In the Main part, I will explain my plan first, and then I will walk you through it. You are going to see all the different steps someone takes when creating any mobile app with react native. Testing the app will finalize this section as it is slightly different from what someone typically does.

The code that resulted from the app is too large to explain in detail. Therefore, I decided to show four examples covering most of my difficulties. Finally, and most importantly, the reflection concludes the report. I explain what I learned and what I could have improved. Reflecting on such a large and complex project can highlight essential parts of what went wrong and what went well. From that, you can deliberately look for crucial mistakes or promising practices to help you with your next project.

Main.map((step) => step.description);

Procedure

Having a plan is not only important in the gym. When you start a new project, taking a bit of time to develop a procedure is going to save you more time in the long term. The main parts contributing to my decision-making and planning for this project consisted of four things. First, I wanted my app to look appealing, but I was not experienced with the react-native framework. Second, I did not know what features to include, but I wanted to start writing posts as soon as possible. This led me to create the procedure you will go through in this section. I will start by mentioning the website, which enabled me to write posts immediately. Then while designing a clean UI in Figma, I debated features for the app. Next, I explain Expo and React Native for future context. After that, I implemented the UI to learn the framework before finally adding the functional features, which are more complex. All of this can be looked up in more detail on the website, as this report is just a summary of the process that took over 130 hours.

Website

After the initial setup of the website with Vuejs, I was able to write posts in the TinyMCE editor and upload them to the Firebase backend. I have not taken any screenshots from the development of the website as this wasn't relevant to the project. However, the changes made after the setup process were counted toward the work time as the final look of the website impacted the posts and, therefore, the process. While gathering ideas for the design, I continued implementing the styling and added additional features. An important one was the timeline which gave me a guideline as to when the various steps should be completed. The amount of time and specific due dates for the different sections can be seen on the image to the right, which was copied from the website. Additionally, I introduced other essential parts to the website, such as the leading question or a tailored version of the abstract.

Figure 1: Timeline of the project

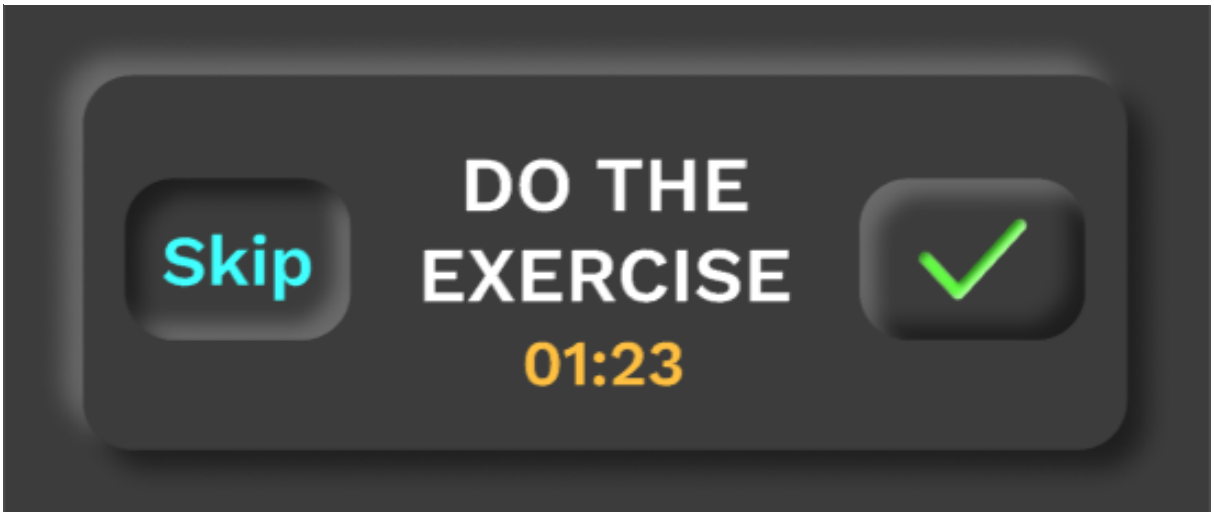
Source: Own Graphic

Design

Continuing, I started considering how I would design my app. The platform I used to assist me with creating drafts is called Figma. It allows you to develop prototypes and full-fledged designs for websites or mobile apps quickly and easily.

Starting off, I decided on a color palette to stick to when coloring the app. On one side, it consists of multiple levels of gray to build contrasts and shadows. On the other side, I chose a gradient from light blue to orange, which is used for titles or special buttons. Meanwhile, I decided on a general style my app would convey. Then, playing into the color selection, I tried designing my whole app with the neomorphic style. Neomorphism uses a highlight and a shadow on opposing sides of an object to portray a 3D look and distinguish it from the background as can be seen in the graphic below. Further explanations of neomorphism and all my design decisions can be found in the posts on the website.

Figure 2: Neomorphism App Example

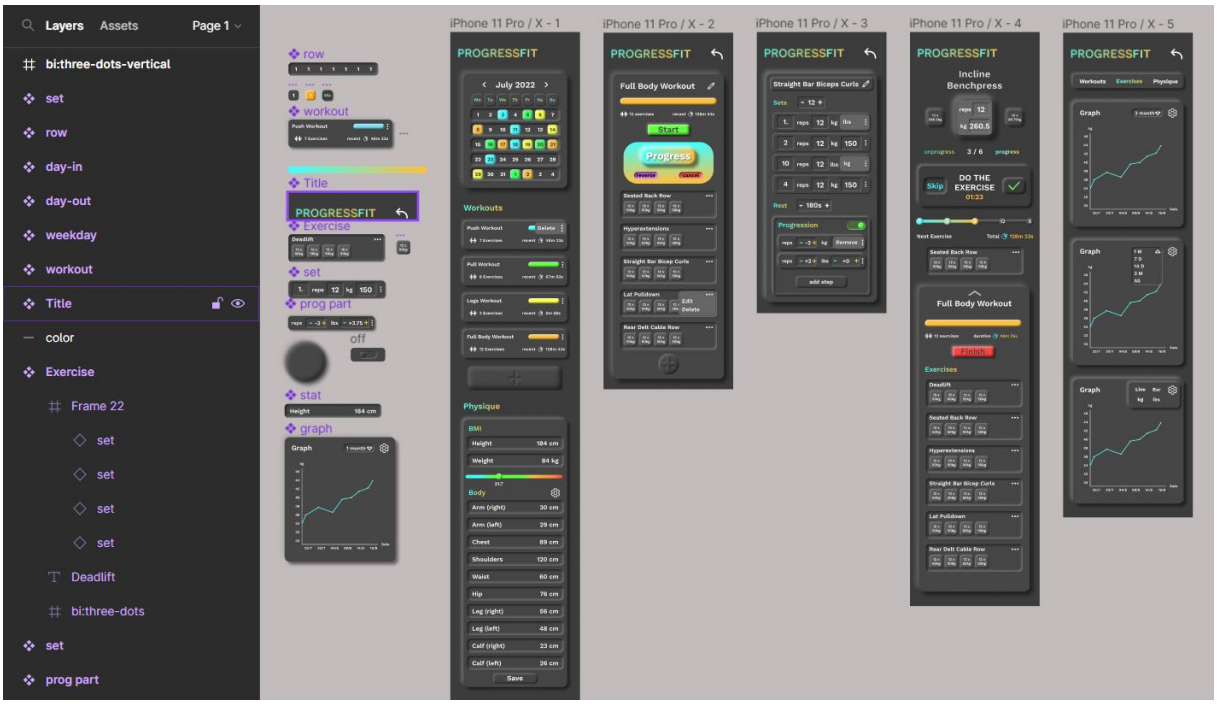


Source: Own Graphic

Over the 16h I spent designing, I kept an eye out for any future problems that could come up. Be it variable text- or number length for which I had to account for, or a popup that would be used later. Additionally, I chose to come up with all the features while designing as I could implement them swiftly into the final product at this time without any significant hassle.

The main challenges in the design process were to push creativity and think ahead. I needed to come up with a new idea for the next feature, and, on top of that, I was already worrying about the implementation and connection thereof into the app. However, after overcoming creative blockades, I made quick progress and ended up being content with the design, which can be seen below, before the schedule. This gave me a buffer which I was able to use up further into the project.

Figure 3: Full Figma Design



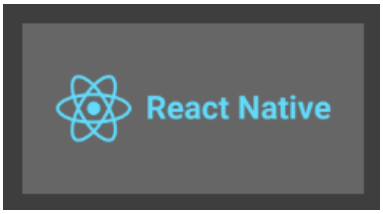
Source: Own Graphic

Expo, React Native

This subchapter focuses on the main building blocks that allowed me to create a mobile application. It should also develop a better understanding of the following three chapters. It deserves to be noted that this isn't an in-depth explanation of the two tools, as you can find well-written documentation and descriptions on their websites.

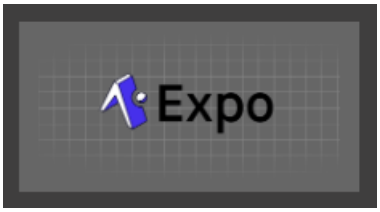
Mobile apps have a specific difficulty to them. You need to write two versions of your app for the two main operating systems (OS) present, IOS, and Android. However, the developers of React Native (RN) solved this problem by allowing you to write your app in JavaScript, which then gets transformed into the specific native versions used by the two OS. However, there still exists another problem.

Figure 4: React Native Logo



Source: Own Graphic

Figure 5: Expo Logo



Source: Own Graphic

When developing and testing, one would have to transform his RN code every time he changed a single line to see the changes. This would take up ages, and that's why there is another tool to solve this challenge called Expo. It manages your workflow, so you don't have to. With Expo, your changes get updated in real-time, and you can see and test them immediately.

Thanks to these two tools, I didn't have to learn a new programming language, and the time it took me to start writing working code was cut down immensely. Finally, thanks to Expo, I can publish my app without paying the app- or play store their developer fee. The way to access the app is described in detail on the website.

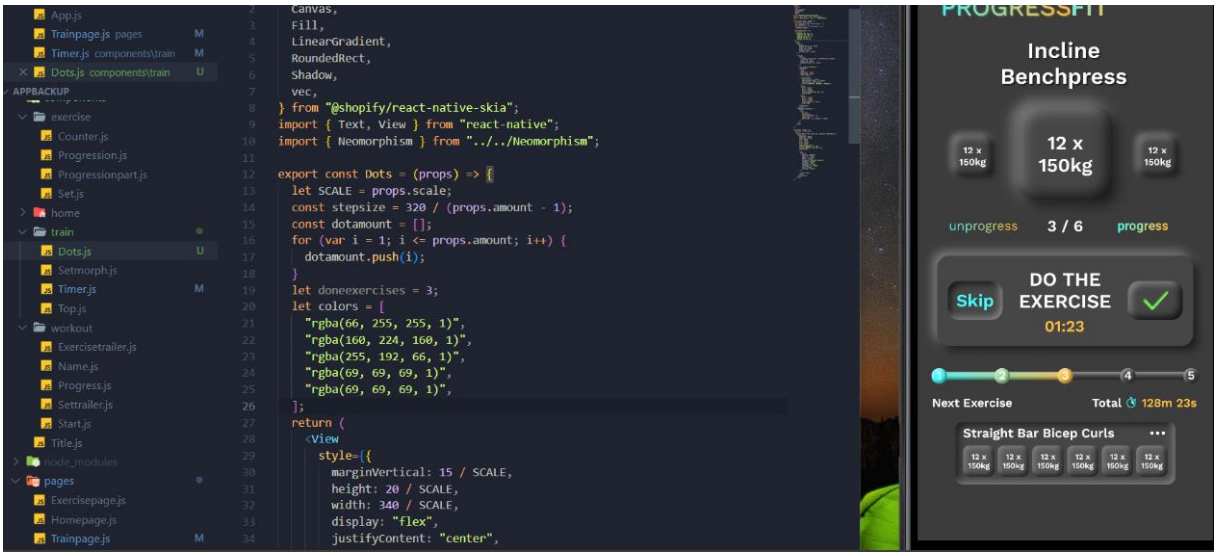
UI

The user interface (UI) describes the part the user can see when using the app. This includes the text, fonts, colors, boxes, buttons, styles, etc. My idea was to copy the design I created in Figma and set it as my UI. Usually, this would be a relatively straightforward process when developing a website. With RN, this was not the case. It uses a similar approach to styling as websites do but with some slight differences. The largest of which was shadowing.

Both Android and IOS only use a limited way of displaying shadows. The limiting was so severe that I could not create the neomorphism effect. Without it, there would be no distinction between the components. Since my whole design was based on having complete control over the shadowing, it was indispensable that I had to find a workaround. This is where code libraries come into play and why I wanted to mention William Candillon specifically. Thanks to his library called react-native-skia, I could control and display the shadows how I needed to.

But it wasn't smooth sailing from then on. The library itself brought out some challenges, especially while implementing animations in the UX part. Moreover, it introduced new components that didn't seamlessly work with the given ones by RN. That's why I decided to write my own component that allowed me to create a neomorphic component just as I would implement one from RN.

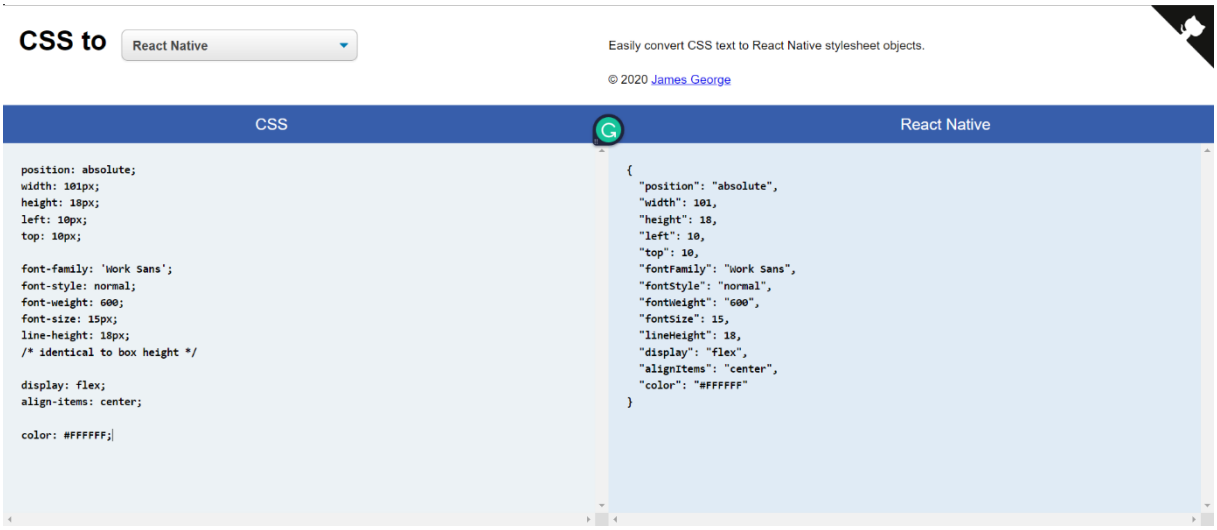
Figure 6: UI Programming Example



Source: Own Graphic

Another difference between the website and RN styling is the syntax. You can copy the CSS styling in Figma and paste it into your website's code. With RN's different styling syntax, this was also not possible right away. First, I was debating whether to create some sort of converter software myself since the difference came down to only a couple of characters. Luckily after some searching, I found a website, as seen in the figure below, that converts Cascading Style Sheet (CSS) styling to the one used in RN. If someone is interested in using it themselves, this would be the link: https://csstox.surge.sh/

Figure 7: CSS Converter Website



Source: Own Graphic

One last difficulty when creating the UI didn't come from the incompatibility with Figma. It was actually similar to the one described in the Design section earlier. I was already looking out for the UX phase and tried to form structures that divided the UI into manageable parts. I knew what features belonged to the different components and tried to simplify the code as much as possible. In retrospect, I did a decent job and only had to change or divide a minor number of files when working on the UX. Further reflections I had will be stated in the Reflection section at the end of this report.

UX

The user experience (UX) incorporates everything which isn't static. This includes the reaction to a button press, swipe, tap, text input, or similar. It gives life to the UI and can change the information displayed. A good UX works smoothly and remains basically unnoticed by the user. I also involved the data handling and saving into the UX as, in my case, the code for the data works hand in hand with the rest of the UX.

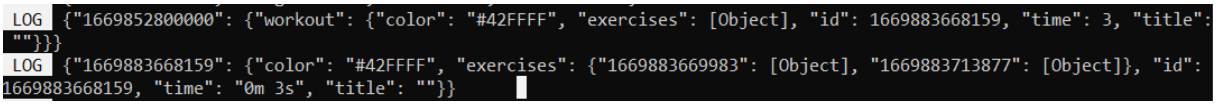
I approached this project in a way that I wanted to work myself up in difficulty. It would keep me engaged with a new problem that was harder than the one I had just solved. This meant that when

applying this logic, I thought the UX was the most complex part of the project. Not only was this the case with the difficulty level, but also with the time it took. As mentioned in the Design section, I had created a time buffer, and the UX was where I used it.

The extent of difficulties and problems I faced in this section can only be described by the number of posts I wrote describing a single function that took me over an hour to develop. There were three main problems I had to overcome. The one that was the most complex must be working with the data. The one that led me to give up on solving was the animation of the neomorphism components from the UI. Lastly, the one that gave me the most errors were the combination of different libraries and functions.

In the app, the user can create and delete workouts, exercises, sets, and more. I wanted that data to be saved even when the app was closed. This led me to choose AsyncStorage as my database. As the name implies, the storage works asynchronously, forcing me to add multiple workarounds to fetch and save data. The worse part was that I wanted to save every change directly, so I didn't have to add save buttons everywhere. This became problematic when I just wanted to add or remove an item. Because of the workarounds mentioned above, this wasn't as straightforward as it sounds and led to large and complex data structures.

Figure 8: Data Example



Source: Own Graphic

As mentioned in the UI section, I created the neomorphism component myself for ease of use. However, when trying to animate it, such as enlarging or shrinking the component, this wouldn't work. And even after dismantling the code and writing the parts from scratch, I couldn't animate it properly. As I was already far into the project at the time, this forced me to give up trying to find a solution. Unfortunately, the complexity of the whole app was too immense to change to account for this animation problem. This is why some parts of the app don't slide or resize properly, but luckily this doesn't affect the function or throw errors.

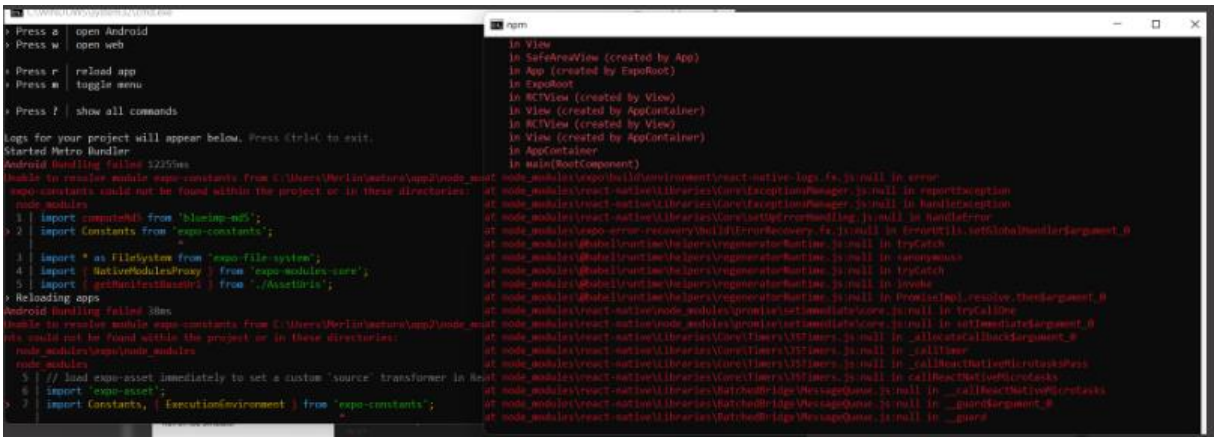
What did throw errors, however, was my combining of libraries and functions thereof. There is a workflow that combines two libraries to create smoother animations. I used this but additionally added a couple more libraries for other use cases. Each has functions to move and manipulate data; some work together, while others throw errors. The problem could have been fixed if I had stuck to one library to handle the data. The same applies to the other two difficulties, and a closer inspection of how I could have done it better can be found in the Reflection at the end.

Testing

There is a testing phase marked in the timeline. However, I have been testing the app through all the steps mentioned above. Unfortunately, just as the difficulty rose as the app progressed through its stages, the testing became more tedious as well. While designing, the testing consisted of looking at it and deciding whether I liked it. With the UI, it was similar. If the correct thing was displayed, all was fine. However, if something wasn't working out, I had to tweak the stylings until the object was the way I wanted.

The UX imposed way more challenges again. I could only see the data if I logged it to the console, and the only way to do that was to use the app to create the data. All the testing had to be done manually, and if any error occurred, such as shown in the example below, I had to restart my Expo session to reset the app and the stored data. I probably could have written test cases; however, due to the complex data structure I described before, the test cases would have been so complex that the amount of time spent wouldn't have been worth it.

Figure 9: Errors



Source: Own Graphic

The tests still had to be done to check the functionality and reliability of the code I wrote. Testing often gets overlooked in a development process. However, much time is spent on it, and the importance dictates how well the app comes out.

const Code_Example = useSharedValue("code");

This next section is dedicated to the code I wrote. Since the whole codebase is over 4500 lines long, I tried to pick out some examples showing the app's main parts. Of course, this is not indicative of all the code I wrote but only for example purposes. The subsections will first show the code, and I will explain some key features of it as generally as possible afterward. Just as with the other sections, a deeper look into the app's inner workings can be found in the posts on the website.

Component Tree

```
return (
  <View>
    {Object.values(data).map((workout) => {
      if (workout.id) {
        return (
          <WorkoutTrailer
            key={workout.id}
            workout={workout}
            nav={props.nav}
            setData={setData}
          />
        );
      }
    })}
    <GestureDetector gesture={gesture}>
      <View style={styles.workoutspluscontainer}>
        <Neomorphism
          center
          settings={{
            ...neostyles.workouts,
            ...{ colorS1: colorS1, colorS2: colorS2 },
          }}
          inset
        >
          <Image style={styles.workoutsplus} source={plus} />
        </Neomorphism>
      </View>
    </GestureDetector>
  </View>
);
```

The component trees dictate how the different components of the section relate to one another. The way this is done in React Native is similar to the HyperText Markup Language (HTML) in websites. There is an opening and closing tag for each component; inside of that, you can add more components in the same way. A good analogy can be to imagine a box where other smaller boxes inside of it divide the space between them. The blue words, such as `<View>`, describe the component type. The yellow keywords, such as `style={}`, further define specific properties of the component, as in this instance, the styling. In the case of the image component, the source property defines what image to display.

Not all the components display a box, text, or image. `<GestureDetector>`, as the name implies, can detect a gesture. In this example, it detects if someone pressed it. It then runs the function given in its gesture property. If you need to display data that changes over time, you can add it into the component tree in curly braces. This tells the program to turn the given data into components on runtime. This applies to the third line of the tree, where the object returns multiple `<WorkoutTrailer>` components depending on the provided data.

Data Converting

```
const doneexlist = doneex.map((ex) => {
      return {
        set: exercises.value[ex.ex].sets[ex.set - 1],
        exercise: exercises.value[ex.ex],
      };
    });
    const doneexercises = {};
    for (let [index, element] of doneexlist.entries()) {
      if (doneexercises[element.exercise.id]) {
        doneexercises[element.exercise.id].sets.push(element.set);
      } else {
        doneexercises[element.exercise.id] = {
          name: element.exercise.name,
          id: element.exercise.id,
          progression: element.exercise.progression,
          rest: element.exercise.rest,
          on: element.exercise.on,
          sets: [element.set],
        };
      }
    }
```

Converting data to save or use it was an essential part of my code, as mentioned in the UX subsection. In this example, I have a list of numbers corresponding to indexes in another list called exercises. I need to save a new exercise object containing the data from the exercises list. For that, I first created a separate list called doneexlist with the correct data from the items included in the exercises list. Therefore, I must go through the first list one number at a time, get the item from the exercises list with the specific index, and add it to the doneexlist. This step is done on lines 1-6. With the correct data now saved in the variable doneexlist, I must go through that list as well and add the data into the new exercise object that itself gets saved in the doneexercises. The structure of the object can be seen in lines 13-18. The whole object creation and adding to the doneexercises is showcased on lines 7-21.

Data Saving

```
const today = new Date();
today.setUTCHours(0, 0, 0, 0);
const sendobj = new Object();
sendobj[today.getTime()] = {
  workout: {
    id: id,
    title: data[id].title,
    color: data[id].color,
    time: Math.floor(totaltime.seconds),
    exercises: doneexercises,
  },
};
Merge("calendar", sendobj);
```

This part shows how I saved the data from the example above to the calendar so the user can revisit the workout at another time. The problem faced here was that I wanted to add the new exercises to a list of already existing ones. Due to the structure and inner workings of the AsyncStorage, you cannot just add data to existing data without overwriting the existing one. However, you can merge different data with different existing data. That is the reason for this example. I need to create an object that differs from the already saved ones.

The way I did this was to add the date when the workout was saved, as that is different for each day. So, in line one, I get the exact time; in line two, I change the time to equal the first second of the day. After that, a new object called sendobject is created, and in line four, I add the date to the object. Additionally, the doneexercises from before are added to the object with some other information. Finally, this new workout and date object gets Merged with the calendar data, which saves it in local storage.

Drawing / Skia

```
<View style={{ display: "flex", alignItems: "center" }}>
          <Canvas style={styles.progresscanvas}>
            <Fill color="transparent" />
            <RoundedRect
              x={0}
              y={0}
              width={110 / SCALE}
              height={30 / SCALE}
              r={15 / SCALE}
              color="red"
            >
              <LinearGradient
                start={vec(55 / SCALE, 0)}
                end={vec(55 / SCALE, 30 / SCALE)}
                colors={["#33FFFF", "#FFCC33"]}
              />
              <Shadow
                dx={4 / SCALE}
                dy={4 / SCALE}
                blur={2 / SCALE}
                color="rgba(230, 255, 255, 0.8)"
                inner={true}
              />
              <Shadow
                dx={-3 / SCALE}
                dy={-3 / SCALE}
                blur={2 / SCALE}
                color="rgba(51, 34, 0, 0.6)"
                inner={true}
              />
            </RoundedRect>
          </Canvas>
          <Text style={styles.progresstext}>PROGRESS</Text>
          <Text style={[styles.progresstext, styles.progresstext2]}>
            PROGRESS
          </Text>
        </View>
```

This last code example shows one of the times when I had to create a custom component with the react-native-skia library. The reason was the addition of the `<LinearGradient>` component in between. This part of the code results in the PROGRESS button, which you can see in the app, website, or Figma design. As the subchapter implies, skia is a graphics / drawing library. Therefore, if you want to use any skia components, they must be contained in a `<Canvas>` component. This lays the foundation for any other components to be "drawn" inside.

Continuing, you can `<Fill>` the canvas with a background color. In my case, I made the background transparent to give the impression of the button being "real" and not "drawn". Next in line comes the box or Rect(angle), which in this example also has Rounded corners. As explained in the first code example, its keywords describe further properties like `width={}` or `height={}`. Inside the `<RoundedRect>`, there are other additions that were not able to be added as keywords. These, in this case, are the shadows and the unique linear gradient. Just as with every other component, further properties such as `color=` are described by the keywords. Finally, as mentioned in the UI section of the Main chapter, the RN components don't work well with the skia ones. That's why the text to be displayed inside the button isn't inside the RoundedRect, which describes the button. With the `style={}` property, I had to manually move its position to be above the button to give the illusion of a "real" button with text.

return { <Reflection /> };

Especially when considering that one of my motivations for this project was to improve my programming skills and reflect on the process, this section is critical. I'm happy to announce that I learned a lot and found a few parts where I could have improved. Another thing to mention is that thanks to this project, I want to try making a second version of my app where I apply all the things mentioned in the next two subchapters.

Process

One thing that stood out when reflecting on the process is how much I had to look out for future complications with the next steps. For example, when designing and implementing the UI, I had to invest more time to account for the UI and UX development. The reason I chose this approach was that I didn't have an idea of what my app should do or what features it should have. Therefore, as explained in the Design chapter, I wanted to work out all that while designing.

A better way I could have done this would be to write down the features, colors, designs, and data structures separately in a brainstorming phase before starting the project. Afterward, I should have implemented the features and data structures without any designs to test the functionalities and expose any incompatibilities or problems with the features. After that, I could design the different components and additionally look out for functional designs. For example, make a quit button red or a start button green so it becomes more apparent to the user what a component's function displays. Finally, I would set the design to the UI and wouldn't have to account for any complications with scaling or moving parts as that has been done beforehand.

As explained before, I had my reasons for doing it the way I did. But when looking back, I believe that even if it is your first time developing an app, you should instead do it the way described here than how I did it.

Coding Practices

This section focuses on how I wrote and documented my code and how to develop a more robust data structure. As mentioned multiple times, loading and saving the app data had limitations and produced many complex solutions. Up to the time of writing, there hasn't been a better storage option than AsyncStorage. However, I found a way better way to handle the loading and saving. First, you should only load and save the data once while using the app. This is when you open and close the app, respectively. There are multiple ways to do this, therefore, I won't explain them in detail here.

Next, you should have all your data in one place where you can access everything from anywhere in your app. You can create this yourself or use a state management tool like Redux. This will simplify the way you handle data immensely. It also allows you to work with your favorite datatypes as you can access everything from anywhere and only need to save it at the end, which no longer imposes a problem.

Another important takeaway is to stick to one library to manage the state of your data, as I already concluded in the UX section. Combined with that is the tip to use the same gesture handlers and animation libraries for all your more complex UX functions. This might be clear to experienced developers, but if you're learning about everything while programming, this can get lost quickly.

Next in line is the importance of testing. Everyone tests their code in some capacity, but when working on a project the size of this one, it becomes an essential part of your process. If you apply all the tips already mentioned, writing tests for your code won't be too difficult. Testing allows you to check multiple edge cases at once without using the app itself, which would take up much time. Also, it might be helpful to test the functionalities of the libraries you're debating on using in a separate smaller project before implementing them into your large complex one.

Finally, this part should be clear to any developer at any level, but even experienced ones sometimes don't do it. You should document your code right after writing it. When you explain the inner workings of the function, you just wrote in a couple of short comments; this can help you in a couple of ways. You might find a bug or exception you didn't account for before. Also, when you revisit your code in the future, it might not be clear what the code does if it wasn't documented. A few quick explanations will save you much time on testing and finding out what the function does.

Closing Sentences

One takeaway I would advise anyone to use for their next project is to document the process in any way. It can be checking off every day you worked on the project or writing a short post as I did. It creates discipline and motivates you to work more, even if it is only to keep your streak going. Also, you can look back on all the progress, which gives you further motivation not to quit when you reach a low point. Finally, for any developer who, like me, hasn't committed to a larger project, I can strongly advocate doing so, even if it is just for the learning experience.

import { List } from Illustrations;

let Links;

My Website : https://matura-website.web.app/

VSCode : https://code.visualstudio.com/

Expo : https://docs.expo.dev/

React Native : https://reactnative.dev/

Android Studio : https://developer.android.com/studio

My Figma : https://www.figma.com/file/5ppSR97yM4pK5sLy3rH1S8/

My GitHub : https://github.com/MerlinMinder

Firebase : https://firebase.google.com/

Vuejs : https://vuejs.org/

TinyMCE : https://www.tiny.cloud/

Williams GitHub : https://github.com/wcandillon

Williams YouTube : https://www.youtube.com/@wcandillon

Grammarly : https://app.grammarly.com/

Ludwig : https://ludwig.guru/

Wordtune : https://www.wordtune.com/

MSWord.Blogspot : https://msword-free.blogspot.com/