Bachelor's thesis

Information and Communications Technology

2024

Hanna Järveläinen

Creating a React Native UI Component Library

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Information and Communications Technology

2024 | 39 pages

Hanna Järveläinen

Creating a React Native UI Component Library

The goal of the thesis was to create a private React Native UI component library using the components from an already-published mobile application. The commissioner had a need to create new applications with similar features, so it was seen beneficial to create a library so that the existing components could easily be reused.

The main technologies used to develop the library were already defined in the mobile application, including TypeScript, React Native, and Expo. An important addition was Storybook that was needed to render the components in isolation during the development. The components were tested using Jest. After the library was ready, it was published as a private package in GitLab.

As a result, a library with 16 components was published. The main requirements, including controlled access to the library and customizable styles, were achieved. Documentation was written for the users of the library as well as for possible future developers, enabling the possibility of adding more components in the future.

Keywords:

React Native, TypeScript, Storybook, mobile development, component library

Opinnäytetyö (AMK) | Tiivistelmä

Turun ammattikorkeakoulu

Tieto- ja viestintätekniikka

2024 | 39 sivua

Hanna Järveläinen

React Native UI-komponenttikirjaston kehittäminen

Opinnäytetyön tarkoituksena oli luoda yksityinen React Native UI-komponenttikirjasto käyttäen jo julkaistun mobiilisovelluksen komponentteja. Toimeksiantajalla oli tarve luoda uusia mobiilisovelluksia samankaltaisilla ominaisuuksilla, joten jo kehitettyjen komponenttien tuominen kirjastoon uudelleen käyttöä varten nähtiin hyödylliseksi.

Kirjaston kehittämiseen käytetyt pääteknologiat oli jo määritelty julkaistussa mobiilisovelluksessa. Näitä olivat TypeScript, React Native ja Expo. Sen lisäksi keskeinen työkalu oli Storybook, jota tarvittiin komponenttien renderöintiin kehityksen aikana. Komponentit testattiin käyttäen Jestiä. Kirjaston valmistuttua se julkaistiin yksityisenä pakettina GitLabissa.

Opinnäytetyön tuloksena oli julkaistu kirjasto, jossa oli 16 komponenttia. Kaikki tärkeimmät vaatimukset täytettiin, sisältäen kontrolloidun pääsyn kirjastoon sekä komponenttien tyylien muokattavuuden. Dokumentaatio luotiin sekä kirjaston käyttäjille että mahdollisille tuleville kirjaston kehittäjille, mahdollistaen kirjaston laajentamisen uusilla komponenteilla tulevaisuudessa.

Asiasanat:

React Native, TypeScript, Storybook, mobiilikehitys, komponenttikirjasto

Content

List of Figures

List of Tables

List of abbreviations

CLI	Command Line Interface
DOM	Document Object Model
OS	Operating System

1 Introduction

The purpose of the thesis was to create a UI component library from the components of the commissioner’s two existing mobile applications, OmaToimari and DigiHelppari. The commissioner of the thesis is Gamy Ry that helps social and healthcare organizations in digitalization. The commissioner has needs to create several mobile applications for different organizations based on the same needs as the former applications but tailored to each organization’s specific needs. This created the idea of creating a library of the components that had already been created, so they could be used again in future applications. The library ensures uniformity and efficiency, resulting in higher quality and a smaller workforce needed for the future applications.

A UI component library is a codebase that can be installed to other, consuming projects. The wanted components can be imported and used in the code to make the development of the application faster. The components are like building blocks that can be taken as many times as wanted and placed in the application to the desired position as demonstrated in Figure 1. The components’ functionalities in the library are tested comprehensively and individually. Consequently, the testing can be centered around the business logic during the development of the library consuming application.

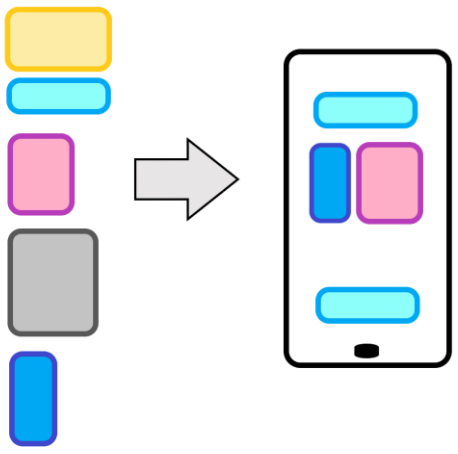


Figure 1. Component library and its components used in a mobile application.

First, the main technologies used in the library are introduced. Then, the requirements set for the project are listed and some of them are shortly discussed. After that, the architectural decisions are justified. The implementation of the project is covered from creating the project to the publication. Testing methods are discussed, and some testing cases are gone through. Finally, the outcome is evaluated, and further development possibilities are proposed.

2 Technologies

In this chapter, the main technologies used in the development of the library are discussed.

2.1 JavaScript

JavaScript is the most popular programming language, with 63.61 % of software developers stating to use it in 2023 [1]. It was originally created to make web pages dynamic, but nowadays it is used widely in different environments, including server-side scripting and mobile applications [2]. JavaScript evolves continuously, and since 2015 a new version has been published yearly.

2.2 TypeScript

TypeScript is a programming language build on JavaScript to detect the errors in the code without running the code [3]. Every value in the code is typed either by TypeScript or the programmer, and TypeScript checks that these values match their types. Running JavaScript code can lead to unexpected behavior without giving any errors, such as logic errors or having undefined as a value. This is why TypeScript can be a valuable tool to find the errors and save time from debugging the code. As TypeScript type checks the code before the code is run, it is not used during the runtime. After the types are checked, the code is compiled to JavaScript that has no information about the types. Declaration files that contain the information about types can be created during the compiling, so that TypeScript can be used in the library consuming projects [4]. TypeScript was used by 38.87 % of software developers in 2023 [1], which is much less than the use of JavaScript. Although it can make the programming faster by preventing errors, every TypeScript user must first learn to use JavaScript, which might explain the lower usage.

2.3 Node.js

Node.js is a JavaScript runtime environment [5]. A runtime environment is always needed to be able run JavaScript code. Node.js runs the same JavaScript engine as Google Chrome, which makes it highly performant. Before Node.js, JavaScript could be run only in browsers. With Node.js, it is possible to write backend applications using JavaScript, but it is also needed to be able to use many JavaScript libraries, like React that was used in this project.

2.4 Libraries

A JavaScript library is a set of code that can be used in other projects [6]. It usually concentrates on a certain topic. As a library can be used by other developers than the ones who developed it, good documentation is important. In addition, a credible library is also tested well before publication. A common way to publish a JavaScript library is to publish it as a npm package, as was done in this project. Packages can be published privately, so the consumers of the library can be controlled, or publicly, so that anyone can use it in their projects. npm is a company that supports development with JavaScript [7]. Its main purpose is to offer tools to share packaged JavaScript code with others. The packages can be published in npm’s own package registry or any other compatible package registry [8]. npm also offers npm CLI; a package manager for node projects that runs from the terminal.

2.5 React

React is a JavaScript library that lets the application to be built from reusable components, which makes it easy to scale and maintain [9]. In addition, it creates a virtual DOM tree that it updates the least possible, which makes it faster to re-render. React is usually written with JSX syntax, that is a markup that lets the JavaScript code use HTML tags and React components [10]. To use JSX syntax with TypeScript, the files have to have `.tsx` extension [11]. Node.js is needed at least for local development [12].

2.6 React Native

React Native is a framework for building native applications for Android and iOS with JavaScript [13]. The application is built with React and native components that are accessed through React Native. The application is developed with JavaScript, but it is rendered using the platform’s native APIs. With React Native it is possible to develop an application in a single codebase that works both on Android and iOS, and platform-specific parts can be incorporated in the same project.

2.7 Expo

Expo is a framework for native mobile and web applications built with React Native [14]. It offers several different developing tools and features, such as access to camera and GPS location. For creating the library, the most essential Expo feature was Expo CLI that provides an easy way of starting a server and the app on a mobile emulator or a mobile device for development purposes.

2.8 Android Emulator

Android Emulator provides virtual Android devices that can be run on a computer to test applications on a native Android platform [15]. It offers different devices and Android API levels, so the application can be easily tested on different screen sizes and Android versions.

2.9 Storybook

Storybook is a development tool that renders UI components in isolation [16]. In addition, it has a control panel where the properties of the component can be easily modified. This way the components can be developed and tested individually detached from the business logic of an application.

2.10 GitLab

GitLab is a platform that provides a wide range of features for the whole process of developing a software [17]. It concentrates around DevOps, but the central features of GitLab related to this project were the code repository and package registry.

3 Requirements

In this chapter, the requirements set by the commissioner are presented. At first, the UI components of two different mobile applications, OmaToimari and DigiHelppari, were planned to be brought to the library. However, considering the amount of work and the available resources, the whole DigiHelppari application was excluded from the project. The components of OmaToimari application were prioritized using MoSCoW method to ensure the most essential ones would be brought to the library, and some other features were added to the requirements as well. In MoSCoW method, the requirements are ordered using four different priority levels, starting from must have and ending to won’t have priorities [18]. The requirements were listed and prioritized as shown in Table 1.

Table 1. Requirements with their priorities.

Requirements	Priority
The library can be extended and has documentation about how to do it	must have
The access to use the library is controlled	must have
Questionnaire components	must have
Components’ styles can be modified	must have
Questionnaire summary components	should have
Planning components	should have
Quiz components	could have
Databank components	could have
Calendar components	could have
Profile components	could have
Components in creating a profile screen	could have
Menu components	could have
Plan components	could have
Accessibility features	won’t have
DigiHelppari app components	won’t have

Some of the requirements are discussed below shortly.

3.1 The access to use the library is controlled

OmaToimari is a health-related application, so it was seen important that the consumers of the library could be controlled to prevent any external user from creating a similar looking application.

3.2 Components’ styles can be modified

The library concentrates on UI components, and it was important that the components would look stylish by default, so the components were brought to the library with the style from OmaToimari application. However, it was seen crucial that the future applications could have their own, distinctive style, so the default styles of the components should be able to be overwritten.

4 Architecture of the module library

The central architectural decisions of the development of the library are discussed in this chapter. They include which programming language to use, what tool to use to render the components, and where to publish the library.

4.1 Programming language

OmaToimari application was created using React Native and TypeScript, so there were two options for the programming language for the library: JavaScript and TypeScript. According to React Native's own documentation, the default programming language for React Native projects is TypeScript [19]. Also, Expo supports TypeScript by default. As discussed earlier, TypeScript will be compiled to JavaScript with declaration files before publishing, so a library created with TypeScript can be consumed both in JavaScript and TypeScript projects, therefore TypeScript was chosen as the programming language for the library.

4.2 Rendering tool

When developing a UI component library, there is no application and designated position in the application where a component will be placed. However, the developer must see and be able to test the component while developing it, so a tool was needed to render the components during development. After research, two tools compatible with React Native were found: Storybook and Bit. Both tools were created originally for web development, and the usage was not as smooth with mobile applications with either of them. After studying and testing both options, Bit was found to be more complex, and it concentrated more on sharing and collaboration tools in a bigger developments team, so Storybook was found to be more suitable and was selected for the project.

4.3 Publication platform

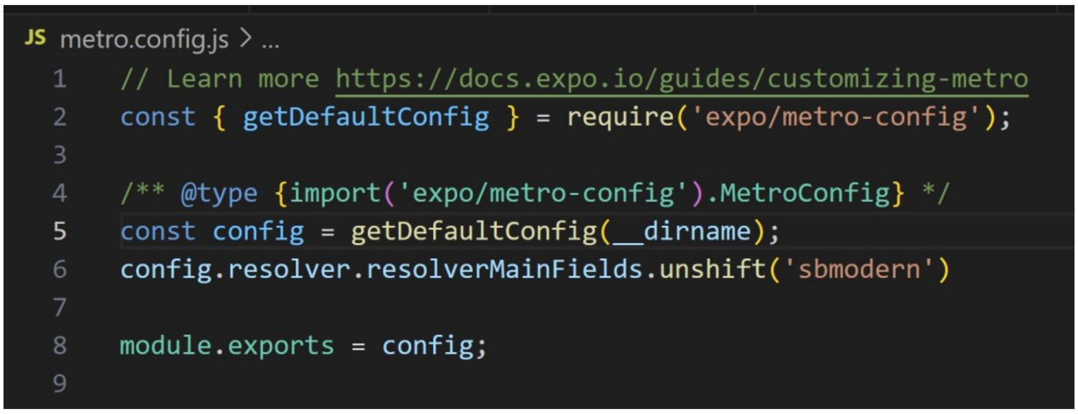
The library had to be published as a package so that other projects could install it and consume it. npm has the most popular JavaScript package registry with public and private packages [20], so it was considered as the first option for the publication platform. However, to be able to publish private packages, a Pro account is needed. One of the most important requirements was that access to the library must be limited, so due to the monthly cost of the Pro account, npm was discarded and other solutions had to be investigated. Turku University of Applied Sciences has its own GitLab domain name where the project repository was stored as well. It was found out that GitLab has a package registry feature with controlled access that would not incur any additional costs, and it also works with npm CLI. The package registry was tested, and it worked well for the needs of the project, so GitLab was chosen to be the package registry platform.

5 Implementation of the library

This chapter discusses the process of creating the library. General points are discussed in their own chapters, and components brought to the library are discussed shortly in their own chapters. Components are divided into chapters based on the screens where they are located in the original application, as they are listed in the requirements as well.

5.1 Creating the base project

First, a new project was created with Expo's app template for React Native and TypeScript. Once the sample project was running on Android Emulator, Storybook was installed to the project. To get Storybook work with React Native, `metro.config.js` file had to be modified as shown in Figure 2.



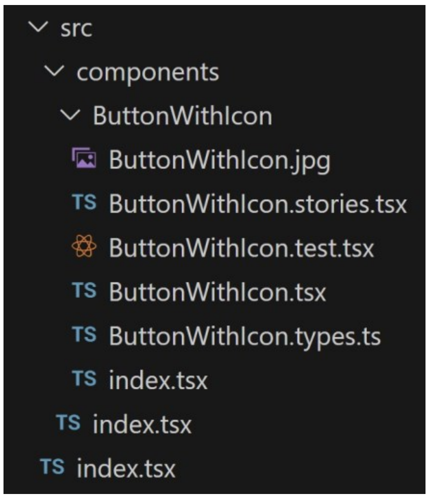
```
JS metro.config.js > ...
// Learn more https://docs.expo.io/guides/customizing-metro
const { getDefaultConfig } = require('expo/metro-config');

/** @type {import('expo/metro-config').MetroConfig} */
const config = getDefaultConfig(__dirname);
config.resolver.resolverMainFields.unshift('sbmodern')

module.exports = config;

```

Figure 2. Content of metro.config.js file.

The entry point of an Expo project is inside the Expo's package under `node_modules` where it registers the `App.tsx` file as the root component. As it is not good practice to modify the code inside an installed package, Storybook had to be exported in `App.tsx` to get Expo run the components, or stories, added to Storybook. After the Storybook's example component was working on Android Emulator, ESLint and Prettier were installed to the project. ESLint is a tool that checks that code follows rules set for it [21], and Prettier is a code formatter that goes through the code and rewrites it to follow a preset style [22]. Several ESLint plugins were added to support the use of TypeScript, React, and Jest.

5.2 Directory structure

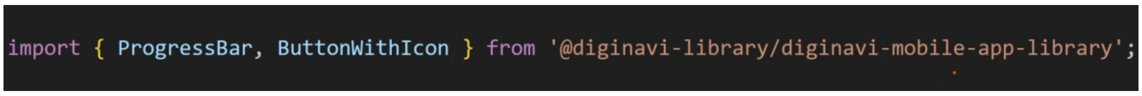
`src` directory was created to the root of the project to contain all the components to be added to the library. `components` directory was created under it, and for every component, a separate folder was created. That folder would contain all the files related to it: React Native component file, type file, Storybook file, index file, test file, and the example image of the component that was also used in the documentation of the component. Only the images imported to the components were placed in a separate `assets` folder. The example structure of a component's path and files related to it are shown in Figure 3.



```
src
  components
    ButtonWithIcon
      ButtonWithIcon.jpg
      ButtonWithIcon.stories.tsx
      ButtonWithIcon.test.tsx
      ButtonWithIcon.tsx
      ButtonWithIcon.types.ts
      index.tsx
    index.tsx
  index.tsx
```

Figure 3. Directory structure.

In addition, an index file was added to the `components` folder and `src` folder. On every level of the directory, the index files were used to export the components to the higher directory. As a result, the consumer of the library can import all the needed components simply by using the library name (Figure 4) instead of having to import every component separately and to know the path of every component.



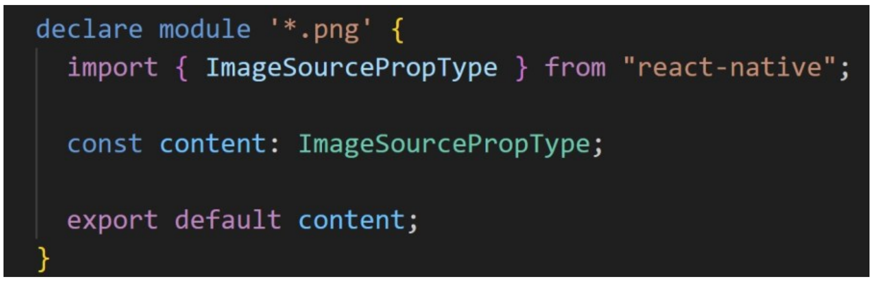
```
import { ProgressBar, ButtonWithIcon } from '@diginavi-library/diginavi-mobile-app-library';
```

Figure 4. Importing the components in the consuming project.

5.3 Restructuring and updating the code

The code style in the original application was not consistent; it contained class components and function components, and the structure of the component files varied. All the components added to the library were restructured to follow the same coding style and structure. All components were changed to arrow function components, types were moved to separate type files, component specific styles were moved to the bottom of the component file, component properties were destructured as they were passed to the component, and repetitive code was restructured by simplifying it. In addition, some components were combined either to simplify the structure and enhance readability or because they were so similar that most of the code in them was identical.

All the types were checked. Many types of component properties were referring to the data of the original application, so they had to be generalized. Types of properties passed to React and React Native built-in components were checked and updated to follow the types of the built-in components' properties. Importing images to component files gave different type and module errors, so a declaration file had to be created for images to match React Native's property type of images. The declaration file content is shown in Figure 5.



```
declare module '*.png' {
  import { ImageSourcePropType } from "react-native";

  const content: ImageSourcePropType;

  export default content;
}
```

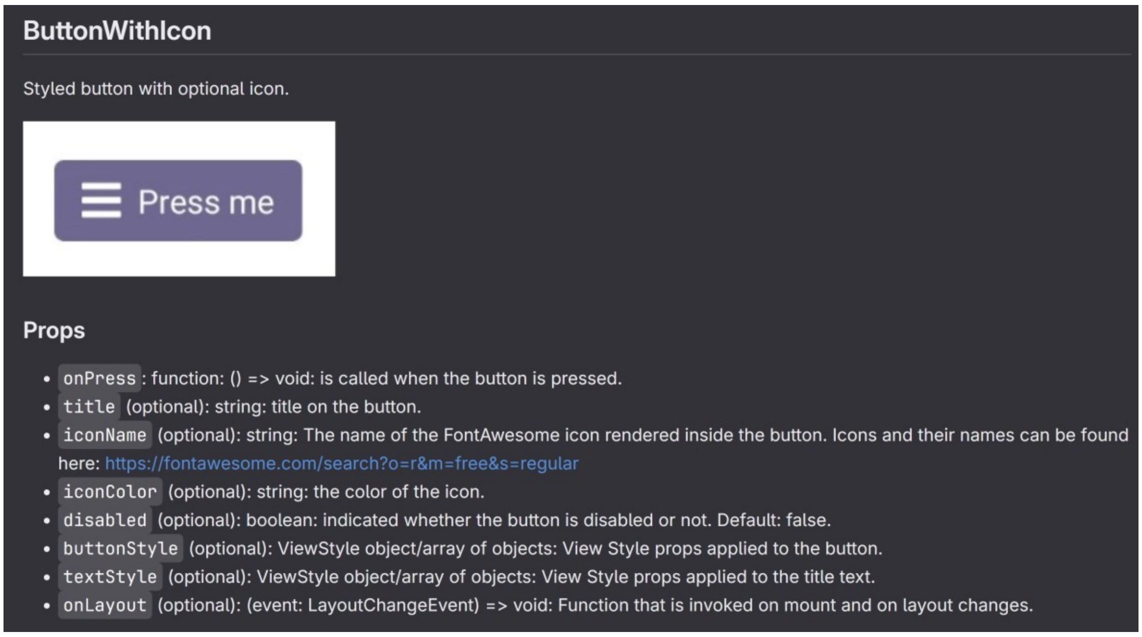
Figure 5. Declaration file for images.

All the third-party libraries used in the original application's components were checked and updated to newer versions.

Because most of the component names referred to their location in the original application, many components were renamed to give them a more general name that would describe their functionality better.

5.4 Documentation

All documentation of the project was written in `README` file. Instructions were written for how to install and use the library in a project, and how to get a local copy of the project repository from GitLab and how to get it running for further development and testing. A chapter was written for every component in the library. For every component, a short description was written, at least one example image was added, and all the component properties were listed with information about the input values with their types and optionality. Example inputs were given when thought necessary. An example of a component's documentation is shown in Figure 6.



ButtonWithIcon

Styled button with optional icon.

Props

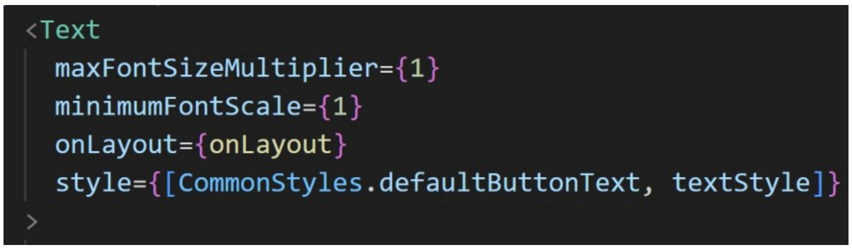
- `onPress` : function: () => void: is called when the button is pressed.
- `title` (optional): string: title on the button.
- `iconName` (optional): string: The name of the FontAwesome icon rendered inside the button. Icons and their names can be found here: https://fontawesome.com/search?o=r&m=free&s=regular
- `iconColor` (optional): string: the color of the icon.
- `disabled` (optional): boolean: indicated whether the button is disabled or not. Default: false.
- `buttonStyle` (optional): ViewStyle object/array of objects: View Style props applied to the button.
- `textStyle` (optional): ViewStyle object/array of objects: View Style props applied to the title text.
- `onLayout` (optional): (event: LayoutChangeEvent) => void: Function that is invoked on mount and on layout changes.

Figure 6. Example of a component's documentation.

In addition, JSDoc comments were added for every component. The comments included the same property descriptions as in `README` file, so that the descriptions could be seen by the consumer of the library while coding.

5.5 Styles

All the components in the library were created with default style taken from the OmaToimari application. One of the most important requirements was to be able to modify the style so that the future applications could have their own distinctive styles. The challenge was to create clear-to-use components without too many properties but to keep them as versatile as possible. Therefore, all the style properties of the components inside the library components were analyzed to decide which ones were essential to create distinctive style. Optional style object properties with descriptive names were created for all these styles. These style objects were passed to the subcomponents as arrays where the first item contained the default style object, and the second item contained the customized style object as shown in Figure 7. In this way, the custom style would overwrite the original style attributes in case they were overlapping.



```
<Text
  maxFontSizeMultiplier={1}
  minimumFontScale={1}
  onLayout={onLayout}
  style={[CommonStyles.defaultButtonText, textStyle]}
>
```

Figure 7. A component with an array of style objects.

The style properties were decided to be style objects, so all the React Native's style attributes could be used to customize the style. As a result, the style of the components could be customized extensively, as shown with an example component in Figure 8.

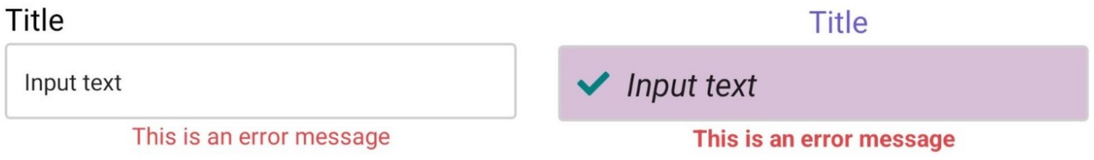


Figure 8. A component with default and customized styles

5.6 Questionnaire components

The process of bringing the components in questionnaire screen to the library is discussed in this chapter. There were three components to be brought to the library: `ProgressBarButton`, `ProgressBar`, and `QuestionAnswerArea`. The components with their original and new names are lister in Table 2.

Table 2. Components in questionnaire screen.

Original component name	**Component name in the library**
ProgressBarButton	StyledButton
ProgressBar	ProgressBar
QuestionAnswerArea	AnswerSlider

`ProgressBarButton` component was renamed `StyledButton`. The original component file contained two fairly simple components. The first one was used only in the second component, it was not exported from the file, so the two components were combined to make the code more readable. Customizable style properties were added for the text, the button container, and the shadow.

`ProgressBar` component relied heavily on the data of the original application, and it also contained two buttons with many hard-coded options for the text in the buttons. To make the component more versatile, the buttons were removed, and the properties of the components were simplified so that they only included the total number of circles, the number of colored circles, and the color of the circles. Most of the original code consisted of evaluating the buttons' texts to be shown and onPress callbacks for pressing the buttons, so the component code simplified remarkably.

`QuestionAnswerArea` was renamed `AnswerSlider`. The original component contained text above and under the slider referring to the original application, so the component was simplified by removing these texts to leave only the core slider functionality: slider with a thumb, pressable plus and minus icons, and the chosen value. The original slider was not allowed to go to zero, so a Boolean property was added to the library version to determine whether zero was an allowed value or not. In addition, maximum value property was added, so the range could be modified. Many properties came from a complex object type including properties that were not needed anymore, so the object was changed to two simple properties: start value and on change callback.

Finally, ten different style objects and color reference properties were added due to the complexity of the component.

5.7 Questionnaire summary components

Questionnaire summary screen included the components listed in Table 3.

Table 3. Components in questionnaire summary screen.

Original component name	**Component name in the library**
QuestionnaireHistoryGraph	HistoryGraph
QuestionnaireSingleGraph	GroupGraph
SummaryButtonBar	SummaryButtonBar
QuestionnaireSummaryGraph and QuestionnairePreviewContent	ButtonBarGraph
ScrollViewWithIndicator	ScrollViewWithIndicator

The similar modifications were applied to the components listed in Table 3 as for questionnaire components as explained in chapter 5.6. The code of two components, `QuestionnaireSummaryGraph` and `QuestionnairePreviewContent`, was so similar that these components were combined into one component.

5.8 Planning components

The components in planning screen are listed in Table 4.

Table 4. Components in planning screen.

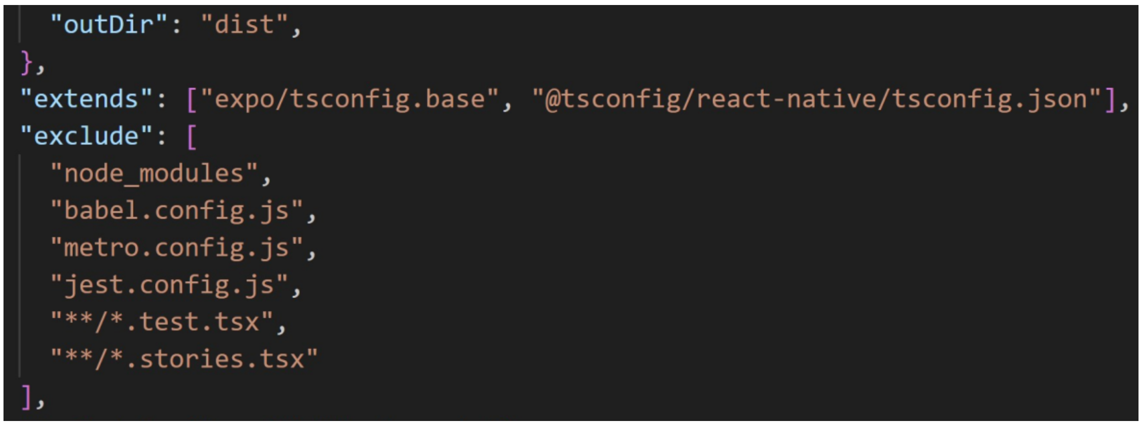
Original component name	Component name in the library
QuestionnaireSelectThreeToPlan	SelectionList
Checkbox	Checkbox
QuestionnairePlanTabContent	ListContainer
QuestionnairePlanDateModal	SelectDays
QuestionnaireAddPlanModal and QuestionnaireEditPlanModal	InputModal

The similar modifications were applied to the components listed in Table 4 as for questionnaire components as explained in chapter 5.6. The code of two components, `QuestionnaireAddPlanModal` and `QuestionnaireEditPlanModal`, was so similar that these components were combined into one component.

5.9 Publishing

The process of creating a build of the project and publishing it is described in this chapter. Modifications had to be made in `tsconfig.json` and `package.json` files, and configuration had to be created to set the package configuration to point to GitLab's package registry.

Before creating the build, `tsconfig.json` file was updated to follow TypeScript's own recommendations for library projects [4]. Most notable change was to add a setting for declaration files to be created. Declaration files are needed to bring the types to the consuming project, since the types are removed from the files as they are converted to JavaScript files. As TypeScript's `tsc` command was needed to create the declaration files in addition to type checking, `tsc` was also used for creating the build. Test and Storybook files were excluded from the build as shown in Figure 9.



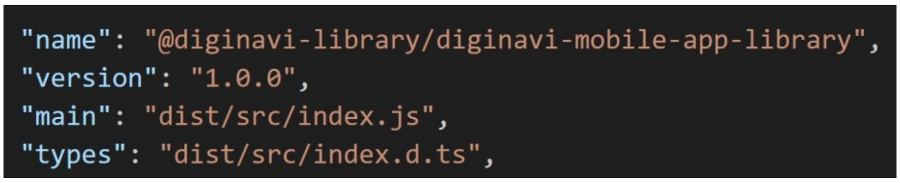
```
    "outDir": "dist",
  },
  "extends": ["expo/tsconfig.base", "@tsconfig/react-native/tsconfig.json"],
  "exclude": [
    "node_modules",
    "babel.config.js",
    "metro.config.js",
    "jest.config.js",
    "**/*.test.tsx",
    "**/*.stories.tsx"
  ],
```

Figure 9. Part of tsconfig.json configuration.

At first, `tsc` command did not output any files, it only checked types. Finally, the reason was found from React Native's base TypeScript configuration that was extended in the project's `tsconfig.json` file. React Native applications are usually compiled with Babel, so `noEmit` was set to false in the base configuration preventing `tsc` from compiling files. This setting could be overwritten by setting `noEmit` option to true. In addition, `outDir` folder was set so that the files to be published would be in a separate folder from the development files.

Also `package.json` had to be modified. Dependencies were checked and updated. Most of the dependencies were listed right, but some essential dependencies, like React and React Native, were moved under peer dependencies. `Private` setting was true by default, and it had to be changed to false to be able to publish the library. `Files` configuration was added to determine which files would be included in the published package. Only the compiled files were added to the list.

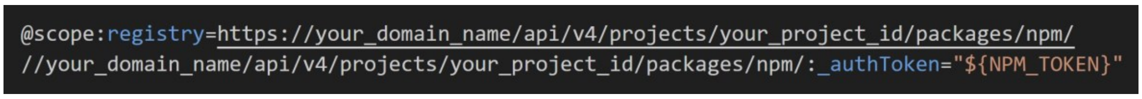
The project was renamed `@diginavi-library/diginavi-mobile-app-library` to follow GitLab's instructions for publishing an npm package in their package registry [23]. `Main` and `types` had to be updated to point to the build's main and type files. These modifications are shown in Figure 10. `PublishConfig` was added with registry set to the project's GitLab package registry URL.



```
"name": "@diginavi-library/diginavi-mobile-app-library",
"version": "1.0.0",
"main": "dist/src/index.js",
"types": "dist/src/index.d.ts",
```

Figure 10. Part of package.json configuration.

To be able to publish a package in GitLab, a personal access token had to be created with read and write package registry permissions. `.nmprc` file was created to the root of the project with content following GitLab's instructions as shown in Figure 11 [23].



```
@scope:registry=https://your_domain_name/api/v4/projects/your_project_id/packages/npm/
//your_domain_name/api/v4/projects/your_project_id/packages/npm/:_authToken="${NPM_TOKEN}"
```

Figure 11. Example content of .npmrc file

After the configurations were done, the library was published as a package using the command line. Semantic versioning was used, and after several test publications, an official version 1.0.0 was published.

6 Testing

React Native’s testing documentation goes through component testing that is put under unit and integration testing categories [24]. The library consists of only UI components that don’t depend on other sources, so most of the tests written for the library could be considered unit tests. Some components were created using other components in the library, so testing of these components could be seen as integration tests as well. As the library consists of only UI elements without business logic, the above-mentioned tests were seen sufficient.

Jest is a popular JavaScript testing framework that works with TypeScript, Node.js, and React [25]. In addition, React Native’s own documentation guides to use Jest for testing [24]. Furthermore, Expo has documentation and additional packages to support testing with Jest [26], so Jest was chosen to be the testing framework for this project. To be able to simulate user interactions in tests, React Native Testing Library was added, too. Its API contains methods for example to fire events like changing text input or pressing components [24]. However, native features cannot be tested with React Native Testing Library, since the tests will not run on a mobile operating system but in Node.js environment. In addition, screen size cannot be mocked because of the same reason.

The original OmaToimari mobile app did not have any written tests, so all the tests for this project had to be written from the beginning. However, OmaToimari had been tested several times manually by the development team, customer, as well as external testing team, so no big bugs were expected to be found.

In total 164 tests were written. Next, different test cases will be discussed on a general level. As suggested by Dodds [27], the tests were written to rely on the elements the same way as a user would rely on them whenever it was possible. This means that for example a button would be accessed by finding the text on it instead of finding it by a class name that the user will not see. Nevertheless, many elements had to be accessed using the components' `testID` property. For example, pressing a button that only has an icon on it could not be accessed without a `testID` property.

Testing the components with graphs was challenging since the graphs were rendered as images without any properties that could be found using the Jest queries. Graphs were tested mainly visually using Android Emulator, and Jest could be used only to check that correct number of child components are rendered when accessing the parent element that contained the graph image. Also, testing icons had similar challenges. Although `FontAwesome5` components used in the project accepted `testID` property, icons were rendered as empty `Text` components without any properties. Because of this, icons were tested the same way as graphs.

Styles were tested to check that the custom styles get combined with the default style and that custom styles are applied over default styles. In addition, it was tested that an array of style objects could be passed to the built-in components.

React Native's `Modal` component has an `onRequestClose` property which is called when the user taps the hardware back button on Android. This could not be tested using Jest since it is a native feature, so it was tested manually using Android Emulator.

When checking that a correct item in a selection list is checked with a check mark as demonstrated in Figure 12, dynamic `testID`s were created as shown in Figure 13 to be able to first query for the parent component that contained both the text and the check mark.

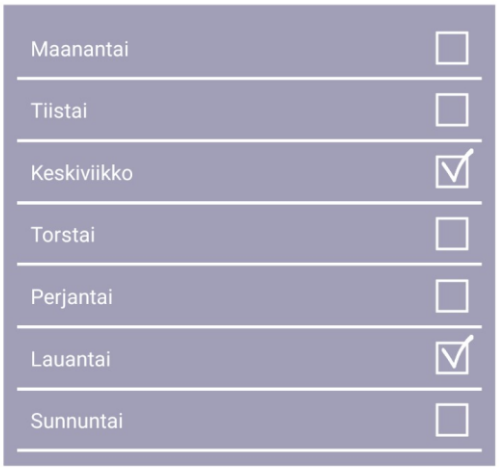
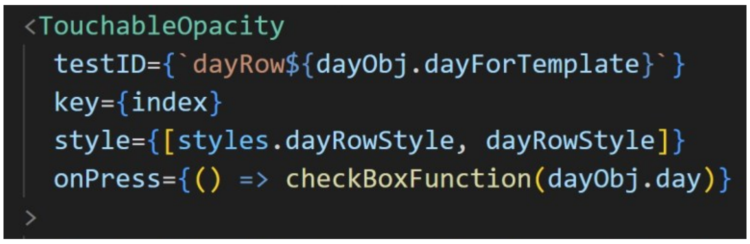


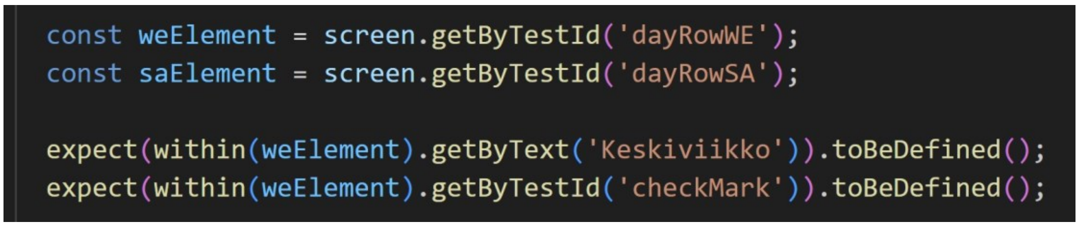
Figure 12. SelectDays component with two days checked.



```
<TouchableOpacity
  testID={`dayRow${dayObj.dayForTemplate}`}
  key={index}
  style={[styles.dayRowStyle, dayRowStyle]}
  onPress={() => checkBoxFunction(dayObj.day)}
>
```

Figure 13. Parent component with dynamic testID property.

After that, it could be verified that the `Text` component with correct text and a check mark image were both present in the parent component as shown in Figure 14.



```
const weElement = screen.getByTestId('dayRowWE');
const saElement = screen.getByTestId('dayRowSA');

expect(within(weElement).getByText('Keskiviikko')).toBeDefined();
expect(within(weElement).getByTestId('checkMark')).toBeDefined();
```

Figure 14. Querying a parent element with a dynamic testID and accessing the child elements.

After the library was published, the published package was tested by installing it to a new project, importing and using all the components in the code, and checking that the components work when the project was run on Android Emulator.

7 Conclusion

The goal of this thesis was to create a private UI component library from components of already existing mobile applications. All the “must have” and “should have” prioritized requirements were fulfilled. In total, 16 components with customizable styles were brought to the library and tested. Documentation was created for both the future consumers and possible developers of the library. Furthermore, the library was published as a private package with controlled access.

As the library relies on other JavaScript libraries and frameworks, such as React Native and Expo, the library must be maintained and updated as these dependencies are updated to ensure that it works with the newer versions that will likely be used in the future consumer applications.

For future development, more components could be brought to the library from the original applications. In addition, when new components are created for commissioner’s new applications, they could be first created and tested in isolation in the library, and after that be brought to the consuming application. This way the components would be available for other applications as well with the same workload, and the testing of the application could concentrate on testing the business logic, as the functionality of the component has already been tested in the library.

The process of creating a library is different from creating an application. When creating a library, the result will be used by another developer in another project. When the needs of the consumer are not known, the developer of the library must take into consideration different possibilities and balance between versatility and simplicity. Also, the configuration is different, and there are less sources for creating and configuring a library than an application. This led to many trials and errors especially in the publication stage, and TypeScript with hundreds of configuration options added challenge in locating the problems. Furthermore, the frontend development tools are mainly created for web development, and they don’t work as seamlessly with mobile app development and may lack features and make configuration and debugging more time consuming. Despite the large number of challenges, they were all conquered and a working library was achieved.

During this thesis, I learned about the differences between developing an app and a library; how it affects the configuration and how in the library versatility needs to be considered. I had not done much testing before this project, so I gained a lot of knowledge in testing.

Looking back at the project, I would do a little test publication at an early stage, with only one component, to have less issues to debug if the library doesn’t work. I tested different tools before choosing the ones used in the project, so I am confident that I chose the best ones for the needs of the project and would not change them if I had to restart from the beginning.

8 References

[1] Statista, "Most used programming languages among developers worldwide as of 2023," June 2023. [Online]. Available: https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/. [Accessed 28 September 2023].

[2] Mozilla, "JavaScript technologies overview," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/JavaScript_technologies_overview. [Accessed 15 January 2024].

[3] Microsoft , "TypeScript for the New Programmer," [Online]. Available: https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html. [Accessed 15 January 2024].

[4] Microsoft, "Modules - Choosing Compiler Options," [Online]. Available: https://www.typescriptlang.org/docs/handbook/modules/guides/choosing-compiler-options.html#im-writing-a-library. [Accessed 25 January 2024].

[5] OpenJS Foundation, "Introduction to Node.js," [Online]. Available: https://nodejs.org/en/learn/getting-started/introduction-to-nodejs. [Accessed 26 January 2024].

[6] R. Ruk, "How to start with JavaScript library development," 24 August 2020. [Online]. Available: https://dev.to/bornfightcompany/how-to-start-with-javascript-library-development-40mb. [Accessed 29 January 2024].

[7] npm, "About npm," [Online]. Available: https://www.npmjs.com/about. [Accessed 14 February 2024].

[8] npm Docs, "npm," 12 January 2023. [Online]. Available: https://docs.npmjs.com/cli/v10/commands/npm. [Accessed 14 February 2024].

[9] O. Hutsulyak, "10 Key Reasons Why You Should Use React for Web Development," TechMagic, [Online]. Available:

https://www.techmagic.co/blog/why-we-use-react-js-in-the-development/#:~:text=React%27s%20component%2Dbased%20architecture%20allows,maintainability%2C%20scalability%2C%20and%20flexibility.. [Accessed 16 January 2024].

[10] Meta Open Source, "Your First Component," [Online]. Available: https://react.dev/learn/your-first-component. [Accessed 16 January 2024].

[11] Microsoft, "JSX," [Online]. Available: https://www.typescriptlang.org/docs/handbook/jsx.html. [Accessed 15 January 2024].

[12] Meta Open Source, "Start a New React Project," React, [Online]. Available: https://react.dev/learn/start-a-new-react-project. [Accessed 16 January 2024].

[13] Meta Open Source, "Core Components and Native Components," [Online]. Available: https://reactnative.dev/docs/intro-react-native-components. [Accessed 29 January 2024].

[14] Expo, "Core concepts," [Online]. Available: https://docs.expo.dev/core-concepts/. [Accessed 29 January 2024].

[15] Google, "Run apps on the Android Emulator," [Online]. Available: https://developer.android.com/studio/run/emulator. [Accessed 29 January 2024].

[16] Chroma Software, "Why Storybook?," [Online]. Available: https://storybook.js.org/docs/get-started/why-storybook. [Accessed 30 January 2024].

[17] GitLab, "Platform," [Online]. Available: https://about.gitlab.com/platform/. [Accessed 30 January 2024].

[18] K. Brush, "MoSCoW method," TechTarget, March 2023. [Online]. Available: https://www.techtarget.com/searchsoftwarequality/definition/MoSCoW-method. [Accessed 29 January 2024].

[19] Meta Open Source, "Using TypeScript," React Native, [Online]. Available: https://reactnative.dev/docs/typescript#:~:text=How%20TypeScript%20and%20React%20Native,tsc%20for%20newly%20created%20applications.. [Accessed 16 January 2024].

[20] npm, Inc., "npm," [Online]. Available: https://www.npmjs.com/. [Accessed 30 January 2024].

[21] OpenJS Foundation, "ESLint Core Concepts," [Online]. Available: https://eslint.org/docs/latest/use/core-concepts. [Accessed 23 February 2024].

[22] Prettier, "What is Prettier?," [Online]. Available: https://prettier.io/docs/en/. [Accessed 23 February 2024].

[23] GtiLab, "npm packages in the package registry," [Online]. Available: https://docs.gitlab.com/ee/user/packages/npm_registry/. [Accessed 25 January 2024].

[24] Meta Open Source, "React Native Testing," [Online]. Available: https://reactnative.dev/docs/testing-overview. [Accessed 10 November 2023].

[25] Meta Open Source, "Jest," [Online]. Available: https://jestjs.io/. [Accessed 10 November 2023].

[26] Expo, "Expo Docs," [Online]. Available: https://docs.expo.dev/develop/unit-testing/. [Accessed 10 November 2023].

[27] K. C. Dodds, "Making your UI tests resilient to change," 7 October 2019. [Online]. Available: https://kentcdodds.com/blog/making-your-ui-tests-resilient-to-change. [Accessed 25 January 2024].

[28] Callstack Open Source, "React Native Testing Library FAQ," [Online]. Available: https://callstack.github.io/react-native-testing-library/docs/faq. [Accessed 10 November 2023].

[29] Khan Academy, "What's a JS library?," [Online]. Available: https://www.khanacademy.org/computing/computer-programming/html-

css-js/using-js-libraries-in-your-webpage/a/whats-a-js-library. [Accessed 29 January 2024].

[30] npm Docs, "registry," 5 January 2023. [Online]. Available: https://docs.npmjs.com/cli/v10/using-npm/registry. [Accessed 14 February 2024].