**React JS Guidelines**

# PREREQUISITES

## Javascript

## **Basic HTML and CSS**

## ES6 Features

## Node and Npm

# 

# OVERVIEW

React is a Javascript library. It is not a framework but an open source project created by Facebook. We can say react to a View layer of MVC pattern.

# SETUP REACT APP

To set up the react app, run the following code in your terminal:

**npx create-react-app react-tutorial**

Once the installation finishes, move to newly created directory and perform:

**cd react-tutorial && npm start**

A window will launch on localhost:3000

# CODING STANDARDS

## BASIC RULE

1. React is a component based library; so the best practice to create one component per file.
2. Use of JSX syntax and only use of React.createElement at the time of initializing app
3. Prefer stateful components when to manage data else for display make functional components.

## 

## NAMING

1. Use .jsx for extension..
2. Use of **PascalCase** for file naming eg: LoginComponent.jsx
3. Use of **CamelCasing** for the instance. Eg:

Import LoginComponent from “./LoginComponent”

const loginComponent = <LoginComponent />

1. The name of the file should be the name of the component itself.
2. Use of variant names instead of DOM component props as propname
3. Use double quotes for jsx attributes. Eg:

<LoginComponent value=”username” />

1. Use **camelCasing** for props name. Eg:

<LoginComponent userName=”username” />

## FUNDAMENTALS

1. Always use ref callbacks Eg:

<LoginComponent

ref={(ref) => { this.myRef = ref; }}

/>

1. Use self close tags that have no child components I component has multiline properties close it on the new line.

<LoginComponent

user=”ABC”

password=”\*\*\*”

/>

1. Use arrow functions for the method invocation.

getScreenInfo = () => {

const dim = Dimensions.get("window");

return dim;

};

1. Binding of function on render calls new function on every render.

getShareDetails={this.getShareDetails.bind(this)}

## ORDERING OF COMPONENT

1. optional static methods
2. constructor
3. getChildContext
4. Any Lifecycle methods
5. Any class level methods
6. Render
7. If redux store use then connect redux store

## ALIAS

1. Aliases is one of the best practice as it helps to make code cleaner
2. All the imports are managed and made easy to maintain

module.exports = {

presets: ['module:metro-react-native-babel-preset', 'module:react-native-dotenv'],

plugins: [

['module-resolver', {

root: './src',

alias: {

Actions: './src/actions',

Utils: './src/utils',

DataStore: './src/data\_store',

UI: './src/ui',

Sagas: './src/sagas',

Routing: './src/routing',

Store: './src/store',

Reducers: './src/reducers',

Constants: './src/constants',

},

}],

],

};

# GUIDELINES FOLLOWED

## DESIGN PATTERN

1. We need a design pattern for high level understanding of the code.
2. It helps to apply proven solutions for common problems.
3. We prefer functional programming in the application as it helps for immutable data structures
4. Here, Immutability doesn’t mean forbidding change but how to handle change
5. We can use functions for composition and as parameters in order to reuse code.
6. The design pattern for our applications were, **DATA DOWN ACTIONS UP**
7. In the above pattern and all design patterns, functional programming plays an important role.

## FOLDER STRUCTURE

1. **package.json:** This File has the list of node dependencies which are needed.
2. **public/index.html:** When the application starts this is the first page that is loaded. This will be the only html file in the entire application since React is generally Written using **JSX** which I will cover later. Also, this file has a line of code **<div id=”root”></div>**. This line is very significant since all the application components are loade**d i**nto this div.
3. **src/index.js**: This is the javascript file corresponding to index.html. This file has the following line of code which is very significant.

**ReactDOM.render(<App />, document.getElementById(‘root’));**

1. The above line of code is telling that **App** Component has to be loaded into an html element with id **root**. This is nothing but the **div element** present in **index.html.**
2. **src/index.css**: The CSS file corresponding to index.js.
3. **src/App.js** : This is the file for **App** Component. **App** Component is the main component in React which acts as a container for all other components.
4. **src/App.css** : This is the CSS file corresponding to **App** Component
5. **src/actions :** This directory contains all the action files required in redux operations.
6. **src/assets :** This directory contains assets require for website like fonts, images etc
7. **src/components :** This directory contains all the components for application. The components may have its own directory consisting of **functions** directory which contains javascript functions required by the component and **stylesheet** directory for storing the stylesheet js / css files. Components directory may also contain below five directories.
   1. AggregateComponents
   2. EnclosingComponents
   3. CommonComponents
   4. GlobalDefinition
   5. UtilityComponents
8. **src/data :** This directory contains data files.
9. **src/reducers :** This directory contains reducer files for redux.
10. **src/store :** This directory contains redux store.js file.
11. **src/stylesheet :** This directory contains stylesheet files for the application.

## REUSABLE CODE

1. Most of the developers prefer duplication of code or components for time being; but the best practice is to have single code run anywhere.
2. Avoid Duplication and create more reusable components and functions.
3. The More reusable code and components helps app to be stable

## CREATING REUSABLE COMPONENT

If we want to make the above component reusable, we need to make it more generic. If a component contains any complex logic inside it, not only does it become difficult to reuse, it also becomes less maintainable.

These are some of the points related to best practices for reusable components in React-based projects.

1. Components should be small and easy to read.
2. Use the prop-types package to structure and validate the props of your components.
3. For styled-components, do not include any functional details, instead pass them as props.
4. For container components, do not include UI details, instead, let the children styled-components take care of them
5. Prefer functional components vs class-based components.
6. Prefer CSS in JS strategy for styled-components for better maintainability.
7. UI variations should be coming in as configurable props whereas data should be coming up as part of a global state like Redux or React Context.
8. Don’t repeat sections wherever it can be avoided, instead use arrays to store those section details and loop over them.
9. For components, add comments only wherever absolutely necessary (to indicate a hack or workaround for a bug).
10. For library utilities (pure functions), write JS-Doc comments.
11. Avoid using anonymous components wherever possible.
12. All files for a component should be part of the same folder.
13. Always use eslint and prettier for keeping your code clean and formatted.
14. Use ES6 features.

## DESTRUCTURING

1. Destructuring objects, array and props; helps to have proper data.
2. It is one of the features of ES6 for fast processing
3. It keeps code clean and also maintainable
4. Destructuring in JavaScript lets you support different variations of parameters.

<LogoTitle {...props} title={"CHECK"} />

## 

## 

## SIMPLE COMPONENTS

1. Don't make components complicated and tight coupled to multiple things.
2. Have separate functions for different functionality.
3. Reusability of code should happen in terms of data maintenance, similar functionality and display purpose.

## CODE FORMATTING

1. Use code formatting plugins such as prettier for maintaining code format
2. Clean and formatted code makes developer to code properly
3. It avoids duplication and complication in code.

## API INTEGRATION

1. For integration of the app and api there are a variety of libraries available.
2. The axios is the library used and recommended.
3. The axios supports all http methods
4. It also includes promises for multiple calls parallelly.
5. Implementation:
   1. Create a file named GlobalUrlCall.js including all axios configuration
   2. Import the method in the file wherever service call required
   3. We can imposed callbacks or promises for the method
6. We can integrate with any 3rd party api having proper url using axios

axios({

method: "get",

url: url,

headers,

}).then((response) => {

callback(response, data);

}).catch((error) => {

callbackError(

{ data: { networkError: "Network Call Failed !" }, status: 400 }

);});

## 

## EXCEPTION HANDLING

1. The exception handling is the key concept in development.
2. In the app, we handled the exceptions with try catch technique.
3. For server call, when data is not return or service call fails; we handle empty data or show user with warning or toast messages
4. Implementation:
   1. Create a function with axios.
   2. Handle catch with callback or messages

axios({

method: "get",

url: url,

headers,

}).then((response) => {

callback(response, data);

}).catch((error) => {

callbackError(

{ data: { networkError: "Network Call Failed !" }, status: 400 }

);

});

## 

1. Following http error codes are handled:
   1. 422: Unprocessable Entity
   2. 401: Unauthorized
   3. 500: Internal Server Error
   4. 503: Service Unavailable
2. Using react Error Boundaries.
   1. Error boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed. Error boundaries catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.
   2. A class component becomes an error boundary if it defines either (or both) of the lifecycle methods static getDerivedStateFromError() or componentDidCatch(). Use static getDerivedStateFromError() to render a fallback UI after an error has been thrown. Use componentDidCatch() to log error information.
   3. Error boundaries work like a JavaScript catch {} block, but for components. Only class components can be error boundaries.

## SPEED OPTIMIZATION

1. The speed optimization is done using Pure components.
2. We used a technique known as “windowing” for handling thousands of data.
3. Redux and in app db can be used for the handling of data which makes the app faster.

## LOCAL STORAGE

1. Local storage is used to store data in browsers.

localStorage.setItem('favoriteflavor','vanilla');

1. Local storage objects can be modified easily.
2. Local storage makes apps faster.
3. Good About Local storage
   1. It is a pure javascript concept.
   2. Web Server creates a cookie but local storage doesn’t need a webserver
   3. Another neat thing about local storage is that it doesn't have as many size constraints as cookies.
   4. Local storage provides at least 5MB of data storage across all major web browsers.
4. Annoying about Local storage
   1. It can only store string data.
   2. This makes it pretty useless for storing data that's even slightly more complex than a simple string.
   3. It is synchronous.
   4. It will slow down your app's runtime.

## HANDLING BROWSERS RELOAD BUTTON

To detect the browser’s reload button click, performance.navigation.type is checked. Below function can be used to detect the reload button click and to perform the operations. This function is called in the constructor of the App component.

reloadListener() {

if (window.performance) {

if (performance.navigation.type === 1) {

//place code here

}

}

}

Another way to detect the browsers reload is to save some value in redux (or whatever you use) store to indicate state after reload and on first route change update it to indicate that route was changed not because of refresh.

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# GUIDELINES FOR USING REDUX

## File structure for redux

The following directories are used to save the redux files in src directory.

* actions: This directory is used to save the actions files.
* reducers: This directory is used to save the reducers files.
* store: This directory is used to save the store file

Naming conventions for redux files and functions are the same as conventions used in coding standards.

The actions folder contains all the Redux action creators for the project. Actions are payloads of information that send data from your application to your store. They are the only source of information for the store.

Action example: below function is used to save the user information in redux store.

export function setAuthParameter(userid, logFlag, defLandingPage, dbName, token,tenantId) {

return {

type: SET\_AUTH\_PARAM,

payload: {

userId: userid,

loggedIn: logFlag,

defaultLandingPage: defLandingPage,

databaseName: dbName,

accessToken: token,

tenantId: tenantId

}

}

}

The reducers folder contains all the Redux reducers for the project, each reducer updates a different part of the application state in response to dispatched redux actions.

reducer example:

export const redirectReducers = (state = INITIAL\_STATE, action) => {

const { type, payload } = action;

switch (type) {

case SET\_AUTH\_PARAM:

return (

{ ...state, ...payload }

)

default:

return state;

}

}

A store is an object that holds the application's state tree. There should only be a single store in a Redux app, as the composition happens on the reducer level. The redux store helper calls createStore() to create the centralized redux state store for the entire react application.

# 

# 

# 

# 

# 

# 

# 

# 

# 

# The Micro-Frontend architecture

Micro-frontend architecture is a design approach in which a front-end app is decomposed into individual, semi-independent “microapps” working loosely together. The micro-frontend concept is vaguely inspired by, and named after, microservices.

This architecture is used to scale the large application into small front-end applications so that many teams can work simultaneously on a large and complex product

The application can be divided into two parts.

* A container: This application is used to render the other micro-frontend apps.
* A micro-frontend: The application which does a specific task only. Eg : a cart feature in e-commerce websites.

There are many ways to implement the micro-frontend architecture.

* Build-time integration
* Run-time integration via iframes
* Run-time integration via JavaScript
* Run-time integration via Web Components

The container application and micro-frontend application are deployed separately. When the application is started, the container application fetches the requested micro-frontend application and renders it.

In short, micro frontends are all about slicing up big things into smaller, more manageable pieces, and then being explicit about the dependencies between them.

### Benefits

* Incremental upgrades
* Simple, decoupled codebases
* Independent deployment
* Autonomous teams

# 

# 

# 

# Using Web-Sockets for push notifications

The web-sockets are used to display the real time messages (messages from server / IOT devices) in the application. For this socket.io library is used. The application receives the message, the message gets added to the message list redux store of application.

When the message is added to redux store, the component in which the related reducer is accessed gets updated. The message data is accessed in the list component which renders the message list.

Steps to use sockets in application.

In the componentDidMount() lifecycle method of the component used to fetch the messages, we create the instance of socket using socketIOClient() function.

The socketIOClient() function takes two arguments. First socketUrl and second the socketOptions. The socketUrl is the server url on which the web-socket server is running. The socket options are used to manage the interactions between client and server.

Some of the events emitted by socket are as follows. These events are used to handle the socket connections.

* connect: Fired upon connection (including a successful reconnection)
* reconnect\_attempt: Fired upon an attempt to reconnect
* connect\_error: Fired upon a connection error
* connect\_timeout: Fired upon a connection timeout
* error: Fired upon an error

The custom events can be used to emit the message or send the data to the client by server.

In the componentWillUnmount method, the connection with the server is disconnected using the socket.disconnect() method.

# SEO or digital **marketing** optimisation

* Meta tags, meta description, keywords added for SEO ranking page content and pages
* React web applications are fast to load and have criss browser support
* Fast page speed load helps in SEO ranking of web app / website
* React web apps are mobile responsive and work seamlessly across multiple viewports
* React web apps allow inclusion of multiple image formats such as webp & svg which improve image optimization and in turn page speed of the web app
* React web app supports minified css and js files for SEO performance