Industrial Engineering Journal
ISSN: 0970-2555
Volume : 53, Issue 1, No. 1, January : 2024

NATIVE ANDROID VS REACT NATIVE

Dr. Mona Deshmukh, Associate Professor, Dept. of Master of Computer Applications, Vivekanand Education Society's Institute of Technology, (VESIT), Mumbai, Maharashtra, India
Dr .Girish Deshmukh Professor, Dept of Mechanical Engineering, A C Patil college of Engineering, Navi Mumbai, India

I. ABSTARCT

Mobile applications are playing an increasingly important role in our daily lives. Since November 2016, there is more network traffic made by mobile devices (48.19%) compared to desktops/laptops (47%).[1] To distribute to most of the users, a mobile application needs to adapt itself into two separate platforms, namely Android and iOS. Evidently, the differences between these two platforms are big and often require different skill sets for developing, such as Java/Kolten only for Android and Object-C/Swift only for iOS. Thus, developers and companies often struggle at dealing with the complexity of developing cross-platform applications. React Native, an open-source cross platform JavaScript framework, which aims at solving the above-mentioned dilemma, was introduced by Facebook on March 2015. It is based on the React framework, which is published by Facebook a few years earlier.[2] React framework is widely used by developers due to its simplicity and easy but also for its effective developing process.[3] React Native allows the use of native modules within the code for specific native-OS features (push notifications, camera, or third-party services). React Native apps are also available in Apple's App Store and Google's Play Store. Sometimes linking libraries between React Native and native modules can be tricky, and a helping hand of an experienced iOS & Android developer is invaluable. The issue is similar for publishing the app: it is a lot easier for native developers who are already familiar with the procedure and documentation needed.

II. INTRODUCTION

This is less of an issue for agencies where developers can usually consult one another freely, and more a struggle for start-ups and internal teams who might not have readily available access to consult anyone fluent in native mobile app development. The fundamental concepts and characteristics for both platforms will be explained and demonstrated. Comparisons, in terms of performance and developing process, between React Native and Native Android will be covered in this paper. Furthermore, to expose the differences between React Native and Native Android application, a fully working React Native application will be rewritten using Native Android as a supplement.

Developing an app in OS native technologies will always give you access to native-OS features (SDKs, sensors, specific hardware) with no problems or workarounds necessary. It means that if you're planning on developing an app heavily relying on OS, using ARKit, needing access to hardware (camera, Touch ID), it is better to consider native solutions rather than cross-platform ones.

It's also no surprise that both Apple (iOS) and Google (Android) provide the necessary support for their native applications, and if they develop a new OS feature or add a new hardware element, the access to it in native OS languages is always much easier and available faster compared to cross-platform solutions.

III. Working with React Native JSX

One of the easiest spots to be observed, when one studies React Native application's code, is the usage of JSX. JSX is a special syntax extension to JavaScript, which is used fundamentally to describe how the UI should show. JSX will be compiled into normal JavaScript object when the application gets compiled. Since JSX is used to describe UI, one may argue JSX is plainly another template language, such as HTML or XAML. But this is incorrect. JSX and normal JavaScript object are multi-

Industrial Engineering Journal
ISSN: 0970-2555
Volume : 53, Issue 1, No. 1, January : 2024
convertible, which means we can write normal JavaScript expression in the middle of JSX. It is worth to mention that using JSX brings the benefit of preventing injection attacks. React DOM processes any inputted value into a regular string before rendering it. Therefore, user can never inject any scripts or commands into your application by its interface.

```
action1 = () => {
  console.log("Click Button 1")
}

action2 = () => {
  console.log("Click Button 2")
}

render() {
  return (
    <View>

      <Text>
        Sample Component
      </Text>

      <TouchableOpacity onPress={this.action1}>
        <Text>
          Button1
        </Text>
      </TouchableOpacity>

      <TouchableOpacity onPress={this.action2}>
        <Text>
          Button2
        </Text>
      </TouchableOpacity>

    </View>
  )
}
```

Figure 2. Workflow of Virtual DOM [5].

Virtual DOM

One of the main reasons why React Native application can run on different platforms is the usage of Virtual DOM. Virtual DOM allows React to manipulate a lightweight DOM tree, that is mapped with the real DOM tree, to gain the performance boost.

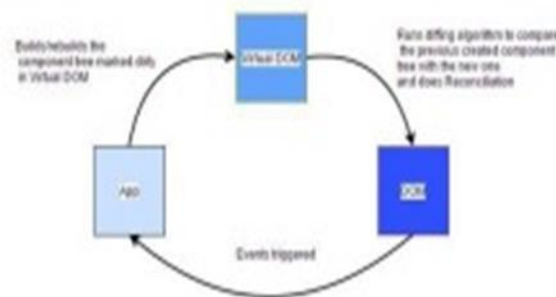

Figure 3. Process Virtual DOM [5].

In React, whenever a JSX element needs to be rendered, its corresponding virtual DOM object will get updated at the same time. Before updating, React will take a snapshot of the current Virtual DOM tree. Therefore, after updating, React can compare the updated DOM tree with the previous snapshot to find the exact parts which need to be re-rendered in the true DOM tree. This procedure requires a set of algorithms and it is called “diffing” in the React world. In addition to the effective “diffing” algorithm, React puts a lot of effort into batching DOMread/write operations. After finding the minimum number of steps to update the Virtual DOM, React executes all those steps in one event loop without touching the Real DOM. And only after the event loop get finished, React will repaint the Real DOM. Thus, there should be exactly one time when the Real DOM needed to be painted. Combining “diffing” and

finding out which nodes in the view tree need to be The software analyzes the strings of the selected phonemes and compares them to a database of known words, phrases, and sentences.

Props and State
In React, there are two major data models, which are called Props and State. They serve for different purposes and have distinct construction. Most observably, Props are set externally, and State is used only within the component's life cycle. However, they both are plain JavaScript objects and have direct impact on triggering render updates. Since Props is set externally by components' parent, Props' key function is to be used as parameters to customize the component when they are created. In other words, Props are the configuration of a component. It may be blank if it matches the needs but once it is set, it can never be changed. This nature is called immutable. On the other hand, State could only be initialized within the component when it gets mounted. Also, one can assign a new value to component's state whenever it suits. It is a common Practice that every user's input should have a corresponding impact to component's State. In a complex interactive React application, component's State often get passed as Props of its children's components.

Redux vs Mobx
Although Redux was chosen as the main design pattern for state management in our case study, there are other popular derivation of the Flux architecture that are developed and maintained by the community. Mobx is one of them. The most noticeable difference in Mobx when comparing to Redux, is the mutability of the state. As mentioned above, instead of altering the existing state, in Redux, the reducers will generate a new state whenever receiving an action. In Mobx, components can easily be notified and directly react to state's changes. Thus, there is no need to have pure functions, such as reducers, acting as the middleware. In another word, state in Mobx is evolved when needed and components subscribe and trigger the evolvement of state. Mobx, with the help of the built-in decorators, such as "@observable" or "@action", can be terser and tidier than Redux when developing. However, the ease and freedom of Mobx does act as a twoedged sword for developers. The lack of restriction on declaring multiple state is a plain violation of the principle

"Single Source of Truth". Moreover, since components directly observe the changes of the model, actions can be no longer a pure JavaScript object. Therefore, the possibilities of recording, serializing, storing and replaying action is gone. All above mentioned aspects raises the complexity for debugging. In conclusion, compares to Redux, Mobx enable one to kickstart the projects with fewer code and cleaner architecture. Its flat learning curve allows itself to be easily merged into the ObjectOriented Programming environment. Nevertheless, the freedom of constructing multiple state is a significant drawback. Strongly relying on Mobx' s internal mechanism to manage state could bring chaos and difficulties when the application grows.

1.2. Working with Native Android Kotlin
Using Kotlin for Android development, you can benefit from:
• Less code combined with greater readability. Spend less time writinyour code and working to understand the code of others.
• Mature language and environment. Since its creation in 2011, Kotlin has developed continuously, not only as a language but as a whole ecosystem with robust tooling. Now it's seamlessly integrated in Android Studio and is actively used by many companies for developing Android applications.
• Interoperability with Java. You can use Kotlin along with the Java programming language in your applications without needing to migrate all your code to Kotlin. • Support for multiplatform development. You can use Kotlin for developing not only Android but also iOS, backend, and web applications. Enjoy the benefits of sharing the common code among the platforms. • Code safety. Less

code and better readability lead to fewer errors. The Kotlin compiler detects these remaining errors, making the code safe.

- Easy learning. Kotlin is very easy to learn, especially for Java developers.
- Big community. Kotlin has great support and many contributions from the

community, which is growing all over the world. According to Google, over 60% of the top 1000 apps on the Play Store use Kotlin.

Widget

Widgets enable users to interact with an Android Studio application page. There are various kinds of widgets, such as Buttons and TextViews. To see all the widgets at your disposal, create a new application project called "Widgets" and select "empty activity".

Call your activity "MainActivity".

There are two components of each Android
activity: the XML (Extensible Markup Language) design (the beauty) and the Java text (the brains). On the activity_main.xml page, you can see the full widgets palette underneath the various layout options. State Management

State management is probably one of the most complicated challenges in Android development. The reasons behind the complexity of state management are tightly associated with Android architecture and in order to come up with a solution, some fundamental Android design decisions need to be understood.

Fragments were an attempt to solve this problem, by introducing onCreateView and onDestroyView and also with setRetainInstance method. With Fragments a clear separation between the View lifecycle and the Fragment lifecycle was created. Still when the Fragment is destroyed, we run into the same problems we have with the Activity.

Routing React Native Navigator

As for React Native framework, one of the core and major feature missing is a fully modernized and native experience navigation [9] . Fortunately, the community has produced and chosen a popular open-source library called "React Native Navigation (RNN)" to help developers. In order IV.to make RNN functional, all screen-components need to be registered as a routing-path at the application's entry point. Since Redux was chosen as our first-class state management, Store of the app states need to be passed as a register parameter. This is beneficial for Redux to turn navigating into dispatching an action, thus navigating could be recorded and replayed for debugging whenever needed. Below figure 5 contains the code snippet for registering component as routing paths.

React Navigator

Navigating to a destination is done using a NavController, an object that manages app navigation within a NavHost. Each NavHost has its own corresponding NavController. Nav Controller provides a few different ways to navigate to a destination, which are further described in the sections below.

To retrieve the NavController for a fragment, activity, or view, use one of the following methods:

Kotlin:

- Fragment.findNavController()
- View.findNavController()
- Activity.findNavController(viewId: Int) Java:

- NavHostFragment.findNavController(Fr agmen)
- Navigation.findNavController(Activity, @IdRes int viewId)
- Navigation.findNavController(View)

After you've retrieved a NavController, you can call one of the overloads of navigate() to navigate between destinations. Each overload provides support for various navigation scenarios, as described in the following sections.

Views

Both React Native and Android Native application respect the design of modularity. Meaning instead of writing apps page by page, developers need to decouple pages into components and then assemble them according to the needs. Using the layout inspector, one can easily observe how a page is constructed by numbers of components. It is worth mentioning that modularity also indicates reusability. A typical example is the usage of the card component in our case study. Card component is a low-level and stateless component/widget, which is self-constraint. In another word, it ensures that with the exact same input, the component will always be rendered in an identical format. To keep its nature of context- irrelevant, the styles need to be implemented and applied inside of the component. This class represents the basic building block for user interface components. A View occupies a rectangular area on the screen and is responsible for drawing and event handling. View is the base class for widgets, which are used to create interactive UI components (buttons, text fields, etc.). The View Group subclass is the base class for layouts, which are invisible containers that hold other Views (or other View Groups) and define their layout properties

Styling

React Native and Android Native holds different opinions regarding what styling essentially is and how it should be applied to views. There are historical and perspective reasons standing behind these differences. In below, we will briefly explain from which spec these distinctions come from and what problem they solve. Not surprisingly, React, as a base framework of React Native, has a strong impact on how React Native applications are organized and developed. From the perspective of web and the influence of Cascading Style Sheet (CSS), styling is treated as primitive rules that can be applied on top of view components. However, this doesn’t mean React Native application can directly use existing CSS files. A special parser class (Stylesheet) is created as a factory in order to generate CSS-like object, which can be furtherly applied onto view components. On the other hand, Styles and themes on Android allow you to separate the details of your app design from the UI structure and behaviour, similar to stylesheets in web design. A style is a collection of attributes that specify the appearance for a single View. A style can specify attributes such as font color, font size, background color, and much more. A theme is a collection of attributes that's applied to an entire app, activity, or view hierarchy—not just an individual view. When you apply a theme, every view in the app or activity applies each of the theme's attributes that it supports. Themes can also apply styles to non-view elements, such as the status bar and window background. Styles and themes are declared in a style resource file in res/values/, usually named styles.xml.

Industrial Engineering Journal
ISSN: 0970-2555
Volume : 53, Issue 1, No. 1, January : 2024

VI. Performance Comparison

As stated in previous chapters, there are numerous differences in the implementations of Android Native and React Native. Thus, the performance of an identical scenario could vary significantly. Technically, Android Native, due to its bottom-to-top architecture, is more efficient and less resource demanding. However, React Native has a more prosperous community. For example, mature libraries that are heavily involved in our case study, such as Redux, are developed originally for React. In all, it is unlikely to present an objective conclusion regarding performance without discussing case by case. 20 In order to measure the performance, some concepts need to be introduced. Frame per second (FPS), as one of the most straightforward factors to be observed, has been used widely as a standard unit to describe the fluentness of an application. One frame means one static picture of the current window. Any minor changes of the current frame will result to producing another new frame. By recording the number of rendered frames in each second on certain device, one can easily compare the performance of two different applications. The goal of this section is to demonstrate an imperfect comparison of the performance between Android Native and React Native application. Two most-seen scenarios of mobile application are chosen as test cases for a more convincing result. All the testing and comparisons will be held on one Android device (One plus A3003) for monitoring the frame rate.

Scroll

Vertical Scrollable list is a typical view in all mobile applications. It is so common that both native Android and iOS provide a special set of view collections ("Recycler View" and "UICollectionView") instant feedback, animation and pre-render. On March 2017, React Native officially deprecated "List View" and introduced the new "Flat List" as the primary component for constructing scrollable list. The new FlatList is designed to optimize the memory usage as well as providing modern features, such as Pull to Refresh, though simplified API. Native Android, on the other hand, has not iterated its list widget often. "List View" is built as a special child widget of the Custom "Scroll View" for displaying a set of data in linear order. To decrease the impact of other factors, extremely simplified example is written using only framework's component/widget. The example essentially consists a vertical scroll list with 1000 items. Each item contains an image and two lines of text. Below figure reveals some of the Disk I/O

Another critical factor regarding to performance is the speed of input and output (I/O). File exchanges internally within the host device can be measured as the speed of disk IO. It is a common scenario for mobile application to communicate with the host device for storing data. Most of the system file operation are handled by a library called "react native-fs" in React Native. This library grants access to the native filesystem for React Native application. Moreover, it provides simplified API for reading most important figures regarding performance. and writing file asynchronously. A similar plugin can be found in Native Android's community by the name "path provider". Both libraries utilize the native optimization of the host's device filesystem.

IV. CONCLUSION

The purpose of this thesis is to establish a comprehensive study between React Native and Native Android This class represents the basic building block for user interface components. A View occupies a rectangular area on the screen and is responsible for drawing and event handling. View is the base class for widgets, which are used to create interactive UI components (buttons, text fields, etc.). The View Group subclass is the base class for layouts, which are invisible containers that hold other Views (or other View Groups) and define their layout properties. When performance is key of application then we should use Native android, and when time, resources is concern then we should you React Native.

V. REFERENCES

[1] Native android navigation:
[2] https://developer.android.com/guide/navigation /navigation-navigate
[3] Kotlin : https://developer.android.com/guide/navigation /navigation-kotlin-dsl
[4] Virtual DOM in ReactJS
[5] URL:https://hackernoon.com/virtual-dom-in-reactjs -43a3fdb1d130
[6] Official documentation of state management: https://levelup.gitconnected.com/managinghttps: //levelup.gitconnected.com/managing- state-in android-f4d042646521state-in-android f4d042646521
[7] Official React bindings for Redux
[8] URL:https://github.com/reactjs/react-redux