**10 Best Practices in Front-end Development (React)**

Certainly, we’ve all experienced this situation: you land a new job and are excited to start. The interviewers paint a bright and charming picture of the company, leading you to make the decision to leave your previous position. After a week of what can only be described as ‘vacation’ — which, in many cases, amounts to little more than a brief break from the routine — it’s finally time to embark on your new professional journey. Expectations are high, you clone the projects to your computer, and suddenly, you’re faced with a true mess: makeshift solutions that make you want to cry ಥ\_ಥ, a lack of standards, and code that seems to have a logic of its own. You look at it and think, ‘What on earth is this? I have no idea how it’s working, but I’m afraid to make any minor changes, as it seems that even the slightest alteration could bring everything to a halt.’

While this situation may seem less common than you think, it’s more common than you might imagine. To prevent this from happening, developers, especially those in more senior roles, need to take the lead. After all, professionals at the C-level, tech leads, Product Owners, and others often don’t have the time or prioritize focusing on code details since they’re more concerned with delivering value and results to the company, regardless of code quality. To avoid this type of challenge, I’ve adopted specific practices in my projects that I consider essential and believe everyone should adopt. While not everyone may remember your code, those who conduct code reviews will certainly notice and commend your attention to detail. This can be one of the most effective ways to stand out in the company and, perhaps, earn a well-deserved promotion.

**1. Use of Absolute Paths Instead of Relative Paths**

When entering a new project, it’s common to encounter paths filled with “../../../../../.” These paths are known as relative paths, and while they are a way to import files, they are not the most recommended approach. The ideal practice is to use absolute paths, which provide the full path to a file. To implement this in your project, some configurations are required, especially if you are using Webpack or TypeScript.

Setting up Webpack (create-react-app):

If you are using create-react-app, configuring absolute paths is relatively straightforward. Start by creating a file in the root of your project called “jsconfig.json” and add the following content:

{  
 "compilerOptions": {  
 "baseUrl": "src"  
 },  
 "include": ["src"]  
}

Setting up TypeScript:

If you’re using TypeScript, add the following configurations to your “tsconfig.json” file:

{  
 "compilerOptions": {  
 "baseUrl": "src",  
 "paths": {  
 "@/\*": ["src/\*"]  
 }  
 },  
 "include": ["src"]  
}

By doing so, you can transform a code snippet that looks like this:

import { Button } from '../../../../components/Button'  
import { Icon } from '../../../../components/Icon'  
import { Input } from '../../../../components/Input'

Into something cleaner and easier to read, like:

import { Button } from '@/components/Button'  
import { Icon } from '@/components/Icon'  
import { Input } from '@/components/Input'

**2. Using “Export Barrel” for Module Organization**

While looking at the code mentioned earlier, I recalled a technique that can significantly enhance code readability and maintenance: the “export barrel,” also known as “re-export.” This approach involves creating a file called “index.js” (or “index.ts” if you are using TypeScript) within a folder and, from this file, exporting all the modules present in the folder.

To illustrate, imagine you have a folder named “components” with the following files: “Button.tsx,” “Icon.tsx,” and “Input.tsx.” With the “export barrel” technique, you would create an “index.ts” file and fill it as follows:

export \* from './Button'  
export \* from './Icon'  
export \* from './Input'

By doing this, when you want to use these components on a page or in another module, you can import them all at once like this:

import { Button, Icon, Input } from '@/components'

This practice simplifies code organization and improves maintenance since it reduces the need to list each component individually. Moreover, it makes the code cleaner and easier to understand, which is crucial in medium to large-scale projects.

**3. Choosing Between “Export Default” and “Named Export”**

As we delve into the topic of the “export barrel,” it’s essential to note that it can conflict with the use of “export default.” If this is not clear, I’ll illustrate the situation with examples:

Let’s go back to our components:

export const Button = () => {  
 return <button>Button</button>  
}  
export default Button

export const Icon = () => {  
 return <svg>Icon</svg>  
}  
export default Icon

export const Input = () => {  
 return <input />  
}  
export default Input

Imagine each of these components is in a separate file, and you want to import all of them at once. If you are accustomed to default imports, you might attempt something like:

import Button from '@/components'  
import Icon from '@/components'  
import Input from '@/components'

However, this won’t work because JavaScript can’t determine which “export default” to use, resulting in errors. You would be forced to do something like this:

import Button from '@/components/Button'  
import Icon from '@/components/Icon'  
import Input from '@/components/Input'

This, however, negates the advantage of the “export barrel.” How can you resolve this dilemma? The solution is simple: use “Named Export,” which is exporting without “default”:

import { Button, Icon, Input } from '@/components'

Another critical issue associated with “export default” is the ability to rename what you’re importing. I’ll share a real-life example from early in my career. I inherited a React Native project where the previous developer used “export default” for absolutely everything. There were screens named “Login,” “Register,” and “ForgotPassword.” However, all three screens were copies of each other with minor modifications. The problem was that, at the end of each screen file, there was an “export default Login.” This resulted in confusion, as the route file imported correctly:

import Login from '../../screens/Login'  
import Register from '../../screens/Register'  
import ForgotPassword from '../../screens/ForgotPassword'  
  
// Further down, the usage in routes:  
  
 {  
 ResetPassword: { screen: ResetPassword },  
 Login: { screen: LoginScreen },  
 Register: { screen: RegisterScreen },  
 }

But when opening the screen files, they all exported the same name:

const login() {  
 return <>tela de login</>  
}  
export default Login

const login() {  
 return <>tela de registro</>  
}  
export default Login

const login() {  
 return <>tela de esqueci minha senha</>  
}  
export default Login

This created a maintenance nightmare with constant confusion and the need for extreme vigilance to avoid errors.

In summary, it’s highly recommended to use “Named Export” in most cases in your project and resort to “export default” only when strictly necessary. There are situations, such as Next.js routes and React.lazy, that may require the use of “export default.” However, it’s crucial to strike a balance between code clarity and compliance with specific requirements.

**4. Proper File Naming Conventions**

Let’s imagine you have a components folder with the following files:

--components:  
----Button.tsx  
----Icon.tsx  
----Input.tsx

Now, suppose you want to separate styles, logic, or types of these components into separate files. How would you name these files? An obvious approach might be the following:

--components:  
----Button.tsx  
----Button.styles.css  
----Icon.tsx  
----Icon.styles.css  
----Input.tsx  
----Input.styles.css

Certainly, this approach can seem somewhat disorganized and challenging to understand, especially when you intend to further divide the component into distinct files, such as logic or types. But how can you keep the structure organized? Here’s the solution:

--components:  
----Button  
------index.ts (exports everything necessary)  
------types.ts  
------styles.css  
------utils.ts  
------component.tsx  
----Icon  
------index.ts (exports everything necessary)  
------types.ts  
------styles.css  
------utils.ts  
------component.tsx  
----Input  
------index.ts (exports everything necessary)  
------types.ts  
------styles.css  
------utils.ts  
------component.tsx

This approach makes it easy to identify the purpose of each file and simplifies the search for what you need. Additionally, if you’re using Next.js or a similar framework, you can adapt the file naming to indicate whether the component is intended for the client or server side. For example:

--components:  
----RandomComponent  
------index.ts (exports everything necessary)  
------types.ts  
------styles.css  
------utils.ts  
------component.tsx  
----RandomComponent2  
------index.ts (exports everything necessary)  
------types.ts  
------styles.css  
------utils.ts  
------component.server.tsx

This way, it becomes extremely simple to distinguish whether a component is meant for the client or server side without the need to open the code for verification. Organizing and standardizing file naming is essential to maintain clarity and efficiency in development projects.

**5. Proper Use of ESLint and Prettier for Code Standardization**

Imagine yourself working on a project with over 10 colleagues, each bringing their own coding style from past experiences. Here’s where ESLint and Prettier come into play. They play a crucial role in maintaining code consistency throughout the team.

Prettier acts as a kind of “guardian” of code formatting, ensuring that everyone adheres to the style guidelines set for the project. If the project’s standard dictates the use of double quotes, for example, you can’t simply opt for single quotes because Prettier will automatically replace them. Furthermore, Prettier can perform various other fixes and formatting, like code alignment, adding semicolons at the end of statements, and more. You can check the specific Prettier rules in the official documentation: Prettier Options.

ESLint, on the other hand, enforces specific rules on the code, helping to maintain a cohesive and coherent codebase. For instance, it can enforce the use of arrow functions over regular functions, ensure that the React dependencies array is properly filled, prohibit the use of “var” declarations in favor of “let” and “const,” and apply naming conventions like camelCase. You can find the specific ESLint rules in the official documentation: ESLint Rules.

The combined use of ESLint and Prettier helps to maintain consistency in source code. Without them, each developer can follow their own style, which can lead to conflicts and maintenance difficulties in the future. Having these tools set up is essential for the longevity of a project, as it helps keep the code organized and easy to understand. If you’re not already using ESLint and Prettier, seriously consider incorporating them into your workflow, as they will greatly benefit your team and your project as a whole.

**6. Husky and Lint-Staged: Reinforcing Code Standardization**

If you are already familiar with ESLint and Prettier, you know that in some situations, it’s possible to circumvent the rules defined by these tools. To ensure compliance with your established code guidelines and avoid formatting issues, it’s highly recommended to use Husky and Lint-Staged.

These two tools play a vital role in your development workflow, allowing you to set up ESLint and Prettier to run before making a commit. This means you won’t be able to commit unless the code is in compliance with the rules you’ve set. You can also configure these tools to check the code before pushing it to the repository.

Furthermore, Husky supports running other scripts or actions before a commit or push, which broadens your possibilities to automate verification tasks and ensure code quality.

Another advantage of Husky and Lint-Staged is their integration with code hosting platforms like GitHub. This allows you to set up automated testing and quality checks before accepting a pull request. This way, you can ensure that the submitted code complies with your established rules, minimizing issues and ensuring consistency.

These tools are essential to prevent developers from making commits with evident issues and to ensure that the code is always aligned with the established guidelines. The combination of ESLint, Prettier, Husky, and Lint-Staged is a highly effective practice for maintaining code quality and standardization in your project.

**7. Custom Hooks for Logic Reusability**

When working with React, it’s common to use navigation hooks provided by libraries like react-router-dom, Next.js, or react-navigation (for React Native). However, these generic navigation hooks lack knowledge of the specific pages in your application, which can lead to limitations. An effective solution is to create custom navigation hooks that are aware of all the pages in your application, making it easier to navigate between them.

Here’s an example of creating a custom navigation hook:

import { useCallback } from 'react'  
import { useNavigate } from 'react-router-dom'  
import type { Routes } from '@/types'  
  
export const useRouter = () => {  
 const navigate = useCallback((path: Routes) => goTo(path), [goTo])  
  
 return { navigate }  
}

There might be some initial resistance due to the apparent complexity of abstracting a navigation hook. However, in the long run, this approach offers several advantages. It simplifies calling the hook and provides autocomplete for the function, making the code cleaner and easier to understand. Moreover, it streamlines maintenance because if you need to change the navigation library in the future, you’ll only need to make changes in the custom hook rather than everywhere the navigation hook is used.

This same idea of creating custom hooks can be applied to other aspects of your application, such as managing cookies, local storage (localStorage), API calls, and more. This approach allows you to easily reuse logic in multiple places in your project, promoting modularity and simplified code maintenance.

**8. The Difference Between Hooks and Utility Functions**

Understanding the distinction between creating a hook and a utility function is crucial, although developers often confuse these two concepts. To illustrate this difference, I’ll share a practical example that reflects a less experienced approach I used to take in my projects.

Inside the Hooks folder:

export const useFormat = () => {  
 const formatHour = (date: number) => {  
 return new Date(date).toLocaleTimeString('pt', { timeStyle: 'short' })  
 }  
 // Other date/time formatting functions  
 return { formatHour }  
}

In this example, I created a hook that actually contains a utility function. This approach is not the most efficient since a new hook is created every time the component using it renders, which can impact performance. Additionally, a hook is specific to a component and not easily reusable in other contexts.

The correct approach would be to create a utility function in a dedicated directory, such as “Utils”:

Inside the Utils folder:

export const formatHour = (date: number) => {  
 return new Date(date).toLocaleTimeString('pt', { timeStyle: 'short' })  
}

In this case, the utility function is created only once and can be used anywhere in your project, irrespective of the context. It is not tied to a specific component and does not generate the overhead of creating hooks with each rendering.

The fundamental distinction is that a hook is a component without a visual part, created externally to React, and called within a specific component, whereas a utility function is created only once outside the React context and can be used anywhere. The choice between hooks and utility functions should be based on the specific needs of your project and code reuse considerations.

**9. Using useCallback and useMemo for Performance Optimization**

The distinction between using useCallback, useMemo, and not using them can have a significant impact on performance optimization in React applications. Essentially, useCallback is employed to memoize functions, while useMemo is used to memoize values. Both serve the purpose of preventing functions or values from being recreated with each component rendering.

This is particularly important due to an intrinsic issue in React: the frequent recreation of functions and values during rendering. While React is efficient, this frequent recreation can lead to unnecessary re-rendering of child components, potentially resulting in a performance drop. It’s essential to note that this occurs due to the way React is designed, and it’s not something easily changed.

The solution to this problem lies in the judicious use of useCallback and useMemo. They allow you to avoid the frequent recreation of functions and values, thereby enhancing your application’s performance. However, it’s crucial to use them judiciously, as excessive use can have the opposite effect and harm performance. Each use of useCallback or useMemo involves two actions: one to create the function or value and another to memoize it. Additionally, some functions or values may be more efficient to recreate than to memoize.

If you are already familiar with this issue, you likely know how to use useCallback and useMemo. However, it’s important to exercise caution, as excessive use can result in performance drops and even hard-to-trace bugs (I guarantee you wouldn’t want to deal with such bugs).

If you are unfamiliar with this issue, it’s highly recommended to read various articles on the subject, as it’s a significant problem with the potential to pose performance challenges and debugging difficulties in React applications.

**10. Logic Separation**

In addition to considering the distinction between hooks and utility functions, it’s crucial to keep pages as simple as possible and separate logic when feasible. By doing so, you can enhance the organization and maintainability of your code. Here are some practices to consider:

Toast Logic: If a page needs to display toasts, it’s a good idea to create a separate component and a dedicated function for displaying these toasts. This way, you keep the logic separate from the presentation on the page.

API Calls: If a page makes API calls, consider using libraries like React Query and keep the logic related to these calls in an external hook. The page itself can then use this hook, keeping it cleaner and focused on presentation.

Reusable Components: For components that are reusable across various parts of your application, place them in a global components directory. If organization becomes confusing this way, you can adopt a component design pattern, such as Atomic Design. However, remember that software development is dynamic, and new approaches may emerge at any time. Evaluate the quality and effectiveness of these approaches for your specific situation.

By following these code organization practices and adopting a mindset of continuous learning, you can improve the quality and efficiency of your work while making the code easier to understand and maintain. This, in turn, will further enhance the value of your work as a developer.

**Conclusion**

Remember that software development is a field in constant evolution, and the ability to adapt and learn continuously is crucial. Each practice and concept discussed here aims to make your code more efficient, readable, and easy to maintain. By implementing these guidelines and continually seeking improvements, you can elevate the quality of your work and add even more value to your career as a developer. Stay updated and keep exploring new approaches, as continuous learning is the key to success in this dynamic and challenging field.

Feel free to reach out to me with any questions, contributions to the article, or suggestions.