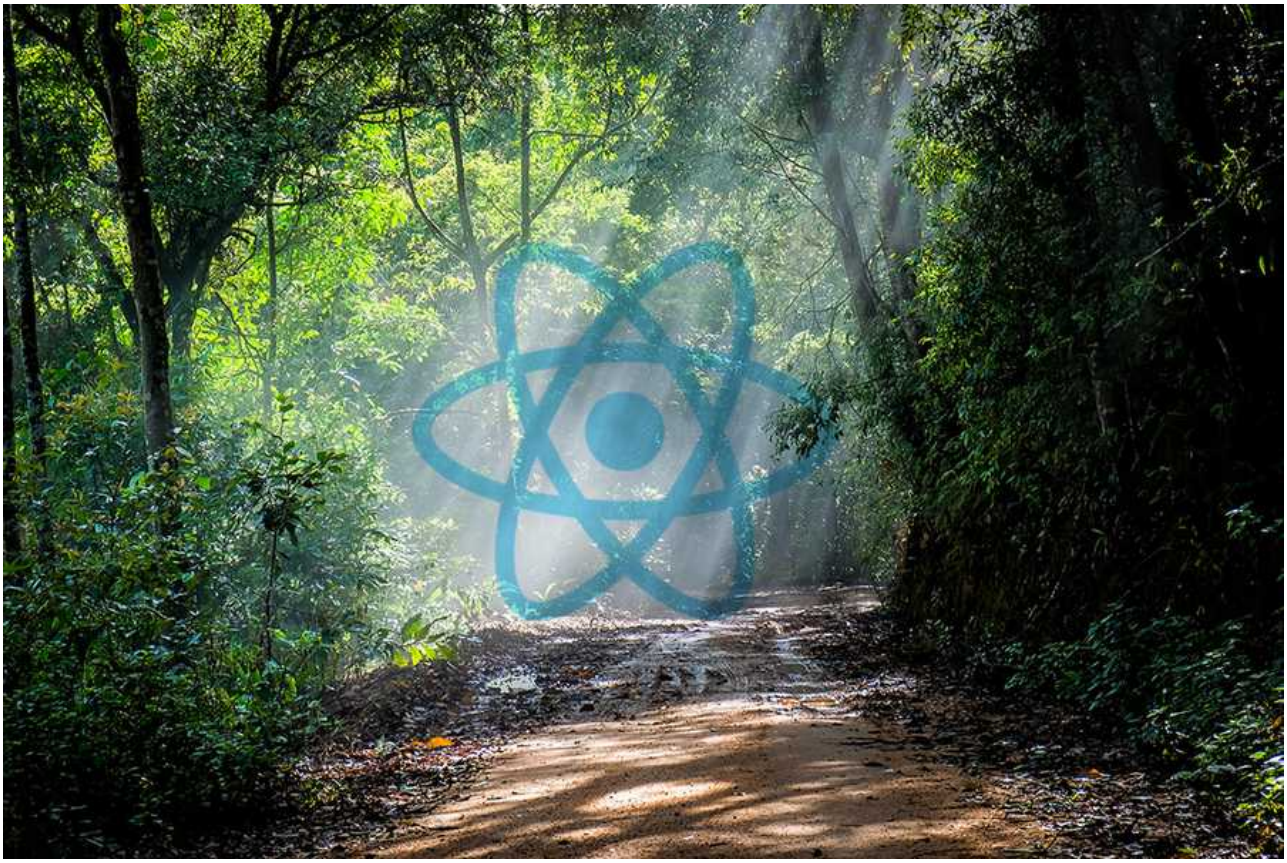
React Folder Structure in 5 Steps

APRIL 06, 2020 BY ROBIN WIERUCH - EDIT THIS POST



How to structure large React apps into folders and files is a highly opinionated topic. I struggled for a while writing about this subject, because there is no right way to do it. However, every other week people ask me about how I structure my React projects. Not only about folder structures for small React projects, but more importantly about scalable React applications. After implementing React applications for a few years now, I want to give you a breakdown on how I approach this matter for my personal projects, for my client's projects, and for my React workshops. It only takes 5 steps, and you decide what makes sense to you and how far you want to push it. So let's get started.

For anyone who says "I move files around until it feels right": This may be alright as a solo developer or in a small team. But is that really something you would do in a cross-functional team of 4 developers with a total of 5 cross-functional teams in your company? At a higher scale of teams it becomes tricky to "just move files around without a clear vision". In addition,

this is nothing I could tell my clients when they ask me about this matter. Hence this walkthrough as reference guide for anyone who is looking for some more clarity.

SINGLE REACT FILE

The first step follows the rule: One file to rule them all. Most React projects start with a *src/* folder and one *src/App.js* file with an App component. At least that's what you get when you are using create-react-app. This App function component just renders something:

```
import React from 'react';

const App = () => {
  const title = 'React';

  return (
    <div>
      <h1>Hello {title}</h1>
    </div>
  );
}

export default App;
```

Eventually this component adds more features, it naturally grows in size, and needs to extract parts of it as standalone React components. Here we are extracting a React list component with another child component from the App component:

```
import React from 'react';

const list = [
  {
    id: 'a',
    firstname: 'Robin',
    lastname: 'Wieruch',
    year: 1988,
  },
  {
    id: 'b',
    firstname: 'Dave',
    lastname: 'Davidds',
    year: 1990,
  },
];

const App = () => <List list={list} />;

const List = ({ list }) => (
```

```
  <ul>
    {list.map(item => (

      <ListItem key={item.id} item={item} />
    ))}
  </ul>
);

const ListItem = ({ item }) => (
  <li>
    <div>{item.id}</div>
    <div>{item.firstname}</div>
    <div>{item.lastname}</div>
    <div>{item.year}</div>
  </li>
);
```

Whenever you start with a new React project, I tell people it's fine to have multiple components in one file. It's even fine in a larger React application, whenever one component is strictly tight to another one. However, in this scenario, eventually your one file will not be sufficient anymore for your React project. That's when we transition to step two.

MULTIPLE REACT FILES

The second step follows the rule: Multiple files to rule them all. Take for instance our previous App component with its List and ListItem components: Rather than having everything in one *src/App.js* file, we can split these components up into multiple files. You decide how far you want to take it here. For example, I would go with the following folder structure:

```
- src/
--- App.js
--- List.js
```

While the *src/List.js* file would have the implementation details of the List and ListItem components, it would only export the List component from the file as public API to this file:

```
const List = ({ list }) => (
  <ul>
    {list.map(item => (
      <ListItem key={item.id} item={item} />
    ))}
  </ul>
);
```

```
const ListItem = ({ item }) => (
  <li>
    <div>{item.id}</div>
    <div>{item.firstname}</div>
    <div>{item.lastname}</div>
    <div>{item.year}</div>
  </li>
);

export default List;
```

Then the *src/App.js* file can import the List component and use it:

```
import React from 'react';

import List from './List';

const list = [
  {
    id: 'a',
    firstname: 'Robin',
    lastname: 'Wieruch',
    year: 1988,
  },
  {
    id: 'b',
    firstname: 'Dave',
    lastname: 'Davidds',
    year: 1990,
  },
];

const App = () => <List list={list} />;
```

If you would take this one step further, you could also extract the ListItem component into its own file and let the List component import the ListItem component:

```
- src/
--- App.js
--- List.js
--- ListItem.js
```

However, as said before, this may take it too far, because so far the ListItem component is tightly coupled to the List component. Hence it would be okay to leave it in the *src/List.js* file. I follow the rule of thumb that whenever a React component becomes a reusable React component, I split it out as a standalone file, like we did with the List component, to make it

accessible for other React components.

FROM REACT FILES TO REACT FOLDERS

From here it becomes more interesting and more opinionated. Every React component grows in complexity eventually. Not only because more logic is added (e.g. more JSX with conditional rendering or React Hooks), but also because there are more technical concerns like styling and tests. A naive approach would be to add more files next to each React component. For example, let's say every React component has a test and a style file:

```
- src/
--- App.js
--- App.test.js
--- App.css
--- List.js
--- List.test.js
--- List.css
```

One can already see that this doesn't scale well, because with every additional component in the *src/* folder we will lose more sight of every individual component. That's why I like to have one folder for each React component:

```
- src/
--- App/
----- index.js
----- test.js
----- style.css
--- List/
----- index.js
----- test.js
----- style.css
```

The naming of these files is up to you. For example, *index.js* may become *component.js* or *test.js* may become *spec.js*. Moreover, if you are not using CSS but something like Styled Components, your file extension may change from *style.css* to *style.js* too. Once you get used to your naming convention, you can just search for "List index" or "App test" in your IDE for opening each file. If you collapse all component folders, you have a very concise and clear folder structure:

```
- src/
--- App/
```

```
--- List/
```

If there are more technical concerns for a component, for example you may want to extract custom hooks into their own file for certain components, you can scale this approach horizontally within the component folder:

```
- src/
--- App/
----- index.js
----- test.js
----- style.css
--- List/
----- index.js
----- test.js
----- style.css
----- hooks.js
```

If you decide to keep your *List/index.js* more lightweight by extracting the ListItem component in its own file, then you may want to try the following folder structure:

```
- src/
--- App/
----- index.js
----- test.js
----- style.css
--- List/
----- index.js
----- test.js
----- style.css
----- ListItem.js
```

Here again you can go one step further by giving the component its own folder with all other technical concerns like tests and styles:

```
- src/
--- App/
----- index.js
----- test.js
----- style.css
--- List/
----- index.js
----- test.js
----- style.css
----- ListItem/
------- index.js
------- test.js
```

```
------- style.css
```

Important: From here on you need to be careful to not nest too deeply your components into each other. My rule of thumb is that I am never nesting components more than two levels, so the List and ListItem folders would be alright, but the ListItem's folder shouldn't have another nested folder.

After all, if you are not going beyond midsize React projects, this is in my opinion the way to go to structure your React components. However, as I mentioned, this is highly opinionated and may not meet everyones taste.

TECHNICAL FOLDER SEPARATION

The next step will help you to structure midsize to large React applications. It separates features from components which are used by more than one component. Take the following folder structure as example:

```
- src/
--- components/
----- App/
------- index.js
------- test.js
------- style.css
----- List/
------- index.js
------- test.js
------- style.css
```

We group our previous React components into a new *components/* folder. This gives us another vertical layer for creating folders for other categories. For example, at some point you may have React Hooks that can be used by more than one component. So instead of coupling the hook tightly to a component, you can put the implementation of it in a dedicated folder which can be used by all React components:

```
- src/
--- components/
----- App/
------- index.js
------- test.js
------- style.css
----- List/
```

```
------- index.js
------- test.js
------- style.css
--- hooks/
----- useClickOutside/
------- index.js
----- useData/
------- index.js
```

This doesn't mean that all hooks should end up in this folder though. React Hooks which are still only used by one component should remain in the component's file or maybe in a separate *hooks.js* file in the component's folder. Only hooks that can be consumed by all React components end up in the *hooks/* folder.

The same strategy may apply if you are using React Context in your project which needs to be accessed globally by all your other files:

```
- src/
--- components/
----- App/
------- index.js
------- test.js
------- style.css
----- List/
------- index.js
------- test.js
------- style.css
--- hooks/
----- useClickOutside/
------- index.js
----- useData/
------- index.js
--- context/
----- Session/
------- index.js
```

From here, there may be other utilities which need to be accessible to your components and hooks. For miscellaneous utilities I usually create a *services/* folder. The name is up to you, but again it's the principal of making logic available to other code in our project which drives this technical separation:

```
- src/
--- components/
----- App/
------- index.js
------- test.js
------- style.css
```

```
        style.css
----- List/
------- index.js

------- test.js
------- style.css
--- hooks/
----- useClickOutside/
------- index.js
----- useData/
------- index.js
--- context/
----- Session/
------- index.js
--- services/
----- ErrorTracking/
------- index.js
------- test.js
----- Format/
------- Date/
--------- index.js
--------- test.js
------- Currency/
--------- index.js
--------- test.js
```

Take for instance the *Date/index.js* file. The implementation details may look like the following:

```
export const formatDateTime = (date) =>
  new Intl.DateTimeFormat('en-US', {
    year: 'numeric',
    month: 'numeric',
    day: 'numeric',
    hour: 'numeric',
    minute: 'numeric',
    second: 'numeric',
    hour12: false,
  }).format(date);

export const formatMonth = (date) =>
  new Intl.DateTimeFormat('en-US', {
    month: 'long',
  }).format(date);
```

Fortunately JavaScript's Intl API gives us excellent tools for date conversions. However, instead of using the API directly in my React components, I like to have a service for it, because only this way I can guarantee that my components have only a slim set of actively used date formatting options available for my application. Otherwise there would be lots of different date formats in a growing application.

Now it's possible to not only import each date formatting function individually:

```
import { formatMonth } from '../../services/format/date';

const month = formatMonth(new Date());
```

But also as a service, as an encapsulated module in other words, what I usually like to do:

```
import * as dateService from '../../services/format/date';

const month = dateService.formatMonth(new Date());
```

It may become difficult to import things with relative paths now. Therefore I always would opt-in Babel's Module Resolver for aliases. Afterward, your import may look like the following:


```
import * as dateService from '@format/date';

const month = dateService.formatMonth(new Date());
```

After all, I like this technical separation of concerns, because it gives every folder a dedicated purpose and it encourages sharing functionality across the application.

DOMAIN FOLDER SEPARATION

The last step will help you to structure large React applications. This happens when you find yourself with lots of subfolders in your technical separated folders. The example doesn't show the full extent, but I hope you get the point:

```
- src/
--- components/
----- App/
----- List/
----- Input/
----- Button/
----- Checkbox/
----- Profile/
----- Avatar/
----- MessageItem/
```

```
----- MessageList/
----- PaymentForm/
----- PaymentWizard/

----- ErrorMessage/
----- ErrorBoundary/
```

From here I would use the *components/* folder only for reusable components (e.g. UI components). Every other component should move to a domain centred folder. The names of the folders are again up to you:

```
- src/
--- domain/
----- User/
------- Profile/
------- Avatar/
----- Message/
------- MessageItem/
------- MessageList/
----- Payment/
------- PaymentForm/
------- PaymentWizard/
----- Error/
------- ErrorMessage/
------- ErrorBoundary/
--- components/
----- App/
----- List/
----- Input/
----- Button/
----- Checkbox/
```

If a PaymentForm needs access to a Button or Input, it imports it from the reusable UI components folder. If a MessageList component needs an abstract List component, it imports it as well. Furthermore, if a service from the previous step is tightly coupled to a domain, then move the service to the specific domain folder. The same may apply to other folders which were previously separated by technical concern:

```
- src/
--- domain/
----- User/
------- Profile/
------- Avatar/
----- Message/
------- MessageItem/
------- MessageList/
----- Payment/
------- PaymentForm/
```

```
------- PaymentWizard/
------- services/
--------- Currency/
----------- index.js
----------- test.js
----- Error/
------- ErrorMessage/
------- ErrorBoundary/
------- services/
--------- ErrorTracking/
----------- index.js
----------- test.js
--- components/
--- hooks/
--- context/
--- services/
----- Format/
------- Date/
--------- index.js
--------- test.js
```

Whether there should be an intermediate *services/* folder in each domain folder is up to you. You could also leave out the folder and put the *ErrorTracking/* folder directly into *Error/*. However, this may be confusing, because ErrorTracking should be marked somehow as a service and not as a React component. As alternative you could also use a kebab-case naming convention, which is better than PascalCase anyway, by going with *error-tracking/* instead of *ErrorTracking/*.

There is lots of room for your personal touch here. After all, this step is just about bringing the domains together which allows teams in your company to work on specific domains without having to touch files across the project.

. . .

Having all this written, I hope it helps one or the other person or team structuring their React project. Keep in mind that none of the shown approaches is set in stone. In contrast, I encourage you to apply your personal touch to it. Since every React project grows in size over time, most of the folder structures evolve very naturally as well. Hence the 5 step process to give you some guidance if things get out of hand.

Show Comments

KEEP READING ABOUT REACT

HOW TO DOCKER WITH CREATE-REACT-APP

Just recently I had to use Docker for my create-react-app web application development. Here I want to give you a brief walkthrough on how to achieve it. First of all, we need a React application...

HOW TO USECONTEXT IN REACT?

React's Function Components come with React Hooks these days. Not only can React Hooks be used for State in React but also for using React's Context in a more convenient way. This tutorial shows...

THE ROAD TO REACT

PORTFOLIO

Online Courses

Open Source

Tutorials

ABOUT

About me

What I use

How to work with me

How to support me

Contact Me Privacy & Terms