React File Structure - How to organise React components - Embracing locality for react file structure

Written by Seth Corker on Benevolent Bytes

React's popularity was its ease of use, it didn't tell you where to put anything which made it easier than frameworks like Angular which had an expectation of your apps structure and dictated it more forcibly. This is also one of the hidden complexities of React, you can easily add it to your project, and it starts simple, but you'll see pretty quickly that without some guidelines on where components and supporting files should be placed, it can get messy fast.

Here's a common approach I used when I started writing React, organisation by type. Then, I'll show you what I moved toward as projects grew. It's a simple technique that allows for a nice developer experience as projects inevitably grow. First, let's see why this problem exists.

Why React is great and why React is terrible

As a developer, the appeal of React was its position within the context of Model-View-Controller architecture. Originally, it was the "View". Handle your business logic and domain modelling however you like and pass it to React to display it. This made it effortless to adopt because you didn't have to change what you were doing too much. You could bring your own models and controller layer. React is still very much a library rather than a framework. This is its biggest strengths, but it's also a weakness when it comes organisation.

React doesn't define any conventions on where your components should go or how they should be organised within a larger system. This is a freedom the developer has to decide. This freedom can be terrifying for new developers. There is no right answer, so you get fragmentation of best practices.

I'll show you two approaches to organising React components, this isn't specific to just React and will work with many other component-based libraries and frameworks. There are tradeoffs with both approaches, as there always is, and it may not suit everyone's taste, but it's what I've found to be useful working on many different projects over the years. Let's jump into the first approach.

The problem with organising by type

Screenshot of VSCode filesystem showing organisation by type

Organising components by type is a common approach to take and why wouldn't it be? We've been doing it since React's inception. Even the maintainers of React have recommended separation by containers and components. Ruby on Rails developers are familiar with the standard hierarchy and scaffolding that maps perfectly from filesystem to Model-View-Controller. It's obvious where each type of file needs to go. Organising by type works well for small projects. With a few components, you're not going to get lost, and it's easy to keep track of everything. Let's take a look at how we might organise files in a React app when organising by type.

How does it work?

The reasoning behind organising by type is that everything that's similar should live in the same place. Your React components should live in a `components` folder. Tests, in a `tests` folder. CSS styles, in a styles folder and so on. Sometimes there might be some changes to this structure if you're using CSS-in-JS solutions or opting for tests closer to the component than the top-level directory but, for the most part, this structure can be seen in many React projects. Once you have more than 10 components, things might start to get more challenging. At larger sizes, it can be overwhelming.

Why it can be problematic

The biggest challenges I've faced when working with structures like this become apparent as a project grows. In the beginning, it's great. You know where everything goes, every component, container, stylesheet and test has its place. As time goes on and the project grows, it becomes more challenging to work with individual components.

Developer experience and scaling is impacted Let's say you want to edit a button component. Open the component's file and start editing away. Then you need to edit your styles, so you also navigate to your styles folder, find the button and open that. Now the component is updated, the tests are broken and some stories need to be updated. You navigate to those respective folders and open `tests/button.test.js` and

`stories/button.story.jsx` too. What happens when you now want to open up the components that consume the button and make changes to them too?

When organising components by type, I often find myself bouncing back and forward in different files in different places to get changes made. I don't want to close certain files because I might need to navigate back to them again, so I just end up with a cluttered workspace with a multitude of tabs open in my editor. In this example it's not so bad but as the level of nesting grows, it can become challenging to find what you need. At that point, it's often easier to search for files and ignore the structure in day-to-day development completely.

What am I missing? An issue arises when components don't all have the same files associated with them. Your standard components might have stories, styles, tests and the component itself but what if it doesn't? It's not easy to figure out at a glance. Did you notice the screenshot I included is missing a test? Maybe, what if there were 10 components or 100? You have to rely on other tools or investigation to figure out the dependencies for your components. If a test is missing, it's not obvious.

Where do I put it? What happens when you decide there's some component logic that only relates to your header component, and you want to break it out into an utils file? The current structure doesn't allow for this. The obvious thing to do is create a new top-level folder called utils. So let's do that. We've made a new `utils/header.js` file, but we've also made a commitment to where files go in the future. We've created a signal for other engineers who will continue to expand on this concept and maybe even break up existing components to meet this new structure. This exacerbates the problem I mentioned earlier, not all components will share the same dependencies and knowing this involves manual investigation. It's clear where to put files when the folder already exists but it's unclear what warrants a new folder's creation.

Components are movable Organising files by type makes it more awkward to move components within the project or to pull them out. You have to go through and find the dependencies for a component that live in very different places across the filesystem, update the dependencies and hope you didn't miss anything. Your imports are often predictable but lengthy and a chore to change. It's not a big deal as refactoring the location of components is unlikely to happen often but when it doesn't, it can be error prone.

So what's a better approach as a project scales?

Organising components by locality

Screenshot of VSCode filesystem showing organisation by locality

How does it work?

Organising by locality is an extension of encapsulation. Everything that a component relies on lives as close to the component as possible. This makes it easy to see the dependencies of a React component because the named folder contains the component itself alongside tests, utils, stories and styles. The names chosen, include the component name, the type of file it is and the extension.

Why I like this folder structure for components

There are many ways to organise components in React and other component-based libraries. This one is one of my favourites because I find it the most flexible. It can start of simple and grow as your project grows. It's easy to use and offers a nice developer experience.

It's clear what's missing When organising by type, it's not immediately obvious if there is a test or style associated with each component. When organising by locality, it's obvious. If a component doesn't have an associated test, stylesheet or story, looking in a single place tells us. Adding these files is easy too, it all happens in the same place.

Scaling is easy Adding a new component is easy. Create a folder with the component name and add your supporting files. As your project grows, these component folders can begin to take on different shapes depending on requirements. A button component might start off being enough but eventually requires different components or styles for different purposes. You could extend this structure without breaking the core organisation principle. All the variants of the buttons are still buttons and have coupling with styles and tests, so they can all live within the same hierarchy.

Screenshot of VSCode filesystem showing additional variants for a button

This gives developers more flexibility when needed, a component can be as simple or complex as it needs to be while still following a consistent structure. If one component needs different stylesheets depending on use, it can have them and the complexity is contained within the component's domain. If you need an utils file just for one component, you can add it to just that component's folder without any obligation to keep adding utils for components that don't need them.

Components are movable As you project grows, these components can be moved around easily. If your supporting styles, and utils are imported using relative imports, it doesn't matter where the component folder moves to. The references won't change. Likewise, your stories and tests can import the files under test using relative imports too. There's no need to follow paths throughout your codebase trying to corral the various spread out dependencies.

Avoiding index.js

In both approaches, I got rid of one of the most annoying issues, `index.js`. It's common to find widespread usage of `index.js` in projects to clean up file paths when importing projects and to act as an entry point for a particular folder. In practice, this can be a pain. Navigating through definitions and landing on an `index.js` means another layer of indirection. You end up importing and exporting files from these files to make a nicer public interface for consuming components and there's nothing worse than looking at your editor trying to find a file when everything is called `index.js`. This is also why both approaches include the component name as the prefix followed by the intent of the file and ending in an extension. This is more easily searchable across the codebase and helpful when editing multiple files to avoid confusion.

Extending this approach

It can be difficult to find the right structure for your project. There may be things you don't like and that's okay. Change it. Organising by locality can be combined with organisation by type when necessary too. This helps with certain frameworks like Next.js where you might want to keep your pages directory, business logic and components separate. It can be applied to atomic design philosophy where your components folder could contain atoms, molecules and organisms. Then within each of those divisions, lives the component folder.

Conclusion

Organising components by locality is my favourite approach so far. It's flexible and scalable. It offers a nice developer experience and isn't difficult to transition to. Organising by locality creates clear boundaries around components which can be useful when you simply want to transplant your components into its own component library. If it doesn't work for you, expand upon it and combine it with what you need. Every team works differently and projects can benefit from different styles of organisation that better suit the teams that work on them. In the end, it doesn't matter how you choose to organise your React app, as long as you choose something.