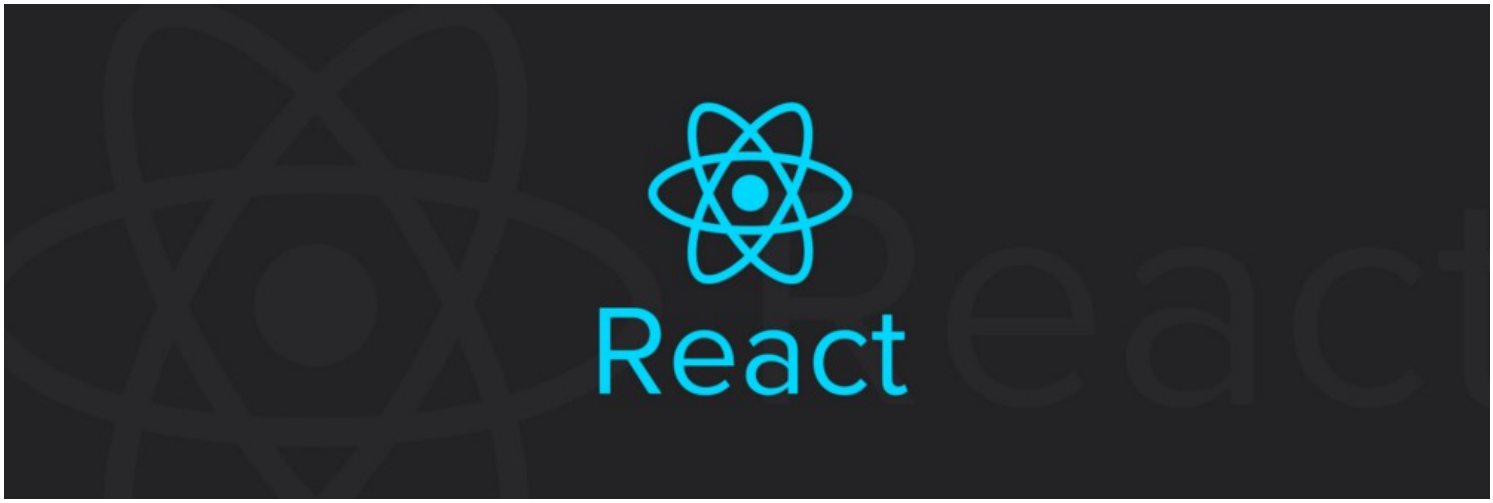
Alexis Mangin Follow
Javascript Application Developer
Apr 27, 2016 · 8 min read



How to better organize your React applications?

I've been working on very large web applications for the past few years, starting from ground zero and, with a dozen other developers, making them scale up to now be used by millions of people. And sometimes, if you didn't start with a good architecture, it can become difficult to keep your code organized.

Nathanael Beisiegel wrote an interesting article where he explains his strategy in the organization of large React applications, but I still wasn't completely satisfied by his approach. So I decided to spend some time to figure out what would be the best way to organize my future React projects.

Note: I use Redux files in all the examples of this article. If you don't know what Redux is, you can see the documentation here.

Note2: The examples use ReactJS, but you can use exactly the same structure for a React-Native application.

What are the challenges when you build an application?

This has happened or will happen to pretty much all developers over the course of their career:

- You build an application for a client with a team of a few developers, everything works very well all together.
- Your client requires new features, fine, you add them.
- Your client asks you to remove some features and add new ones, it starts to get complicated, you didn't think about that, but you make it work even though it's not perfect.
- Your client now wants you to change another feature, remove some others and add another one that wasn't expected. At this point, you grab the scotch tape and start patching some code. You are not very proud of this.
- 6 months later, after some other iterations, the code of the application gets really complicated to read and understand, everything looks like some Italian spaghetti pasta.

Until the day your client decides to create a new version of the application, with some fresh new code and features. In some cases, you end-up keeping complicated legacy code that lives with the new code, and this becomes even harder to maintain. And all of this happened because your app wasn't properly designed from the beginning.

. . .

When I started to learn React, I found a few very good articles explaining how to create Todo lists or very simple games. Those articles were very useful to understand the basics of React, but I quickly got to a point where I wasn't able to find much about how I could use React to build actual applications, something with a few dozens pages and hundreds of components.

After some research, I learned that every React boilerplate project on Github results to similar structures, they organize all the files by type.

This might look familiar to you:

```
/src
  /actions
    /notifications.js

 /components
    /Header
    /Footer
    /Notifications
      /index.js

  /containers
    /Home
    /Login
    /Notifications
      /index.js

  /images
    /logo.png

  /reducers
    /login.js
    /notifications.js

  /styles
    /app.scss
    /header.scss
    /home.scss
    /footer.scss
    /notifications.scss

  /utils

  index.js
```

This architecture might be okay to build your website or application, but I believe that it is not the best architecture.

When you organize your files by type, as your application grows, it often becomes difficult to maintain. By the time you realize this, it's too late and you will have to invest a lot of time and money to change everything, or to support what you have for the next few years.

The good thing with React is that you can structure your application in any way you like. You are not forced to follow a certain folder

structure, React is simply a javascript library.

What could be a better approach to organize your application?

For a couple of years I worked for a financial institution which used Ember as their main javascript framework to build all their new web applications. One interesting thing about Ember is the ability to structure your project by features, instead of by type. And this changes everything.

Pods in Ember are great but still limited, and I wanted something much more flexible. After a few experiments, trying to find what would be the best structure, I got to a point where I decided to group all related features together, and nest them as needed. This is what I use now:

```
/src
  /components
    /Button
    /Notifications
      /components
        /ButtonDismiss
          /images
          /locales
          /specs
          /index.js
          /styles.scss
      /index.js
      /styles.scss

  /data
    /users
      /actions.js
      /api.js
      /reducer.js

  /scenes
    /Home
      /components
        /ButtonLike
      /services
        /processData
      /index.js
      /styles.jsx

    /Sign
      /components
```

```
          /FormField
        /scenes
          /Login
          /Register
            /locales
            /specs
            /index.js
            /styles.scss

    /services
      /api
      /geolocation
      /session
        /actions.js
        /index.js
        /reducer.js

    index.js
    store.js
```

Each component, scene or service (a feature) has everything it needs to work on its own, such as its own styles, images, translations, set of actions as well as unit or integration tests. You can see a feature like an independent piece of code you will use in your app (a bit like *node modules*).

To work properly, they should follow these rules:

- A component can define nested components or services. It cannot use or define scenes.
- A scene can define nested components, scenes or services.
- A service can define nested services. It cannot use or define components or scenes.
- A data feature is standalone.
- Nested features can only use other entities that are defined by a parent entity.

Note: By parent feature, I mean a parent, grandparent, great-grandparent etc… You cannot use a feature that is a "cousin", this is not allowed. You will need to move it to a parent to use it.

Let's break this down.

Components

You all already know what a component is, but one important thing in this organization is the ability to nest a component into another component.

Components at the root level of the *components* folder are global and can be used anywhere in your application. But if you decide to define a new component inside another component (nesting), this new component can only be used by an entity that is a parent.

Why would you want to do that?

When you develop a large application, it happens quite often that you need to create a component that you definitively know you won't reuse anywhere else, but you need it. If you add it at the root level of your *components* folder, it will get lost with hundreds of components. Sure, you could categorize them, but when it's time to do some clean-up, you won't remember what they are all for or if they are still being used somewhere.

Although, if you define at the root level only the main components of your application, such as buttons, form fields, thumbnails, but also more complicated one like listComments, formComposer with their own children components, it gets much easier to find what you need.

Example:

```
/src
  /components
    /Button
      /index.js

    /Notifications
      /components
        /ButtonDismiss
          /index.js

      /actions.js
      /index.js
      /reducer.js
```

- *Button* can be used anywhere in your application.

- *Notifications* can also be used anywhere. This component defines a component *ButtonDismiss*. You cannot use *ButtonDismiss* anywhere else than in the *Notifications* component.
- *ButtonDismiss* uses *Button* internally, this is authorized because *Button* is defined at the root level of *components*.

Scenes

A scene is a page of your application. You can see a scene just like any component, but I like to separate them into their own folder.

If you use React-Router or React Native Router, you can import all your scenes in your main index.js file and setup your routes.

With the same principle components can be nested, you can also nest a scene into a scene, and also define components or services into a scene. You have to remember that if you decide to define something into a scene, you can only use it within the scene folder itself.

Example:

```
/src
  /scenes
    /Home
      /components
        /ButtonShare
          /index.js

      /index.js

    /Sign
      /components
        /ButtonHelp
          /index.js

      /scenes
        /Login
          /components
            /Form
              /index.js
            /ButtonFacebookLogin
              /index.js
          /index.js

        /Register
          /index.js
```

```
      /index.js
```

- *Home* has a component *ButtonShare*, it can only be used by the Home scene.
- *Sign* has a component *ButtonHelp*. This component can be used by *Login* or *Register* scenes, or by any components defined in those scenes.
- *Form* component uses *ButtonHelp* internally, this is authorized because *ButtonHelp* is defined by a parent.
- The *Register* scene cannot use any of the components defined in *Login*, but it can use the *ButtonHelp*.

Services

Not everything can be a component, and you will need to create independent modules that can be used by your components or scenes.

You can see a service like a self-contained module, and could eventually be reused in any of your applications. What it does is really up to you, from a simple set of utility methods to a very complex subset of your application.

```
/src
  /services
    /api
      /services
        /handleError
          /index.js
      /index.js
```

```
    /geolocation
    /session
      /actions.js
      /index.js
      /reducer.js
```

. . .

Edit (Nov 2016):
I have recently taken the decision to add an additional folder to this structure, called `data`. This change is more for organizational reason than functionality.

Data

A data entity is very similar to a service. You can see it as your bridge/an adapter between the server API and the client. You might not need this if your website or mobile app doesn't process content from an API.

It is in charge of most of the network calls your app will make, get and post content, and transform payloads as needed before being sent, or stored in the store of your app (such as Redux). In most projects, the data coming from an API is used by more than one scene or component, and needs to be available from anywhere.

. . .

Wrapping up

I've been working with this architecture for the past few months on a personal project built with React-Native, and I can tell you this saved me a lot of time. It's much more simpler to have all related entities grouped together, it makes things easier to work with.

This architecture is one of many other ways to organize your project, that's the way I like it now and I hope this will help you improve yours!

Feel free to add a comment below or contact me directly if you have any question, I'll be more than happy to help.

Edit (Nov 2016): I have been asked by many people to show case a working example of such a project. I recently published on Github a simple project that uses this architecture: https://github.com/alexmngn/react-native-authentication

Have fun!

More articles from me

- How to use Redux on highly scalable javascript applications?
- What are the main differences between ReactJS and React-Native?
- The essential boilerplate to authenticate users on your React-Native app

About me

Hi, I'm Alexis! I'm a full-stack developer who has been programming for over 14 years. I have always had a passion for technologies since I was young. I'm now specialized in Javascript development, I enjoy to spend countless hours learning and playing with new technologies and use them in my next projects.