Technical Document: React Native Project Structure and Best Practices

Content:

1. What is a Polyfill?

A polyfill is a piece of code that provides functionality not natively supported by a certain environment. Polyfills can be written in two forms:

Direct Import in Main File: Use for polyfills that extend JavaScript or React Native APIs, like react-native-get-random-values, with no special bundling needs.

Configure in metro.config.js: Use for Node.js-dependent packages (e.g., buffer, process) that require custom module resolution or bundling in React Native.

1.2. Why Use a Polyfill in React Native?

In React Native, polyfills are often used to add support for certain web or Node.js features that React Native doesn't include by default. Polyfills ensure that your app can rely on these features regardless of the platform it's running on. You may need a polyfill when:

- You're using libraries that depend on web-specific APIs or Node.js features that aren't built into React Native.

- You want to add features like URL handling, secure random number generation, or buffer management, which aren't natively supported.

To implement a polyfill globally, you can configure it in the "metro.config.js" file.

1.3. The metro.config.js File

The "metro.config.js" file customizes the behavior of Metro, React Native's bundler. It's particularly useful for:

1. Fixing Network Issues: If your app needs a fixed IP address for debugging, such as on a public network, you can set it here.

2. Adding Custom Assets: If your app uses custom fonts, images, or other assets that Metro doesn't support by default, configure their paths in this file.

3. Adding Extra Node Modules: To use specific Node.js modules like “buffer” or “process”, which aren’t included by default in React Native, you can add them in the “metro.config.js” file.

1.4. Implementing Polyfills in “metro.config.js”

1. Import Default Configuration:

const { getDefaultConfig } = require(expo/metro-config): This line imports Expo’s default Metro configuration, which you’ll build upon.

2. Export Asynchronous Configuration:

module.exports = (async () => { ... })(): This async function allows Metro to fetch the default configuration and then modify it. The async setup is useful for any operations that require waiting, like fetching the configuration.

3. Fetch and Customize Default Configuration:

const defaultConfig = await getDefaultConfig(__dirname): This line gets the default Metro configuration for your project, which you can then modify.

4. Add Extra Node Modules:

```
defaultConfig.resolver.extraNodeModules = {
  ...defaultConfig.resolver.extraNodeModules,
  buffer: require.resolve('buffer'),
  process: require.resolve('process'),
};
```

Here, you add polyfills like” buffer” and “process” as extra Node modules. This allows your app to use these features globally, even though they aren’t part of React Native by default.

2. Hooks and Why We Use Them in React Native

Hooks are special functions that let you add features like state and lifecycle methods to functional components in React Native. In essence, hooks make it simpler to organize and manage component behaviors in a clean and reusable way. We need hooks in React Native because they make it easy to manage data and control what happens in the app, all within components.

- useState: This hook lets components hold and update their own data. For example, if you want a button to count how many times it’s clicked, “useState” can remember the count. In other programming languages, you might manage this with variables outside the component or with other complex methods, but “useState” makes it simple and keeps everything inside the component.

- useEffect: This hook helps with tasks that happen around the component, like fetching data or setting up timers. For example, if you want to load data from an API when the component appears on the screen, “useEffect” lets you do that easily. Essentially, it handles events like loading data and listening for changes. This can include setup and cleanup tasks based on when the component appears, updates, or disappears from the screen.

- **useContext:** This hook makes it easy for a component to access data that's shared across multiple components without having to pass it down as props. For instance, if you have a theme setting that all components need, you can use "useContext" to access it directly.

- Essentials:

- Most components will use "useState" and "useEffect" as these are fundamental to managing data and reacting to changes. Additional hooks like "useContext" are used based on the needs of the project, particularly when you need to share data across different parts of the app.

3. Imports: Index vs Components

- Polyfills: Import polyfills (like "react-native-url-polyfill") directly in a component if only that component needs it. If multiple components need the same polyfill and you want to make sure it's always available, import it in the "index" file.

- Global Stuff (like styles or app-level libraries):

- Import these in the "index" file to make them accessible everywhere in the app.

- Component-Specific Imports:

- Only import what the component itself needs, like "useState" or "View", directly in that component file.

-Index: Imports in index file are just for anything that should be accessible everywhere, and keep component imports limited to what's specifically needed for that component.

4. Project Structure: Practical Layout and StyleSheet Usage

In a React Native component, the structure typically follows this order:

4.1. Imports:

Start by importing any necessary libraries and components needed for the component's functionality (like React, hooks, and UI elements).

4.2. Hooks and Functions:

- **Define Hooks Inside the Main Function**: Within the main component function, start by setting up any hooks needed to manage state and handle lifecycle events. This approach keeps the state management and lifecycle logic immediately accessible in the component, providing a clear and logical flow from state setup to usage.

- **Write Helper Functions After Hooks:** Following the hooks, you can define any helper functions or additional logic required for the component. Multiple functions can be written inside the main function, and they will only be accessible from within the main component itself. This encapsulation ensures that all helper functions can access state variables and other hooks directly, maintaining the component's modularity and readability.

4.3. StyleSheet Usage:

- Use "StyleSheet.create" within the component to define styles that are specific to it.

- Place the “StyleSheet” definitions after all functions and logic within the component.

4.4. Exporting the Component:

- Conclude with “export default ComponentName” as the last line. This allows the component to be imported into other components or the “index” file, making it accessible throughout the app.

Note: React Native doesn’t use traditional CSS. Multiple StyleSheet objects can be used per component, but keep styles modular and localized to specific components.