Thoughts on Folder Structure

There's been lots of discussion over what constitutes a good folder structure for a Redux application. The typical approaches generally fall into two categories: "file-type-first" (folders like `/reducers`, `/components`, etc), and "feature-first", also sometimes referred to as "pods", "domain-based", or "feature folders" (folders that each have all the file types for a given feature).

The original Redux examples use a "file-type-first" approach, but a lot of the recent articles and discussion have shown some convergence on the "feature-first" approach. There's tradeoffs either way - "file-type-first" makes it really easy to do something like pulling together all the reducers, but the code for a given feature can be scattered around, and vice versa for "feature-first".

My own current approach is mostly a "feature-first"-style approach. It's got some similarities to the approach described by Max Stoiber in his article How to Scale React Applications. The main differences are:

- I prefer to give my files full unique names, rather than having files named "actions.js" and "reducer.js" in every folder. This is mostly for ease of finding files and reading file names in editor tabs. I also would rather give component files unique names, rather than naming them `SomeComponent/index.js`.
- I've experimented with using `index.js` files to re-export functions and components upwards as a sort of "public API" for nested folders. I'm not entirely sold on this idea, as I'll explain in a minute.
- I put these folders grouped under a folder named `features`, rather than `containers`
- I personally use a mixture of thunks and sagas for my side effects. I use thunks for simple async and complex synchronous work, and sagas for more complex async or decoupled logic.
- I also prefer to use "absolute imports", such as `from`

`"features/someFeature/SomeComponent"`, rather than multi-level relative imports.

- I do have a `common` folder at the top level of the source, next to `features`. This is where I put truly generic reusable components, utility functions, and other logic that may get reused in multiple places throughout the application.

I'm not going to say that I'm 100% happy with my current approach. In particular, having a path like `"features/featureA/FeatureAList/FeatureAList.jsx"` gets kinda silly and repetitive. On the other hand, it's *very* clear exactly where that file is coming from.

I mentioned I'm not convinced on using `index.js` files to re-export "public APIs" for a folder. It can simplify import paths a bit, but may also mean that more files might get affected and rebuilt by a development server like Webpack due to the additional dependency entanglements. At one point it seemed like that the time needed to hot-reload changes in my app had gone up considerably, and I'm not sure how much was due to just having more code, and how much is due to a more entangled dependency tree causing more files to be affected. The proliferation of `index.js` files and re-exports can also be annoying.

I would say using `index.js` files is most useful if you have a parent component and several child components as separate files in the same folder, and you really only want to have the parent component get imported by other parts of the app. In that case, it does shorten the import path by just a bit, and also keeps the rest of the app kind of ignorant as to whether it's a single component file by itself or some kind of "compound component" that's actually several files in a folder together.

I'll freely admit that I'm still trying to figure things out myself. There's been quite a few good articles written about project structures, which I have collected in the Project Structure section of my React/Redux links list. I'd generally suggest using a "feature folder" approach, but Redux itself won't care at all. The important thing is that you, and anyone else working on your project, should be able to understand where the code for a given capability actually lives, and find that structure to be maintainable.

So, with those caveats, we’re at a point where we should start extracting files to a more maintainable structure, but there’s a couple tweaks we have to make first.