REACT NATIVE: A BRIEF INTRODUCTION TO MODERN CROSS-PLATFORM MOBILE APPLICATION DEVELOPMENT

Tamara RANISAVLJEVIĆ
Darjan KARABAŠEVIĆ
Miodrag BRZAKOVIĆ
Gabrijela POPOVIĆ
Dragiša STANUJKIĆ

Abstract*: React Native is a framework for building native iOS and Android applications using JavaScript. The React library is the key feature. React Native has the same design as React, allowing composing a rich mobile user interface from declarative components, but uses native components instead of web components to render a user interface. React Native has a powerful composition model and recommendation is to use composition instead of inheritance to reuse code between components. To build the content of a component, React Native provides props. Components are in a tree-like structure and data flow through components is unidirectional. In addition to props, components can also have an internal state. Each of the prop and state change triggers a complete re-render of the component. Two main Hooks enable managing a component internal state and component lifecycle. Global state management is provided by Redux and Context API. Context API is a React built-in functionality to share data across the application without having to pass through props. It is like a global value which can be accessed anywhere through the application component tree. Redux is a state management container used for handling all the application related data. All changes to the data happen through reducers and all data is maintained in a global store.*

Keywords*: android, component, iOS, React Native, state management*

INTRODUCTION

React Native is a cross-platform framework used for the development of native mobile applications using JavaScript and React. It is based on an idea to allow developers to write high-quality native applications for iOS and

Android using familiar Web technologies. The beginning of this framework dates back to January of 2015. It was created and published by Meta Platforms, Inc. However, it did not originally support the development of mobile applications for the Android platform, until September 2015 (Dabit, 2019). Today, in addition to iOS and Android platforms, it is possible to develop applications utilising React Native for Windows, MacOS, Web, multiple TV platforms and devices, with the help of third-party libraries (Building for TV Devices; React Native for Windows + macOS).

BACKGROUND

JavaScript is a dynamic scripting language, which was developed to support the browser with the feature of asynchronous communication and for user interaction with the web page components. In other words, it instructs the browser to make changes to page elements after loading a web page. In JavaScript, data changes in memory and it is bound to a view in the user interfaces (UI). That means when data is modified in JavaScript, which is in memory, the data will be changed in the UI as well. In turn, when the data changes in the user interface, more precisely in the Document Object Model (DOM) by clicking a button or any other event, it is also updated in memory, keeping the two in sync. In complex and large applications with multiple views representing data in one of models and, as adding more models and more views, this two-way data binding ends up in an infinite event loop where one view updates a model, which in return updates a view. That is the huge disadvantage of JavaScript and that is why it is not suitable for creating large high-efficiency applications (Paul, Nalwaya, 2016).

React, also known as ReactJS, is an open-source JavaScript library originally created by Jordan Walke. It is backed and maintained by Meta Platforms, Inc (Team). It is used as the View (V) in the Model-View-Controller (MVC). React is designed for solving problems on the Web in a way that allows rendering complex UI with high performance. This library has become extremely popular since its introduction in 2013, with many established companies taking advantage of its quick rendering, maintainability, and declarative UI, among other things. It can be said that React is the backbone of creating powerful single page Web applications. The basic fundamental behind React is the concept of virtual DOM (Aggarwal et al, 2018).

Traditional Web applications are slow because they use the DOM and DOM manipulations are expensive. For the application to be fast and efficient, the usage of the DOM must be as little as possible, because the time required to modify and redraw the DOM can be extremely long. The solution proposed by the React library is to keep a representation of the DOM in memory, called a virtual DOM, and make all changes there (Rawat, Mahajan, 2020). Like the actual DOM, the virtual DOM is a node tree that lists elements and their attributes and content as objects and properties. The virtual DOM renders subtrees of nodes based upon state changes and it can be rendered either at client side or server side and communicate back and forth (Aggarwal et al, 2018). Whenever a request for changing the page content is made, the changes are reflected to the memory residing virtual DOM first. After that a diff() algorithm compares the virtual DOM and the browser DOM, and then the required changes only are reflected to the browser DOM, instead of re-rendering the entire DOM. The goal is that when making changes to memory, React applies the minimum number of changes necessary to align the actual DOM with the virtual DOM. Furthermore, this mechanism provides a gigantic boost to the performance of applications. This characteristic of the virtual DOM is not only important, but the ultimate key feature of React library (Paul, Nalwaya, 2016).

REACT NATIVE ARCHITECTURE

The cross-platform capability of React Native is possible due to its unique architecture. Figure 1. presents several different segments of React Native architecture.

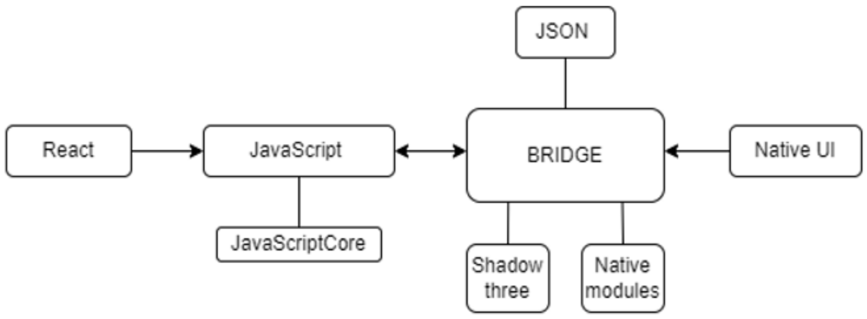

Figure 1. React Native Architecture Scheme
Source: Matijević, 2021

The Native Module system exposes instances of native classes to JavaScript as JavaScript objects, thereby allowing to execute arbitrary native code from within JavaScript. In the case of iOS native modules are written in Objective C or Swift, while in the case of Android modules are written in Java or Kotlin. React Native renders native components by invoking platform-specific APIs. For instance, to render UI components on iOS, React Native uses either Objective C or Swift APIs. As for Android mobile applications, it will be Java or Kotlin. But for writing React Native applications, a developer would hardly ever need to write native code. In fact, a developer doesn't need to know Objective C or Kotlin to create the React Native apps (Gaba, Ramachandran, 2019; Native Modules Intro).

The JavaScript Virtual Machine, also known as JavaScript Bundle, is the engine that runs all JavaScript code written in React Native apps. On both iOS and Android simulators and devices, React Native uses JavaScriptCore. JavaScriptCore is a framework that allows JavaScript code to be run on mobile devices, for instance. On iOS devices, this framework is directly provided by the OS. Android devices don't have JavaScriptCore, so React Native bundles it along with the application itself. This slightly increases the size of the application on Android devices. When the application runs on the device, JavaScriptCore is used to run the JavaScript code. However, in case of debugging the application, the JavaScript code will run inside Chrome. Chrome uses the V8 engine and uses WebSockets to communicate with the source code. It is important to note that the V8 engine and JavaScriptCore are different environments, and errors can occur only when the debugger is connected, but not when the application is running normally on a mobile device (Gaba, Ramachandran, 2019).

The core element of React Native architecture is the Bridge. React Native Bridge is written in C++/Java. The bridge transforms the JavaScript code into source, native code and vice versa. It translates JavaScript into platform-specific components. Simply put, the Bridge gets JavaScript call, then it leverages Objective C, Swift, Kotlin or Java APIs, which allows the original display of the application (Gaba, Ramachandran, 2019).

It is important to emphasise that React Native runs all layouts on separate threads. React Native uses three threads - JavaScript thread, Shadow thread and the Main thread.

The JavaScript thread is where the logic runs and will decide what must be rendered on the screen. This is where the application JavaScript code is executed and where API calls are made. The Shadow thread is a background thread which executes operations from the JavaScript thread. This is where the layout of the app is calculated and passed to the application's interface The Main thread is also known as the UI thread since only this thread can make changes to the UI. The Bridge system uses the React library to display the application on the device. The process does not affect the user experience because these asynchronous calls take place separately from the

Main thread. Therefore, the Main thread is the one React Native application runs on and is used for native Android or iOS user interface rendering (Gaba, Ramachandran, 2019; Matijević, 2021).

The JavaScript thread and the Main thread do not communicate directly but work by sending asynchronous JSON messages. The exchange of asynchronous JSON messages is very efficient. In other words, these two threads depend on the Bridge. JavaScript thread also uses Bridge to transfer data to Shadow thread. The JavaScript thread achieves this by serialising data in JSON format and sending it as a string. This also happens when transferring data from the Shadow to the Main Thread (Matijević, 2021).

REACT NATIVE RE - ARCHITECTURE

Since 2018, Meta Platforms, Inc has been working on a new React Native architecture. It will become open source during this year. The new architecture design differs from the current one in many ways. As shown in Figure 2, it consists of the new Native Turbo Module system and the new Renderer Fabric (Corti, 2022).

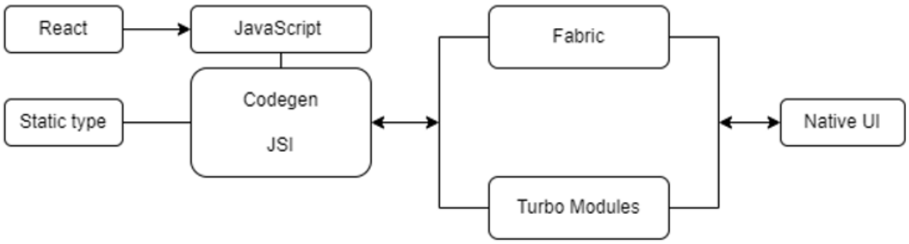

Figure 2. React Native Re-Architecture Scheme
Source: Matijević, 2021

The absence of the Bridge is noticeable in the new architecture. Given the problems caused by the Bridge, such as the dependence of the Main and JavaScript threads on the Bridge, asynchronous operation that does not guarantee data transfer and significant delays in data transfer, it is not surprising that the Bridge has been replaced by a new component called JavaScript Interface (JSI). The main goal of this component is to enable JavaScript and Native parties to be able to communicate without an additional step in the form of a Bridge (Matijević, 2021).

The novelty brought by JSI is the fact that the JavaScript package no longer has to depend on JavaScriptCore. This means that in the future, the JavaScriptCore engine could be replaced by another JavaScript engine, such as the Chrome Engine V8. This further means that application development and debugging mode could be in the same environment. Moreover, by using JSI, JavaScript will be able to keep references to C++ Host objects and call methods on them, which in turn will allow JavaScript and Native components to recognize each other and communicate with each other directly (Matijević, 2021).

Turbo Modules are basically like current Native Modules, but are implemented to behave differently. The best feature of Turbo Modules is that they are lazy loaded, which means that JavaScript code will load each module only when needed and have a direct reference to it. This can significantly improve start-up times for applications with many source modules (Khoroshulia, 2020).

Fabric is a new React Native rendering system, a conceptual evolution of the outdated rendering system. The basic principles are the integration of higher rendering logic in C++, improving interoperability with host platforms and unlocking new features for React Native. One of the new features is that Fabric allows you to create a shadow tree directly in C++ with UI Manager, which improves the response of the user interface by eliminating the need to skip strings. With the new implementation of multi-platform display systems, each platform benefits from performance improvements that are inspired by the limitations of a single platform (Fabric; Matijević, 2021).

Further, instead of communicating with the JavaScript site via Bridge, Fabric uses JSI to expose the user interface functions of the JavaScript site, resulting in direct communication on both sides. Since React Native is single-threaded, which means that when one component is rendered, the others have to wait in line, this control will allow JavaScript to have priority rows for the user interface. This means that it can prioritise time-sensitive user interface tasks and execute them synchronously over all others (Khoroshulia, 2020; Matijević, 2021).

The features of Native modules, that require synchronous data access, cannot be fully exploited in the current React Native architecture. That's why another useful tool has been introduced into the new React Native architecture, and that is Codegen. The Codegen tool will automate compatibility between JavaScript threads and source threads and ensure that they are synchronised. In addition, Codegen will define the interface elements used by Turbo Modules and Fabric. All of this is expected to eliminate the need to duplicate the code and allow data to be sent without uncertainty (Matijević, 2021).

REACT FUNDAMENTALS

React Native brings the power of React to mobile development. It is built on top of React, utilises the React library as dependency and it uses the same declarative approach to constructing user interfaces as React for the Web (React Fundamentals).

In the world of React, a component is the elementary building block of an application and represents a declarative description of a visual feature on a page. The declarative nature of components promotes the predictability of their output. They accept arbitrary entries, called props, and return React elements that describe what should appear on the screen. Conceptually, components are like JavaScript functions. Namely, they serve the same purpose as JavaScript functions, but they work individually to return JSX code as elements for UI (Components and Props). JSX stands for JavaScript XML. It is simply a syntax extension of JavaScript. It allows directly writing elements in React and React Native, within JavaScript code. The components are also considered to be independent bits of code that can be reused. The two types of components are class and functional components (Aggarwal et al, 2018; React Fundamentals).

In addition to solving some of the common problems faced when creating JavaScript applications, React components are modular and emphasise composition over inheritance, which makes code immensely reusable and testable. Strictly speaking, React has a powerful composition model (Composition vs Inheritance). In essence, it means complex or derivative components are built, instead of using the concept of object-oriented inheritance or something akin to object-oriented inheritance, using a composition to build up complexity from simple building blocks. That also means code reuse is primarily achieved through composition rather than inheritance. Composition has other uses besides making increasingly more complex components from smaller, simpler building blocks. Composition can also be used to make derivative components. Additionally, a React component often has rendering logic, markup declaration, and even styles in the same file, which promotes the portability of code and the ability to write shared libraries of components (Masiello, Friedmann, 2017).

State is a way to handle data in React and React Native components. The state contains data specific to the component that may change over time. The state is user-defined, and it should be a plain JavaScript object.

Updating state re-renders the UI of the component and any child component relying on this data as props. Props or properties are how data is passed down through the React or React Native application to child components. Updating props automatically updates any components receiving the same props. Props are similar to HTML attributes. Simply put, props are used for customising React components. Difference between state and props is that state is mutable while props are immutable. Mutable means changeable. On the contrary, if something is immutable, it can never be changed. This means that state can be updated in the future while props cannot be updated (Caspers, 2017; Dabit, 2019).

React lifecycle methods can be understood as a series of events that occur from the emergence of the React component to the end of its existence. There are three main stages in a React component lifecycle. Those are creation or mounting, updating, and deletion or unmounting. React lifecycle methods are available in a React component and are executed at specific points in the component's lifecycle. They control how the component functions and updates. Lifecycle methods differ in class and functional components. Each has its own set of lifecycle methods. Mounting methods are called when an instance of a component is being created and inserted into the DOM. An update can be caused by changes to props or state. These methods are called when a component is being re-rendered due to changed state or props. Unmounting method is called when a component is being removed from the DOM (Caspers, 2017; Dabit, 2019).

REACT NATIVE CORE COMPONENTS

With React Native, Android and iOS views can be invoked with JavaScript using the React component. At runtime, React Native creates the corresponding Android and iOS views for those components. Because React Native components are backed by the same views as Android and iOS, React Native apps look and perform like any other apps. These platform-backed components are called Native Components. React Native comes with a set of essential, ready-to-use Native Components known as Core Components. Core Components are considered as UI elements written within JSX. There are twelve basic React Native elements, and the most commonly used are shown in Table 1. below (Core Components and Native Components).

React Native UI Component	**Android View**	**iOS View**	**Web Analog**	**Description**
< View >	<ViewGroup>	<UIView>	<div> (non-scrolling)	A container that supports layout with flexbox, style, some touch handling, and accessibility controls.
<Text>	<TextView>	<UITextView>	<p>	Displays, styles, and nests strings of text and even handles touch events.
<Image>	<ImageView>	<UIImageView>	<img>	Displays different types of images
<ScrollView>	<ScrollView>	<UIScrollView>	<div>	A generic scrolling container that can contain multiple components and views.
<TextInput>	<editText>	<UITextField>	<input type='text'/>	Allows the user to enter text.

Table 1. Frequently used Core Components and their Native and Web Analogs
Source: Core Components and Native Components

LOCAL STATE MANAGEMENT USING HOOKS

Initially, the state of a component could only be manipulated if it was a class component, using class-specific life cycle methods (State and Lifecycle). That has changed since the release of React 16.8 in October of 2018. Hooks are a new addition in this React version. They let you use state and other React features without writing a class. Hooks are basically functions that allow manipulation of the state and lifecycle methods from functional components. In other words, Hooks bring to functional components things that were once only possible with classes, such as the ability to work with React and React Native local state and effects through useState and useEffect. Since useState and useEffect Hooks are responsible for local state management, these two Hooks are considered as major ones (Introducing Hooks).

The major two Hooks rules that must always be followed are the following:

1. Never call Hooks from inside a loop, condition or nested function. Instead, always use Hooks at the top level of React function, before any early returns. By following this rule, it is ensured that the Hooks are called in the same order each time the component is rendered.

This is what allows React to properly preserve the state of the Hooks between multiple useState and useEffect calls.

2. Don’t call Hooks from regular JavaScript functions. Instead, call Hooks from React function components or from custom Hooks. By following this rule, it is provided that all stateful logic in a component is clearly visible from its source code (Rules of Hooks).

The most important and often used Hook is useState. The purpose of this Hook is to handle state, whenever any of its data changes, React re-renders the UI. React will preserve this state between re-renders. The useState consists of three parts, namely the state variable, the function to set the state, and the initial value and it returns a pair, the current state value and a function that allows updating it (Using the State Hook).

The useEffect Hook is one of the tools used for managing the component's state and deals with the component's lifecycle. It allows implementation of all of the lifecycle Hooks from within a single API function, unlike class components which use three of their lifecycle methods for the same purpose. Just like the name implies, it carries out an effect each time there is a state change and adds the ability to perform side effects from a function component. By default, it runs after the first render and every time the state is updated. React Native performs the clean-up when the component unmounts and this Hook provides a mechanism for clean-ups, in addition to the update mechanism. It is important to point out that this is the optional clean-up mechanism for effects. Every effect may return a function that cleans up after it. This allows keeping the logic for adding and removing subscriptions close to each other (Hooks API Reference; Using the Effect Hook).

GLOBAL STATE MANAGEMENT USING REDUX

In MVC architectures, it is common for data to flow back and forth through the controller component. Data flows into the controller from views as the user interacts with the application, and data flows out of the controller to the view as the underlying data model is updated. On the contrary, React Native is featured with unidirectional data flow between the states and views in an application. This means data can flow in a single direction between the application states and views (Dabit, 2019).

State describes the condition of the application at a specific point in time and the React Native UI is rendered based on that state. Although component states and props can process data in simple React Native applications, using Hooks tools, it is difficult to accurately handle data for more complex systems. It can be said that passing data through the component tree in React is quite complicated. In order to receive data in a low-level component, the data has to be transferred as props through many middle-level components unnecessarily. This process results in writing a bunch of extra code and adding unused properties to middle-level components. To solve this problem, there are many state management libraries (Caspers, 2017).

Among the very popular third-party libraries React Native uses is Redux. Redux is a pattern and implementation library for managing and updating application state, using events called actions. Redux helps manage the so-called global state, a state that is needed across many parts of React Native application (Dabit, 2019; Redux Overview and Concepts). In other words, by using Redux, the state of the application can be stored globally and divided among multiple components. Redux was inspired, for the most part, by Flux. Redux draws on the ideas of Flux and adds in immutability and the principles of functional programming in an attempt to bring sanity to frontend applications that are growing in complexity on a regular basis (Masiello, Friedmann, 2017). It serves as a centralised store for state that needs to be used across the entire application, with rules ensuring that the state can only be updated in a predictable fashion. Redux components are action, reducer and store (Garreau, Faurot, 2018).

An action is a plain object that describes the intention to cause change with a type property. It must have a type property which tells what type of action is being performed. An action object can have other fields with additional information about what happened. By convention, that information is in a field called payload. An action creator is a function that creates and returns an action object. The only way to change the state of an application is to dispatch an action, using reducer function (Garreau, Faurot, 2018; Redux Overview and Concepts).

A reducer is a function that receives the current state and an action object. Thus, actions and states are held together by a reducer. An action is dispatched with an intention to cause change. This change is performed by the reducer. Reducer is the only way to change states in Redux, makes decisions on how to update the state if necessary and returns the new state. Reducers must always follow some specific rules. First, they should only calculate the new state value based on the state and action arguments. Second rule of reducer is that they are not allowed to modify the existing state. Instead, they must make immutable updates, by copying the existing state and making changes to the copied values. Reducer must not have side effects is the last rule (Caspers, 2017; Redux Overview and Concepts).

A store is a state container which holds the application state. Redux can have only a single store in an application. A store is an immutable object tree in Redux. The store is created by passing in a reducer. In order to create a store from reducer, Redux uses the utility createStore. This is a function that takes in a reducer and returns an object with several methods that allow interaction with the store. Since there is only a single store in Redux applications, the createStore function should only ever be called once in an application (Caspers, 2017; Redux Overview and Concepts).

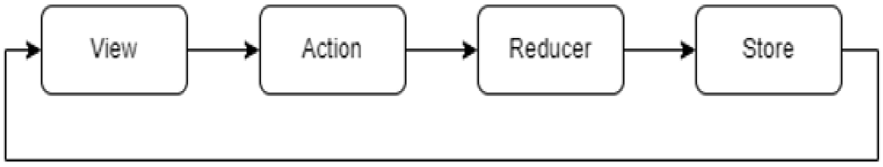

Figure 3. React Native – Redux data flow
Source: Garreau, Faurot, 2018

As presented in Figure 3. Redux unidirectional data flow includes 4 following steps:

1. Components together build up the View. The only purpose of the View is to display the data passed down by the store. By interacting with the application’s View, the user triggers an action.
2. An action is sent or dispatched from the view which are payloads that can be read by reducer. Thus, the reducer function is called with the current state and the dispatched action. It reads the payloads from the actions and then updates the store.
3. The store notifies the View by executing their callback functions.
4. The View retrieves updated state and re-renders again (Garreau, Faurot, 2018).

Since Redux itself is synchronous, a Redux store doesn't know anything about async logic. It only knows how to synchronously dispatch actions, update the state by calling the root reducer function, and notify the UI that something has changed. Reducers must never contain side effects, but in case of async calls it is not possible to avoid them. Redux middlewares were designed to enable writing logic that has side effects. Commonly, middlewares are used to deal with asynchronous actions in applications. They function as a medium to interact with dispatched action before reaching the reducer (Async Logic and Data Fetching).

The first principle of Redux is that all application state is contained within a single store, which is most often a JavaScript object. Redux uses a single store and has reducer functions that are responsible for managing smaller parts of the global state. The second principle of Redux is that the application state is immutable. This means that the object representing the state, at no point, should be modified in any way by any component. Reducer functions are used to create a new state object when an action is dispatched, leaving the old state unmodified. The third and final principle of the Redux framework is that all functions that compute the new state must be pure functions. Pure functions are functions that produce no side-effects and are deterministic–for a given set of inputs, the output will always be the same. In Redux, pure functions are reducer functions (Masiello, Friedmann, 2017).

In general, Redux can integrate with any UI framework, and is most frequently used with React and React Native. React-Redux is the official package that lets both React and React Native components interact with a Redux store by reading pieces of state and dispatching actions to update the store (Caspers, 2017).

GLOBAL STATE MANAGEMENT USING CONTEXT API

In addition to props, there is another way parent elements can pass values down to the children elements in React Native. It is called Context, and it works in much the same way as props, except that it does not have to be explicitly passed down the component tree. Instead, if an element provides its children with Context, any child, no matter how far down the tree, can have access to it. Therefore, Context was introduced to overcome the problem of passing props down component tree, by providing a way to pass data through the component tree without having to pass props down manually at every level (Masiello, Friedmann, 2017).

Unlike Redux, which is known as a global state management technology independent of React, Context has been part of the React library since its 16.3 version, and has become a widely used global state management solution since release of the 16.8 version, thanks to the introduction of another important Hook (Context; Introducing Hooks). The useContext Hook is the one that allows working with React's Context API, which itself is a mechanism to allow us to share data within its component tree without passing through props (Hooks API Reference).

The usage of Context is based on its API. The Context API includes the React.createContext, Context.Provider, Context.Consumer, Class.contextType and Context.displayName. The last two are not frequently used (Context).

React.createContext creates a context object. When React Native renders a component which subscribes to this context object, it will read the current context value from the matching Provider in the component tree. When a component does not have a matching Provider in the component tree, it returns the defaultValue argument and only then is the defaultValue used (Context).

Every Context object has a component which allows consuming components to subscribe to context changes. When a consumer component asks for something, it finds it in the context and provides it to where it is needed. Simply put, it acts as a delivery service. In other words, the Provider component accepts a value prop to be passed to consuming components that are descendants of this Provider. One Provider can be connected to many consumers and can be nested to override values deeper within the component tree. All consumers that are descendants of a Provider will re-render whenever the Provider's value prop changes (Context).

Context.Consumer is a React Native component that subscribes to context changes. It is used to request data through the provider and manipulate the central data store when the Provider allows it and it requires the function as a component. The functional component receives the current context value and then returns a React node. The value argument, which is passed to the function, will be equal to the value prop of the closest Provider for this context in the component tree. If there is no Provider for this context, the value argument will be equal to the defaultValue which was previously passed to createContext function (Context).

In the end, there is another way for the functional component to connect to a Context. Each child functional component can access the Context by calling useContext Hook that requires a parameter to identify which Context to connect to. The solution provided by useContext Hook is a much prettier way to consume context than using Context.Consumer components. The useContext Hook is incredibly helpful when applying it to components consuming multiple contexts (Context; Hooks API Reference).

Overall, there are three mostly used steps to set up a global state management in React Native applications using Context. First step includes createContext. During the following second step Context Provider and global state are created. Final step involves a call to useContext Hook to get state from child components (Context; Hooks API Reference).

CONCLUSION

The best thing about working with React Native is that the application uses standard web technologies like JavaScript, yet is fully native. In other words, React Native applications are blazing fast and smooth and equivalent to any native application built using traditional Android and iOS technologies like Objective-C, Swift or Kotlin. However, React Native does not compromise in terms of performance and overall experience, like popular hybrid frameworks that use web technologies to build iOS and Android apps. This is also the reason why, since its release, React Native has been a widely accepted framework for mobile application development. Without the need to learn a fundamentally different set of technologies for each mobile platform, React Native approach called *Learn once, write anywhere* is completely justified.

REFERENCES

Aggarwal, Sanchit. et al. 2018. ’’Modern Web-Development using ReactJS’’ in International Journal of Recent Research Aspects. Vol. 5(1), pp. 133-137.

’’Async Logic and Data Fetching’’ in Redux Documentation. Redux – Den Abramov. Available: https://redux.js.org/tutorials/essentials/part-5-async-logic. (Accessed: 10.02.2022.)

’’Building For TV Devices’’ in React Native Documentation. Meta Platforms, Inc. Available: https://reactnative.dev/docs/building-for-tv. (Accessed: 01.02.2022.)

Caspers, Matthias. 2017. ’’React and Redux’’ in Rich Internet Applications w/HTML and Javascript. The Carl von Ossietzky University of Oldenburg. pp.11-15

’’Components and Props’’ in React Documentation. Meta Platforms, Inc. Available: https://reactjs.org/docs/components-and-props.html. (Accessed: 01.02.2022.)

’’Composition vs Inheritance’’ in React Documentation. Meta Platforms, Inc. Available: https://reactjs.org/docs/components-and-props.html. (Accessed: 01.02.2022.)

’’Context’’ in React Documentation. Meta Platforms, Inc. Available: https://reactjs.org/docs/context.html. (Accessed: 11.02.2022.)

’’Core Components and Native Components’’ in React Native Documentation. Meta Platforms, Inc. Available: https://reactnative.dev/docs/intro-react-native-components. (Accessed: 03.02.2022.)

Corti, Nicola. 2022. ’’React Native - H2 2021 Recap’’ in React Native Blog. Meta Platforms, Inc. Available: https://reactnative.dev/blog/2022/01/21/react-native-h2-2021-recap. (Accessed: 05.02.2022.)

Dabit, Nader. 2019. ‘’React Native in Action’’, Manning Publications Co. New York. United States of America. pp. 3-27

‘’Fabric’’ in React Native Documentation. Meta Platforms, Inc. Available: https://reactnative.dev/docs/fabric-renderer. (Accessed: 05.02.2022.)

Gaba, Rahul. Ramachandran, Ahul. 2019. ’’React Native Internals’’ in React made Native easy. GitBook. Available: https://www.reactnative.guide/3-react-native-internals/3.1-react-native-internals.html. (Accessed: 04.02.2022.)

Garreau, Marc. Faurot, Will. 2018. ’’Redux in Action’’. Manning Publications Co. New York. United States of America. pp. 10-19.

’’Hooks API Reference’’ in React Documentation. Meta Platforms, Inc. Available: https://reactjs.org/docs/hooks-reference.html. (Accessed: 09.02.2022.)

’’Introducing Hooks’’ in React Documentation. Meta Platforms, Inc. Available: https://reactjs.org/docs/hooks-intro.html. (Accessed: 09.02.2022.)

Khoroshulia, Stanislav. 2020. ’’How React Native App Development Works Under the Hood’’ in MobiDev Blog. MobiDev Corporation. PDF Available: https://mobidev.biz/blog/how-react-native-app-development-works. (Accessed: 05.02.2022.)

Masiello, Eric. Friedmann, Jacob. 2017. ’’Mastering React Native’’. Packt Publishing Ltd. Birmingham. pp. 190-207.

Matijević, Maja. 2021. ’’React Native's upcoming re-architecture’’ in Collective Mind Dev Blog. Collective Mind Development d.o.o. Available: https://collectivemind.dev/blog/react-native-re-architecture. (Accessed: 05.02.2022.)

’’Native Modules Intro’’ in React Native Documentation. Meta Platforms, Inc. Available: https://reactnative.dev/docs/native-modules-intro. (Accessed: 05.02.2022.)

Paul, Akshat. Nalwaya, Abhishek. 2016. ’’React Native for iOS Development’’. Apress Media, LLC. United States of America. pp. 2-6

Rawat, Prateek. Mahajan, Archana. 2020. ’’ReactJS: A Modern Web Development Framework’’ in International Journal of Innovative Science and Research Technology. Vol 5(11). pp. 698-702

’’React Fundamentals’’ in React Native Documentation. Meta Platforms, Inc. Available: https://reactnative.dev/docs/intro-react. (Accessed: 03.02.2022.)

’’React Native for Windows + macOS’’ in React Native for Windows + macOS Documentation, Microsoft Corporation, Available: https://microsoft.github.io/react-native-windows/. (Accessed: 01.02.2022.)

’’Redux Overview and Concepts’’ in Redux Documentation. Redux – Den Abramov. Available: https://redux.js.org/tutorials/essentials/part-1-overview-concepts. (Accessed: 10.02.2022.)

’’Rules of Hooks’’ in React Documentation. Meta Platforms, Inc. Available: https://reactjs.org/docs/hooks-rules.html. (Accessed: 09.02.2022.)

’’State and Lifecycle’’ in React Documentation. Meta Platforms, Inc. Available: https://reactjs.org/docs/state-and-lifecycle.html. (Accessed: 02.02.2022.)

’’Team’’ in React Community. Meta Platforms, Inc. Available: https://reactjs.org/community/team.html. (Accessed: 01.02.2022.)

’’Using the Effect Hook‘’ in React Documentation. Meta Platforms, Inc. Available: https://reactjs.org/docs/hooks-effect.html. (Accessed: 09.02.2022.)

’’Using the State Hook’’ in React Documentation. Meta Platforms, Inc. Available: https://reactjs.org/docs/hooks-state.html. (Accessed: 09.02.2022.)

NOTES ON THE AUTHORS

Tamara RANISAVLJEVIĆ, is a student associate at the Faculty of Applied Management, Economics and Finance, University Business Academy in Novi Sad. E-mail. tamara.ranisavljevic@gmail.com

Darjan KARABAŠEVIĆ, Ph.D. is a Vice-dean for Scientific Research and an Associate Professor of Management and Informatics at the Faculty of Applied

Management, Economics and Finance, University Business Academy in Novi Sad, Serbia. E-mail. darjan.karabasevic@mef.edu.rs.

Miodrag BRZAKOVIĆ, Ph.D., is a Professor and Council President at the Faculty of Applied Management, Economics and Finance, University Business Academy in Novi Sad. E-mail. miodrag.brzakovic@mef.edu.rs.

Gabrijela POPOVIĆ, Ph.D., is an Associate Professor at the Faculty of Applied Management, Economics and Finance, University Business Academy in Novi Sad, Serbia. E-mail: gabrijela.popovic@mef.edu.rs.

Dragiša STANUJKIĆ, Ph.D. is an Associate Professor of Information Technology at the Technical Faculty in Bor, University of Belgrade. E-mail. dstanujkic@tfbor.bg.ac.rs.