React Tips

8 Hands-on Tips on How To Continuously Optimize Your Published Mobile App

React Native is an excellent framework for creating cross-platform mobile applications. It enables you to produce a top-notch, native-like application experience supported by a single codebase across multiple platforms.

The underlying architecture in React Native consists of a UI main thread where all the animations and transitions happen and a JavaScript thread to run the business logic and process touch events. When the app has performance issues, the thread may become unresponsive, which results in a jittery and clunky UI experience. Although React Native has been built with performance in mind, there are still cases where you might fall into newbie traps when developing React Native applications. This is why we compiled some tips on optimizing your React Native app performance.

#1 Remove console statements before releasing the app

Developers often use console.log for debugging purposes, which can unnecessarily overload the JavaScript thread; hence you should remove it from the production application. There's an easy way to remove all console logs from your application's release bundle by installing the babel plugin:

Execute:

```
npm I babel-plugin-transform-remove-console —save-dev
```

Then modify your .babelrc file under your project root directory:

```
{
 "env": {
   "production": {
     "plugins": ["transform-remove-console"]
   }
 }
}
```

And now all your console statements are removed from the release bundle.

#2 Use FlatList to render large arrays of items

Sometimes, you want to display a large number of items in a grid or list-like view. In such cases, you often need to use ScrollView. If you use ScrollView for a small number of items, that's fine, but as the number of items rendered increases, performance degrades because all items are rendered in advance.

If your FlatList is still rendering slowly, you can go a step further and implement the getItemLayout to optimize the rendering speed of the FlatList by skipping the measurement of the rendered items.

#3 Use nativeDriver with Animated API

React Native provides a unique API for delightful UI animations that look alive to the end user; however, this also has its downsides. The JavaScript thread runs animations by the frame (aiming to run at 60 frames per second), so a complex animation can begin to drop the frames whenever the JavaScript thread is busy with heavy computations, resulting in an unresponsive and heavy UI. To avoid this, you can send the animations to the UI thread by setting useNativeDriver to true on all the animated components.

#4 Cache your app images using local storage

The React Native Image component provides an out-of-the-box option to store images in a local cache, to avoid subsequent requests to the network to fetch the images.

The React Native Image component uses a prop called cache to allow caching of images. Here is an example:

```
<Image
  source={{
    uri: 'https://reactjs.org/logo-og.png',
    cache: 'only-if-cached',
  }}
  style={{ width: 400, height: 400 }}
/>
```

#5 Avoid anonymous functions

Anonymous functions are used a lot in JavaScript, but in React Native, they can cause various performance issues due to how JSX works.

```
<TouchableOpacity
  onPress={() => increment(count + 1)}
  title="Click me!"
/>
```

The code above will recreate the function passed to onPress on each re-render, which causes severe performance degradation.

To avoid this, use named functions:

```
const
handlePress = () => setCount(count + 1)
<TouchableOpacity
  onPress={handlePress}
  title="Click me!"
/>
```

This way, the handle for onPress is only created once.

#6 Use React.memo() to avoid unnecessary re-renders

In version 16.6, React Native introduced a new concept called memoization. Under the hood, the memoization process allows a component that has no change in props to not re-render.

To apply memoization to your components, use React.memo HOC:

```
export default React.memo(AppBar);
```

The application bar will only re-render if the props that are passed to the component change. This concept saves a lot on useless re-renders. Combine React.memo with Redux to avoid local state in your components; all your components can be memoized, significantly improving your app's performance.

#7 Use useMemo and useCallback hooks to avoid unnecessary re-renders

useMemo works similarly to React.memo, but it's a hook that you can directly apply to functions in your components, commonly leveraged for heavy computational tasks. useMemo will cache the function result and only recalculate it if one of their dependencies changes.

```
const
handlePress = useMemo(() => setCount
(count + 1), [count]);
<TouchableOpacity
  onPress={handlePress}
  title="Click me!"
/>
```

In the above example, the handlePress will only change if the count changes. When the count does not change and your component is re-rendered, it won't affect the handlePress. Instead, it will load it from the cache.

useCallback hook works very similarly to the useMemo hook; the only difference is that the useMemo returns the memoized value, while useCallback returns memoized callback.

#8 Rewrite performance-critical parts or heavy computational operations into native libraries and wrap them in a Native Module

A React Native app may occasionally need to access a native platform API that is not by default available in JavaScript, such as the native APIs for using Apple Pay or Google Pay. You can develop high-performance, multi-threaded code for image processing or reuse some existing Objective-C, Swift, Java, or C++ libraries without reimplementing them in JavaScript.

The Native Module system lets JavaScript run any native code from within your JavaScript code by exposing instances of Java/Objective-C/C++/Kotlin/Swift (native) classes as JavaScript objects. Although we don't anticipate this functionality to be included in the standard development process, its existence is crucial. In the absence of a native API that your JavaScript code can use, you would typically write your own custom native module.

React Native provide on their website thorough information on how to get started with Native Modules. You can easily get acquainted with setting up and developing the Native module.

In this scenario, you need Android and iOS development expertise because you will have to build libraries in Swift and Kotlin.

Resolute

sales@resolutesoftware.com

Let's talk about your technology requirements.

Get in touch

USA

MA 01701, Framingham,
945 Concord St,

+1-617 386-9697

Deloitte. Company to watch | Tech Fast 50

Business Professional Services

Clutch ★★★★★

People First Company Award 2020 | 2021 | 2022