×

Manujith Pallewatte

# How to Organize Your React + Redux Codebase

Manujith Pallewatte

Mar 3, 2020 • 14 Min read • 21,814 Views

Mar 3, 2020 • 14 Min read • 21,814 Views

Web Development React

## Introduction

React is one of the most unopinionated frontend frameworks in existence. From the selection of states, androuting to managing your code structure, React does not inherently provide any guidelines. Comparatively, Angular provides a much better insight into where and how the building blocks should be placed in the code. This puts React developers in a difficult position at the start of a project. Regardless of our experience, we all find it extremely difficult to formulate the perfect codebase structure at the beginning of a new project.

In general, React project structures are often iteratively evolved alongside the project's scope and complexity. When new libraries are added, such as Redux and React Router, the initial structure needs to be refactored to accommodate the added complexity. With pressure on deadlines for the project's completion, the refactoring gets stuck in backlog until the project is completely unmaintainable.

In this guide, we will explore several directory structures that are used in production-grade applications and analyze the pros and cons of each. It is important to keep in mind that no single structure universally fits every project. Depending on the project's size, scope, complexity, and future aspects, the most suitable structure varies. Thus, this guide can help as a starting point in choosing the correct initial structure so that future refactoring is minimal.

Let's use a model project to evaluate our different codebase

ere we have the following features:
rticles (or blog posts), categories, and users
2. A home page that shows a list of categories and a list of articles
3. A category page that shows category-specific information and articles

While the feature set looks simple at a glance, the actual codebase will span across multiple files and directories. We also assume that we use Redux to manage our application state.

## The Flat Structure

First, we'll explore the most common and easiest structure in use. I call it the *flat structure* since it has minimal directory nesting and is quite straightforward. It follows the principle of separating the logic and view in the root level and then adding Redux related directories to the mix.

```
└── src
    ├── actions
    │   ├── articleActions.js
    │   ├── categoryActions.js
    │   └── userActions.js
    ├── api
    │   ├── apiHandler.js
    │   ├── articleApi.js
    │   ├── categoryApi.js
    │   └── userApi.js
    ├── components
    │   ├── ArticleComponent.jsx
    │   ├── ArticleListComponent.jsx
    │   ├── CategoryComponent.jsx
    │   ├── CategoryPageComponent.jsx
    │   └── HomePageComponent.jsx
    ├── containers
    │   ├── ArticleContainer.js
    │   ├── CategoryPageContainer.js
    │   └── HomePageContainer.js
    ├── index.js
    ├── reducers
    │   ├── articleReducer.js
    │   ├── categoryReducer.js
    │   └── userReducer.js
    ├── routes.js
    ├── store.js
    └── utils
        └── authUtils.js
```

Directory functions, in brief, include the following:

- components - Contains all 'dumb' or presentational components, consisting only of HTML and styling.
- containers - Contains all corresponding components with logic in them. Each container will have one or more component depending on the view represented by the container. For example, `HomePageContainer` would have `ArticleListComponent` as well as

121

reducers - All Redux reducers
API - API connectivity related code. Handler usually involves setting up an API connector centrally with authentication and other necessary headers.
- Utils - Other logical codes that are not React specific. For example, `authUtils` would have functions to process the JWT token from the API to determine the user scopes.

store.js is simply the Redux store and the routes.js aggregates all routes together for easy access.

**Note:** Defining all routes in a single file has been a deprecated as a practice, according to new React Router docs. It promoted segregating routes into components for better readability. Check React Router Docs for a better understanding.

With the above understanding, let's analyze why and why not to use a flat structure.

### Pros

1. Easier readability with flat structures. You could easily do a filename search.
2. Developer onboarding is easy.

### Cons

1. Need to edit multiple files/directories to add a new function. Let's say we need to have a comment feature. We need to add `commentAction` to actions, `commentReducer` to reducers, `CommentComponent` to component, and `CommentContainer` to containers.
2. Redux state is everywhere. The actions, reducers, and sometimes types are in separate directories.
3. When the codebase grows, a lack of inner structure makes it hard to maintain. For example, at a glance, we could not see the components that are used by `HomePageContainer`.
4. Container-Component split doesn't make sense in certain instances, such as the pages.

With the above issues, we could do a slight improvement by introducing the `pages` directory as a way of providing some organization.

```
    └── src
        ├── actions
        │   ├── articleActions.js
        │   ├── categoryActions.js
        │   └── userActions.js
        ├── api
        │   ├── apiHandler.js
        │   ├── articleApi.js
        │   ├── categoryApi.js
        │   └── userApi.js
        ├── components
        │   └── ArticleComponent.jsx
        ├── containers
        │   └── ArticleContainer.js
        ├── index.js
        ├── pages
```

121

```
    │   │       └── CategoryPageComponent.jsx
    │   └── HomePage
    │       ├── components
    │       │   ├── ArticleListComponent.jsx
    │       │   ├── CategoryComponent.jsx
    │       │   └── HomePageComponent.jsx
    │       └── HomePageContainer.js
    ├── reducers
    │   ├── articleReducer.js
    │   ├── categoryReducer.js
    │   └── userReducer.js
    ├── routes.js
    ├── store.js
    └── utils
        └── authUtils.js
```

Now with the above improvement, the directory structure provides some context into the actual positioning of the various components in the app. At a glance, it is clear that `HomePageComponent`, `ArticleListComponent`, and `CategoryComponent` are part of the `HomePage`. As an important side effect, now the things that remain on the components and containers directory at the root level are the shared components that do not directly belong to any one page. So, we could go one step further and group them into a `common` directory.

```bash
└── src
    ├── actions
    │   └── ...
    ├── api
    │   └── ...
    ├── common
    │   ├── components
    │   │   └── ArticleComponent.jsx
    │   └── containers
    │       └── ArticleContainer.js
    ├── index.js
    ├── pages
    │   └── ...
    ├── reducers
    │   └── ...
    ├── routes.js
    ├── store.js
    └── utils
        └── ...
```

This looks much better. If your app does not has a huge application state, and is rather view- and logic-heavy, the above structure should work. It provides significant clarity and maintainability. But if your Redux code is also growing with the rest of the features, you will soon find that you need a better organization for the state as well.

## The View-State Split

The *view-state split* improves upon the previous structure to simply give a better organization to the state. It separates the view and logic-heavy components from the state component, but introduces

121

121

bash

```
└── src
    ├── api
    │   ├── apiHandler.js
    │   ├── articleApi.js
    │   ├── categoryApi.js
    │   └── userApi.js
    ├── common
    │   ├── components
    │   │   └── ArticleComponent.jsx
    │   └── containers
    │       └── ArticleContainer.js
    ├── index.js
    ├── pages
    │   ├── CategoryPage
    │   │   ├── CategoryPageContainer.js
    │   │   └── components
    │   │       └── CategoryPageComponent.jsx
    │   └── HomePage
    │       ├── components
    │       │   ├── ArticleListComponent.jsx
    │       │   ├── CategoryComponent.jsx
    │       │   └── HomePageComponent.jsx
    │       └── HomePageContainer.js
    ├── routes.js
    ├── state
    │   ├── article
    │   │   ├── articleActions.js
    │   │   └── articleReducer.js
    │   ├── category
    │   │   ├── categoryActions.js
    │   │   └── categoryReducer.js
    │   ├── middleware.js
    │   ├── store.js
    │   └── user
    │       ├── userActions.js
    │       └── userReducer.js
    └── utils
        └── authUtils.js
```

Changes in the above structure are simple. The state is now nested with one more level where actions and reducers of a particular application feature are grouped. With this, finding where the changes need to be done for a particular feature is visible at once. For example, if your API decides to send articles tags and now you want to show them in your Articles components, you first edit the `api`, then the `state`, and finally update the `ArticleComponent`.

### Pros

1. Adding new application features and maintaining current features is easy.
2. The state is well organized, no confusion on the placement.
3. All Redux codes are concentrated in one location, so refactoring is easy. For example, if you decide to use Redux Toolkit after a while, you know that you only need to make changes on the files inside `state` directory.

### Cons

1. View and state are separated. If your app has hundreds of features, finding state corresponding to a particular view is cumbersome.
2. No proper location for feature related, logic code. Anything that

121

`utils` directory, which again separates the portion of feature code rom feature component code.
3. Developer onboarding is not trivial.

While this structure has a few issues, it can be accommodated for the majority of project use cases. But if the project has a lot of moving parts and the development team is large, the above issues start to become blocking points—especially if the project is being developed by a distributed team (open source projects are prime examples). Then we need a better organization that allows developers to work on individual application features without disrupting the entire codebase.

## The Application Feature Split

I was first introduced to the *application feature-based split* through Node Best Practices by Yoni Goldberg. It is aimed at providing structure for nodejs projects, which are equally unopinionated. It provides a scalable model to overcome the common issues in using MVC pattern on node backends. In brief, it advises splitting directories by application features rather than code functions. For example, in our app, we have the following three features:

1. Article
2. Category
3. User

These are also known as domains. By splitting them as such, we could group all functional code related to an application feature inside a directory so that a developer can concentrate only on the particular directory. Let's explore how we could fit the pattern to our app:

bash
```
├── api
│   ├── apiHandler.js
│   ├── articleApi.js
│   ├── categoryApi.js
│   └── userApi.js
├── article
│   ├── Article.jsx
│   ├── ArticleList.jsx
│   └── state
│       ├── articleActions.js
│       └── articleReducer.js
├── category
│   ├── Category.js
│   └── state
│       ├── categoryActions.js
│       └── categoryReducer.js
├── category-page
│   └── CategoryPage.jsx
├── common
│   └── state
│       ├── commonActions.js
│       └── commonReducers.js
├── home-page
│   ├── HomePageContainer.js
│   └── HomePage.jsx
```

121

```
├── routes.js
├── store.js
└── user
    ├── authUtils.js
    └── state
        ├── userActions.js
        └── userReducer.js
```

In the above structure, the following changes are made:

1. Each application feature is kept in a separate directory
2. All views, logic, and state of a particular feature are grouped inside the corresponding directory (API can also be brought inside)
3. Containers and components are not split into two but rather aggregated. It was observed that splitting view and logic does not create a significant benefit unless the view is being reused by other logics. So `Article.js` is a combination of `ArticleContainer` and `ArticleComponent`.

**Pros**

1. Application features are separated.
2. The codebase has better scalability, maintainability, and readability.
3. Developers intuitively know from where to import feature specific functions. If you need to access article related actions, they are inside the article directory.

**Cons**

1. It's not very transparent on when and when not to separate a set of codes as an application feature. For example, comments can be part of the article feature since comments would only appear in articles. But comments could also be taken out as a separate directory.
2. Developer onboardinng is harder compared to the above other structures.

Even with the issues outlined above, this structure seems to be the most functional out of the options we discussed in the guide. After refactoring more than a few codebases using each of these options, I have fixed on it for any project with significant complexity. In a production-grade application, many smaller application features are required, including notifications, error feedback, centralized loading, auth handling, etc., and with a feature-based structure, adding and removing them is easy.

## Conclusion

Structuring your React + Redux codebase at the beginning is a confusing task for most frontend developers. Since the framework itself does not provide strict guidelines, we are forced to use trial-and-error based methods to find the best-suited structure for the project. In this guide, we explored a few common methods of organization, analyzing the pros and cons of each. While one structure does not fit all different project requirements, we can reference the above

The Application Feature S 121 uctures as starting points. This greatly minimizes refactoring effort 121 the future.

121

LEARN MORE

**SOLUTIONS**

Pluralsight Skills (/product/skills)

Pluralsight Flow (/product/flow)

Government (/industries/government)

Gift of Pluralsight (/gift-of-pluralsight)

View Pricing (/pricing)

Contact Sales (/product/contact-sales)

Skill up for free (/product/skills/free)

**PLATFORM**

Browse library (/browse)

Role IQ (/product/role-iq)

Skill IQ (/product/skill-iq)

Iris (/product/iris)