BUILDING A HYBRID APPLICATION USING REACT NATIVE
Abstract

<table>
<thead>
<tr>
<th>Author(s)</th>
<th>Type of publication</th>
<th>Published</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sairanen, Pauli</td>
<td>Bachelor’s thesis</td>
<td>Spring 2020</td>
</tr>
<tr>
<td>Number of pages</td>
<td>60</td>
<td></td>
</tr>
</tbody>
</table>

Title of publication

**Building a hybrid application using React Native**

Name of Degree

Bachelor of Information Technology

Abstract

The objective of this thesis was to create a hybrid application that could be used at various events organized by the customer. The subgoals of the thesis were formed based on the requirements the customer had for the application.

Research was done in order to demonstrate the different technologies used in mobile development as well as pros and cons of using either native, web or hybrid technologies for development.

React Native was used as the development platform in this thesis. Further research was conducted in order to gain a better understanding of the operating principles of React Native and React, as well as other technologies required in the development process.

The process of creating a hybrid application consisted of different work phases. The purpose of each phase was to implement the required features for the application in an organized manner.

As a result, a working hybrid application was created according to the requirements specifications. The application was released on Google Play store and used at the event organized by the customer.

Keywords

React, React Native, hybrid application, mobile development, JavaScript
# Tiivistelmä

<table>
<thead>
<tr>
<th>Tekijä(t)</th>
<th>Julkaisun laji</th>
<th>Valmistumisaika</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sairanen, Pauli</td>
<td>Opinnäytetyö, AMK</td>
<td>Kevät 2020</td>
</tr>
<tr>
<td></td>
<td>Sivumäärä 60</td>
<td></td>
</tr>
</tbody>
</table>

**Työn nimi**  
**Building a hybrid application using React Native**

**Tutkinto**  
Insinööri (AMK)

**Tiivistelmä**

Opinnäytetyön tarkoituksena oli luoda hybridimobiilisovellus, jota voidaan käyttää erilaisissa asiakkaan järjestämissä tilaisuuksissa. Opinnäytetyön osatavoitteet muodostuivat asiakkaan sovelluksen sovellusta koskevista vaatimuksista.


Sovelluksen kehittämisprosessi koostui useista työvaiheista. Jokaisen työvaiheen tarkoituksena oli luoda sovelluksen vaatimusten mukaisia toimintoja järjestelmällisesti.

Lopputuloksena saatiin kehitetty määräysten mukainen toimivaa hybridimobiilisovellus. Sovellus julkaistiin Google Play-kaupassa, ja sovellus oli myös käytössä asiakkaan järjestämässä tapahtumassa tavoitteiden mukaisesti.

**Asiasanat**  
React, React Native, hybridisovellus, mobiili kehittäminen, JavaScript
# TABLE OF CONTENTS

1  INTRODUCTION .................................................................................................................. 1  
2  MOBILE APPLICATIONS ..................................................................................................... 2  
   2.1  Native Application ......................................................................................................... 2  
   2.2  Web application ........................................................................................................... 2  
   2.3  Hybrid Applications .................................................................................................... 3  
3  TECHNOLOGIES .................................................................................................................. 4  
   3.1  Node.js .......................................................................................................................... 4  
   3.2  React ............................................................................................................................ 4  
      3.2.1  JSX ....................................................................................................................... 4  
      3.2.2  Functional Components ....................................................................................... 5  
      3.2.3  Lifecycle Methods and Hooks ............................................................................ 5  
      3.2.4  Props .................................................................................................................... 6  
      3.2.5  State ..................................................................................................................... 7  
      3.2.6  Redux and global state ......................................................................................... 8  
   3.3  React Native .................................................................................................................. 10  
      3.3.1  Native Rendering components ............................................................................ 10  
      3.3.2  React Native Bridge ............................................................................................ 11  
      3.3.3  React Native components ................................................................................... 12  
      3.3.4  Custom components ............................................................................................ 14  
4  THE SAWMILLEVENTS APPLICATION ............................................................................ 15  
   4.1  Planning ......................................................................................................................... 16  
   4.2  Development ................................................................................................................ 16  
      4.2.1  Setting up the environment and the project .......................................................... 16  
      4.2.2  Adding version control ....................................................................................... 17  
      4.2.3  File structure of the project .............................................................................. 17  
      4.2.4  Constants .............................................................................................................. 20  
      4.2.5  Creation of screens with FlatList ........................................................................ 21  
      4.2.6  Main Navigation Screen ..................................................................................... 26  
      4.2.7  Other finished screens ......................................................................................... 30  
      4.2.8  Navigation ............................................................................................................ 36  
      4.2.9  Login ..................................................................................................................... 38  
      4.2.10 HTTPS requests and backend integration ............................................................ 40  
   4.3  Publishing and distribution .......................................................................................... 44
<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface. Allows different programs to request and exchange data with each other</td>
</tr>
<tr>
<td>back-end</td>
<td>Server side of an application that handles storing and distribution of the data</td>
</tr>
<tr>
<td>cross-platform</td>
<td>Application that has the same look and functionality regardless of the operating system</td>
</tr>
<tr>
<td>framework</td>
<td>Software or collection of software and libraries for building a certain type of software product</td>
</tr>
<tr>
<td>front-end</td>
<td>Application that is used by the end-user. Commonly used interchangeably with user interface (UI)</td>
</tr>
<tr>
<td>HTTPS</td>
<td>Hyper Text Transfer Protocol Secure. Used for transferring information over the Internet</td>
</tr>
<tr>
<td>hybrid application</td>
<td>Mobile application built using web and native technologies together to achieve a cross-platform compatibility</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment. A software with graphical user interface, that is used for creating software</td>
</tr>
<tr>
<td>JavaScript Core</td>
<td>Open source JavaScript engine</td>
</tr>
<tr>
<td>JDK8</td>
<td>Java Development Kit 8. Used in Android Development</td>
</tr>
<tr>
<td>Node.js</td>
<td>JavaScript development platform</td>
</tr>
<tr>
<td>npm</td>
<td>Node package manager. Allows downloading and installing JavaScript libraries</td>
</tr>
<tr>
<td>React</td>
<td>JavaScript library for creating user interfaces. Developed by Facebook</td>
</tr>
<tr>
<td>React-Native</td>
<td>Hybrid app development framework developed by Facebook</td>
</tr>
<tr>
<td>UI</td>
<td>User interface</td>
</tr>
<tr>
<td>V8</td>
<td>JavaScript engine created by Google</td>
</tr>
<tr>
<td>XML</td>
<td>Extensive Markup Language. A data exchange format</td>
</tr>
</tbody>
</table>
1 INTRODUCTION

Mobile application development is getting more popular as the number of mobile devices has widely increased. This provides a growing number of people to gain access to the internet regardless of location, making these people seen as potential customers.

Different hybrid application frameworks have emerged and are offering more options for developers to choose from. Creating a hybrid application can be quick and cost efficient, which usually is a major factor in today´s fast moving business world.

The Finnish Sawmills Association is an interest organization for the Finnish sawmills industry that promotes the business of its member companies by overseeing their interest. The Association organizes various events throughout the year where representatives of its member companies are brought together to discuss the topics surrounding the business. The headquarters of the association is located in Helsinki and the total turnover was 329 000 euros in 2018 (Finder 2020).

The purpose of this thesis was to produce a working hybrid application for Finnish Sawmills Association using React Native as the development platform. The application was meant to be used at the Wood From Finland 2020 event organized by the customer. Other smaller goals were that the application should have a responsive UI, which would scale accordingly on different devices and screen sizes. Also, the internal structure of the application should support different events and should be able to display different types of data and images. Furthermore, the application should be published and be available from Google Play and also possibly from App Store. The theory section of this thesis will cover the technologies used in this project.
2 MOBILE APPLICATIONS

There are multiple approaches how a mobile application can be created. The three main categories of approaches are native, web and hybrid. In this chapter the native and web approaches will be explained briefly in order to state how the hybrid approach utilizes those two technologies.

2.1 Native Application

The two major mobile operating systems are Android, developed and maintained by Google, and iOS, developed and maintained by Apple. Both operating systems run native applications created exclusively using a platform specific programming language and an IDE (Integrated Development Environment). For Android the platform specific programming language is either Java or Kotlin. For iOS the language is either Swift or Objective-C. (Arshed 2018.)

When performance is the most important aspect of the application, the native approach is recommended. However, as native applications are made using different programming languages and tools, creating the same application separately for each platform is costly and time consuming.

2.2 Web application

Web technologies include HTML, CSS and JavaScript. HTML is used in creating the structure or containers for the contents that are to be shown by a web-browser. CSS is used in adding styles or visual effects, such as colors, shapes and fonts. JavaScript, a scripting / programming language is used in creating dynamic features and handling the logic of the web application.

Web applications are accessed by the browser over the Internet. Nowadays these applications are usually built to support both mobile and desktop devices.

This approach provides the lowest development costs and enables applications to run on any device using a modern web browser. However, in order to work, web applications require a reliable internet connection. Since web applications are executed in a browser, the user experience and performance of these applications is inferior, compared to native applications. (Arshed 2018.)
2.3 Hybrid Applications

The separation of the operating systems and programming languages has driven the development of solutions that allow developing mobile applications that can be executed on both operating systems using only one code base. The common idea in hybrid development is to utilize technologies that already are cross-platform. Nowadays modern web browsers are supported by the most common operating systems, which makes web technologies cross-platform and thus a good choice for hybrid development.

Multiple hybrid application frameworks have been created using web technologies as the base. The most common approach has been to build a web application and run it in a WebView, which is basically a web browser running inside a native application. This approach provides a simpler development process but lacks native performance and responsiveness in more demanding applications.

Another approach is to combine web technologies with native building blocks of both iOS and Android operating systems. Instead of using a HTML and CSS, these applications are written only in JavaScript, which is translated into the native user interface elements for both iOS and Android. (Butusov 2019.) Under the hood of the application, an open source JavaScript engine is used to execute the JavaScript code. (Facebook.github.io JavaScript Runtime)

As native applications are created using different programming languages and tools, creating the same application for each platform is costly and time consuming. Using a hybrid approach requires smaller developer teams and makes development faster. (Butusov 2019.)
3 TECHNOLOGIES

3.1 Node.js

Node.js is an asynchronous JavaScript runtime environment created by Ryan Dahl. It is built on top of V8, a JavaScript engine created by Google. (Ornbo 2012, 7.)

According to the official website of Node.js:

Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js’ package ecosystem, npm is the largest collection of open source libraries in the world (Node.js 2020).

Using Node.js, developers are able to create standalone applications using JavaScript instead of running the code only in browser. With the installation of Node.js comes npm, a node package manager that allows downloading and installing different JavaScript libraries to be used in development. These libraries are also called Node Modules.

3.2 React

React is a JavaScript library developed by Facebook for building user interfaces. It has gained wide popularity among developers for its component-focused approach and because it is quick to learn. React was originally designed for web development, and later React-Native, a cross-platform mobile application development framework, was created based on React.

The strength of React is its simplicity and size. Since React is only a library and not a full framework it is much faster to learn. The modular structure of the library allows developers to create new features without rewriting existing code. (React 2020.)

In the following chapter the core features of React will be explained briefly.

3.2.1 JSX

JSX stands for JavaScript XML and has similar syntax with XML or HTML with the opening and closing <> </> -symbols. JSX is an extension to the JavaScript language. The idea behind JSX is to allow developers to write HTML or code with similar syntax in JavaScript, which then is translated into pure JavaScript code. (Rajan 2017.)
As seen in Figure 1, HTML-tag `<h1>` is used inside JavaScript code directly. The example is a simple React component, which displays the text “Hello World” on the screen. In the code above the render function handles rendering of the visuals on the screen.

### 3.2.2 Functional Components

React components can be written either by using classes or as functional components, which are basically normal JavaScript functions.

```javascript
const HelloWorld = () => <h1>Hello, World!</h1>
```

According to official React documentation, developers should not use inheritance in React, thus using functional components makes code easier to read and understand. Functional components do not have a render() function. (React 2020.)

### 3.2.3 Lifecycle Methods and Hooks

Class-based React components have so-called ‘Lifecycle methods’, which would be the same as initialization or clean-up functions in other programming languages. As React supports functional components, React team has created so called ‘Hooks’ that allows using the lifecycle methods without writing a class.
3.2.4 Props

Props, short for properties, are values that can be passed from a parent component to its child component. Props work in a similar manner as parameters in other programming languages.

![Image of code snippet]

**Figure 3. Functional component using props**

In Figure 3, a functional component is created on row 11. Above it, on row 6 the component is used inside a parent component. A prop called ‘name’ is used, and a string value of “Paul” is set to it. Below, in the child component definition on row 11, props are received as parameters. On row 12, the value inside the prop ‘name’ is accessed with `props.name`, which returns the value of “Paul”.

To summarize the flow of data:

1. A String value “Paul” is passed onto a created prop ‘name’
2. The functional component receives props as parameter: `Hello = props`
3. Inside the functional component, the value is again accessed by writing `props.name`
3.2.5 State

State is used to handle the internal state data of the component or alternatively of the whole application. If the state of the component is changed, the component is re-rendered on the screen. Inside functional components, the state is implemented and modified using a React feature called Hooks.

In order to create the state, the **useState()** hook must be used. In order to update the state, the **useEffect()** hook must be used. The **useEffect()** hook is used for initialization, updating and clean-up functions. Whenever changes are made, the **useEffect()** hook is used.

```jsx
import React, {useState, useEffect} from 'react'

const ExampleComponent = props => {
  return {
    <Hello name="Paul" /> // Using the functional component
  }
}

function Hello(props) {
  const [nameInState, setNameInState] = useState(''); // State of the component
  useEffect(() => {
    setNameInState(props.name) // Changing the state of the component
  },[props.name]) // Setting dependencies, when the effect should be ran
  return <h1>Hello, {nameInState}</h1> // Rendering name to screen
}

export default ExampleComponent
```

Figure 4. Functional component using state and props

In order to use the State Hook, **useState** must be imported from the React library as shown in Figure 4 on row 1. After this, a state is created on row 10. The state contains two variables in an array, the first being a variable that contains the data of the current state. In this case the variable is called ‘nameInState’. The second variable in the array is used to replace the old state data with new state data. In this case the variable is called ‘setNameInState’. The state is initialized to be an empty string at the end of row 10.
Hello, Paul

Figure 5. React app in browser

On the first render cycle, in the place of the name ‘Paul’ is an empty string. However, in the code inside of the `useEffect()` function on row 12, the state is changed into a new value, which in this case is ‘Paul’. As mentioned before, once the state of a component changes, this will cause the component to re-render. As shown in Figure 5, once the React web application is executed, it displays the data passed down as a prop.

3.2.6 Redux and global state

The state of a component is private and is not accessible from outside of the component. If data needs to be passed into a component, using props is required. This, however, would cause complexity in larger applications with many different components and states. Instead of passing values as props between components, an open-source library called Redux is used. Redux allows creating a global state that holds all of the state data used by the application. React components are able to tap into the global state and get the required data. If any component changes the data in the global state, the update is passed on to all components that are using the same data. (Redux 2020.)
As shown in Figure 6 above, the data in the application flows in one direction. As the state and data are stored only in one place and the data flows only in one direction, tracing bugs and errors can be done more easily.

Redux works using actions and reducers. An action is a payload of information sent to a reducer. Reducer is a JavaScript function containing a switch case structure where every case corresponds to each action created. The reducer updates the data in the global state, which then is redistributed to the components requiring the data. (Redux 2020.) An illustration of how actions, reducers and global state work can be seen below in Figure 7.
3.3 React Native

React Native is a hybrid mobile application development framework created by Facebook. React Native uses React as a core technology, allowing mobile applications to be developed in a similar manner to websites or web applications using JavaScript. As React Native is built upon React, all of its features are also available.

3.3.1 Native Rendering components

The runtime performance of React Native applications comes from the use of native rendering APIs and UI components such as views, buttons and lists to mention a few. The development team behind React Native have created ready-to-use building blocks for developers that are compiled into corresponding counterparts of the UI elements for both the iOS and Android operating system. Instead of HTML tags which are used in web development, JSX is used in React Native development.

For example, when a developer needs to use a button in their application, a ready-made component can be used by writing `<Button>`. When JavaScript code is compiled into an application, the button component is translated into a native button component for both operating systems. (See Figure 8.)

Another example is React Native’s `<View>` component. During compilation the View is translated into UIViewController for iOS and into Fragment for Android. Both UIViewController and Fragment are classes that are responsible for the representation of graphical elements, the UI itself. This approach allows writing the UI using JavaScript and JSX which in the end gives the application a native look and feel. (Eisenman 2020, React Native 2020.)

![Figure 8. React Native application compilation](image-url)
3.3.2 React Native Bridge

By default, iOS and Android operating systems cannot run applications written in JavaScript. A common idea for developers to work out the differences in code would be to translate JavaScript into Objective-C and Java. This however is troublesome due to the fact that both native programming languages are strongly typed while JavaScript is not. (Gaba 2020).

Instead of translating the code, React Native uses JavaScriptCore and React Native Bridge under the hood. JavaScriptCore is an open source JavaScript engine that is natively used on iOS for running JavaScript in Safari browser. When React Native app is compiled for Android, the engine is bundled into the application together with the source code. React Native Bridge communicates between the JavaScript code and native code during the execution of the application. (Gaba 2020, Frachet 2020.)

When the application gets executed, the native code starts the JavaScript engine and thus the JavaScript thread. The business logic written in JavaScript is then executed on the JavaScript thread by the engine, which will make asynchronous calls through the Bridge to the main, native thread, that will carry out the tasks given. Once the native thread has done a given task, a response is sent back using the Bridge. (Gaba 2020.)

Because the communication between the two executions are asynchronous, there is no blockage, in other words no waiting period in the execution between tasks. This allows better performance when rendering the view. By contrast, if the calls would be synchronous instead of asynchronous, the execution would have to be paused on one thread while waiting for the other thread to execute the task and to respond. This would cause unresponsiveness when using the application.
As illustrated in Figure 9, the division of React Native application and operating system can be seen. The React Native application has the business logic written in JavaScript and a JavaScript engine (JavaScriptCore) to execute the code. It also has the React Native Bridge which communicates between the React Native application and the mobile operating system.

3.3.3 React Native components

React Native provides developers the basic UI components which can be used to build more complex components. Components are written in JSX. In the following section the basic components will be explained briefly.

<View> is the most fundamental component. It acts as a wrapper to other components similar to <div> in HTML

<Text> is used for displaying text

<Image> is used for displaying an image

<TextInput> is used for text input field, which allows inputting text in the application by using on-screen keyboard

<ScrollView> provides a scrollable container, which can host other components
<StyleSheet> is used for writing styles similar to CSS

<Button> is a basic button component which recognizes touch

<FlatList> is a component for rendering a lazy-loading list

<ActivityIndicator> displays a platform specific loading-animation indicator

<Alert> displays a platform specific alert

<Touchable> and <TouchableWithoutFeedback> are used for creating custom button components, or components that require touch recognition features (React Native, 2020.)

Figure 10. Basic React Native components in iOS

In Figure 10, some of the basic components provided by React Native are shown. The Text input component and image component required using some styles to make them visible on screen. Each component is wrapped in a View component, which does not show visibly on screen but allows to create spacing between the components.
3.3.4 Custom components

The idea in React Native is to merge basic components into more complex components. A basic custom component skeleton can be used in the beginning of creation of any component.

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

const CustomComponent = props => {
  return {
    <View style={styles.container}>
      <Text>Text</Text>
    </View>
  }
}

const styles = StyleSheet.create(
  {
    container: {
      flex: 1,
      alignItems: 'center',
      justifyContent: 'center'
    }
  }
)

export default CustomComponent
```

Figure 11. React Native component skeleton with styles

As React Native uses React, in Figure 11 on row 1 an import is required. React Native components are imported using the syntax seen on row 2. On row 4 a functional component is created. Inside the return statement, the imported React Native components can be used to create a view. On row 12 a stylesheet object is created. Inside the curly braces of the object, individual style objects can be created and styles with similar syntax to CSS can be written. On row 6 the created style object called ‘container’ is passed to the view component through a style prop. On row 16 the component is exported. The name of export has to match the name of the component. This way the component can be imported into any other JavaScript file used in the project.
4 THE SAWMILLEVENTS APPLICATION

The SawmillEvents application is a hybrid application designed to be used in the conference events organized by The Finnish Sawmills Association. The initial idea of the application is to provide information about the event such as programme, other participants and their contact information, logos of the sponsors, speakers and their presentation topics, maps of the venue, general information about the event itself and also to serve as a way to send feedback.

The application was designed to use a client–server architecture. The frontend of the application was created by using React Native. The backend was created by using Node.js, Express and MongoDB. The backend was created to store the data that the mobile application would fetch and display.

As seen below in Figure 12, the iOS and Android versions of the mobile application are using the HTTP GET request to fetch the data from the server. The server hosts a MongoDB database that is used to store and query the requested data.
4.1 Planning

The first step in this thesis project was to arrange a meeting with a representative of the Finnish Sawmills Association and to gather the requirements for the application.

The requirements of the application were the following:

- Display data about programme, participants, sponsors, speakers, general information, maps of the venue and to provide a way to send feedback.
- Only the pre-registered participants would be able to log into the application and view data.
- The application would be published on Google Play and possibly App Store.

The screens were designed and drawn on paper displaying the required UI elements and interactions, for example how the end-user would navigate between screens in the application.

4.2 Development

4.2.1 Setting up the environment and the project

Setting up the React Native environment requires the installation of the following dependencies: Node.js, npm, Python and JDK8 (Java Development Kit 8). Each dependency can be downloaded and installed from their official websites.

In order to build and test the Android version of the application, installation of Android Studio is required. For the iOS version of the application, a Mac computer with the installations of Xcode, Xcode CLI and CocoaPods are required. As Xcode, Xcode CLI and CocoaPods are not compatible with other operating systems, iOS development can only be done on a Mac computer running OSX.

To sum up all required installations:

- Node.js and npm
- Python
- JDK8
- Android Studio
- Xcode, Xcode CLI and CocoaPods (Required for iOS)
- React Native CLI via npm
Once Node.js and npm are installed, the following command was used to install the React Native CLI (Command Line Interface) as seen in Figure 13.

![Figure 13. Installation of React Native CLI](image)

Next, the CLI was used to create the React Native project as seen in Figure 14.

![Figure 14. Creation of the React Native project](image)

### 4.2.2 Adding version control

The chosen version control system for this thesis project was Git. GitHub was used to store the git repository containing the project.

In order to make workflow easy with version control and to keep the project folders logical, the folder containing the React Native project and frontend code was named `frontEnd`. Following the same naming convention, the folder containing backend code was named `backEnd`. A folder containing both `frontEnd` and `backEnd` folders was pushed to Git. This way the whole project would be backed up using a single Github repository.

### 4.2.3 File structure of the project

Once the project was set up, an appropriate folder structure was created for the required elements used in the application. Below, in Figure 15, the file structure of the project can be seen. In the following section, each folder and its purpose will be explained briefly.
Figure 15. File tree of the project
Starting from top:

_Tests_ folder was automatically generated by React Native but was not used in this project.

**Algorithms** folder was created to contain JavaScript files containing algorithmic logic, if required.

**Android** folder was autogenerated by React Native. It contains the Android version of the project. The Android folder can be opened with Android Studio, and the Android version of the application can be executed on either physical device or emulator.

**Assets** folder contains imported images and fonts.

**Components** folder contains reusable components, such as card component, which is used universally. The folder also contains multiple renderItem components, which are used by different List components.

**Constants** folder contains JavaScript files which are used to describe constants used throughout the application, for example global colors.

**Data** folder contains JSON files used for testing different components, which are rendering lists based on the JSON data.

**Ios** folder was autogenerated by React Native. It contains the iOS version of the project. The folder can be opened with Xcode and the iOS version of the application can be executed on either physical device or simulator.

**Models** folder contains JavaScript files describing data models.

**Navigation** folder contains EventNavigator.js file, in which the navigation hierarchy of the application is written.

**Node_modules** folder contains all JavaScript libraries and dependencies installed by npm.

**Screens** folder contains JavaScript files for each view, or screen. The screens are used by the navigation to allow the end-user to move / navigate from one screen to another.

**Store** folder is used together with Redux to handle the global state and data of the whole application.
4.2.4 Constants

The constants folder contains the Colors.js file, which specifies the color scheme of the application. As shown in Figure 16, the constants are given names and hexadecimal values. If the colors defined by hexadecimal values are changed in the file, the changes are applied everywhere, where the Colors.js file is imported, and the constants are used.

Figure 16. The color scheme of the application
4.2.5 Creation of screens with FlatList

Each screen was created using a React Native component skeleton. See Figure 17. As screens are simply React Native components, the same component skeleton can be used in creation of screens and reusable components. Screens act as a parent to all reusable components and are themselves used by the navigation.

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

const CustomComponent = props => {
    return (  
        <View>  
            <Text>{props.text}</Text>  
        </View>  
    )
}

const styles = StyleSheet.create({})

export default CustomComponent
```

Figure 17. React Native component skeleton

Most of the screens are lists which display the corresponding data. As all screens containing a list are built in a similar manner, only one example is represented in this thesis.
Figure 18. Programme Screen component

In Figure 18 the component skeleton is turned into a screen component, which will display a list on the screen. On row 4, test JSON data is imported that is later to be used by the list to display the data. On row 5, a custom component is imported, which will be used to display the data inside the list.

The FlatList component requires three props in order to function correctly:

1. data prop on row 11 is used to pass the JSON data for the list.
2. keyExtractor prop on row 12 is not necessary but is used to avoid warnings. In this case it takes the index of an item in the imported data to be used as a unique key in the list.
3. renderItem on row 13 is used to return a React Native component for each data object in the imported data.

On row 14, the imported custom component is used for this list. Inside the component, seven different props are defined, each corresponding to a data field in the used JSON file. Below in Figure 19, the data about the programme is shown. Some of the fields in the data may be empty, as there are breaks and intermissions in the programme during which there are no speakers.
Figure 19. JSON data for Programme component

The FlatList component will iterate over every object in the JSON data, passing the data through props into the ProgrammeItem component, which has placeholders for the data.

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

const ProgrammeItem = props => {

    const time = props.time;
    const location = props.location;
    const description = props.description;
    const speaker = props.speaker;
    const titleOfSpeaker = props.titleOfSpeaker;
    const specialTitleOfSpeaker = props.specialTitleOfSpeaker;
    const companyOfSpeaker = props.companyOfSpeaker;

    if (time && location && description && speaker && titleOfSpeaker && specialTitleOfSpeaker && companyOfSpeaker) {
        return (
            <View style={styles.card}>
                <View style={styles.timeContainer}>
                    <Text>{props.time}</Text>
                </View>
                <View style={styles.locationContainer}>
                    <Text style={styles.location}>{props.location} </Text>
                </View>
                <View style={styles.descriptionContainer}>
                    <Text style={styles.description}>{props.description} </Text>
                </View>
                <View style={styles.speakerContainer}>
                    <Text style={styles.speaker}>{props.speaker} </Text>
                </View>
                <View style={styles.titleOfSpeakerContainer}>
                    <Text style={styles.titleOfSpeaker}>{props.titleOfSpeaker} </Text>
                </View>
                <View style={styles.specialTitleOfSpeakerContainer}>
                    <Text style={styles.specialTitleOfSpeaker}>{props.specialTitleOfSpeaker} </Text>
                </View>
                <View style={styles.companyOfSpeakerContainer}>
                    <Text style={styles.companyOfSpeaker}>{props.companyOfSpeaker} </Text>
                </View>
            </View>
        )
    }

    return null;
}
```

Figure 20. ProgrammeItem component

In Figure 20 from row 5 till row 11, the values carried in props are passed into local variables. In the JSON data used, some of the fields are empty on purpose. In order to
avoid rendering blank elements on the screen, a comparison is done to render only the fields which contain data. On row 14, a comparison is made to check if variables \textit{time}, \textit{location} and \textit{description} contain values and if the rest of the variables are empty. On rows 18, 22 and 23 the values of \textit{time}, \textit{location} and \textit{description} are used for rendering.

To summarize the flow of data:

1. JSON file contains the data.
2. JSON file is imported to ProgrammeScreen component which contains a FlatList component.
3. Data is passed to FlatList component through a prop.
4. FlatList component iterates over the data.
5. For each object in the data a ProgrammeItem component is used, and the data entries are passed to the component through props.
6. Inside ProgrammeItem component, the data is rendered conditionally.

A Similar way of using the ListView component together with the renderItem component was also used for \textit{Participants}, \textit{Speakers} and \textit{Sponsors} screens. The other screens, which would not show a list, did not require the creation of a renderItem component. However, other custom components were used in these screens instead.
Figure 21. Finished Programme Screen

Figure 21 shows a screen which displays the programme of the current event. The containers in the list are rendered conditionally, depending on whether the data fields are empty or not. This causes the containers with less information to appear smaller and those with more information to appear larger.
4.2.6 Main Navigation Screen

As the main navigation of the application was requested by the customer to be built from scratch, the MainNavigationScreen component was created. It was implemented in a similar manner to other screens utilizing FlatList and renderItem components with the difference that each renderItem component on the screen would be a clickable button that would navigate further into the application.

```javascript
import React from 'react'
import { View, Text, StyleSheet, FlatList, Dimensions, Platform } from 'react-native'
import NavigationTile from '../components/NavigationTile'
import Colors from '../constants/Colors'

const numberOfColumns = 2

const naviScreenData = [
  { id: 1, title: 'Programme', link: 'Programme', icon: Platform.OS === 'android' ? 'md-calendar' : 'ios-calendar' },
  { id: 4, title: 'Participants', link: 'Participants', icon: Platform.OS === 'android' ? 'md-contact' : 'ios-contact' },
]

const MainNavigationScreen = props => {
  return {
    <View style={styles.listContainer}>
      <FlatList
        data={naviScreenData}
        numColumns={numberOfColumns}
        renderItem={(itemData) =>
          <NavigationTile
            title={itemData.item.title}
            icon={itemData.item.icon}
            navigationLink={itemData.item.link}
          />
        }
      />
    </View>
  }
}
```

Figure 22. MainNavigationScreen component

In Figure 22, a MainNavigationScreen component is created on row 19. Above in the code on row 7, a constant `numberOfColumns` is given the value of 2. This value is used in the FlatList component to create two columns instead of one. From row 8 till row 16, an array called naviScreenData is created. This array contains hardcoded data and that is used by the FlatList component to render the buttons on screen.
Above in Figure 23, a custom button component is created that will be used in the MainNavigationScreen. On row 10, a custom button component is defined. This component is used on the iOS version of the application. On row 11, if the operating system is Android with a platform version over 21, a different custom button component is used. This allows to dynamically switch the type of button used by the application to enable certain visual effects depending on whether the operating system executing the application is iOS or Android.

On row 26, a custom Card component is used to give the button a certain visual look. On the same row, the Card component is given a style through a prop. This style enables adjusting the size of the button according to the screen size of the device running the application. On row 27, a component called TouchableComponent is used to enable touch recognition. On row 29, an onPress prop is defined that allows calling a function when a touch is recognized on the button. On row 30, a navigation function is fired, which
navigates the application to a new screen depending on which button was pressed. The navigation link used to define the screen to which the application will navigate, was passed as a prop from the MainNavigationScreen component.

From row 34 till row 37, a third-party library is used in order to use icons in the buttons. Props name, size and color are required to display the icons. The developer needs to find the proper names for the icons from a corresponding website. On row 35, the name of the icon is passed as a prop from MainNavigationScreen. On row 36, the size of the icon is calculated dynamically according to the screen size of the device running the application. On row 37, the color of the icon is defined to match the color scheme defined in the constants folder.

```
const styles = StyleSheet.create(
  {
    invisibleTile: {
      flex: 1,
      margin: 15,
      height: Dimensions.get('window').width / 100 * 45,
    },
    tile: {
      flex: 1,
      margin: 15,
      borderRadius: 20,
      height: Dimensions.get('window').width / 100 * 45,
      aspectRatio: 1,
    },
    touchable: {
      height: Dimensions.get('window').width / 100 * 45,
    },
    contentContainer: {
      flex: 1,
      justifyContent: 'center',
      alignItems: 'center',
    },
  }
)
export default withNavigation(NavigationTile)
```

Figure 24. StyleSheet object with dynamic styles

Above, in Figure 24, on row 47 a stylesheet object is created. Inside this object on row 57, the height of the button is calculated using the width of the window of the physical device. On row 58, the aspect ratio of the button is set to 1. This combination allows to create square buttons, which have an ability to scale regarding to the screen size. In order to call the navigation function from inside of the created button, the component needs to be exported in a certain way. On row 70, the NavigationTile component is exported with navigation.
Figure 25. Finished Main Navigation Screen

Figure 25 shows a screen, which displays the main navigation of the application. As specified earlier the buttons are rendered in two columns and the size of each button and icon is calculated according to the device’s screen size. This way the UI looks the same in any device that would run the application.
4.2.7 Other finished screens

Figure 26. Finished Speakers Screen

Figure 26 shows a screen, which displays the speakers of the current event. The screen uses the FlatList and renderItem architecture to achieve the current visuals. Additional to this, an Image component was used to display the corresponding image of the speaker.
Figure 27 shows a screen, which displays the sponsors of the current event. The screen uses the FlatList and renderItem architecture to achieve the current visuals. The Card component with nested Image and Touchable component is used as a renderItem, which allows to turn each element into a touchable button. By touching the Card component, a web browser is opened, and the end user is redirected to the website of the corresponding sponsor. This is done using a third-party library called React-Native-Communications.
Figure 28 shows a screen, which displays the participants of the current event. The screen uses the FlatList and renderItem architecture to achieve the current visuals. Additionally, on the customer’s request a search functionality was implemented using a third-party search bar component from the React-Native-Elements library. The regular JavaScript `filter()`, `includes()` and `sort()` functions were used to create the searching and filtering logic. When the end user types text into the search field, the results which do not match the criteria will be filtered away and the list will only display the participants whose first name, last name or company does match to the search text typed.

Furthermore, the phone number and email are Touchable components that allow end users to tap on them. When a phone number is tapped, the end user is prompted whether they want to call the number. If the email is tapped, the default email client of the phone is opened. If email settings are defined, a new email is created with the recipient field auto-filled with the tapped email address. This is done using a third-party library called React-Native-Communications.
Figure 29 shows a screen, which displays the venue of the current event. In order to implement the tab view functionality with zoomable images, several third-party libraries were used. The React-Native-Tab-View library was used to implement the tab view and the React-Native-Image-Pan-Zoom library was used to make images support pitch-to-zoom functionality.
Figure 30. Finished About Screen

Figure 30 shows a screen, which displays general information on the current event. The component displays static text using the Text component. Touchable components are used to create the hyperlinks. By tapping the hyperlink, a web browser is opened, and the end user is redirected to the corresponding website. In the same way as in the Participants screen, once the email is tapped, a new email is created with the recipient field automatically filled by the tapped email address.
Figure 31 shows a screen, which displays a button to send feedback. This button component is created similarly to the NavigationTile component used in the MainNavigationScreen. Like in the Participants and About screens, by tapping the button, a new email is created with the recipient field prefilled by the event organizer’s email address.
4.2.8 Navigation

In order to navigate between screens, a third-party library called React Navigation was used. React Navigation provides simple way to build a navigation structure for the application and handles the native navigation animations for both platforms. React Navigation was installed using npm.

```
import { createAppContainer, createSwitchNavigator } from 'react-navigation'
import { createStackNavigator } from 'react-navigation-stack'
import { Platform } from 'react-native'
import Colors from './constants/Colors'
import LoginScreen from './screens/user/LoginScreen'
import MainNavigationScreen from './screens/event/MainNavigationScreen'
import SelectEventsScreen from './screens/user/SelectEventScreen'
import AboutScreen from './screens/event/AboutScreen'
import ParticipantsScreen from './screens/event/ParticipantsScreen'
import ProgrammeScreen from './screens/event/ProgrammeScreen'
import SpeakersScreen from './screens/event/SpeakersScreen'
import SponsorsScreen from './screens/event/SponsorsScreen'
import VenueScreen from './screens/event/VenueScreen'
import FeedbackScreen from './screens/event/FeedbackScreen'
```

Figure 32. Importing screen components for Navigation

On row 1 of the JavaScript file, the used libraries are imported as shown in Figure 32. From row 7 till row 17, all of the screens created are imported to be used in the navigator.
Figure 33. Navigation using React Navigation

The navigator itself is a JavaScript function, which takes a JavaScript object as an argument. In Figure 33, inside the JavaScript object, from row 50 till row 58, the imported screens are defined as key-value pairs.

In this project two different navigator types were used; a SwitchNavigator and a StackNavigator. SwitchNavigator is used when navigation is allowed to happen only in one direction. In this application, for example, after a successful login the end-user is navigated into the next screen, but the user is not able navigate back to login screen.

StackNavigator is used in all other cases of navigation. It keeps all screens in a stack and displays only the topmost on the screen of the device. When the user navigates around in the application, the screens are pushed onto the stack and popped from it.

As seen in Figure 33, on row 48, a constant called EventNavigator is a stack navigator object, which has all of the imported screens. The screens are in the stack in same order as shown in the image, topmost being the SelectEventScreen, displayed on row 50.

At the end of the file on row 65, createAppContainer() function is called. This function takes a navigator as an argument. Whichever navigator is passed as the argument is shown on the screen of the device once the application is executed.
4.2.9 Login

As this mobile application was developed faster than the backend, the implementation of login was made locally. The application contains the data about participants which can be used for login.

```javascript
const LoginScreen = props => {
  const [inputEmail, setInputEmail] = useState('')
  const [inputPassword, setInputPassword] = useState('')
  const [isLoading, setIsLoading] = useState(false)
  const eventPassword = 'WFFC2020'

  // Check if user exits
  setIsLoading(true)
  for (const object of participantData) {
    if (object.Email === inputEmail && eventPassword === inputPassword) {
      setLoading(true)
      loginFailed = false
      console.log('Authentication success!')
      props.navigation.navigate('EventNav')
      return
    } else {
      setLoading(false)
      loginFailed = true
    }
  }
  if (loginFailed == true) {
    Alert.alert('Login failed', 'Incorrect login credentials', [{ text: 'Okay' }])
  }
```

Figure 34. State and login variables

In the login screen, the end-user is asked to give email and password as credentials for the login. As seen in Figure 34, a local state was used to contain the data about the given email and password on rows 11 and 12. Also, the password that would allow the login was set for the event on row 14.

Figure 35 contains the code used in the login function. The participant data was imported in JSON format at the beginning of the file. Once the end-user taps the login button, the login function is fired. On row 39, the function iterates through the list of participants and on row 40, checks that if the given email can be found and that if the given password matches the password set for event.

The defined isLoading state is used for displaying an ActivityIndicator component. By default, the state is false, which hides the ActivityIndicator component. On row 38, before a loop that iterates over the participant data starts, the isLoading state is set to true. This causes the login screen to re-render and the ActivityIndicator to be visible as long as the
loop runs. This way, if for any reason the login takes longer than expected, feedback is provided for the end-user that the login process is ongoing.

On row 44, if the email and password match, the user is navigated to the next screen from login screen. If not, a variable called loginFailed is set to true on row 48. On row 52, this variable is used to create an alert to let the end-user know that the credentials provided were incorrect.

Figure 36. Finished Login Screen

Figure 36 shows a screen, which displays input fields for email and password and a button to login. If the given credentials do not match, the end user will be prompted that the credentials are incorrect. Given correct credentials the end user will be navigated to the main navigation screen.
4.2.10 HTTPS requests and backend integration

Until this point, the application has been using local JSON files. As the frontend components were verified to function properly using the local data, the same data was fetched from the server using HTTPS GET request. HTTPS stand for Hyper Text Transfer Protocol Secure. Unlike normal HTTP request, HTTPS transfers data using encrypted connection. It is required by both iOS and Android to use HTTPS in the applications created.

In order to fetch and distribute the data inside the application in organized manner, a global state was created. The global state acts as a container which stores the current state of the application and the data fetched from the backend. From the global state, each frontend component can fetch the data it requires. If the data in the global state is changed, it is provided again for each component using that data.

To create the global state, a JavaScript library called Redux is used. It is installed using npm. Along with Redux, a middleware called Redux Thunk is also installed using npm. Redux works using actions and reducers. Actions and reducers however are synchronous which makes them incompatible with asynchronous HTTPS requests. This issue is solved by using a JavaScript library called Redux Thunk. Redux Thunk allows to make HTTPS requests inside actions. Only after the data is received from the server is the action holding the fetched data dispatched to a reducer.

Figure 37. Action with HTTPS request
In Figure 37, on row 6 a function for fetching data is created. On row 8 a HTTPS request is made, and the results are stored in a variable. The `await` keywords are used to wait for the response from the server to arrive before executing the code further.

As the data from server is not necessarily automatically in the same format as what is required by the components inside the application, a data model is used to create new objects using the data from the response.

On row 11 a new array is created to hold the created data objects. From row 13 till row 18 the data inside the response is iterated over and the new data objects are created and pushed into the new array.

At the end on row 24 the action along with the new array of data objects is dispatched to a reducer.

In Figure 38, on row 1 the action created in previous file is imported. From row 4 till row 6 the global state is created. On row 8 a reducer is created which uses the initial global state and receives actions as arguments. On row 10, once an action with the given name is received, the code below gets executed. On rows 12 and 13 a new state is returned that will update the global state with the fetched data, that was carried over by the action dispatched earlier.

---

```javascript
import { SET_EVENTS_METADATA } from './actions/eventData'

// Set the data as initial state
const initialState = {
  eventsMetaData: []
}

const eventDataReducer = (state = initialState, action) => {
  switch (action.type) {
    case SET_EVENTS_METADATA:
      console.log('entering Set events case')
      return {
        eventsMetaData: action.eventsMetaData
      }
    default:
      return state
  }
}

export default eventDataReducer
```

---

Figure 38. Reducer for event data
In order to use the global state in the application, the starting point of the application needs to be modified inside the app.js file.

```javascript
import React from 'react'
import { createStore, combineReducers, applyMiddleware } from 'redux'
import { Provider } from 'react-redux'
import ReduxThunk from 'redux-thunk'
import EventNavigator from './navigation/EventNavigator'
import eventDataReducer from './store/reducers/eventData'

// Add additional reducers to here, to use in the app
const rootReducer = combineReducers(
  eventData: eventDataReducer
)

const store = createStore(rootReducer, applyMiddleware(ReduxThunk))

export default function App() {
  return(
    <Provider store={store}>
      <EventNavigator />
    </Provider>
  )

}
```

Figure 39. The starting point of the application

In Figure 39, from row 2 till row 4 the required imports to use Redux and Redux Thunk were added. On row 7 the reducer that was created earlier is imported. On row 10 a constant called rootReducer is created that hold the results of a combineReducers() function. If the application would have more reducers, this function would combine them all together for use. On row 14 a Redux store is created. The store contains the global state of the application and uses all of the reducers combined in order to update the global state. It also takes ReduxThunk variable as an argument in order to use it inside the actions.

On row 16 is the starting point of the created React Native application. Normally it returns only the Navigator component that holds all of the screen components of the application. To use the Redux store, the navigator is wrapped with a Provider component and the previously created store constant is passed through a prop. As the navigator contains all of the screen components, and the screen components are built using smaller components, the whole application gains access to the Redux store, and thus the global state created.
Above in Figure 40, the flow of data of the whole application is visualized. The data used in the application is stored in the global state from which it flows down to nested screens and components.
4.3 Publishing and distribution

The main publishing platform for iOS is App Store and Google Play for Android. As Android is an open source platform, it allows alternative distribution methods such as downloading and installing the application from a website or from an installer send as an email attachment. To publish the application in Google Play requires less effort compared to Apple’s App Store that has much more strict requirements for publication.

4.3.1 Publishing for Android

To publish the application to Google’s Play Store the developer needs to take into account of the following requirements.

- Paid developer account with a single fee of 25$
- Application must not crash at any point during execution
- Application needs to be compatible with recent Android versions
- If application has a login, a test account should be provided

In this thesis project as the application was to be published under the name of the Finnish Sawmills Association, the developer account was created by the customer. After it was decided that the first version of the SawmillEvents application was good enough, it was released and published on Google Play store.

For the application to be released on Google Play store, it is required to be signed with an upload key. This is done using a command line tool called keytool. It is provided by installation of JDK8.

A keystore and required keys were generated using the following command shown by Figure 41.

```
Macbook-Pro macbook$ keytool -genkeypair -v -keystore suomensahat.keystore -alias suomensahat -keyalg RSA -keystore 2048 -validity 18080
```

Figure 41. Keystore generation
Once the command was executed, the generated suomensahat.keystore file was placed into android/app directory and the gradle properties of the application were edited for the application to use the keystore file as shown in Figure 42.

```
# Project-wide Gradle settings.
# IDE (e.g. Android Studio) users:
# Gradle settings configured through the IDE will override
# any settings specified in this file.
#
# For more details on how to configure your build environment visit
# http://www.gradle.org/docs/current/userguide/build_environment.html
#
# Specifies the JVM arguments used for the daemon process.
# The setting is particularly useful for tweaking memory settings.
# Default value: -Dxms1024m -Dxmx512m
# org.gradle.jvmargs=-Xmx2048m -XX:MaxPermSize=256m
# When configured, Gradle will run in incubating parallel mode.
# This option should only be used with decoupled projects. More details, visit
# http://www.gradle.org/docs/current/userguide/multi_project_builds.html#sec:decoupled_projects
# org.gradle.parallel=true
android.useAndroidX=true
android.enableJetifier=true

MYAPP_UPLOAD_STORE_FILE=suomensahat.keystore
MYAPP_UPLOAD_KEY_ALIAS=suomensahat
MYAPP_UPLOAD_STORE_PASSWORD=********
MYAPP_UPLOAD_KEY_PASSWORD=********
```

Figure 42. Android application gradle properties

Finally, to use the generated keystore the following code shown by Figure 43 was added to the android/app/build.gradle file.

```
release {
    if (project.hasProperty('MYAPP_UPLOAD_STORE_FILE')) {
        storeFile file(MYAPP_UPLOAD_STORE_FILE)
        storePassword MYAPP_UPLOAD_STORE_PASSWORD
        keyAlias MYAPP_UPLOAD_KEY_ALIAS
        keyPassword MYAPP_UPLOAD_KEY_PASSWORD
    }
}
```

Figure 43. Using keystore in Android application

To build a release version of the React Native application, the following command seen in Figure 44 was used.

```
MacBook-Pro:frontend macbook$ ./gradlew bundleRelease
```

Figure 44. Building a release of Android application
This command generates .abb file, an Android app bundle file, which contains the .apk file, an Android application Package file. The .abb file is a distribution file used for uploading the application to Google Play while .apk file is an installation file used by Android operating system to install the application to the device.

The portal for uploading the application can be found at play.google.com. Once logged in with the developer account, an application to release the built mobile application was created inside Google Play Console as seen in Figure 45.

Figure 45. Applying to release the mobile application

Next step was to upload the .aab file to the Store. To achieve this, a new App Release was created in the Google Play Console. Every time the application is to be updated a new release needs to be created which will replace the old release of the published application.

Figure 46. Creating a new release on Google Play

A new release was created by clicking the CREATE RELEASE button as seen in Figure 46.
The New release page allows to drag and drop the generated .aab file, which uploads the application to Google Play store’s servers. In the same page the name of the release must be specified with the notes of what is included in the release.

Additionally, the following information and files were also required for the application to be released in Google Play store:

- Title of the application
- Short description
- Full Description
- Graphical assets (icon of the application)
- Two screenshots from a real device running the application
- Feature graphic (shown as a banner in Google Play store)
- Application type and category
- Content rating (autogenerated after an inquiry form is filled)
- Email address for contact
- Privacy Policy (if the target audience of the application are adults, this is not required)
- Countries from where the application can be downloaded
- If the application contains advertisements or not
- Verification that the application meets Android Content Guidelines
- Verification that you acknowledge that the application may be subject to US export laws as the application is to be uploaded to servers residing in the United States

After the required fields were filled and the required files provided the application was ready to be published.
Figure 47. Released Application in Google Play store

Once the publication process was completed after several days, the application was successfully released and has since been available in the Google Play store as shown by Figure 47.
4.3.2 Publishing for iOS

One of the goals of this project was to release the application for both iOS and Android. As mentioned earlier, the first release for Android was done successfully. However, Apple is much more strict with the requirements of the applications to be released and the review process takes longer. Even when the UI of the application is working as intended, the first iOS version of the application is still under development to match all of the required criteria.
5 RESULTS AND CONCLUSIONS

The goal of this thesis was to produce a working hybrid application for a customer, using React Native. The application was meant to be used for the first time in the Wood From Finland 2020 event organized by The Finnish Sawmills Association in February 2020. The application was required to work on both iOS and Android operating systems, to have a responsive UI, that allows the application to be used on any sized device, to fetch and display data and images stored on a backend server and to be published to an official application distribution platform such as Google Play and App Store.

This chapter summarizes how the goals were reached, as well as reflections and thoughts of what could have been done differently, how the application could be developed further and what React Native was like to learn and to use for development.

The SawmillEvents application was built by following different tutorials and applying what I learned on the way. The actual development of the application was conducted in the order described in this thesis. As a result, the first version of the SawmillEvents application was created.

The UI of the SawmillEvents application was implemented in a way that allows the application to calculate the size of the displayed graphical elements based on the actual screen size of the device. This way the UI is responsive and will work on any mobile device regardless the screen size. As the idea of React Native is to use only one code base to create both the iOS and Android versions, the SawmillEvents application will work on any modern iPhone or Android phone. As the UI was designed to look similar on both iOS and Android versions, not too much platform specific code was required during the development phase.

The data used in the SawmillEvents application was handled by creating a server that holds the data and allows the mobile application to fetch the data. This way the application always gets fresh copy of the data. If the data needs to be changed on the server, the application would then fetch a new version of the data and display the changes. Inside the application, the data is stored in a global state, from where it is distributed to different components of the application. This internal structure of the application allows to display data about different events if such data is provided to the server.

The first version of the application was released in the Google Play store. The iOS version of the SawmillEvents application is planned to be released once further development reaches the point that the application matches the requirements of the App Store.
The first version of the application was used in the Wood From Finland 2020 event organized by the Finnish Sawmills Association. The participants of the event were encouraged to download and use the application. The feedback on the application was received during the event directly from the participants and through the feedback section of the application itself. The feedback given was appreciative and positive.

According to the customer the features implemented in the SawmillEvents application fulfilled the purpose and goals of the project. Additionally, the visuals of the mobile application matched the graphical assets (website and handouts) of the conference event. This allowed the application to be easily associated with the event. The graphical elements of the application were easy to understand, and the user interface was easy and logical to use. The deployment of the Android version of the SawmillEvents application was simple and the application worked as intended right out of the box.

If a similar project would be conducted again, I would not change much regarding how this project was carried out. The structure of how the application should be built and the different work phases introduced in this thesis were found to be effective as they removed unnecessary bottlenecks or other ‘what to do next’ problems during development.

Regarding future projects, I would put more focus on learning backend development. As most mobile applications have a backend, understanding how the backend works more profoundly is a valuable asset to any developer.

Some of the features requested by the customer were left out of this thesis project due to the time and effort required in order to make those features function as intended. For example, the implementation of maps, chat and a game were excluded from this project. The structure of how the application was built allows adding new features easily. The requested features could be implemented in future versions of the application in the way described below.

The implementation of maps would be the easiest to create as this could most likely be done by using a third-party library. However, using a map usually requires using an API (Application Programming Interface) provided by either Google, Apple or some other provider. Using these APIs is not necessarily free and may require a license depending on the type of the application where they are used in.

The implementation of a chat would require a lot more coding on the backend side. The frontend of the chat could be another screen component that would have a list of participants. By tapping a participant on the list, the application would navigate into a new screen that would show the chat between the end-user and the selected participant.
However, implementing a real-time functioning chat system would require storing all conversations between each participant to the backend, querying this data and providing it to the frontend in a continuous manner.

The amount of time and effort of implementing a game in React Native would highly depend on the type of a game requested. A quiz-type game could probably be created without too much effort. However, a game including advanced graphics, animations and sounds could be troublesome as React Native is not a game development platform. Regardless of the type of a game, the implementation would also require more coding on the backend side.

Learning and using React Native felt very similar to learning web development. If a developer is familiar with web development and React, the development phase in React Native can be intuitive and fast. For me, learning React Native meant learning React concepts at the same time. This was achieved by following online video tutorials, that allowed me to grasp the required concepts of React and React Native and to build the mobile application using the framework.

In this thesis project a hybrid mobile application was created successfully with the requested features. The application was published in the Google Play store and was used by participants at the Wood From Finland 2020 event as intended. The customer was satisfied with the results of the project. React Native proved to be fairly easy to learn and was found as a rich framework for creating a hybrid application similar to the SawmillEvents application.
REFERENCES


