Bridge Document from React to React Native:

Table of Contents:

Summary:

This document provides me with key concepts to know for transitioning from React to React native. This also provides me with the key differences between React and React Native

Core Architecture Differences

Runtime Environment:

	React	**React Native**
JS Engine	V8 Browser Engine	Hermes
Rendering	DOM	Native UI Components
Threading	Single threaded	Multi Threaded
Debugging	Browser	React Native Debugger / Flipper

Why Hermes Engine?

- Small APK Size suitable for Mobile Applications
- Less Memory Usage
- Improved startup time

How Rendering Native UI Components work?

- We write a React Native Code.
- It is then converted to a virtual DOM.
- Then React Native Bridge comes into play to translate the given command into the respective instructions in the native mobile systems.
- The older architecture was Bridge and the newer architecture is Fabric, which is more efficient in translating.
- The component is then rendered in the mobile devices.

Component Translation Guide

Basic Elements:

React	React Native
	<View>
/ <p> / <h>	<Text>
<img>	<Image>
<input>	<TextInput>
<button>	<TouchableOpacity> / <Pressable>
<a>	<TouchableOpacity> + Linking
/	<FlatList> / <SectionList>

Interactive Components:

Web Component	React Native Equivalent	Key Differences
<button onClick={}>	<TouchableOpacity onPress={}>	onPress vs onClick
<input onChange={}>	<TextInput onChangeText={}>	Different event handling
<select>	<Picker> (community)	No native select
<form>	Custom component	No native form wrapper

Styling and Layout:

- React Styling:

```
// React (CSS-in-JS or CSS files)
const styles = {
  container: {
    display: 'flex',
    flexDirection: 'row',
    backgroundColor: '#ffffff',
    padding: '16px',
    borderRadius: '8px'
  }
}
```

```
// React Native (StyleSheet)
const styles = StyleSheet.create({
  container: {
    flexDirection: 'row',        // flex is default
    backgroundColor: '#ffffff',
    padding: 16,                 // no units, defaults to dp
    borderRadius: 8
  }
});
```

- Layout Differences:

Flexbox: Default in React Native, optional in React
Units: No px/em/rem - numbers default to density-independent pixels
Box Model: No margin collapse
Positioning: Limited positioning options

Navigation Systems

Concept	React (Web)	React Native	Explanation
Router	React Router	React Navigation	Both are libraries that handle routing (or "moving between pages/screens")
Routes	<Route path="/users">	<Stack.Screen name="Users">	On the web, routes use URLs. In mobile, screens are defined in a stack (remember deck of cards).
Navigation	navigate('/users')	navigation.navigate('Users')	On the web, you use a URL. On mobile, you use the screen's name from the stack.
Parameters	URL params / query strings	Route params	Web uses URL path or query strings. Mobile passes a JS object with parameters.
History	Browser history	Navigation stack	Web works like a browser's history. Mobile works like a stack: push a new screen, pop to go back.

Navigation in react:

- We use a browser to move to different pages.
- BrowserRoutes is used to define routes.
- Using navigate, we can navigate to the required page.
- In react, we can pass data in the URL as query parameters.

Navigation in react native:

- The pages are stored in navigation stack.
- We use stack.navigator and Use stack.screen to define the screen name and the component for the screen.
- We use navigation.navigate to navigate to different screens.
- In react native, we can pass json object as data while navigating to different screens.

Code example for both

In React:

```
// App.jsx
<BrowserRouter>
  <Routes>
    <Route path="/profile" element={<Profile />} />
  </Routes>
</BrowserRouter>

// Move to the Profile page
navigate('/profile')

// Can Pass data in URL
navigate('/profile?id=123')

// In Profile.jsx
const location = useLocation();
const query = new URLSearchParams(location.search);
const id = query.get('id'); // id is “123”
```

In React Native:

```
/ App.ts (navigator setup)
<Stack.Navigator>
  <Stack.Screen name="Profile" component={ProfileScreen} />
</Stack.Navigator>

// Navigate to the Profile screen with data
navigation.navigate('Profile', { id: 123 });

// In ProfileScreen.tsx
const route = useRoute();
const { id } = route.params; // id is 123
```

State management:

State management is a core concept in both React (Web) and React Native and they both share the same react library under the hood.

Storage solutions:

Web	React Native	Use Case
localStorage	AsyncStorage	Simple key-value
sessionStorage	In-memory state	Session data
IndexedDB	SQLite (via library)	Complex data
Cookies	Secure storage libraries	Sensitive data

API Integration and Networking:

- Working with API is the same in both React and React Native.
- The difference lies in the rendering of fetched data.
- In React, we can use Map to display fetched data.
- In React Native, we can use <FlatList> to render fetched data.
- FlatList is used to make the displayed content scrollable.
- Both inbuilt fetch and third-party axios can be used to work with APIs.

Example:

```
fetch('https://api.example.com/users') .then(response =>
response.json()) .then(data => setUsers(data));
```

Key Differences:

- CORS is not applicable in React Native. It is browser specific.
- File Upload requires different FormData handling.

Development Workflow:

Development Workflow:

Aspect	React	React Native
Dev Server	Webpack Dev Server	Metro
Hot Reload	Fast Refresh	Fast Refresh
Debugging	Browser DevTools	React Native Debugger
Device Testing	Browser	Simulator/Physical device

Tools and Commands:

For React:

npm start
npm run build

For React Native:

npx react-native start
npx react-native run-ios
npx react-native run-android

Deployment Process:

Prerequisites for deploying in IOS App Store:

A Mac (macOS)
Apple Developer Account
Xcode
App icon and splash screen assets ready

Deploying Expo managed workflow:

1. Install Expo CLI
npm install -g expo-cli

2. Build Your App
npx expo install eas-cli
npx eas build:configure
npx eas build -p ios

3. Create App on App Store Connect

4. Submit to App Store
    - After the build completes, Expo gives you a .ipa (iOS binary file)
    - Upload it using: Transporter app or npx eas submit -p ios --latest

5. Prepare App Store Listing

6. Submit for review

Prerequisites for deploying in Play Store:

Google Play Developer Account
App icon, splash screen, screenshots, app description
Keystore (for signing your app)
Android device/emulator to test builds

Deploying Expo managed workflow:

1. Install Expo CLI and EAS
npm install -g expo-cli
npm install -g eas-cli

2. Configure EAS Build
npx eas build:configure

3. Build the Android App
npx eas build -p android

This creates an .apk (for local install) or .aab (required for Play Store).

4. Get the .aab File
Once build completes, download the .aab file from Expo.

5. Create App on Play Console

6. Upload .aab
- Go to Release > Production > Create Release
- Upload the .aab file

7. Submit for review

Performance Optimization:

1. Bundle Size Optimization:

React Web:

We can use code splitting and lazy loading, which can deliver the content only when needed.

React Native:

We cannot use code splitting in React Native rather we can use Ram Bundle which can be used to split the Js code into modules reducing the size of the memory and improving the startup time.

2. Rendering & the Bridge:

React Web:

- Uses Virtual DOM.which calculates minimal real DOM changes and applies minimal real DOM changes.

React Native:

- React Native does not manipulate HTML.
- UI rendering occurs on native threads and is controlled by Javascript via Bridge.
- The bridge can become inefficient when sending too many UI updates at once or sending images frequently.
- To overcome this, minimize communication via the bridge and use Batch updates and avoid frequent re-renders.

3. Images optimization:

React:

Uses compressed formats.
Uses lazy loading.

React-Native:

Use appropriate pixels
Avoid rendering large images or rendering many images at once.
Consider using libraries to cache and improve performance.

4. List Rendering

React:

Libraries like react-window, react-virtualized help render large lists efficiently.

React Native:

- Never use .map() for large lists.
- Use FlatList instead, which is highly optimized with virtualization.

Debug Startegies:

1. React Native Debugger:

A powerful app that combines Chrome devtools, Redux Devtools, and Network Inspector.

2. Flipper:

A debugging platform for mobile apps. Flipper works well with both Android and iOS and can inspect React Native apps using the built-in React DevTools plugin.

3. Test on Physical Devices:

Emulators and simulators often don’t reflect real-world performance. So try to test the application in a real mobile device.

4. In-App Debugging Techniques:

Use console.log and also use conditional rendering to find where the error occurs.

5. Linting & Type Safety:

Use ESLint with React Native plugin to catch common bugs and also use TypeScript for static typing.