**React Native**

1. **Diffrance between useMemo and useCallback**

useMemo and useCallback are both React hooks used for memoization, but they serve different purposes.

**useMemo**

* **Purpose**: Memoizes a value (result of a computation).
* **Usage**: When you want to avoid recalculating an expensive value unless its dependencies change.
* **Returns**: The memoized value.

const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);

**import React, { useMemo } from 'react';**

**function ExpensiveComponent({ a, b }) {**

**const expensiveValue = useMemo(() => {**

**// Expensive calculation here**

**return a + b;**

**}, [a, b]);**

**return <div>{expensiveValue}</div>;**

**}**

**useCallback**

* **Purpose**: Memoizes a function.
* **Usage**: When you want to avoid recreating a function unless its dependencies change. This is particularly useful for passing functions to child components to prevent unnecessary re-renders.
* **Returns**: The memoized function.

**const memoizedCallback = useCallback(() => {**

**doSomething(a, b);**

**}, [a, b]);**

**import React, { useCallback, useState } from 'react';**

**function ParentComponent() {**

**const [count, setCount] = useState(0);**

**const handleClick = useCallback(() => {**

**setCount(count + 1);**

**}, [count]);**

**return <ChildComponent onClick={handleClick} />;**

**}**

**function ChildComponent({ onClick }) {**

**return <button onClick={onClick}>Click me</button>;**

**}**

**Use Cases**:

* useMemo: Use when you have an expensive calculation and want to avoid recalculating it on every render unless its dependencies change.
* useCallback: Use when you want to memoize a function to avoid unnecessary re-renders of child components or avoid recreating the function on every render.

**import React, { useMemo } from 'react';**

**function CalculationComponent({ num }) {**

**const calculatedValue = useMemo(() => {**

**// Imagine this is an expensive calculation**

**return num \* 2;**

**}, [num]);**

**return <div>Calculated Value: {calculatedValue}</div>;**

**}**

**import React, { useState, useCallback } from 'react';**

**function Parent() {**

**const [count, setCount] = useState(0);**

**const increment = useCallback(() => {**

**setCount((prevCount) => prevCount + 1);**

**}, []);**

**return (**

**<div>**

**<p>Count: {count}</p>**

**<Child increment={increment} />**

**</div>**

**);**

**}**

**function Child({ increment }) {**

**return <button onClick={increment}>Increment</button>;**

**}**

1. **HOC:**

"HOC" in React Native stands for Higher-Order Component. It's a pattern used in React (and by extension, React Native) to reuse component logic. An HOC is a function that takes a component and returns a new component with added functionality.

**What is an HOC?**

An HOC is not a feature of the React API itself but a pattern that emerges from React's compositional nature. It's a function that:

1. Takes a component as an argument.
2. Returns a new component with enhanced behavior.

Syntax

**const EnhancedComponent = higherOrderComponent(WrappedComponent);**

**Use Cases**

1. **Code Reusability**: HOCs allow you to reuse logic across multiple components. For example, if multiple components need to fetch data from an API, you can create an HOC to handle the data fetching and pass the data down to each component.
2. **State Abstraction**: HOCs can manage complex state logic and pass down the necessary state and handlers as props to the wrapped component. This keeps the component simpler and focused on rendering UI.
3. **Enhancing Components**: You can use HOCs to add extra props or methods to a component, modify its behavior, or wrap it with additional functionality like logging, error handling, or styling.

**import React from 'react';**

**import { View, Text } from 'react-native';**

**const ItemList = ({ items }) => (**

**<View>**

**{items.map((item, index) => (**

**<Text key={index}>{item}</Text>**

**))}**

**</View>**

**);**

**export default ItemList;**

**import React, { useState, useEffect } from 'react';**

**const withDataFetching = (url) => (WrappedComponent) => {**

**return (props) => {**

**const [data, setData] = useState([]);**

**const [loading, setLoading] = useState(true);**

**useEffect(() => {**

**fetch(url)**

**.then(response => response.json())**

**.then(data => {**

**setData(data);**

**setLoading(false);**

**});**

**}, [url]);**

**if (loading) {**

**return <Text>Loading...</Text>;**

**}**

**return <WrappedComponent {...props} items={data} />;**

**};**

**};**

**const ItemListWithFetching = withDataFetching('https://api.example.com/items')(ItemList);**

**// Usage in your app**

**export default function App() {**

**return (**

**<View>**

**<ItemListWithFetching />**

**</View>**

**);**

**}**

**Benefits of Using HOCs**

* **Separation of Concerns**: HOCs allow you to separate concerns by keeping the data-fetching logic separate from the UI components.
* **Improved Readability**: By abstracting complex logic into HOCs, the readability and maintainability of your components improve.

1. **Setting up environment variables in React Native.**
   1. Using react-native-config

Installation

**npm install @envfile/react-native-config**

Configuration:

**Create a .env file** in the root of your project:

**API\_URL=https://api.example.com**

**API\_KEY=your\_api\_key**

**Modify platform-specific files**:

**iOS**: In your Xcode project, add a new "Run Script Phase" in "Build Phases":

bash

**cp .env.${CONFIGURATION} $SRCROOT/.env**

**Android**: Modify android/app/build.gradle:

project.ext.envConfigFiles = [

debug: ".env",

release: ".env.production",

]

apply from: project(':react-native-config').projectDir.getPath() + "/dotenv.gradle"

**Accessing Environment Variables**:

import Config from 'react-native-config';

// Access variables

const apiUrl = Config.API\_URL;

const apiKey = Config.API\_KEY;

1. **Firebase Crashlytics:**

Firebase offers a package specifically for React Native that integrates Crashlytics, Firebase's crash reporting tool. It provides real-time crash reporting, analytics, and insights into app stability. You can find more details and installation instructions in the

**Step 1: Set Up Firebase Project**

1. **Create a Firebase Project**:
   * Go to the Firebase Console, create a new project or select an existing one.
2. **Add Firebase to Your Project**:
   * Follow the Firebase instructions to add Firebase to your Android and iOS apps. You’ll need to download google-services.json (for Android) and GoogleService-Info.plist (for iOS) and place them in the respective directories (android/app for Android, and the root directory of your iOS project).

**Step 2: Install Firebase SDK and Crashlytics**

1. **Install Firebase and Crashlytics Packages**:
   * Use npm or yarn to install Firebase and Crashlytics packages:

**npm install @react-native-firebase/app @react-native-firebase/Crashlytics**

**Step 3: Initialize Firebase and Crashlytics**

**import { AppRegistry } from 'react-native';**

**import App from './App';**

**import { name as appName } from './app.json';**

**import { getFirebase } from '@react-native-firebase/app';**

**AppRegistry.registerComponent(appName, () => App);**

**// Initialize Firebase**

**getFirebase();**

**Configure Crashlytics**:

* Configure Crashlytics to initialize early in your app, ideally right after initializing Firebase:

**import crashlytics from '@react-native-firebase/crashlytics';**

**crashlytics().setCrashlyticsCollectionEnabled(true);**

**Step 4: Test Crash Reporting**

1. **Generate a Test Crash**:

To ensure Crashlytics is working, you can generate a test crash. For example:

**import crashlytics from '@react-native-firebase/crashlytics';**

**// Force a crash (for testing purposes only!)**

**crashlytics().crash();**

**Monitor Crash Reports**:

* Crash reports will appear in the Firebase Console under **Crashlytics** once they are reported from the app.

1. **Custom Hooks**

**1.**  **Reusability**: They encapsulate logic that can be reused across different components without duplicating code.

 import { useState, useEffect } from 'react';

const useFetchData = (url) => {

const [data, setData] = useState(null);

const [loading, setLoading] = useState(true);

const [error, setError] = useState(null);

useEffect(() => {

const fetchData = async () => {

try {

setLoading(true);

const response = await fetch(url);

const result = await response.json();

setData(result);

} catch (error) {

setError(error);

} finally {

setLoading(false);

}

};

fetchData();

// Cleanup function (optional)

return () => {

// Perform cleanup, e.g., cancel ongoing requests

};

}, [url]); // Dependency array to rerun effect when url changes

return { data, loading, error };

};

export default useFetchData;

**import React from 'react';**

**import { View, Text, ActivityIndicator } from 'react-native';**

**import useFetchData from './useFetchData'; // Assuming the custom hook is defined in useFetchData.js**

**const App = () => {**

**const { data, loading, error } = useFetchData('https://api.example.com/data');**

**if (loading) {**

**return <ActivityIndicator size="large" />;**

**}**

**if (error) {**

**return <Text>Error: {error.message}</Text>;**

**}**

**return (**

**<View>**

**<Text>Data: {JSON.stringify(data)}</Text>**

**</View>**

**);**

**};**

**export default App;**

**Benefits of Custom Hooks**

* **Code Reusability**: Avoid duplicating logic across components by encapsulating it in custom hooks.
* **Simplifies Component Logic**: Keeps components clean and focused on UI concerns by moving stateful logic outside.
* **Composability**: Custom hooks can be composed together, allowing complex behavior to be built from simpler hooks.
* **Improved Testing**: Easier to test hooks independently of components, enhancing testability of application logic.

**7. React Navigation Methods**

**Navigate to a New Screen**

* **navigation.navigate(routeName, params)**: Navigate to a specified screen identified by routeName. You can also pass optional params to the destination screen.

// Example usage:

navigation.navigate('Details', { itemId: 1 });

2. **Go Back to Previous Screen**

* **navigation.goBack()**: Go back to the previous screen in the navigation stack.

3. **Push a New Screen onto the Stack**

* **navigation.push(routeName, params)**: Push a new screen onto the stack, optionally passing params.

// Example usage:

navigation.push('Details', { itemId: 2 });

4. **Replace Current Screen with Another Screen**

* **navigation.replace(routeName, params)**: Replace the current screen with a new one. This is often used for authentication flows or resetting the navigation stack.

// Example usage:

navigation.replace('Home');

5. **Pop from the Stack**

* **navigation.pop()**: Pop the current screen from the navigation stack.

6. **Pop to Specific Screen**

* **navigation.popToTop()**: Pop all screens from the stack except the first one.

=🡺 Diffrance between NPM and NPX

**npm (Node Package Manager)**

* **Purpose:** npm is primarily used to manage packages (libraries or modules) in Node.js. It is a tool for installing, updating, and managing dependencies in your project.
* **Common Commands:**
  + npm install <package>: Installs a package and adds it to your node\_modules folder.
  + npm install -g <package>: Installs a package globally on your system.
  + npm init: Initializes a new Node.js project and creates a package.json file.
  + npm run <script>: Runs a script defined in the package.json file.

**npx (Node Package Executor)**

* **Purpose:** npx is a tool for executing Node packages directly from the command line, without needing to install them globally first. It can run packages that are either locally or globally installed, or even directly from the npm registry.

**Common Uses:**

* **Run Local Packages:** If a package is installed locally in your project, you can run it using npx. This is useful for running tools like create-react-app, eslint, etc., without needing to install them globally.

npx create-react-app my-app

**Run Packages from npm Registry:** You can execute a package from the npm registry without installing it.

npx cowsay "Hello, world!"

**Key Differences**

* **Installation vs. Execution:** npm is focused on package management (installing, updating, removing packages), while npx is focused on package execution (running packages without installing them globally).
* **Global Installation:** With npm, you often need to install packages globally to use them from the command line. With npx, you can run them directly without global installation.
* **Project Context:** npx automatically uses the local version of a package if it's available, ensuring that the correct version for your project is used.

🡺 how to download file and at time of downloading if network is disconnect and after coming network I want to download file from where stop the downloading at time of disconnect network.

🡺Bluetooth sharing

🡺In-app purchase

🡺payment gateway

🡺push notification

🡺associability

🡺 biometric face and keypad