Nguyen Nguyen

Improving the accessibility of REGames mobile application in React Native

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor’s Thesis

1 December 2021

Abstract

Author:	Nguyen Nguyen
Title:	Improving the accessibility of REGames mobile application in React Native
Number of Pages:	51 pages + 0 appendices
Date:	1 December 2021
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Mobile Solutions
Supervisors:	Hannu Alakangas, Project Manager Patrick Ausderau, Senior Lecturer

The purpose of this thesis was to improve the accessibility of the REGames application built using React Native technology, and to list several good practices in integrating accessibility in mobile development. A brief introduction of React and React Native was also discussed in the thesis. Additionally, the study brought up some other aspects of the project, such as technology stack, tools, and code organization.

Previously, Fitseven Oy, which is the owner of the software, had had a contract with Punos Mobile Oy to build the application using React Native. At the time of implementing the thesis project, the involved parties desired to raise the accessibility standard of the application. The implementation of this thesis project, which included the consolidation of the application's accessibility, was a part in a bigger development of the whole application carried out by Punos Mobile Oy. The research and implementation of this study was carried out using a vast amount of official documentation, guidelines, and e-books. Topics which are related to the project, such as React Native, accessibility, and design, are all covered in those sources.

By applying those theoretical resources into practice, the status of the application was analyzed, and its inclusive aspect was bettered. As a result, noticeable enhancements in accessibility, such as the zoomable image component, which allows users zooming in and out important highly detailed images, were made on the application. However, as accessibility is an enormous field, numerous things could be improved further. For example, when exiting the view of the zoomable image component, the focus of the assistive technology should point back to the original image component instead of on the top most component in the view as it is now. Such features will enrich the user experience of the application, making it more accessible.

Keywords: Accessibility, UI/UX design, mobile, React Native, frontend

Contents

List of Abbreviations

API: Application Programming Interface.

HTML: HyperText Markup Language.

iOS: iPhone Operating System.

JSON: JavaScript Object Notation.

MVP: minimal viable product.

OS: operating system.

W3C: World Wide Web Consortium.

WCAG: Web Content Accessibility Guidelines.

XML: Extensible Markup Language.

1 Introduction

Fitseven Oy owns a business model built on top of escape game, in which the users, or to suit the context, players, will try to explore rooms with many quizzes, and to proceed and win the game. The players have to answer the quizzes based on the hints associated with each quiz. The application acts as a medium between the users and the core game, helping the user choose the room they prefer based on difficulty, retrieve the quizzes and hints, answer the quizzes, and track their progress and statistics, such as elapsed time or total score.

The application was developed by Punos Mobile Oy using React Native. The technology uses the syntax of React, which is a web technology, to build applications on multiple platforms, including web, mobile, and desktop. The company finished the application's main functionality at the end of 2020. The European Accessibility Act being enforced by Finnish legislation on the latest of the 23rd of June of 2020 for mobile applications [1] gave the company a chance to offer Fitseven Oy an improvement in the accessibility of the application.

Among top countries with the highest population ageing rate [2], people who are 65 years old and older account for 19.9% of Finnish population [3]. Despite not being as active in using mobile technology as younger generation, or may meet difficulties in interacting with digital devices due to their affection, the elder still benefit greatly from what the technology could provide. Not to mention those whose parts of functionalities are impaired, or people with normal constitution but interacting with the devices in disadvantaged condition. With the availability of accessibility technologies, they are now able to interact with the world around more easily.

The purpose of the thesis is to enhance accessibility of the REGames mobile application. Besides, essential understanding of the accessibility concept, as well as several good practices of building an application using React Native will also be covered, including accessibility-wise practices and code organization.

2 Current State Analysis

Since 2020, accessibility has become a notable topic in Finnish software development industry due to the enforcement from the government in 2019 [1]. The maturity of accessible technology in accompanied with the robust enforcement and regulatory actions of accessibility in digital applications and services paves the way for massive integration of accessible features and options not only into the existing applications and services, but also into all the ones released afterwards. As similar to most of organizations which are not specialized in accessibility, Fitseven Oy wishes for maximizing the inclusion of REGames mobile application within their reachability. Together with Punos Mobile Oy, the two companies agreed on a target of qualifying the AA compliance level with Web Content Accessibility Guidelines (WCAG) 2.0 (see 3.2.1). Additionally, Punos Mobile Oy would also try to improve the application's accessibility based on the accessibility guidelines of Apple for iPhone Operating System (iOS), and of Google for Android, as much as what the company could deliver.

Originally, the development of the application was conducted under well supervision in terms of accessibility. Several inclusive user experience and user interface studies were made at that time to ensure high accessibility quality of the software right at the beginning. However, later, the application was developed with minimal consideration of inclusion. Instead, the development team decided to accelerate the developing rate by focusing on delivering a minimal viable product (MVP) with only core features being functional. And at that time, accessibility was not in the priority list. This was quite common and understandable at that moment since the enforcement of the accessibility law and the inclusion-wise development process was still new. That is particularly true for the case of projects of start-ups largely due to the cost optimizing reason. Fortunately, after the hectic period of developing the MVP, both parties came to a consensus of implementing accessibility features as much as both parties could afford. Although integrating inclusive components in an existing non-fully-accessible application is a tedious and time-consuming job, the act of providing those accessibility supports in the

application should always be appreciated.

3 Essential Concepts

This chapter will cover crucial concepts which act as the theoretical framework of the thesis. They are the idea of accessibility and the core technology stack of the project. As the document focuses on the inclusive aspects of a mobile application, the accessibility concept will be elaborated in more details compared with the choice of technology.

3.1 Accessibility

The term **mobile application accessibility** means the extent to which a diverse human population could use a mobile application. Therefore, it does not restrict to elderly people or those with impairments, but includes users who may be in an environment which keeps them from fully obtaining the benefits of the application, such as those who have to use their phone by one hand while the other hand taking care of their baby. The following subsection will present the main kinds of impairment and environmental obstacle people usually incur when consuming digital content [4].

3.1.1 Impairments

In the book Accessibility for Everyone, the author, Laura Kalbag, suggests five major categories of impairments preventing users from adequately perceiving the content of a website [4], and, equivalently, in obtaining the information from a mobile application.

Visual impairments are ones that affect the ability of seeing. Those impairments are classified into two sub-types, color blindness and eyesight loss. And in those two, even smaller classification is made to understand specifically how those disadvantages affect the visibility and then to provide meticulous solutions. The visual demonstration of those color blindness syndromes can be found in Figure 1.

Figure 1: A kingfisher in its original photo, then duplicated (left to right) with simulated four sub-types of colour blindness, which are, respectively, deuteranopia, protanopia, tritanopia and monochromacy (Copied from Kalbag, Laura [4])

For eyesight loss syndromes, Figure 2 provides a graphical illustration of how it looks like. People who experience visual impairments may benefit from written forms of the visual content they consume and designs which do not rely on colour as the sole meaning expression, such as alternative text for images, transcript for videos, or buttons with text and icons and distinguishing colours.

Figure 2: That kingfisher again, this time simulating (left to right) macular degeneration, glaucoma, and diabetic retinopathy (Copied from the book Accessibility for Everyone [4])

Auditory impairments are ones that affects the ability of hearing. The decline in aural faculty falls into two groups: conductive hearing loss—when sound transmission is disrupted along its way toward the inner ear due to foreign objects or idiosyncrasy in the ear structure—and sensorineural hearing loss—when the nerves in the ear are damaged causing distortion of sounds and thereby difficulty in speech comprehension.

Motor impairments are ones that affect the physical interaction with things. Those disabilities may range from “cerebral palsy which affects muscle control and movement”, “neural-tube defects which cause weakness, paralysis, and abnormal eye movement”, to “muscle and joint conditions like muscular dystrophy, arthritis, and Ehlers-Danlos syndrome which cause pain and difficulty moving” [4]. The symptoms of the motor impairments pose hardship in basic interactions with, in the scope of the thesis, mobile devices. Therefore, developers should expect and handle the struggle in activities like text inputting or navigating in the device’s system using touch screen in their applications.

Cognitive impairments - ones that affect the ability of understanding something. Sharing the attribute of huge diversity in symptoms with the motor impairment, the incompetence in perception affects the way users memorize content, task and in-app navigation state, or how people pay attention to a certain amount of information in an application, or the capability of processing features and content in an application. Besides innate abnormalities in cognition, low literacy is also considered as a reasonable challenge to get users absorb information comfortably, with the number of unlettered adults approximates to 13.422% of the world population in 2019 [5]. To mitigate the disadvantages caused by those impairments, many solutions are suggested—simplifying the content hierarchy as much as possible, using plain language that helps non-native people understand the context and content effortlessly, or conveying the meaning of a feature in multiple forms, for example, an **Accessibility** button with the corresponding text label and a human figure icon and the tint color of purple, as can be seen in Figure 3.

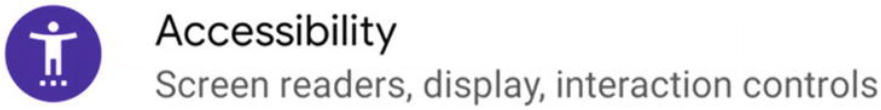

Figure 3: Android combines color, shape, text, and an image to convey meaning (Copied from Whitaker, Rob [6])

Vestibular disorders and seizures are ones that cause discomfort, for instance,

dizziness or vertigo, when a person sees on-screen motions, flashing or flickering light, and several other triggering factors. Users having those syndromes greatly benefit from the option of limiting the frequency of triggering factors, or completely eliminating the triggering factor from the design of an application. Fortunately, those options are usually provided in the global settings of most mobile operating systems nowadays. For example, on iOS, users could turn off animation effects globally on their device through an option called Reduce Motion. On Android, things are slightly more complicated, as users have to go to Developer Options and turn off each animation effect options.

3.1.2 Environmental Factors

Beside environmental factors which pose direct impact on physical interactions of the user to the mobile device and could be attenuated mainly by enhance in hardware, such as weather events like rain, storm, or extreme temperature conditions, this subsection will mostly cover several subtle elements which plays a vital role in user experience in a mobile application [4].

Regarding **device configuration**, at the moment, a multitude of options of mobile device model are available for users to choose. They come from many manufacturers with different configurations, such as screen size, pixel density, screen orientation, sensors, processing capacity. That means responsive design, and sometimes, a different version of an application, is necessary for maximizing the amount of users from varying backgrounds. In mobile application development industry, the term **responsive design** is often understood as a design which could work in different screen sizes and orientation. Therefore, responsive design is one of many cornerstones in mobile accessibility.

In spite of the technology industry enjoying a stable and high-speed **Internet connectivity** in most countries, unreliable connection still exists not only in developing countries but also in developed ones, even in urban areas. For example, in a corner of several coffee shops in downtown Helsinki—the capital of Finland—the internet connection via WiFi or cellular may be still slow due to the

blockage of the wall. The careful consideration of making an application be able to work with low bandwidth, or better to work offline, should be embraced in the application design process, to a degree fitting the nature of the application.

Regarding **languages**, the number of people who do not speak the official languages of the country they are living on vary among countries. According to the data of 2020, in Finland, 7.8% of the population used languages other than Finnish and Swedish [3], which are two official languages of Finland. Additionally, in Finnish society, the fact of having two official languages requires digital applications and services in the country support the two languages, and some of them even support English. In the case of international services, for example, on the operating system (OS) of Apple's mobile devices, iOS, many different languages are supported, including those with different characters and reading directions like English versus Arabic. If developers would like to provide such a wide support in languages, they should take care of the fonts to ensure that all characters are rendered without error. Lastly, simplicity in using languages allows the application or service to be understood by as many people as possible.

3.2 Accessibility Guidelines and Standards

Accessibility is a substantial field which covers numerous aspects in product design and system design. In the mobile software development, a handful of standards and guidelines exist to help designers and developers evaluate the accessibility of their applications. They assume a prominent place as accessibility auditing points throughout the process of building and maintaining the software.

Accessibility guidelines could be considered as a list of checkboxes of good practices to follow when implementing accessible features in an application. While this may sound rigid since accessibility, in the end, is for improving human experience in using a tool, it provides a solid framework which has been researched and verified by a great deal of studies and test-to-fail procedures. In addition, it saves resources for projects with limited funding or a narrow base of users, especially in initial stages. Guidelines which have been referenced in this

thesis will be described in this section.

3.2.1 Web Content Accessibility Guidelines (WCAG)

According to the World Wide Web Consortium (W3C) website [7]:

> WCAG is developed through the W3C process in cooperation with individuals and organizations around the world, with a goal of providing a single shared standard for web content accessibility that meets the needs of individuals, organizations, and governments internationally.

The name of the guideline may confusingly suggest that it is only appropriate for web usage. However, this is not true. The name originated in the period of website proliferation from 1990s to 2000s, almost a decade before the thriving of mobile application like it is nowadays. A great deal of criteria in the guideline are used as benchmarks of accessibility in mobile software development [4].

WCAG is separated into four principles. **Perceivable** principle refers to “Information and user interface components must be presentable to users in ways they can perceive” [8]. In other words, developers should implement multiple alternatives of presenting the content as well as the navigating and controlling features in an application. For instance, besides using text, icon and color to present an interactive element like in Figure 3, the element should be developed in a way that allows assistive technology, such as VoiceOver in iOS or TalkBack in Android, to reading out its content and meaning. In this case, the assistive software should read ”Navigates to Accessibility settings”.

Operable principle says “User interface components and navigation must be operable” [8]. This means that an application must facilitate its usage for a wide range of users as much as possible. For instance, an application conforming to these principles should allow users navigating using external keyboards, and ensure no existence of keyboard traps—components which are focusable using the keyboard, but users cannot get out of them using keyboard.

Understandable principle suggests “Information and the operation of user interface must be understandable” [8]. As the name suggests, this principle requires the content, design, and the behaviour of an application to be clear to its users. One of the factors contributing to intelligibility of an application is the predictability of the application’s features. An application should maintain consistency in design operation. For example, if a button with a destructive function, such as a Sign Out button, is colored red, all buttons with destructive purposes should be colored the same, and no button with non-destructive purposes should be colored red throughout the application. In comparison to the first principle, the term **understandable** means that the application should be able to make users understand the content so that they could interact with it, while **perceivable** aims to make the users capable of accessing the data.

Robust principle says “Content must be robust enough that it can be interpreted reliably by a wide variety of user agents, including assistive technology” [8]. This principle suggests that an application should work seamlessly with any assistive technology available on a certain platform.

At the time of this thesis, the latest official version of WCAG is 2.1. However, the thesis would refer to the version 2.0 of WCAG because all parties participating in this thesis project agreed to use that version based on the fact that they were more confident of their knowledge about it.

To meet the needs of various developer groups and different contexts, the WCAG 2.0 guidelines consist of multiple levels of conformance. A series of criteria to fulfill in order to comply with WCAG 2.0 could be found in the quick reference of WCAG 2.0 in W3C website [9]. Each criterion is mapped to a level of conformance.

Level A is the minimum level. This level usually consists of basic criteria for those applications which freshly begin developing accessibility features. For example, according to the **Perceivable** principle, to reach level A, prerecorded audios should be attached with captions [10].

Level AA is the intermediate level. To reach this level, an application has to meet

level A and level AA criteria. Most businesses and organizations concerned with accessibility should target this level. For instance, based on the **Perceivable** principle, to reach level AA, live audio content should be attached with captions [11].

Level AAA is the optimum level. To reach this level, an application has to meet level A, level AA, and level AAA criteria. Those institutions which work in the field of accessibility should be interested in this level, as it requires a considerable amount of resources to meet its high-demanding criteria. For example, in agreement with the **Perceivable** principle, to reach level AAA, prerecorded audio content should be provided together with sign language [12].

3.2.2 Android, iOS, and React Native Accessibility Guidelines

Because the REGames application is built using React Native targeting Android and iOS platforms, the accessibility guidelines in both platforms are critical resources in improving the accessibility of the application. These guidelines provide more platform-specific details than the WCAG 2.0 does. A full list of guidelines and best practices of accessibility could be found in Android Developers website for Android [13], or Apple Developers website for iOS [14].

In addition to the importance of the two native accessibility guidelines, the ones of React Native are also a key reference in the thesis. Acting as the core technology in the project, almost all of the actual works are implemented in React Native. The React Native accessibility guidelines is actually a documentation of accessibility Application Programming Interface (API) mapped from React Native to native platforms. However, the guidelines do suggest relevant contexts to use the API appropriately. A detailed version of the guidelines could be found at React Native official website [15].

Although the thesis will not cover those full lists in detail, a couple of points which are not mentioned in the thesis could be named here as examples. For instance, the `accessibilityIgnoresInvertColors` property of a view component in React

Native is used to immunize the look of the view against Invert Colors mode on iOS. Another example is the `accessibilityState` property, which is used to describe the state of a component, such as being disabled, or being selected, to the assistive technology. Later, the study will cover several guidelines and best practices of both platforms applied as real implementations in the thesis project.

3.3 Technology stack

This section will cover the technology used in the thesis project, which are React, React Native and Typescript. They are all related to Javascript, which is a prominent interpreted programming language in web technology [16; 17].

3.3.1 React and React Native

React is a Javascript library for building the user interface of a website. It allows splitting the complex gigantic user interface into smaller isolated parts called **components** [18]. React applications are written using a syntax extension to Javascript called JSX. The syntax essentially extends Javascript with Extensible Markup Language (XML). Because of the shallow learning curves and the great performance, the library becomes more and more popular in the web development industry. According to Stack Overflow's surveys in 2020 [19], React, under another name of React.js, ranked second in the list of the most popular web frameworks. Originally introduced in 2011 by Jordan Walke, a software engineer at Facebook, the library was officially released in 2013 and is now maintained by the same enterprise and a community of active contributors [20].

Two years after the launch of React, in 2015, Facebook released React Native, a Javascript framework built on top of React for developing native applications in mobile platforms, mainly for iOS and Android. With community-supported projects, React Native later is able to support many other platforms, such as Windows, macOS [21], or tvOS [22]. Thanks to that, those developers who could use React web technology could now build mobile applications in multiple platforms also, using only a single technology. This speeds up the development

time, and allows code reusing and knowledge sharing. Nevertheless, the thesis would focus only on the support of the library to iOS and Android. An example of a React Native view file using JSX syntax can be seen in Listing 1.

```
import React from 'react';
import { Text, View } from 'react-native';

const YourApp = () => {
  return (
    <View style={{ flex: 1, justifyContent: "center", alignItems:
        "center" }}>
      <Text>
        Try editing me!
      </Text>
    </View>
  );
}

export default YourApp;
```

Listing 1: An example of a React Native view file using JSX syntax [23]

The code in the Listing 1 demonstrate the definition of a simple view of a mobile application, with a text of *Try editing me!* aligned in the center of the screen. The JSX syntax inherited from React is firstly parsed into normal Javascript, then compiled into the platform's native code. After rendering the code above in a mobile device, a layout as illustrated in Figure 4 should appear on the screen of the device.

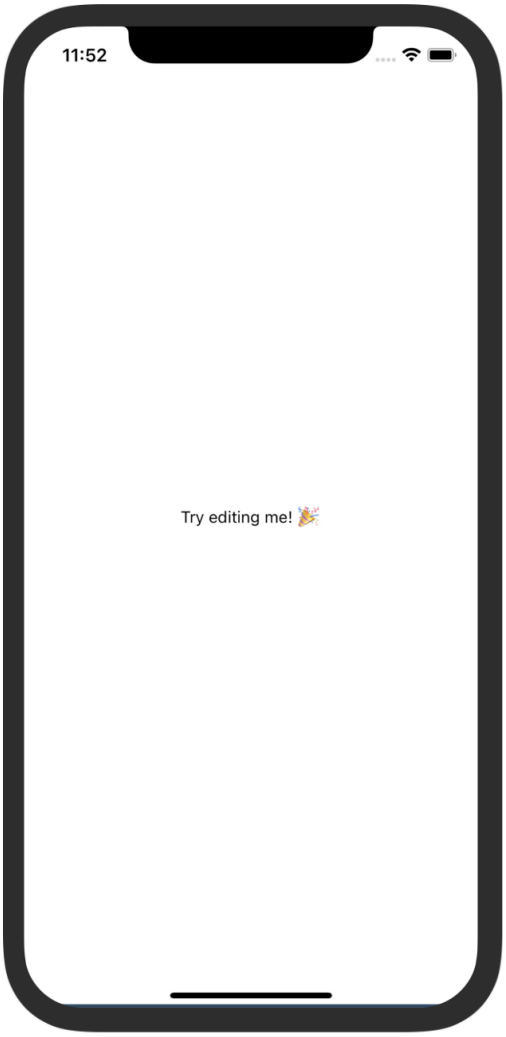

Figure 4: A view rendered on the example code in an iOS device [23]

Despite using the similar syntax of JSX compared to React, under the hood, React Native triggers the native rendering API of the host platform, ensuring native components with native-like experience in a mobile application. And it provides high performance [24]. This approach was totally novel in the hybrid software development field at that time, when using web view to render user interface components was a common practice. Additionally, due to the fact that React Native is Javascript, running a separate Javascript thread in a mobile application, during the development, developers do not have to rebuild the application to see changes reflected. Instead, the framework provides a feature called **live-reloading**, which allows automatically and manually updating changes in the application without actually reloading it in native threads. At first, this may not seem to be impressive, but adding up the time saved by the fast live-reloading, it

really helps minimize the developing time of an application [24].

3.3.2 Typescript

Typescript is an open-source superset of Javascript. This means that any feature available in Javascript is available in Typescript. On top of Javascript, Typescript adds additional syntax to catch errors early in the editor [25]. As the name suggests, Typescript brings code analysis to the underneath Javascript through static data type checking. Typescript code will be converted into Javascript, so it could run anywhere Javascript could with appropriate setup, from browser to Node.js environment [25]. The language is developed and maintained by Microsoft and a developer community.

The Typescript engine will perform the type checking during the compilation of a Typescript application. Thus, developers have to define the type of data before using them. This is diametrically different from Javascript, in which developers freely use any types of data without declaring the types in advance, but coming with a cost of having a notoriously high amount of potential bugs and difficulties in debugging.

In the Typescript world, the JSX syntax of React and React Native is still valid. The only difference is that instead of using the `.jsx` file format, now the extension of Typescript files is `.tsx`. An example of a React view file using Typescript can be found in Listing 2.

```
1  import * as React from "react";
2
3  interface UserThumbnailProps {
4    img: string;
5    alt: string;
6    url: string;
7  }
8
9  export const UserThumbnail = (props: UserThumbnailProps) =>
10   <a href={props.url}>
11     <img src={props.img} alt={props.alt} />
12   </a>
```

Listing 2: An example of a React view file using Typescript (Copied from Typescript's homepage [25])

In Punos Mobile, Typescript becomes a standard for building applications based on Javascript, including web applications, server applications, or mobile ones. The company always tries to integrate Typescript whenever possible in their projects. And REGames is not an exception. By using Typescript, the code base of the project is more readable, maintainable, and minimizes avoidable typing errors [26].

4 Practical Works

Essentially, REGames is an application which consumes text and image data as quizzes, and allows users to input their text answer. It also contains navigating interfaces, such as authentication, scrolling views of game selection, or a drawer component to toggle the display of other navigating options. Therefore, when designing accessible features, the developer team did not have to deal with narrating video contents and or reducing animation effects. Instead, the spotlight shifted to optimizing the clarity and perceptibility of text and image contents, and of the navigation system of the application. After referencing the accessibility guidelines, and comparing criteria with the status of existing features of the application, all of the involved parties agreed on a list of implementations as following.

- Ensure adequate color contrast
- Limit the dynamic size of text
- Enhance the compatibility of elements with VoiceOver and TalkBack
- Implement the zoomable hint image of quizzes
- Improve the accessibility of images

Except the first item of ensuring adequate color contrast, which had been implemented before the MVP period, all other items were done after the period. The thesis will cover each item in details in their corresponding separate section.

4.1 Ensure Adequate Color Contrast

Color contrast implies the discrepancy in light between a background and anything on top of it, i.e the foreground. A graphic user interface with the color of

foreground elements being too similar to the background's is difficult for users to distinguish the elements from the background. Sufficiently high color contrast ensures that a majority of people could fully perceive its visual content. [27.]

The quality of the color contrast is represented by on a numerical value named **color contrast ratio**. To calculate the value, one does a division of the relative luminance of the lighter color through the relative luminance of the darker color. The outcome of the division is demonstrated as a ratio whose value varies from 1:1, the lowest contrast, to 21:1, the highest contrast. [28.]

According to the criterion 1.4.3 of Distinguisable principles of WCAG 2.0 about the contrast [29] targeting level AA, text and images of text should be displayed with contrast ratio of at least 4.5:1, except for large scale text with a lower contrast ratio of 3:1, and for decorative text. Content text—for example, question text of a quiz, text of navigating elements—for example, the label of a button, or any text which is meaningful for users to interact with the application should sufficiently contrast against the background it stays on. In REGames case, a large amount of text is in white (hex code `#FFFFFF`) on backgrounds in contrasting colors, such as in black (hex code `#000000`, with contrast ratio of 21:1), or in multiple shades of red from the background image (hex code ranging from `#210000` of the darkest shade with the contrast ratio of 19.72:1, to `#8A021` of the lightest shade with the contrast ratio of 10:1). There is also text in non-white colors on the black background, such as that is in brownish yellow (hex code `#BEAA3C`, with contrast ratio of 9:1), in red (hex code `#F53727`, with contrast ratio of 5.46:1), or in green (hex code `#8EE187`, with contrast ratio of 13.27:1), as can be seen in Figure 5.

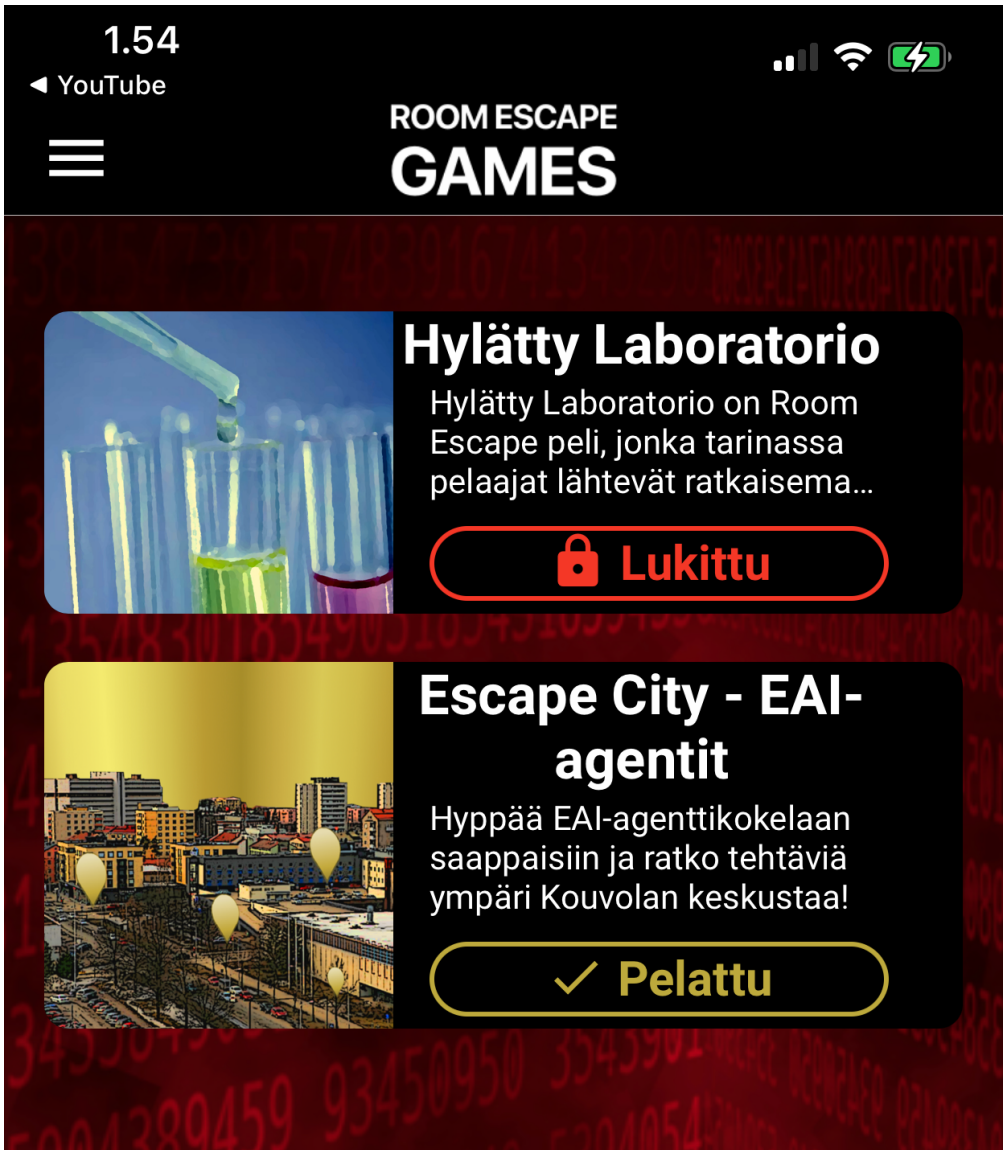

Figure 5: Text in different colors in the home screen of the REGames application

It is noticeable that in the Figure 5, its content also demonstrates a good example of using multiple means to convey the message of a component. As could be observed, the game status is expressed by three different ways, in text, such as *Lukittu*, *Pelattu*, *Pelaa* in Finnish, meaning *Locked*, *Already Played*, *Ready to Play* in English, respectively; by color, such as red for locked games, brownish yellow for already-played games, and green for ready-to-play games; and by icon, such as a locked lock for locked games, a tick for played games, and an unlocked lock for ready-to-play games.

The process of color selecting had been done in the design phase of the project. A multitude of tools which support checking color contrast and suggesting adequately contrast pair of colors are available in the market, both for free and payable. In this project, WebAIM tool[1] was used for checking color contrast, while Color Safe tool[2] was used for generating pairs of well contrasting colors. Those are all free online tools with features for accessibility and a friendly user interface. After the design phase, the color presets were used correctly in the code and were rendered as expected in the application compared to the design.

4.2 Limit The Dynamic Size of Text

Both Android and iOS support adjusting text size in throughout the OS. At the time of the thesis, users may find that feature in Accessibility option in the built-in Settings application of the corresponding OS. The feature allows users to change the text size globally in their device to fit their visibility, ranging from being very small at the low extreme to being huge at the high extreme. This is especially helpful for those who struggle with the default font size of the OS. For instance, the elder may prefer large text size usually due to the decrease in their accommodation of eye. However, most of the time, the high extreme of this feature goes beyond the responsiveness of all application layout, breaking the readability of the design.

To resolve this issue, the `Text` component of React Native exposes a property named `maxFontSizeMultiplier` to cap the maximum value of font size flexibility. To set the limit multiplier, developers should pass a number value greater than or equal `1` to the property. Those `Text` components with such a property will increase their font size to a certain predefined multiplier, then will stop growing even if users continue the the increment in global font size. The discrepancy between two different values of `maxFontSizeMultiplier` in the same `Text` component is demonstrated in Figure 6.

[1] https://webaim.org/resources/contrastchecker/
[2] http://colorsafe.co/

Figure 6: Font size flexibility is different between `Text` components with different `maxFontSizeMultiplier` values at maximum font size scaling in iOS. On the left, the value is not specified, while on the right, the value is set to `2`.

However, this leads to another couple of issues. Firstly, before the implementation of this thesis project, developers had used a common approach of importing the `Text` component from React Native framework and using it directly in the rendering of a parent component. The approach is totally fine for small projects or demo projects. Nevertheless, for medium and above projects, this is not ideal. Anytime developers want to make changes in some similar `Text` components, they have to scrutinize throughout the application for those resembling components scattering all around and edit them, one by one. This is laborious and error-prone. Dummy code lines of the usage of `Text` component in the previous implementation of the project is demonstrated in Listing 3, where the component is imported at line 2, and then is used at lines 11 and 12.

```
1  import React from "react";
2  import { StyleSheet, Text, View } from "react-native";
3
4  const styles = StyleSheet.create({
5      paragraph: { ... },
6      title: { ... },
```

```
});

export const ParentComponent = () => (
  <View>
    <Text style={styles.title}>Title</Text>
    <Text style={styles.paragraph}>Paragraph</Text>
  </View>
);
```

Listing 3: A dummy example of the usage of `Text` component in the project in the previous implementation

Secondly, different kinds of text may have different suitable values of `maxFontSizeMultiplier`. For instance, a quiz consists of a title and question text, and as the title is usually more succinct than the paragraph of question description, it could endure a larger flexibility in scaling font size. This is one of many factors which build the hierarchy of the content of an application. An application comprises a finite amount of text types with different priorities and specifications. Those differences form the content hierarchy of the application. We could observe the hierarchy from multiple perspectives. One of them is the atomic design, which is ”a mental model to help us think of our user interfaces as both a cohesive whole and a collection of parts at the same time” [30]. The model consists of five stages, which are, respectively from the most simple to the most complicated, atoms, molecules, organisms, templates, and lastly, pages. Atoms represent all base styles of the simplest components in an application. In the case of the thesis project, `Text`-based components could be considered as a type of atom. In general, the atomic design model shares several common understanding with the criterion 1.3.1 of Adaptable principles of WCAG 2.0 about the Information and Relationships [31] targeting level A.

Those two issues above could be resolved by creating customized atomic `Text`-based components, then using them all over the application in replacement of the plain built-in `Text` component from React Native. Several methods of implementing the tailored components are available, including purely using React Native basic components. However, considering those `Text`-based components are stateless, i.e. they do not have their own distinctive business logic but only logic helping render the layout, the introduction of a third-party library named

`styled-components` is appropriate here.

The package mainly bring benefits in two facets. Firstly, it allows CSS syntax in styling components. To apply decorating styles to components, React Native uses a model of objects with fields' name being similar with but not identical to CSS properties' name. For example, `backgroundColor` in React Native is equivalent to `background-color` in CSS. By using `styled-components` library, developers could use orthodox CSS syntax in Javascript or Typescript file for styling components. This is quite helpful, given that React Native applications are written on React syntax, which is a web technology.

Secondly, the package provides a more concise syntax of creating customized stateless atomic components compared with doing the same thing using only intrinsic components. The concision of syntax when using the library is demonstrated in Listing 4.

```
...
// Using styled-component
import styled from "styled-components/native";

const CustomizedText = styled.Text`
    color: white;
    font-size: 16px;
`;

// Using plain React Native Text component
import React from "react";
import { StyleSheet, Text, TextProps } from "react-native";

const CustomizedText: React.FC<TextProps> = ({children, style,
    ...props}) => (
    <Text {...props} style={[{ color: 'white', fontSize: 16 },
        style]}>
        {children}
    </Text>
);

// With StyleSheet object
const styles = StyleSheet.create({
    customizedText: {
        color: 'white',
```

```
24          fontSize: 16
25      }
26  });
27
28  const CustomizedText: React.FC<TextProps> = ({children, style,
        ...props}) => (
29      <Text {...props} style={[styles.customized, style]}>
30          {children}
31      </Text>
32  );
```

Listing 4: Difference in syntax between using `styled-components` library (from line 3 to 8), and not using (from line 11 to 18, and from line 21 to 32)

With all benefits being illustrated, several layers of customized `Text` components in a few separate files were created. Firstly, `BasicText.tsx` file contains the layer of universal styles of text around the application, for example, most of text should be white. The content of the file is briefly demonstrated in Listing 5.

```
1  import styled, { css } from 'styled-components/native';
2  import { WHITE } from '@components/Colors';
3
4  export const StandardTextStyle = css`
5    color: ${WHITE};
6    font-family: 'Roboto';
7  `;
8
9  export const StandardText = styled.Text`
10   ${StandardTextStyle}
11 `;
```

Listing 5: The definition of the basic styled `StandardText` component

The `css` function of `styled-components` accept a string of CSS properties and generate a string of the same properties which could be then reused in components created by `styled-components`. Secondly, `AccessibleText.tsx` file contains the layer of universal accessibility configurations of text around the application. This is where the property `maxFontSizeMultiplier` is set, as demonstrated in Listing 6.

```
1  import styled, { css } from 'styled-components/native';
2
3  import { StandardText } from './BasicText';
4  import { FONT_SCALE_MAP } from '@config/accessibility';
```

```
5
6  export const FixedSizedText = styled(StandardText).attrs({
7      maxFontSizeMultiplier: FONT_SCALE_MAP.FIXED
8  });
9
10 export const MediumScalableText = styled(StandardText).attrs({
11     maxFontSizeMultiplier: FONT_SCALE_MAP.MEDIUM
12 });
13
14 export const LargeScalableText = styled(StandardText).attrs({
15     maxFontSizeMultiplier: FONT_SCALE_MAP.LARGE
16 });
```

Listing 6: The definitions of the font-size-scalable `Text`-based components

The `attrs` method, as seen at lines 6, 10 and 14 of Listing 6, accepts an object of properties of the corresponding component, then passes them down to the base component. In this case, it essentially sets the value of the `maxFontSizeMultiplier` property to the corresponding `FONT_SCALE_MAP` value. At the time of implementing the accessibility improvement in the project[3], the typing compatibility of `styled-components` with Typescript was dubious. Therefore, it was decided to use a more verbose syntax for some components which secure the type safety. At the time of writing this thesis[4], the compatibility has been significantly improved, which has made the code above reliable. Lastly, `StructuralText.tsx` file contains the layer of structuring text components in hierarchy. From here, hierarchical text components will be imported to use universally in the application. The content of the file is briefly demonstrated in Listing 7.

```
1  import styled, { css } from 'styled-components/native';
2
3  import { LargeScalableText, MediumScalableText } from
       './AccessibleText';
4
5  export const SmallTextCSS = css`
6    font-size: 18px;
7  `;
8
9  export const ParagraphText = styled(MediumScalableText)`
10   ${SmallTextCSS}
```

[3] April and May 2021
[4] September 2021

```
11 `;
12
13 const HeaderText = styled(LargeScalableText)`
14   text-align: center;
15 `;
16
17 export const NormalHeaderText = styled(HeaderText)`
18   font-size: 28px;
19 `;
20
21 export const LargeHeaderText = styled(HeaderText)`
22   font-size: 36px;
23   font-weight: bold;
24 `;
```

Listing 7: The definitions of structural text components, such as those from line 9 to line 24

With those being set, structural text components are ready to replace the built-in `Text` component of React Native. From here, instead of importing the default text component, structural components in the `StructuralText.tsx` file should be imported instead and be reused throughout the application. The code in Listing 8 shows how those text components are typically used in a screen.

```
1  import React from 'react';
2  import styled from 'styled-components/native';
3
4  import { BLACK } from '@components/Colors';
5  import { SCREEN_WIDTH } from '@components/Sizing';
6  import {
7    NormalHeaderText,
8    ParagraphText,
9  } from '@components/Text/StructuralText';
10
11 const TitleText = styled(NormalHeaderText)`
12   font-weight: bold;
13   margin-bottom: 15px;
14   margin-top: 20px;
15 `;
16
17 const DescriptionText = styled(ParagraphText)`
18   background-color: ${BLACK}99;
19   border-radius: 15px;
20   padding: 20px;
```

```
21    width: ${SCREEN_WIDTH * 0.95}px;
22  `;
23
24  const Game = () => {
25      // Logic here
26      return (
27          <ScrollView>
28              <TitleText>{title}</TitleText>
29              <DescriptionText>{description}</DescriptionText>
30          </ScrollView>
31      );
32  };
```

Listing 8: Typical use of structural text components in the application

Developers could use those hierarchical text components straight, or extend them as what the Listing 8 shows at lines 11 to 22. This keeps the design of the application consistent, and complies with the Open/Closed Principle, which, according to Meyer, states that

> Modules should be both open and closed. [...] A module is said to be open if it is still available for extension. [...] A module is said to be closed if it is available for use by other modules. [32]

This pattern also repeats for many other component implementations throughout the application. Thanks to this approach, all text in the application is appropriately scalable using accessibility features of the OS.

4.3 Enhance The Compatibility of Elements with TalkBack and VoiceOver

TalkBack is a built-in accessibility service on Android OS. It helps blind and low-vision users hear content available on a device's screen, and interact with the content via a set of defined gestures. Therefore, the users are able to navigate around the OS regardless of their visibility. VoiceOver is an equivalent solution on iOS platform. Both features will change how an application functions when they are enabled. A bounding box with a distinctive border will appear at the element presently selected to visually highlight it on the screen. The users could interact with the application through defined gestures, such as swiping up or down to move forward or backward to the next or previous element, respectively, instead of

using default ones, such as swiping up or down to scroll to that direction. How those two services look like on their corresponding OS are illustrated in Figure 7.

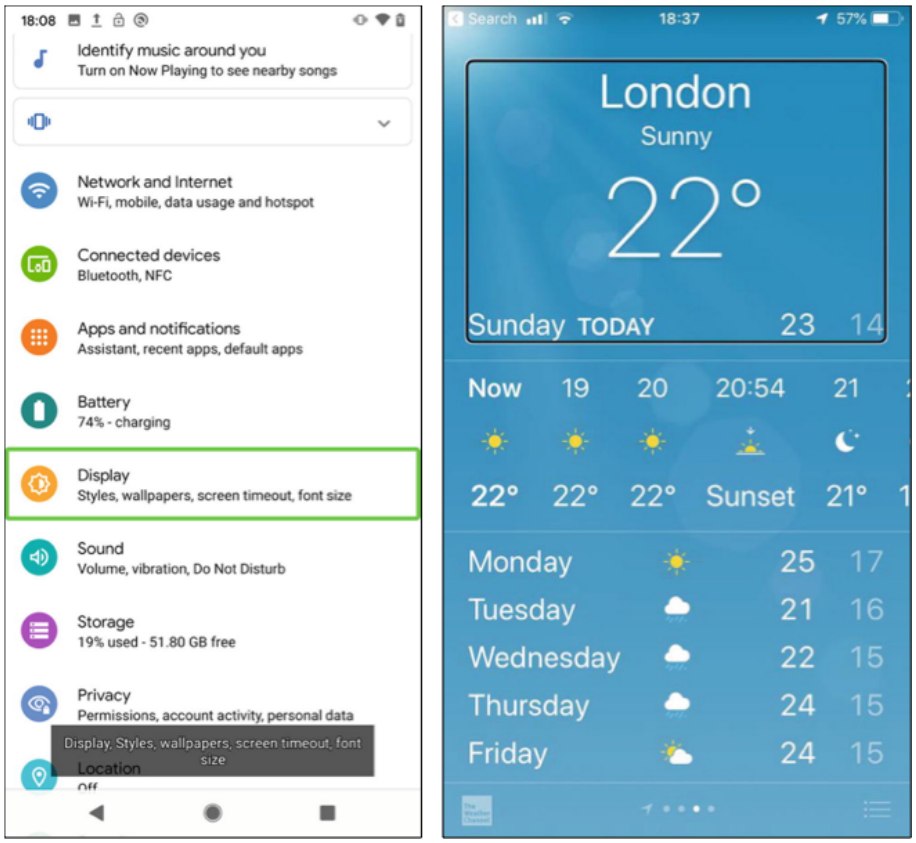

Figure 7: On the left, Android Display settings with TalkBack highlighting the control and reading its content. On the right, VoiceOver highlighting the top Summary Element in Apple's Weather application [6]

By default, all interactive view components in Android, iOS, and React Native are accessible, which means that they could be interacted with by assistive technology such as TalkBack and VoiceOver. Some separate components with a connected meaning could be bound together into a semantic component, or a semantic view. For example, in React Native, a common approach of creating a button component is wrapping a `Text` component inside a `TouchableOpacity` component, as could be seen at lines 14 to 16 in Listing 9. The former will act as the label of the button, while the latter will be responsible for handling visual changes and programmatic events when a user taps the button. The implementation of such a typical button in React Native code is demonstrated in Listing 9.

```
import React from "react";
import { StyleSheet, Text, TouchableOpacity, TouchableOpacityProps }
    from "react-native";

const styles = StyleSheet.create({
    wrapper: { ... },
    label: { ... },
});

interface IProps extends TouchableOpacityProps {
    label: string;
}

export const TypicalButton = ({label, onPress}: IProps) => (
  <TouchableOpacity onPress={onPress} style={styles.wrapper}>
    <Text style={styles.label}>{label}</Text>
  </TouchableOpacity>
);
```

Listing 9: A typical button component in React Native

The screen reader service will now treat the button component as a whole instead of two separate components `TouchableOpacity` and `Text`. It is worth noting that as the `TouchableOpacity` component is an interactive one, it is accessible innately. To form a semantic component without using interactive components, developers could wrap related child components inside a non-interactive `View` and set the property `accessible` of that wrapper as `true`.

Native platforms expose a great deal of accessibility API. Some of them are in the form of properties of the view component. Among those, the following will be covered in this thesis:

Content description (Android), or **Label** (iOS), is the first string read by the screen reader technology when an accessible component, without any special configurations, receives focus. Most views innately hold a content description/label, which is the view's text value. For example, the content description/label of a `<Text>Hello World</Text>` component is *Hello World*. For components which do not have text value, or their text value is lengthy, developers need to add content descriptions/labels manually. Furthermore, language usage of this property requires a few stipulations. A content description/label needs to be

succinct and descriptive, ideally in one word, such as *Submit*, *Login*. Developers should not include the type of component, for example, *button*, as the platform's assistive technology will do that by itself; manually adding such information is redundant and may cause noise to users. Also, the content description/label as a text value should begin with a capital letter and not end by a full stop [6]. In React Native, developers could modify the content description/label value using the property `accessibilityLabel`, which is available in all view components.

Hint (both Android and iOS) is usually used to provide more context to users, especially of what consequences will happen after the users interact with an element, such as *Navigates to home screen.*. As could be observed from the example, the hint should be capitalized and end with a period. Besides, Apple suggests considering the hint as a kind guidance to a friend. Hence, using *Navigates to home screen.* is preferred compared with using *Navigate to home screen.*, which sounds more like a instruction. The hint is read at last after the content description/label; and it is possible for the users to skip the hint in settings. Thus, it is recommended using hints as a fallback to provide context, and not relying on them to convey vital information [6]. In React Native, developers could modify the hint value using the property `accessibilityHint`, which is available in all view components.

Besides, React Native also offers a property named `accessibilityRole`, which will tell what a component is to the assistive technology so that it could announce the purpose of the component to users [33]. For example, the REGames application employs a game card component, which is basically an item in the home menu. It contains data of the corresponding game such as poster image, title, description. The component is implemented by wrapping text and image components inside a `TouchableOpacity`. That `TouchableOpacity` wrapper's `accessibilityRole` should be set as `"menuitem"` so that the screen reader will pronounce it out to the user. The simplified implementation of the component in code is demonstrated in Listing 10, while how it looks like in the application is illustrated in Figure 5.

```
import React from "react";
import { TouchableOpacityProps } from "react-native";
import styled from "styled-components/native";

const Wrapper = styled.TouchableOpacity.attrs({
  activeOpacity: 0.5,
})``;

interface IProps extends TouchableOpacityProps {
  gameDescription: string;
  gameStatus: string;
  gameTitle: string;
}

const GameCard = ({ gameDescription, gameStatus, gameTitle, onPress
    }: IProps) => (
  <Wrapper
    accessibilityLabel={`A game named ${gameTitle}, with the
        description of ${gameDescription}, and the status is
        ${gameStatus}`}
    accessibilityHint="Navigates to the preview screen of the
        selected game."
    accessibilityRole="menuitem"
    onPress={onPress}
  >
    ...
  </Wrapper>
);
```

Listing 10: The simplified implementation of the game card

The order of the content in which the assistive technology speak out should be firstly the content description/label, then the role, and lastly the hint, for instance, `A game named Carnivorous Garden, with the description of "try to escape the man-eating garden", and the status is playable. Menu item. Navigates to the preview screen of the selected game`. It may sound verbose in comparison with a standard button, but this is an edge case in which it is necessary to provide adequate information of the game to the client.

It should be aware of the fact that all content in the REGame application, including accessibility content, are in Finnish. All examples in this thesis are illustrated in English, but in the project, they are translated into Finnish. Still, all translated accessibility text keep fulfilling the requirements of aforementioned accessibility

properties. `accessibilityRole` value will be automatically translated based on the current language of the device the application is installed on. Instead of hard-coding the text as in the example above, all static text values are organized in a locale JavaScript Object Notation (JSON) file and imported wherever needed. At this phase, the application supports only Finnish. The translation was made and refined by several Punos Mobile employees, including the project manager of the thesis project, Hannu Alakangas.

Several other components are also mapped with different meaningful accessibility role. The respective values of `accessibilityRole` those elements are assigned to can be seen in Table 1.

Table 1: Components with their corresponding `accessibilityRole`

Component	**Description**	**`accessibilityRole`**
`Loader`	A spinning loading indicator	`"progressbar"`
`Timer`	A semantic component which displays time	`"timer"`
`Picker`	A collapsible menu which consists of multiple items inside for selection	`"spinbutton"`
`ScrollView`	A scroll view which contains game cards	`"menu"`
`CheckBox`	A checkbox which displays boolean value and allows toggling the value	`"switch"`

The `"spinbutton"` role may sound strange and unintuitive, but it is a common role which indicates that a component provide a range of defined selections and users could choose either option through the component [34]. Therefore, a dropdown menu or a picker, in mobile development language, should fall into that category. Another technical term is `"switch"`. It is not bizarre, but without a relevant context, it may be quite abstruse. A switch is a native component in mobile platforms, so it is available on both Android and iOS. It is used to display a boolean, i.e `true` or `false`, value, and to allow users toggling the value. The current implementation of the application employs a checkbox for that purpose instead of a native switch component. Checkbox is a common solution of indicating boolean value from web technology. Thus, it is alien to mobile environment. However, since changing it

back to a switch was not agreed to be in the scope of this thesis, it was decided to improve its accessibility by setting its `accessibilityRole` to `"switch"`.

Lastly, a couple of values of the property `accessibilityTraits` are assigned to the text of the game timer to inform the screen reader service about how to interact with a component based on its functionality [6]. This property is originally from iOS only, with the same name, and is mapped to the corresponding property in React Native. Therefore, it only benefits VoiceOver and iOS users. A good way to illustrate how this property works is through an example right in the application. In any escape room game, the application always shows a timer to indicate how much time remains for a user completing the game. The timer operates in two phases. It updates every second, and will stop counting when the given playing time is over. In the former active phase, the text value of the timer changes at a high recurrence. Hence, setting `accessibilityTraits` as `"frequentlyUpdate"` is an appropriate option. It communicates to VoiceOver that the assistive technology should concern about the value of the timer component at relevant intervals. In the latter phase, as the text will not change anymore, it should be considered as static text. Therefore, now updating the value of `accessibilityTraits` into `"text"` makes more sense. With that being set, VoiceOver treats the timer component as normal text. In code, the implementation will look as similar as Listing 11.

```
1  import React, { useState } from "react";
2  import styled from "styled-components/native";
3
4  import { parseTimeValue } from "./helpers";
5
6  interface IProps {
7      ...
8  }
9
10 const Timer = ({...}: IProps) => {
11     const [time, setTime] = useState(...);
12
13     return (
14       <Container>
15         ...
16         <ValueText accessibilityTraits={time !== 0 ?
               "frequentUpdates" : "text"}>
17             {parseTimeValue(time)}
```

```
18            </ValueText>
19          </Container>
20      );
21  };
```

Listing 11: The simplified implementation of the game timer

As could be seen in the code in the Listing 11 at lines 16 and 17, when the `time` value is equal 0, the application deems that all the playing time is used up. It then stops the timer. The value of the property `accessibilityTraits` is updated equivalently to reflect the change in the functionality of the timer. At this time, the application works better with assistive technology by providing enhanced awareness of context to the users.

4.4 Implement The Zoomable Hint Image of Quizzes

Quizzes in a game of the application are rendered from a certain amount of data. One of them is the hint image of a quiz, which supports players with visual clues. This image is rendered as a square image with the measurement of each size being equal to the width of the device's screen. The actual look of the image in the app is illustrated in Figure 8 .

Figure 8: The hint image of a quiz before the accessibility improving implementation

Before the implementation of this thesis project, the image was not interactive. Users might see the image but could not do much in case some details in the image are small and need to be zoomed in for clarification. This is especially common in mobile device use where the screen size is limited in comparison with devices equipped with large screen, such as laptop or desktop. A built-in solution available on both Android and iOS platforms is the magnification feature. In Android, the feature has the same name, while in iOS, it is called **Zoom**. And it should not be confused with another accessible feature named **Magnifier**, which functions in a totally different way. When being turned on in the OS's Accessibility settings, depends on the platform, users are able to perform certain sets of gestures to zoom in a part of the screen. For example, in iOS, users could use three fingers to double tap on the screen to zoom in the area they tap on.

However, the built-in approach exposes a few inconveniences. Firstly, it requires an extra step of activating an option in the Accessibility settings. Secondly, the gestures of zooming in demands additional efforts, such as use of auxiliary fingers, that may not be possible in some circumstances. Thirdly, the global effect throughout the OS of the feature may not be what the users desires in many cases. Therefore, more subtle solutions are appreciated. In the thesis project, it was decided to implement a zoomable image component using a third-party package named react-native-image-viewer[5]. The library provides several features of displaying images. One of them is an image component which aids zoom-ability. This component is helpful because of not only the magnification but also the support in enlarging the image in several different gestures. For example, besides zooming in and out using pinch gestures, users could double tap on the image to perform the same action. Additionally, the implementation of this feature only takes effect locally within the application, and does not require users to turn on the global zoom feature.

To use the magnifiable image component, a modal is needed as the container of the component. Modal is a component that conditionally displays on top of a contextually main view and deactivates the view below. The idea is turning the static hint image into a pressable image component, and by tapping on the component, the user will trigger the appearance of an image modal which includes a zoomable version of the same image. The modal also contains a button which allows the user interacting with to exit the modal, when necessary. The look of the modal is illustrated in Figure 9.

[5]https://github.com/ascoders/react-native-image-viewer

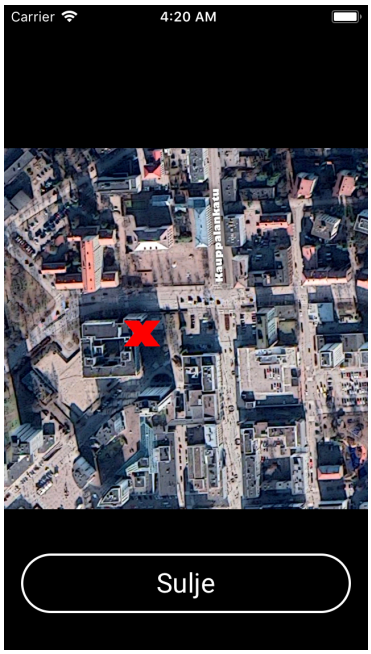

Figure 9: The look of the modal which contains the zoomable image

It is noticeable that even in the modal, the accessibility textual values should not be forgotten. And in the real code, again, they are actually imported from a Finnish locale JSON file. In code, the implementation should look as Listing 12.

```
1  import { Modal } from 'react-native';
2  import ImageViewer from 'react-native-image-zoom-viewer';
3  import styled, { css } from 'styled-components/native';
4
5  import CapsuleButton from '@components/Buttons/CapsuleButton';
6
7  const StyledWrapper = styled.View`
8      ...
9  `;
10
11 interface IProps {
12   onCloseBtnPress: () => void;
13   url: string;
14   visible: boolean;
15 }
16
17 const AccessibleZoomableImage = ({ onCloseBtnPress, url, visible }:
       IProps) => {
18   ...
19
20   const images = [{ url }];
21
```

```
22    return (
23      <Modal {...{ visible }}>
24        <ImageViewer
25          imageUrls={images}
26          renderFooter={() => (
27            <StyledWrapper>
28              <CapsuleButton
29                label={"Close"}
30                accessibilityHint={"Double click to close the zoomable
                      image modal window."}
31                onPress={onCloseBtnPress}
32              />
33            </StyledWrapper>
34          )}
35        />
36      </Modal>
37    );
38  };
```

Listing 12: The simplified implementation of the `AccessibleZoomableImage` component

At this time, in the quiz screen, the hint image component should be wrapped inside an interactive component such as a button. However, due to the importance of retaining the clarity of the hint image, its look should not be changed with the purpose of indicating that the image is now interactive. Instead, text with the same meaning were placed below the image so that the user could notice the feature and use it if needed. The look of the image area of the quiz view after the implementation of the feature is shown in Figure 10.

Figure 10: The look of the image area of the quiz view after the implementation of the zoomable image feature

While the look of the view does not change significantly, under the hood, the code is updated considerably. Firstly, to enable in the interactivity of the image, it is necessary to wrap the non-interactive `Image` component inside the non-interactive `TouchableOpacity` component so that they will combine into an image button. Secondly, React Native accessibility API suggests the value of `"imagebutton"` for the property `accessibilityRole` to distinguish between an image button and a normal image. Thanks to that, assistive technology could communicate the interactivity of the component to users. At this time, it is necessary to separate the image button component into separate ones with distinct functionalities to improve its reusability. The following Listing elaborates how those components look in code, starting with the reusable `AccessibleImageButton` component in Listing 13.

```
1  import React from 'react';
```

```
2  import { TouchableOpacity, TouchableOpacityProps } from
       'react-native';
3  import styled, { css } from 'styled-components/native';
4
5  export interface IBaseAccessibleButtonProps extends
       TouchableOpacityProps {
6    children?: ReactNode | ReactNode[];
7  }
8
9  const BaseAccessibleButton = ({ children, ...props }:
       IBaseAccessibleButtonProps) => (
10     <TouchableOpacity {...props}>
11       {children}
12     </TouchableOpacity>
13   ),
14 );
15
16 export interface IBaseAccessibleButtonProps extends
       TouchableOpacityProps {
17   children?: ReactNode | ReactNode[];
18 }
19
20 export const AccessibleImageButton = ({ ...props }:
       IBaseAccessibleButtonProps) => (
21     <BaseAccessibleButton
22       {...props}
23       accessibilityRole="imagebutton"
24     />
25   ),
26 );
```

Listing 13: The implementation of the reusable `AccessibleImageButton` component

As could be seen in Listing 13, the `AccessibleImageButton` is just a normal `TouchableOpacity` wrapper whose `accessibilityRole` property is pre-assigned as `"imagebutton"`. With `AccessibleImageButton` being implemented, the hint image component is ready to be separated into an independent component as demonstrated in Listing 14.

```
1  import React from 'react';
2  import { Image, ImageProps } from 'react-native';
3
4  import {
5    AccessibleImageButton,
```

```
6     IBaseAccessibleButtonProps,
7   } from '@components/Buttons/AccessibleButton';
8
9   interface IProps {
10    buttonProps: IBaseAccessibleButtonProps;
11    imageProps: ImageProps;
12  }
13
14  const HintImage = ({ buttonProps, imageProps }: IProps) => (
15    <AccessibleImageButton {...{ ref }} {...buttonProps}>
16      <Image {...imageProps} />
17    </AccessibleImageButton>
18  );
19
20  export default HintImage;
```

Listing 14: The simplified implementation of the `HintImage` component

Now, every components are all set. The last step is plugging the `HintImage` component in the game view. Then, it is crucial to add the instruction text below to gently give a visual clue to normal users about the interactivity of the hint image, and to add appropriate text values of accessibility features to the `HintImage` for those who need support from the assistive technology. Those ideas are demonstrated in code as shown in Listing 15.

```
1   import React, { useState } from 'react';
2   import styled from 'styled-components/native';
3
4   import { SmallText } from '@components/Text/StructuralText';
5
6   const InstructionText = styled(SmallText)`
7     ...
8   `;
9
10  interface IProps {
11      image: string;
12      ...
13  }
14
15  const Game = ({ image }: IProps) => {
16      // Logic here
17      const [isZoomableImageVisible, setIsZoomableImageVisible] =
            useState(false);
18
19      const onCloseButtonOfZoomableImagePress = () => {
```

```
20          setIsZoomableImageVisible(false);
21      };
22
23      return (
24          <ScrollView>
25              {isZoomableImageVisible && (
26                  <AccessibleZoomableImage
27                    onCloseBtnPress={onCloseButtonOfZoomableImagePress}
28                    visible={isZoomableImageVisible}
29                    url={image}
30                  />
31              )}
32              <HintImage
33                  buttonProps={{
34                    accessibilityHint: "Double click to open the
                          zoomable image modal window.",
35                    onPress: () => {
36                      setIsZoomableImageVisible(true);
37                    },
38                  }}
39                  imageProps={{
40                    source: { uri: image },
41                  }}
42              />
43              <InstructionText>
44                  {"Click on the image to open the zoomable image modal
                        window"}
45              </InstructionText>
46          </ScrollView>
47      );
48  };
```

Listing 15: The simplified implementation of the whole zoomable image module in the game view

As can be observed from Listing 15, a boolean state variable named `isZoomableImageVisible` is created with the initial value of `false` using `useState` function from React. This variable determines whether the zoomable image should display or not. Initially, the zoomable image does not appear. Along with initiation of the `isZoomableImageVisible` variable, the invocation of the `useState` function also results in a function named `setIsZoomableImageVisible`, which is used to change the value of the `isZoomableImageVisible` variable. The function is used in the `onPress` handler of the `HintImage` component to open the zoomable image modal when the user taps on hint image, and in the

`onCloseButtonOfZoomableImagePress`, which is then passed to `onCloseBtnPress` handler of the zoomable image modal, to close the modal when the user taps on the *Close* button (*Sulje* in Finnish). The hint image and the zoomable image share the same use of the `image` variable, which is a link to a remote image fetched from the network.

With that being stated, the zoomable image feature is now fully functional. Users could now interact with the hint image to open the zoomable image modal, then magnify any part in the image to observe more clearly.

4.5 Improve The Accessibility of Images

Images play a vital role in REGames application. Some provide crucial visual information which will severely disrupt the user experience if they do not exist, for example, the hint image of each quiz. Others act as graphical ornaments, which boost the visual appeal of the application and consolidate the feeling of authenticity of the content. For example, in a game with the theme of detective, after successfully solving a certain amount of quizzes, a user will be honored by a digital achievement named *Master Detective*. In this case, a graphical image of a delicate magnifier may be an appropriate avatar for such achievement, which will enhance the realisticness and excitement of the user.

Besides the logo and the background image, all other images are pulled from the network. As with other assets from the Internet, the risk of broken connection or any types of data corruption should be invariably considered. A common practice in mobile world is replacing the actual image by a pictorial placeholder in the loading time, or when problems occur. However, since all of the remote images are precious to the application, the placeholder image is probably not satisfactory. Therefore, it was decided to follow the common practice of the web technology, using alternative text for the missing images.

React Native does not supply any straightforward API in the `Image` for that feature like the `alt` attribute of `<img>` tag in HyperText Markup Language (HTML) of the

web world. To implement the feature, the idea is initially rendering the normal `Image` component, and when the error of displaying the actual image happens, the application will switch to rendering the alternative text of the image. Now, because the accessible image contains its own state of whether displaying the image or the alternative text, as well as the component itself is reusable in replacement of every single normal image around the application, it is necessary to put it into a discrete component. In code, the look of the implementation is demonstrated in Listing 16.

```
1  import React, { useState } from 'react';
2  import {
3    Image,
4    ImageErrorEventData,
5    ImageProps,
6    NativeSyntheticEvent,
7    TextStyle,
8  } from 'react-native';
9
10 import { SmallText } from '@components/Text/StructuralText';
11
12 export interface IAccessibleImageProps extends ImageProps {
13   alt: string;
14   altTextStyle?: TextStyle;
15 }
16
17 const AccessibleImage = ({
18   alt,
19   altTextStyle,
20   ...props
21 }: IAccessibleImageProps) => {
22   const [isFailedToLoad, setIsFailedToLoad] = useState(false);
23
24   const onError = (error: NativeSyntheticEvent<ImageErrorEventData>)
         => {
25     setIsFailedToLoad(true);
26   };
27
28   return isFailedToLoad ? (
29     <SmallText accessibilityLiveRegion="polite"
           style={altTextStyle}>
30       {alt}
31     </SmallText>
32   ) : (
33     <Image {...props} {...{ onError }} accessible
           accessibilityLabel={alt} />
```

```
34    );
35  };
36
37  export default AccessibleImage;
```

Listing 16: The simplified implementation of the `AccessibleImage` component

As could be seen in the Listing 16, the implementation of the image with alternative text is placed into a separate component named `AccessibleImage`. The component accepts all properties of the built-in `Image` component of React Native, in addition to its own compulsory string-type `alt` property and optional `altTextStyle` property, which shares the same shape of the style object of the integrated `Image` component in the ecosystem. While the former determines the content of the alternative text, the latter is responsible for the visual look of the text. Therefore, developers are able to set the content of the alternative as well as styling it as their wish. Furthermore, the boolean state named `isFailedToLoad` with the initial value of `false` concludes if the image loading process fails or not to render the alternative text or the image itself, respectively. The value of the state could be modified to `true` by the `setIsFailedToLoad` function, which is used in the `onError` handler of the `Image` component. As the name suggests, the handler will be called when the loading process of the image crashes.

Besides, the `Image` component is now set as `accessible` with the value of the `accessibilityLabel` being the value of the `alt` property. This means that when the image is loaded successfully, assistive technology could interact with it and will speak out the alternative text to the user. That is valuable to those players who are not able to directly see the image because of some certain disadvantages, as now they could listen to the verbal description of the context of the image.

It is worth noticing that the value of the `accessibilityLiveRegion` property of the alternative text component is set as `"polite"`. Live region is an accessibility concept available only in Android platform. It is an area whose updated content will be notified to the user even if the current focus of TalkBack is not on the area. This is helpful when developers would like to alert the user to certain updates of in the application using assistive technology. The property accepts either of the two

following value: `"polite"` and `"assertive"`. The former will subtly wait for ongoing speeches to complete, then the update will be announced. On the other hand, the latter is more intrusive intervening the present voice and informing the update immediately. Unless the situation suggests a definitely good reason to urgently warn the user of what happens, it is recommended to use `"polite"` as much as possible [6]. In React Native, the live region feature is mapped to the `accessibilityLiveRegion` property. Besides being used in the `AccessibleImage` component, the `accessibilityLiveRegion` property is also set in all error message text components so that they will notify the user of their advent in case errors occur.

At this point, the `AccessibleImage` component is available and could be used throughout the application wherever a normal image is needed. For example, in the main menu of the application, all available games are illustrated as card items in a vertical scrolling view. Each game card contains a theme image of the corresponding game and some information text, as can be observed in Figure 11.

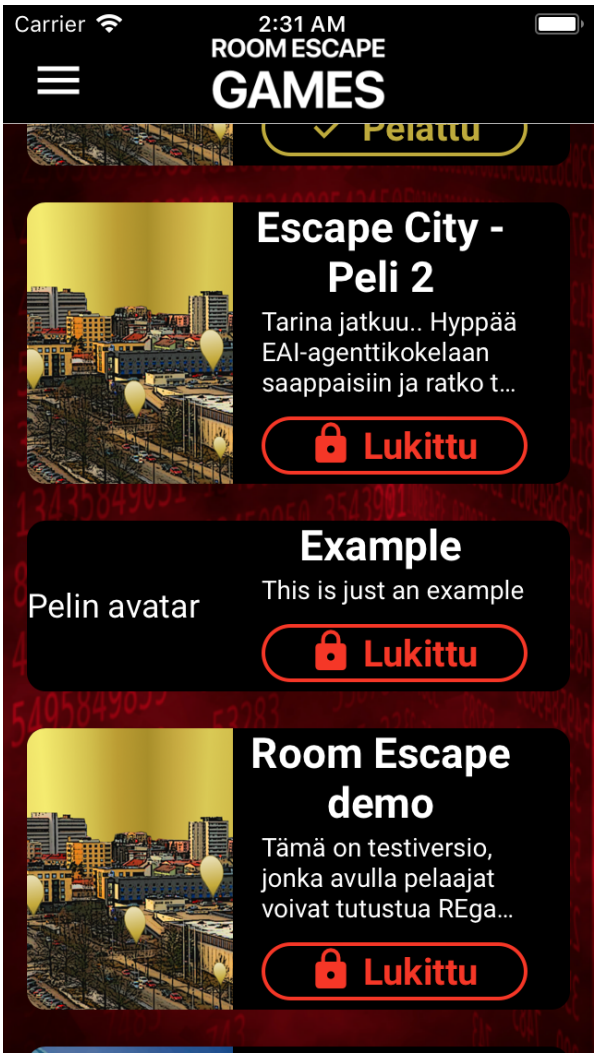

Figure 11: Game cards with accessible images implemented

The visual presentation of the component, however, is not able to demonstrate its non-visual features. To understand those, having a look at the code is necessary. In code, the game card demonstrated in the Figure 11 looks as illustrated in Listing 17.

```
1  import React from 'react';
2  import styled from 'styled-components/native';
3
4  import { IGameData } from '@config/Types';
5
6  const Wrapper = styled.TouchableOpacity``;
7
8  const Container = styled.View`
9    ...
10 `;
```

```
11
12  const CardImage = styled(AccessibleImage)`
13    ...
14  `;
15
16  interface IProps {
17    game: IGameData;
18  }
19
20  const GameCard = ({ game }: IProps) => (
21    <Wrapper>
22      <Container>
23        <CardImage
24          alt={game.alt || "Avatar of the game"}
25          altTextStyle={{ width: "38%", alignSelf: "center" }}
26          source={{ uri: game.image }}
27        />
28      </Container>
29    </Wrapper>
30  );
31
32  export default GameCard;
```

Listing 17: The simplified implementation of the `GameCard` component, which contains an instance of the accessible image

In the Listing 17, the accessible image is the `CardImage` component, which is basically the `AccessibleImage` component with additional styling. Conspicuously, the implementation grants the authority of determining the alternative text to the backend of the application through the `alt` field of the game data variable named `game`, which is fetched from a remote database. If, for any reasons, the `alt` field is not available in the game data, the application will apply the default alternative text of *Avatar of the game* (*Pelin avatar* in Finnish) to the image of the card game. Because most of the images in the application are remote assets, this approach makes sense and beneficial to Fitseven Oy. The same solution is applied universally to all remote images in the application.

With those being set up, most of the images in the application are accessible. Users, with or without the help of assistive technology, would benefit from the feature even if the Internet connection is disrupted.

5 Conclusion

The objective of this thesis was to improve the accessibility of the REGames application developed using React Native. Furthermore, through the implementation of accessible features, several good practices in integrating accessibility in mobile development were mentioned in the study, such as allowing the backend controlling the accessibility label of component, or creating a hierarchical typographic component system and reusing it instead of using arbitrary typographic components around an application. However, since accessibility is a gigantic concept, further improvements outside of the scope of the study are invariably available.

The enforcement of accessibility regulation over the web and mobile world requires businesses to seriously cultivate the idea of inclusion in their existing and upcoming digital services. To operate legally, web and mobile applications now must fulfill certain accessibility standards. This is an enormous step in equality when a great amount of the population are able to access the information and use digital products with sufficient support based on their physical and environmental conditions. It also gives developers an opportunity to expand their understanding in accessibility as a concept and through practical implementation in their expertise.

As a cross-platform mobile development technology, React Native provides numerous tools to improve the accessibility of an application. Thanks to built-in accessible solutions of the technology, the implementation of the thesis did successfully enhance the interactivity of the REGames application with assistive technology on both Android and iOS platforms. Besides, learning from the implementation of the zoomable image component, developers should consistently seek ways to improve the user experience which may not be existing by default, or may be available but with certain drawbacks.

After the implementation of this thesis project, all involved parties witnessed the

clear advancement of the accessibility of the application. They also gained useful insight about the inclusion aspect in mobile development. At the time, the application could support users through working appropriately with assistive technology. Color contrast in the user interface was calibrated so that users could easily distinguish between elements on the screen. Additionally, the application also resolved the previous issue of fixed-size hint images by offering the zoomable image solution, which allows user magnifying the images for a better view. Lastly, all remote images in the application now includes their own contextual information. Thus, in case the loading process causes issues, the user could understand the meaning the images try to express.

The study tried to deliver knowledge of accessibility, not only in the mobile development industry but also generally in daily life. Also, it attempted to explain the reason behind any action in the implementation with relevant theoretical references, or based on experience of the author in the software industry field. Nonetheless, any project customarily incurs a certain number of limitations, possibly in finance, human relationship, and time budget. Hence, the responsibility of the development team is to maximize the inclusion of their applications in balance with those restraints.

References

1 Valtiovarainministeriö. 2019. Saavutettavuus. Online. <https://vm.fi/saavutettavuusdirektiivi>. Visited on 08/09/2021.

2 Jones, Katie. 2020. These Countries Are Aging the Fastest - Here's What It Will Mean. Online. <https://www.weforum.org/agenda/2020/02/ageing-global-population>. Visited on 08/09/2021.

3 Statistics Finland. 2021. Population. Online. <https://www.stat.fi/tup/suoluk/suoluk_vaesto_en.html>. Visited on 08/09/2021.

4 Kalbag, Laura. 2017. Accessibility for Everyone. New York: A Book Apart.

5 The World Bank. 2020. Literacy Rate, Adult Total (% of People Ages 15 and Above). Online. <https://data.worldbank.org/indicator/SE.ADT.LITR.ZS>. Visited on 08/09/2021.

6 Whitaker, Rob. 2020. Developing Inclusive Mobile Apps: Building Accessible Apps for iOS and Android. New York: Apress.

7 The World Wide Web Consortium. 2021. Web Content Accessibility Guidelines (WCAG) Overview. Online. <https://www.w3.org/WAI/standards-guidelines/wcag/>. Visited on 08/17/2021.

8 — 2016. Web Content Accessibility Guidelines (WCAG) Overview. Online. <https://www.w3.org/TR/UNDERSTANDING-WCAG20/intro.html>. Visited on 08/17/2021.

9 — 2019a. How to Meet WCAG (Quick Reference). Online. <https://www.w3.org/WAI/WCAG21/quickref/?versions=2.0>. Visited on 08/18/2021.

10 — 2019b. How to Meet WCAG (Quickref Reference). Online. <https://www.w3.org/WAI/WCAG21/quickref/?versions=2.0&showtechniques=124%5C%2C126#audio-only-and-video-only-prerecorded>. Visited on 11/05/2021.

11 — 2019c. How to Meet WCAG (Quickref Reference). Online. <https://www.w3.org/WAI/WCAG21/quickref/?versions=2.0&showtechniques=126%5C%2C124#captions-live>. Visited on 11/05/2021.

12 The World Wide Web Consortium. 2019d. How to Meet WCAG (Quickref Reference). Online. <https://www.w3.org/WAI/WCAG21/quickref/?versions=2.0&showtechniques=126%5C%2C124#sign-language-prerecorded>. Visited on 11/05/2021.

13 Android Developers Website. 2021. Build More Accessible Apps. Online. <https://developer.android.com/guide/topics/ui/accessibility>. Visited on 08/18/2021.

14 Apple Inc. 2021. Accessibility. Online. <https://developer.apple.com/design/human-interface-guidelines/accessibility/overview/introduction/>. Visited on 08/18/2021.

15 Facebook and React Native Community. 2021a. Accessibility. Online. <https://reactnative.dev/docs/accessibility>. Visited on 08/18/2021.

16 Mozilla Developer Network. 2021a. JavaScript. Online. <https://developer.mozilla.org/en-US/docs/Web/JavaScript>. Visited on 08/24/2021.

17 — 2021b. A Re-introduction to JavaScript (JS tutorial). Online. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript>. Visited on 08/24/2021.

18 Facebook Open Source. 2021. React – A JavaScript library for building user interfaces. Online. <https://reactjs.org/>. Visited on 11/01/2021.

19 Stack Overflow. 2020. Stack Overflow Developer Survey 2020. Online. <https://insights.stackoverflow.com/survey/2020#technology-web-frameworks>. Visited on 08/24/2021.

20 Education Ecosystem. 2021. React.js History - Education Ecosystem. Online. <https://www.education-ecosystem.com/guides/programming/react-js/history>. Visited on 11/01/2021.

21 React Native for Windows + macOS · Build Native Windows and macOS Apps with Javascript and React 2019. Online. <https://microsoft.github.io/react-native-windows/>. Visited on 08/24/2021.

22 react-native-tvos. 2019. Github Repository of react-native-tvos. Online. <https://github.com/react-native-tvos/react-native-tvos>. Visited on 08/24/2021.

23 Facebook and React Native Community. 2021b. Introduction · React Native. Online. <https://reactnative.dev/docs/getting-started>. Visited on 09/07/2021.

24 Eisenman, Bonnie. 2015. Learning React Native: Building Native Mobile Apps with JavaScript. Sebastopol, California: O'Reilly Media, Inc.

25 Microsoft and Typescript Community. 2012. TypeScript: JavaScript With Syntax for Types. Online. <https://www.typescriptlang.org/>. Visited on 09/07/2021.

26 Grynhaus, B., Hudgens J., Hunte R., Morgan M., Stefanovski W. 2021. The TypeScript Workshop: A Practical Guide to Confident, Effective TypeScript Programming. Birmingham: Packt Publishing Ltd.

27 Bureau of Internet Accessibility, Inc. 2019. What is color contrast? Online. <https://www.boia.org/blog/what-is-color-contrast>. Visited on 04/20/2019.

28 Accessibility Developer Guide. 2019. How to calculate colour contrast. Online. <https://www.accessibility-developer-guide.com/knowledge/colours-and-contrast/how-to-calculate/>. Visited on 12/15/2019.

29 The World Wide Web Consortium. 2008a. Understanding Success Criterion 1.4.3 | Understanding WCAG 2.0. Online. <https://www.w3.org/TR/UNDERSTANDING-WCAG20/visual-audio-contrast-contrast.html>. Visited on 09/17/2021.

30 Frost, Brad. 2016. Atomic Design. Pittsburgh, Pennsylvania: Brad Frost.

31 The World Wide Web Consortium. 2008b. Understanding Success Criterion 1.3.1 | Understanding WCAG 2.0. Online. <https://www.w3.org/TR/UNDERSTANDING-WCAG20/content-structure-separation-programmatic.html>. Visited on 09/20/2021.

32 Meyer, Bertrand. 2000. Object-Oriented Software Construction - Second Edition. New York: Prentice Hall PTR.

33 Facebook and React Native Community. 2020. Accessibility · React Native. Online. <https://reactnative.dev/docs/accessibility#accessibilityrole>. Visited on 09/24/2021.

34 Digital A11Y. 2018. WAI-ARIA: Role=Spinbutton. Online. <https://www.digitala11y.com/spinbutton-role/>. Visited on 09/24/2021.