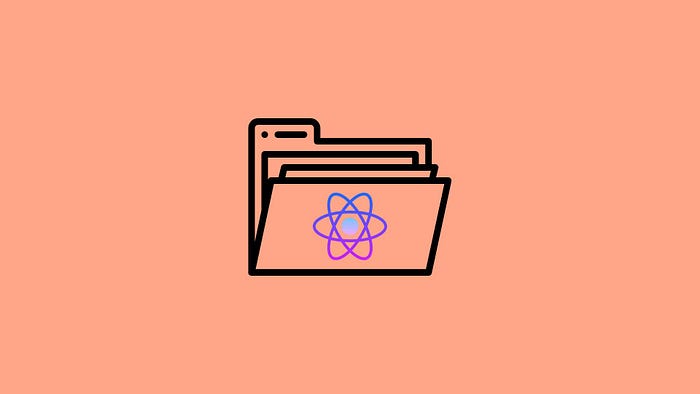
**React Folder/File Structure Patterns and Tips: Part 1s**

Hello everyone, welcome to **#SundayTechMusings**:) First of all, I’m very glad because my articles getting likes and comments. If you have any suggestions for topics, please don’t hesitate to contact me, I would be very happy.

Today I will talk about my favorite folder structures and tips. When I wrote my first React application, I remember all the time I was looking for how the structure should be. So I know a lot of new programmers are confused about that. I will try to tell the best practices that I learned. I will divide this topic into two parts. The first one will be about folder structures and the second will be about best practices.



React Folder

**Before my recommendations, Is there a recommended way to structure React projects?**

[React Documentation](https://legacy.reactjs.org/docs/faq-structure.html) answers this question:

*React doesn’t have opinions on how you put files into folders. That said there are a few common approaches popular in the ecosystem you may want to consider.*

Yes, maybe you have already seen that. So what are these popular approaches?

* Grouping by features or routes
* Grouping by file type
* Atomic Design

We can extend this list but I will explain only three of these.

**1- Grouping by features or routes (Feature-Based Pattern)**

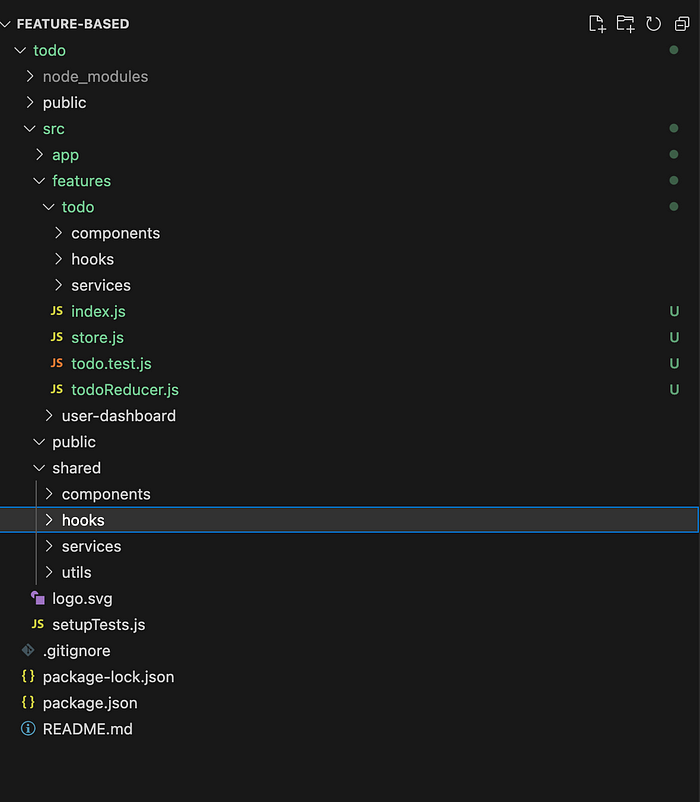


file structure (Feature-Based Pattern)

The Feature-Based Pattern is like organizing your code in a React app by grouping similar things together. Each group has all the stuff needed for a specific job in the app, like building blocks, tools, and other necessary things.

Imagine these groups as separate boxes that you can add, remove, or change without messing up the rest of the app. This way, you can also use the same stuff in different parts of the app to save time and effort.

I will create a simple folder structure with feature-based.



file structure

In the above screenshot, I’ve created four main folders.

* **src/app**: It’s a main application folder.
* **src/features**: We hold all features in that folder. Each feature has its own components, services, tests, .etc in this folder.
* **src/shared**: In the application, we will have a lot of common components, services, etc. We will keep these common files in the shared folder.
* **public**:This is for static files like images, icons, etc.

If you have a big application, or if you think the application will grow too much, you can consider using a feature-based folder structure. In this pattern, we can separate our features and it will be a very clear folder structure.

**Advantages**

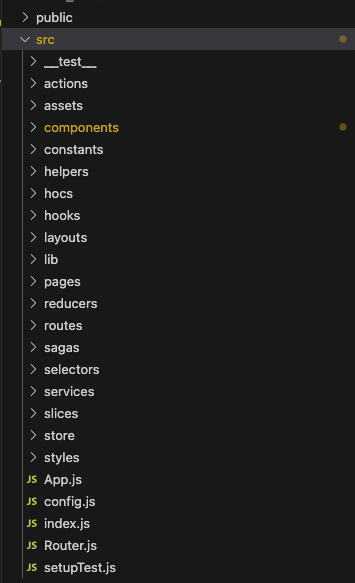
The best thing about this setup is that it’s really easy to add or change the code. Because the code is divided into different parts (features), you can easily add new parts or improve the ones already there. This split also makes the codebase simpler, as some files can be considered private, which helps you understand the code better.

Another good thing is that the code outside the features section is usually straightforward to grasp. Most of the important stuff is neatly packed inside the features section. This makes it simpler to understand and work with the code.

**Disadvantages**

The main downside of this setup is that it can get a bit complicated. If you’re working on a big application, this added complexity isn’t a big problem, because it actually makes the whole project less complicated. However, if your app is small and only has a few parts/pages, this system might end up being too much. Many of the folders might not even be used, or they might only have a few files. Because of this, I suggest using this folder structure mainly for bigger and more advanced projects that really need this extra separation.

**2- Grouping by file type**



react folder structure

This is my favorite folder structure for small-medium-size projects. We create folders for each type in the src folder. Every folder could have a **common folder** that contains shared utilities.

In this pattern, we avoid too many nestings, unlike feature-based patterns. But if we have a lot of features in the project it can be a complex structure.

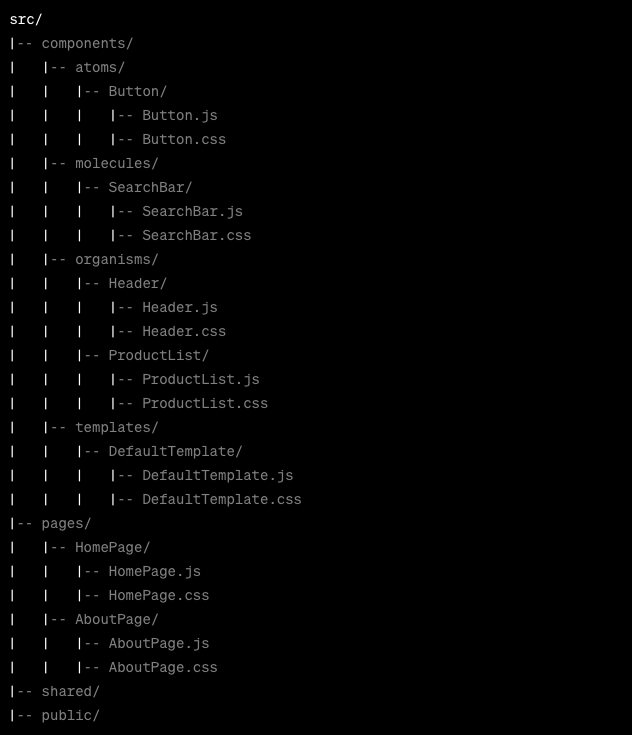
**Advantages**

* **Easy to Find:** You can quickly locate files because they’re organized by their purpose.
* **Less Messy:** When your app gets bigger, this keeps things tidy and helps prevent confusion.
* **Readability:** Others can understand your code better when it’s organized logically.
* **Reusable Code:** You can easily reuse common pieces of code in different parts of your app. Which is my reason for using this pattern.

**Disadvantages**

* **Lots of Folders:** It can get overwhelming with many folders, making it harder to find things.
* **Learning Curve:** New team members might take time to learn where everything is.
* **More Updates:** Changing file locations or names can cause problems with how files are connected.
* **Some Complexity:** Sometimes, files might belong to more than one category, which can be tricky.

**3- Atomic Design**



Atomic design folder structure

I believe you heard about Atomic Design in your life at some point. Atomic Design is a methodology for creating design systems and organizing code in a way that promotes reusability, scalability, and maintainability. It’s a hierarchical approach that breaks down user interfaces into smaller, reusable components.

In Atomic Design, components are organized into five distinct levels based on their complexity and reusability. These levels are:

1. **Atoms:** Atoms are the smallest building blocks of your UI. They represent individual HTML elements like buttons, input fields, labels, and more. Atoms are highly reusable and have minimal dependencies.
2. **Molecules:** Molecules are combinations of atoms. They represent simple UI components that have a specific functionality. For example, a search bar might be a molecule composed of an input atom and a button atom.
3. **Organisms:** Organisms are more complex UI components that combine multiple molecules and/or atoms to form a functional unit. An example could be a header component that includes a logo (atom), navigation (molecule), and a search bar (molecule).
4. **Templates:**Templates are higher-level structures that provide a layout for arranging organisms and molecules on a page. They define the overall structure of a page without containing the actual content.
5. **Pages:** Pages are instances where you populate templates with actual content and data. They represent the final product that users interact with.

**Conclusion**

My advice is to choose your familiar file structure. Just try to avoid too many nestings. If you have deep directory nesting, the project can be bad DX(Developer Experience).

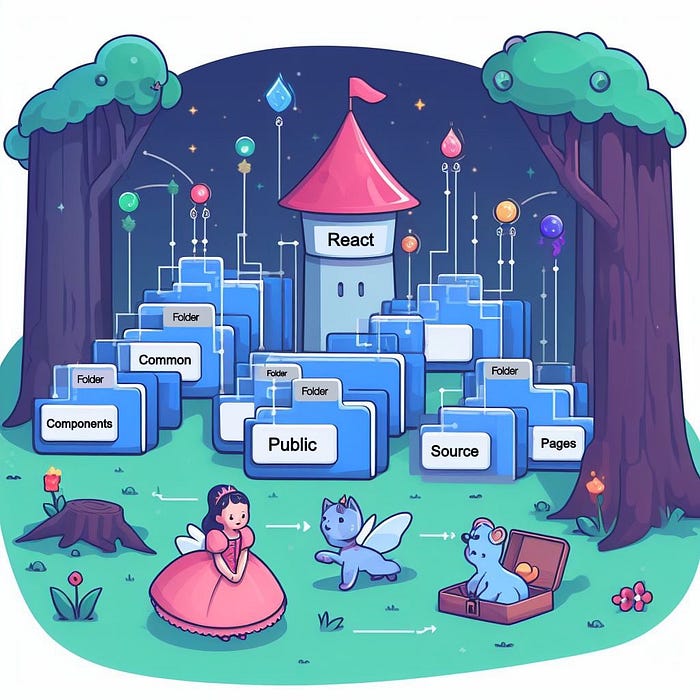
React says don’t overthink it. *Don’t spend more than five minutes* to choose file structure. This is React documentation’s advice. As the project grows larger, you can mix these patterns and that would be not a problem.

I will write Best Practices in part 2, until that happy hacking :)

So, I guess that’s all from me for this week. For more advanced JavaScript and React articles, feel free to follow me. :)

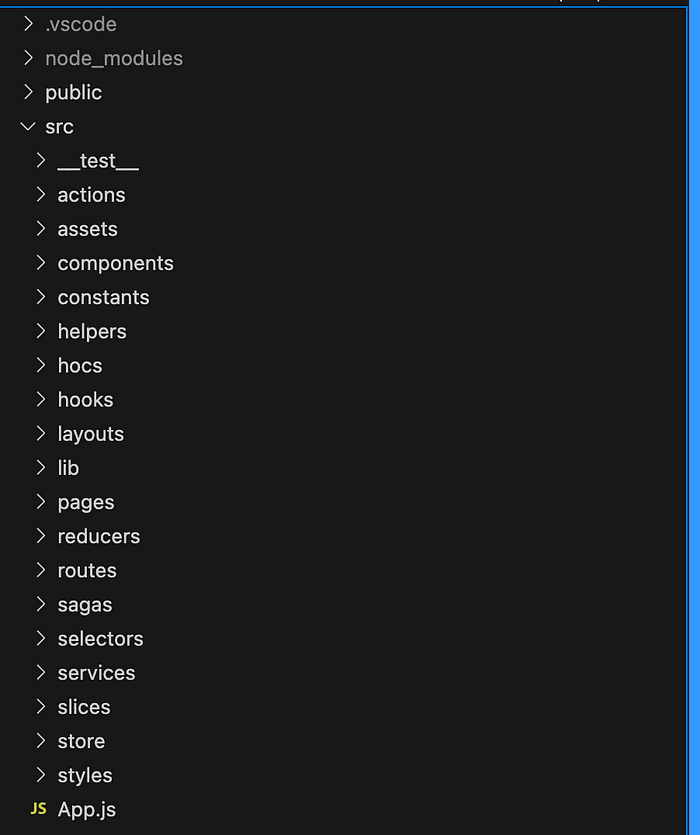
**React Folder/File Structure Patterns and Tips: Part 2**

Hello everyone. Welcome to [**#SundayTechMusings**](https://www.linkedin.com/feed/hashtag/?keywords=sundaytechmusings&highlightedUpdateUrns=urn%3Ali%3Aactivity%3A7033126066730774529)**.**I know, I haven’t written an article for a while and I’m truly sorry for this.[In the previous article](https://medium.com/@caglayanyanikoglu/react-folder-file-structure-patterns-and-tips-part-1-b8e55bda446f), I talked about React folder structure types. In this second part, I will continue with some folder tips and best practices. In my projects, I always use these methods and I want to share them with you.



React Folders

Usually, I use the “Grouping by file type” pattern in my projects. Because I like to commonize my functions a lot. But as I mentioned in my previous writing, if the project is big, a “Feature-based” pattern would be better.

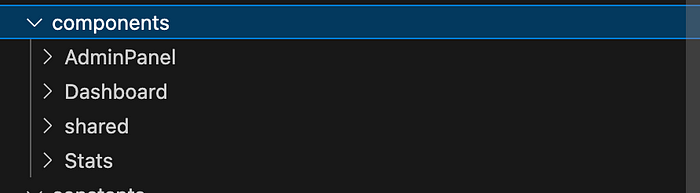


In the screenshot above, you can see, I separated files by their type. I will not talk about folder details because I’ve already written in a previous article.

**What are the tips for folder and file structures in React?**

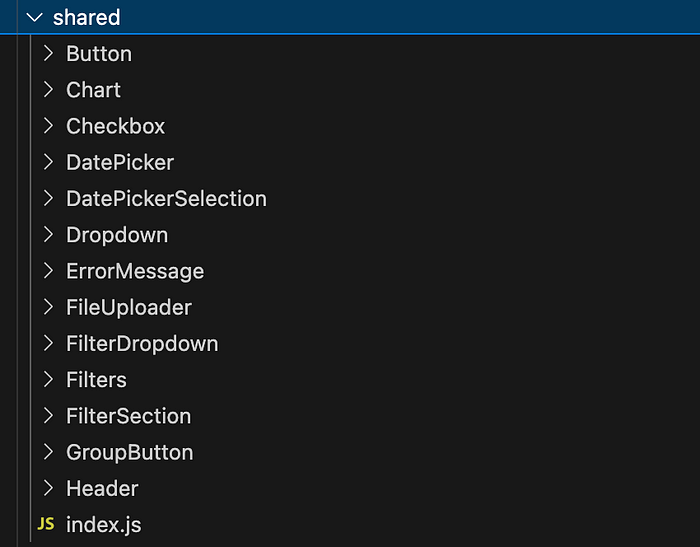
**1- ) “shared” folder for common things**

We have a components folder right, every project has this. In this folder, we hold page components in their name.



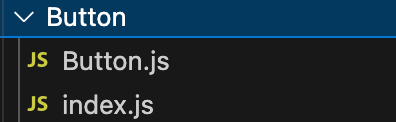
For example, you have 3 main pages in your application and you created their pages and their related components. For sure, on every page, there will be specific components for only that page. After you implement these components, you will need “Button”, “Modal”, etc components. These components will be used for multiple pages.

I always create a “***shared***” folder to put common components into that folder. You can see common components in the screenshot below:



**2-) “index.js” for every component, page, etc…**

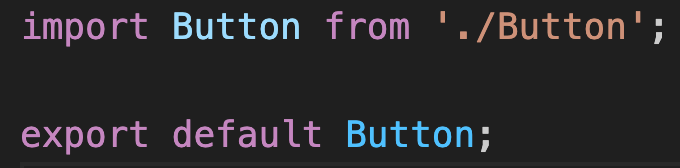
In this pattern, we create “index.js” for every component and page. I love this pattern but what are the benefits?



In the application, if we want to import a Button component, we should call it like this:

import Button from "./components/shared/Button/Button.js";

It’s very bad looking though. “***Button/Button.js”.***Let’s create index.js.

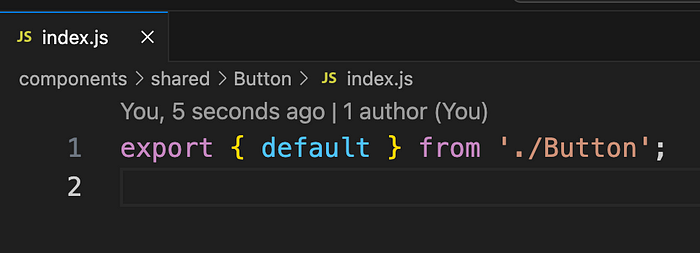


index.js

Basically, we import the Button component and export it from index.js. We don’t have to write “Button/index.js”. Because it already looks for index.js.

import Button from "./components/shared/Button";

Now it seems more clear.

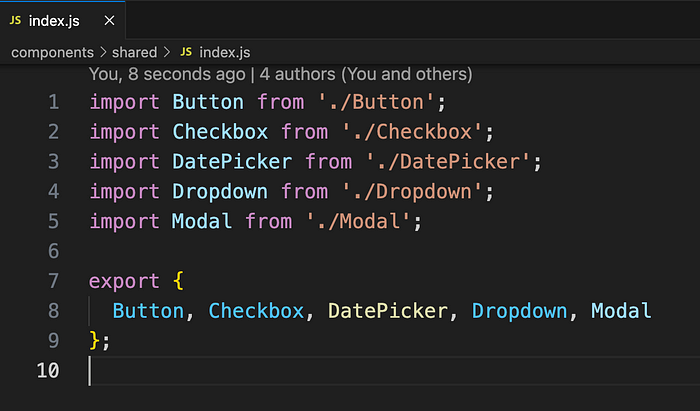


index.js

There’s also, an easy and more fancy way to import and export components as seen in the screenshot above :).

**3-) Import files to one file and export from it**

For example: In “shared” components, I’ve also created index.js. I import all components from “shared/index.js” like the below screenshot.



shared/index.js

*But why?*

For example, in some components, I need the “Modal”, “Button” and “Dropdown” components. My code will be like this:

import Button from "./shared/Button";  
import Modal from "./shared/Modal";  
import Dropdown from "./shared/Dropdown";

It will cause a lot of lines and complexity will go crazy. When I export all these components from shared/index.js, this code transforms into this:

import { Button, Dropdown, Modal } from "./shared";

It’s great, right? Usually in one component, there are a lot of internal and external imports. And if we do not use this method, we can not follow which file comes from where.

**4-) Get rid of relative path hell**

import Modal from '../../../../../../components/Shared/Modal';

Most likely, you called some file with a similar relative path like above. When we want to import a file from somewhere on another main path, we have to go main path then we have to call it. It’s very confusing and bad-looking, right? And if we change the file place, we have to fix all routes.

Create jsconfig.json in your main folder. Then add this below code.

{  
 "compilerOptions": {  
 "baseUrl": "src"  
 },  
 "include": [  
 "src"  
 ],  
 "exclude": [  
 "node\_modules",  
 "build"  
 ]  
}

After that, you can call files with **absolute path**. For example:

import Modal from 'components/Shared/Modal';

**5-) Keep components length short**

Last but not least, I always try to keep my component in maximum line height. This is for me “200” lines for advanced components. I always separate big components into small parts.

So, I guess that’s all from me for this week. For more advanced JavaScript and React articles, feel free to follow me. You are welcome to write your responses, I always read and answer them. :)