React – Good practices

Practices

1. Using Functional Components and Hooks Instead of Classes

- In React, you can use class or functional components with hooks. You should, however, use functional components and hooks more often as they result in more concise and readable code compared to classes.

2. Avoid using State (if possible)

- React state keeps track of the data which when changed triggers the React component to re-render. When building React applications, avoid using state as much as possible since the more state you use, the more data you have to keep track of across your app.
- One way of minimizing the use of state is by declaring it only when necessary. For instance, if you are fetching user data from an API, store the whole user object in the state instead of storing the individual properties.

2. Avoid using State (if possible)

Instead of doing this:

```
const [username, setusername] = useState('')

const [password, setpassword] = useState('')
```

Do this:

```
const [user, setuser] = useState({})
```

3. Organize Files Related to the Same Component in One Folder

- When deciding on a project structure, go for a component-centric one. This means having all the files concerning one component in one folder.
- If you were creating a **Navbar** component, create a folder called **navbar** containing the **Navbar** component itself, the style sheet, and other JavaSript and asset files used in the component.
- A single folder containing all of a component's files makes it easy to reuse, share, and debug. If you need to see how a component works you only need to open one single folder.

4. Opt for Fragments Instead of Divs Where Possible

- React components need to return code wrapped in a single tag usually a **<div>** or a React fragment. You should opt for fragments where possible.
- Using **<div>** increases the DOM size, especially in huge projects since the more tags or DOM nodes you have, the more memory your website needs and the more power a browser uses to load your website. This leads to lower page speed and potentially poor user experience.
- One example of eliminating unnecessary **<div>** tags is not using them when returning a single element.

5. Follow Naming Conventions

- You should always use PascalCase when naming components to differentiate them from other non-component JSX files. For example: **TextField**, **NavMenu**, and **SuccessButton**.
- Use camelCase for functions declared inside React components like **handleInput()** or **showElement()**.

6. Avoid Repetitive Code

- If you notice you are writing duplicated code, convert it into components that can be reused.
- For example, it makes more sense to create a component for your navigation menu instead of repeatedly writing the code in every component that requires a menu.
- That’s the advantage of a component-based architecture. You can break down your project into small components you can reuse across your application.

7. Avoid Complicated Structures

- Instead of passing the props object, use object destructuring to pass the prop name. This discards the need to refer to the props object each time you need to use it.

7. Avoid Complicated Structures

For example, the following is a component that uses props as is.

```
const Button = (props) => {
 return <button>{props.text}</button>;
};
```

With object destructuring, you refer to the text directly.

```
const Button = ({text}) => {
 return <button>{text}</button>;
};
```

8. Keep components small and function-specific

- Function-specific components can be standalone, which makes testing and maintenance easier.
- Each small component can be reused across multiple projects.
- Components executing general functions can be made available to the community.
- With smaller components, it’s easier to implement performance optimizations.
- It’s easier to update smaller components.
- Bigger components have to perform harder and may be difficult to maintain.

9. Name the component after the function

- It’s a good idea to name a component after the function that it executes so that it’s easily recognizable.
- For example, **ProductTable** – it conveys instantly what the component does. On the other hand, if you name the component based on the need for the code, it can confuse you at a future point of time.
- Another example, it’s preferable to name a component Avatar so that it can be used anywhere – for authors, users or in comments. Instead, if we name the component **AuthorAvatar** in the context of its usage, we’d be limiting the utility of that component.
- Besides, naming a component after the function makes it more useful to the community as it’s more likely to be discovered.

Further reading

Further reading

- https://www.makeuseof.com/what-are-web-components/
- https://www.codeinwp.com/blog/react-best-practices/

- https://reactjs.org/docs/hooks-reference.html#usereducer

- https://redux.js.org/tutorials/fundamentals/part-3-state-actions-reducers