React Coding Best Practices

1. Coding Standards and Best Practices

Think of coding standards as a set of rules, techniques, and best practices to create cleaner, more readable, more efficient code with minimal errors. They offer a uniform format by which software engineers can use to build sophisticated and highly functional code.

2. Need for Implementing Coding Standards

Writing code to create apps isn't just enough; the code should be as clean and maintainable as your app. Before jumping into the good practices to follow, it's essential to realize the reasons behind the same. It becomes crucial for you to understand coding conventions because of the following reasons:

1. Offers *uniformity* to the code created by different engineers.
2. Enables the creation of *reusable* code.
3. Makes it easier to *detect errors*.
4. Make code *Reliable*, *Scalable*, and *Maintainable*.
5. Improves the *readability* that lets developers easily understand the codebase
6. Boost programmer efficiency and generates *faster results*

3. Achieving Coding Standards in React

Below explained are few of the guidelines and tips in achieving the best coding standards in React.

3.1 Create a Good Folder Structure

Organizing your files and folders inside your React application is mandatory for maintainability and scalability.

Having this in mind when planning or starting your application can make a huge difference on the long run. Consider below folder structure as an example:

```
└── /src
  ├── /assets - Contains static assets such as  images, svgs, company logo etc.
  ├── /components - reusable components like navigation bar, buttons, forms
  ├── /services - JavaSript modules
  ├── /store - redux store
  ├── /utils - utilities, helpers, constants.
  ├── /views/pages - majority of the app pages would be here
  ├── index.js
  └── App.js
```

3.2 Keep Correct Order of Import Modules

Sometimes need to use many different modules and our component imports look a little bit messy.

```
import { Auth } from 'aws-amplify';
import React from 'react';
import SidebarNavigation from './components/SidebarNavigation';
import { EuiPage, EuiPageBody } from '@elastic/eui';
import { keyCodes } from '@elastic/eui/lib/services';
import './index.css'
import HeaderNavigation from './components/HeaderNavigation';
import Routes from './Routes';
```

There are many different views on the ideal order of imported modules. We can keep the following order while importing modules:

1. Standard modules

2. Third-party modules
3. Your code imports (components, etc.)
4. Imports specific to the module (e.g. CSS, PNG, etc.)
5. Code only used for tests

After this quick refactor our imports start to look much cleaner.

```
import React from 'react';
import { Auth } from 'aws-amplify';
import { EuiPage, EuiPageBody } from '@elastic/eui';
import { keyCodes } from '@elastic/eui/lib/services';
import HeaderNavigation from './components/HeaderNavigation';
import SidebarNavigation from './components/SidebarNavigation';
import Routes from './Routes';
import './index.css'
```

3.3 Destructure *props*

Destructuring objects (especially props) in JS can substantially reduce repetition in your code. Without destructuring props, our code will look like this:

```
import React from 'react';
const CoffeeCard = props => {
  return (
    <div>
      <h1>{props.coffee.name}</h1>
      <p>Price: {props.coffee.price}$</p>
      <p>Size: {props.coffee.size} oz</p>
    </div>
  );
```

```
};

export default CoffeeCard;
```

As you see we have to repeat ‘props.coffee’ part every time we want to get a prop. Fortunately, there is another cleaner way to do it.

```
import React from 'react';

const CoffeeCard = props => {

  const { name, price, size } = props.coffee;

  return (

    <div>

      <h1>{name}</h1>

      <p>Price: {price}$</p>

      <p>Size: {size} oz</p>

    </div>

  );

};

export default CoffeeCard;
```

3.4 Use Fragments

In our components, we often return multiple elements. A single React component can’t return multiple children, so we usually wrap them in a ‘div’. Sometimes that solution can be really problematic. Let’s take a look at this example:

The Columns component contains some ‘td’ elements. As we can’t return multiple values, we need to wrap these elements in a ‘div’.

```
import React from 'react';

const Columns = () => {

  return (

    <div>

      <td>Hello</td>
```

```
      <td>World</td>
    </div>
  );
};
export default Columns;
```

As a result, we get an error, because we're not allowed to put 'div' in 'tr' tag. The solution is to use 'Fragment' tag - just as below:

```
import React, { Fragment } from 'react';
const Columns = () => {
  return (
    <Fragment>
      <td>Hello</td>
      <td>World</td>
    </Fragment>
  );
};
export default Columns;
```

We can treat 'Fragment' as an invisible 'div'. It wraps elements in the child component, brings them to parent and disappears.

3.5 Use Presentational and Container Components

It's a good idea to split components of your app into Presentational (dumb) and Container (smart) components

3.5.1 Presentational Components

1. Focus mainly on the UI. They are responsible for how components look like.
2. All data is provided by props. Dumb components shouldn't make API calls. That's the job for smart components.

3. They don’t require app dependencies other than UI packages.
4. They may include state, but only for manipulating UI itself - they shouldn’t store application data.
5. Examples of dumb components: *loaders, modals, buttons, inputs*.

3.5.2 Container Components

1. They don’t focus on styling and typically don’t include any styling.
2. They are used to manipulate data. They can fetch, capture changes, and pass down application data.
3. They are responsible for managing state, re-rendering components, etc.
4. They may require app dependencies, call Redux, Lifecycle methods, APIs, Libraries, etc.

Benefits of using Presentational & Container Components

1. Better readability
2. Better reusability
3. Easier to test

3.6 Use Styled-components

Styling React components have always been quite problematic. Finding misspelled class names, maintaining large CSS files, handling compatibility issues sometimes can be painful.

Styled Components allow you to write CSS in JavaScript using tagged template literals. To start with Styled Components you’ll need to add the styled-components library to your ReactJS project using NPM:

```
npm i styled-components
```

Find the below code snippet using styled components:

```
import React from 'react';

import styled from 'styled-components';

const Grid = styled.div`

  display: flex;

`;

const Col = styled.div`

  display: flex;

  flex-direction: column;

`;

const MySCButton = styled.button`

  background: ${props => (props.primary ? props.mainColor : 'white')};

  color: ${props => (props.primary ? 'white' : props.mainColor)};

  display: block;

  font-size: 1em;

  margin: 1em;

  padding: 0.5em 1em;

  border: 2px solid ${props => props.mainColor};

  border-radius: 15px;

`;
```

3.7 Learn Different Component Patterns

To ensure you don't end up with unmaintainable and unscalable code, learning different component patterns is essential as you become more experienced in React. Knowing the different patterns is a good foundation.

But the most important aspect about it is that you know when to use which pattern for your problem. Every pattern serves a certain purpose. For example, the **compound component pattern** avoids unnecessary ***prop-drilling*** of many component levels.

3.8 Use Implicit return

Use the JavaScript feature of implicit ***return*** to write beautiful code. Let's say your function does a simple calculation and returns the result.

Bad Practice:

```
const add = (a, b) => {
    return a + b;
  }
```

Good Practice:

```
const add = (a, b) => a + b;
```

3.9 Use a Linter and Follow its Rules

A linter doesn't only help you in terms of maintaining a distinguishable import order of your dependencies. It helps you write better code in general. When you're using ***create-react-app***, there's already **ESLint** configured, but you can also set it up completely on your own or extend the rules of a pre-configured ruleset.

A linter basically observes the JavaScript code you're writing and reminds you of errors you'd more likely catch when executing the code. Another great benefit is that you can also adjust style checking.

3.10 Quotes for Attributes

Use double quotes for JSX attributes and single quotes for all other JS.

Bad Practice:

```
<Foo bar='bar' />

<Foo style={{ left: "20px" }} />
```

Good Practice:

```
<Foo bar="bar" />

<Foo style={{ left: '20px' }} />
```

3.11 JSX in Parentheses

If your component spans more than one line, always wrap it in parentheses.

Bad Practice:

```
return <MyComponent variant="long">
           <MyChild />
         </MyComponent>;
```

Good Practice:

```
return (
    <MyComponent variant="long">
      <MyChild />
    </MyComponent>
);
```

3.12 Test Your Code

Having a vision for the code you're about to write also helps you to maintain a sharp focus on serving that vision.

Tests can also serve as a kind of documentation, because for a new developer who is new to the codebase it can be very helpful to understand the different parts of the software and how they're expected to work.

It's a good practice to write test cases for each component developed as it reduces the chances of getting errors when deployed. You can check all the possible scenarios through unit testing and for that, some of the most commonly used React test frameworks you can use are ***JEST*** and ***ENZYMES***.

3.13 Use Ternary Operators

Avoid using multiple if-else statements, by replacing them with ternary operators. Let's say you want to show a particular user's details based on role.

Bad Practice:

```
const { role } = user;

if(role === ADMIN) {
  return <AdminUser />
}else{
  return <NormalUser />
}
```

Good Practice:

```
const { role } = user;

return role === ADMIN ? <AdminUser /> : <NormalUser />
```

3.14 Take Advantage of Object Literals

Object literals can help make our code more readable. Let's say you want to show three types of users based on their role. You can't use ternary because the number of options is greater than two.

Bad Practice:

```
const {role} = user

switch(role){
  case ADMIN:
    return <AdminUser />
  case EMPLOYEE:
    return <EmployeeUser />
  case USER:
    return <NormalUser />
}
```

Good Practice:

```
const {role} = user

const components = {
  ADMIN: AdminUser,
  EMPLOYEE: EmployeeUser,
  USER: NormalUser
};

const Component = components[role];

return <Componenent />;
```

3.15 Integrate Typescript

Using TypeScript has many upsides like static type checking, better code completion in your IDE, improved developer experience, and catching type errors while you write the code. There may be reasons you don't want to use TypeScript inside your React application.

But at a bare minimum it is recommend that you use **prop-types** and **default-props** for your components to ensure you don't mess up your props.

3.16 Don't Define a Function Inside Render

Don’t define a function inside render. Try to keep the logic inside render to an absolute minimum.

Bad Practice:

```
return (
    <button onClick={() => dispatch(ACTION_TO_SEND_DATA)}>    // NOTICE HERE
      This is a bad example
    </button>
)
```

Good Practice:

```
const submitData = () => dispatch(ACTION_TO_SEND_DATA)

return (
  <button onClick={submitData}>
    This is a good example
  </button>
)
```

3.17 Put CSS in JavaScript

Try to avoid raw JavaScript when you are writing React applications because organizing CSS is far harder than organizing JS.

Bad Practice:

```
// CSS FILE

.body {
    height: 10px;
  }

  //JSX

  return <div className='body'>

  </div>
```

Good Practice:

```
const bodyStyle = {
```

```
    height: "10px"
  }

  return <div style={bodyStyle}>

  </div>
```

3.18 Use Lazy-loading / Code splitting

To avoid burdens on bundling , there's a technique called code splitting where you split up your bundle into the pieces of the code your user needs. This is supported by the most common bundlers like ***Webpack***, ***Rollup***, and ***Browserify***.

The great benefit of it is that you can create multiple bundles and load them dynamically. Splitting up your bundle helps you to lazy load only the things that are needed by the user.

3.19 Remove JS Code From JSX

Move any JS code out of JSX if that doesn't serve any purpose of rendering or UI functionality.

Bad Practice:

```
return (
    <ul>
      {posts.map((post) => (
        <li onClick={event => {
          console.log(event.target, 'clicked!'); // <- THIS IS BAD
          }} key={post.id}>{post.title}
        </li>
      ))}
    </ul>
  );
```

Good Practice:

```
const onClickHandler = (event) => {
    console.log(event.target, 'clicked!');
 }

 return (
   <ul>
     {posts.map((post) => (
       <li onClick={onClickHandler} key={post.id}> {post.title} </li>
     ))}
   </ul>
 );
```

3.20 Handle Errors Effectively

Handling errors effectively is often overlooked and underestimated by many developers. However, we realize that it's mandatory in the long run to have a solid error handling inside your application. Especially when the application is deployed to production.

3.20.1 React Error Boundary

This is a custom class component that is used as a wrapper of your entire application. With the lifecycle method ***componentDidCatch()*** you're able to catch errors during the rendering phase or any other lifecycles of the child components.

The static function ***getDerivedStateFromError()*** is called during the render phase and is used to update the state of your ErrorBoundary Component

```
class ErrorBoundary extends React.Component {

    constructor(props) {

      super(props);

      this.state = { hasError: false };

    }

    static getDerivedStateFromError(error) {

      return { hasError: true };

    }

    componentDidCatch(error, errorInfo) {

      //log the error to an error reporting service

      errorService.log({ error, errorInfo });

    }

    render() {

      if (this.state.hasError) {

        return <h1>Oops, something went wrong.</h1>;

      }
```

```
      return this.props.children;
    }
  }
```

3.20.2 Use try-catch to Handle Errors

This technique is effective to catch errors that might occur inside asynchronous callbacks. Using try-catch helps us catch any error that might occur during that API call. For example, this could be a 404 or a 500 response from the API.

```
const UserProfile = ({ userId }) => {
    const [isLoading, setIsLoading] = useState(true)
    const [profileData, setProfileData] = useState({})
    useEffect(() => {
        // Separate function to make of use of async
        const getUserDataAsync = async () => {
            try {
                // Fetch user data from API
                const userData = await axios.get(`/users/${userId}`)
                // Throw error if user data is falsy (will be caught by catch)
                if (!userData) {
                    throw new Error("No user data found")
                }
                // If user data is truthy update state
                setProfileData(userData.profile)
            } catch(error) {
                // Log any caught error in the logging service
                errorService.log({ error })
                // Update state
                setProfileData(null)
```

```
            } finally {
                // Reset loading state in any case
                setIsLoading(false)
            }
        }
        getUserDataAsync()
    }, [])
     if (isLoading) {
        return <div>Loading ...</div>
    }
    if (!profileData) {
        return <ErrorUI />
    }
    return (
        <div>
            ...User Profile
        </div>
    )
}
```

3.20.3 Logging Errors

Catching and handling errors effectively is one part, logging them properly is another. Once you've set up your error handling inside your application, you need to log them persistently. The most frequently used way is the good old ***console.log***.

This might be good during development when you want a quick log, but once your application is deployed to production it becomes useless. For that reason, we need a logging service created by our own or a third-party one, such as ***Sentry.***

3.21 Implement the useReducer Hook earlier

Probably one of the most frequently used hooks in React is ***useState.*** Having a lot of different useState hooks is always a great sign that the size and therefore the complexity of your component is growing.

To increase the legibility of your component, there is the ***useReducer*** hook. ***useReducer*** is usually preferable to ***useState*** when you have complex state logic that involves multiple sub-values or when the next state depends on the previous one. ***useReducer*** also lets you optimize performance for components that trigger deep updates because you can pass dispatch down instead of ***callbacks***.

3.22 Use Shorthand for Boolean props

Often there are scenarios where we pass boolean props to a component.

```
<RegistrationForm hasPadding={true} withError={true} />
```

But you don't need to do it necessarily like this because the occasion of the prop itself is either truthy (if the prop is passed) or falsy (if the prop is missing). A cleaner approach would be:

```
<RegistrationForm hasPadding withError />
```

3.23 Avoid Curly Braces for String props

```
<Paragraph variant={"h5"} heading={"A new book"} />
```

You don't need the curly braces in that case because you're allowed to directly use strings inside your props. When you want to attach a className to a JSX Element you're most likely using it also directly as a string.

With plain strings, like in the example, it would look like this:

```
<Paragraph variant="h5" heading="A new book" />
```

3.24 Use Snippet Extensions

In Visual Studio Code, for example, there are certain extensions available that increase your productivity a lot. One type of these extensions are snippet extensions. The great benefit about them is that you don't have to write all that boilerplate code again

3.25 Integrate Self-closing Tags

In React you've got the opportunity to pass children elements to a component, which are then available to the component via its children property. Those components are often called ***composite components***. In that case you have to use an opening tag and a closing tag of course:

```
<NavigationBar>
  <p>Home</p>
  <p>About</p>
  <p>Projects</p>
  <p>Contact</p>
</NavigationBar>
```

But when there are no children needed, there isn't any sense in using an opening and closing tag.

```
<NavigationBar></NavigationBar>
```

Instead of doing this, it is recommended that we just use the component as a self-closing element like the input tag in HTML, that doesn't take children as well.

```
<NavigationBar/>
```

3.26 Good Commenting Practices

Use comments to explain why you did something, not how you did it. If you find yourself explaining how you did something, then it's time to make your code self-explanatory.

Another point is not to write comments that are obvious and redundant - for example:

```
// Prints out the sum of two numbers
console.log(sum);
```

Write comments that are legal, informative, explanatory of intent, and offer clarification,

3.27 Pure Components and Memo

React.PureComponent and ***Memo*** can significantly improve the performance of your application. They help us to avoid unnecessary rendering.

```
import React, { useState } from "react";

export const TestMemo = () => {
  const [userName, setUserName] = useState("faisal");
  const [count, setCount] = useState(0);

  const increment = () => setCount((count) => count + 1);

  return (
    <>
      <ChildrenComponent userName={userName} />
      <button onClick={increment}> Increment </button>
    </>
  );
};

const ChildrenComponent =({ userName }) => {
  console.log("rendered", userName);
  return <div> {userName} </div>;
};
```

Although the child component should render only once because the value of count has nothing to do with the ChildComponent . But, it renders each time you click on the button.

Let's edit the ChildrenComponent to this:

```
import React ,{useState} from "react";

const ChildrenComponent = React.memo(({userName}) => {
    console.log('rendered')
    return <div> {userName}</div>
})
```

3.28 Utilize ES6 Standards

Always ***destructure*** your props. Destructuring your props helps make your code cleaner and more maintainable. It also makes assigning object properties to variables feels like much less of a chore.

Know where to use ***spread/rest*** operators. You shall read about it before actually using it. Try using optional chaining if possible. Use ***const*** instead of ***var*** or ***let*** . const lets you check that a

variable should always stay constant. let indicates that the values of the variables can be changed.

Start preferring ***arrow functions (=>)*** for more cleaner code. Arrow functions allow simplifying your code to a single line.

3.29 Common Naming Conventions

Your filenames should always have consistent casing and extension. Either use ***.js*** or ***.jsx*** as explained in code structure for extensions. And ***PascalCase*** or ***camelCase*** for filenames.

In React, name your file the same as the React component inside that file i.e. without a hyphen in the filename. For example: ***Registration-Form*** → ✕ . ***RegistrationForm*** → ✔ .

Use well-descriptive names for *variables/functions/modules/Actions*, keeping in mind that it is application-specific so that even a third party or new developer can easily understand you code.

It's bad to use the underscore prefix (_) for a React component's internal methods because underscore prefixes seem to be used as a convention in other languages to denote private objects or variables. But everything in JavaScript is public. And there is no native support for privacy. So, even if you add underscore prefixes to your properties, it'll not make them private.
_onClickSubmit() → ✕

When making reducer functions, write Action types as *domain/eventNames*. For example: ***ADD_TODO*** and ***INCREMENT*** (in CAPITALS) as this matches the typical conventions in most programming languages for declaring the constant values.

Talking about cases in a React component, use ***PascalCase*** for the same , and for their instances, use ***camelCase***. For example:

```
const loginForm = <LoginForm />;
import LoginForm from './loginForm';
```

3.30 Use JSX ShortHand

Try to use JSX shorthand for passing boolean variables. Let's say you want to control the title visibility of a Navbar component.

Bad Practice:

```
return (
    <Navbar showTitle={true} />
  );
```

Good Practice:

```
return(
    <Navbar showTitle />
  )
```

4. References

1. https://levelup.gitconnected.com/5-tips-to-write-better-react-code-a5bca3f9531c
2. https://www.freecodecamp.org/news/10-points-to-remember-thatll-help-you-master-coding-in-reactjs-library-d0520d8c73d8/
3. https://betterprogramming.pub/21-best-practices-for-a-clean-react-project-df788a682fb
4. https://www.freecodecamp.org/news/best-practices-for-react/
5. https://www.loginradius.com/blog/async/guest-post/react-best-coding-practices/
6. https://www.browserstack.com/guide/coding-standards-best-practices