New Architecture of RN

### Key Features in React Native 0.74:

#### Synchronous Layout and Effects:

* Adaptive UI Development: The new synchronous layout feature resolves issues with onLayout events, where state updates often occurred after rendering, leading to visual glitches.
* Smooth User Experience: Now, React Native schedules updates alongside layout measurements, preventing intermediate visual states.

In adaptive UIs, the synchronization between layout and state ensures a smoother transition, preventing users from noticing "jumps" between renders.

#### Automatic Batching:

* Optimized Performance: Automatic batching groups multiple state updates in one event loop, reducing unnecessary re-renders and improving performance.

Examples include:

* Social Media Apps: When liking a post, multiple state updates (like counts, UI changes) happen simultaneously.
* E-Commerce Apps: Updating cart count, price, and button UI during an item addition can be batched into a single render to avoid excess updates.

#### startTransition for Concurrent Features:

* Concurrent Updates: New concurrent features like startTransition allow developers to prioritize UI updates. Marking lower-priority updates gives React the ability to pause and handle higher-priority tasks, improving responsiveness.

#### JavaScript Interface (JSI):

* Direct Native Integration: The previous asynchronous bridge between JavaScript and native code is replaced with JavaScript Interface (JSI), enabling direct method calls between JavaScript and C++.
* Performance Boost: JSI removes the need for serialization, enhancing execution efficiency and reducing overhead.

### Additional Changes:

* Android SDK Bump: The minimum Android SDK version is now Android 6.0.
* App Size Reduction: React Native 0.74 apps occupy around 13% less space, saving around 4MB on devices.
* API Updates: PushNotificationIOS API is deprecated and moved to @react-native-community/push-notification-ios.

#### Key Updates in React Native 0.74:

#### Yoga 3.0:

#### Enhanced Performance: Yoga 3.0 is an optimized layout engine responsible for calculating the size and positioning of elements in React Native apps.

#### Improved Flexibility: It provides better handling of complex layouts, making UI development more efficient.

#### Compatibility: Yoga 3.0 is backward-compatible, ensuring existing layouts don’t break while benefiting from performance improvements.

#### Yoga 3.0 plays a significant role in how layouts are handled under the hood, ensuring that apps can perform better and manage more complex UI structures without noticeable slowdowns.

#### Bridgeless New Architecture:

#### Bridgeless Communication: This update removes the need for the "bridge" that connects JavaScript and native code. In earlier versions of React Native, the communication between these environments relied on the bridge, which could cause performance bottlenecks.

#### Native Modules: By removing the bridge, native modules are now directly exposed to JavaScript, offering better communication and quicker response times. This leads to significant improvements in app performance.

#### Migration Path: Developers can gradually adopt the new architecture without major disruptions, allowing for a smoother transition in existing apps.

#### The bridgeless architecture provides developers with a more streamlined and faster system, reducing delays and increasing the overall performance of React Native apps, particularly when interacting with native code.

#### Build Speed Improvements:

#### Faster Builds: React Native 0.74 introduces build system optimizations that reduce the time required to build projects.

#### Parallel Compilation: It supports parallel compilation, utilizing multi-core processors more efficiently. This reduces developer downtime and accelerates the development cycle, particularly for larger projects.

#### For developers working on bigger apps, these improvements help speed up development, making it easier to iterate on changes quickly.

#### Expanded Hermes Support:

#### Performance Gains: Hermes, a JavaScript engine optimized for React Native, has been further integrated into this release. Hermes focuses on faster startup times, reduced memory usage, and overall better performance for JavaScript execution.

#### Smaller Bundle Sizes: Hermes produces smaller bundle sizes, making it especially useful for apps targeting resource-constrained devices.

#### Broader Ecosystem Support: This release expands the compatibility of Hermes with third-party libraries and tools, ensuring developers can fully leverage the performance advantages of the engine without compatibility issues.

#### Hermes is now the default JavaScript engine for React Native apps, and its performance benefits make it easier to create highly responsive apps.

#### Other Improvements:

#### Debugging Tools: Debugging in React Native has been made easier with enhanced error messages and stack traces, helping developers troubleshoot issues more efficiently.

#### Bug Fixes: This release addresses several bugs, improving stability and reliability.

#### Community Contributions: React Native continues to benefit from its open-source community, with various contributions improving code quality, documentation, and feature sets.

#### Conclusion:

#### React Native 0.74 represents a significant step forward for the framework. The introduction of Yoga 3.0, bridgeless architecture, build speed improvements, and expanded Hermes support are key highlights that make app development faster, more efficient, and scalable. For developers already using React Native, adopting these updates will help them build more performant and responsive apps while minimizing development bottlenecks. The focus on performance optimizations and developer experience ensures that React Native remains a top choice for mobile app development.

MEMORY LEAKS

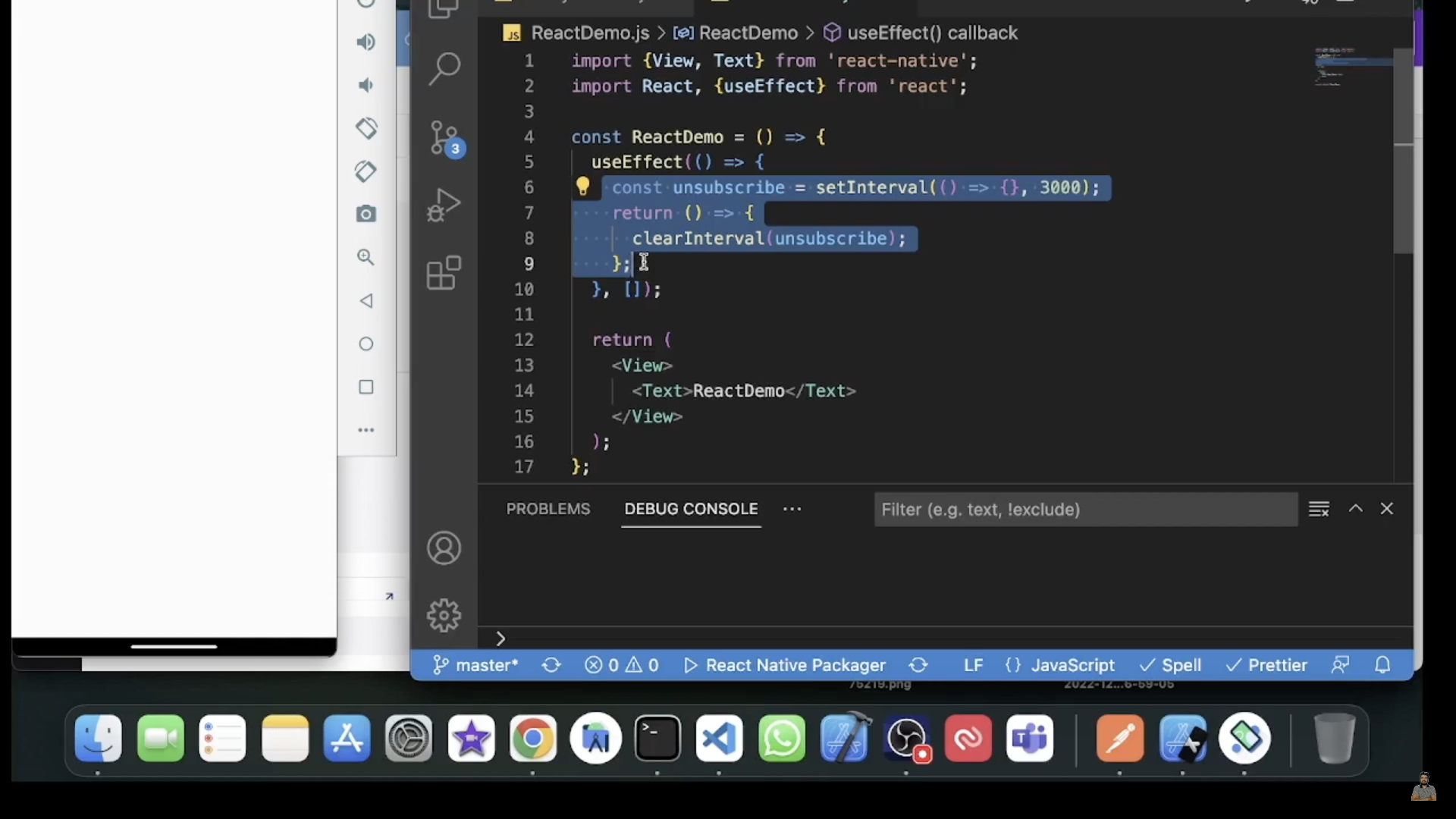
In this section, we will start with the basic definition of memory leaks in React Native applications. Memory Leak is a phenomenon that occurs when an application does not release the memory it has used, resulting in the consumption of system resources for no real benefit. This can happen when variables or resources are retained in memory after they are no longer needed, and are never reclaimed.

Factors Causing Memory Leaks:

This section will dive into the specific causes of memory leaks in React Native apps. We will look at common situations that programmers may encounter, including (DRIW):

* Do not release resources: This includes not unsubscribing to events, disconnecting sockets, or releasing external resources after they are no longer needed.
* Reference error: Objects are not released when there are no longer references to them, leading to a memory leak.
* Infinite loops: Infinite loops in an application can consume a lot of memory and lead to leaks.
* Working with big data: Processing big data can consume a lot of memory and lead to leaks if not managed carefully.

https://medium.com/@hbnguyen/detecting-and-handling-memory-leaks-in-react-native-apps-1453fea08d4d



### Handling Memory Leaks(CUMG)

1. Cleaning up Listeners and Timers:

Ensure you clean up event listeners, subscriptions, or timers by using componentWillUnmount for class components, or useEffect cleanup functions for functional components:  
js  
Copy code  
useEffect(() => {

const timer = setTimeout(() => {

// some code

}, 1000);

return () => clearTimeout(timer); // cleanup

}, []);

1. Using useEffect Correctly:
   * Ensure that useEffect dependencies are correctly set to prevent unnecessary reruns of the effect and component re-renders.
2. Memory Management Libraries:
   * Use libraries like React Native Reanimated that are designed to optimize animations and memory usage in a React Native app.
3. Garbage Collection:
   * React Native relies on JavaScript’s garbage collector to release memory from objects that are no longer referenced. But manual cleanup of resources is essential to ensure that this process runs effectively.

App crashes

App crashes in React Native can occur due to a variety of reasons, ranging from code errors to memory issues and platform-specific bugs. Below is a breakdown of the common causes of crashes and potential solutions to help mitigate them.

### Common Causes of React Native App Crashes:

#### 1. JavaScript Errors:

* Cause: React Native is powered by JavaScript, and unhandled JavaScript errors (such as incorrect syntax, null values, or undefined references) can lead to crashes.
* Solution: Implement proper error handling mechanisms using try-catch blocks and handle unexpected errors gracefully. Using libraries like Sentry or Bugsnag can help capture errors and provide detailed logs for troubleshooting.

#### 2. Memory Leaks:

* Cause: Memory leaks occur when an app retains objects in memory that are no longer needed, leading to excessive memory usage and crashes.
* Solution:
  + Ensure cleanup of event listeners, subscriptions, and timers when components are unmounted.
  + Use useEffect cleanup functions in functional components to remove unnecessary resources.
  + Regularly profile memory usage using tools like Android Studio Profiler or Xcode Instruments.

#### 3. Incorrect Native Module Integration:

* Cause: Native modules allow you to integrate native code with JavaScript, but incorrect implementation or version mismatches can lead to app instability and crashes.
* Solution:
  + Ensure proper integration by following the documentation and use the right version of the native libraries.
  + Test native modules extensively to ensure compatibility with both iOS and Android platforms.

#### 4. Insufficient Error Handling for Asynchronous Code:

* Cause: React Native apps often rely on asynchronous operations like API calls or database interactions. Unhandled promise rejections or errors in asynchronous code can lead to crashes.
* Solution:
  + Use try-catch blocks to handle errors in async functions.
  + Properly handle rejected promises with .catch() and avoid unhandled exceptions in code.

#### 5. Platform-Specific Code:

* Cause: Differences between iOS and Android platforms can cause issues, especially when using platform-specific libraries, APIs, or features.
* Solution:
  + Use Platform APIs (Platform.OS) to write platform-specific logic where necessary.
  + Test apps on both platforms thoroughly, and make sure to handle platform differences like permissions, UI layouts, or

#### 8. Incorrect Permissions or App Configuration:

* Cause: Missing or incorrect permission configurations in iOS and Android manifest files can cause apps to crash when accessing features like the camera, GPS, or storage.
* Solution:
  + Double-check the required permissions in AndroidManifest.xml and Info.plist.
  + Test scenarios like permission denial or revocation to handle these cases gracefully.

### Best Practices to Prevent App Crashes:

1. Use Type Checking and Linting Tools:
   * Tools like TypeScript and ESLint help detect potential issues early during development by enforcing strict type checks and coding standards.
2. Testing and Debugging:
   * Regularly test the app in different scenarios, including edge cases and error conditions.
   * Use React Native Debugger and Flipper to analyze logs and identify potential performance bottlenecks or errors.
3. Crash Reporting:
   * Implement crash reporting services like Sentry, Firebase Crashlytics, or Bugsnag to monitor app performance in production and capture detailed crash reports.

ANIMATION ISSUES ON PREVIOUS RN VERSION FOR IOS

Before React Native version 0.74, there were several known issues and limitations with animations, particularly on iOS. Some of these issues included:

1. Performance Bottlenecks:
   * React Native animations were often handled on the JavaScript thread. If the JS thread became busy with other tasks, such as heavy computation or complex component rendering, animations could stutter or lag. This was more noticeable on lower-end iOS devices.
   * The animation performance was not always smooth, especially for more complex or long-duration animations.
2. Lack of Native Driver Support:
   * Earlier versions of React Native lacked the full use of the native driver for animations. The native driver allows animations to run entirely on the native side (without involving the JS thread), which greatly improves performance and smoothness. Without this support, animations were dependent on the JS thread, which could lead to frame drops.
3. Limited Interactions Between Native and JS Animations:
   * Animations that required interactions between the native code and JavaScript, such as using native navigations or gestures in parallel with animated views, could suffer from inconsistent timing, causing janky animations.
4. Limited API Capabilities:
   * The animated library had fewer features and flexibility. Developers often needed to write more custom code to achieve complex animations or use third-party libraries for advanced animations.
5. React Native Reanimated Issues:
   * Reanimated 1.x, the popular library for building smooth animations, had its own set of limitations on iOS. Reanimated 1.x animations still involved the JS thread for many tasks, making them less reliable in performance-critical applications.
   * The introduction of Reanimated 2.x significantly improved animation performance by allowing animations to be declarative and run entirely on the UI thread, but it required manual integration and wasn’t part of React Native by default.
6. iOS-specific Glitches:
   * On iOS, developers faced some platform-specific quirks, like inconsistent behavior with layout animations or issues with opacity and transform properties, which led to visual glitches or undesired effects.
7. Gesture Handling Delays:
   * In some cases, especially for gesture-based animations, the handling of gesture events could experience delays, as they relied on communication between the native and JS threads. This could result in slower animations or gestures that didn’t feel as responsive as they should be.

With React Native version 0.74 and later, many of these problems have been addressed, thanks to better integration of the native animation driver and improved support for Reanimated and gesture libraries.

Here are examples for each of the mentioned causes of React Native app crashes, with potential solutions and code snippets.

### 1. JavaScript Errors

Cause: Unhandled JavaScript errors like incorrect syntax, null values, or undefined references can lead to crashes.

Solution: Use try-catch blocks and error-handling libraries like Sentry or Bugsnag.

js

Copy code

import \* as Sentry from "@sentry/react-native"; // Example using Sentry

function fetchData() {

try {

// Intentionally referencing an undefined variable to cause an error

let result = someUndefinedVariable;

} catch (error) {

console.error("Error occurred:", error);

Sentry.captureException(error); // Logs the error to Sentry

}

}

### 2. Memory Leaks

Cause: Retaining objects in memory after they're no longer needed can cause excessive memory usage and crashes.

Solution: Clean up event listeners, timers, and subscriptions.

js

Copy code

import { useEffect } from 'react';

function ExampleComponent() {

useEffect(() => {

const interval = setInterval(() => {

console.log('Running...');

}, 1000);

// Cleanup on unmount

return () => {

clearInterval(interval);

console.log('Cleanup done');

};

}, []); // Empty dependency array means this runs only once

}

Memory Profiling: Use Android Studio Profiler or Xcode Instruments to monitor memory usage and detect leaks.

### 3. Incorrect Native Module Integration

Cause: Incorrect implementation of native modules or version mismatches can destabilize the app.

Solution: Properly follow documentation and ensure version compatibility.

js

Copy code

// Example of linking a native module correctly

import { NativeModules } from 'react-native';

const { NativeExample } = NativeModules;

// Use the native module safely

NativeExample.someNativeFunction()

.then(result => console.log(result))

.catch(error => console.error('Error in native module:', error));

Tip: Use react-native link to properly integrate modules and check compatibility with the latest React Native version.

### 4. Insufficient Error Handling for Asynchronous Code

Cause: Unhandled promise rejections or async errors can lead to crashes.

Solution: Handle errors in async functions using try-catch and .catch() for promises.

js

Copy code

async function fetchData() {

try {

const response = await fetch('https://api.example.com/data');

const data = await response.json();

console.log(data);

} catch (error) {

console.error('Async error:', error);

}

}

// Using promises with .catch()

fetch('https://api.example.com/data')

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

.catch(error => console.error('Promise rejection:', error));

### 5. Platform-Specific Code

Cause: Differences between iOS and Android may lead to platform-specific issues, especially with libraries.

Solution: Use Platform.OS to handle differences between platforms.

js

Copy code

import { Platform } from 'react-native';

const styles = {

button: {

backgroundColor: Platform.OS === 'ios' ? 'blue' : 'green',

}

};

const AppButton = () => {

return (

<button style={styles.button}>

{Platform.OS === 'ios' ? 'iOS Button' : 'Android Button'}

</button>

);

};

Tip: Test thoroughly on both platforms and handle permission differences using platform-specific code.

### 6. Incorrect Permissions or App Configuration

Cause: Missing or misconfigured permissions in Android or iOS files can cause crashes.

Solution: Ensure correct permissions in AndroidManifest.xml and Info.plist.

Android Example (AndroidManifest.xml):

xml

Copy code

<uses-permission android:name="android.permission.CAMERA" />

iOS Example (Info.plist):

xml

Copy code

<key>NSCameraUsageDescription</key>

<string>This app requires access to the camera to take photos.</string>

### Best Practices to Prevent App Crashes:

#### 1. Use Type Checking and Linting Tools

Solution: Use TypeScript for strict typing and ESLint for enforcing coding standards.

TypeScript Example:

ts

Copy code

function add(a: number, b: number): number {

return a + b;

}

add(1, 2); // Correct

add('1', 2); // Type error

ESLint Example:

bash

Copy code

npm install eslint

npx eslint . --fix

#### 2. Testing and Debugging

Solution: Use tools like React Native Debugger and Flipper to analyze logs and identify bottlenecks.

bash

Copy code

# Running the React Native debugger

react-native start --reset-cache

#### 3. Crash Reporting

Solution: Implement services like Sentry, Firebase Crashlytics, or Bugsnag to monitor crashes in production.

bash

Copy code

# Sentry Crash Reporting

npm install @sentry/react-native

js

Copy code

Sentry.init({ dsn: 'your-dsn' });

These examples provide practical solutions and preventive measures for handling common crash causes in React Native apps

When making HTTPS requests from the front end (in a web or mobile app), the process involves securely sending data to and receiving data from a server over the Hypertext Transfer Protocol Secure (HTTPS). Here's an overview of how HTTPS works in the context of front-end development:

### Key Steps in HTTPS Communication from the Front End:

1. Establishing a Secure Connection (SSL/TLS Handshake): When the front end makes an HTTPS request, it first establishes a secure connection with the server using SSL/TLS (Secure Sockets Layer/Transport Layer Security). The browser (or mobile app) and the server negotiate encryption protocols and exchange cryptographic keys. This ensures that the data sent over the network is encrypted.
2. Certificate Validation: The browser or app checks the server's SSL/TLS certificate to confirm that it is issued by a trusted certificate authority (CA). This prevents man-in-the-middle attacks, ensuring that communication is happening with the intended server.
3. Request and Response: Once the secure connection is established, the browser (or app) sends an HTTP request (e.g., GET, POST, PUT) encrypted over TLS. The server decrypts the request, processes it, and sends back an encrypted HTTP response. The browser or app decrypts the response for further use.
4. Encryption and Decryption: All data exchanged between the client (front end) and the server is encrypted during transmission. This includes sensitive information like passwords, personal details, or payment data, which is why HTTPS is important for secure communication.

### How HTTPS Provides Security:

1. Encryption: HTTPS encrypts the request and response, preventing anyone from reading the transmitted data.
2. Data Integrity: HTTPS ensures that the data sent and received hasn't been altered in transit.
3. Authentication: SSL/TLS certificates verify the server's identity, preventing impersonation and ensuring secure communication with the intended recipient.

### Front-End HTTPS Call Lifecycle:

1. Initiate: The browser or mobile app initiates an HTTPS request (e.g., fetch or Axios).
2. Handshake: The client and server perform an SSL/TLS handshake to establish a secure channel.
3. Send: The client sends encrypted data (HTTP request) to the server.
4. Receive: The server processes the request and sends back an encrypted response.
5. Decrypt: The client receives and decrypts the server's response.

By using HTTPS, front-end applications ensure secure, encrypted communication between the client and the server, protecting sensitive information from being intercepted.

REDUCE APP SIZE

Reducing the size of a React Native app can improve performance and make it more efficient to download and install. Here are some strategies to minimize the app size:

### 1. Use Hermes (for Android)

Hermes is a JavaScript engine optimized for React Native apps, which can reduce app size, especially for Android apps.

To enable Hermes, update your android/app/build.gradle:  
gradle  
Copy code  
project.ext.react = [

enableHermes: true // Add this line to enable Hermes

]

* Then, rebuild your app.

### 2. Remove Unused Libraries

* Go through your package.json file and remove any unnecessary dependencies.
* Run npm uninstall or yarn remove to ensure unused libraries are not included in the final build.

### 3. Enable ProGuard (for Android)

ProGuard helps by obfuscating code and removing unused code from your project.

In your android/app/build.gradle, enable ProGuard for release builds:  
gradle  
Copy code  
android {

buildTypes {

release {

minifyEnabled true

proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'

}

}

}

### 4. Shrink Resources (for Android)

Android Studio can shrink unused resources to reduce the app size.

Add the following to your android/app/build.gradle file under buildTypes:  
gradle  
Copy code  
release {

shrinkResources true

}

### 5. Optimize Image Assets

* Use optimized image formats (like WebP) instead of heavier ones like PNG or JPEG.
* Compress images before adding them to the app using tools like [TinyPNG](https://tinypng.com/) or [ImageOptim](https://imageoptim.com/).
* Consider loading images dynamically from a server or CDN instead of bundling them with the app.

### 6. Use Code Splitting and Lazy Loading

Code splitting ensures that only necessary code is loaded at runtime, reducing the initial bundle size.

* Implement lazy loading in components using React’s React.lazy() and Suspense to load components only when needed.

### 7. Use Smaller Libraries

* Replace large libraries with lighter alternatives. For example, instead of using a full-fledged navigation library, consider using a smaller one if your navigation needs are simple.
* Explore libraries like lodash-es which can import specific functions rather than the entire library.

### 8. Reduce JavaScript Bundle Size

Use metro.config.js to optimize your JavaScript bundle size:  
js  
Copy code  
module.exports = {

transformer: {

minifierConfig: {

mangle: {

keep\_fnames: true,

},

output: {

ascii\_only: true,

},

sourceMap: false,

compress: {

drop\_console: true, // Remove console logs

},

},

},

};

### 9. Use Native Modules Carefully

* If you are using native modules, ensure that you are not adding unnecessary or unused native code.
* Audit your native dependencies to ensure they are essential for your app.

### 10. Disable Unused Fonts and Icons

If your app includes custom fonts or icons, ensure you're only bundling the ones in use.

* For fonts, selectively load only the required fonts using tools like react-native-fontawesome or react-native-vector-icons in a modular way.

### 11. Optimize Android APK/Bundle

* Use the Android App Bundle (AAB) instead of APK. It enables Google Play to deliver smaller APKs to users based on their device’s configurations.

In android/app/build.gradle:  
gradle  
Copy code  
android {

bundle {

abi {

enableSplit true

}

}

}

This will generate APKs per architecture (arm64, x86, etc.), reducing the overall size.

By applying these techniques, you should be able to significantly reduce the size of your React Native app, making it more efficient and easier to distribute.

Difference between FLatList and ScrollView

**In React Native, both FlatList and ScrollView are used to render scrollable views, but they are suited for different use cases and offer different features and performance considerations. Here's a comparison of FlatList vs. ScrollView:**

### **1. ScrollView**

**The ScrollView component renders all of its child components at once, regardless of how many there are. It's suitable for small lists or layouts where you don't need to worry about performance.**

#### **Key Features of ScrollView:**

* **Renders All Items: It renders all the child components at once, so if you have 1000 items, all of them are rendered immediately, which can cause performance issues with large data sets.**
* **Works for Small Data Sets: Ideal for rendering small lists or layouts where the total number of items is known and limited.**
* **Flexible Layout: Can hold multiple components (views, text, images, etc.) inside it, not just lists.**

#### **ScrollView Example:**

**js**

**Copy code**

**import React from 'react';**

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

**const ExampleScrollView = () => {**

**return (**

**<ScrollView>**

**<View>**

**{Array.from({ length: 100 }).map((\_, index) => (**

**<Text key={index}>Item {index + 1}</Text>**

**))}**

**</View>**

**</ScrollView>**

**);**

**};**

**export default ExampleScrollView;**

#### **When to Use ScrollView:**

* **When you have a limited number of items or components to display.**
* **When all items need to be rendered simultaneously (e.g., for non-list layouts or small datasets).**

### **2. FlatList**

**The FlatList component is optimized for rendering long lists of data. It only renders items that are currently visible on the screen and lazily renders the rest as the user scrolls, improving performance for large data sets.**

#### **Key Features of FlatList:**

* ***Lazy Rendering*: Only renders the items visible on the screen, improving performance with large lists.**
* **Virtualized List: Uses a virtualized list to manage memory and performance, making it more efficient for large data sets.**
* **KeyExtractor: Requires a unique key for each item, often handled via the keyExtractor prop.**
* **Optimized for Long Lists: Suitable for large datasets, where rendering all items at once would be inefficient.**

#### **FlatList Example:**

**js**

**Copy code**

**import React from 'react';**

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

**const ExampleFlatList = () => {**

**const data = Array.from({ length: 1000 }).map((\_, index) => ({**

**key: index.toString(),**

**value: `Item ${index + 1}`,**

**}));**

**return (**

**<FlatList**

**data={data}**

**renderItem={({ item }) => (**

**<View>**

**<Text>{item.value}</Text>**

**</View>**

**)}**

**keyExtractor={(item) => item.key}**

**/>**

**);**

**};**

**export default ExampleFlatList;**

#### **When to Use FlatList:**

* **When rendering large datasets that could cause performance issues if rendered all at once.**
* **When you want to optimize memory usage and performance by rendering only visible items (lazy loading).**

### **Key Differences:**

| **Aspect** | **ScrollView** | **FlatList** |
| --- | --- | --- |
| **Rendering** | **Renders all child components at once.** | **Renders only the items visible on the screen.** |
| **Performance** | **Can cause performance issues with large datasets.** | **Optimized for large datasets, better performance.** |
| **Use Case** | **Suitable for small, fixed-length lists or layouts.** | **Suitable for large or dynamically loaded lists.** |
| **Memory Usage** | **High memory usage with large datasets.** | **Efficient memory usage, only renders what's visible.** |
| **Flexibility** | **Can hold any component (images, views, etc.).** | **Primarily for rendering lists of items.** |
| **Scroll Performance** | **May cause slow scrolling with a large number of items.** | **Smooth scrolling even with a large number of items.** |
| **Lazy Loading** | **Does not support lazy loading of items.** | **Supports lazy loading of items.** |

### **Performance Considerations:**

* **ScrollView is suitable when rendering a small number of items (e.g., a form with several fields, a list of 10-20 items).**
* **FlatList is better when you have a large dataset (e.g., 100+ items) since it doesn’t render everything at once, which improves performance and memory efficiency.**

### **Conclusion:**

**Use ScrollView for small, simple layouts or limited data. For larger, dynamically generated lists, always opt for FlatList to avoid performance issues, as it provides better memory management and optimized rendering.**

**After logging in our app in react native what details are stored in keychain in iphone so that user does not have to login again**

In a React Native app on iOS, the **Keychain** is used to securely store sensitive information so that a user does not have to log in again after successfully authenticating. Typically, the following details are stored in the Keychain:

1. **Access Token**: A token generated by the server upon successful login, usually a JWT (JSON Web Token) or OAuth token. This token is used for authenticating subsequent API requests without needing to re-enter login credentials.
2. **Refresh Token**: If using a token-based authentication system (like OAuth), a refresh token may also be stored. This allows the app to request a new access token when the current one expires without requiring the user to log in again.
3. **User ID or Identifier**: Some apps store a user identifier (like a user ID or email) to easily retrieve the user's account or profile without performing another authentication step.
4. **Session Data** (Optional): Any session-related data that can help in identifying the user across sessions, though sensitive data like passwords should not be stored here.

**Best Practices**:

* **Encryption**: Ensure that tokens and sensitive information are encrypted when stored.
* **Token Expiration**: Implement token expiry and a refresh mechanism to ensure users remain logged in securely.
* **Session Management**: Periodically clean up expired or unused tokens from the Keychain for security.

In React Native, libraries like [react-native-keychain](https://github.com/oblador/react-native-keychain) are commonly used to interact with the Keychain.

Mobile Security -[react-native-keychain](https://github.com/oblador/react-native-keychain)

**Debouncing v/s Throttling**

Debouncing and throttling are both techniques used in programming, particularly in web development, to control how often a function is executed in response to events like scrolling, resizing, or keystrokes. Here's a breakdown of each:

### **Debouncing**

* Definition: Debouncing ensures that a function is only executed after a certain amount of time has passed since the last time it was invoked. If the function is triggered again before the time has elapsed, the timer resets.
* Use Case: Ideal for scenarios where you want to wait until the user has finished an action. Commonly used in:
  + Search input fields (to avoid sending a request on every keystroke).
  + Window resize events (to run a function after the user has stopped resizing).
* Example: If a user types in a search box, the debounce function will only execute the search after they've stopped typing for, say, 300 milliseconds.

### **Throttling**

* Definition: Throttling allows a function to be executed at most once in a specified amount of time, regardless of how many times it is triggered. This means the function will run at regular intervals.
* Use Case: Best for scenarios where you want to limit the rate at which a function is called. Commonly used in:
  + Scroll events (to update the UI at regular intervals as the user scrolls).
  + Resize events (to run a function at fixed intervals while resizing).
* Example: If a user is scrolling down a page, the throttled function might execute every 200 milliseconds, ensuring that it doesn’t overwhelm the system with too many calls.

### Summary

* Debouncing: Waits for a pause in events before executing.
* Throttling: Ensures execution at a fixed rate, regardless of how many events occur.

Choosing between them depends on the specific behavior you want to achieve in your application.

**Axios v/s Fetch**

Axios and Fetch are both popular tools for making HTTP requests in JavaScript, but they have some key differences. Here’s a breakdown:

### **1. Syntax and Usability**

**Axios**: Uses a simpler syntax for making requests. It returns a promise that resolves to the response data directly, making it easier to work with.  
javascript  
Copy code  
axios.get('/api/data')

.then(response => console.log(response.data));

**Fetch**: Has a more verbose syntax and requires additional handling to parse the response. You need to explicitly call .json() to extract the JSON data.  
javascript  
Copy code  
fetch('/api/data')

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

.then(data => console.log(data));

### **2. Response Handling**

* **Axios**: Automatically transforms response data to JSON if the content type is application/json. It also throws an error for HTTP status codes that indicate failure (4xx, 5xx).
* **Fetch**: Does not throw an error for HTTP errors (you need to check response.ok). You must manually check the status and handle errors accordingly.

### **3. Request and Response Interceptors**

* **Axios**: Supports request and response interceptors out of the box, allowing you to modify requests or handle responses globally.
* **Fetch**: Does not have built-in interceptors, so you would need to implement any custom logic manually.

### **4. Cancellation of Requests**

* **Axios**: Supports request cancellation using cancel tokens.
* **Fetch**: Uses the AbortController API to cancel requests, which is a bit more complex to set up.

### **5. Cross-Site Request Forgery (CSRF)**

* **Axios**: Automatically sends cookies with requests, which is useful for CSRF protection if configured properly.
* **Fetch**: Requires manual configuration of the credentials option (e.g., credentials: 'include') to include cookies in cross-origin requests.

### **6. Browser Support**

* **Axios**: Works in all browsers and has no compatibility issues, as it uses XMLHttpRequest under the hood.
* **Fetch**: Supported in most modern browsers but not in Internet Explorer. A polyfill is needed for IE support.

### **7. File Uploads**

* **Axios**: Simplifies file uploads by handling FormData automatically.
* **Fetch**: Requires a bit more setup for file uploads, but it can still handle FormData.

### **Conclusion**

* **Axios** is often preferred for its simplicity, built-in features, and better error handling, making it a good choice for most applications.
* **Fetch** is a native option that is more lightweight but requires more manual handling and configuration.

Choosing between them often comes down to personal preference and project requirements!

<https://www.freedium.cfd/https://javascript.plainenglish.io/building-secure-authentication-in-the-frontend-jwt-cookies-and-local-storage-%EF%B8%8F-6fa34d59509e>

Here is a list of 100 frequently asked React Native interview questions at the SDE-2 level, compiled from various sources:

### **Basics and Concepts (1-25)**

1. What is React Native, and how does it differ from React?
2. Explain the role of JSX in React Native.
3. How do you handle state in React Native?
4. What is the purpose of setState() in React Native?
5. Describe the lifecycle methods in React Native.
6. How do you handle props in React Native?
7. Explain the significance of the render() method.
8. What is Flexbox, and how is it used in React Native?
9. How does React Native handle UI layouts differently than HTML/CSS?
10. Explain how React Native bridges work with native code.
11. What are functional and class components in React Native?
12. How do you manage navigation in React Native?
13. How is a list rendered using FlatList?
14. Explain the concept of higher-order components.
15. What are the advantages of using React Native over native development?
16. How does React Native manage component state?
17. What are controlled and uncontrolled components?
18. How do you implement touch events in React Native?
19. Explain the importance of Redux in React Native.
20. How do you manage side effects in React Native?
21. What is the purpose of the useEffect() hook?
22. Describe error boundaries in React Native.
23. How is performance optimization achieved in React Native?
24. What is the role of the ActivityIndicator component?
25. How do you debug React Native applications?

### **Advanced Questions (26-50)**

1. How do you handle deep linking in React Native?
2. How is navigation state managed across screens?
3. How do you implement animations in React Native?
4. What is the difference between ScrollView and FlatList?
5. How do you optimize a React Native app for performance?
6. Explain the significance of lazy loading.
7. What are pure components, and how do they differ from regular components?
8. How do you manage memory leaks in React Native?
9. How do you handle push notifications?
10. Explain the concept of code splitting.
11. What are the advantages of using hooks in React Native?
12. How does React Native handle threading?
13. How do you manage async operations in React Native?
14. What are the best practices for handling API calls?
15. How do you handle gestures in React Native apps?
16. What is the purpose of the PermissionsAndroid module?
17. How do you handle media assets efficiently?
18. Explain the use of react-native-gesture-handler.
19. How do you ensure accessibility in React Native apps?
20. How do you handle screen orientations and responsive designs?
21. How do you ensure security in React Native apps?
22. What are some performance bottlenecks in React Native apps, and how do you avoid them?
23. Explain how the virtual DOM works in React Native.
24. What is the Context API, and how is it used in React Native?
25. How do you handle updates and versioning in React Native apps?

### **Platform-specific Questions (51-75)**

1. How does React Native handle platform-specific code for Android and iOS?
2. What is the purpose of the Platform module in React Native?
3. How do you integrate native modules in React Native?
4. How do you access native device features in React Native?
5. What is the difference between Android and iOS navigation in React Native?
6. How do you handle different screen sizes in React Native?
7. How do you handle permissions in Android and iOS apps?
8. How do you optimize React Native apps for iOS and Android?
9. What is the role of AppDelegate.m and MainActivity.java in React Native?
10. How do you build and release a React Native app on the Google Play Store?
11. How do you build and release a React Native app on the Apple App Store?
12. How do you manage layouts for iOS notch screens?
13. How do you integrate third-party SDKs in React Native?
14. Explain the use of react-native-device-info.
15. How do you handle keyboard events in React Native?
16. How do you manage app startup times on mobile devices?
17. How do you implement offline mode in React Native?
18. How do you handle platform-specific UI designs?
19. Explain the process of linking native libraries.
20. How do you manage app size optimization in React Native?
21. How do you handle iOS-specific gestures in React Native?
22. How do you implement dark mode in a React Native app?
23. How do you manage background services in React Native?
24. What are the differences in handling notifications in iOS and Android?
25. How do you implement splash screens in React Native?

### **Real-world Scenarios (76-100)**

1. Describe a challenging bug in a React Native project and how you resolved it.
2. How do you manage different environments (development, staging, production) in React Native?
3. How do you implement real-time features like chat in React Native?
4. Explain the process of migrating a React app to React Native.
5. How do you integrate payment gateways in React Native apps?
6. How do you manage push notifications for both iOS and Android?
7. What are the best practices for code structuring in React Native?
8. How do you handle app crashes and debugging in production?
9. How do you implement analytics and user tracking in React Native?
10. How do you manage app versioning and updates in React Native?
11. How do you ensure backward compatibility in React Native apps?
12. How do you handle multilingual support in React Native?
13. How do you manage app state across sessions?
14. How do you implement biometric authentication in React Native?
15. How do you manage background tasks in React Native?
16. How do you handle network timeouts and retries in React Native?
17. How do you improve load times in React Native apps?
18. How do you implement social login in React Native?
19. What are some common pitfalls when working with React Native?
20. How do you manage app performance on low-end devices?
21. How do you implement complex animations in React Native?
22. How do you test React Native apps on different devices?
23. How do you implement a custom splash screen in React Native?
24. How do you manage different themes (light/dark) in React Native?
25. How do you monitor and track app performance in production?

**What are controlled and uncontrolled components?**

* **Controlled**: Managed by React, with state handling through props or state.
* **Uncontrolled**: DOM handles state directly, without React controlling the value (e.g., form inputs with refs).

**What is the role of the ActivityIndicator component?** ActivityIndicator provides a visual indicator for loading states, usually during async operations like fetching data.

**What is the role of AppDelegate.m and MainActivity.java in React Native?**

AppDelegate.m is the entry point for iOS applications, managing the lifecycle events. MainActivity.java plays the same role for Android, where native configurations and lifecycle management are handled.

**How do you build and release a React Native app on the Google Play Store?**

* Generate an Android APK or AAB using Gradle.
* Sign the APK/AAB with a release key.
* Use Play Store Console to upload and distribute the app.
* Follow the guidelines for publishing, including privacy policies and app ratings.

**How do you build and release a React Native app on the Apple App Store?**

* Archive the app using Xcode.
* Create a release build by setting the Release scheme.
* Upload the build using Xcode or Transporter to App Store Connect.
* Ensure proper provisioning profiles, signing certificates, and permissions.

### **Advanced Integration and Tools (76-100)**

1. **How do you handle background tasks and scheduling in React Native?**Background tasks in React Native can be managed using react-native-background-fetch for periodic task scheduling and react-native-background-task for tasks that need to run even when the app is not in the foreground. You can also use headless JS tasks, which allow background services to run without a UI.
2. **What is react-native-code-push and how is it used?**react-native-code-push is a service provided by Microsoft that allows developers to push updates directly to users' apps without requiring a full app store release. It works for over-the-air (OTA) updates by synchronizing JavaScript bundle changes, offering a faster release cycle for minor changes and bug fixes.
3. **How do you handle in-app purchases in React Native?**The react-native-iap library allows integration of in-app purchases (IAPs) for both Android and iOS. You can use it to implement subscriptions, consumable products, and non-consumable items, while handling purchase flows and restoring purchases.
4. **What are the pros and cons of using Expo vs. vanilla React Native?**
   * **Expo**: Easier to set up and use, provides managed services (like asset management, push notifications). However, it has limitations with native code and third-party libraries requiring custom native modules.
   * **Vanilla React Native**: Offers full flexibility with native code integration but requires more setup and maintenance compared to Expo.
5. **How do you debug React Native apps?**Debugging React Native apps can be done using tools like:
   * **React Native Debugger**: A standalone app that integrates Chrome DevTools.
   * **Flipper**: Facebook’s extensible debugging tool for mobile apps.
   * **Remote JS Debugging**: Debugging JavaScript through Chrome DevTools by enabling "Debug JS Remotely."
6. **How does the metro bundler work in React Native?**Metro is the bundler responsible for transforming and packaging JavaScript code in React Native apps. It works by transpiling code with Babel, creating a source map, and then bundling it into a format that can be executed by the JavaScript engine (Hermes or JSC).
7. **How do you implement authentication in React Native apps?**For authentication, you can use:
   * **Firebase Authentication** for integrating email/password, social logins (Google, Facebook).
   * **OAuth2** via react-native-app-auth.
   * Custom JWT-based authentication by making API calls to the backend and storing tokens using AsyncStorage or secure storage.
8. **What is Hermes, and how does it benefit React Native apps?**Hermes is a JavaScript engine optimized for running React Native apps on Android. It improves startup times, reduces memory usage, and provides better overall performance compared to the default JavaScriptCore (JSC).
9. **How do you handle routing and deep linking in React Native?**Deep linking allows opening specific screens in a mobile app from external URLs. React Navigation’s Linking API or react-native-deep-linking can handle routing, while react-navigation provides methods for setting up screen paths.
10. **What are some best practices for state management in React Native?**State management can be handled using:
    * **Context API** for global state without Redux.
    * **Redux** for large-scale apps with complex state flows.
    * **Recoil** and **MobX** are also popular choices for reactive state management. Best practices include avoiding deep prop drilling, organizing state into slices, and using memoization to avoid unnecessary re-renders.
11. **How do you handle form validation in React Native?**Form validation can be handled with libraries like Formik, Yup, and react-hook-form. They provide structured form handling, input validation, and error management, which helps in building robust and user-friendly forms.
12. **How do you handle file uploads in React Native?**File uploads can be handled using the react-native-document-picker or react-native-image-picker to access files from the device. The axios library is typically used to send POST requests with form data, including files, to an API server.
13. **What is the role of babel in React Native?**Babel is a JavaScript compiler that transpiles modern JavaScript (ES6/ES7) into a format compatible with older JavaScript engines (like JSC). In React Native, Babel converts JSX and ESNext syntax into plain JavaScript that the native environment can execute.
14. **How do you measure app performance in React Native?**Performance can be measured using tools like:
    * **React Native Profiler**: For tracking component re-renders and memory usage.
    * **Flipper**: For monitoring performance metrics such as FPS, network activity, and memory.
    * **Firebase Performance Monitoring**: For tracking app load times and network performance.
15. **What is the difference between controlled and uncontrolled components?**
    * **Controlled components**: The component’s state is controlled by React, typically with useState or setState, meaning any changes trigger re-renders.
    * **Uncontrolled components**: The component’s state is managed internally using refs to access DOM elements directly without being controlled by React’s state management.
16. **How do you secure sensitive data in React Native apps?**Use react-native-keychain or react-native-encrypted-storage to securely store sensitive information (e.g., tokens, passwords). Avoid storing sensitive data in AsyncStorage as it is not encrypted. Implement HTTPS and SSL pinning for secure data transmission.
17. **How do you handle app updates and versioning in React Native?**React Native supports over-the-air updates using services like CodePush for delivering JS and asset changes without requiring a full app store release. For native code changes, follow the standard release process via Play Store or App Store.
18. **How do you manage multi-language support (i18n) in React Native?**Multi-language support is implemented using libraries like react-native-i18n or react-i18next. These libraries help in managing translations, formatting dates/currency, and dynamically switching between languages in the app.
19. **What are the lifecycle methods in React Native functional components?**Functional components don’t have lifecycle methods like class components, but hooks like useEffect, useLayoutEffect, useState, and useReducer provide similar lifecycle management for initializing, updating, and cleaning up components.
20. **How do you implement social login in React Native?**Social login can be implemented using the Firebase Authentication SDK or libraries like react-native-fbsdk-next (for Facebook login), react-native-google-signin (for Google login), and react-native-apple-authentication (for Apple login).
21. **How do you manage app permissions in React Native?**Permissions are handled using the PermissionsAndroid module for Android and adding necessary permissions to Info.plist for iOS. Unified libraries like react-native-permissions can handle both iOS and Android permissions in one place.
22. **How do you implement biometric authentication in React Native?**Biometric authentication (fingerprint, FaceID) can be implemented using react-native-touch-id or react-native-keychain. These libraries allow authentication using device-level biometric security measures.
23. **How do you handle asynchronous storage in React Native?**Async storage can be handled using the AsyncStorage API, which provides a simple, unencrypted key-value storage system. For more secure options, libraries like react-native-encrypted-storage can be used.
24. **What are some methods to reduce JavaScript bundle size in React Native?**
    * Use **tree shaking** to eliminate dead code.
    * Split code into smaller bundles using dynamic import().
    * Remove unused dependencies.
    * Use tools like metro-config to customize bundling behavior.
25. **How do you handle memory management in React Native?**React Native manages memory automatically, but developers should avoid memory leaks by cleaning up listeners, preventing long-running timers, and managing large objects in memory. The use of useEffect cleanup functions ensures side effects are managed correctly.
26. **How do you manage layouts for iOS notch screens?**React Native’s SafeAreaView handles safe areas and prevents content from being obscured by notches or status bars on iPhones with a notch (e.g., iPhone X). You can also use third-party libraries like react-native-safe-area-context.
27. **How do you integrate third-party SDKs in React Native?**SDKs can be integrated using native modules. For Android, you modify build.gradle and MainActivity.java, and for iOS, you add the SDK to Podfile and configure it in AppDelegate.m. Some SDKs come with React Native bindings for easier integration.
28. **Explain the use of react-native-device-info.**react-native-device-info is a library that provides information about the device running the app, such as OS version, device name, battery level, screen resolution, and more. This is useful for device-specific behavior or diagnostics.
29. **How do you handle keyboard events in React Native?**React Native provides the Keyboard API to detect and manage keyboard events like showing/hiding the keyboard, and adjusting layouts accordingly. You can also use libraries like react-native-keyboard-aware-scroll-view to automatically adjust UI when the keyboard appears.
30. **How do you manage app startup times on mobile devices?**To reduce app startup times:
    * Optimize bundle size with code-splitting.
    * Minimize the use of synchronous native modules.
    * Lazy-load non-essential features.
    * Use Hermes engine for Android to reduce JS load times.
31. **How do you implement offline mode in React Native?**Offline mode can be handled using:
    * AsyncStorage or libraries like redux-persist to cache data locally.
    * Service workers or caching strategies for offline-first behavior.
    * Libraries like react-native-netinfo to detect network status changes.
32. **How do you handle platform-specific UI designs?**React Native allows platform-specific components and styles using Platform.select() or different style sheets for iOS and Android. You can also use libraries like react-native-paper and react-native-elements for cross-platform design consistency.
33. **Explain the process of linking native libraries.**React Native used to rely on manual linking (via react-native link) but now prefers auto-linking (RN >= 0.60). Auto-linking automatically detects and links native modules when running react-native run-android or run-ios. For manual linking, you'd edit the native Android/iOS project files to integrate third-party libraries.
34. **How do you manage app size optimization in React Native?**To reduce app size:
    * Remove unused assets and libraries.
    * Use Proguard (Android) and Bitcode (iOS).
    * Optimize images and use WebP.
    * Reduce JavaScript bundle size by splitting code and removing dead code.
35. **How do you handle iOS-specific gestures in React Native?**Use react-native-gesture-handler to manage complex gestures like swipe, tap, and drag. For iOS, native gestures (e.g., swipe-to-go-back) are often implemented by default via React Navigation.
36. **How do you implement dark mode in a React Native app?**
    * Use the useColorScheme() hook provided by React Native to detect whether the device is in dark or light mode.
    * Apply conditional styles based on the theme.
    * Libraries like react-native-appearance can help manage light/dark themes.
37. **How do you manage background services in React Native?**Background tasks, such as fetching data or handling notifications, can be handled using:
    * react-native-background-fetch for periodic background jobs.
    * react-native-push-notification for handling background notifications.
38. **What are the differences in handling notifications in iOS and Android?**
    * **iOS**: Uses APNs for push notifications, requiring device token registration. Notification behavior is controlled via Info.plist.
    * **Android**: Uses Firebase Cloud Messaging (FCM) and requires permissions in AndroidManifest.xml. Android has more customizable notification channels.
39. **How do you implement splash screens in React Native?**Splash screens are implemented natively. On Android, it involves modifying styles.xml and launch\_screen.xml, and on iOS, it requires adding the splash image in the LaunchScreen.storyboard. Libraries like react-native-splash-screen can streamline this process.

Barrel Exporting

Usecallback -> Memoize a function

Usememo -> memoize a value

**Flutter** and **React Native** are two popular frameworks for building cross-platform mobile applications. Both allow developers to create apps for iOS and Android with a single codebase, but they differ in their approach, architecture, and ecosystem. Here's a comparison of the two:

### **1. Programming Language**

* **Flutter**: Uses **Dart**. Dart is not as widely known as JavaScript, but it’s easy to learn, especially for developers familiar with object-oriented programming.
* **React Native**: Uses **JavaScript** (and optionally **TypeScript**). JavaScript is one of the most popular languages, which can be an advantage for many developers.

### **2. Performance**

* **Flutter**: Known for near-native performance because it compiles to native ARM code for iOS and Android. It avoids the JavaScript bridge that React Native uses, resulting in faster performance, especially for graphics-heavy apps.
* **React Native**: Uses a JavaScript bridge to communicate with native components, which can sometimes introduce performance issues, especially for complex animations or calculations. However, performance is often good enough for many standard apps.

### **3. User Interface (UI)**

* **Flutter**: Comes with a rich set of customizable, pixel-perfect widgets that are part of the framework, ensuring consistent performance and appearance across platforms. Flutter gives developers more control over the look and feel of the app.
* **React Native**: Relies on native components, which means the app looks and feels more native to the platform but may behave slightly differently on iOS and Android. For customized UI elements, additional libraries are often needed.

### **4. Development Speed**

* **Flutter**: Offers **Hot Reload** similar to React Native, which helps speed up development by instantly applying changes. However, learning Dart and Flutter’s UI components can introduce a learning curve for new developers.
* **React Native**: Also has **Hot Reload** and is favored for fast development, especially for developers already familiar with JavaScript and React. The large ecosystem and number of third-party libraries can further accelerate development.

### **5. Ecosystem and Libraries**

* **Flutter**: While the Flutter ecosystem is growing rapidly, it still lacks the sheer volume of libraries available in the JavaScript ecosystem. However, official Flutter packages are generally high-quality and well-maintained.
* **React Native**: Has access to the massive JavaScript ecosystem, meaning many libraries are available. However, compatibility issues between libraries and React Native can arise, and sometimes native modules need to be written or integrated.

### **6. Community and Support**

* **Flutter**: Backed by Google, Flutter’s community is growing rapidly, and it has strong support from the company. There are many resources, tutorials, and plugins available, but it’s still catching up to React Native in terms of community size.
* **React Native**: Developed by Facebook, it has a larger community and a wealth of resources, tutorials, and third-party libraries. Being in the JavaScript/React ecosystem also adds to its popularity and support base.

### **7. Stability and Maturity**

* **Flutter**: Though relatively newer, it’s considered stable and is used by major companies like Alibaba, Google Ads, and Tencent.
* **React Native**: More mature, having been in the market for longer. It is used by well-known companies like Facebook, Instagram, and Airbnb (before they moved to native).

### **8. Learning Curve**

* **Flutter**: For developers unfamiliar with Dart or Flutter’s widget-based approach, there may be a steeper learning curve.
* **React Native**: Easier for JavaScript and React developers to pick up, thanks to its use of familiar syntax and tools.

### **9. Native Functionality**

* **Flutter**: Offers rich APIs to interact with native code, and platform-specific code can be written when necessary. However, you might need to dig into native code more often when working with less common functionalities.
* **React Native**: Has a wide range of libraries for integrating with native functionalities. Native code is often necessary when dealing with complex or less commonly used native features.

### **10. Use Cases**

* **Flutter**: Ideal for developers who want consistent performance and design across platforms and don’t mind using Dart.
* **React Native**: Suitable for developers who already work with JavaScript/React and want to reuse their existing skills for mobile app development.

### **Conclusion:**

* **Choose Flutter** if you want:
  + High-performance apps, especially with heavy custom UI or animations.
  + A more consistent cross-platform design and performance.
  + To work in the Dart language.
* **Choose React Native** if you:
  + Are familiar with JavaScript and React.
  + Want access to a larger ecosystem of libraries.
  + Need a quicker learning curve with the potential to reuse web development knowledge.

Both frameworks are excellent, and the choice often comes down to specific project needs, team expertise, and performance/UI requirements.