Share

Coding Style Guide - updated

AR Created by Alexandru-Doru Radulescu
Last updated yesterday at 1:52 AM • 4 min read • Analytics

Feature Folders Overview

In the src folder, code is organised using a hybrid approach: most folders are `feature` or `domain` based coupled with 2 function folders: `global` and `api`.

```
src
  |– api
      |– users
          |– hooks
              |– useUsersQuery.ts
          |– queries
              |– usersQuery.ts
          |– interfaces
              |– User.ts
          |– utils
              |– formatUser.ts

  |– users
      |– components
          |– UserListContainer.tsx
          |– UserList.tsx
          |– UserDetailsContainer.tsx
          |– UserDetails.tsx
      |– layout
          |– UserPageLayout.tsx
      |– pages
          |– UserListPage.tsx
          |– UserDetailsPage.tsx
      |– hooks
      |– interfaces
      |– enums
      |– constants
```

There are a number of benefits to organising code this way.

1. Code serving the same *function* (for example, a ProfileContainer and a PostContainer) doesn’t tend to change together. Code of the same *feature* (for example, a ProfileContainer and a ProfileView) does tend to change together. Therefore, it’s easier to make changes when code that needs to be changed is grouped together.
2. You can create strict module boundaries. This is helpful for reducing the complexity of interwoven dependencies.
3. A module can set up a public interface for other modules to use. This can reduce the issue of accidentally using code outside of its intended context, and makes the code more… well… modular.
4. The 2 function folders, `global` and `api` cover those elements which are reused across multiple features. Using strictly a feature-based folder system could bring circular dependencies and other confusing behaviour.
5. The 2 function folders are to be separated (vs `api` under `global`) as both of them tend to become very large yet quite independent from each-other.

Import Strategy

With Javascript / Typescript, to get all of the benefits presented above, you have to maintain strict index files, as well as be vigilant in how you import code from different places. Not adhering to the pattern laid out below can unfortunately result in weird behaviour and circular dependencies.

Index Files

Index files should only be used to export files that will be available in the public interface of the module. Anything that is not a part of the public interface of the module should not be exported in an index file. There should also not be any other code other than exports in an index file.

Exports in index files should follow these rules:

1. All exports should be relative paths, only to sibling files or folders.
2. If exporting interfaces or types, you have to use a wildcard *. This is because CRA forces the isolatedModule Typescript flag, which does some weird stuff when trying to re-export interfaces and types.
3. If exporting another index file from a sub-folder, use the wildcard *.
4. Sometimes we export multiple functions, classes, constants, etc from a single file while not all would be needed in the public interface, such as for tests or to be used in other sibling files. In this case, make sure to use **specific exports** in the index file and not the wildcard option

Here is an example showing these rules:

Structure:

```
src
  |- users
      |- components
          |- UserListContainer.tsx
          |- UserList.tsx
          |- index.ts
      |- utils
      |- interfaces
      |- index.ts
```

src/users/components/index.ts

```
export * from './UserListContainer.tsx
```

We do not export `UserList.tsx` as it is only needed and used under `UserListContainer`

src/posts/index.ts

```
export * from './components'
export * from './interfaces'
```

We do not export `utils` as it is only used within this feature-domain.

File Imports

.

When importing code from other files, there are some rules that need to be followed to prevent accidental circular references, as well as keep code modular and portable.

1. Don't use any default exports, only named exports. (The one exception to this is React components that are used in the dynamic imports for code splitting, which have to be default exports. The example of this is our Page components.)
2. When importing code from the *same* feature folder, use relative paths and import directly from files, not from index files. (Importing from the root index of the same feature folder is one of the main culprits of circular references.)
3. When importing code from a *different* feature folder, use absolute paths and only import from its root index file. (The root index file will provide its public interface.)
4. When importing an asset (like an image), use an absolute path directly to the file.

There are also some standards in place for organising imports, just to keep code readable and consistent: (Imports should be made into "groups" with a new line in between them)

1. 3rd-party imports
2. Imports from external modules (should all be absolute paths)
3. Imports from the same module (should all be relative paths)
4. Assets

Here is an example showing these rules:

Structure:

```
src
  |- api
      |- users
          |- hooks
              |- useUsersQuery.ts
          |- queries
              |- usersQuery.ts
          |- interfaces
              |- User.ts

  |- users
      |- components
          |- UserListContainer.tsx
          |- UserList.tsx
      |- interfaces
          |- userCategories.tsx
```

UserListContainer

*src/users/components/*UserListContainer.*tsx*

```
import React from 'react' // 3rd party imports first

import { useUsersQuery } from 'src/api' // External modules second

import { UserList } from './UserList.tsx' // Same modules 3rd
import { userCategories } from './interfaces'

export const UserListContainer ...
```

Folders vs Files

For each `feature/domain` folder in our code, some areas must follow clearly the folder structure defined above, including `/components`, `/layout`, `/pages` folders:

- individual files for individual components, features
- `index.ts` file exporting the relevant features, classes, etc up.

Some other parts can be smaller or larger, depending on the feature, including:

- interfaces
- enums
- constants
- utils

For these, the following rules apply:

- If there is only a small number of simple functions/interfaces/constants/enums then a single file can be used, such as `constants.ts` or `enum.ts`
- If or when there is a large number of constructs in one such file, then a folder structure must be adopted with individual files for each construct and an \`index.ts\` file for exporting.