Expertise
and insight
for the future

Aleksei Fomushkin

Practical application of advanced React Native concepts

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

30 September 2019

Author Title Number of Pages Date	Aleksei Fomushkin Practical application of advanced React Native concepts 35 pages 30 September 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Mobile Solutions
Instructors	Ilkka Kylmäniemi, Senior Lecturer

The purpose of this thesis is to briefly explain what ReactJS and React Native are and what new and advanced development techniques can be utilized in cross-platform mobile development. In practice, the techniques were applied when developing a functional application from start until finishing.

The study covers the key features of ReactJS and React Native and the reasons to choose the framework for developing the cross-platform application which was designed for the project described here.

The advanced concepts which were researched and implemented into the application source code are relatively new or not covered enough. They include TypeScript support for static type checking and better development flow in general, React hooks and Context API for a new way to manage the global application state, and Animated API for providing better user experience to users.

Due to constraints, some features such as audio support, background mode, native iOS and Android modules, performance optimizations and publishing of the applications are not dealt with in the thesis, but can be observed in the application itself and accessed through Apple and Google stores.

The final practical project, which utilized the React Native concepts, was successfully launched on both main mobile platforms. The concepts also allowed building and integrating new features, such as an alternative dark theme, faster and with less effort. Due to modern development techniques, it came out to be performant even on dated mobile devices with smooth transitions and animations.

Keywords	React Native, TypeScript, Hooks, Animation, Context API

Contents

List of abbreviations

UI	User Interface. Field of human-computer interaction, is the space where interactions between humans and machines occur.
CLI	Command-Line Interface. Means of interacting with a computer program where the user (or client) issues commands to the program in the form of successive lines of text (command lines).
ES6	ECMAScript 6. Scripting-language specification standardized by Ecma International in ECMA-262. The 6th edition was initially known as ECMAScript 6 (ES6) and later renamed as ECMAScript 2015.
JSX	JavaScript XML. An extension to the JavaScript language syntax.
XML	Extensible Markup Language. A markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.
DOM	Document Object Model. A cross-platform and language-independent interface that treats an XML or HTML document as a tree structure wherein each node is an object representing a part of the document.
API	Application Programming Interface. An interface or communication protocol between a client and a server intended to simplify the building of client-side software.
JSON	JavaScript Object Notation. An open-standard file format that uses human-readable text to transmit data objects consisting of attribute-value pairs and array data types (or any other serializable value).
HOC	Higher-Order Component. An advanced technique in React for reusing component logic.

1 Introduction

The goal of this thesis is to introduce modern and advanced ways of developing cross-platform React Native applications. It will demonstrate the capabilities of ReactJS and the React Native framework during the development of a practical application.

The application is based on the idea of High Intensity Interval Training and allows utilizing newly introduced concepts such as React hooks or Context API by managing the global application state and component behavior. It will allow developers to replace third-party global state management libraries with a simple yet powerful solution. [1]

Apart from the new API, other modern and advanced topics will be covered. The TypeScript capabilities of static type checking, implementation of performant and eye-pleasing animations and other features offered by powerful and developer-friendly React Native [2] will be demonstrated in the thesis.

1.1 About ReactJS and React Native

React Native is a cross-platform JavaScript framework made for developing mobile applications on both Android and iOS. With the help of third-party libraries, the applications utilizing React Native can also be developed for Windows (Universal Windows Platform) [3], Web (React-Native-Web), MacOS and various TV-platforms. [4]

The framework is built on top of React (also known as ReactJS), an open source JavaScript library, which was introduced in 2011 by Jordan Walke, and later developed at Facebook. The React library was designed to provide an easy and effective way to develop web user interfaces, and due to its low learning curve, high performance and powerful functionality, it got public attention and became one of the most popular front-end framework nowadays. [5]

Two years after the release of the web library, Facebook introduced React Native to revolutionize mobile development by creating a fast and easy way to create performant applications on both major mobile platforms with maximum code reuse and compatibility [5].

In 2019, according to a Stack Overflow survey [6], React Native is the most popular cross-platform mobile framework as seen in the next Figure.

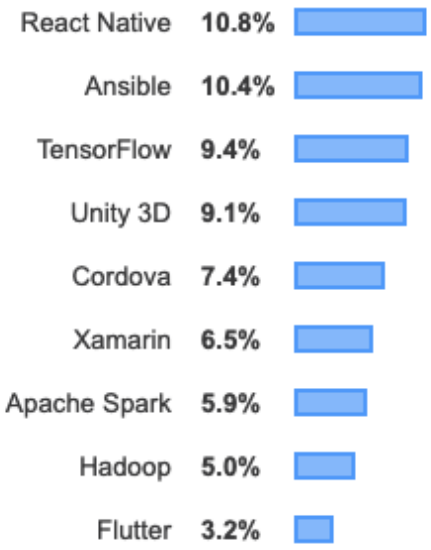

Figure 1. Popularity of React Native and other frameworks according to Stack Overflow [6].

1.2 Reasons to choose React Native

One of the main reasons to prefer React Native to other solutions is shorter development time. A developer needs to write the code only once to be able to use it on different platforms. Additionally, there is no need to waste time on code recompilation and application restarts. Live and Hot reload made it possible to apply changes in the code in the matter of milliseconds. In the latest React Native release Fast Refresh was introduced, which brought even better instant feedback to code changes and added support for React hooks. [7]

Thanks to React Native architectural design, developers only have to write the code once to create cross-platform platform software that is almost indistinguishable from the native counterparts for a specific system. Furthermore, React Native allows one to build hybrid applications which combine both native and JavaScript codebase. [8]

That is the reason why one of the biggest advantages of React Native over native applications is the speed of development. Some companies even report that cross-platform capabilities of this framework helped them speed up the development time up

to 300%. [9] That rapid code delivery allows even small companies to create innovative mobile applications with fast speed and limited resources.

Even though React Native is not entirely “native”, it still relies on the same building blocks as iOS and Android applications providing the same level of user experience and almost the same level of performance [9].

However, with native application development, every time an update for an application is created, the application bundle is required to go through the entire build process and upload the new version to Apple Store and/or to Google Play. The stores must approve all the changes, and only after that the users of the application, who still need to install the update manually, can receive the changes. [9]

Thanks to React Native JavaScript bundling and third-party tools such Expo or CodePush, the entire updating process is greatly simplified. It usually runs locally by using JavaScript files embedded in the application. Furthermore, when a developer is ready to deliver the updates, they can simply publish them, and the update will be applied on each user’s device the next time they relaunch the application. [10]

Last but not least is the great debugging support provided by React Native and third-party tools such as Reactotron and React Native Debugger. Since the same application is delivered across multiple platforms, significantly less time and effort are required to spot and remove errors and bugs. One fix usually eliminates the bug on both iOS and Android, which provides consistent behavior across the codebase. [9]

2 Advanced concepts

During the last few years, React Native has evolved from a simple internal Facebook hackathon project into one of the most used cross-platform solutions on the market. With increasing community interest towards this framework, more and more ideas and advanced approaches are introduced by Facebook and third-party developers. The increasing competition from Flutter, another popular cross-platform framework from Google, forces Facebook to spend more resources towards development of React Native and to be more open to the community needs. [8] In this thesis, some of the main advanced development concepts are going to be reviewed and implemented and their practical implementation displayed.

2.1 Typescript

Typescript is an open-source superset of JavaScript developed and maintained by Microsoft. It brings optional static typing checks to JavaScript, the feature that many developers miss in the language. [11]

The main difference between statically and dynamically typed languages is in how they perform the data type checking. Statically typed do it during the compilation of an application while dynamically typed perform the checking during the execution of an application. In other words, statically typed languages require the developer to define the types before their usage, while dynamically typed languages allow skipping this step. [11]

For developers who have switched to JavaScript from statically typed languages such as Java or C++, this, in most cases, looks illogical and weird. It might even lead to serious issues in projects of a large scale. For example, a developer can spend many hours on debugging only to discover that one of the variables has become “NaN”. [12]

Typescript was released to public in October 2012, after a few years of development by Microsoft [13]. The language was developed by Anders Hejlsberg who also created Turbo Pascal, Delphi and C# before that. TypeScript was made due to the issues arising with development of complex applications, such as the ones used by Microsoft or other JavaScript users. The developers of TypeScript tried to find a solution which will not

break the compatibility with JavaScript and its cross-platform support. This led to the creation of a JavaScript compiler with extra syntax language extensions. [12]

TypeScript is reverse-compatible with JavaScript and compiles to the latter. In fact, an application written in TypeScript can be executed by any modern browser or can be used together with server-side platform such as Node.js. This allows slowly refactoring an old application from JavaScript into TypeScript one file at a time by adding types support gradually. [12]

The main difference with the predecessor of TypeScript, JavaScript, is static type allocation, full class support and module importing. These qualities allow speeding up the development process, improved readability, refactoring and reusability of code, help with error search at the stage of development and compilation and in theory speed-up the application execution. [12]

There are multiple default annotation types predefined such as “number”, “boolean”, and “string”. Weak or dynamically typed structures can have “any” type. A developer can also define their own types. [12]

Originally React and React Native had a default type checker called PropTypes, which gave the developer a set of validators which can be used for checking that the data the developer has entered is correct. Overtime, PropTypes were replaced by more advanced solutions such as Flow and TypeScript. An example of PropTypes usage can be found in the next Listing. [14]

```
import React from 'react';
import PropTypes from 'prop-types';

class MyComponent extends React.Component {
  static propTypes = {
    foo: PropTypes.number.isRequired,
    bar: PropTypes.string,
  };

  render() {
    return <div>{this.props.bar}</div>;
  }
}
```

Listing 1. An example of PropTypes type declaration [14].

Flow is an improved static typing library developed by Facebook and commonly used with React and React Native. It comes as a default type checker for all new projects

made with React Native CLI and provides better features and functionality compared to depreciated PropTypes. [15] An example usage of PropTypes can be found in the next Listing.

```
// flow
import * as React from 'react';

type Props = {
  foo: number,
  bar?: string,
};

class MyComponent extends React.Component<Props> {
  render() {
    this.props.doesNotExist; // Error! `doesNotExist` prop was not defined.

    return <div>{this.props.bar}</div>;
  }
}
```

Listing 2. An example of Flow or TypeScript type declaration [15].

Both Flow and TypeScript allow using static typing, but using them is not enough in a serious project if a developer needs to use the power of ES6 and future versions of JavaScript. Flow is designed as a static code analyzer, which means that one needs to use a transpiler for the code, but by using TypeScript the developer also obtains the support of the latest ECMAScript features and better checking performance. [15]

To easily setup TypeScript into a new project, all a developer needs is to specify a ready-made template [12] and then remember to create all new files with .ts or .tsx (to support JSX syntax) extensions. A terminal command to perform initial setup can be found in the next Listing.

```
react-native init MyAwesomeProject --template typescript
```

Listing 3. New React Native project setup with TypeScript support [12].

The result folder structure can be seen in the next Figure.

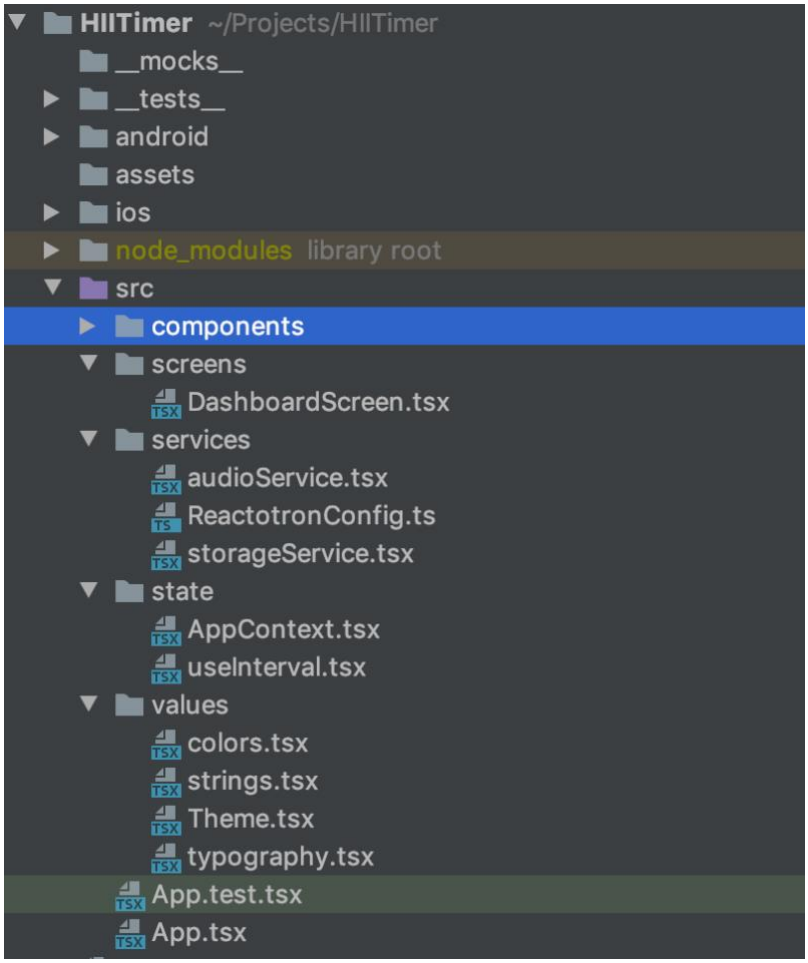

Figure 2. Example file structure in the project covered in this thesis.

As can be seen in the Figure above, the project structure after applying TypeScript remains completely the same as with regular JavaScript projects; the main difference is that now a project will have static type support which is related to static typing benefits.

2.2 Context API

The new version of React 16.3 introduced a new API called Context. In a traditional React application, the data is transferred from top to button or from parent to a child. This approach might be too complicated for some props (such as selected language, UI-theme or other global application settings) which need to be passed to a multitude of components located in different places of the application. The new API allows passing the props directly ignoring the component tree. [16]

Context is a pretty straightforward API which in theory can help with developing complex applications without usage of third-party libraries such as Redux or MobX. The API reduces the amount of the boilerplate required to create a global application state and makes the code readability much easier. [16] The practical implementation of the API together with React hooks is going to be demonstrated later in this thesis.

```
const ThemeContext = React.createContext('light');

class App extends React.Component {
  render() {
    // Use a Provider to pass the current theme to the tree below.
    // Any component can read it, no matter how deep it is.
    // In this example, we're passing "dark" as the current value.
    return (
      <ThemeContext.Provider value="dark">
        <Toolbar />
      </ThemeContext.Provider>
    );
  }
}

// A component in the middle does not have to
// pass the theme down explicitly anymore.
function Toolbar(props) {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

class ThemedButton extends React.Component {
  // Assign a contextType to read the current theme context.
  // React will find the closest theme Provider above and use its value.
  // In this example, the current theme is "dark".
  static contextType = ThemeContext;
  render() {
    return <Button theme={this.context} />;
  }
}
```

Listing 4. Example usage of Context API [16].

2.3 Hooks

Hooks were introduced into React in version 16.8 to allow the usage of application state and other React features without using React Classes [17].

Even though component-oriented architecture allows reusing views in the application, one of the biggest issues a developer meets is how to reuse the logic located in the

required component state between other components. When there are components with similar state logic, but no good solutions to reuse the logic, the code in the constructor and live cycle methods can become/be duplicated. To fix this issue, developers usually use high-order components and render props. [17]

Both methods have their disadvantages which can lead to a complication of the application codebase.

Hooks are aiming to solve the issues allowing writing functional components which have access to state, context, lifecycle methods, ref and others without writing React Classes [17]. An example usage of React hooks can be observed in the next Listing.

```
import React, { useState } from 'react';

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Listing 5. Example usage of React hooks [17].

As can be seen in the above Listing, the Class component is replaced with a functional component, the local state is removed and instead the new “useState” function is used.

The best way to understand hooks is to determine how the behavior normally used while writing the classes can be reproduced with hooks. Usually used classes often need the following:

- To use state.
- To use life cycle methods such as componentDidMount() or componentDidUpdate().
- To have access to Context APIs which were mentioned before. [17]

With React hooks, similar behavior can be replicated in purely functional components:

- To access state, useState() hook is used.
- Instead of using componentDidMount() or componentDidUpdate(), life cycles useEffect() hook is used.
- Instead of static contextType property, useContext() hookis used. [17]

Various unofficial React hooks implementations have been introduced into React Native before, but starting from React Native version 0.59 no additional setup is required and hooks can be used in any functional component a project has. Similar to TypeScript implementation which was described above, a developer or a team can slowly refactor their entire code base to use hooks by replacing React Class based components one at a time. [18]

2.3.1 Example usage of useState() hook

Local component state is a crucial part of React. It allows declaring component variables, which controls the component, and changing its behavior and looks based on the state changes. [19] A simple example of standard class-based state usage is shown in the next Listing.

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  render() {
    return (
      …
    );
  }
}
```

Listing 6. The example usage of state in a Class-based component [19].

To start using the useState() hook, all that is needed is converting the class into a function. Then the hook needs to be added, declaring a new state variable called “count” and a method to change the variable, called “setCount”, as seen in the next Listing.

```
function Example() {
  const [count, setCount] = useState(0);
    return (
     …
    );
}
```

Listing 7. The example usage of state in a functional component [19].

The initial state can also be resolved here by passing an argument to useState() which can be of any type.

useState() hook does not limit the developers to only one state variable. They can declare as many of the hooks as they need as seen in the following Listing.

```
  const [count, setCount] = useState(0);
  const [user, setUser] = useState(0);
  const [email, setEmail] = useState(0);
  const [password, setPassword] = useState(0);

}
```

Listing 8. The example usage of multiple useState() hooks in one functional component.

It is important to mention that the syntax of using “setSomething” or “useSomething” is not mandatory but it is considered as a best practice and should be followed when using existing hooks or creating one’s own, custom hooks [17].

2.3.2 Example usage of useEffect() hook

The second most used new hook is useEffect() which simulates class-based methods such as componentDidMount() or componentDidUpdate(). In other words, it allows executing side effects in functional components. Examples of side effects can be API calls, DOM re-rendering and event listener subscriptions [1].

By declaring the useEffect() method in a functional component, a developer lets React know that they want to perform a certain action after rendering.

```
useEffect(() => {
  function handleStatusChange(status) {
    // Do something
  }
  // Specify how to clean up after this effect:
  return function cleanup() {
    //Perform a cleanup action
  };
}, [count]); // Only re-run the effect if count changes
```

Listing 9. useEffect() hook example.

As can be seen in the Listing above, to perform an action after rendering is all that is needed to put the action inside the useEffect() hook as the first argument.

An important thing to know about useEffect() hook is how to avoid performance issues. By default, the hook will be called every time the component re-renders, which can cause a number of unnecessary computations. To avoid this one should pass the second argument to the hook as can be seen in the code example above. The argument accepts an array of arguments the developer would need to observe, and the argument will make sure that the first argument of useEffect() is triggered only when one of the observables changes. If the developer needs to avoid triggering the re-render and call useEffect() only once during the mounting of the component, the hook requires to pass an empty array as the second argument. [1]

As with the useState() case, the useEffect() hook can be declared multiple times in the same functional component in case multiple side effect observations are required. [1]

2.3.3 Other hooks and rules of using them

The Context API, which was described before, allows the nested child components to obtain a property of the parent component without going through the entire component tree [16].

The useContext() hook makes the usage of Context easier and clearer. It takes a context object, which at first returns React.createCntext, and with the following calls the hook returns the current value of the context. [16]

```
const value = useContext(AppContext);
```

Listing 10. Example of useContext() usage [16].

React hooks are quite straightforward to implement, but there are certain rules that need to be followed:

- Hooks must be declared only at the top level of the function. They should not be called inside cycles, conditions or inner functions.
- Hooks must be used only inside functional components. They should not be used inside a regular JavaScript function. The only exception is when the user creates custom hooks. [17]

It is possible to create custom hooks when a developer needs to reuse state logic in multiple components. First, the hook is declared as a separate function. Then, it can be imported and reused across the application. [17]

```
Export default function useUserStatus(userID) {

      // Write the logic here

      return isRegistered;
}
// Now it can be called in other components
function CheckRegistration(props) {
      const isRegistered = useUserStatus(props.user.id)
      // Write the logic here
}
```

Listing 11. Example usage of custom-created hook.

It is important to mention that the states of the two functional components do not affect each other in any way. Using hooks is only a way to reuse the logic of the state, but not the state itself. Moreover, every call to the hook returns a totally isolated state. The same hook can be reused multiple times in the same component. [17]

User-made hooks are more like an agreement than a feature. If a function starts from “use” and calls other hooks, it is considered as a custom hook. It allows better code readability and usage of automatic linters to find bugs in the code that uses hooks. [17]

In the application developed in the project this thesis deals with, all the hooks mentioned above were used. Also useReducer() and useRef() were used. In addition, custom hooks needed to manipulate the application were created.

Many cases exist in which the default and custom hooks should be used such as animations and timers . Some of them are described in this thesis when the practical application of the React hooks is dealt with.

2.4 Animations

When dealing with modern mobile applications, users are used to a good UI and UX, smooth animations and transitions. Smooth animations at 60 frames per second contribute to great user experience, but for a long time implementing an animation was a real issue in React Native. [20] The issue was solved with the creation of Animated library which is now integrated into the core React Native framework.

The animated API provides a variety of animation types which can be applied for React Native elements such as what is listed below:

- .decay() – starts with an initial velocity and gradually slows to a stop.
- .spring() – provides a simple spring physics model.
  .timing() – animates a value over time using easing functions. [21]

In most cases only timing() is needed. The method implements element acceleration and deceleration by default. The following Listing shows the timing() example.

```
Animated.timing(this.state.xPosition, {
  toValue: 100,
  easing: Easing.back(),
  duration: 2000,
}).start();
```

Listing 12. Code example of Animated.timing() implementation.

Animations can be triggered by starting the start() method which can also return a callback on completion. Multiple animations can also be combined by calling the sequence() method and by passing the desired animations as an array argument. It is also possible to combine multiple animated values using addition, multiplication, division or modulo to create a new animated effect. [21]

One can also use the interpolation method to map input ranges to output ranges with linear interpolation or using easing functions. [21]

One of the main limitations of React Native is the usage of the native bridge where the JavaScript code is sent to Native in an asynchronous way. This can cause serious visual issues if left uncheck especially when used for heavy UI computations such as animations. [22]

Since Animated API is serializable, it is strongly recommended to use the native driver for rendering animations by moving the animation logic to the native side completely to improve performance and to avoid blocking the UI thread [23]. To use the native driver, the “useNativeDriver: true” property is required in the animation configuration as shown in the next Listing.

```
Animated.timing(this.state.animatedValue, {
  toValue: 1,
  duration: 500,
  useNativeDriver: true, // <-- Add this
}).start();
```

Listing 13. Example code of native driver usage.

It is good to know that there are some limitations on the usage of native driver presented in the current implementation of Animated API. At the moment, it is possible to animate only non-layout properties such as transform and opacity, but the flexbox or position properties will not work with the native driver. [23] It is recommended to try to optimize the code to utilize the native driver as much as possible. The practical application of the Animated library will be discussed later in this thesis by demonstrating how the property in the application was implemented.

3 Practical implementation of advanced concepts

3.1 Project description and design

To demonstrate the capabilities of ReactJS and React Native, it was decided to create a simple, yet functional application, and make it usable by publishing it on both iOS and Android through Apple Store and Google Play.

The idea of the application is to provide the user with a configurable timer for multi-purpose High Intensity Interval Training, Tabata or other similar training techniques which rely on the usage of repeating sets with various intervals. The application must be quick to understand, fast to configure and easy to operate during exercises.

The goal of HIIT training is to have multiple repeating time periods, each having different exercise intensity to stimulate the user's body to activate long-lasting fat burning metabolism processes. The training consists of mixing high intensity exercises and periods of rest. It improves body and heart endurance and helps with weight loss. Swimmers, runners, cyclists and other athletes use this type of training to increase their speed although the most popular side effect of interval training is fast calories usage and fat burning and as a result the loss of weight. [24]

A good example of the interval training is a running routine where jogging is combined with sprinting: for the first 50-100 meters a person sprints in their maximum tempo to raise the pulse to the border between aerobic and anaerobic zones. After that they rest for 100-200 meters, waiting for the pulse to drop, and then repeat the sequence again. The intervals of minimum and maximum intensity are usually repeated 5-15 times during the training session and the maximum effective time for fat burning is about 30 to 45 minutes. The HIIT technique can be adjusted to any level of physical condition and usually includes from 2 to 4 repeating periods of different intensity. [24]

The application designed in the final year project has two main UI states: session configuration and active session. During the session configuration, the user is able to select the amount of total sets and total rounds and also configure individual time of sets.

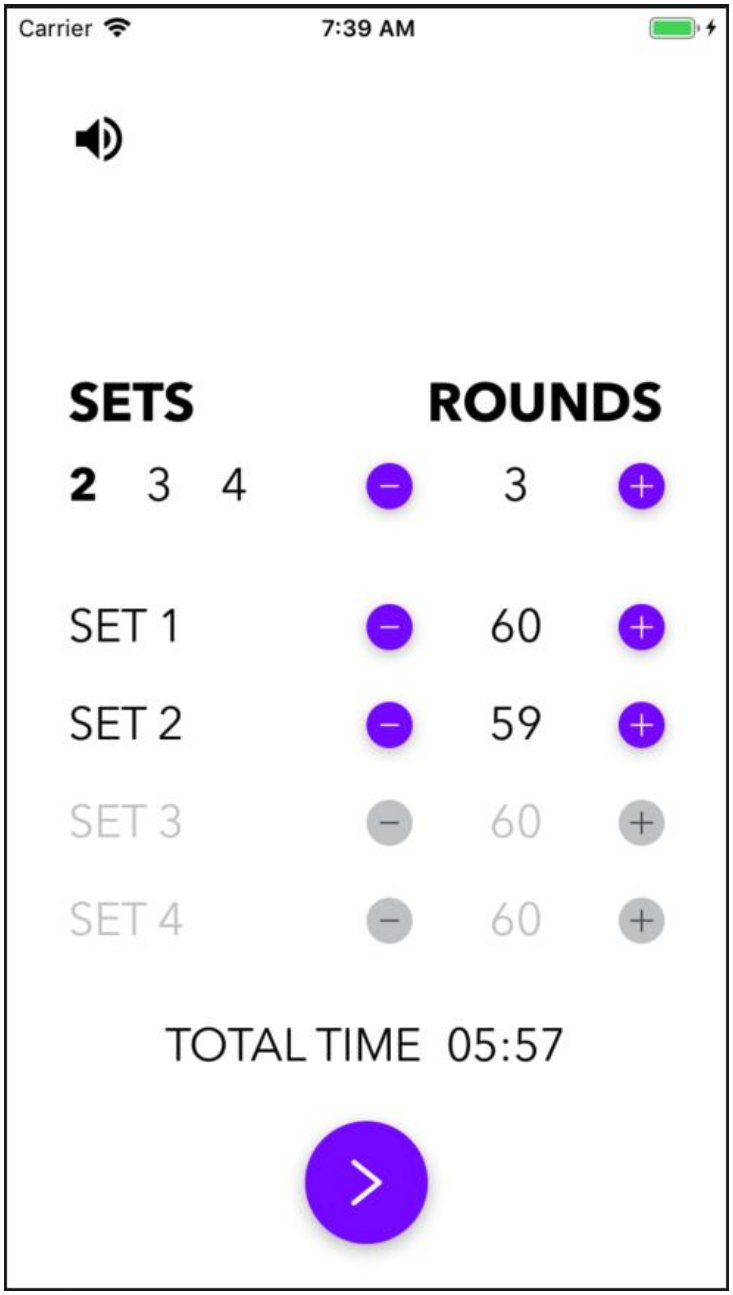

Figure 3. Session configuration UI.

Amounts of sessions and rounds are configurable through touchable elements, where the sets number is restricted to the range from 2 to 4 sets, and rounds have a minimum of 1 and no upper limit. “Plus” and “Minus” controls regulate the total amount of rounds. Set time is configurable in the same manner, but the numbers themselves are also editable with the keyboard as seem in the next Figure.

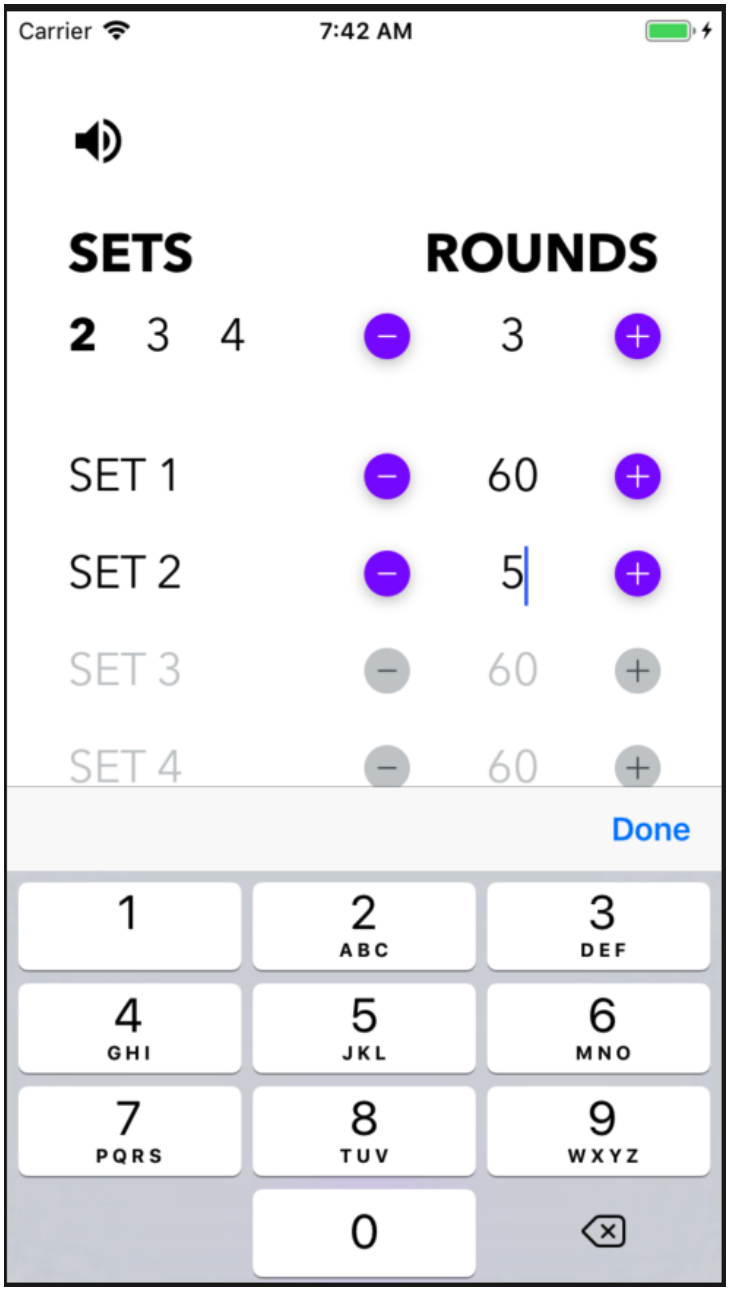

Figure 4. Session configuration with keyboard enabled.

As can be seen in the Figure above, the keyboard avoiding view is implemented which shrinks spacing between elements to provide better visibility for the text input elements when the keyboard is active. Validation checks to control user input are also implemented with usage of hooks, and they will be described in detail in the upcoming chapters.

All changes in rounds, set amount and individual set times are animated and immediately reflected in calculation of total time thanks to global state and hooks controlling all the internal logic.

After the desired training program is configured, the user can press ‘start’. The button changes the UI state to activated timer as seen in the next Figure.

Figure 5. Activated session UI.

During the activated session, most of the UI is replaced to allow easy readability and controls during the training session. The session time that is left is indicated by the central timer and the animated circular progress bar. After the set is completed, the circular bar uses animated transition to reset itself to default position after which a new set begins. After every set, the background color is also changed depending on the set number. If the user pauses/resumes the timer, the background is changed to/from grey to better indicate the state of the timer as seen in the next Figure.

Figure 6. Paused timer UI.

Alongside the background changes, an extra text, “Paused”, appears on top of the session time left and one of the control buttons is changed to play/resume icon.

If the user wishes to end the training prematurely, they can press the “close” button to trigger a confirmation popup as seen in the next Figure.

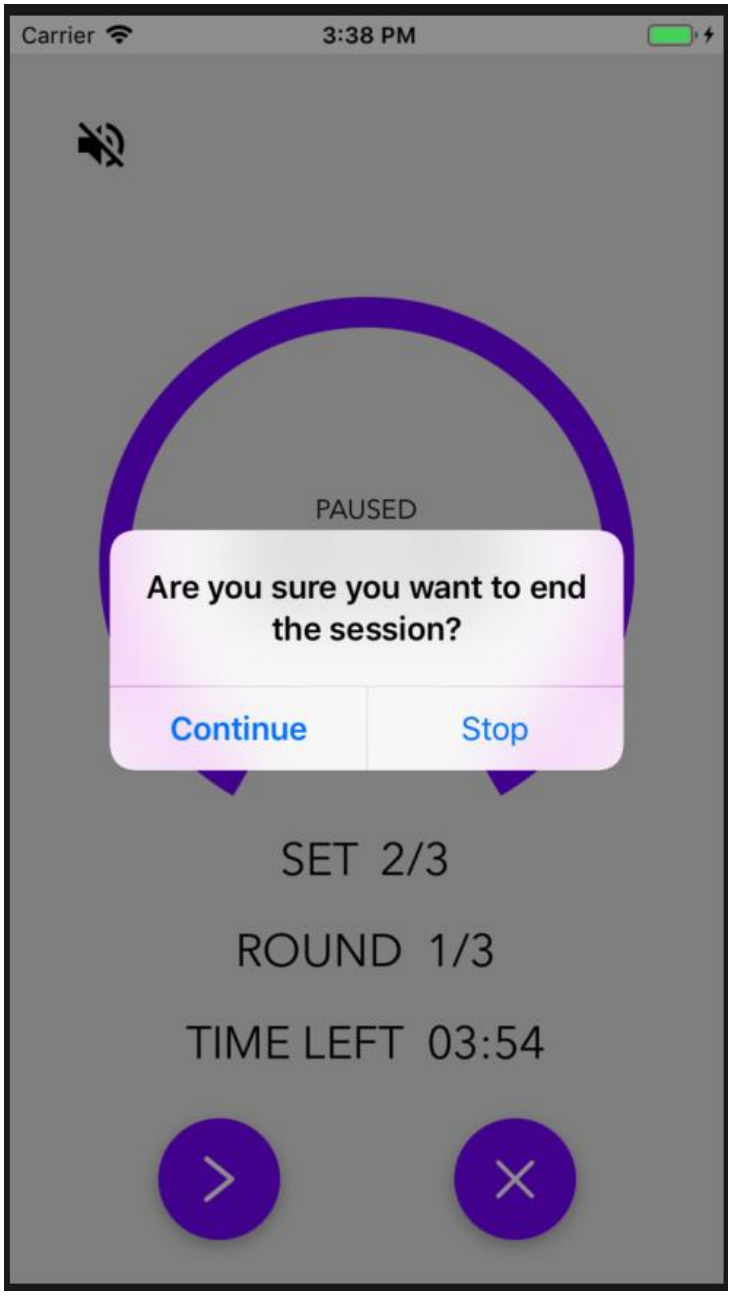

Figure 7. End session confirmation popup UI.

The user then can either continue with their training or stop the session and reset the progress. In that case, the user is returned to the session configuration UI. The same

end result is triggered when the training session runs out of time. Then user can readjust the configuration and start over or wait until their next training session.

3.2 Project structure

Since there are no established best practices related to organizing a React Native project, the project structure follows a pattern which has been used in previous personal and customer projects with some influence of Android folder naming, as one can see in the next Figure.

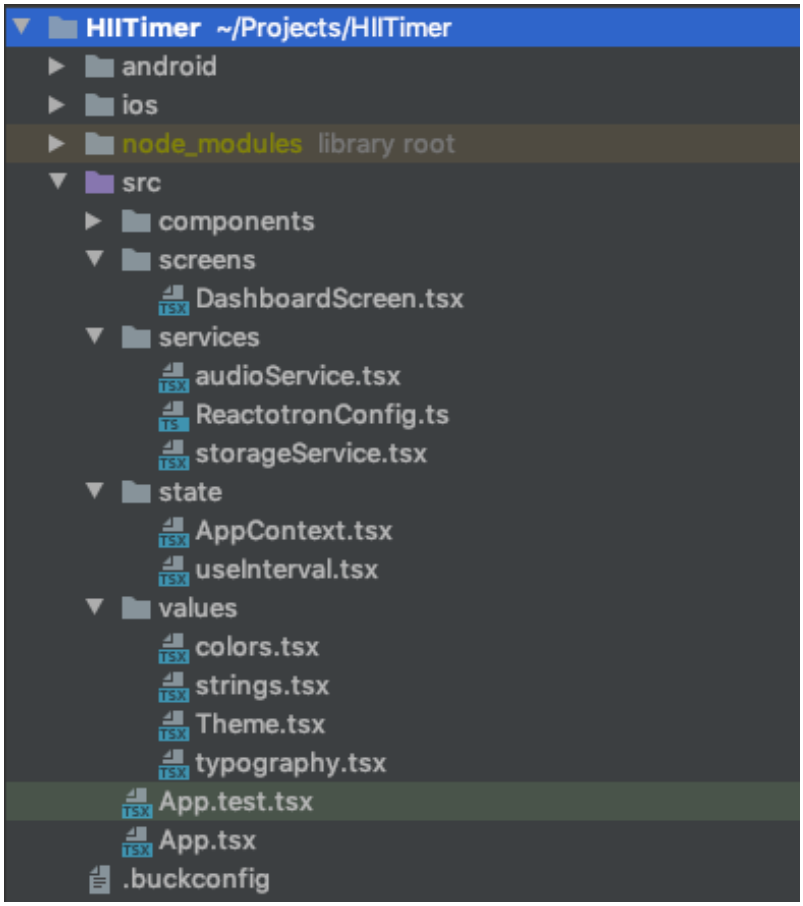

Figure 8. The file structure of the project.

As can be seen, the main source code is located under the “src” folder apart from default, auto-generated React Native “android”, “ios” and “node_modules” folders. The entry point of the project is located in the “App.tsx” file where the initialization of the global state is happening.

“DashboardScreen.tsx” is a container for all the conditional UI logic. Since the UI of the application is quite minimalistic, there is no need for extra screens and third-party navigational libraries. In the future when an extra screen will be added for user settings, it can be wise to add a “react-navigation” library to allow the application to have animated transitions and navigation logic handling.

Apart from component behavior, “DashboardScreen” contains extra logic which renders the screen UI based on extra conditions. The first part uses the “useEffect” hook to trigger a custom storage service function responsible for the initial application launch after installation. In that case the application loads default timer values. If the application has already been used by the user and the values have been modified before, then the application will load the values to the global state overriding the default ones.

The second part also utilizes the “useEffect” hook to show the main UI only after the application logic was loaded to memory and the global state was retrieved from the local phone storage. It utilizes the third-party library called "react-native-bootsplash" which allows a smooth transition from the start of the application to ready-to-use state by showing the splash screen until the application is ready for user interactions.

The third part utilizes the “react-native-keep-awake” third-party library and forces the phone screen to be always on. It is turned off by default and activated only when the session timer is either in activated or in paused states. It is made to allow the user to be always aware of the current progress, and in future iterations of the application it will be togglable in the application settings.

The “services” folder contains configuration for a debugging tool called Reactotron and audio and storage services. Reactotron is a desktop application for inspecting ReactJS and React Native applications. It is used for viewing the application state, showing API requests and responses, doing performance benchmarks and having many other useful debugging and benchmarking capabilities. [25]

Audio service utilizes the “react-native-sound” third-party library and is used to play audio files during the active session timer near the end of time. It uses two different audio files found on free to use websites and plays either one file at a time or a combination of files to alert the user of different stages of the training.

Storage service is responsible for monitoring the initial application launch and for preserving global state across application launches and sessions. It stores the entire global state as a stringified JSON object and updates the storage every time the global state changes.

The “values” folder keeps track of all the minor application variables such as localization strings for various languages, design tokens such as colors, typography and spacing and general application theming. Moving the values outside of local components and keeping them in one place allows for better maintenance and better readability over time and in general keeps the project clean, and adding new features remains easy.

The main chunk of files is located in the “components” folder as seen in the next Figure.

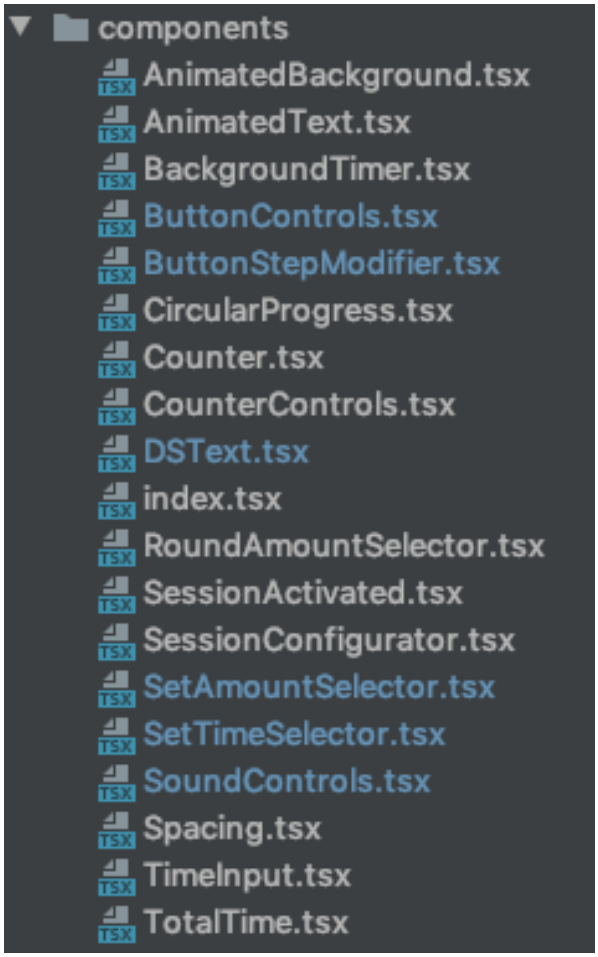

Figure 9. Structure of components folder.

In this folder, the majority of UI and logic related components responsible for various functionality are kept. ReactJS and React Native heavily rely on component driven

development; hence, everything is componentized, such as spacing between UI elements and background counter logic as well as animations and sounds. The component driven approach allows moving components around with ease and speeds up the development.

Finally, the “state” folder is responsible for maintaining the global application state and for keeping custom hook implementation. Both will be described in detail in the next chapter.

3.3 Implementation of hooks and Context API in the application

As described previously in this thesis, React hooks and Context API are the new ways of developing ReactJS and React Native applications, allowing class components to be abandoned and clean functional components to be utilized instead.

The main global state logic of the application is located in “AppContext.tsx”. As shown in the next code example, the initial state is declared first.

```
export const initialState: State = {
  counterStatus: 'stopped',
  setsTime: [60, 60, 60, 60],
  totalRounds: 2,
  currentRound: 1,
  totalSets: 3,
  currentSet: 1,
  timeSession: 60,
  timeSessionLeft: 60,
  totalTimeLeft: 360,
  stateLoaded: false,
  volumeState: 'on',
}
```

Listing 14. Code example of initial state.

As can be seen in the Listing, the entire application state is managed only by a dozen lines which are loaded during the initial application start. Then the application uses the initial state in building App Context as seen in the next code snippet.

```
const StateCtx = createContext(initialState)
const DispatchCtx = createContext((() => 0) as Dispatch<Action>)

export const Provider = ({ children }) => {
  const [state, dispatch] = useReducer(appReducer, initialState)
  return (
    <DispatchCtx.Provider value={dispatch}>
```

```
      <StateCtx.Provider value={state}>{children}</StateCtx.Provider>
    </DispatchCtx.Provider>
  )
}
export const useDispatch = () => useContext(DispatchCtx)
export const useGlobalState = () => {
  return useContext(StateCtx)
}
```

Listing 15. Code snippet of App Context logic.

As seen in the above snippet, the App Context is broken into two parts: State Context and Dispatch Context. The first one is responsible for retrieving the global application state. It obtains the initial state as an argument to populate the state at the start of the application. Dispatch Context is responsible for updating the global application state through Actions.

The provider component wraps both of the State and Dispatch contexts into a Higher Order Component taking a child component, applying Context API logic and functionality to it and returning the new component which is now aware of changes in the global application state. As seen in the next code snippet, the Provider component is used at the root of the application component tree and wraps around the main Dashboard Screen component.

```
const App = () => {
  return (
    <Provider>
      {Platform.OS === 'android' && (
        <StatusBar
          backgroundColor={Theme.colors.primaryColor}
          barStyle="light-content"
        />
      )}
      <DashboardScreen />
    </Provider>
  )
}
```

Listing 16. App.tsx return method with Provider wrapper inside.

To use Context API with hooks later in the application, one must declare “useDispatch” and “useGlobalState” custom hooks in the desired component and use the hooks to display or change the global application state as seen in the next code snippet.

```
const DashboardScreen = () => {
  const dispatch = useDispatch()
  const state = useGlobalState()

  state.counterStatus === 'stopped'
```

```
    ? KeepAwake.deactivate()
    : KeepAwake.activate()

  useEffect(() => {
    checkFirstLaunch({ dispatch }).then()
  }, [])

  useEffect(() => {
    if (state.stateLoaded) {
      RNBootSplash.hide({ duration: 500 })
    }
  }, [state.stateLoaded])

  const changeVolumeState = () => {
    dispatch({
      type: 'changeVolumeState',
    })
  }
```

Listing 17. Usage example of “useDispatch” and “useGlobalState” custom hooks.

Here the “state” object can be used to access all the global state variables needed. The “useEffect” hooks in this scenario also utilize both “state” and “dispatch” objects to perform component lifecycle methods. If the global state requires modification, the “dispatch()” function can be called and an argument consisting of action type and optionally payload can be passed to provide extra data to the reducer. As seen in the next code snippet, the app reducer function obtains as arguments the current state and the action triggered by “dispatch()”.

```
const appReducer = (state: State, action: Action): State => {
  switch (action.type) {
      case 'changeVolumeState':
        let volumeState
        if (state.volumeState === 'on') volumeState = 'vibro'
        else if (state.volumeState === 'vibro') volumeState = 'off'
        else volumeState = 'on'
        mergeAppState({ volumeState })
        return {
          ...state,
          volumeState,
        }
  }
}
```

Listing 18. An example of App Reducer’s action.

As can be observed, the custom logic of dealing with the global state can be implemented, depending on the action type, and then the state can be overwritten in the “return” part of the reducer. It is important to note that not only the updated part of the state must be included in the “return” statement, but also the other unmodified parts of the state. Otherwise, the application loses every part of the state except for the newly modified one.

3.4 Implementation of animations

While developing the application in the final year project, it was decided to divide custom-made animations into two groups: text-based and background-based animations.

Text-based animations are implemented into the application in the form of a Higher-Order Component. They take a component such as text or text input and apply extra logic to the component allowing the component to change its shape and size based on the external state changes.

As can be seen in the next code snippet, the Higher-Order Component allows transforming and animating any child component, not only the text-based ones.

```
const AnimatedText = props => {
  const [animation] = useState(new Animated.Value(1))
  const animationStyles = {
    transform: [{ scale: animation }],
  }

  useEffect(() => {
    if (!props.disabled) {
      Animated.sequence([
        Animated.timing(animation, {
          toValue: 1.2,
          duration: 100,
          useNativeDriver: true,
        }),
        Animated.timing(animation, {
          toValue: 1,
          duration: 300,
          useNativeDriver: true,
          easing: Easing.bounce,
        }),
      ]).start()
    }
    return () => {}
  }, [props.trigger])

  return (
    <Animated.View style={[props.style, animationStyles]}>
      {props.children}
    </Animated.View>
  )
}
```

Listing 19. Animated text component code.

Here the usage of Animated API and React hooks are combined to create a simple yet effective animated container for child components.

The above example of the Higher-Order functional component serves as a good example of the usage of hooks. “useState” is used to store the local animation state and it receives “new Animated.Value(1)” as the initial state. “useEffect” acts as a lifecycle function and waits for the functional component to receive a “trigger” prop to activate the animation sequence and change animation styling applied to a child component as seen in the next code example.

```
<AnimatedText trigger={state.totalRounds}>
  <DSText style={s.controls}>{state.totalRounds}</DSText>
</AnimatedText>
```

Listing 20. Animated text example usage in the application.

This particular usage of Animated API was found by trial and error, while trying to achieve a bouncing effect for a temporarily resized element. After the “trigger” prop is activated, the child element is resized by 120% during a time period of 100 milliseconds. Then the element shrinks back to 100% during the next 300 milliseconds while also being affected by the “easing: Easing.bounce” property. Both animations have “useNativeDriver” set to true to allow better performance on slower devices.

An animated background has a more complex logic as seen in the next code snippet. Apart from applying the animations, the background color should also be changed at the same time as the animation change.

```
const AnimatedBackground = props => {
  const state = useGlobalState()
  const [animation] = useState(new Animated.Value(1))
  const [backgroundColor, setBackgroundColor] = useState(Theme.colors.white)
  const [rippleColor, setRippleColor] = useState(Theme.colors.white)
  const animationStyles = {
    backgroundColor: rippleColor,
    transform: [{ scale: animation }],
  }
  useEffect(() => {
    if (!props.disabled) {
      setRippleColor(changeBackgroundStyle())
      Animated.sequence([
        Animated.timing(animation, {
          toValue: 1,
          duration: 1,
          useNativeDriver: true,
        }),
        Animated.timing(animation, {
          toValue: 25,
          duration: 500,
          useNativeDriver: true,
        }),
      ]).start(() => {
        setBackgroundColor(changeBackgroundStyle())
      })
```

```
    }
  }, [state.counterStatus, state.currentSet])
```

Listing 21. Animated background implementation code.

The implementation of Animated API is similar to the Animated text component described previously, but the global application state is used. When the current set is changed, the application switches background colors, choosing the next one in line. When the status of the timer changes, the application also replaces the background color depending on the status and current set. The animation itself is quite straightforward and just transforms a circle of 32 by 32 pixels to 2500% of its original value in half a second. After the animation ends, the screen background changes to the color used in the animation to keep the change permanent until the next set or status change.

4 Conclusion

The goal of the thesis was to briefly explain what ReactJS and React Native are and go through their development history and usage. After a general introduction, new and advanced developing techniques and concepts were introduced and described. Later, all that information was applied in practice to develop a real-world application.

The application which was developed was built with React Native, and it utilized advanced concepts such as TypeScript, React hooks and Context API. On top of that, various extra features for enhancing user experience were introduced such as animations, sounds, and different themes.

The application was successfully finished and included all the advanced concepts covered in the theoretical part of the thesis. The concepts allowed developing the application without any issues and with less time. The resulting codebase is easy to build upon, and introducing new features can be done in a fast and reliable manner. The application itself was already published and is available on Apple and Google stores.

At the moment of writing this thesis, React Native repository has about 17,000 commits, 87 branches, 2,000 contributors, 80,000 stars and is one of the most popular repositories on GitHub [26].

React Native is used in hundreds of thousands of projects across the world. There are various areas of use which include B2B applications, internal company software and educational tools. The JavaScript framework is used by large corporations such as Facebook, Google, Instagram, Microsoft, Uber, Tesla, and LinkedIn. It is one of the most used cross-platform solutions and its popularity is keep growing. [5]

JavaScript and TypeScript are of the most popular development languages according to a Stack Overflow development survey carried out in 2019 [6] as seen in the next Figure.

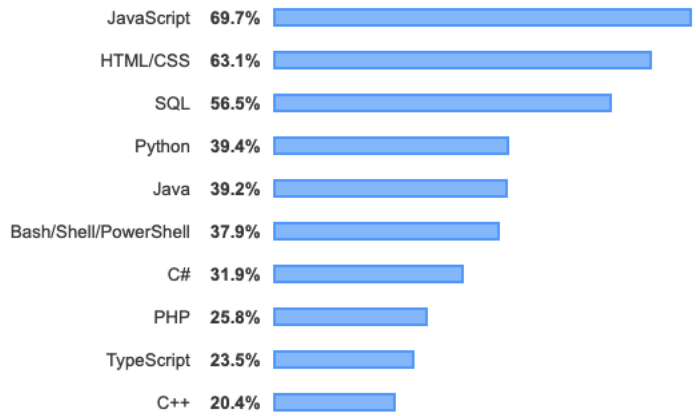

Figure 10. List of programming languages sorted by popularity according to Stack Overflow [6].

As seen in the Figure, almost 70% of all developers use JavaScript [6].

Having the declarative programming paradigm, which focuses on specifying the intended results instead of providing guidelines as to how to achieve them, React Native applications have understandable and easy-to-read code. The framework is scalable and easy to learn especially for beginner programmers. [9]

The framework is open-sourced so anyone can contribute with code improvements, documentation, new components and libraries. In addition, the project itself receives strong support from Facebook allowing the framework to be continuously improved. One of the goals of the 2019 roadmap is a large-scale re-architecture of React Native [8], which implies that Facebook is really committed to the future of the framework.

Combined with ReactJS being the most popular front-end framework [6] React Native has the strongest base of potential developers and possibility for a very bright and long future.

References

[1] S. Ryzhov, "From Redux to Hooks: A Case Study," [Online]. Available: https://staleclosures.dev/from-redux-to-hooks-case-study/. [Accessed 25 October 2019].

[2] J. Samanta, "Guide to React Native App Development! Why React Native is the future of App Development?," 14 October 2019. [Online]. Available: https://hackernoon.com/guide-to-react-native-app-development-why-react-native-is-the-future-of-app-development-b2372db9b775. [Accessed 25 October 2019].

[3] "Creating Universal Windows Apps with React Native," Microsoft, 26 May 2016. [Online]. Available: https://www.microsoft.com/developerblog/2016/05/26/creating-universal-windows-apps-with-react-native/. [Accessed 29 September 2019].

[4] A. Calazans, "One React Native Video Interface. 12 Platforms.," Youi.tv, 28 March 2019. [Online]. Available: https://www.youi.tv/one-react-native-video-interface-12-platforms/. [Accessed 25 October 2019].

[5] "A short Story about React Native," Job Ninja, 15 January 2018. [Online]. Available: https://jobninja.com/blog/short-story-react-native/. [Accessed 20 August 2019].

[6] "Developer Survey Results 2019," Stack Overflow, 2019. [Online]. Available: https://insights.stackoverflow.com/survey/2019#most-popular-technologies. [Accessed 26 September 2019].

[7] "Announcing React Native 0.61 with Fast Refresh," Facebook, 19 September 2019. [Online]. Available: https://facebook.github.io/react-native/blog/2019/09/18/version-0.61. [Accessed 30 September 2019].

[8] "Open Source Roadmap," Facebook, 1 November 2018. [Online]. Available: https://facebook.github.io/react-native/blog/2018/11/01/oss-roadmap. [Accessed 28 September 2019].

[9] "Building a React Native App for 80 Million Users," Wix, 1 October 2016. [Online]. Available: https://youtu.be/abSNo2P9mMM?t=2034. [Accessed 25 September 2019].

[10] "Deploying React Native Applications using App Center," ariya.io, 24 April 2019. [Online]. Available: https://ariya.io/2019/04/deploying-react-native-applications-using-app-center. [Accessed 27 October 2019].

[11] "TypeScript," 17 August 2019. [Online]. Available: https://en.wikipedia.org/wiki/TypeScript. [Accessed 20 July 2019].

[12] "Using TypeScript with React Native," Facebook, 7 May 2018. [Online]. Available: https://facebook.github.io/react-native/blog/2018/05/07/using-typescript-with-react-native. [Accessed 20 July 2019].

[13] "Microsoft augments JavaScript for large-scale development," InfoWorld, 1 October 2012. [Online]. Available: http://www.infoworld.com/d/application-development/microsoft-augments-javascript-large-scale-development-203737. [Accessed 20 July 2019].

[14] "Typechecking With PropTypes," Facebook, [Online]. Available: https://reactjs.org/docs/typechecking-with-proptypes.html. [Accessed 25 October 2019].

[15] "Introduction to type checking with Flow," Facebook, [Online]. Available: https://flow.org/en/docs/getting-started/. [Accessed 25 October 2019].

[16] "Context," Facebook, 2019. [Online]. Available: https://reactjs.org/docs/context.html. [Accessed 21 July 2019].

[17] "Introducing Hooks," Facebook, 2018. [Online]. Available: https://reactjs.org/docs/hooks-intro.html. [Accessed 22 June 2019].

[18] "Releasing React Native 0.59," Facebook, 12 March 2019. [Online]. Available: https://facebook.github.io/react-native/blog/2019/03/12/releasing-react-native-059. [Accessed 27 October 2019].

[19] "State and Lifecycle," Facebook, 2013. [Online]. Available: https://reactjs.org/docs/state-and-lifecycle.html. [Accessed 22 July 2019].

[20] T. Kol, "Moving Beyond Animations to User Interactions at 60 FPS in React Native," 2 March 2017. [Online]. Available: https://hackernoon.com/moving-beyond-animations-to-user-interactions-at-60-fps-in-react-native-b6b1fa0ba525. [Accessed 13 July 2019].

[21] "Animated," Facebook, 2018. [Online]. Available: https://facebook.github.io/react-native/docs/animated. [Accessed 13 July 2019].

[22] S. Boyar, "Let's Get Moving: An Introduction to React Native Animations — Part 2," 6 March 2019. [Online]. Available: https://medium.com/@shaneboyar/react-native-animated-tutorial-8543c9df4530. [Accessed 14 July 2019].

[23] "Animations. Using the native driver.," Facebook, 2018. [Online]. Available: https://facebook.github.io/react-native/docs/animations#using-the-native-driver. [Accessed 15 July 2019].

[24] "Rev up your workout with interval training," Mayo Clinic, 22 March 2019. [Online]. Available: https://www.mayoclinic.org/healthy-lifestyle/fitness/in-depth/art-20044588?pg=1. [Accessed 17 August 2019].

[25] "What is Reactotron?," Infinite Red, Inc. , 2019. [Online]. Available: https://github.com/infinitered/reactotron. [Accessed 12 September 2019].

[26] "GitHub Search," GitHub, [Online]. Available: https://github.com/search?p=2&q=stars%3A%3E100&s=stars&type=Repositories. [Accessed 26 October 2019].