Beselam Asefa

Building Android Component Library Using Jetpack Compose

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

7 March 2022

Abstract

Author:	Beselam Asefa
Title:	Building Android Component Library Using Jetpack Compose
Number of Pages:	42
Date:	7 March 2022
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Mobile Solutions
Supervisors:	Panu Puro, Senior Software Developer Toni Spännäri, Senior Lecturer

---

The goal of this project was to build a component library using Jetpack Compose, which is a modern Android UI toolkit. Jetpack Compose is written in Kotlin and the code can be reused or shared easily. In the world of software development, it is common to reuse or share a piece of code between different projects. It saves the development time and cost and makes the code easily maintainable. This component library is made for Elisa, a well-known, Finnish telecommunication company. The library provides commonly used UI components such as buttons, typography, theme, dialog, colours, snackbar, image cards, vertical list items and animations. Android developers at Elisa use this component to develop different products that follow the company's UI guidelines.

The thesis covers different approaches used to build component libraries based on the Architectural layer of Jetpack Compose. The component library was built using the Material layer of Jetpack Compose. This layer is the topmost layer of Jetpack Compose. The Material layer includes Material design components and the best practices and features used to build user interfaces such as accessibility features.

The component library provides many advantages for the client company users, developers, and product owners. Developers can access commonly used components from this library; therefore, they will not rebuild components from the ground every time when they need them. This saves the development cost and time. The components are built considering the Material Design System best practices; therefore, products using this library will have a better user satisfaction.

Keywords: Jetpack Compose, User Interface, Android Library

Contents

List of Abbreviations

XML: Extensible Markup Language. It is a markup language that defines a set of rules for encoding documents in both human-readable and machine-readable format.

UI: User Interface. It is a space where users interact with computers.

API: Application Programming Interface. It is a piece of software code which allows multiple programs to communicate.

APK: Android Application Package. It is a package used to distribute and install Android Application.

AAR: Android Archive. A binary distribution format.

MVC: Model-View-Controller. A software design pattern.

MVVM: Model-View-ViewModel. A software design pattern.

1 Introduction

It is common to use a component library in different web and hybrid mobile application projects, but it is not common in native mobile projects, and different reasons can be mentioned for this. The fact that native platforms are not being flexible for integrating the custom libraries can be one reason. Currently, both major mobile platforms (iOS and Android) have introduced a simple and fast way of building a user interface. Google has introduced Jetpack Compose for Android.

Jetpack Compose is a modern UI toolkit for building native Android UIs. Jetpack compose accelerates and simplifies UI development on Android. It also quickly brings applications to life with less code.

In the final year project, a component library is built using Jetpack Compose. Component library is a local or a cloud-based software package that contains a collection of reusable UI components. The library provides multiple components used to build Android applications. The library includes components, such as buttons, top app bars, theme, text, typography, alert dialog, snackbar, animations, Canvas draw path, image card and vertical list components. The library components support both dark and light themes and some components have multiple variants.

The library is built considering the design guidelines of Elisa, a well-known, Finnish telecommunication company. The library is used by Android developers at Elisa. The thesis will explain in detail about the fundamentals of Jetpack Compose, the building process of the library, the library integration in a real-world project, and the outcomes of using the library.

2 Fundamentals of Jetpack Compose

2.1 Introduction to Jetpack Compose and the Technology Stack

Innovations in Mobile application development often come in waves or trends. Whether the current trend is about language safety and stability, or performance improvements of the platform, it is always important to keep up with the trends, because some of them radically change the way of developing applications. Nowadays building declarative and reactive programming is a trending issue. Developers are moving away from the classic imperative Model-View-Controller (MVC) approach to a more reactive approach such as Model-View-ViewModel (MVVM), where data streams are updated within a ViewModel, and the UI reacts to the change by redrawing itself or simply updating sections of the UI. [1.]

The expectations around UI development have grown. Today, it is hard to build an application and meet what the user's needs without having an elegant UI with animation and motion. These requirements are things that did not exist when the old Android UI Toolkit was created. To address the technical challenges of creating a polished UI quickly and efficiently Google introduced Jetpack Compose, a modern UI toolkit that helps developers to develop a better native Android application with a better speed.

2.1.1 Jetpack Compose Technology Stack

Jetpack Compose is a complete rewrite of the Android UI toolkit from the ground up. Therefore, it is important to understand the different components that made up Jetpack Compose and how they are connected.

The development and device host are the two major parts that categorise the components that make Jetpack compose [1]. Figure 1 shows the two major building blocks of Jetpack compose and the technology stacks found in each category.

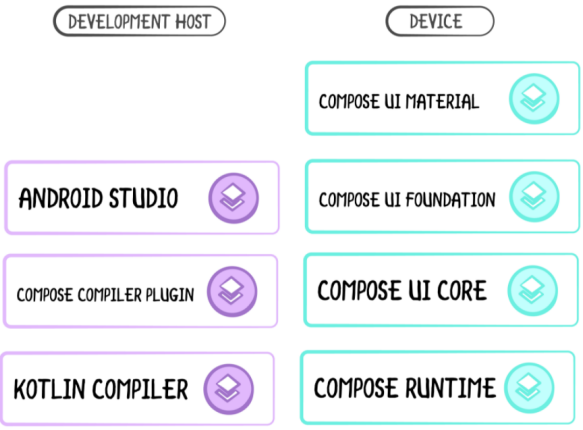

Figure 1 Jetpack Compose technology stack. [1].

- Development Host. The development host contains all the tools that help to write code. Jetpack Compose is written in Kotlin and uses Kotlin features, this makes Jetpack Compose so flexible and easy to use. At the bottom of Development Host there is Kotlin Compiler. It is used to compile Jetpack Compose source files into bytecode. On top of the Kotlin compiler, there is a Compose Compiler Plugin. This plugin works at the Type system level and at the Code generation level to change the types of composable functions. On top of the Compose Compiler Plugin, there is Android Studio, which includes Compose-specific tools, simplifying the work the developers do with Compose. [1.]

- Device Host. The second part of the Jetpack Compose technology stack is in the device. It is the environment that runs the Jetpack Compose code. As shown in figure 1, at the bottom of the device section, there is a Compose Runtime. At its core, the Compose logic does not know anything about Android or the user interface. It only knows how to emit specific items, and work with tree structures. Compose UI Core handles UI related tasks such as input management, measurement, drawing, and layout. The Runtime and UI core layers support the widgets that the next

layer provides. The Compose UI foundation contains basic building blocks such as Text, Row, Column, and default interactions. Finally, there is the Compose UI Material layer. It is an implementation of the Material Design system on the top of the Foundation layer. It provides Material components out of the box, making it easy to use Material Design in Android application development. [1.]

2.2 Benefits of Jetpack Compose

2.2.1 The Declarative Paradigm

Jetpack Compose uses a declarative paradigm. In a declarative way of programming, the expected result will be described instead of describing every step to achieve the goal. [1.]

The old Android UI system uses an imperative paradigm. Unlike the declarative paradigm, In the Imperative paradigm the programmer must describe the steps to change a state of a component. The imperative paradigm is complicated, and it requires more development time. For example, in the old Android UI system a RecyclerView component is used to render a list. It displays a large set of data efficiently. However, the RecyclerView component needs an Adapter for binding the data to the view and a ViewHolder for each item on the list. Rendering different types of data using RecylerView component can be more complicated and time-consuming.

In declarative programming, there is no need to focus on describing how a program should operate, but what the program should accomplish.

2.2.2 Composition

The old Imperative Android UI system is built based on inheritance. Each child view will inherit from their Parent View. When the new class inherits from an existing class, it reuses (or inherits) the existing class methods, and the new

class can add new methods and fields to a new situation. The relationship is always between a child and a parent, and it is always an “is” relationship. In the old android UI system, the TextView which is used to display text inherits the view class, so TextView is a View. [2.]

Jetpack Compose uses functions in place of classes. Therefore, the method of relationship is quite different from the old Android UI system. Jetpack Compose uses composable functions to display something on the screen. Composable function defines the application UI programmatically by describing how the UI should look and providing data dependencies [1]. To create a composable function that has a new feature, it only requires to nest other composable functions into an existing composable function.

Figure 2 illustrates composition in Compose. The blue composable is the parent and the green Composables are the children. When a new feature is required by the green composable it can be achieved by adding another composable inside the parent composable. The parent composable can have an infinite number of children which means it can have an infinite number of new features.

The parent composable can have a reference to the child composable, this way it can use the logic it needs from the child composable.

Composition is more flexible than inheritance since there is no rule on how many children it can have.

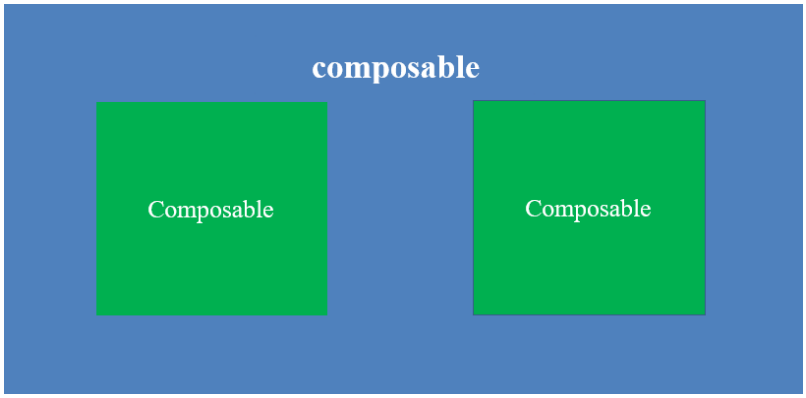

Figure 2 Composition in Jetpack Compose

2.2.3 Encapsulation

Encapsulation is the ability of an object to hide its data [3]. In the old Android UI toolkit, views can manage state, and they expose call-backs that can be used to capture the change in that state.

Compose is designed in a way that the UI is a representation of data. That means that the UI components are not responsible for managing the state. They represent the state

Compose has a unidirectional top-down data flow. As shown in Figure 3, data should flow down to the UI, and events should flow from the UI up. By following a unidirectional data flow, it is possible to decouple the composable that displays state in the UI from the parts of the application that store and change state. This increases the testability of the application and UI consistency. Figure 3 shows data in Jetpack Compose flowing from the data source down to the UI and the event flow from the UI up in Jetpack Compose.

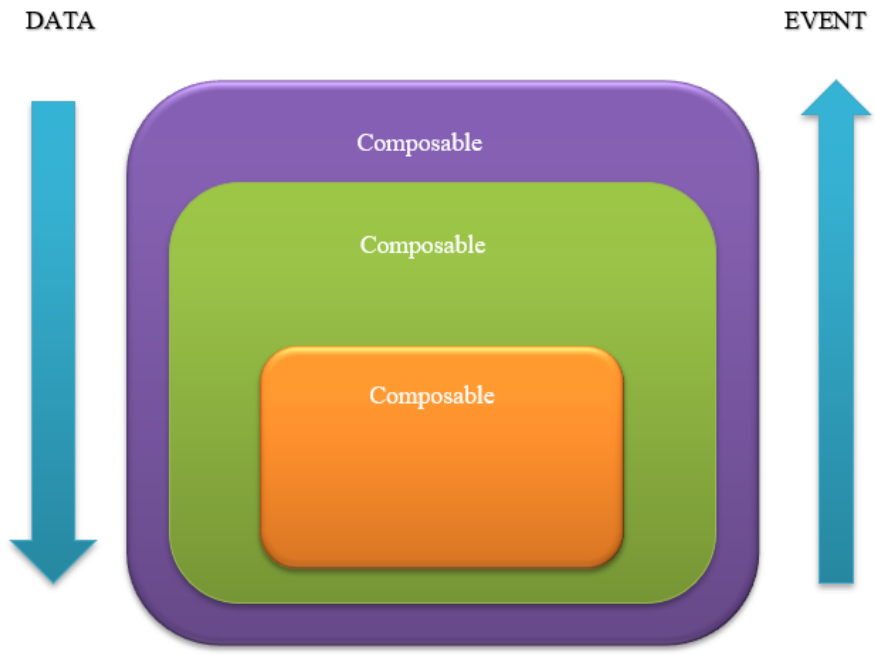

Figure 3 Data and event flow in Jetpack Compose. [1].

2.2.4 Recomposition

In Jetpack Compose, recomposition allows any Composable function to be re-invoked at any time to update its values. As shown in Figure 4, an event is created from the UI. The event created in the UI can make a change to the state that triggers a recompositing and update to the display value. Unlike in the old Android UI system, specifying an update call-back or a LifecycleOwner is not needed, as the Composable can implicitly serve as both. This is very useful when updating the UI with the new state.

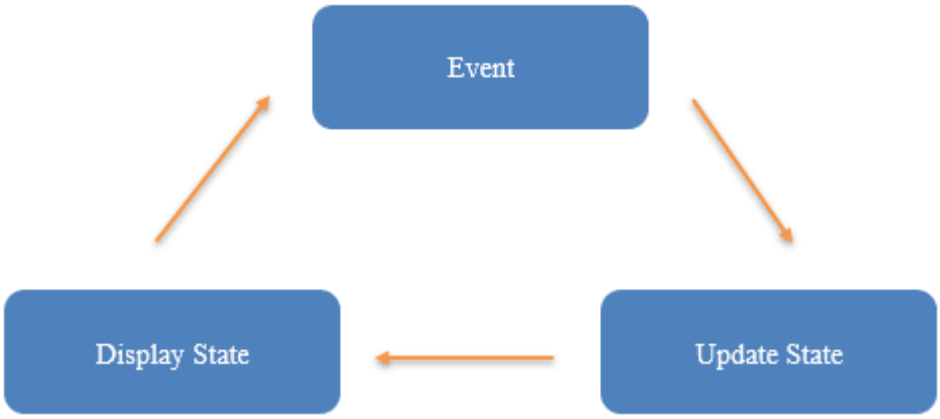

Figure 4. Jetpack Compose UI update loop. [1].

2.2.5 Modern Language

The old Android UI system is written in Extensible Markup Language (XML) which is a markup language used to describe data. It offers a standardised way to represent textual data [4]. Android developers must understand XML in addition to the Kotlin or Java programming languages to develop Android applications using the old UI system.

Jetpack Compose is written in Kotlin, for Kotlin. No longer are some parts written in Java or Kotlin, and some parts in XML. Instead, everything is Kotlin [1]. Therefore, there is no need to study an additional language for Android UI

development. It is also much easier to trace through code when all code is written in the same language.

Developers can take advantage of all the benefits of idiomatic Kotlin in every part of the application development. That includes modern language features, paradigms like functional programming, conciseness, and readability.

3 Android Module and Component Library

3.1 Introduction to Component Library

A component library is a local or a cloud-based software package that contains all the designed components of an application or a software. Component library promotes a single source of truth for the components and reduces code duplication. Therefore, developers using a component library work consistently and efficiently across multiple products. [5.]

The common approach to structure a component library is Atomic Design. Atomic Design is developed by Dave Olsen and Brad Frost. It is a methodology for building a design system with five main building blocks, which combines to promote scalability, modularity, and consistency. The Atomic Design is composed of atoms, molecules, organisms, templates, and pages. This methodology helps to build simple and maintainable component libraries by building large and complex components from smaller and simpler components. [6.]

Figure 5 shows the five fundamental building blocks of the Atomic Design methodology.

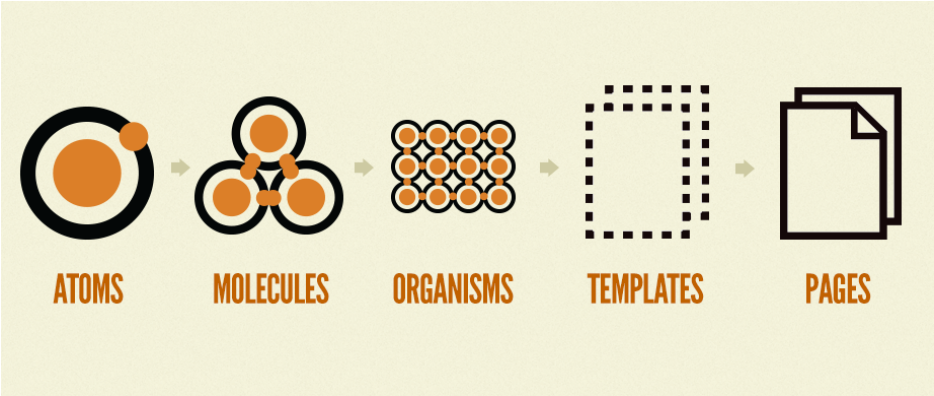

Figure 5. Elements of Atomic Design. [7].

A component is an element of a component library that could be built from the ground up or combining other existing components. Components usually contain some sort of logic or properties that can change their behaviour and style. Components can be dependent on a framework such as Jetpack Compose, Vue or React. If a project is an Android application, it makes sense to use Jetpack Compose to build the components. Building a page of an application using a component library requires integrating the library with the project and picking the required components from the library. It is like a puzzle; all the components are available in the library and the developer is expected to pick and combine the parts to build the project. The developer is also able to create new components or customise the existing ones. [7.]

The benefit of a component library is not based on the size of the implementing project. A small project has as much to gain from a component library as a larger project. A component library consists of reusable components. It could be a folder inside a project with common components used throughout the application. It could be a distributed package on a remote library repository such as Jitpack, npm or Maven Central. It could also be a part of a bigger design system.

A component library can be large, or it can be small. There are no official rules for what a component library means. It depends on the project and the developer needs. Component library can be a collection of components that is

created by a developer locally or a public library from a big company such as React, Vue, Material Design, Ant Design, and Semantic UI. [8.]

3.1.1 Benefits of Using a Component Library

A component library is a useful tool to have, most importantly for projects that should scale and develop fast. A project using a design system built from a component library leverages speedy and impressive growth. Building a component Library as part of a company design system consumes resources and time. However, by optimising development across many platforms, offering accessible, customizable, production-ready, and reusable code components such as typography, colours, image card, dialogs, and buttons. Component libraries promote faster development and growth. This will provide a long-term gain for the company. [5.]

The most important benefits of a component library include the following:

- Single source of truth. One of the main benefits of using a component library is that it serves as a single source of truth for the UI components. This reduces the risk of any variation between the company products. All the logic and styling can be found in a single place. This makes it easier to share. Developers can very quickly start a new project and have access to the components straight away. Changes to the library component will be reflected on all the projects using the component library. React is a primary example of a well-known open-source framework from Facebook that was initially developed as a component library for the company. [8.]

- Easy maintenance. By having all components in a single place, it makes it much easier to maintain. Updating the library components or adding new features to the existing components is done once and will be reflected on every implementation of that component. Releasing updates

is much faster and more consistent. It is also easier for new developers to get an overview of the project architecture. [8.]

- Testing. In a component library the components are grouped and implemented in one place. This makes it easier to create a well thought out test suite for the components. The tests become more applicable and important to have. When adding new components or features, it is also easy to use Test Driven Development by writing the new tests before implementing the feature. [8.]

- Customizable and flexible components. Recreating a component every time when requiring customization is time consuming and expensive. It also will have an impact on the project development. Using a component library helps to minimise this risk. Developers can reuse the components instead of recreating them. Having an easily customizable and reusable component will help a project to scale faster and be more flexible. For example: if users are not happy with the newly released image card style, the developer can change the image card style or functionality in the component library. This change will affect all image cards in the project. This way a developer can spend more time on creating other components, instead of spending time on replacing all image cards with the new style. [5.]

- Improve collaboration. Component library helps developers in multiple teams work together as a team on developing the library components. Besides improving collaboration between developers, it will also improve the collaboration with UI and visual designers. If a UI designer designs a new component, the designer will check the library and collect all the necessary elements for creating a new component. The front-end team starts developing the components right after the UI designer designs the

component. This increases the collaboration between different teams working on the library. [5.]

A component library is a single file, folder or repository that consists of all the components and styles used in a software or an application. This can include image cards, themes, input fields, buttons, icons, dialog, text, colours, and different UI kits. Using a component library will improve consistency and let developers and designers work together. A collaborative process among developers, marketers, designers, product owners, and the other stakeholders is a perfect way to grow a sustainable component ecosystem.

3.2 Android Architecture

Android is a popular mobile operating system. There are over three billion active Android devices worldwide. Android is an open-source platform based on Linux. Android was originally developed by a start-up company called Android. In 2005 Google purchased the company and took over the operating system as well as the development team. Android is a powerful development framework that includes everything that is used to build a native Android application. The operating system also enables developers to deploy native applications to a wide variety of devices such as phones, watches, and tablets. [9.]

Android’s unified application development approach allows developers to develop applications for Android in general and run the application on different devices that are powered by Android operating system. Android has powerful APIs, a diverse and huge ecosystem of users, a growing developer community, outstanding documentation and has no development or distribution costs.

Android is open-source and freely available to manufacturers. The operating system does not have predefined software or hardware setup. However, the base operating provides many common features, such as messaging, media support, storage, connectivity, hardware support and multi-tasking. [10.]

Android architecture contains multiple components for supporting different android device needs. The Android operating system is built using a Linux kernel and multiple C++ and C libraries. [11.]

Figure 6 shows the software components of the Android operating system which is divided into four main layers and five sections.

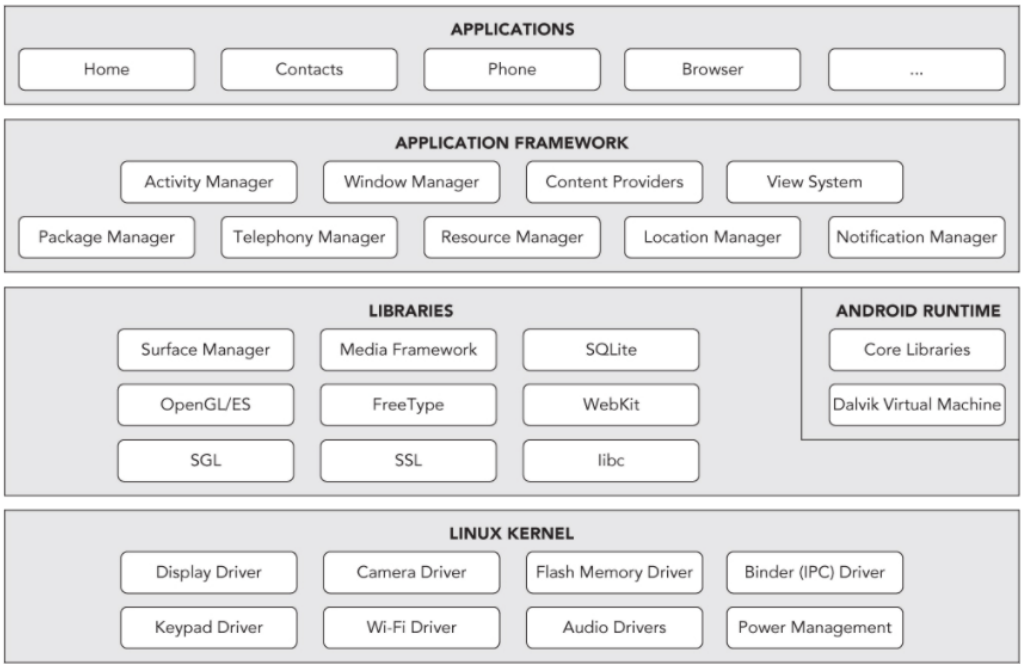

Figure 6. Major architectural components of Android framework. [10].

3.2.1 Linux Kernel

This is the foundation of the Android operating system that contains a set of essential hardware drivers such as display, keypad, and camera. This layer allows device manufacturers to manufacture hardware drivers for the kernel and create abstraction between the device hardware. Also, the Linux kernel manages all the things that Linux is good at, such as device drivers, security, power management and networking.

3.2.2 Libraries

This layer contains the files that provide the basic features of an Android operating system. It runs on the top of the Linux kernel and provides various C and C++ core libraries such as Graphics, Surface Manager, OpenGL, and Media. [10]. For example, the OpenGL library provides support for drawing. Therefore, an application can use it to manipulate graphics.

3.2.3 Android runtime

This layer is one of the most important parts of the Android operating system that provides a collection of libraries that enables application development using the Java programming language. This layer of Android operating system contains the Dalvik virtual machine, which is Java Virtual Machine designed for Android to enable Android applications to run in their own process. [10.]

3.2.4 Application framework

The layer provides a generic abstraction for hardware access and several important methods that expose various capabilities of the Android operating system. It also manages the application resources and the user interface. [11.]

It provides different services such as activity manager, notification manager, view system and package manager, which are helpful for the application development.

3.2.5 Applications

All native and third-party applications are built on this layer of the Android operating system. The application layer uses the services and classes that are available in the application framework and runs within the Android Run Time. Applications that are preinstalled in Android devices such as Browser, Phone,

Messenger, Photo and Camera, as well as third-party applications that users install from Android application market runs in this layer. [11.]

3.3 Android Modules for the Component Library

Android application development requires Android Studio, which is a Java based integrated development environment used to develop Android applications. Android studio is a powerful code editing tool that uses code templates, Gradle based build system, GitHub integration and emulator to support application development. [12.]

Every Android project has one or more modalities that contains the project source code and resource files. These modalities include Library modules, Android application modules and Google App Engine modules. Android Studio works on a module-based project structure. Modules are additional software components that provide a container for application source code, resource files, build files and Android manifest files. Android Studio can handle multiple modules in a single project. Having multiple modules in an Android Studio project gives the possibility to work on one project instead of multiple projects. Therefore, developers can create a more organised application. [13.]

The component library that is built in the thesis project uses Android Studio for the development. The library used two modules, one Library module for building the library and another Application module for testing and demoing the library components.

Creating a new Android project automatically creates an application module [12]. The name of the default application module is “app”. It is a container for the application resource files, source code and module-level settings. The component library uses this module for testing the components and building the catalogue application.

Android Studio provides the following application modules:

- Phone & Tablet Module
- Android TV Module
- Wear OS Module
- Glass Module

The library uses the Phone and Tablet application module for the catalogue application. The component library also used an Android Library Module for building the library components. The structure of the Android Library Module is the same as an Android application module. The module can contain everything needed to build an application, including resource files, manifest file, and source code. Android library compiles into an Android Archive (AAR) file that can be used as Android application module dependency. The Android Library module contains the shareable Android source code and resources that can be referenced by other modules in an Android project. [14.]

4 Guiding Principles and Development Approach

Companies need an effective way to manage the user experience and user interface of their digital products. It is essential that the company's digital product will scale effectively without requiring frequent additions or rework of design files and assets while maintaining a consistent UI system. UI component libraries are a collection of interface elements that can be reused across multiple products to give consistent experiences for the users. This section describes the development approach and the guiding principles used for the component library development.

4.1 Guiding Principles

Guiding principles are values that set a standard for the functionalities and behaviours of the components in the library. A well-structured and implemented

guiding principles ensure that the components provide the expected functionalities. The library components are built by following the design principles listed in this section.

4.1.1 Design Guidelines

Design guidelines are a collection of rules that creates a unified identity when connecting multiple components within a brand, such as colours, logo, and typography. The library components should follow Elisa’s design guidelines.

4.1.2 Reusability and Flexibility

The Library UI components should be flexible and reusable. When a component is reusable and flexible it can be used multiple times in a single project or across multiple projects and the usage can be changed to fit the given situation.

To achieve this in Jetpack Compose there are certain guidelines which should be followed.

- Components should be free from complex business logic
- Avoid using fixed constraints for components that portray meaning for the user such as Button. This will allow components to grow when to content size grows
- Consider using methods that allow components to grow accordingly over the methods that allow a fixed component size.
- When a fixed size modifier must be used for the height or the width of a component, the opposite axis modifier should use minimum constraint modifier.

4.1.3 Accessibility

Accessibility is about building things that everyone can use. Elderly people and people with disabilities can perceive, understand, navigate, and interact with the product. [15;16.]

The library components should support accessibility to allow different users to obtain information despite their individual needs. Semantics properties of Compose provide information about UI elements that are displayed to the user. Compose uses these semantics properties and the Android framework APIs to pass information to accessibility services which transform what is displayed on screen to a more suitable format for a user with a specific need. [17.]

There are certain rules and best practices that should be followed to have good accessibility coverage throughout the library and fulfil the client company Accessibility standards.

- Colour and Contrast: based on Elisa's accessibility standard the text should have a contrast ratio of 4.5:1 or greater with the background and a 3:1 contrast ration or greater for text larger than 18 pixel.

- Language: use localization string with string resources.

- Provide non-text components with content descriptions and keep the labels concise.

- Do not include the control type of the components in the content description.

- Do not include states such as On and Off in the content description.

- Provide hints for the EditText component.

4.1.4 Testability

Testing helps to determine whether the UI components meet the expected requirements and ensure that they are free of defects. Each component should be tested and easy to test to assure they are working as expected.

4.2 Development Approach

Jetpack Compose has four major architectural layers that combine to create a complete stack. The higher-level components are built by combining the functionality of the lover levels. Based on the architectural layers of Jetpack Compose, three development approaches are proposed to build the library. This section shows the advantages and disadvantages of each approach regarding the level of control, customization they can provide, and the selected development approach the library used.

Figure 5 shows the architectural layers of Jetpack Compose. Runtime is the lowest architectural layer. It provides Compose's programming model building blocks and state management. Material is the upper architectural layer of Jetpack Compose. It provides ready to use Material Design components.

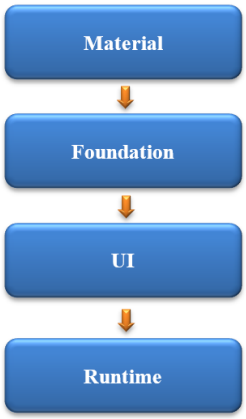

Figure 7. Jetpack Compose major architectural layers

4.2.1 Foundation Layer Approach

This approach is based on the Foundation layers of Jetpack Compose, which is the building block of Compose UI components like Row and Column. Building UI custom components using this layer gives better control and customization compared to the components built using the Material layer. It also minimises the project Gradle dependency since it does not require the dependency for the upper Material layer. However, building this layer required building components from the ground and it does not include all the best practices like accessibility features and theming by default. This approach requires much effort and longer development time to build a single UI component with all the best practices which are available by default in the Material layer.

4.2.2 Material Layer Approach

This approach creates custom components by wrapping the Material layer components of Jetpack Compose. The Material layer is the highest architectural layer of Jetpack Compose. It provides the Material Design System implementation for Compose UI including theming, accessibility features, and UI components.

Custom UI components built on this layer have less control and customization compared to foundation layer components. The custom components are built on top of the material components therefore they can only be customised based on the offered customization specified in the Material Design System. When a component requires customization which is not supported by the Material later component this approach will drop the Material layer and use the Foundation layer for that single component.

4.2.3 Forking

This approach uses the Material component implementation as a reference to build custom components. Components in this approach will have all the

benefits from the Material Design System best practices. They can also be customised as required. However, the components will not get any bug fixes or future additions from the Material component. This will create a compatibility issue during updating versions of dependent libraries which in turn increases the maintenance time and cost.

All the approaches mentioned have their own advantages and disadvantages. Selecting the right approach depends on which one can fulfil most of the requirements with a minimum risk. This library is built using the Material approach. The Material layer components by default are built by considering the best practices for mobile components by the Material Design System. It also supports theming. This approach will help to increase the development process since most of the features are included. The accessibility feature is one of the requirements which this library should provide which is again included by default in the Material components.

5 Development Workflow and Components

5.1 Development Process

Software development process is a step-by-step process to bring a product's concept from the ideation stage to implementation. The development process divides the software development work into smaller and sequential subprocesses. Regardless of the size and scope of a project, following a good development process will make a project development a success. The component library development process flow spans four key phases.

5.1.1 UI Design

The purpose of a UI design is to deliver effortless and seamless user experiences with a polished look. Having an attractive and modern user interface lets the users engage more with the product and benefits from all its features.

This phase requires a good collaboration with the designer team for creating interactive, intuitive, and user-friendly UI components.

5.1.2 Development

After planning and designing the components in the first phase, in this phase each component is developed according to the design. The library components are developed regarding the guiding principles. The components are flexible, easy to use, testable and accessible for all users.

The development process involves the use of a continuous integration and continuous delivery tool called GitHub Actions. GitHub Actions is a continuous integration and continuous delivery tool used to automate a project build, test, and deployment pipeline using a workflow and virtual machines. The component library workflows build and test the components for every pull request to your project repository and store the AAR and APK artefacts for production usage. [18.]

5.1.3 Testing

Testing is a practice to check whether the software product matches the requirements and to make sure that software product is free from defects. Software components are executed using automated or manual tools. Testing is crucial for any software product because it can identify early any bugs or errors in the software, and it can be solved before delivery of the software product.

The component library is built using Jetpack Compose. Jetpack Compose does not use Android View system. Therefore, testing the Jetpack Compose components requires a different approach. Jetpack Compose provides testing APIs to work with Composable by finding Composable, verifying their attributes, and performing user actions. The component library uses the Testing In Isolation pattern to test each component in the library. Using this pattern components are tested individually to make sure they are working as expected.

Listing 1 shows the usage of ComposeRule API to test the Expandable card component functionality. The test finds the component using its test tag and tests the call-back functionality.

```
class ExpandableCardTest {

    @get:Rule
    val composeTestRule = createComposeRule()

    @Test
    fun expandableCardShouldCallOnClickWhenTapped() {
        val expanded = mutableStateOf(false)
        composeTestRule.setContent {
            ElisaComposeTheme(true) {
                ExpandableCard(
                    modifier = Modifier.testTag(expandableCardTestTag),
                    expanded = expanded.value,
                    cardHeaderTitle = {
                        TitleMediumText(text = expandableCardHeaderText,
modifier = Modifier.testTag(
                            expandableCardHeaderComposableTestTag))
                    },
                    cardBody = {
                        BodyMediumText(text = expandableCardBodyText, modifier
= Modifier.testTag(
                            expandableCardBodyComposableTestTag))
                    }, onClick = {
                        expanded.value = !expanded.value
                    })
            }
        }

        val exCard = composeTestRule.onNode(hasTestTag(expandableCardTestTag),
useUnmergedTree = true)
        exCard.performClick()
        assert(expanded.value)

       }
    }
```

Listing 1. Component testing using ComposeTestRule

The component library uses GitHub Actions to automate the testing. Making a pull request to the library repository triggers the testing. The workflow build, test and share the testing results.

5.1.4 Documentation

Software documentation is a part of good software development. Software documentation helps everyone to understand the product, capability, interface, task, and quickly search and find any component within the document. The library components are documented using the library repository ReadMe file.

The component library documentation explains in detail about the library technology stack, integration, component implementation, parameters, variants and usage.

5.2 Integration

Hosting makes a component library available on the World Wide Web. Therefore, it will be accessible to the public or a certain group of users based on the security configuration of the library owner. The component library built in the thesis project is an android project that runs on a Java virtual machine. Android projects can be hosted on multiple package repository sites such as Jitpack, Maven Central and GitHub Packages. These sites build the projects and provide ready to use artifacts such as APK and AAR.

The client company does not have a Maven Central account to host the library on Maven Central. The other package repository sites are not preferred by the client company; therefore, the library is not hosted in a package repository site. However, the library has a remote GitHub repository which is used to build the project and store the build artifacts. Therefore, the library integration involves the usage of this GitHub repository. There are three methods to integrate this library with a local project.

5.2.1 Download AAR

The library source code is available as a remote GitHub repository. There is a GitHub Action workflow in the library remote GitHub repository that builds and stores the build artefacts such as AAR and APK. Therefore, integrating the library with this method requires downloading the AAR and adding it into the lib folder of the android application module.

5.2.2 Submodule

The library source code is available as a remote GitHub repository. There is a GitHub Action workflow in the library remote GitHub repository that builds and stores the build artefacts such as AAR and APK. Therefore, integrating the library with this method requires downloading the AAR and adding it into the lib folder of the android application module.

5.2.3 Source Dependency

The library source code is available as a remote GitHub repository. There is a GitHub Action workflow in the library remote GitHub repository that builds and stores the build artefacts such as AAR and APK. Therefore, integrating the library with this method requires downloading the AAR and adding it into the lib folder of the android application module.

5.3 Components

The component library contains selected UI components used to build Android applications. It provides components such as theme, colours, typography, text, top app bar, button, alert dialog, snackbar, image card, divider, expandable card, vertical list card, canvas draw path and shimmering animation.

State in Jetpack Compose is any value that can change over time. Jetpack Compose uses the remember Composable to store the state in memory. The library Component are stateless. It means the components do not hold any state. This makes the component flexible and reusable.

5.3.1 Colour

Colour is a critical part in a design system. The library includes the full range of colours defined in Elisa’s design system. The UI components use these colours to maintain the brand personality throughout your UI.

The colours are separated into four categories based on their role, these are Core, Accent, Neutral and Status colours.

- Core colours. Core colours are the colours that are deeply related to the brand. Figure 6 shows the Brand Blue and White colours which are the core colours of the client company.

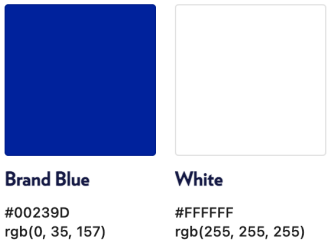

Figure 8. Core colours

- Accent colours. These are a set of complementary colours that support the client company's core colours. Figure 7 is a picture taken from the library catalogue application. It shows a list of accent colours available in the library.

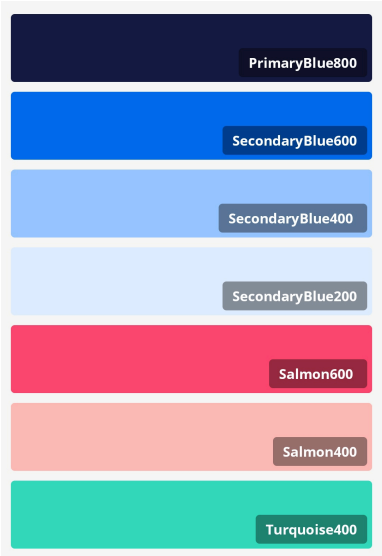

Figure 9. Accent colours

- Neutral colours. Alongside the core and accent colours, neutrals can be used as a light background colours and shades. Figure 8 is a picture from the library catalogue application. It shows a list of neutral colours for dark and light themes.

Figure 10. Neutral colours

- Status colours. Status colours are used in notifications to indicate status.

5.3.2 Typography

The library provides typography with different text styles. The typography used two different typefaces to build different text styles.

- Verlag. It is the primary font and is at the core of the client company brand. It is used in heading styles.

- Open Sans. It is the main font for body texts.

5.3.3 Top App Bar

Top app bar provides content and actions related to the current screen such as navigation, screen headlines, and actions. Top app bars use a colour fill instead of drop shadow to create separation from content. The top app bar component default width is the same as the width of the parent component or the device.

The library contains two types of top app bar components, these are TopAppBar and TopAppBarMedium components. Both components are highly flexible, and they can be customised in many ways. Figure 10. shows the regular and medium top app bar components with their default look and with action and navigation icons added.

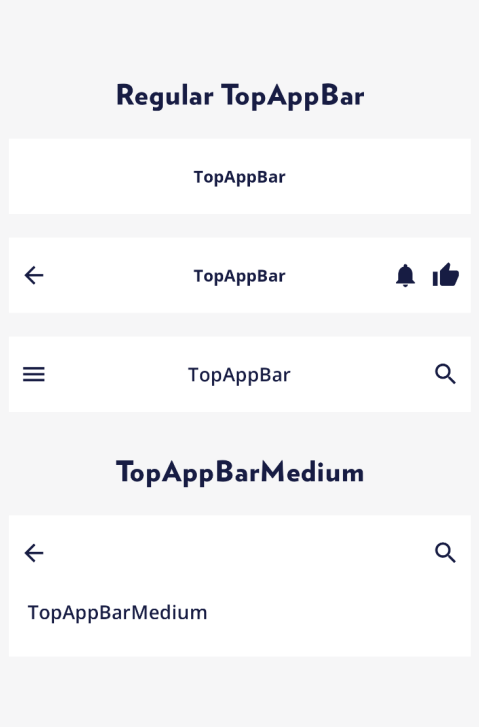

Figure 11. Top app bar components

Material design provides a Scaffold composable that implements the basic Material Design layout structure. The Scaffold composable has default slots for

the common components used in a screen. TopBar is one of the slots provided by Scaffold that accept top app bar components.

The library top app bar components can be implemented on its own or it can be implemented with the Scaffold composable. Listing 1 shows TopAppBar implementation with a Scaffold.

```
Scaffold(
    modifier = Modifier,
    topBar = {
        TopAppBar(
            navigationIcon = {
                IconButton(onClick = { /* call-back */ }) {
                    Icon(
                        painter = PainterResource(id = R.drawable.arrow_back),
                        contentDescription = " content description here"
                    )
                }
            },
            title = { LableLargeText(" Regular TopAppBar") },
            actions = {
                IconButton(onClick = { /* call-back */ }) {
                    Icon(
                        painter = PainterResource(id = R.drawable.Menu),
                        contentDescription = " content description here ")
                }
            },

        )
    },
    content = { innerPadding ->
        LazyColumn(
            verticalArrangement = Arrangement.Center
            contentPadding = innerPadding,
        ) {
            // lazy column items
        }
    }
)
```

Listing 2. Regular TopAppBar component implementation

5.3.4 Button

Buttons communicate the user action with the system. Buttons are usually placed throughout the UI, in places such as cards, dialogs, modal windows, forms, and toolbars.

The library provides four types of buttons. The buttons are filled button, filled tonal button, outlined button and icon button. The buttons have a rectangular shape.

- Filled button. The button has a higher visual impact. Therefore, it is used for important actions that complete a flow. The button has a solid BrandBlue background colour for the light theme and a SecondaryBlue600 background colour for the dark theme. Figure 11 shows the Filled button component in light theme.

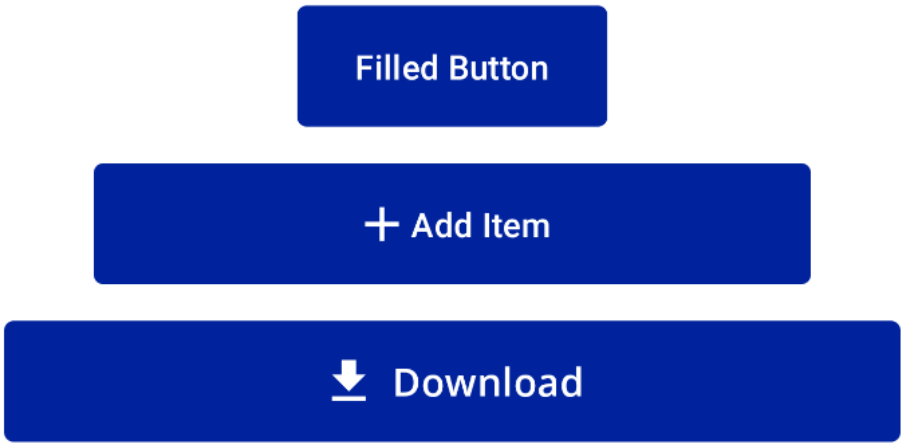

Figure 12. Filled button component

- Filled tonal button. Filled tonal button is a middle ground button between the filled button and outlined button. The button is used in contexts where the actions require higher priority than the outlined button. Figure 12 shows the Filled tonal button component in light theme.

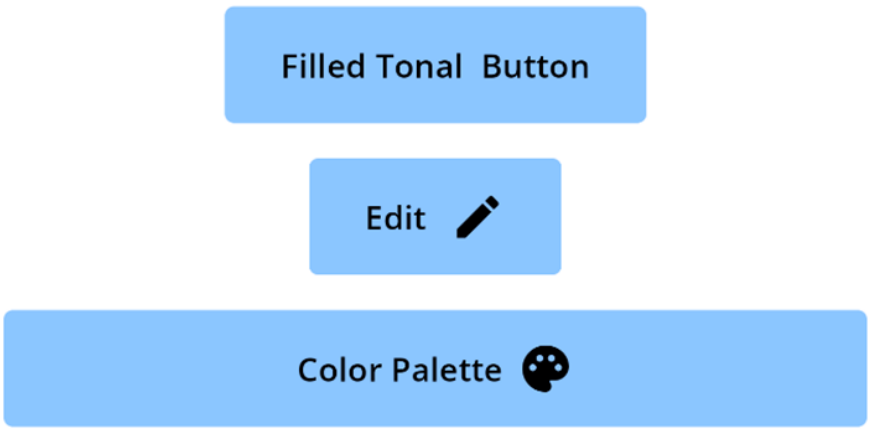

Figure 13. Filled tonal component

- Outlined button. Outlined button is a medium-emphasis button. Outlined button displays a stroke around the button container, and by default has no background colour. Figure 13 shows the Outlined button component.

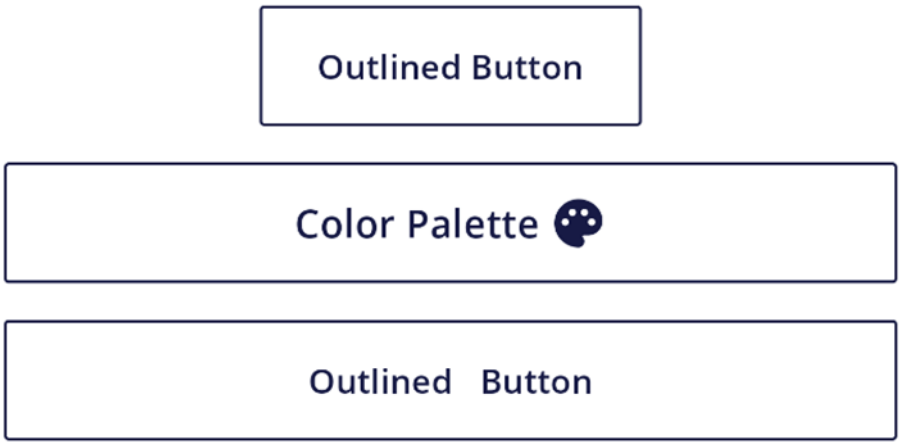

Figure 14. Outlined button component

- Icon button. Icon button is a clickable icon that represents actions. The button content is centred inside the icon button. The content should typically be an Icon. Figure 14 shows the Icon button. As shown in the picture the icon button does not have a background colour.

Figure 15. Icon button component

Using the button components requires the component library integration with the implementing project. Listing 2 shows the implementation of all buttons available in the library.

```
// filled button

    FilledButton(onClick = { /* called when clicked */ }) {
        Text ("Filled Button")
    }


// filled tonal button

    FilledTonalButton(onClick = { /* called when clicked */ }) {
        Text ("Filled Button")
    }


// Outlined button

    OutlinedButton(onClick = { /* called when clicked */ }) {
        Text ("Otlined Button")
    }


// Icon button

    IconButton(onClick = { /* called when clicked */ }) {
        Icon (imageVector = Icons.Filled.Add, contentDescription = "add item
")
    }

}
```

Listing 3. Button components implementations

5.3.5 Image Cards

Image card is a component used to display images. It has a compulsory image and content description parameters. Other parameters are optional. The library contains four variants of image card components.

- Regular image card. The regular image card displays a single image, caption, and icon in a vertical arrangement. It has a click listener for detecting a click event when the card is clicked. Figure 15 shows a regular image card with a placeholder image, caption section and icon section.

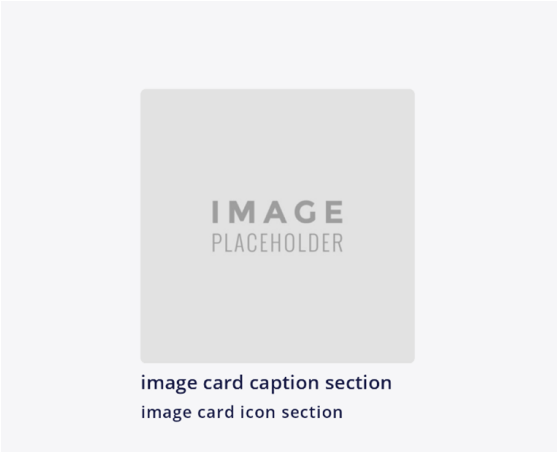

Figure 16. Regular image card component

- Animating Image card. This image card displays a single image, caption, and icon in a vertical arrangement. It has a similar layout with the regular image card. The image card has a click, long click and double click listener. The image card animates when clicked.

- Gradient background image card. This image card displays a single image, caption, -and icon in a vertical arrangement. The image card has a gradient background that fills the width of the device. Figure 16 shows the gradient image card with a placeholder image, caption section and icon section.

Figure 17. Gradient background image card component

- Text overlay Image card. This image card uses the image as the card background and displays different contents of the card over the image. Figure 17 shows the text overlay image card component layout and sections.

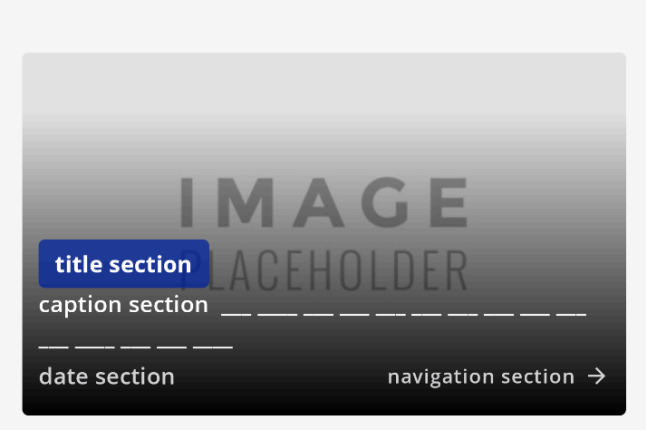

Figure 18. Text overlay image card

Using the image card components requires the component library integration with the implementing project. The image cards have default values for all parameters except the image and content description parameters. Therefore, implementation is fast and easy. Listing 2 shows the implementation of the regular image card with and without icon and caption sections.

```
// default regular image card implementation

ImageCard(
    cardWidth = 200.dp,
    painter = image,
    imageContentDescription = "content description ",


// regular image card implementation with icon and caption



ImageCard(
    cardWidth = 200.dp,
    painter = image,
    imageContentDescription = "content description ",
```

```
text = {
    LabelLargeText(text = "image card caption section ")
},
icons = {
    LabelMediumText(text = "image card icon section ")
})
```

Listing 4 . Regular image card component implementation

5.3.6 Expandable Card

The Expandable Card shows the important data on the top and if the user is interested in the content the user can click to expand the card and can see the related data in detail. Figure 18 shows the expandable card in expanded state.

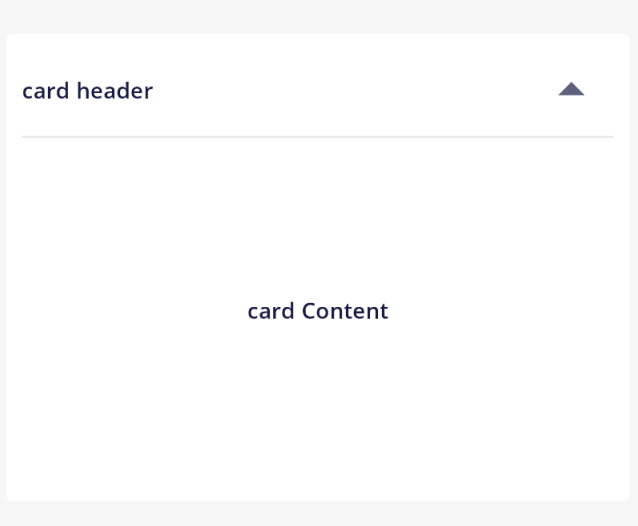

Figure 19. Expandable card component

Using the expandable cards requires the library integration with the implementing project. Listing 4 shows implementation of the expandable card with the state variable for expanding and collapsing the card.

```
val  isExpanded = remember {
     mutableStateOf(false)
 }

ExpandableCard(expanded = isExpanded,
     cardHeaderTitle = {
         LabelLargeText(text = "card header")
     },
     cardBody = {
                      LabelLargeText(text = "card Content")


     }
}
```

Listing 5. Expandable card component implementation.

6 Future Developments and Applications Using the Component Library

The component library has multiple integration methods for android applications implementing the library. Currently the component library is used by a Catalogue application, Demo application and Elisa Kirja application. Elisa Kirja a production application from Elisa Oy.

6.1 Catalogue Application

The main purpose of the catalogue application is to demonstrate the list of available components in the test the library. In addition to this it also serves to test the library components. All the UI components of the catalogue application such as the theme, top app bar and the texts are from the component library. Figure 19 shows different screens of the catalogue application in light theme.

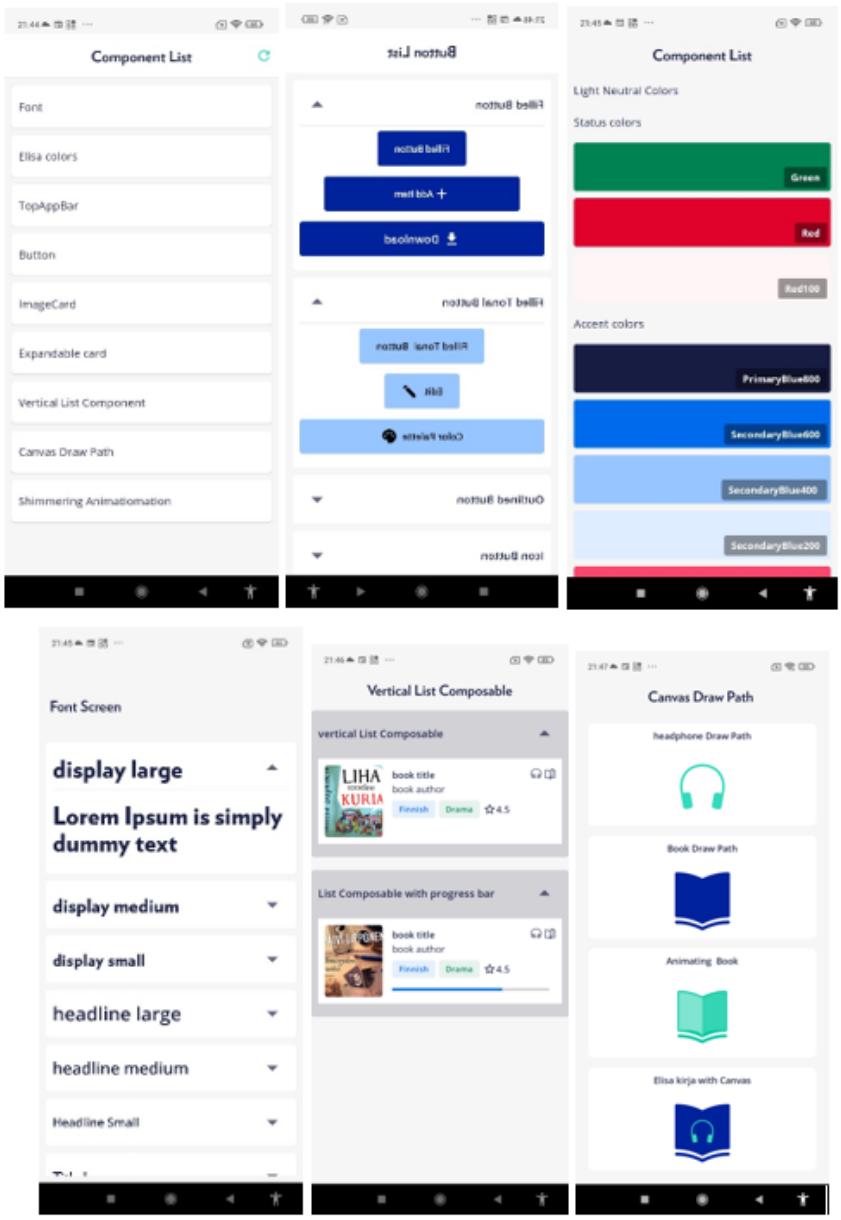

Figure 20. Screenshots of the catalogue application

6.2 Demo Application

The demo application is a separate project from the library project. The purpose of this application is to show the library integration with the real project. It also shows the usage of all components in the library. The UI of the demo application is fully built using the library components. It uses the theme component from the component library to support dark and light themes. Figure

20 shows multiple screens of the Demo application in both dark and light themes.

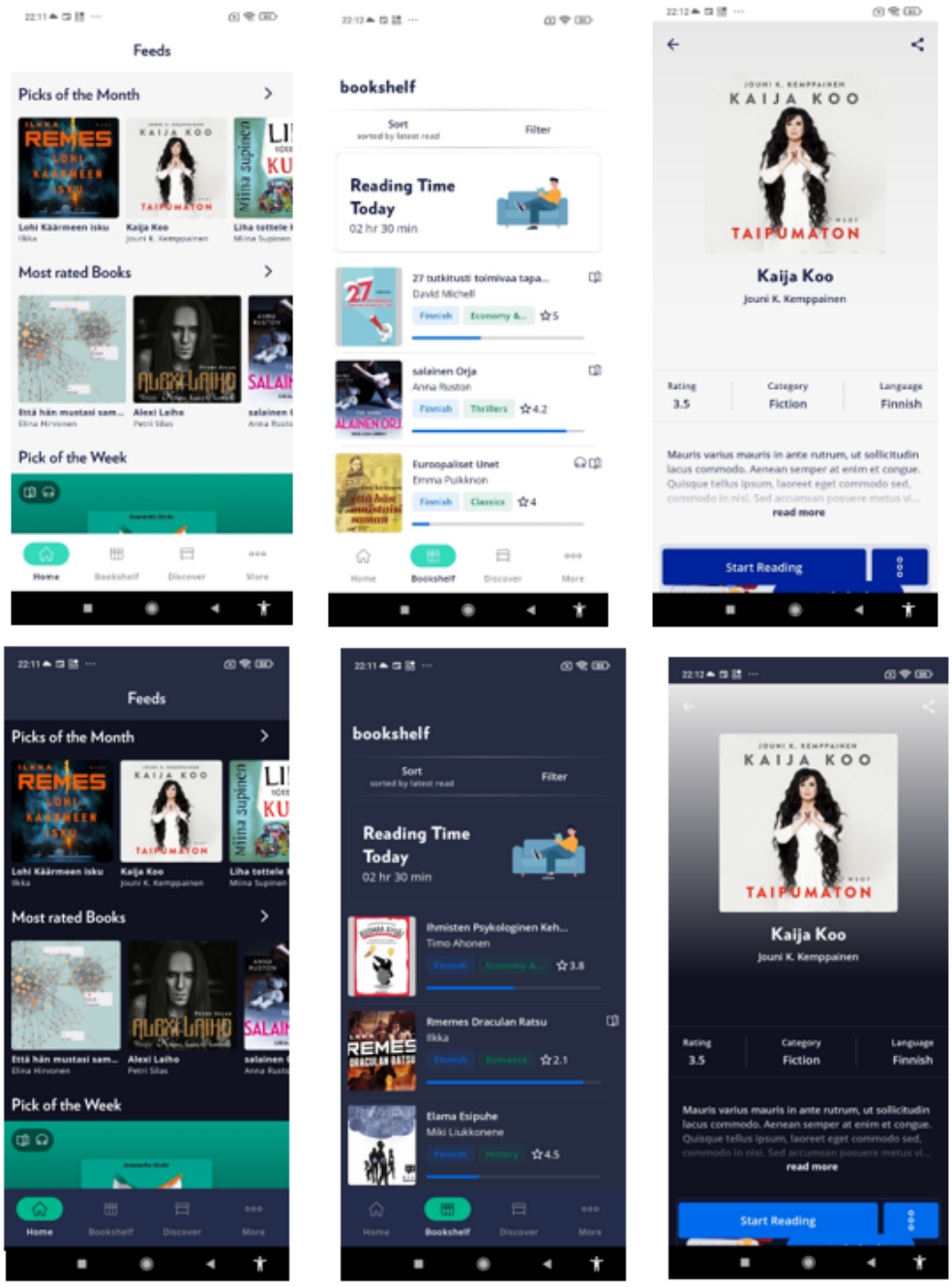

Figure 21. Screenshots of the demo application

The demo application makes a network request, and it simulates the usage of the component library in production application. The Accessibility is also tasted manually and with an accessibility scanner.

6.3 Component Library Future Developments

The component library used the Material architectural layer of the Jetpack Compose to develop the components. The Material layer uses the Material You library from the Material Design System. By default, Material You library provides styled components, theming system, icons, dynamic colours, and ripple Indications. The component library used this component to create custom components that follow the client company design guidelines.

The Material You library of Jetpack Compose is in its early stage of development. Therefore, it does not provide some UI components such as Text Input and progress bar. Also, the existing components are not stable, and they are subject to changes in the future.

The library provides enough UI components to start working with a new android project. The client company is happy with the status of the library and the first phase of the library development has ended. However, the library development will continue because it lacks some important UI components, and it requires frequently updating the Jetpack Compose and Material You dependencies.

7 Conclusion

The purpose of this project was to build a component library using Jetpack Compose. Jetpack Compose is a modern Android UI toolkit used to build a better and efficient UI. Jetpack Compose and its architectural layers are briefly explained in this thesis. A component library can be built using different architectural layers of Jetpack Compose. There is no perfect way, and each approach has its own advantages and disadvantages. Selecting the right approach depends on the complexity of the library components, the company

guidelines, development time and cost. If a company has limited development time, and if the components do not require a lot of complexity, it is beneficial to use the Material layer to build the library. The Material layer contains different components that are made considering the best practices of the Material Design System to build a user interface. Material components also support accessibility features. If the company has enough development time and a large enough budget, it is better to use the foundation layer. This layer gives every possibility to customise components.

The component library built in the final year project provides a theming system, colours, alert dialog, snackbar, buttons, top app bars, typography, texts, vertical list component, image cards and animations. At this stage the library provides only selected components. Therefore, it needs further development to include additional components.

The library is integrated and used in multiple applications such as the demo application, catalogue application and Elisa Kirja application. The library helped to develop the applications in a short time with an elegant UI. The component library is developed for Elisa; therefore, developers at Elisa can integrate and use it. This library makes Elisa’s Android applications have a uniform look that represent the company’s brand. This in return gives good customer satisfaction. It also minimises the development cost and time in the long run.

References

1 Buketa, Denis & Balint, Tino. 2021. From Jetpack Compose by Tutorials. Electronic book. Razeware LLC. https://www.raywenderlich.com/books/jetpack-compose-by-tutorials. Accessed 22 October 2021.

2 Horstmann, Cay. 2020. Core Java Volume I Fundamentals. Boston: Addison-Wesley. Electronic book. O'Reilly Safari Online. Accessed 16 October 2021.

3 Horton, John. 2021. Android Programming for Beginners - Third Edition. Packt Publishing. Electronic book. O'Reilly Safari Online. Accessed 20 October 2021.

4 Joshi, Bipin. 2017. Beginning XML with C# 7: XML Processing and Data Access for C# Developers. Apress. Electronic book. O'Reilly Safari Online. Accessed 16 October 2021.

5 Brake, Ronald. 2017. Advantages of a Component Library. Online. open social. https://www.getopensocial.com/blog/open-source/advantages-component-library. Accessed 28 February 2022.

6 Abrams, Jeremy. 2021. Everything you need to know about atomic design. Online. Bootcamp. https://bootcamp.uxdesign.cc/everything-you-need-to-know-about-atomic-design-c7310d8a0bbe. Accessed 28 February 2022.

7 Frost, Brad. 2016. Atomic Design. Electronic book. Brad Frost. https://atomicdesign.bradfrost.com/table-of-contents/. Accessed 28 February 2022.

8 Langvad, Rasmus. 2020. What is a Component Library. Online. https://langvad.dev/blog/what-is-a-component-library/. Accessed 28 February 2022.

9 Griffiths, Dawn & Griffiths, David. 2021. Head First Android Development, 3rd Edition. Electronic book. O'Reilly Safari Online. Accessed 25 February 2022.

10 Dimarzio, Jerome. 2016. Beginning Android Programming with Android Studio, 4th Edition. Wrox. Electronic book. O'Reilly Safari Online. Accessed 10 February 2022.

11 Meier, Reto & Lake, Ian. 2018. Professional Android. Wrox. Electronic book. O'Reilly Safari Online. Accessed 14 January .2022.

12 Smyth, Neil. 2021. Android Studio 4.2 Development Essentials - Java Edition. Packt Publishing. Electronic book. O'Reilly Safari Online. Accessed 10 January 2022.

13 Hagos, Ted. 2020. Learn Android Studio 4: Efficient Java-Based Android Apps Development. Apress. Electronic book. O'Reilly Safari Online. Accessed 15 October 2021.

14 Raywenderlich Tutorial Team; Costeira, Richardo; Sen, Subhrajyoti & Sturt, Kolin. 2021. Real World Android by Tutorials. Razeware LLC. https://www.raywenderlich.com/books/real-world-android-by-tutorials. Accessed 22 October 2021

15 Padolsey, James. 2020. Clean Code in Javascript. Packt Publishing. Electronic book. O'Reilly Safari Online. Accessed 16 October 2021.

16 Manning, Susan & Johnson Kevin E. 2020. Online Learning for Dummies. For Dummies. Electronic book. O'Reilly Safari Online. Accessed 17 October 2021.

17 Phillips, Bill; Stewart, Chris & Marsicano, Kristina. 2017.Android Programming: The Big Nerd Ranch Guide, Third Edition. Big Nerd Ranch Guides. Electronic book. O'Reilly Safari Online. Accessed 10 October 2021.

18 Geisshirt, Kenneth; Emanuele, Zattin; Olsson, Aske & Voss, Rasmus. 2018. Git Version Control Cookbook. Packt Publishing. Electronic book. O'Reilly Safari Online. Accessed 18 December 2021.