Visa Soininen

Jetpack Compose vs React Native – Differences in UI development

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technology

Bachelor's Thesis

1 November 2021

Abstract

Author:	Visa Soininen
Title:	Jetpack Compose vs React Native – Differences in UI Development
Number of Pages:	32 pages
Date:	1 November 2021
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communications Technology
Professional Major:	Mobile Solutions
Supervisors:	Petri Vesikivi, Principal Lecturer Peter Hjort, Senior Lecturer

---

This thesis was written from a topic that was thought out together with Metropolia UAS. This study includes a comparison between Jetpack Compose and React Native. These frameworks were compared both from a developer's standpoint and from a user's point of view by performance testing.

During this study there were two applications developed. Both applications utilize the same API. The applications were both given a simple set of specifications and the development work needed was compared between the two. The study includes chapters explaining how the frameworks function and why they were developed and what needs do they fulfil. Performance was measured with profiling tools provided by each framework and with Perfetto. All testing was done on a OnePlus 7 -device running Android OS.

Results imply that from a developer's standpoint React Native offers a faster way of developing the application. It is also capable of running on iOS-devices. In terms of performance Compose was quicker in most of the tests. This implies that React Native works better for creating prototypes and small applications, but for larger applications with large amounts of functionality Compose is better suited for.

This thesis can be used to determine which framework should be used when a new application is entering its development stage. This study offers results and opinions on different aspects of developing that can help developers decide which framework will be the best fit for them.

Keywords: UI development, Jetpack Compose, React Native

Tiivistelmä

Tekijä:	Visa Soininen
Otsikko:	Ohjelmistokehityksen eroavaisuudet Jetpack Composen ja React Nativen välillä
Sivumäärä:	32 sivua
Aika:	1.11.2021
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Mobile Solutions
Ohjaajat:	Yliopettaja Petri Vesikivi Lehtori Peter Hjort

Insinöörityön tarkoituksena oli vertailla suorituskykymittausten avulla Jetpack Compose- ja React Native -käyttöliittymäkehyksiä sekä sovelluskehittäjän että sovelluksen käyttäjän näkökulmasta.

Työssä kehitettiin kaksi sovellusta, jotka molemmat hyödyntävät samaa rajapintaa. Sovelluksille annettiin samat yksinkertaiset vaatimukset, minkä jälkeen työkaluja vertailtiin niin suorituskyvyn kuin ohjelmoinnin helppouden osalta. Suorituskykyvertailua tehtiin kummankin työkalun tarjoamilla ohjelmistoilla sekä Perfetto-suorituskykyprofiloijan avulla. Kaikki testit tehtiin Android-käyttöjärjestelmällisellä OnePlus 7 -puhelimella.

Tuloksista voidaan päätellä, että ohjelmoijan näkökulmasta React Native on soveltuvampi työkalu, jos sovellus täytyy kehittää lyhyessä ajassa. React Nativella kehitetty sovellus kykenee toimimaan myös iOS-laitteilla. Suorituskyvyn kannalta Composen avulla kehitetty sovellus oli suurimmassa osassa testeistä nopeampi kuin React Nativella kehitetty sovellus. Saatujen tulosten mukaan React Native toimii hyvin prototyyppien ja pienten sovellusten tekemiseen, mutta Compose on parempi työkalu, kun tehdään suuria sovelluksia, joissa on paljon toiminnallisuuksia.

Tätä työtä voidaan hyödyntää, kun on tarve luoda uusi sovellus, mutta sovelluksen kehittämiseen käytettävästä työkalusta ei ole tehty päätöstä. Työ tarjoaa kehittämistyön eri osa-alueisiin näkemyksiä, jotka saattavat vaikuttaa päätöksentekoon.

Avainsanat: käyttöliittymäkehitys, Jetpack Compose, React Native

Contents

List of Abbreviations

UI:	User interface. Mean to provide the user a way to interact with the software.
API:	Application Programming Interface. An API allows the application to communicate with another service.
RN:	React Native. A cross-platform application development framework.
JSC:	JavaScriptCore. Engine used to interpret JavaScript code.

1 Introduction

There is a large variety of frameworks that are used to develop native mobile applications. The target of the thesis was to compare one of the newest frameworks, Jetpack Compose with one of the most widely used frameworks, React Native. The task was to determine factors that should be considered when starting development on a new application and which of the frameworks would be a better fit for the developer's needs.

This study was performed to find and analyze similarities and differences between these two frameworks from a developer's standpoint. This can be useful for developers who are starting a new project but are not sure whether they should use actual native tools or in this case React Native.

This thesis focuses on comparing the performance of these technologies. The applications performance was evaluated using Perfetto to obtain low-level data from the device and test responsiveness and rendering performance with simple use-cases. Performance is one of the most important deciders for users if they are committed to continue using an application. Even seemingly small delays can deter the user away from using it.

Two applications were developed during this study. Both applications were given the same specifications and they use the same API to obtain similar data to compare performance and development architecture equally. The application development phase was documented, and these applications were compared with each other to find each technology's strengths and weaknesses.

2 Mobile UI development

Over the last couple of years mobile phones are becoming more of a lifestyle than only being a way to communicate with contacts. In June 2019 50.71% of Googles search queries were made from mobile devices. From the whole world's population more than two million people only use their mobile device to access

the internet. Currently this equates to 51% of mobile phone users across the world. [1]. Reported by the World Advertising Research Center (WARC) this percentage will rise to 72.6% by 2025 [2].

This fast-moving trend has prompted many companies to provide more efficient tools for programmers to develop applications and “mobile-first”-design websites.

2.1 Native application development

Native application development expresses that the application is being developed specifically to a specific platform such as Android or iOS. These applications are built with Java/Kotlin for Android and Swift/Objective-C for iOS. Applications built using Native frameworks can access each API provided by the device. [3]. The ability to access each API gives the developer access to low-level hardware information and the ability to read values from sensors such as gyroscopes and accelerometers.

Developing native applications tend to provide the user with a greater experience since they are equipped with better performance and a larger set of tools to utilize. Native development frameworks provide the developer ways to theme their applications to match the system user experience better to avoid large contrast between the operating system and the developed application. [3]. This creates a more familiar and therefore a more pleasant user experience.

Downsides of native development include having to manage two codebases and therefore employing at least two teams working on the project. This also means that there will be more time spent to reach a viable product since both applications require to be developed separately to support Android and iOS. [3.]

2.2 Battle for the best performance

Performance is amongst the most important aspects when developing a user-friendly application. A longer than three second loading time gets an application

removed from the users' phone 40% of the time. [4]. Tools such as Android Studio and React Developer Tools provide performance monitoring to help developers minimize loading times and further optimize their applications. Performance can be divided into networking efficiency, responsiveness, and overall speed of the application.

2.2.1 Networking efficiency

Networking performance is limited by the size of the data, location of the client and the available bandwidth. Applications can benefit substantially if data is stored on the clients' device instead of fetching the data each time it needs to be shown. In case the application does not require an internet connection it can be used offline provided that mandatory data is already stored on the device. [4.]

2.2.2 Touch responsiveness

Using a touchscreen requires the device to react to the touch as soon as possible to create a sense of interaction between the user and the device. If the device or application does not immediately respond to the users input, it makes the user experience feel clumsy. Response speed is restricted by the devices processing power, technology used to develop the application and sub-optimal coding practices.

If the input given requires time to render the desired output, it should meanwhile show a temporary output such as a loading bar to acknowledge the users input is being processed.

Natively developed applications usually are more responsive as they do not have to interpret cross-platform code such as JavaScript to function. React Native renders views natively but inputs are handled by communicating with the JavaScriptCore introduced later in this thesis. This can lead to the application taking longer to respond.

2.2.3 Animation fluidity

Animations are a large part of a successful application. They provide the user with visual guidance and enhance the liveliness of the application. Downside of using complex animations is that they require more computing power than a static view. Rendering these animations can result in diminished battery life and dropped frames.

Dropped frames occur when the device cannot keep up with the computations. This means that the device will skip rendering the frames which in result will lead to choppy animations. Most modern phones are equipped with a 60Hz monitor. This means that the device will update the view 60 times per second. As a result, the animations should try to reach at least 60 frames per second to avoid a bad user-experience. To achieve 60 frames per second, the application has approximately 16 milliseconds to compute and render each frame.

In 2017 ROG published the first mobile phone with a 90Hz display [5]. This led to many other phone manufacturers publishing their own high-refresh-rate devices with many models supporting displays as fast as 120Hz. These faster displays demand even faster computation to render animations without dropping frames.

3 React Native – Bringing React to the mobile world

React Native (RN) is a cross-platform developing tool developed by Facebook that acquired one of the largest userbases after its initial release in 2015. The strength of RN is being able to create Android and iOS applications writing almost exclusively JavaScript and maintaining only one codebase.

3.1 Origins

React Native was first introduced to the public in March 2015. Prior to its public release it was used in Facebooks internal development as they were facing issues developing with the tools provided by Google and Apple for native

application development. In 2012 when Facebook began its transition to a mobile first company, they bumped into major problems trying to render their current application in web views from Android and iOS developing toolkits. The largest hinderances were the lack of a keyboard API, gesture and touch event recognition and checking if an image has finished loading. [6.]

3.2 Architecture

Before React Native was published cross-platform development required the use of web views to render the application. RN allows developers to write native code with Kotlin/Java for Android and Swift/Objective C for iOS. Writing native code is rarely mandatory to achieve the same results as with using JavaScript. Cases where the developer must write native code are usually related to hardware functionalities such as requesting values from the device's gyroscope.

Compiling the JavaScript to iOS is done by using the JavaScriptCore (JSC) that is the engine behind Safari [7]. JSC enables the device to interpret JavaScript programs inside applications developed with Swift, Objective-C or C-language [8].

For Android the JSC is included in the application bundle during compiling. As seen in Figure 1 both platforms use RN Bridge to communicate between the virtual machine JSC and the Native modules rendered on the UI.

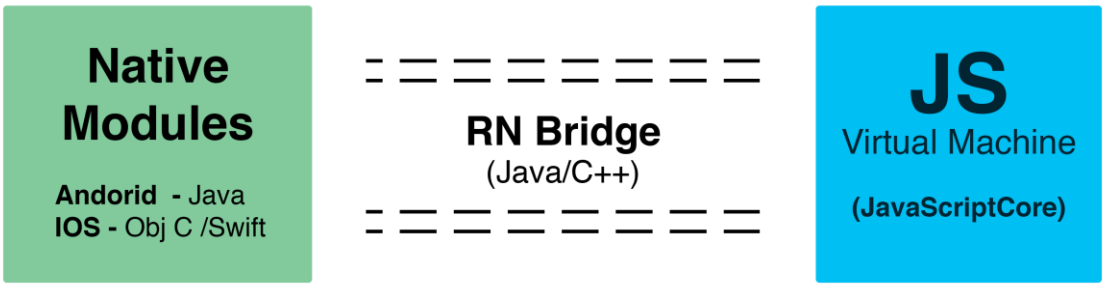

Figure 1. Communication between native modules and the JavaScript code [8].

The RN Bridge in Figure 1 is responsible for communicating with the device with tasks such as which views should be rendered and handle requests for hardware information from the device. JSC sends the required information to the device and after the native thread handled the request it will send back a confirmation that the tasks were executed. [7.]

Upon launching a React Native application there are two threads initialized: the Main thread and the JavaScript thread. It is possible to create more threads to move heavy workloads from the forementioned threads. Events such as UI component taps are handled by the Main thread, and it is responsible for sending the information to the JSC using the RN Bridge. JavaScript thread is where the code written in React Native executes. [7.]

3.3 Syntax

React Native projects are developed with either JavaScript or TypeScript. RN supports the use of EcmaScript6 to allow use of anonymous arrow functions and other modern conventions used in web development.

3.3.1 Class declaration

React Native encourages the use of re-usable views often referred to as components. Listing 1 shows how a page-component was written for the project that was developed during this work. The component is created using the functional component approach utilizing the arrow function from JavaScript ES6.

```
const FeedScreen = ({ navigation }) => {
  const [listofSubreddits, setListofSubreddits] = useState({})
  const [isLoaded, setIsLoaded] = useState(false)

  useEffect(() => {
    Requests.getSubreddits().then((items) => {
      const i: any = items
      setListofSubreddits(i)
      setIsLoaded(true)
    })
  }, [])

  return (
    <SafeAreaView style={styles.container}>
      <Text style={styles.header}>My Subreddits</Text>
      {isLoaded ? <SubredditList list={listofSubreddits} /> : <Spinner />}
    </SafeAreaView>
  )
}
```

Listing 1. Creating a functional RN component that renders a list of objects requested by a HTTP-request.

As seen in Listing 1 useState is used to remember a state in between rendering the component. UseState is a hook provided by React that allows the component to remember variables in between re-renders of the component. Updating a state triggers an automatic re-rendering of the component and therefore normal class variables are reinitialized with their default values. States can be accessed to determine how the component should be rendered after their value was updated. [9.]

The return value of a functional component is one or more JSX tags. JSX is a mark-up language that follows similar syntax as a HTML-file but instead of HTML tags such as <p> for a paragraph JSX uses <Text> provided by React. React provides several default components for the basic use-cases such as paragraphs, switches, and buttons but for project specific and more complex features the components must be developed manually. The return value inside Listing 1 shows how to render a custom component called SubredditList inside another component. The SubredditList component is shown in Listing 2.

```
const SubredditList = ({ list }) => {
  useEffect(() => {
  }, [list])

  return (
    <View>
      <FlatList
        data={list.data.children}
        renderItem={({ item }) => <SubredditListItem item={item} />}
        keyExtractor={item => item.data.created.toString()}
      ></FlatList>
    </View>
  )
}
```

Listing 2. A functional component that renders a list of objects using the list provided to it as a parameter.

There are multiple classes that can render dynamic length lists. For this thesis a FlatList was used. FlatList contains two mandatory properties, data and renderItem. Data is the list of objects and renderItem determines how a single item of the list should be rendered. [10]. In Listing 3 is the functional component called SubredditListItem that is passed to the renderItem-property in Listing 2.

```
const SubredditListItem = ({ item }) => {
  return (
    <TouchableOpacity style={styles.container}>
      <Image
        source={{ uri: item.data.icon_img ? item.data.icon_img : null }}
        style={styles.image}
      ></Image>
      <Text style={styles.text}>{item.data.display_name_prefixed}</Text>
    </TouchableOpacity>
  )
}
```

Listing 3. Component determining the look of a single item rendered in a list. This component is used in SubredditList shown in Listing 2.

Parameters are passed to components as props which is a special keyword. The props-parameter can be exploded using the curly brackets to directly access properties to avoid having to use redundant “props”-prefix. This is demonstrated in Listing 3.

3.3.2 Styling components

Components are styled using a StyleSheet-object. For web-developers changing over to React Native is very simple as the properties are mostly the same as

CSS-properties but instead of using hyphens they use camel case. Listing 4 demonstrates the similarity between CSS and RN styling.

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: "row",
    backgroundColor: Colors.grey,
    paddingVertical: 8,
    margin: 2,
    borderWidth: 1,
    borderRadius: 4,
  },
  text: {
    fontSize: 22,
    color: Colors.black,
    alignSelf: "center",
  },
  image: {
    width: 35,
    height: 35,
    resizeMode: "contain",
    borderRadius: 500,
    marginHorizontal: 8,
  },
});
```

Listing 4. StyleSheet-object with classes to style different elements in SubredditListItem-component.

These StyleSheet-objects can be passed to the component with the style-property as shown in Listing 3.

4 Jetpack Compose – New competitor among UI frameworks

Jetpack Compose is a framework developed by Google for creating native Android applications using Kotlin [11]. Compose had its first stable public version released in July 2021 [12].

Supporting modern features such as component previews and animation previews are some of the most prevalent selling points for reasons to start using Compose.

4.1 What is Jetpack Compose

Jetpack Compose is a framework that helps developers use modern developing practices by utilizing reusable components and providing built-in options to implement dark theme and animations in projects. Compose is built on top of the original Jetpack architecture allowing developers to still benefit from all the functionality available on native Android development.

Compose heavily encourages the use of reusable components in projects called Composables. These composables can be nested to reuse as much code as possible. Composables can be elements such as a button or list of objects that can be reused with different data sources without having to write boilerplate code. Listing 5 shows an example of a composable with a string-parameter being rendered on the UI.

```
class FeedActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ComposeReaderTheme {
                HelloWorldText(text = "Hello world!")
            }
        }

    }
}

@Composable
fun HelloWorldText(text: String) {
    Text(text = text)
}
```

Listing 5. Rendering a composable in an activity. The composable is created outside of the Activity and is initialized in the setContent-function of the Android activity.

As seen in Listing 5 the Activity contains a setContent-function similarly as in traditional Android development. Since Compose does not use XML to layout its views it is not provided with a reference to an XML-file. The function is instead passed a Composable. In Listing 5 the structure is wrapped with a Theme-composable that passes each theme property to its children.

4.2 Architecture

Instead of using XML to layout components Compose handles everything inside the Kotlin-files. Compose uses a Kotlin compiler plugin that comes with the Compose framework. The @Compose annotation is a keyword for Compose that changes the type of the object and allows Compose to recognize it as a Composable [13]. As seen in Figure 2, Compose uses a Gap Buffer data structure to handle the recomposition.

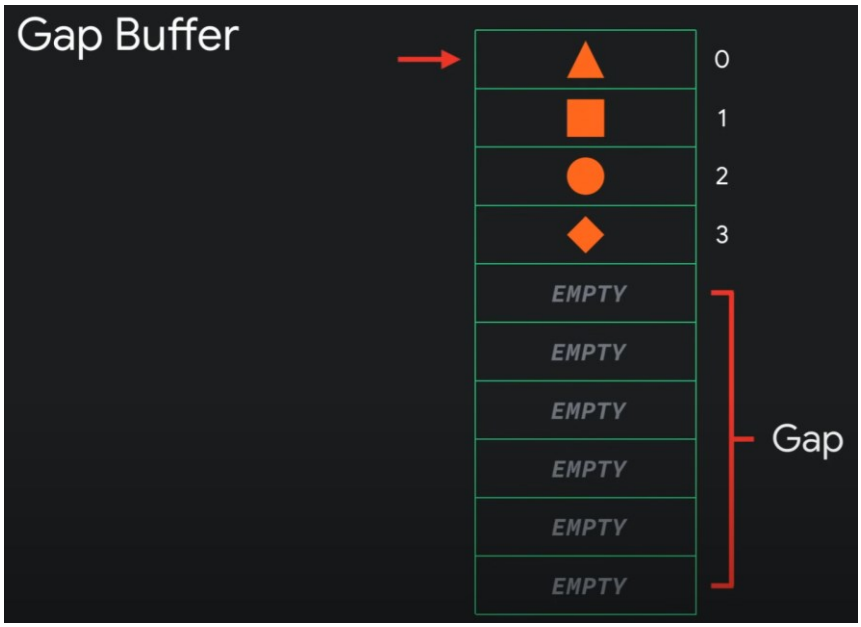

Figure 2. How Compose utilizes a data structure called Gap Buffer [13].

Figure 2 shows that components are inserted in the Gap Buffer in the order that they appear in. If during one the render-cycles the UI hierarchy has changed, and new elements must be inserted between components the empty slots are moved to the current location. [13]. This can be seen in Figure 3.

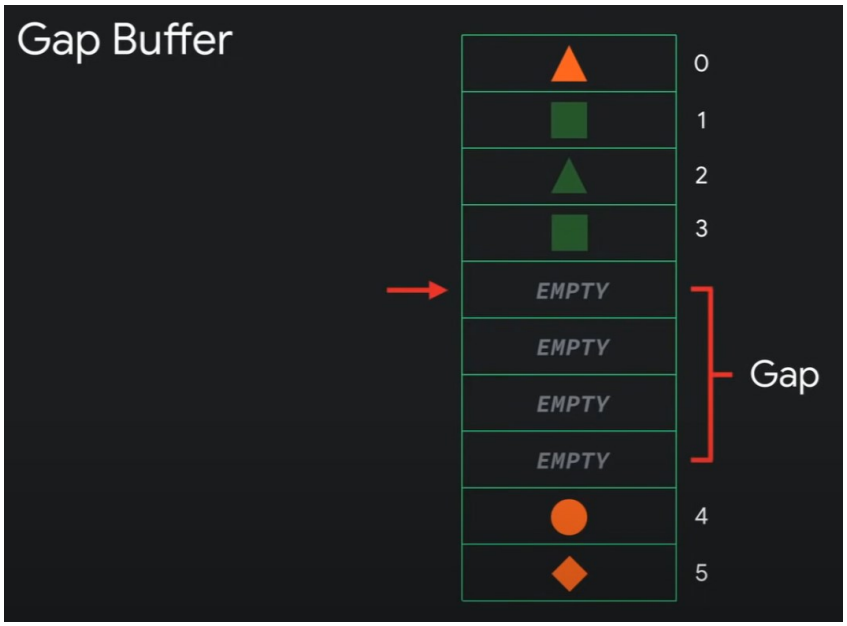

Figure 3. Gap inside the Gap Buffer is inserted in the middle of the buffer if new elements are added [13].

As shown in Figure 3, the Gap is inserted where the new components will be inserted to make room for them. Composables that appear after the new additions are pushed to the bottom of the buffer. [13.]

Composables must be annotated with the @Composable annotation to be injected with a Compose-object and an Integer-object during compiling as seen in Figure 4.

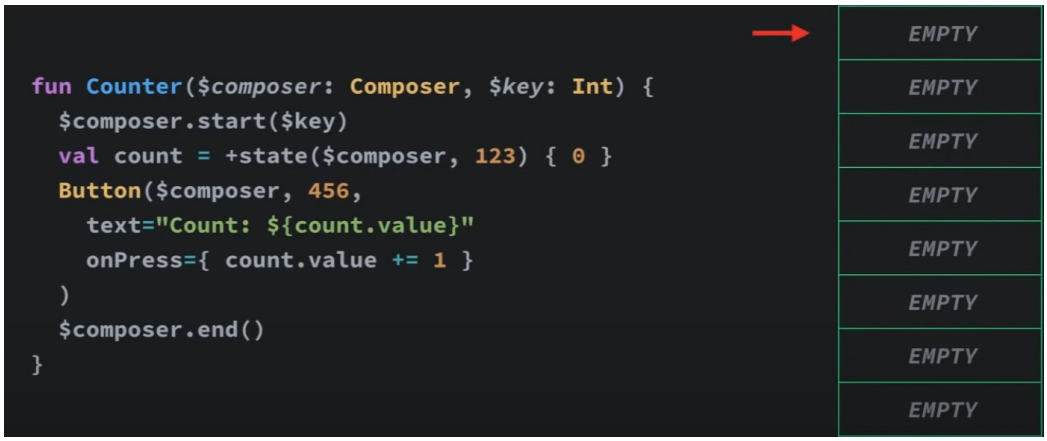

Figure 4. Composables are injected with a Composer-object and an Integer during compiling [13].

Figure 4 shows the variables that are injected when the composable is compiled. The functions start with an internal Composer-function called start and end with the same objects function called end. Figure 5 highlights how elements are inserted in the Gap Buffer.

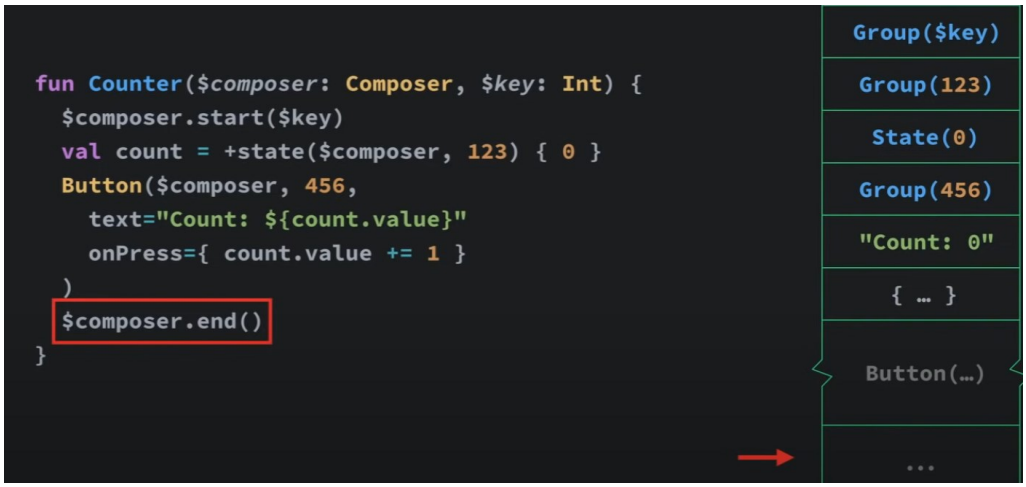

Figure 5. Variables being added to the Gap Buffer sequentially [13].

As seen in Figure 5 the Gap Buffer is filled with variables in the order that they appear in. These variables are assigned a group and the group is used to differentiate the objects from each other. If the group-key does not match while iterating through the Gap Buffer, the compiler knows that there has been a change in the UI. [13] This is where the empty slots are inserted as shown in Figure 3.

Compose renders the application by using a ComposeView-object. ComposeView extends View from Android and the Kotlin compiler plugin handles how the Compose-code should be rendered on the display. [14.]

4.3 Syntax

Building UIs with Compose is similar to how React Native UIs are built. It utilizes reusable components to minimize the amount of code needed and to avoid repetition. Elements are nested inside each other in a similar fashion as in HTML. Example component is shown and explained in the next chapter.

4.3.1 Class declaration

Compose components are written the same way as normal Kotlin-functions with the difference being they do not return a value and are annotated with a @Composable-annotation. Listing 6 shows the structure of a composable with a state.

```
@Composable
fun MessageCard(msg: Message) {
    Row(modifier = Modifier.padding(all = 8.dp)) {
        Image(
            painter = painterResource(R.drawable.profile_picture),
            contentDescription = null,
            modifier = Modifier
                .size(40.dp)
                .clip(CircleShape)
                .border(1.5.dp, MaterialTheme.colors.secondaryVariant,
                 CircleShape)
        )
        Spacer(modifier = Modifier.width(8.dp))
        var isExpanded by remember { mutableStateOf(false) }

        Column(modifier = Modifier.clickable { isExpanded = !isExpanded }) {
            Text(
                text = msg.author,
                color = MaterialTheme.colors.secondaryVariant,
                style = MaterialTheme.typography.subtitle2
            )

            Spacer(modifier = Modifier.height(4.dp))
            Surface(
                shape = MaterialTheme.shapes.medium,
                elevation = 1.dp,
            ) {
                Text(
                    text = msg.body,
                    modifier = Modifier.padding(all = 4.dp),
                    maxLines = if (isExpanded) Int.MAX_VALUE else 1,
                    style = MaterialTheme.typography.body2
                )
            }
        }
    }
}
```

Listing 6. Structure of a composable that remembers its state and utilizes Composes styling conventions [11].

As seen in Listing 6 the state is initialized with the word remember and it is passed a mutableStateOf-object. The state is stored in memory and any updates done to the object will trigger the re-rendering of the composable. This is called recomposition. [11]. Recomposition is also triggered if a child of the composable updates the value. To update a composable the developer is encouraged to call

the composable function with new data instead of using a setter to change the data. Re-rendering the activity requires more computing power and consumes more battery than updating only composables that request recomposition. [15.]

4.3.2 Styling elements

Compose provides the developer with a tool to preview composables. These previews allow the developer to do changes such as update paddings and margins with immediate updating instead of building the project again. The preview-tool can show multiple composables at the same time to avoid having to transition in the application manually and it can render the composables in multiple themes concurrently. At the expense of performance, previews can be started in interactive mode. Interactive mode enables gesture controls, data updates and animations on the composable preview to test the functionality without the need for an emulator or a physical device.

Styling composables is done inside the function instead of passing it a style-object or using XML (Extensible Markup Language) similarly as in traditional Android development. Figure 6 shows the preview generated by using the composable created in Listing 6.

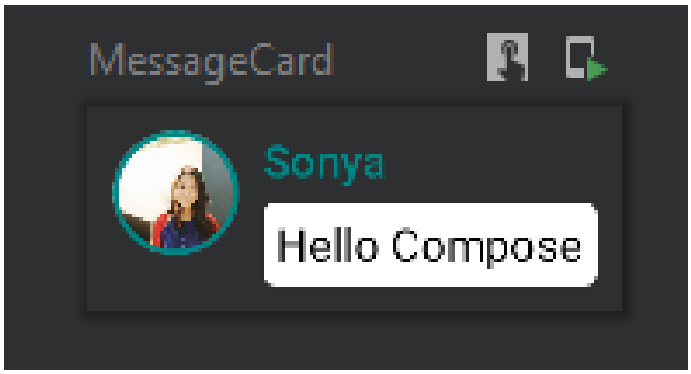

Figure 6. Preview generated by Android Studio with code shown in Listing 6.

As seen in Listing 6, each views' properties are split when compared to traditional Android development. Properties that all of the views share are passed inside their modifier-property. These include properties such as paddings, margins, and sizing. The modifier is passed a Modifier-object that stores all these values.

Utilizing the Modifier-object allows Android Studio to show changes done to these modifiers inside the previews immediately. Live previews are useful when the developer is trying to mimic a prototype as close as possible.

When a composable is ran in Interactive mode its animation values can be inspected at any given time. The animation can be paused, slowed down to debug and fine tune the animation more easily. [16.]

5 Developing the applications

Target of the thesis was to create two applications with the same specifications. One was developed using React Native and the other with Jetpack Compose. These applications and their development were compared with each other to determine factors that should be accounted for when starting to develop a new project.

5.1 Reddit API as a data source

Reddit is a social media that lets its users subscribe to communities that interest them and communicate with users who share their interests. These communities are described as subreddits. Subreddits are prefixed with “r/” ending with the name of the subreddit (e.g., r/cars or r/dogs). Each subreddit allows users to create new posts and comment and vote on existing ones [17]. Reddit possesses over 52 million active daily users [17.] which made the service a reliable source for large amounts of data. Reddit provides developers access to their data if their terms are followed correctly. These terms include rules such as oAuth2-authentication must be implemented, and the application must not include Reddit in its name. [18; 19.]

5.2 Specifications of the application

To compare the technologies in similar conditions the two applications were given the same specifications:

- User can login to their personal Reddit-account with oAuth2-validation.
- Show the user a list of subreddits they are subscribed to.
- Implement a bottom navigation bar.
- Display popular posts from the subreddits the user is subscribed to.

5.3 Developing the Jetpack Compose -application

The Compose application was developed using Android Studio as the integrated development environment.

5.3.1 Creating the project

Creating a Jetpack Compose project requires Android Studio Arctic Fox (version 2020.3.1). Android Studio provides an empty template Compose-project from its New Project -tab. [20]. This process required no changes from the user and took less than five minutes to build the application on a physical device or an emulator.

Upon creation many of the dependency versions were outdated but Android Studio can change each version to their latest stable version by hovering the dependency and clicking the latest version. To access a broader selection of Compose-features there were dependencies that can be added to the projects gradle-file. These dependencies were provided in the setup documentation.

5.3.2 Documentation available

Googles developers have added Compose its own section in their developer documentation. All the properties and classes for Compose are documented and explained with a great deal of detail. The documentation contains many step-by-step tutorials that thoroughly explain how composables are built and rendered.

Sample projects are linked to display projects that are developed using proper coding conventions.

Upside of the documentation residing in the same domain allows developers to easily access other Android documentation. Both documentations share the same documentation conventions.

5.3.3 Navigation

Jetpack Compose bottom navigation relies on a NavController-object that is part of Androids androidx-library. The NavController is responsible for remembering the navigation stack and handles the transitions between components. [21]. The development of the bottom navigation bar started by defining the different pages as shown in Listing 7.

```
sealed class NavigationItem(var route: String, var icon: ImageVector, var
title: String) {

    object Feed :
           NavigationItem("feed", Icons.Rounded.Home, "Feed")
    object All :
           NavigationItem("all", Icons.Rounded.Book, "r/All")
    object Profile :
           NavigationItem("profile", Icons.Rounded.VerifiedUser, "Profile")
}
```

Listing 7. Sealed class that holds the three different NavigationItems needed for the bottom navigation bar.

Each NavigationItem is defined inside the sealed class with three properties shown in Figure 7. Route is of type String, and it is provided to the NavController to navigate to the correct component. Icon is used as the graphic for the item and title is the text displayed under the icon. These are the three different pages needed for the developed application.

The next step was to implement the layout for the bottom navigation bar. This is shown in Listing 8.

```
@Composable
fun BottomNavigationBar(navController: NavController) {
    val items = listOf(
        NavigationItem.Feed,
        NavigationItem.All,
        NavigationItem.Profile
    )
    BottomNavigation(
        //Style properties
    ) {
        val navBackStackEntry by navController.currentBackStackEntryAsState()
        val currentRoute = navBackStackEntry?.destination?.route
        items.forEach { item ->
            BottomNavigationItem(
                icon = { Icon(item.icon, contentDescription = item.title) },
                label = { Text(text = item.title) },
                //Style properties
                selected = currentRoute == item.route,
                onClick = {
                    navController.navigate(item.route) {
                        navController.graph
                          .startDestinationRoute?.let { route ->
                              popUpTo(route) {
                                  saveState = true
                              }
                        }
                        //State saving
                    }
                }
            )
        }
    }
}
```

Listing 8. Composable for the bottom navigation bar.

As shown in Listing 8 the objects initialized in Listing 7 are passed to a composable called BottomNavigation. BottomNavigation is a class provided by Compose. Optional properties were cut out of the example to only highlight necessary values.

The final step to implementing a functional navigation bar is to insert the composables to the root of the activity. This is shown in Listing 9.

```
@Composable
fun MainScreen() {
    val navController = rememberNavController()
    Scaffold(
        topBar = { TopBar() },
        bottomBar = { BottomNavigationBar(navController) }
    ) {
        Navigation(navController)
    }
}

@Composable
fun Navigation(navController: NavHostController) {
    NavHost(navController, startDestination = NavigationItem.Feed.route) {
        composable(NavigationItem.Feed.route) {
            ListOfSubreddits()
        }
        composable(NavigationItem.All.route) {
            ListOfPosts()
        }
        composable(NavigationItem.Profile.route) {
            ProfilePage()
        }
    }
}
```

Listing 9. Inserting the navigation items to the root of the activity.

Listing 9 shows a similar design pattern that is used in the React Native application covered later in this thesis. In Compose Scaffold closely resembles a component called SafeAreaView used in React Native. These are both used to avoid displaying elements inside display notches, embedded front cameras, and other non-standard device features. The Scaffold is being passed the composable from Listing 8.

The MainScreen composable is passed a reference to the Navigation composable that defines the components that should be rendered in each route. The earlier mentioned NavController is initialized in the MainScreen composable.

5.3.4 Network requests from the API with LiveData

Connecting the application to the Reddit API started with creating the ViewModel. The ViewModel is responsible for storing the LiveData in a variable and fetching the network request. The ViewModel is shown in Listing 10.

```
class SubredditListModel : ViewModel() {
    val _list: LiveData<SubredditList> get() = list
    val list = MutableLiveData<SubredditList>()

    init {
        val accessToken = BearerHandler
                            .bearerHandler.getAccessToken()
                            .toString()

        val gson = Gson()
        val httpclient = OkHttpClient()
        val request = Request.Builder()
            .url("https://oauth.reddit.com/subreddits/mine/subscriber.json")
            .header("User-Agent", "App by Developer")
            .header("Authorization", "Bearer $accessToken")
            .build()

        viewModelScope.launch {
            Thread {
                val response = httpclient.newCall(request).execute()
                list.postValue(
                    gson.fromJson(
                        response.body!!.string(),
                        SubredditList::class.java)
                )
            }.start()
        }
    }
}
```

Listing 10. ViewModel-class used to fetch and store the list of subreddits the user is subscribed to.

As seen in Listing 10, to utilize the LiveData design pattern there are two variables of the list of subreddits. As soon as the network request returns a value it is set to the list-variable. Setting the updated value is done with the MutableLiveData-objects postValue-function. This list variable is observed in a Composable and is used to render the list on the display as shown in Listing 11.

```
@Composable
fun ListOfSubreddits(model: SubredditListModel = viewModel()) {
    val subredditList = model._list.observeAsState()
    LazyColumn {
        subredditList.value?.data?.let {
            items(it.children) { subreddit ->
                Subreddit(subreddit.data)
            }
        }
    }
}
```

Listing 11. Composable function that observes the state from the ViewModel. This composable utilizes a LazyColumn Composable that can render a list that carries a dynamic length.

The Composable shown in Listing 11 shows how Compose is used to render a list of items using a LazyColumn. LazyColumn follows the same design patterns as a RecyclerView in Android [22]. It requires a dataset to render, in this case a list of subreddits received from the ViewModel. Each item is given their styling and layout properties in similar fashion as shown in Listing 6.

5.4 Developing the React Native application

The React Native application was developed using Expo. Expo provides its own application called Expo Go that can be downloaded from the App Store or Play Store. Using Expo Go provides the developer with a way to build and share the application during development. Expo provides their own development libraries that allow developers to access a quick way to utilize a larger variety of features to make developing the application easier.

Visual Studio Code was used as the integrated development environment as it offers plugins to format JavaScript correctly.

5.4.1 Creating the project using Expo

There are multiple ways to initialize a RN-project with Expo. For this thesis Node Package Manager (npm) was used. When installing npm the installation includes npx (Node Package Execute). Listing 12 shows the process of creating a project with the name SampleProject and starting the Expo service.

```
npx react-native init SampleProject
cd SampleProject
expo start
```

Listing 12. Initializing and launching a RN-project with the name SampleProject.

As shown in Listing 12 starting the Expo service opens Metro bundler on a new tab on the developers default browser. Metro bundler provides a QR-code that can be read with Expo Go application to build the project on the users' device. The bundler shows every device that compiled the application and outputs all logging events from them. It carries the functionality to test the applications

performance and share it with other users to let them test the application without needing to install it on their device. This can be utilized to check if any device is emitting either console warnings or errors.

5.4.2 Documentation available

RN possesses a great amount of easily understandable documentation that is available online. Each view has all its properties thoroughly explained and contain multiple code examples on how to use them. React Native shares a great deal of functionality with React allowing developers to refer to its documentation also.

5.4.3 Creating the bottom navigation

SafeAreaProvider is a component from Expo that allows the application to correctly consider notches, front cameras, and other device-specific elements that other devices may not be equipped with [23]. Therefore, SafeAreaProvider is simply there to make developing easier and not mandatory for navigation or other reasons.

During development there was an observation that some React Native components behave differently on iOS-devices and Android-devices. These were components such as the status bar and the header bar. These components are given their own default styling properties depending on the platform they are built on.

React Native provides its own libraries to implement a bottom navigation bar without the need to import any third-party libraries. As seen in listing 13 the root function of the application is wrapped inside a NavigationContainer.

```
const Stack = createNativeStackNavigator()
export default function App() {
  return (
    <SafeAreaProvider>
      <NavigationContainer>
        <Stack.Navigator
          /*
           Styling properties
           */
          initialRouteName="Login"
        >
          <Stack.Screen name="Login" component={LoginScreen} />
          <Stack.Screen
            name="Tabs"
            component={Tabs}
            options={{ headerShown: false }}
          />
          <Stack.Screen name="PostDetails" component={PostDetails} />
        </Stack.Navigator>
      </NavigationContainer>
    </SafeAreaProvider>
  )
}
```

Listing 13. RN-application wrapped inside navigation components.

As seen in Listing 13 the Stack-object handles events such as a back-button press to return to a previous screen. Inside the stack the different pages are nested on top of each other. The application that was developed required a separate login-page and the page with the bottom navigation bar. Therefore, two Screen-objects were nested inside the stack. These Screen-objects require a name that is used to navigate to the correct screen, and they require a component that defines how the screen should be rendered. As seen in Listing 14, the tabs are very similarly defined as the Stack Navigator but instead of nesting Stack.Screen-objects there are Tab.Screen-objects.

```
const Tab = createMaterialBottomTabNavigator()
const Tabs = () => {
  return (
    <Tab.Navigator
      shifting={true}
      barStyle={{ backgroundColor: Colors.lightGrey }}
      activeColor={Colors.black}
      inactiveColor={Colors.darkGrey}>
      <Tab.Screen
        name="Feed"
        component={FeedScreen}
        options={{
          tabBarIcon: ({ color }) => (
            <MaterialCommunityIcons name="home" color={color} size={28} />
          ),
        }}
      />
      <Tab.Screen
        name="r/All"
        component={DetailsScreen}
        options={{
          tabBarIcon: ({ color }) => (
            <MaterialCommunityIcons name="fire" color={color} size={28} />
          ),
        }}
      />
      <Tab.Screen
        name="Profile"
        component={ProfileScreen}
        options={{
          tabBarIcon: ({ color }) => (
            <MaterialCommunityIcons
              name="account-key"
              color={color}
              size={28}
            />
          ),
        }}
      />
    </Tab.Navigator>
  )
}
```

Listing 14. Bottom navigation bar tab-hierarchy. Each screen of a navigation bar is nested inside the Tab.navigator.

As seen in Listing 14 each screen is passed their name, component and the icon used in the bottom navigation bar with a tabBarIcon-property. For this project a third-party library MaterialCommunityIcons was used to implement the icons.

5.4.4 Fetching the data

Network requests were done with vanilla JavaScript without the use of third-party libraries. Listing 15 shows an example fetch-request of the subreddits the user is subscribed to.

```
getSubreddits: () => {
    let list
    return new Promise((resolve, reject) => {
      try {
        fetch("https://oauth.reddit.com/subreddits/mine/subscriber.json", {
          method: "GET",
          headers: {
            Accept: "application/json",
            "Content-Type": "application/json",
            "User-Agent": "App by developer",
            Authorization: `${bearer}`,
          },
        }).then(async (response) => {
          await response.json().then((json) => {
            list = json
            resolve(list)
          })
        })
      } catch (error) {
        console.log(error)
      }
    })
  },
```

Listing 15. A function that returns a Promise containing all the subreddits that the user is subscribed to.

As seen in Listing 15 the function returns a Promise-object. The Promise is resolved as a list of items in JSON-format. The data is fetched with a GET-request with headers providing Reddit with the mandatory information on who is requesting the data. User-Agent is used to verify that the developer is the same developer as registered on the Reddit API.

Figure 7 shows the list of subreddits rendered on the screen inside a FlatList-object.

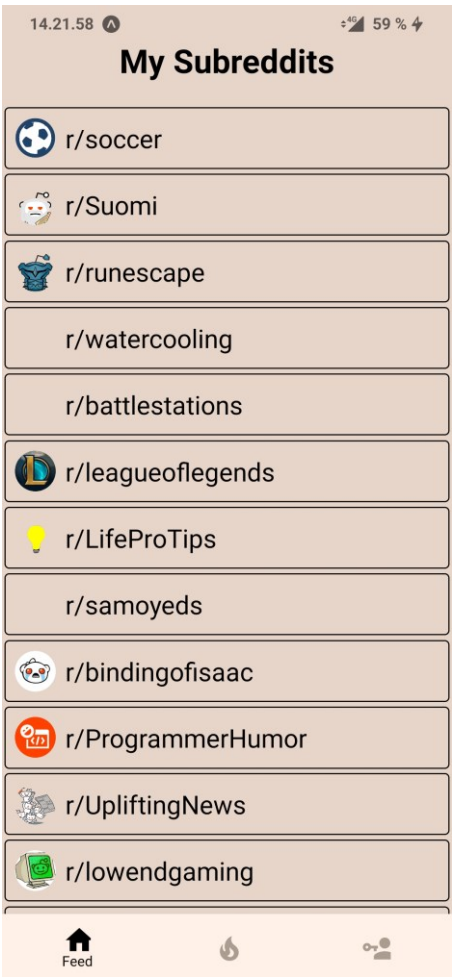

Figure 7. Rendering data using the function from Listing 15 on the first page of the bottom navigation bar.

Figure 7 displays the subreddits the user is subscribed to. The list uses code from Listings 1, 2, 3 and 15. React Native can benefit from the Promise-architecture from JavaScript which enables the use of design patterns found in web development.

6 Performance testing

The developed applications were compared in multiple different performance categories using Perfetto and the built-in application profiling provided by each framework. Perfetto is a performance monitoring software.

6.1 Compiling

Two different bundling times were measured. The first test was the initial time taken to bundle the application and build it on the device. The second value is the subsequent build times after the initial one. Subsequent builds were built five times and the average time was reported. For Compose this was reported by Android Studio and for React Native reporting was done by the Metro Bundler. Results are shown in Figure 8.

Figure 8. Initial bundling time in seconds. Each value is an average of 5 results.

App	Initial bundle time	Subsequent build time
Compose	22.5 seconds	1112 milliseconds
React Native	21.5 seconds	54 milliseconds

As seen in Figure 8 the initial bundling times are close to one another. Considering this is only the initial bundling time it is a non-factor in development. Subsequent build times on the other hand are significantly quicker on React Native. This makes it more enjoyable to do small tweaks and changes to the application. Compose does support live editing of properties but they are restricted to properties such as paddings, margins, and text sizes.

6.2 Touch responsiveness

Rendering speed was tested with Perfetto. The test activity for both frameworks had a button without any text inside it. Once the button was clicked, the whole activity was re-rendered, and five new buttons were added below the original button. The time started once the Android devices' SurfaceFlinger-service reported a new touch. The time was stopped when the RenderThread-service was put back to sleep. These results are shown in Figure 9.

Figure 9. Time it takes from the device reporting a touch to end re-rendering the activity. Each value is an average of five results.

App	Time elapsed
Compose	450 milliseconds
React Native	371 milliseconds

As Figure 9 shows, React Native managed a slight edge over Compose in re-rendering an activity. The time difference is large enough to be noticeable by the user but Compose was not far behind.

6.3 Dynamic lists – The benchmark for performance

Nearly every application must implement a dynamic list to render on the UI. Therefore, it is a prime benchmark to compare different technologies as it requires CPU and GPU -computing while also showing how memory-efficient the framework is when its rendering dynamic lists. Figure 10 shows that a native application was able to render the list with just a fifth of the CPU power required by React Native.

Figure 10. Resources used for rendering a list of 100 objects each with an image loaded from a URL. [24]

App	CPU %	Memory Consumed (Max)	Battery Usage
Native Android	2.6	72 Mb	56.6 mAh
Flutter	5.6	106 Mb	69.2 mAh
React Native	12.1	128 Mb	78.7 mAh

As seen in Figure 10 the React Native application used almost twice as much memory and 39% more battery to output the list on the screen. This will negatively impact the users' device battery life when using the application. High battery consumption is due to the large amount of processing power needed by the CPU. It is plausible to assume that rendering different views also requires more computing power and memory on React Native.

6.4 Animations using Lottie

Animations are more and more important as companies are trying to achieve the most user-friendly UIs possible. Animations make the application feel more responsive, easier to follow and keeps the user interested in the application. Lottie is a library that renders Adobe After Effects animations natively on Android, iOS, and web applications without the need to create the animation by hand in the codebase [25].

Figure 11 shows that while React Native consumed slightly less CPU power, it required over 50% more memory to render the animation.

Figure 11. Resources consumed by rendering a Lottie animation on the UI. [24]

App	CPU %	Memory Consumed (Max)	Battery Usage
Native Android	16.8	172 Mb	14.76 mAh
Flutter	11.3	238 Mb	13.40 mAh
React Native	14.5	264 Mb	13.17 mAh

As seen in Figure 11 the React Native application required a considerably larger amount of memory to perform the animation. If there are multiple concurrent animations running it can impact the rendering time of a frame. Increased computation time can lead to dropped frames and a choppy user experience.

7 Conclusion

Every time a new development project is started there must be a decision on which technology will be used. This thesis compared two frameworks and excluded other viable options.

When comparing the difficulty of developing a project with Compose and React Native, it should be considered that because of React Natives maturity and large userbase there are more tutorials, libraries, and discussion around React Native. Having access to great documentation provided by Facebook and endless discussions online on different bugs and problems made developing the application faster than with Compose. Compose being released in the summer of 2021 made finding discussions online harder. This hindered the problem-solving process when something did not work as intended. Since Compose is still very new there is a plausible reason to believe that there will be more information found on the internet after developers have been using it for a longer time.

In terms of design patterns, I believe Compose, and other object-oriented programming languages make it easier to split code into maintainable small parts. React Native relies on using several pre-built implementations of classes and components, which can make it harder to understand the application's high-level architecture, since abstraction layers are hidden from the developer. React Native does a large part of its work hidden from the developer when it is transforming elements into native iOS/Android-views.

If the company developing the application possess the resources to manage two codebases and hire developers who can develop the project with native technologies separately then they should be preferred. Performance of a native application is a valid reason to pick native technologies over cross-platform tools to create a better user experience.

Native applications performed better in many sections of the performance testing. Results lean over to Compose being faster or close to even in each category. React Native was slightly quicker in the rendering test. The differences are

difficult for the user to notice since they are small and information such as memory usage or battery consumption are not displayed. If the application has many animations or does heavy computations, it will be better to stay away from cross-platform tools to avoid effecting the user experience.

When it comes to build times excluding the initial bundling, React Native is the dominant tool. Compose previews and Android bundling usually take seconds, while React Native building could be described as instant.

This research could be continued by developing an application with a larger scope. The developed application could be tested from several aspects that were not considered in this study. Continuing the application development process could lead to finding more advantages or disadvantages specific to each framework.

Cross-platform tools can be useful if the company is interested in creating a prototype application. It is faster and cheaper to develop an application using a cross-platform framework since they make the application available to more users and only require the work to be done once. This opens the possible route to see if an application would interest possible users and it can be used as a prototype for user testing. After identifying if there is a market for such application, it can be developed using for example Compose and SwiftUI, which is Apple’s equivalent to Jetpack Compose.

References

1 Handley, Lucy. 2019. Nearly three quarters of the world will use just their smartphones to access the internet by 2025. Online. CNBC. <https://www.cnbc.com/2019/01/24/smartphones-72percent-of-people-will-use-only-mobile-for-internet-by-2025.html>. Read 27.8.2021.

2 Rathi, Reshu. 2021. Mobile-First Web Design: Why You Should Make It A Priority In 2021? Online. Lambdatest. <https://www.lambdatest.com/blog/mobile-first-web-design/>. Read 27.8.2021.

3 Marchuk, Anastasiya. 2021. Native Vs Cross-Platform Development: Pros & Cons Revealed. Online. uptech. <https://www.uptech.team/blog/native-vs-cross-platform-app-development/>. Read 8.10.2021.

4 Vaughn, Lance. 2019. The Importance of Improving Mobile App Performance. Online. iTexico. <https://www.itexico.com/blog/the-importance-of-improving-mobile-app-performance>. Read 26.8.2021.

5 Singh, Ritik. 2021. (Updated) List of Phones with 90Hz and 120Hz Display Refresh Rate. Online. Gadgets To Use. <https://gadgetstouse.com/blog/2021/04/05/list-of-smartphones-with-90hz-and-120hz-refresh-rate-display/>. Read 25.10.2021.

6 The history of React Native: Facebook's Open Source App Development Framework. 2016. Online. TechAhead. <https://www.techaheadcorp.com/blog/history-of-react-native/>. Read 28.9.2021.

7 React Native Internals. Online. React Native. <https://www.reactnative.guide/3-react-native-internals/3.1-react-native-internals.html>. Read 1.10.2021.

8 JavaScriptCore. Online. Developer Apple. <https://developer.apple.com/documentation/javascriptcore>. Read 1.10.2021.

9 State. Online. React Native. <https://reactnative.dev/docs/state>. Read 11.10.2021.

10 FlatList. Online. React Native. <https://reactnative.dev/docs/flatlist>. Read 12.10.2021.

11 Jetpack Compose Tutorial. Online. Developers Android. <https://developer.android.com/jetpack/compose/tutorial>. Read 5.10.2021.

12 Bellini, Anna-Chiara. Jetpack Compose is now 1.0: announcing Android's modern toolkit for building native UI. Android Developers Google Blog. <https://android-developers.googleblog.com/2021/07/jetpack-compose-announcement.html>. Read 1.10.2021.

13 Understanding Compose (Android Dev Summit '19). Online. Android Developers. <https://www.youtube.com/watch?v=Q9MtlmmN4Q0>. Accessed 25.10.2021.

14 ComposeView. Online. Developers Android. <https://developer.android.com/reference/kotlin/androidx/compose/ui/platform/ComposeView>. Read 26.10.2021.

15 Thinking in Compose. Online. Developers Android. <https://developer.android.com/jetpack/compose/mental-model#recomposition>. Read 5.10.2021.

16 Compose tooling. Online. Developers Android. <https://developer.android.com/jetpack/compose/tooling >. Read 5.10.2021.

17 Dive Into Anything. Online. Reddit. <https://www.redditinc.com>. Read 26.8.2021.

18 Wardle, Josh. 2015. API. Online. GitHub. <https://github.com/reddit-archive/reddit/wiki/API>. Read 26.8.2021.

19 Reddit API Access. 2016. Online. Reddit. <https://www.reddit.com/wiki/api>. Read 26.8.2021.

20 Use Android Studio with Jetpack Compose. 2021. Online. Developers Android. <https://developer.android.com/jetpack/compose/setup>. Read 27.8.2021.

21 Navigating with Compose. 2021. Online. Developers Android. <https://developer.android.com/jetpack/compose/navigation>. Read 5.10.2021.

22 Lists. Online. Developers Android. <https://developer.android.com/jetpack/compose/lists>. Read 26.10.2021.

23 SafeAreaContext. Online. Expo documentation. <https://docs.expo.dev/versions/latest/sdk/safe-area-context/>. Read 18.10.2021.

24 Chellakannu, Gunalan. 2020. Android App's Performance – Native vs Flutter vs React Native. Online. Perficient. <https://blogs.perficient.com/2020/11/02/android-app-native-vs-flutter-vs-react-native/>. Read 28.9.2021.

25 Lottie for Android, iOS, Web, React Native, and Windows. Online. AirBnb. <https://airbnb.io/lottie/#/>. Read 28.9.2021.