# UI Builder –
# an Interface Building Tool for Generating React Native Code for Mobile & Web

Edvin Havic

DEPARTMENT OF DESIGN SCIENCES
FACULTY OF ENGINEERING LTH | LUND UNIVERSITY
2019

MASTER THESIS

jayway.
by devoteam

# UI Builder – an Interface Building Tool for Generating React Native Code for Mobile & Web

Edvin Havic

**June 11, 2019**


LUND
UNIVERSITY

# UI Builder – an Interface Building Tool for Generating React Native Code for Mobile & Web

Copyright © 2019 Edvin Havic

*Published by*
Department of Design Sciences
Faculty of Engineering LTH, Lund University
P.O. Box 118, SE-221 00 Lund, Sweden

*Publicerad av*
Institutionen för designvetenskaper
Lunds Tekniska Högskola, Lunds universitet
Box 118, 221 00 Lund

Subject: Interaction Design (MAMM01)
Supervisor: Joakim Eriksson
Co-supervisor: Hans Löfgren (Jayway)
Examiner: Kirsten Rassmus-Gröhn

Ämne: Interaktionsdesign (MAMM01)
Huvudhandledare: Joakim Eriksson
Bitr. handledare: Hans Löfgren (Jayway)
Examinator: Kirsten Rassmus-Gröhn

**Abstract**

This thesis investigates how a visual design tool for building cross-platform application interfaces for iOS, Android, and web can be constructed. While there are plenty of tools for building and designing mobile and web apps, there are currently no visual design tools for building cross-platform apps for all three platforms.

Beyond this, the thesis explores potential solutions for problems related to code generation, mobile and web app development, as well as accessibility support across multiple platforms – all while focusing on the user experience of the visual design tool. To find solutions to the problems within these areas, a visual design tool was developed with support for several key features: high quality code generation, an intuitive way to visually build well designed and accessible user interfaces, and a unified codebase for the three target platforms.

The work process was highly iterative and user-centered, and was split into multiple phases and utilizing several usability testing methods, such as heuristic evaluations and user testing.

The results found during this work shows how a visual design tool for cross-platform development can look and function, as well as giving potential solutions to how high quality code generation and accessibility support can be implemented, within in the context of cross-platform applications.

**Keywords:** interface design tool, user-centered design, accessibility, react native, cross-platform development

**Sammanfattning**

Detta examensarbete undersöker hur ett visuellt designverktyg för byggande av användargränssnitt till iOS, Android, och webben kan konstrueras. Även fast det finns en hel del verktyg för att designa och bygga mobil- och webbappar, så finns det i nuläget inga visuella designverktyg för att bygga en gemensam app för alla tre plattformar.

Utöver detta, så undersöker arbetet potentiella lösningar till problem relaterade till kodgenerering, mobil- och webbapputveckling samt tillgänglighetsstöd (eng. *accessibility support*) på flera plattformar – samtidigt som fokus ligger på användarupplevelsen av designverktyget. För att hitta lösningar till problemen inom de angivna områdena har ett visuellt designverktyg utvecklats med stöd för flera viktiga funktioner – högkvalitativ kodgenerering, ett intuitivt sätt att visuellt bygga väl utformade och tillgängliga (eng. *accessible*) användargränssnitt, samt en gemensam kodbas för de tre målplattformarna.

En iterativ arbetsprocess har använts, och fokus har genomgående legat på användaren av systemet. Processen delades upp i flera faser, och flera typer av användbarhetsutvärderingar har använts – bl.a. heuristiska utvärderingar och användartestning.

Resultaten som presenterats i detta arbete visar hur ett visuellt designverktyg för flera plattformar kan se ut och fungera. Utöver detta så har även potentiella lösningar till hur högkvalitativ kodgenerering och tillgänglighetsstöd kan implementeras presenterats.

**Nyckelord:** designverktyg för användargränssnitt, användarcentrerad design, tillgänglighet, react native, utveckling för flera plattformar

# Preface

The work presented in this Master's thesis was conducted at Jayway Malmö, by me, Edvin Havic, during the spring of 2019.

I would like to thank the people at Jayway, as well as the people that have provided guidance, materials, and feedback throughout this process, more specifically:

My supervisor at Jayway, Hans Löfgren, for giving invaluable feedback, ideas, insights, and rewarding discussions.

Jayway's Quantum Studio Lead, Martin Gunnarsson, for providing me with the necessary hardware, work space, and opportunity to do my work at Jayway.

My supervisor at LTH, Joakim Eriksson, for providing great guidance, support, and expertise for both the thesis and research related questions.

# Contents

# Glossary

| | |
|---|---|
| API | Application Programming Interface. |
| Backend | Backend refers to the data access layer in an application. For example, the code generator in UI Builder is a part of the backend. |
| Boilerplate | Boilerplate is a skeleton/structure with code to get a minimal base to start from. |
| CSS | Cascading Style Sheet. |
| Document Object Model | Document Object Model is a programming API for HTML documents. It defines the logical tree structure of a HTML document using tags. |
| DOM | Document Object Model. |
| Frontend | Frontend refers to the presentational layer in an application, i.e. the things the user interacts with. |
| npm | Node Package Manager. npm is two things - a CLI tool and an online repository. npm is used to fetch npm packages from the npm repository for use in e.g. a project. |
| npm package | An npm package is a bundled collection of resources (i.e. code) readily available to install from npm. |
| OS | Operating System. |
| React Prop | A React prop is a property set on a specific React component to change e.g. a behaviour or look for that specific component. |
| UI | User Interface. |
| UI Builder | The name of the developed visual design tool. |
| UX | User Experience. |

# 1 Introduction

Generating code with visual design tools is nothing new – especially when it comes to web development. Multiple targets for a programming language isn't a new concept either, take Java for example – a programming language popularized because of its cross-platform support. Within the last five years, technologies such as React and React Native have transformed both how web and mobile applications are developed. With these two technologies further blurring the line between mobile and web development, combining multiple codebases into one would seem like the next logical step, pushing the slogan *"write once – run everywhere"*[1] even further.

Building cross-platform, accessible, and well designed apps today is a tedious process – multiple design tools, programming languages, APIs, and best practices for each platform creates challenges for both developers and designers. There are of course already great visual design tools for specific tasks – take Android Studio's design tool for example, used specifically for Android app User Interfaces (UIs). While Android Studio's design tool is great for Android apps, different tools are needed for other platforms. Building apps for iOS? Use Xcode's Interface Builder. Building web apps? Pick your poison, there's many to choose from.

While all these tools are great at their specific task, maintaining three different codebases for iOS, Android, and web is – while many times necessary – tedious. Even creating simple app prototypes for each platform is time consuming, and depending on which tech stack you've selected, requires knowledge about several programming languages.

These problems raises some questions:

- Can you create a visual design tool for all three platforms?
- How do you handle platform differences?
- Can you decrease the time spent from idea to prototype? Perhaps even from idea to functional product?

The goal of this thesis is to explore potential answers to these questions.

## 1.1 Scope and Goal

The main goal was to create an interface design tool (called the UI Builder from here on out) able to generate high quality React/React Native code for both iOS, Android, and web using the same codebase. There was a focus on three main aspects, which were:

[1] A slogan popularized by Sun Microsystems to illustrate the features of Java [1].

- **Code Quality**
  For the generated code to be usable, it needs to be of high quality – there's no point in using a visual design tool to generate code if the code is of poor quality.
- **Handling accessibility in the different platforms**
  Different platforms have different approaches on how to handle accessibility. The visual design tool needs to be able to unify the differences in how accessibility is handled on each platform.
- **User Experience (UX) of the Interface Design Tool**
  Using a visual design tool needs to be as painless as possible, something that can be a go-to tool for creating, for example, prototypes.

There were some limitations made to decrease the size of the scope:

- **Web, Android, and iOS *only*.**
  There are ports of React Native made to work on for example Universal Windows Platform (UWP). This thesis was limited to the three mentioned platforms.
- **Components will only take mobile devices into account.**
  Responsive components should be a part of the UI Builder, it was however not implemented and will be considered as future work. Components should work on both desktop and mobile, but was designed with a mobile viewport in mind.
- **No functionality for implementing logic.** The UI Builder's focus was on accessibility and design, not logic implementation.

The work described in this thesis is split into 8 sections, each describing an integral part of the process:

- **Technical Background**
  The technical background section goes over all relevant technologies used.
- **Theory**
  This section describes the theories used to support the work.
- **Method**
  This section describes the methods employed throughout the process.
- **Investigation, Design, and Implementation Phases**
  These three sections describe the bulk of the work.
- **Discussion and Conclusion**
  These two sections is where the work done is analyzed and concluded.

# 2 Technical Background

The scope of this thesis spans across multiple types of technologies and platforms, making it heavily reliant on previous work within areas such as web & mobile accessibility, and iOS & Android development. This section describes the technologies related to the work done in this thesis.

## 2.1 React for Web

React is an open-source JavaScript (JS) library for building user interfaces for the web, released by Facebook in 2013. Early on, React was coupled with the Document Object Model (DOM), but was subsequently split into two separate packages, `react` and `react-dom`. This meant that the React core didn't depend on the browser anymore – only `react-dom` did. This made the foundation for React Native.

## 2.2 React for Mobile – React Native

React Native, much like React, is an open-source JavaScript framework for building user interfaces for iOS, and Android. It uses the same core library as React, but instead of manipulating a virtual DOM (using `react-dom`) it runs as a JavaScript background process using the device's JavaScript engine, and manipulates the UI via a bridge – a part of the `react-native` package. This is where React Native differs from traditional apps written in JS – React Native apps are actually *native* [2].

## 2.3 Expo for React Native

Expo is a popular tool built as an additional layer on top of React Native [3]. Using Expo has several benefits such as a more streamlined development process, using their app for iOS and Android. The app allows developers to test on both platforms without additional setup, such as having an Apple Developer Account. Expo also provides a Software Development Kit (SDK) for easily using the camera, GPS, accelerometer, etc.

Expo also comes with drawbacks – mainly bloated application sizes and limitations on which libraries can be used.

A compelling reason to use Expo instead of purely using React Native is to have a better Developer Experience – using it enables cross platform (development) support for both iOS and Android development, whereas using React Native without Expo

would disallow iOS development on Windows and Linux. Expo also provides a web interface for the Metro Bundler[2], further simplifying development.

## 2.4 Unifying Codebases Using React Native for Web

The library `react-native-web` makes it possible to use components and APIs written for React Native on the web, using `react-dom`. While the name might be confusing (isn't `react-native-web` just regular React?), the aim of the project is to create a platform agnostic UI framework – meaning that one component should be usable on both iOS, Android, and web [4]. `react-native-web` can be seen as a complex polyfill[3], emulating the React Native API using `react-dom` and CSS.

React Native made it possible for cross-platform development between iOS and Android, and using `react-native-web` the target platforms expand to include the web as well. This does of course come with its drawbacks – but the benefits outweigh them.

## 2.5 Accessibility Support – Platform Differences

Accessibility, in the context of software, is the design of a product for people with different kinds of disabilities, such as visual or motor impairment [5]. Each platform handles accessibility differently, with varying support. Both mobile platforms provide APIs for their assistive technologies, such as screen readers – VoiceOver on iOS [6] and TalkBack on Android [7]. For the web there are different types of screen readers, either available as a plugin in the browser or natively in the OS, such as ChromeVox [8] for Chrome and VoiceOver [9] for macOS. React Native has unified the two mobile APIs [10] into one – making it possible to develop with accessibility in mind with React Native as well.

While support for accessibility is more limited on mobile, the web has more clear defined standards and guidelines on how to make accessible applications – WCAG [11] being the most notable guideline and WAI-ARIA [12] being the most notable technical specification for the web.

Even though React and React Native code is generally similar, the accessibility surface API differs a bit. React Native uses different types of (React) props, such as `accessible`, `accessibilityLabel` and `accessibilityHint`, whereas React (or to be more precise – HTML) uses different types of `aria-*` attributes, such as `aria-label`, `role`, and `aria-controls`. `react-native-web` provides some common API integrations for simplifying accessibility without having to write separate code for each platform; `accessible`, `accessibilityLabel`, `accessibilityLiveRegion`,

[2]Metro Bundler is the JavaScript bundler used by React Native.

[3]A polyfill is code that emulates a feature in e.g. browsers, that do not support the feature natively.

Table 2: Example on how accessibilityRole is mapped to each platform

| **accessibilityRole** | **React Native** | **React (HTML Equivalent)** |
|---|---|---|
| article | `<View accessibilityRole="article" />` | `<article role="article" />` |
| banner | `<View accessibilityRole="banner" />` | `<header role="banner" />` |
| label | `<Text accessibilityRole="label" />` | `<label />` |
| link | `<Text accessibilityRole="link" />` | `<a role="link" />` |
| main | `<View accessibilityRole="main" />` | `<main role="main" />` |

`accessibilityRole`, and `importantForAccessibility`. Each prop is mapped differently depending on the platform [13].

Table 2 lists an example of how the `accessibilityRole` prop is mapped depending on the target platform.

## 2.6 Defining and Measuring Code Quality

A measurement of code quality is hard to quantify – many aspects of the code are somewhat subjective, while some aspects are considered more objective. For example: JavaScript has a feature called Automatic Semicolon Insertion, making semicolons for ending lines unnecessary in most cases. Whether or not to use semicolons is a highly subjective matter for the most part, and has sparked many discussions in the JavaScript community on whether or not using them is considered good practice. It's essential to be able to distinguish which aspects of the code are subjective, and which ones aren't.

There are some standards trying to quantify code (or more generally, software) quality – such as the CISQ Quality Model [14]. Google Lighthouse [15] is a great tool as well (and is partially based on the WAI-ARIA specification), since it provides audits for e.g. performance and accessibility, with the downside of being for the web only. Using parts of it as a base for measuring quality would give an objective view of the quality in some areas.

Since the generated code from the application is in the range of around a hundred lines of code, CISQ's model doesn't translate very well. It does, however, serve as a base for what to look for. Combining CISQ's Quality Model and parts of Lighthouse's measurements would serve as a solid, measurable foundation of aspects that, at least partially, defines code quality in an objective way. A concise list of what defines (a subset of) code quality in the context of the generated code would be the following:

- Accessibility
- Readability
- Maintainability
- General best practices

There are many more aspects of code quality, such as different coding principles (DRY, SOLID, etc.)[4] and tests (unit testing, regression testing, etc.). These aspects are important as well – however, they are outside of the scope of this thesis and hard to apply to the generated code.

### 2.6.1 Formatting with Prettier

A large part of the readability aspect of the code is based on formatting. *Prettier* is an opinionated code formatter for multiple programming languages, including JavaScript [16]. It integrates with several popular code editors, and has a command line interface as well, which the UI Builder will use. All auto generated code will run through Prettier with the following settings:

Table 3: Prettier formatting options

| **Rule** | **Value** | **Description** |
|---|---|---|
| **print-width** | 100 | The line width. Lines longer than 100 characters will be split at a suitable place. |
| **trailing-comma** | es5 | Whether or not to use trailing commas in objects and arrays. Setting it to 'es5' allows the code to be formatted with trailing commas complying with the ECMAScript 5 spec. |
| **single-quote** | true | Whether or not to use single or double quotes for strings. |
| **no-semi** | true | Whether or not to use semi colons. |

All other rules are set to Prettier's default options.

[4]DRY (Don't-Repeat-Yourself) and SOLID (mnemonic acronym) are programming principles used by the programmer to improve the quality of the software being written.

# 3 Theory

In this section, several fundamental design principles, theories, and processes are described. The theories are used to structure the the work process, and to justify and support the results.

## 3.1 User-Centered Design (UCD)

User-centered design is a process which puts focus on the user using the system, based on the ISO-standard 9241-210:2010 [17]. The standard describes six principles to ensure that a design is user-centered:

1. The design is based upon an explicit understanding of users, tasks and environments.
2. Users are involved throughout design and development.
3. The design is driven and refined by user-centred evaluation.
4. The process is iterative.
5. The design addresses the whole user experience.
6. The design team includes multidisciplinary skills and perspectives.

Looking at these principles, the main underlying principle of user-centered design is that the user is involved at each iteration of the design process. This doesn't mean that the user should be consulted at all times – it means that user's needs, behavior, etc. is the central focus in all stages.

According to Gould and Lewis, there are three principles that leads to a good system, described by Preece et al. [18]:

1. **Early focus on users and tasks.**
   This means first understanding who the users will be by directly studying their cognitive, behavioral, anthropomorphic, and attitudinal characteristics. This required observing users doing their normal tasks, studying the nature of those tasks, and then involving users in the design process.
2. **Empirical measurement.**
   Early in development, the reactions and performance of intended users to printed scenarios, manuals, etc. is observed and measured. Later on, users interact with simulations and prototypes and their performance and reactions are observed, recorded, and analyzed.
3. **Iterative design.**
   When problems are found in user testing, they are fixed and then more tests and observations are carried out to see the effects of the

fixes. This means that design and development is iterative, with cycles of "design, test, measure, and redesign" being repeated as often as necessary.

## 3.2 User Testing

User testing is a fundamental technique used in the design process to evaluate usability, as well as allowing insights into where friction emerges when using a system. These insights can then serve as basis for improving the system [18].

User testing is usually modelled as a controlled experiment where a set of users use the evaluated system, all while data is recorded through some medium, such as note taking, audio and video recording, etc. How data is collected is called a moderating technique, and there are multiple different types, each with their own pros and cons. A common moderating technique is called Concurrent Think Aloud (CTA), a technique where the user is encouraged to think aloud during the test session, while their thoughts are recorded by e.g. note taking [19].

CTA has several advantages, such as allowing the moderator to see and understand the test participant's thoughts and emotions throughout the test session, as well as to get real time feedback during the test session. It does, however, come with some drawbacks, such as increasing the actual time taken to complete a task, since the participant has to dedicate some time and attention to talk during the session [19].

## 3.3 System Usability Scale (SUS)

System Usability Scale (SUS) is a standardized questionnaire created by John Brooke in 1986 [20], and is a good method for quickly evaluating a system's usability. It consists of 10 questions with five options ranging from "strongly agree" to "strongly disagree" [20]. It consists of five positive questions and five negative ones, alternating between one of each throughout the questionnaire. This is done to negate acquiescence bias, which is a type of response bias where the users filling out a questionnaire tend to agree with all questions indicate a positive connotation [21].

The ten questions are a variation of the following:

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.

6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

The scoring range for SUS is between 0 and 100. This shouldn't be considered as percentage score, since the scoring isn't linear. Based on previous research, a score of 68 and above is considered above average (and by extension under 68 is considered below) [20], and a score of 80.3 and over is considered great. A score of 51 and under is considered very poor [22].

To calculate a user's score, each question's options are mapped from 1 - 5. For each positive question (i.e. each odd), 1 is subtracted from the selected option. For each negative question (even), 5 is subtracted from the selected option. By adding the resulting values and then multiplying by 2.5 you get the final score for the user [22].

## 3.4 Heuristic Evaluation

Heuristic evaluation is a type of usability inspection method, created by Jakob Nielsen and his colleagues to be able to identify usability problems in UIs [18]. It is seen as quite informal, as opposed to for example user testing, and therefore shouldn't replace it either [23].

Using this technique is done by the evaluator (i.e. an expert) by applying a set of known principles and guidelines to evaluate if UI elements, accessibility features, navigation structure, etc. are acceptable [18]. Using this technique has several advantages, such as being very quick and inexpensive, easily combined with other usability testing methods, as well as giving a clear answer to what can be improved in a design. As with all techniques, it also has its disadvantages – evaluators can be biased, and they are required to have extensive domain knowledge to be able to apply the heuristics effectively [23].

Listed below is one of the most popular guidelines for evaluating usability, developed by Nielsen and his colleagues in 2001 [18].

1. **Visibility of system status** –
   Always keep users informed about what is going on, through providing appropriate feedback within reasonable time
2. **Match between system and the real world** –
   Speak the users' language, using words, phrases and concepts familiar to the user, rather than system oriented terms

3. **User control and freedom** –
   Provide ways of allowing users to easily escape from places they unexpectedly find themselves, by using clearly marked 'emergency exits'
4. **Consistency and standards** –
   Avoid making users wonder whether different words, situations, or actions mean the same thing
5. **Help users recognize, diagnose, and recover from errors** –
   Use plain language to describe the nature of the problem and suggest a way of solving it
6. **Error Prevention** –
   Where possible prevent errors occurring in the first place
7. **Recognition rather than recall** –
   Make objects, actions, and options visible
8. **Flexibility and efficiency of use** –
   Provide accelerators that are invisible to novice users, but allow more experienced users to carry out tasks more quickly
9. **Aesthetic and minimalist design** –
   Avoid using information that is irrelevant or rarely needed
10. **Help and documentation** – Provide information that can be easily searched and provides help in a set of concrete steps that can easily be followed

While Nielsen's guidelines isn't used in this thesis, it serves as an example of how a usability inspection list can look like, and a similar list will be used with the help of Google Lighthouse.

# 4 Method

The work process is split into three main phases – the investigation phase, design phase, and implementation phase, with the implementation phase being the most extensive. The process will be iterative and user-centered throughout all phases. An overview of the iterative work process is shown in fig. 1.

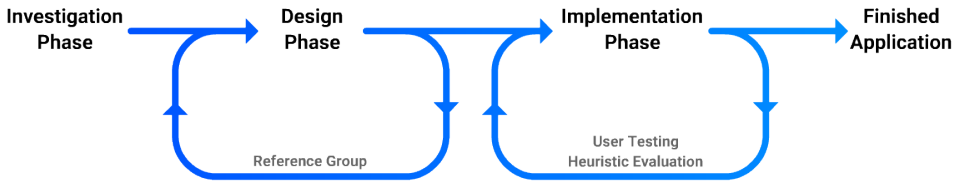



Figure 1: Figure demonstrating the iterative work process

### Investigation Phase

This phase is where I will be looking at existing products and tools, understand the needs of the user, as well as create a Lo-Fi mock-up of the UI Builder to get a general idea of the direction the UI Builder should take.

### Design Phase

This phase will be a cyclic phase, where I will decide on the general functionality, design, and structure of the UI Builder. This is done with the help of a reference group, consisting of two developers.

### Implementation Phase

The implementation phase will be the most extensive phase, and will be a cyclic phase consisting of multiple iterations driven by heuristic evaluations with Google Lighthouse and expert reviews, as well as a user testing.

# 5 Investigation Phase

This phase outlines the foundation of the UI Builder with the scope and goal in mind – what can we say about the user? What should the UI Builder look like to be user friendly?

### Initial Mock-up

Before any actual development took place, several different types of design tools were studied, such as Adobe XD, Android Studio, and Figma. Each design tool had different use cases – for example, Android Studio's design tool is tightly coupled with the code editor as well, since its supposed to be used in conjunction with the code editor. In other words, the design tool in Android Studio isn't supposed to be a complete solution, only a complement to other tools. In contrast, Adobe XD is a complete solution – but strictly for designing and prototyping. Despite this, each design tool had a similar look and work flow. Fig. 2 breaks down the structure of both Adobe XD and Android Studio, highlighting their similarities.

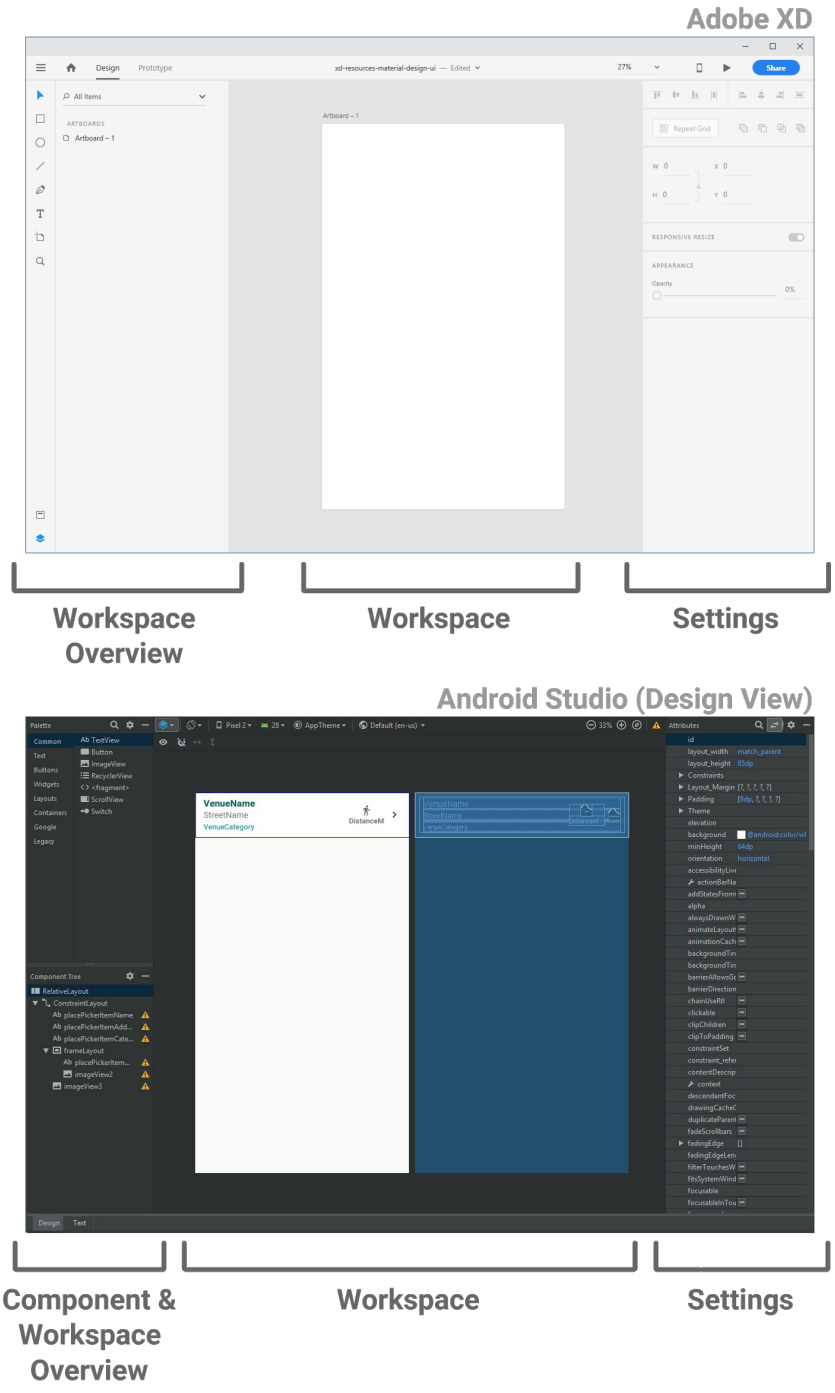



Figure 2: Figure showing the basic layout structure of Android Studio's design view and Adobe XD

It quickly became clear that all studied design tools had a common base layout – two columns on each side used for tools and components, and a centered workspace. Based on these findings a Lo-Fi mock-up was made to outline how the UI Builder was going to be structured (fig. 3).

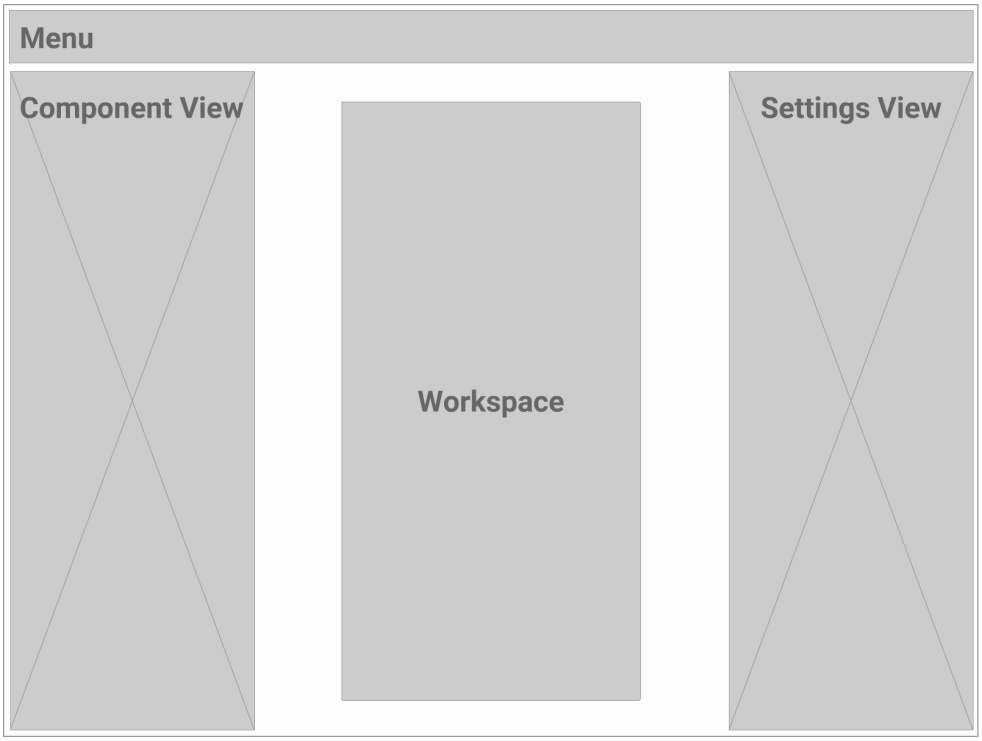



Figure 3: Initial mock-up of the UI Builder application

This was basically the "if it isn't broken, don't fix it" approach. The layout was good enough for the purpose of the UI Builder, was known to work (based on its prevalence in so many interface design tools), and users with any experience with these tools (and many others) were used to it.

### Understanding the User

During this stage, some thought was put into who the user of the UI Builder was going to be. Basically, a range of users were the target for the application, ranging from developers with little to no design experience, to designers with at least some programming experience. This meant that some assumptions could be made – for example, assuming that they're familiar with some form of Integrated Development Environment (IDE) or design tool.

# 6 Design Phase

In this phase the UI Builder's general structure, functionality, and UI design were outlined.

## 6.1 User Interface (UI)

### 6.1.1 UI Builder

With the feedback of a reference group consisting of two developers (one web developer), a Mid-Fi prototype based on the Lo-Fi prototype was designed (fig. 4), outlining the interface of the UI Builder.

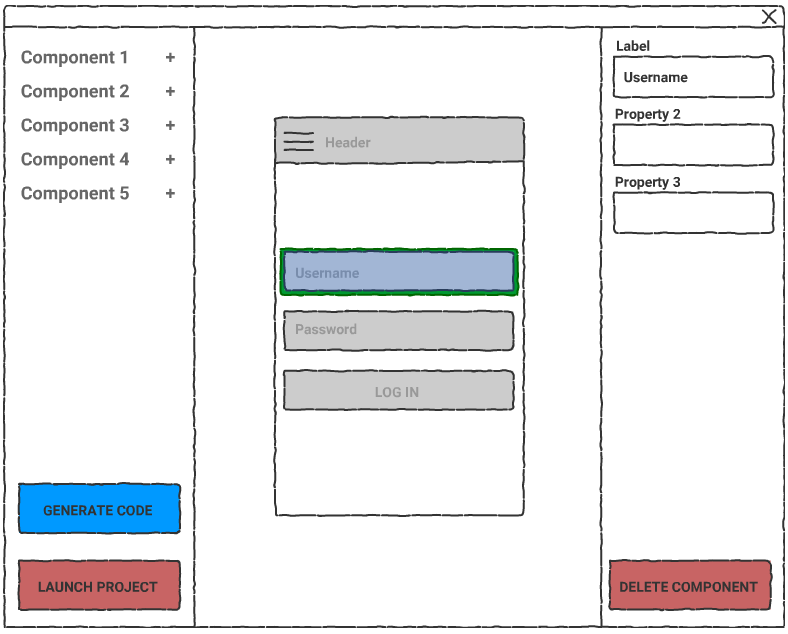



Figure 4: Mid-Fi drawing outlining the functionality of the UI Builder

**Picking the right application framework** – Ideally, the UI Builder would work on any OS – a common shortcoming of several design tools, such as Sketch (which is macOS only). Tools like Figma and UXPin works around this by being web based, which comes with some drawbacks, such as limited access to the local filesystem. After looking at different frameworks it looked like Electron, developed by GitHub, was a good solution: cross-platform, web based, and with access to the local system. This meant that UI Builder could be developed as a desktop application that works

on Windows, macOS, and Linux, using web technologies for the UI (in this case, React), and full local system access using Node.js.

**Picking the right design** – The UI Builder had to have a solid design foundation to build from, since the UI to a large extent affects the UX as well. Google's Material Design seemed to fit the bill – it's a very popular collection of design guidelines and principles created by Google [24], meaning that it's battle tested and has a lot of community support. Since the UI Builder's frontend was going to be written in React, the popular React UI component library `material-ui` was the perfect library for implementing the Material Design guidelines.

### 6.1.2 Cross-Platform Components

Using Material Design had another benefit – it works great with mobile UIs as well (it is the standard on Android after all). Because of this, Material Design was a good fit for the generated code's cross-platform components as well. There were some limitations on which library could be used (or if the implementation had to be done from scratch), since the components had to work both on mobile and web. After looking at a few libraries, `react-native-paper` was selected. It was a reasonably good implementation of the Material Design guidelines, and it worked well with web, Android, and iOS.

## 6.2 User Experience (UX)

### 6.2.1 Building UIs with the UI Builder

The UI Builder needed an intuitive way of building app UIs. A good way to intuitively interact with "building blocks" is through drag-and-drop functionality. A mock-up of the intended drag-and-drop functionality is shown in fig. 5. The figure shows how it's intended to manipulate the UI layout in the UI Builder's workspace.

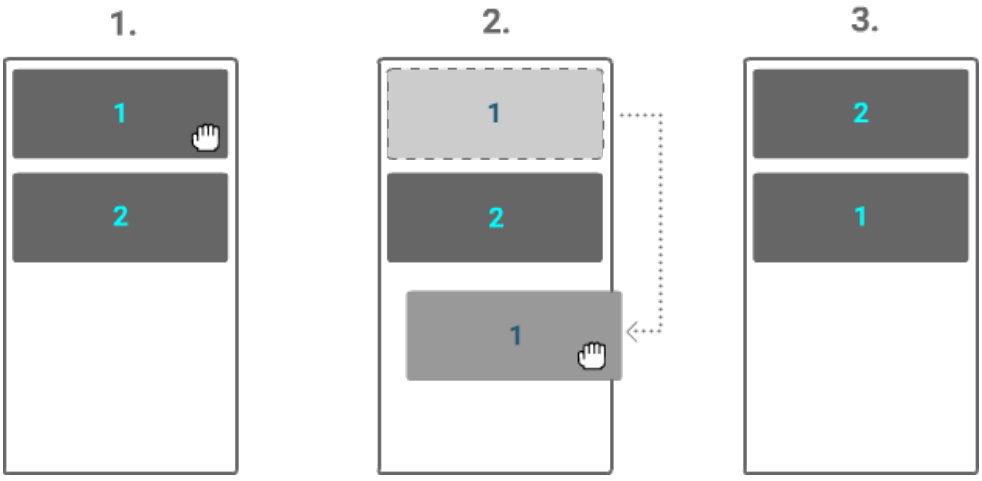



Figure 5: Mock-up showing the intended drag-and-drop functionality of the UI builder

### 6.2.2 Live Reload

Live reload is a tool that can tighten the feedback loop of designing a UI and seeing it on a device. The live reload works by monitoring the file system for changes, and triggers a rebuild of the React Native and/or web app based on that change. The intended live reload flow of the UI Builder can be seen in the sequence diagram in fig. 6.

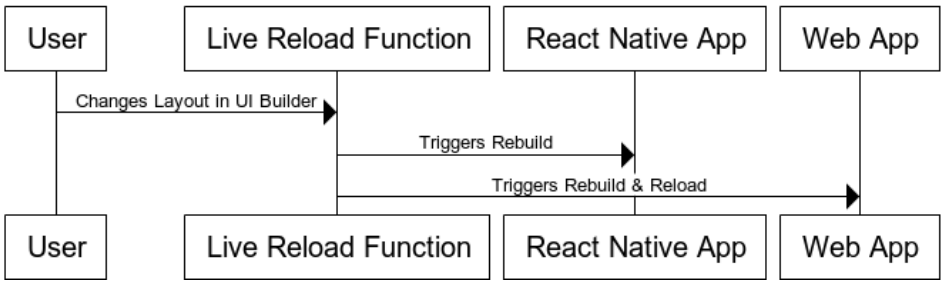



Figure 6: Sequence diagram describing the intended the live reload functionality in the UI Builder

# 7 Implementation Phase

The implementation phase was the most extensive phase, consisting of multiple iterations. It covers code generation, user testing, testing accessibility & code quality, and finalizing the interface of the UI Builder.

## 7.1 Interface Design

The UI was created with the help of `material-ui`, a collection of React UI components implemented with Google's Material Design guidelines in mind, meaning that the UI Builder had a solid foundation for creating a good User Experience.

Based on the initial Lo-Fi (fig. 3), Mid-Fi prototype (fig. 4), and the other decisions made during the design phase, the resulting application UI shown in fig. 7 was developed.

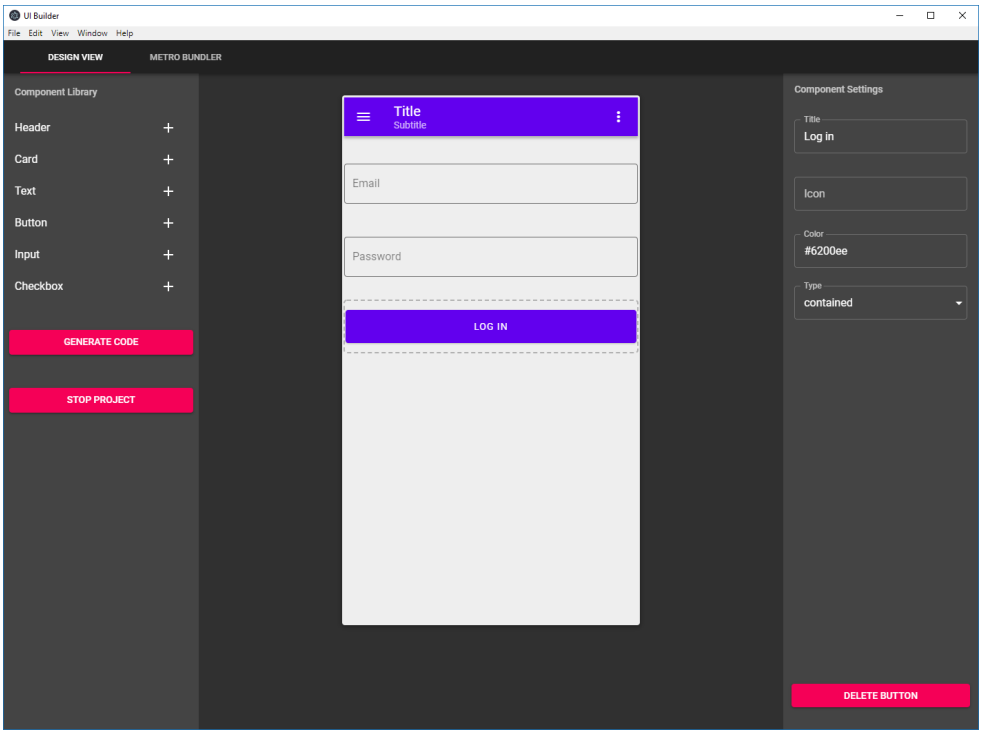



Figure 7: Implemented design view tab of the UI Builder.

### 7.1.1 Layout Manipulation

A key feature implemented in the UI Builder was the ability to move components around, as well as adding child components to some. This was implemented using the library `react-beautiful-dnd`, a drag-and-drop library with a lot of focus on UX. Using this library meant that several aspects of the interaction with the UI Builder became more intuitive, mainly by leveraging the built in animations of `react-beautiful-dnd`. An example is shown in fig. 8, which shows the dragging state of a card component.

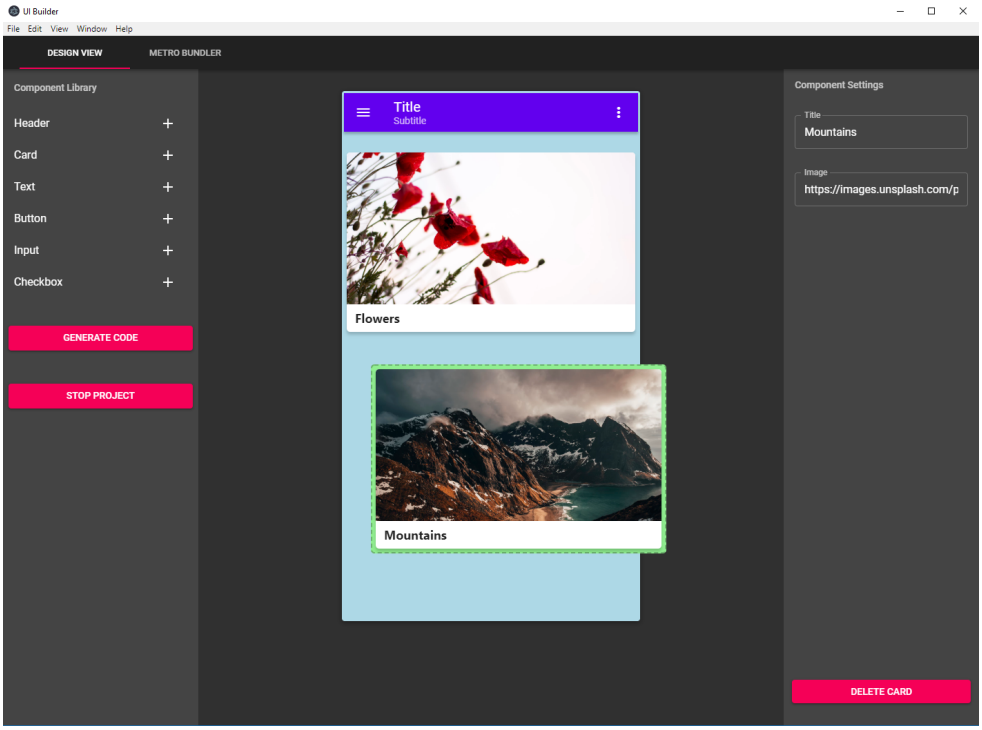



Figure 8: What the UI Builder looks like when dragging the card component with the title **Mountains**.

#### Adding Child Views

Since the `Card` component was basically a container, it needed a way of intuitively adding child components to it. The action was implemented as a drag-and-drop feature as well, shown in fig. 9.

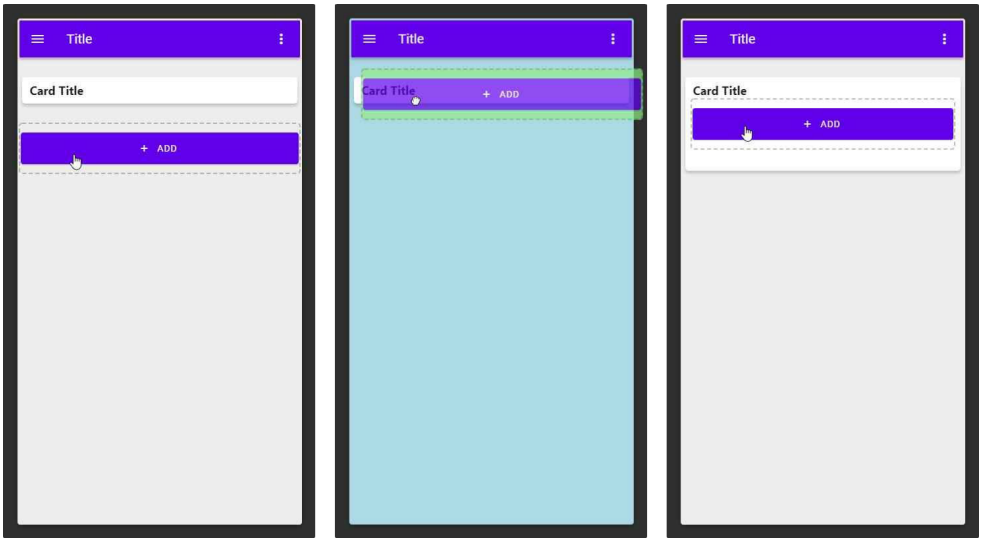



Figure 9: Figure showing the action of adding a `Button` into a `Card`. Note the decreased opacity of the `Button` component in the second image, indicating the drop functionality.

### 7.1.2 React Integration

React (web) integration was done with the use of a modified version of `create-react-app`, which relies on an internal library named `react-scripts`. This library is a "batteries included" CLI which starts a preconfigured Webpack bundler as well as a web server, serving the generated web app on the local machine's port `3001`, with support for live reloading. Running a predefined command[5] (done from the UI Builder's backend) in the generated code's project root started both the web server and Webpack.

The web server and bundler was started by clicking the *Launch Project* button in the design view tab. This in turn launched the OS's default browser and loaded the web app on `http://localhost:3001`.

### 7.1.3 React Native Integration

Integrating React Native with the UI Builder was done by leveraging Expo's CLI. Running `expo start` (done from the UI Builder's backend) in the generated code's project root starts React Native's Metro Bundler (with live reloading support included) as well as a web server running a user interface for the Metro Bundler. This

[5]The following command: `cross-env PORT=3001 react-app-rewired start`. Normally, **react-app-rewired** is replaced with **react-scripts**, however, a modified Webpack configuration required **react-app-rewired**.

interface was integrated into the UI Builder and accessible through a separate tab view, seen in fig. 10.

Integrating this interface had several benefits:

- The user never has to touch a terminal to run the app on a device or simulator
- The QR code allows live testing on any Android or iOS device within seconds by scanning it with the device
- Logs from all test devices are shown in one place
- No network limitations (unlike when using the React Native CLI only), made possible by Expo's utilization of `ngrok`[6]

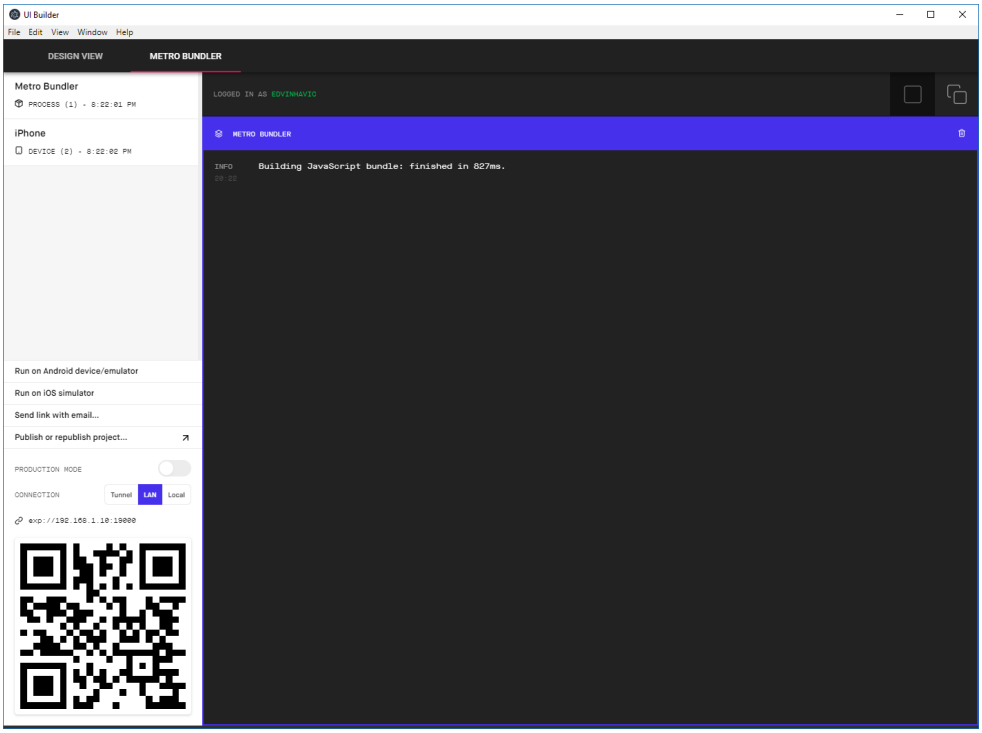



Figure 10: The Metro Bundler tab in the UI Builder.

Expo's Metro Bundler was started by clicking the *Launch Project* button in the design view tab. Accessing the generated app was done by either launching a simulator, or by having the Expo app installed on a device and scanning the QR code.

[6] **ngrok** is a CLI tool that creates a tunnel from the public internet (by exposing a public URL) to a specified port on the local computer.

## 7.2 Boilerplate for the Generated Code

Since the generated code was going to be used for both React and React Native (by using `react-native-web`), traditional boilerplate projects (using `create-react-app`[7] for example) weren't sufficient. Developing a boilerplate suitable for usage with web and mobile proved to be challenging, since the boilerplate had to support multiple things both related to mobile and web, such as Expo, JSX, and component libraries. Much of this was achieved by starting with the `create-react-app` boilerplate and tailoring it to the specific requirements.

The biggest problem was that when publishing packages for web use (e.g. component libraries such as `material-ui`), the standard practice is to transpile the code with a transpiler[8], most likely Babel. This is done since the source code might be written in JSX, TypeScript, or it might be using language features not supported in browsers yet. However, this is not standard practice with packages meant to be used with React Native. Since browsers can't understand anything other than vanilla JavaScript, packages that weren't transpiled made the boilerplate unable to run on the web. The most obvious solution would be to transpile all code dependencies to make sure the code can run on all platforms. This does however cause two issues – the first being that there are thousands of dependencies, meaning that the build time would dramatically increase. The second issue is that the external dependencies does not only contain JavaScript code – it contains platform specific code that the transpiler can't and shouldn't touch. These issues were mitigated by essentially cherry-picking which packages and files were needed to be transpiled.

To build the native components (in this case, `react-native-paper`) for the web, an alias for `react-native` was created, to `react-native-web`. This essentially replaced all occurrences of `react-native` in the source code with `react-native-web`.

The boilerplate structure is shown in fig. 11.

[7] **create-react-app** is a popular npm package for generating boilerplate code, created by Facebook.

[8] A JavaScript transpiler is a program that takes JavaScript (or supersets of JavaScript such as JSX and TypeScript) code as input and outputs equivalent code in a specified version of JavaScript.

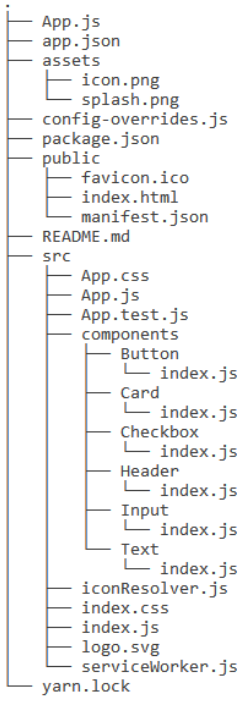



Figure 11: Boilerplate structure for the generated code.

## 7.3 Implementing Cross-Platform Components

The purpose of creating the components were to create building blocks for the generated application, as well as having draggable components in the UI Builder.

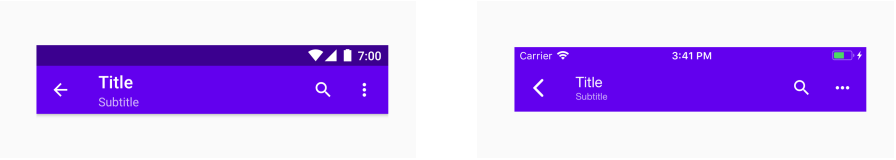
Each component was split into two – one for the visual editor, and one used for the actual generated code. Both parts are nearly identical visually, with some differences in code, such as error boundaries[9] and wrappers to make them draggable in the editor. This also meant that there were some double maintenance which made code changes problematic, since changes had to be kept consistent with each other. It was, however, a necessary evil since the components weren't identical for each use case and the code wasn't easily shareable between the two codebases.

A minimal implementation was made with six different components, making it possible to create a variety of different views. The component library could be vastly expanded with more components, however, the six components were deemed to be enough to demonstrate the basic functionality of the application.

[9]Error boundaries are a part of the **react-dom** library, used to catch JavaScript errors thrown by any child view wrapped by the boundary.

### Header

Used as a header at the top of the screen. It can contain a title, subtitle, and icons on the left and right.

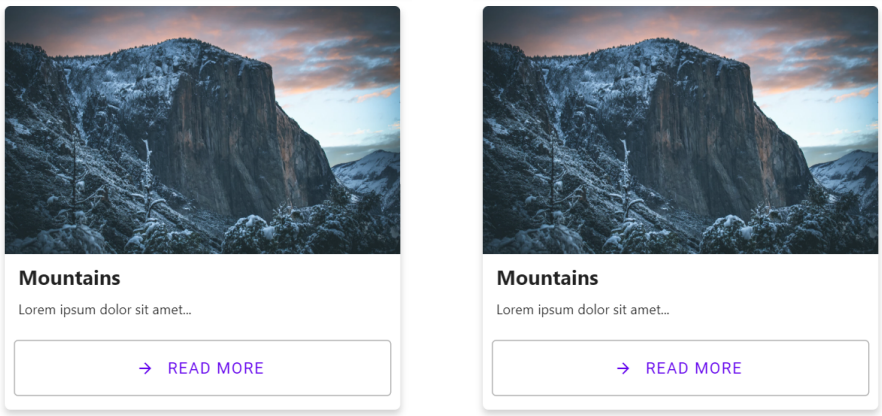



(a) Android (b) iOS

Figure 12: Example header on Android and iOS

### Card

Used as a content and action container about a single subject. A card can contain children such as buttons and text, as well as have a heading and an image.

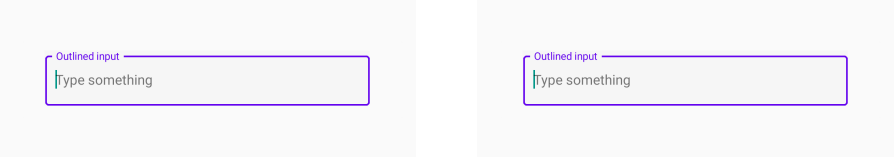



(a) Android (b) iOS

Figure 13: Example card on Android and iOS

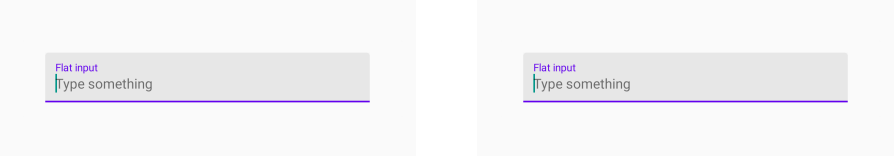
### Text Input

A component used for text input. A text input can be either outlined (fig. 14) or flat (fig. 15) and has support for passwords, labels, and placeholders.




(a) Android (b) iOS

Figure 14: Example outlined input on Android and iOS

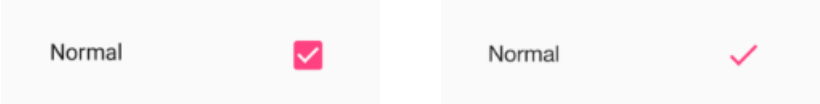



(a) Android (b) iOS

Figure 15: Example flat input on Android and iOS

### Checkbox

A component used for selecting one or more options in a set.




(a) Android (b) iOS

Figure 16: Example checkbox on Android and iOS

### Text View

A Text View is used for displaying text such as paragraphs and titles. It supports different colors, font sizes, alignments, and weights.


(a) Android (b) iOS

Figure 17: Example text view with on Android and iOS. Font size 24, font weight 600, center aligned, and #**222** set as color.

## 7.4 Mapping React Components to Code

The components used in the UI Builder are basically a visual representation for a data structure used for generating the actual code. For example, the `Button` component's data structure is seen in listing 1. The structure borrows a lot of concepts from component data representation in React – for example, `propTypes` [25] and `children` [26].

```
{
  displayName: 'Button',
  name: 'button',
  component: Button,
  children: [],
  canHaveChildren: false,
  props: {
    title: 'Button',
    color: '#6200ee',
    icon: null,
    type: 'contained',
  },
  propTypes: {
    title: 'string',
    icon: 'string',
    color: 'string',
    type: {
      type: 'string',
      oneOf: ['text', 'outlined', 'contained'],
    },
  },
},
```

Listing 1: Button default JSON representation.

This data structure (listing 1) is used in the UI Builder's backend for generating code. Fig. 18 shows an example layout in the UI Builder (and the corresponding result for iOS in fig. 19), and listing 2 shows the generated code based on that layout.

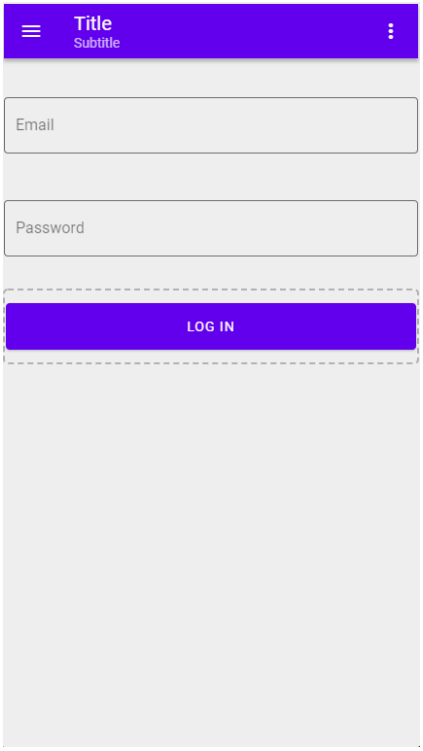



Figure 18: Example layout in UI Builder.

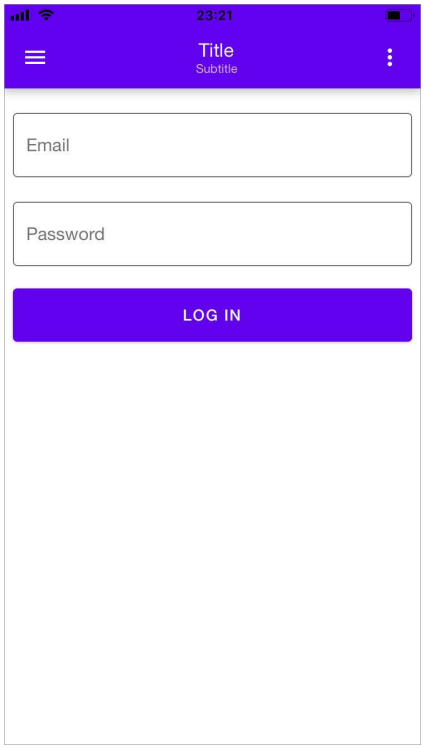



Figure 19: Corresponding result on iOS, based on the layout shown in fig. 18.

```
import React from 'react'
import { View, ScrollView, StyleSheet } from 'react-native'

import Header from './components/Header/'
import Input from './components/Input/'
import Button from './components/Button/'

const styles = StyleSheet.create({
  container: {
    flex: 1,
  },

  scrollContentContainer: {
    padding: 8,
  },
})

const App = () => (
  <View style={styles.container}>
    <Header
      title="Title"
      subtitle="Subtitle"
      leftIcon="menu"
      rightIcon="more-vert"
      placement="center"
      backgroundColor="#6200ee"
      foregroundColor="white"
    />
    <ScrollView contentContainerStyle={styles.scrollContentContainer}>
      <Input placeholder="email@example.com" mode="outlined" label="
          Email" password={false} />
      <Input placeholder="Password" mode="outlined" label="Password"
          password={true} />
      <Button title="Log In" color="#6200ee" type="contained" />
    </ScrollView>
  </View>
)

export default App
```

Listing 2: Generated code for the views seen in fig. 18 and 19.

## 7.5 User Testing

### 7.5.1 Method

At this point user testing was started, since the UI Builder was deemed functional enough to give meaningful feedback from a test. A moderating technique called Concurrent Think Aloud (CTA) was used, meaning that the user is encouraged to think aloud, and in turn gives real-time feedback and without having to mentally take notes on what they thought at a given moment [19]. The screen and the user's

voice was recorded as well. This was done to be able to go back and check for possible problems and pain points, as well as seeing what the user was thinking at a given time. The test was performed on a group of users ($n = 5$) with varying programming and design experience, since that was the target group. A summary of each test can be seen in appendix A.

The tests were structured as two goal based scenarios, one basic (login screen) and a more advanced one (card based feed). This was done to evaluate the complete implementation of the application. Each user was given a short description of both scenarios and an outline of what the application does without giving any information about how they should complete the given tasks. The users were also given a short list of available icon names that they could use for some components[10].

After completing the two tasks, the user was asked to fill in a SUS questionnaire (described in 3.3). The 10 questions asked were the following:

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

After the SUS was filled in, the user was asked three additional questions:

- Did the live reloading functionality (instant feedback on the phone/browser) aid your design process in any way?
- Were there any parts of the application that you feel could be improved?
- Other feedback, if any.

After filling in the form, a short discussion was had with the user, taking notes on additional things the user might have had to say.

#### Scenario 1: Implementing a Login screen

This was the first scenario the user was given. A crude sketch (see fig. 20) was given to the user to simulate something a client or any other person would draw as an idea.

[10]Three different icon names were given: **menu**, **face**, and **image**.

No information was given other than the short list of icon names.

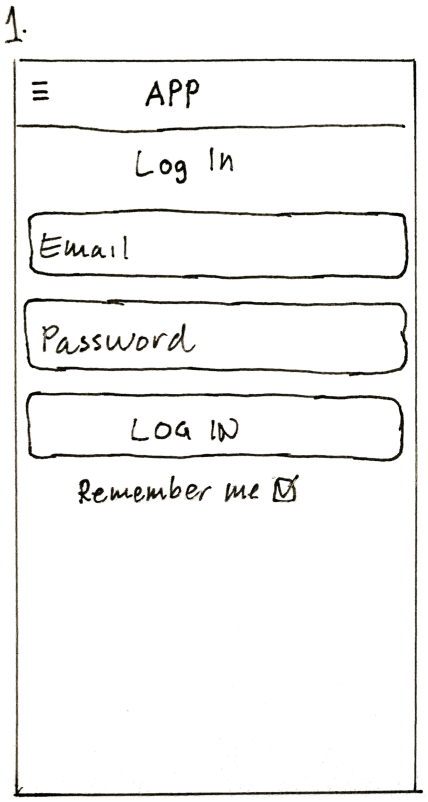



Figure 20: Sketch of a login screen used for the first scenario

**Scenario 2: Implementing a card based feed**

After getting a bit familiar with the UI builder in the first part of the user test, a more complex sketch was given (fig. 21). The point of the more complex layout was to test a broader spectrum of the application, such as adding images to cards, adding child components, and using icons.

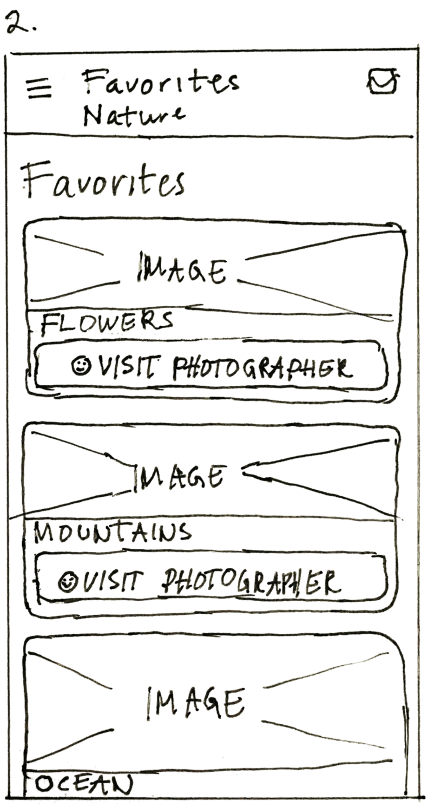



Figure 21: Sketch of a card based feed used for the second scenario

### 7.5.2 Results

Averaging the SUS score from each user gave a score of 82.5, a surprisingly good number. The feedback overall was positive, though there were some pain points, meaning that there is room for improvement. Summarizing the feedback from each user gave some concrete areas that could be improved, as well as what worked well. This summary can be seen in table 4.

Figures 22 to 31 shows the results for each question in the SUS questionnaire.

Table 4: Summary of positive and negative feedback based on user tests.

| Positive | Negative |
|---|---|
| Users were very quick to get up and running with minimal information about the application. The time taken to complete Scenario 1 varied between 3 - 9 minutes, and Scenario 2 varied between 5 - 11 minutes[11]. | Some prior knowledge about e.g. components would be handy[12]. For example, knowing that the `Image` prop on the `Card` component is a URL would be good, as well as knowing which components can have child components[11]. |
| Most users (80%) used the instant feedback on the phone/browser at least once during development[11], and felt that it improved their experience[12]. | Some UI "traps", such as the delete button being mistaken for a confirm button[11] [12]. |
| The components were flexible enough to cover common use cases[11]. | Not selecting the newly created component (when adding a component) potentially confuses users, since most expect the added component to be selected by default[11]. |
| The resulting web & iOS app's "look n' feel" was satisfactory for the users[12], and mostly followed Google's Material Design guidelines[11]. | Not being able to drag & drop from the component library can confuse users, since there is drag & drop functionality in the workspace view[11]. |
| There was a "wow" factor when the user used the live reload feature[12]. | General features that users expect are missing, such as copy-paste functionality for components[12]. |
| | More advanced layouts such as split views are missing[11]. |
| | Some confusing component names. For example, several users mistook the `Header` component for a heading, when the component that should've been used was the `Text View` component[11]. |

[11] Objective data, either observed or noted during user testing.
[12] Subjective data, either from the test participant or the test leader.
[13] Score range is 1 - 5, where 1 = Strongly disagree, 5 = Strongly agree.

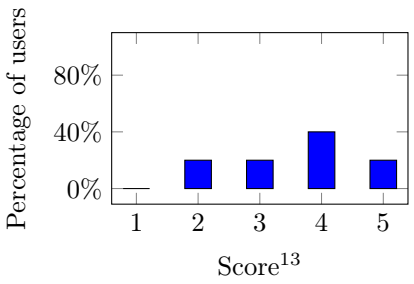



Figure 22: Question #1: I think that I would like to use this system frequently.

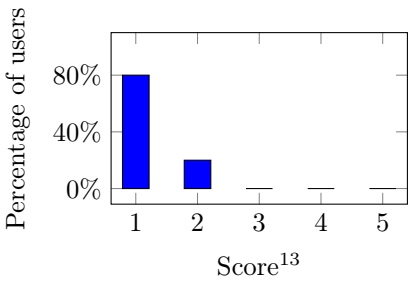



Figure 23: Question #2: I found the system unnecessarily complex.

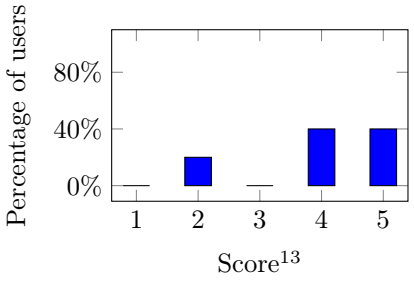



Figure 24: Question #3: I thought the system was easy to use.

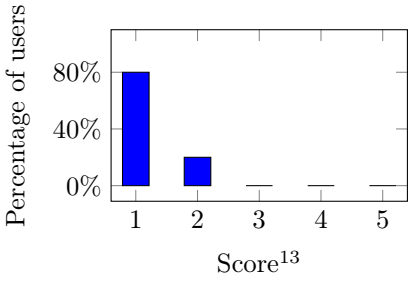



Figure 25: Question #4: I think that I would need the support of a technical person to be able to use this system.

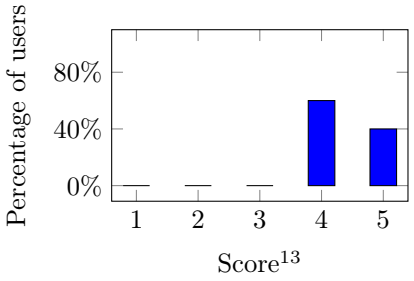



Figure 26: Question #5: I found the various functions in this system were well integrated.

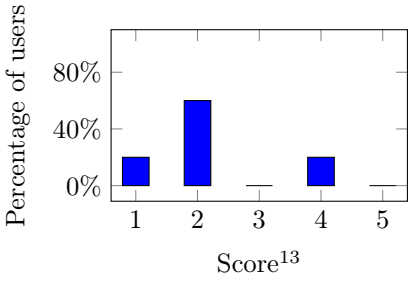



Figure 27: Question #6: I thought there was too much inconsistency in this system.

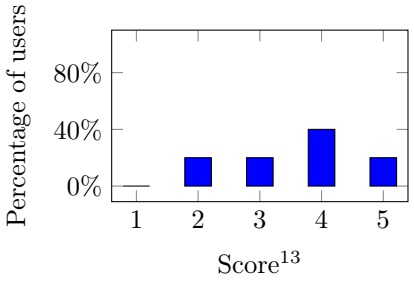



Figure 28: Question #7: I would imagine that most people would learn to use this system very quickly.

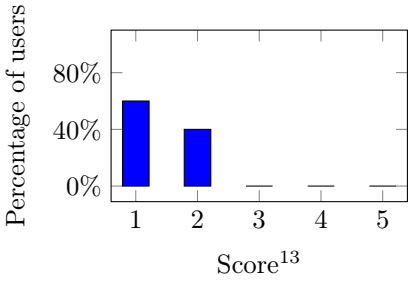



Figure 29: Question #8: I found the system very cumbersome to use.

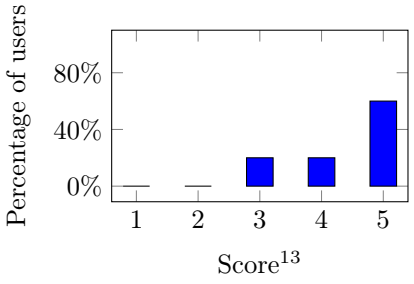



Figure 30: Question #9: I felt very confident using the system.

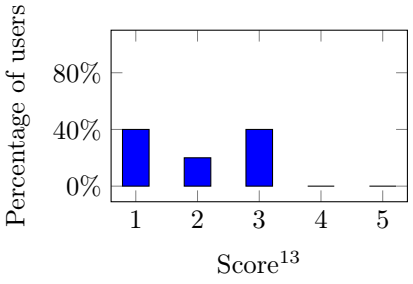



Figure 31: Question #10: I needed to learn a lot of things before I could get going with this system.

## 7.6 Accessibility Audit with Lighthouse

Using Lighthouse enabled testing multiple aspects of the generated web app, mainly: Accessibility, Performance, Best Practices, and Progressive Web App (PWA) aspects. In this case, the performance aspect of the audit was completely ignored, since the audit was run on non-minified[14] code in development mode, meaning that performance was impacted. The PWA aspect was also ignored.

A simple view was designed in the UI Builder, shown in fig. 32. This view tested 4 out of 6 components, meaning that the accessibility score could vary depending on which other ones were used.

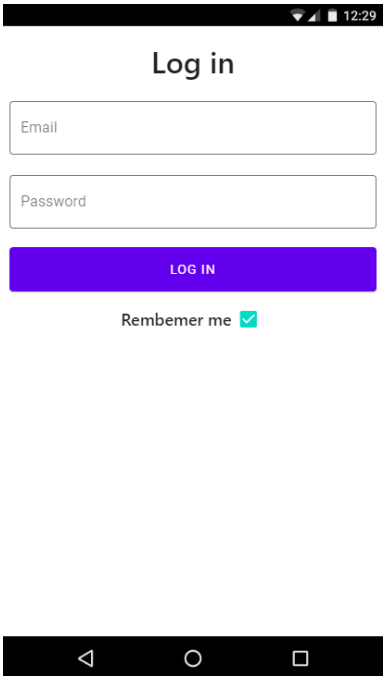



Figure 32: View used for the heuristic evaluation with Lighthouse, rendered on the web.

Running the Lighthouse audit on this view gave an overall score shown in fig. 33. Since both the PWA and performance aspects were ignored, the only score that was in need of improvement was the accessibility score.

[14] JavaScript code that is going to be run in a browser is usually minified and optimized by a bundler, such as Webpack.


Figure 33: Overall Lighthouse score for the view shown in fig. 32.

### Looking Closer at the Accessibility Results

The Lighthouse accessibility audit looks at a total of 46 different aspects. Of those aspects, 17 weren't applicable, since they were related to elements what weren't used (e.g. `<audio>`, `<video>`, `<iframe>`, etc.). Additionally, 12 of the remaining aspects had to be manually checked, meaning that the accessibility score was based on 17 aspects. The accessibility results showed that 14 aspects passed, and 3 had failed. The aspects passed were the following:

1. `[aria-*]` attributes match their roles
2. `[role]`s have all required `[aria-*]` attributes
3. Elements with `[role]` that require specific children `[role]`s, are present
4. `[role]`s are contained by their required parent element
5. `[role]` values are valid
6. `[aria-*]` attributes have valid values
7. `[aria-*]` attributes are valid and not misspelled
8. Background and foreground colors have a sufficient contrast ratio
9. Document has a `<title>` element
10. `[id]` attributes on the page are unique
11. `<html>` element has a `[lang]` attribute
12. `<html>` element has a valid value for its `[lang]` attribute
13. `[user-scalable="no"]` is not used in the `<meta name="viewport">` element and the `[maximum-scale]` attribute is not less than 5
14. No element has a `[tabindex]` value greater than 0

The failed aspects were the three following:

1. Buttons do not have an accessible name
2. Image elements do not have `[alt]` attributes
3. Form elements do not have associated labels

Looking closer at the failed aspects showed that the `Checkbox` and `Input` components weren't accessible enough. Looking at the HTML revealed that the checkbox

consisted of a styled `<div>` element with the attribute `role` set to `"button"`, which is incorrect (`role="checkbox"` would be correct in this case). It also had a `<div>` (representing the checkmark) with the `role` attribute set to `"img"`, but with the required `alt` attribute missing. This meant that aspect 1 and 2 were related to the checkbox.

The HTML also revealed that the `Input` component didn't have an actual `label`, meaning that aspect 3 was related to this component.

## 7.7 Improving Web Accessibility

Since both the `Input` and `Checkbox` component were implemented using the `react-native-paper` library, there were two options to improve the accessibility: Either modify the library's source code, or swap the components for a better implementation when running on the web. The latter was chosen in this case.

Using `react-native-web`'s `Platform.OS` API, the checkbox component was swapped for `material-ui`'s `Checkbox` component when rendered on the web. This was done since the `material-ui` library adheres to the Material Design guidelines and the checkbox was implemented with accessibility in mind.

### Accessibility Results

Swapping out the `Checkbox` component when rendering on the web improved the accessibility, as can be seen in 34.


Figure 34: Overall Lighthouse score for the view shown in fig. 32 after improving the checkbox component.

The accessibility score could have been further improved by modifying the `Input` component as well. This would result in a perfect accessibility score, since only failed aspects 2 and 3 were now passing by fixing the `Checkbox` component.

## 7.8 Improving Native App Accessibility

Testing the view shown in fig. 32 with iOS's VoiceOver revealed that the `Input` component could be improved. When selecting an input, the VoiceOver assistant didn't

read the label, meaning that the user wouldn't get the information required to correctly fill in the input. Fortunately, React Native input components have support for the `accessibilityLabel` prop. Two approaches could be taken, each with its drawbacks. Either a `accessibilityLabel` input could be exposed to the designer in the UI Builder, or the already existing `label` could be used for the `accessibilityLabel` as well. The latter was chosen, since this is the desired behavior in the majority of cases and didn't add any complexity to the UI Builder interface.

#### Accessibility Results

Adding the `accessibilityLabel` to the `Input` component resulted in the VoiceOver assistant being able to read the input correctly.

The accessibility could be further improved by using other accessibility props, such as the `accessibilityHint`. However, using `accessibilityLabel` was deemed good enough for this usecase, and had the added benefit of not adding more complexity to the UI Builder interface.

## 7.9 Improving the UI Based on User Testing

After analyzing the user tests it became clear that there were some improvements (noted in table 4) to the UI that could be made – mostly in the form of adding signifiers. The updated UI can be seen in fig. 35.

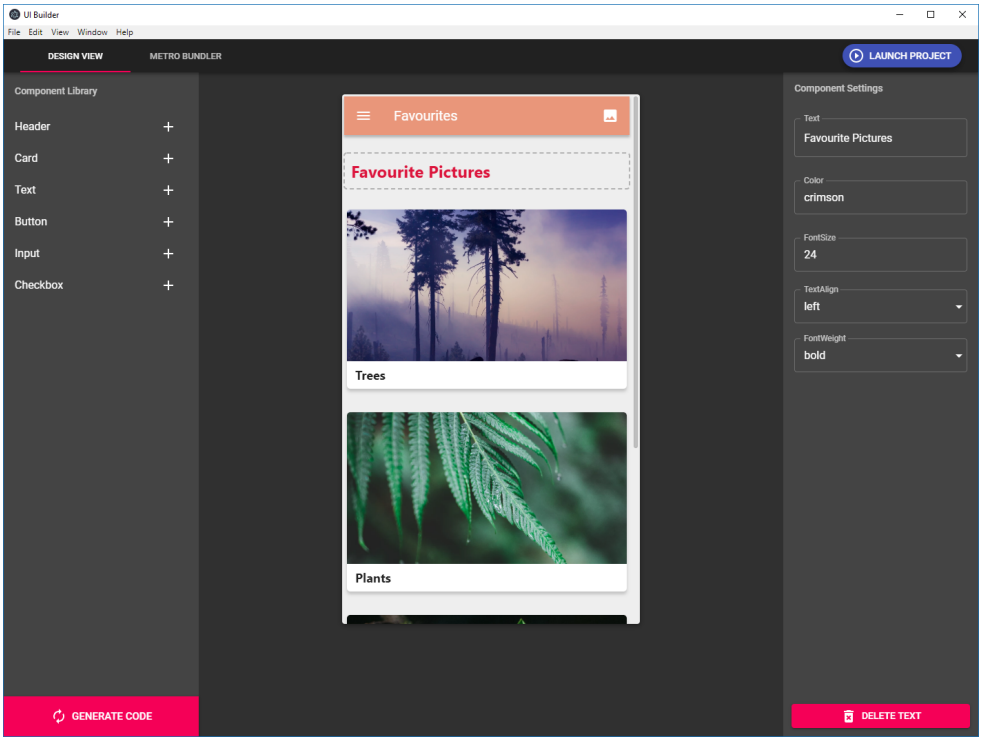
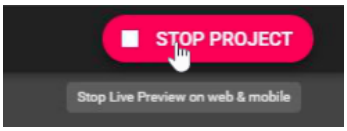



Figure 35: UI Builder's updated UI based on user testing

The following changes were made:

**The Launch Project Button** This button, positioned in the left drawer before, was moved to the top right corner in the menu bar. This was done because the button isn't used often. The color was changed to a primary color, since it still is one of the primary actions when starting the UI Builder. When clicked however, it changes to a secondary color, since it isn't as important after the initial click. A tooltip and icons were added as well, further signifying its action. The changes can be seen in fig. 36.


(a) Launch Project button on hover




(b) Launch Project button on hover (when live reload is started)

Figure 36: The Launch Project button's two states on hover.

**Generate Code Button** This button received some minor changes as well. It was moved to the bottom left corner, an icon was added, and its size was increased. The changes can be seen in fig. 37.

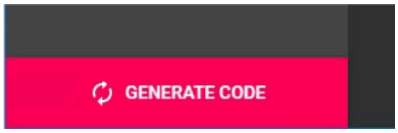



Figure 37: The updated Generate Code Button

**Delete Component Button** Only a delete icon was added to this button. Seen in fig. 38.

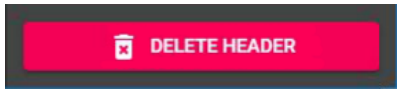



Figure 38: The updated Delete Component Button

**Add Component Buttons** Only a tooltip was added to these buttons to signify that they were supposed to be clicked, not dragged – something that initially confused users during the tests. The change can be seen in fig. 39.

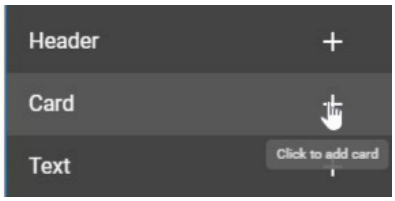



Figure 39: The updated Add Component Button

## 7.10 Evaluating the Generated Code Quality

Since the UI Builder isn't supposed to be a complete tool for building apps, the generated code must be of good quality to enable developers to build upon the foundation that the UI Builder provides. Using the code quality metrics defined in 2.6, another heuristic evaluation was done on generated code produced by the UI Builder. The code used for the evaluation, as well as the expert's answers, can be found in appendix B. The following questions were asked:

1. Is the code readable and easy to understand?
2. Does the code follow general best practices?

3. How many changes would you need to do in order to be able to use the code?
4. Is the code of high enough quality to be usable as a foundation?

The evaluation in this case was done by a developer with 4 years of experience with React, as well as 7+ years of experience with JavaScript. The result was a net positive, with some minor room for improvement. For example, creating identical components leads to repeating the same code over and over – a side effect of not being able to implement any logic. Overall, the code was deemed to be of high enough quality to be used as a base for further development.

# 8 Discussion

This section discusses limitation, future work, as well as what went well in the process – and what could've been done better.

First off, looking at the user tests, the results from the tests were great – a SUS score of 82.5 is far above average. While I do believe that the "actual" score isn't very far from that, given the small sample size ($n = 5$), it needs to be considered as a source of error. Another aspect to consider is the fact that the test participants were "sampled by convenience", meaning that they were people available and close by during the time the tests were conducted. The reasoning for these two error sources is of course sound – finding a large, random, sample size within the target group wouldn't be feasible given the time frame and scope of the thesis. The users were, however, representative of the target users, and a lot of objective and qualitative (along with the quantitative) data could still be gathered from the users. This also meant that a lot of the interpretations made from the data could be used as, for example, a foundation for upcoming iterations.

Taking these error sources into consideration, the user testing still gave reliable and usable results. The final iteration of the UI Builder proved to be intuitive and simple to use – something that points towards an effective user testing methodology.

## 8.1 Limitations

As with most projects, there were several limitations – mostly due to manpower and time constraints. As the sole developer of the UI Builder, a lot of things that would be great to have implemented, simply had to be skipped due to being too big of a feature, or due to not having enough time to implement them. The most notable limitations were related to the library `react-beautiful-dnd` – while fitting the use case of the UI Builder to about 90%, the remaining 10% required were simply not worth implementing, since it would require modifying a large part of the library's source code. For example, `react-beautiful-dnd` does not support nested components. Some clever workarounds in the implementation enabled the UI Builder to have support for it, but more advanced features such as multiple layers of nesting were off limits. This in turn meant that the UI Builder couldn't support creating more complex layouts, at least not without a large rewrite of `react-beautiful-dnd`.

## 8.2 Future Work and Improvements

As mentioned in section 8.1, the library `react-beautiful-dnd` was at times like trying to fit a square peg into a round hole. While it provided a lot of things for free, either developing a custom drag-and-drop implementation or modifying the library's

source code (it's open source after all) would be required, in order to have a better starting point for future work.

And speaking of future work – there's a lot that can be done. The UI Builder has served as a great proof of concept, but it needs a lot of work to become a viable tool for everyday use. Some potential starting points for improvements (in no particular order) could be:

- **Expanding the component library**
  Why this is considered an improvement is quite clear – a greater selection of components gives the user a lot more flexibility and opportunities when creating apps. Beyond just expanding the selection, creating responsive components could yield better designs on other viewports and resolutions as well, such as desktop browsers.
- **Importing custom components**
  The current state of the UI Builder can be considered as a bit of a "cookie cutter" approach when it comes to design, even though it has a bit of customizability built in. Enabling users to import custom components would make the UI Builder appeal to users who want components more tailored to their needs.
- **Multiple design views**
  The UI Builder's workspace is currently a single 16:9 aspect ratio window – resembling the average smartphone viewport. Allowing other viewports and resolutions would most likely improve the User Experience.
- **Enabling logic implementations**
  A side effect of not being able to use any logic in the designs is that it leads to a lot of repeated code. Consider the card based feed shown in fig. 21 – having a list of identical components leads to a lot of repeated code, something that could easily be mitigated by, for example, mapping an array of data and dynamically generating the components.
- **Better accessibility support**
  While the accessibility audits (sections 7.7 and 7.8) showed that the accessibility support was acceptable, there can always be better tooling for accessibility support. A good starting point would be supporting a wider range of accessibility related props provided by `react-native-web`, such as `accessibilityHint` for example.

# 9 Conclusion

Circling back to the original goals mentioned in section 1.1 – have they been reached? Did the work succeed in providing a visual design tool for three different platforms? In short, yes. However, the work done in this thesis has barely scraped the surface of the possibilities and issues with this cross-platform approach. Taking a closer look at each goal gives a bit more detail about each area:

***Code Quality***
*"For the generated code to be usable, it needs to be of high quality — there's no point in using a visual design tool to generate code if the code is of poor quality."*

For the scope defined in 1.1 as well as the definitions of code quality defined in 2.6, this was definitely successful, both from an accessibility, readability, and maintenance standpoint. With that said, the designs generated weren't very complex, even though they were based on common layout patterns found in many apps. More complex designs would put more stress on having well thought out components to build with, and a carefully designed code generation system. The code generation in the UI Builder has made a solid foundation for future work by providing a method for both generating and formatting high quality React Native code, albeit in a very limited scope.

***Handling accessibility in the different platforms***
*"Different platforms have different approaches on how to handle accessibility. The visual design tool needs to be able to unify the differences in how accessibility is handled on each platform."*

Based on the heuristic evaluations, the accessibility support on each platform was on an acceptable level for the test cases defined in 7.5. Accessibility support heavily reliant on the developer properly implementing it – and the UI Builder has at least laid the foundation for creating accessible user interfaces.

***User Experience (UX) of the Interface Design Tool***
*"Using a visual design tool needs to be as painless as possible, something that can be a go-to tool for creating, for example, prototypes."*

Based on the user testing done during the development of the UI Builder, many parts of the implementation could be considered a good User Experience – mainly by combining existing concepts such as drag-and-drop and live reload into a unified experience. Beyond these features, the UI Builder has proven to be a good tool for creating prototypes quickly – something that a user can almost immediately test on three different platforms. This is something of value as well.

### Some final thoughts

As mentioned earlier, the UI Builder has barely scratched the surface. It has proven to be a useful tool and a good proof of concept, but to become an everyday tool for app and web development there's a lot more work to be made.

The work has shown how different, already existing, technologies can be combined to to rethink how to approach app and web development, as well as providing potential solutions to several problems related to both cross-platform development and visual design tools.

# References


[1] Write once, run anywhere?, 2002. URL https://www.computerweekly.com/feature/Write-once-run-anywhere. [Last Accessed: May 4, 2019].

[2] Facebook. React Native - A framework for building native apps using React. *React Native Blog*, n.d. URL https://facebook.github.io/react-native/. [Last Accessed: April 18, 2019].

[3] Expo, n.d. URL https://expo.io/features.

[4] Nicolas Gallagher. Nicolas Gallagher - Twitter Lite, React Native, and Progressive Web Apps, Aug 2017. URL https://www.youtube.com/watch?v=tFFn39lLO-U. [Timestamp: 01:25 - 01:40] [Last Accessed: April 18, 2019].

[5] Shawn Lawton Henry, Shadi Abou-Zahra, and Judy Brewer. The Role of Accessibility in a Universal Web. In *Proceedings of the 11th Web for All Conference*, W4A '14, pages 17:1–17:4, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2651-3. doi: 10.1145/2596695.2596719. URL http://doi.acm.org/10.1145/2596695.2596719.

[6] Apple. Vision Accessibility - iPhone. n.d. URL https://www.apple.com/accessibility/iphone/vision/. [Last Accessed: April 18, 2019].

[7] Test your app's accessibility. *Android Developers*, n.d. URL https://developer.android.com/guide/topics/ui/accessibility/testing. [Last Accessed: April 18, 2019].

[8] Introducing ChromeVox, n.d. URL https://www.chromevox.com/. [Last Accessed: April 18, 2019].

[9] Apple. Vision Accessibility - Mac. n.d. URL https://www.apple.com/accessibility/mac/vision/. [Last Accessed: April 18, 2019].

[10] Accessibility - React Native. *React Native Blog*, n.d. URL https://facebook.github.io/react-native/docs/accessibility. [Last Accessed: April 18, 2019].

[11] W3C WAI. Web Content Accessibility Guidelines (WCAG) Overview, n.d. URL https://www.w3.org/WAI/standards-guidelines/wcag/. [Last Accessed: April 18, 2019].

[12] W3C WAI. WAI-ARIA Overview, n.d. URL https://www.w3.org/WAI/standards-guidelines/aria/. [Last Accessed: April 18, 2019].

[13] Nicolas Gallagher. necolas/react-native-web, n.d. URL https://github.com/necolas/react-native-web/blob/master/docs/guides/accessibility.md. [Last Accessed: May 24, 2019].

[14] CISQ. Code Quality and Related Standards. n.d. URL https://it-cisq.org/standards/. [Last Accessed: April 18, 2019].

[15] Google Lighthouse. *Google Developers*, n.d. URL https://developers.google.com/web/tools/lighthouse/. [Last Accessed: April 18, 2019].

[16] Prettier – Opinionated Code Formatter. *Prettier*, n.d. URL https://prettier.io/. [Last Accessed: April 18, 2019].

[17] ISO. *Ergonomics of human system interaction-Part 210: Human-centred design for interactive systems*. International Organization for Standardization ISO 9241-210:2010, 2010.

[18] Jenny Preece, Yvonne Rogers, and Helen Sharp. *Interaction Design: Beyond Human-Computer Interaction*. J. Wiley & Sons, 2002.

[19] Assistant Secretary for Public Affairs. Running a Usability Test. *Usability.gov*, May 2014. URL https://www.usability.gov/how-to-and-tools/methods/running-usability-tests.html.

[20] Assistant Secretary for Public Affairs. System Usability Scale (SUS). *Usability.gov*, Sep 2013. URL https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html. [Last Accessed: May 24, 2019].

[21] Dorothy Watson. Correcting for Acquiescent Response Bias in the Absence of a Balanced Scale: An Application to Class Consciousness. *Sociological Methods & Research*, 21(1):55–88, 1992. doi: 10.1177/0049124192021001003. URL https://doi.org/10.1177/0049124192021001003.

[22] Nathan Thomas. How To Use The System Usability Scale (SUS) To Evaluate The Usability Of Your Website, Jan 2019. URL https://usabilitygeek.com/how-to-use-the-system-usability-scale-sus-to-evaluate-the-usability-of-your-website/. [Last Accessed: May 18, 2019].

[23] Assistant Secretary for Public Affairs. Heuristic Evaluations and Expert Reviews. *Usability.gov*, Oct 2013. URL https://www.usability.gov/how-to-and-tools/methods/heuristic-evaluation.html. [Last Accessed: May 24, 2019].

[24] Google. Material Design, n.d. URL https://material.io/design/. [Last Accessed: May 24, 2019].

[25] Facebook. Typechecking With PropTypes, n.d. URL https://reactjs.org/docs/typechecking-with-proptypes.html. [Last Accessed: May 20, 2019].

[26] Facebook. JSX In Depth, n.d. URL https://reactjs.org/docs/jsx-in-depth.html#children-in-jsx. [Last Accessed: May 20, 2019].

# Appendices

# A User Tests

## A.1 Participant 1

Participant 1 is a 3rd year Computer Science student with some programming experience and limited experience when it comes to UI design.

### A.1.1 Survey Results

Table 5: Participant 1's SUS results.

| **Question No.** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
|---|---|---|---|---|---|---|---|---|---|---|
| **Score** | 3 | 2 | 4 | 1 | 4 | 2 | 5 | 2 | 5 | 3 |

**Did the live reloading functionality (instant feedback on the phone/browser) aid your design process in any way?**

Yes, it was nice to have visual feedback available easily to make sure that everything works.

**Were there any parts of the application that you feel could be improved?**

Maybe there could be a short tutorial on how to use the various components and what can be merged with what.

**Other feedback, if any.**

None.

### A.1.2 Test Results

Time taken for completing the login screen was 7 minutes and 36 seconds.
Time taken for completing the card based feed was 10 minutes and 47 seconds.

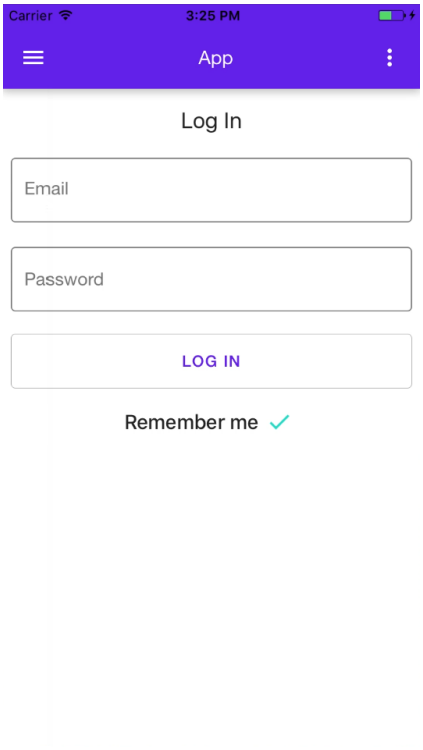



Figure 40: Participant 1's completed login screen, shown on an iOS simulator.

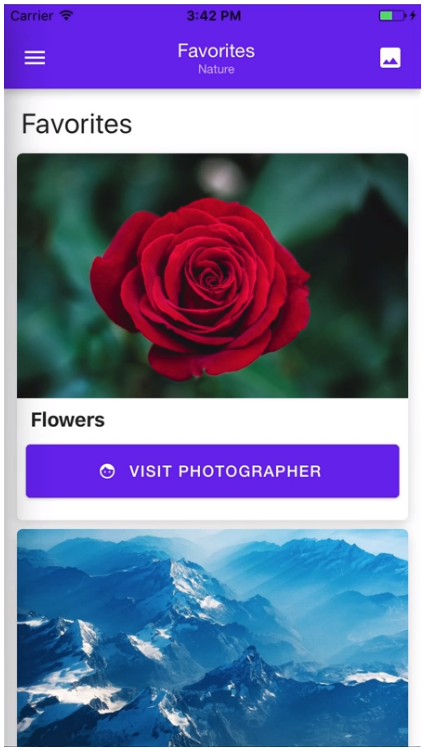



Figure 41: Participant 1's completed card based feed, shown on an iOS simulator.

## A.2 Participant 2

Participant 2 has several years of web development experience, leaning more towards backend development. Participant 2 has domain specific knowledge (React) and has experience in UI/UX design.

### A.2.1 Survey Results

Table 6: Participant 2's SUS results.

| **Question No.** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
|---|---|---|---|---|---|---|---|---|---|---|
| **Score** | 4 | 1 | 4 | 1 | 4 | 2 | 5 | 1 | 5 | 3 |

**Did the live reloading functionality (instant feedback on the phone/browser) aid your design process in any way?**

Didn't use it much until the end. So no, the view in the app was sufficient for the design.

**Were there any parts of the application that you feel could be improved?**

There was some knowledge you needed before hand. This could probably be integrated in the application.

**Other feedback, if any.**

The application felt smooth and worked great. Encountered a weird bug but other than that it was fine. It was quite easy to use.

### A.2.2 Test Results

Time taken for completing the login screen was 3 minutes and 5 seconds.
Time taken for completing the card based feed was 4 minutes and 40 seconds.

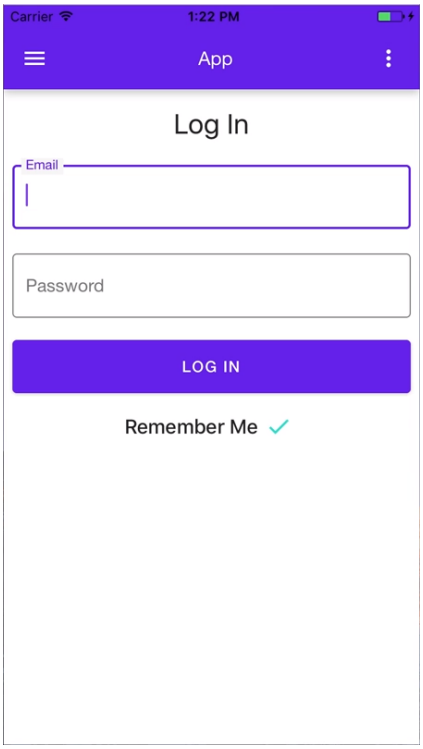



Figure 42: Participant 2's completed login screen, shown on an iOS simulator.

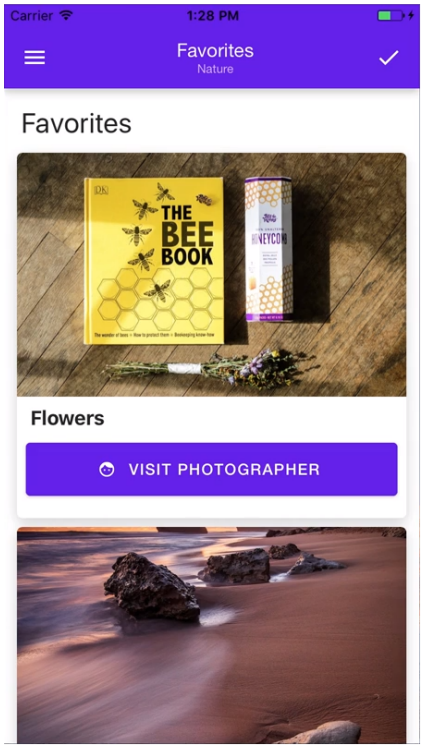



Figure 43: Participant 2's completed card based feed, shown on an iOS simulator.

## A.3 Participant 3

Participant 3 is a 4th year Computer Science student with some programming experience and limited experience when it comes to UI design.

### A.3.1 Survey Results

Table 7: Participant 3's SUS results.

| **Question No.** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
|---|---|---|---|---|---|---|---|---|---|---|
| **Score** | 4 | 1 | 5 | 2 | 5 | 1 | 5 | 1 | 4 | 1 |

**Did the live reloading functionality (instant feedback on the phone/browser) aid your design process in any way?**

Definitely.

**Were there any parts of the application that you feel could be improved?**

Some, but only in the addition of more features to expand what is possible to create.

**Other feedback, if any.**

Copy-pasting of components would be handy, as well as other features from more advanced visualizers. Perhaps a closer connection to the logic implementation would be useful, as the current application excels at UI components, but not in connecting these components with logic.

### A.3.2 Test Results

Time taken for completing the login screen was 9 minutes and 28 seconds.
Time taken for completing the card based feed was 8 minutes and 26 seconds.

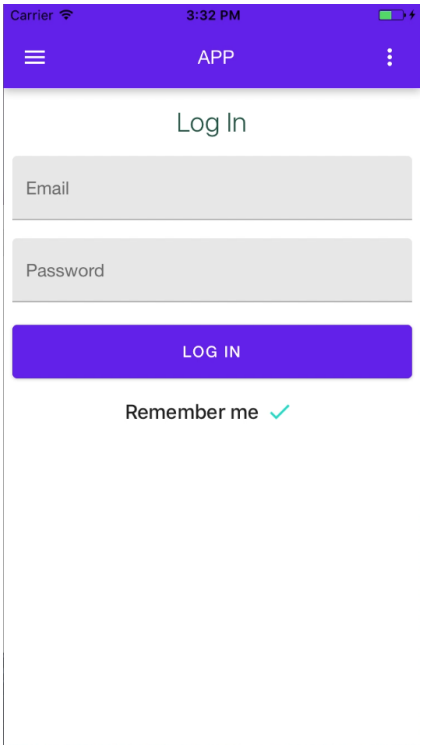



Figure 44: Participant 3's completed login screen, shown on an iOS simulator.

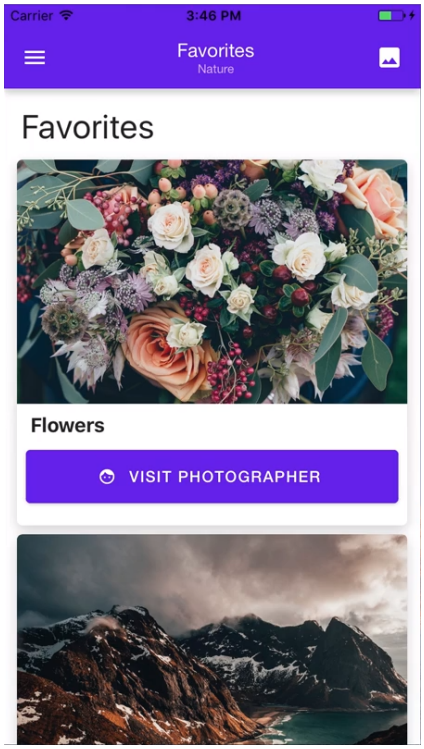



Figure 45: Participant 3's completed card based feed, shown on an iOS simulator.

## A.4 Participant 4

Participant 4 is a 5th year Computer Science student with both domain specific knowledge (React) & experienced in UI/UX design.

### A.4.1 Survey Results

Table 8: Participant 4's SUS results.

| **Question No.** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
|---|---|---|---|---|---|---|---|---|---|---|
| **Score** | 5 | 1 | 5 | 1 | 5 | 2 | 3 | 1 | 5 | 2 |

**Did the live reloading functionality (instant feedback on the phone/browser) aid your design process in any way?**

Det var trevligt att man kan se det man gjort direkt. Man ser ändå direkt i redigeringsprogrammet vad man har gjort så just för design processen var det inte helt nödvändigt. Men jag hade inte velat ta bort bort det då det ändå är bra att kunna se att något faktiskt händer sen.

**Were there any parts of the application that you feel could be improved?**

Jag hade velat att den komponenten man valt att skapa(alltså när jag tyckt på add button) så borde den "markeras" direkt så att jag direkt kan editera den utan att behöva klicka på den först.

**Other feedback, if any.**

Det är en kul idé och jag gillar att det är ett prototypprogram som är väldigt kopplat till den sortens komponenter man kommer att använda sen när man ska utveckla appen. Det är mycket bättre än att först göra den i t.ex. Sketch och sen ändå behöva ändra massa saker för att allt man ritar inte alltid går att översätta till t.ex. React eller vad man nu använder.

### A.4.2 Test Results

Time taken for completing the login screen was 7 minutes and 9 seconds.
Time taken for completing the card based feed was 7 minutes and 25 seconds.

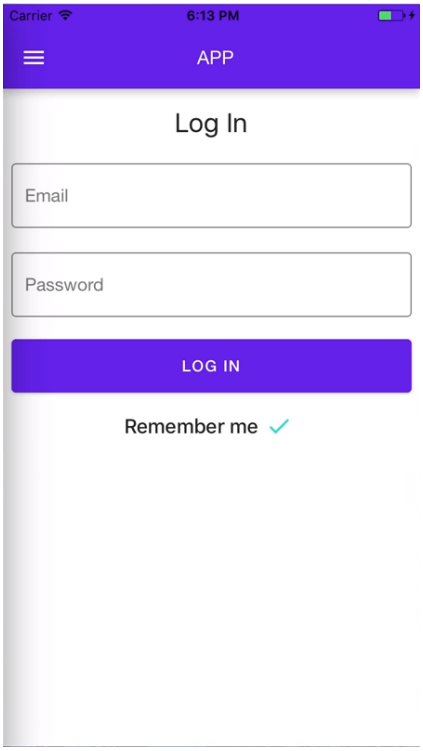



Figure 46: Participant 4's completed login screen, shown on an iOS simulator.

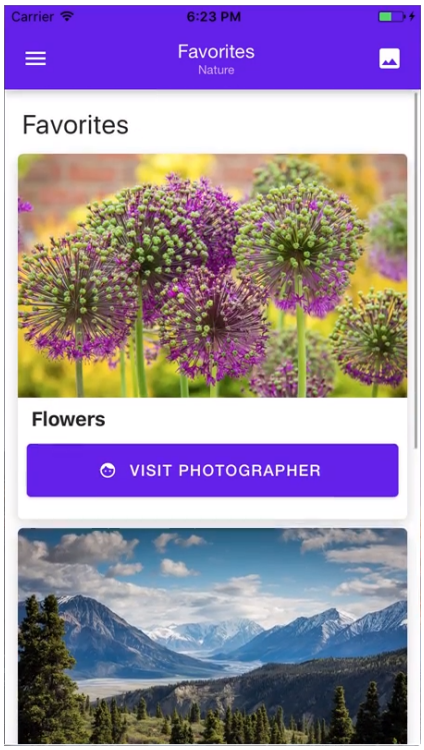



Figure 47: Participant 4's completed card based feed, shown on an iOS simulator.

## A.5 Participant 5

Participant 4 is a 5th year Computer Science student with both domain specific knowledge (React) & experienced in UI/UX design.

### A.5.1 Survey Results

Table 9: Participant 5's SUS results.

| **Question No.** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
|---|---|---|---|---|---|---|---|---|---|---|
| **Score** | 2 | 1 | 2 | 1 | 4 | 4 | 5 | 2 | 3 | 1 |

**Did the live reloading functionality (instant feedback on the phone/browser) aid your design process in any way?**

Yes, it felt nice.

**Were there any parts of the application that you feel could be improved?**

Flexibility and "traps", like the pink delete button and some select/drag-and-drop functions.

**Other feedback, if any.**

Nice job!

### A.5.2 Test Results

Time taken for completing the login screen was 5 minutes and 57 seconds.
Time taken for completing the card based feed was 5 minutes and 52 seconds.

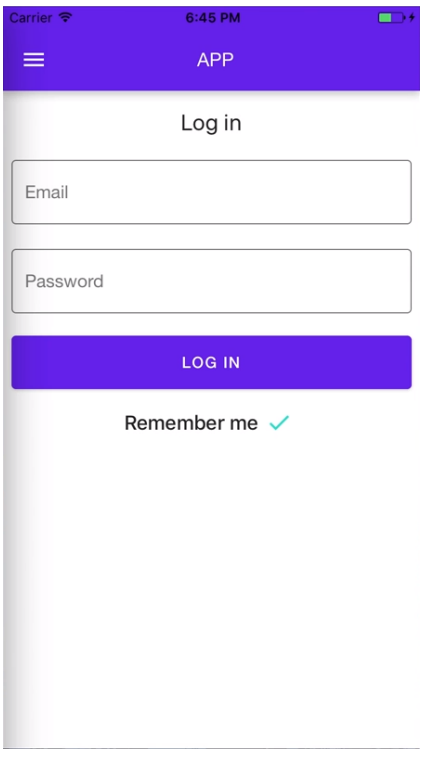



Figure 48: Participant 5's completed login screen, shown on an iOS simulator.

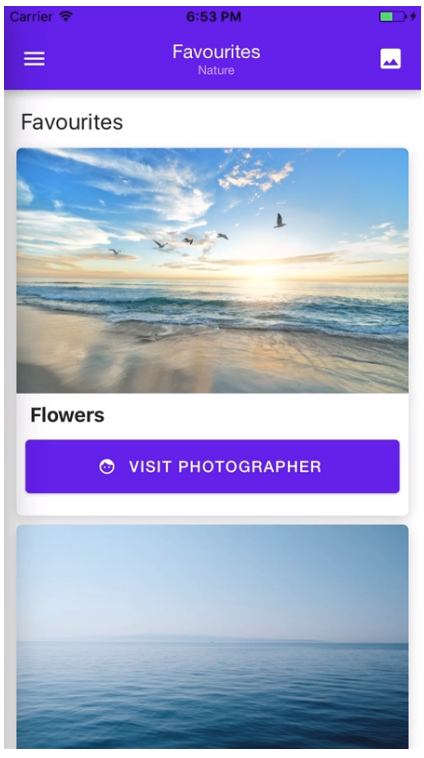



Figure 49: Participant 5's completed card based feed, shown on an iOS simulator.

# B Generated Code Evaluation

## B.1 Expert Answers

**Is the code readable and easy to follow?**

Yes. It looks like it's been processed by Prettier (and probably has?). This looks something like i would write myself, not something generated.

**Does the code follow general best practices?**

Yes. As I said, it looks like something I would write myself & processed via Prettier.

**How many changes would you need to do in order to be able to use the code?**

With the coming of [React] hooks, I would probably make the functions actual blocks with a return statement, just to avoid having to do the rewrite later down the road. That said, if I saw this in a review I would not comment on it, it is extremely nitpicky. And obviously, if you are going to customize it by e.g. adding theming or internationalization I would have to do some modifications, but only slight ones and it is to be expected (meaning that you would have to do it with non-generated code as well).

Minor changes, such as removing the `password={false}` in the username input and just type `password` instead of `password={true}` could also be done.

**Is the code of high enough quality to be usable as a foundation?**

Definitely. I'd feel confident using this off the bat without any modifications.

## B.2 Code Sample 1

```
import React from 'react'
import { View, ScrollView, StyleSheet } from 'react-native'

import Header from './components/Header/'
import Input from './components/Input/'
import Button from './components/Button/'

const styles = StyleSheet.create({
  container: {
    flex: 1,
  },

  scrollContentContainer: {
    padding: 8,
  },
})

const App = () => (
  <View style={styles.container}>
    <Header
      title="Title"
      subtitle="Subtitle"
      leftIcon="menu"
      rightIcon="more-vert"
      placement="center"
      backgroundColor="#6200ee"
      foregroundColor="white"
    />
    <ScrollView contentContainerStyle={styles.scrollContentContainer}>
      <Input placeholder="email@example.com" mode="outlined" label="
          Email" password={false} />
      <Input placeholder="Password" mode="outlined" label="Password"
          password={true} />
      <Button title="Log In" color="#6200ee" type="contained" />
    </ScrollView>
  </View>
)

export default App
```

Listing 3: Code sample used for the heuristic evaluation.

## B.3 Code Sample 2

```
import React from 'react'
import { View, ScrollView, StyleSheet } from 'react-native'

import Header from './components/Header/'
import Text from './components/Text/'
import Card from './components/Card/'
import Button from './components/Button/'

const styles = StyleSheet.create({
  container: {
    flex: 1,
  },

  scrollContentContainer: {
    padding: 8,
  },
})

const App = () => (
  <View style={styles.container}>
    <Header
      title="Favourites"
      leftIcon="menu"
      rightIcon="image"
      placement="center"
      backgroundColor="darksalmon"
      foregroundColor="white"
    />
    <ScrollView contentContainerStyle={styles.scrollContentContainer}>
      <Text color="crimson" fontSize="24" textAlign="left" fontWeight="
          bold">
        Favourite Pictures
      </Text>
      <Card title="Trees" image="https://picsum.photos/200/300" />
      <Card title="Plants" image="https://picsum.photos/200/300" />
      <Card title="Flowers" image="https://picsum.photos/200/300" />
      <Button title="See More" color="crimson" type="outlined" />
    </ScrollView>
  </View>
)

export default App
```

Listing 4: Code sample used for the heuristic evaluation.

# C Source Code

## C.1 UI Builder

The entirety of the UI Builder's source code is hosted on GitHub, and is released under an MIT license. It can be found at `https://github.com/edvinh/ui-builder`.

## C.2 Generated Code Boilerplate

The custom boilerplate for the generated code is hosted on GitHub as well, and can be found at `https://github.com/edvinh/react-native-web-expo-boilerplate`.