An opinionated basic Next.JS files and directories structure

Pablo A. Del Valle H. May 20, 2020 · 5 min read

Next.JS is fairly loose in how you can organize your files and directories; if you establish a good structure from the start-up, your subsequent work will be easier, faster, predictable, and scalable

Photo by Alev Takil on Unsplash

As far as the required files and directories you must create to start a new project, **Next.JS is not very strict** (as opposed to, say Ember).

You can fairly say that at the minimum you'll need a "**pages/**" directory and "**index.js(x)**" file within it. If you were to create overrides for your app or document then you have an "**_app.js(x)**" and a "**_document.js(x)**" files also inside the "**pages/**" folder. The "**public/**" directory is another folder

that Next.JS looks into for static files to be emitted into the final build directory. Other than these files and directories, everything goes.

Here I share my **personal opinion** of a 'files and directories structure' that works for me. Having said that, there are as many approaches as there are developers, and you can choose to **follow a path or combine several avenues of structuring your project**.

Rules: what your files and directories structure should strive to be

The rules of thumb are:

- **Comfortable**: Establish an environment that makes you **comfortable**. Avoid the importing spaghetti; try the recently released Absolute imports and Aliases.
- **Dynamic**: Your setup should **not slow you down** when you work.
- **Predictable**: It should feel **predictable**, that is to say, when you are looking for a file, **you can anticipate** where that file is, even if it was written by a coworker.

- **Flexible**: Even when I label this structure as “opinionated”, it can **change and adapt** to unexpected situations and be ready for improvements. In a year from now, this structure can change. It needs to be **flexible**.
- **Scalable**: Following up on the previous item, the structure should allow itself to **grow and scale in a safe way**. Adding components, pages, or libraries should not disturb the rest of the codebase.
- **Separate concerns:** Do not mix roles. A module (or class, or file or component) should only deal with one single interest. This follows conventions of abstraction and modularity.

Basic structure

```
/root
  \_ /.next/
  \_ /components/
      \_ Button/
          \_ button.spec.jsx
          \_ button.styles.jsx
          \_ index.jsx
  \_ /constants/
      \_ theme.js
```

```
    \_ page.js
\_ /contexts/
    \_ Locale/
       \_ index.js
    \_ Page/
       \_ index.js
\_ /pages/
    \_ _app.jsx
    \_ _document.jsx
    \_ about.jsx
    \_ index.jsx
\_ /providers/
    \_ Locale/
       \_ index.js
    \_ Page/
       \_ index.js
\_ /public/
    \_ favicon.ico
    \_ header.png
\_ /redux/
    \_ actions/
       \_ users/
          \_ index.js
       \_ products/
          \_ index.js
    \_ reducers/
       \_ users/
          \_ index.js
       \_ products/
          \_ index.js
    \_ store/
       \_ index.js
    \_ types/
       \_ index.js
\_ /shared/
    \_ jsons/
        \_ users.json
    \_ libs/
        \_ locale.js
    \_ styles/
```

```
        \_ global.css
\_ /widgets/
    \_ PageHeader/
        \_ index.jsx
\
\_ .eslintignore
\_ .eslintrc
\_ .env
\_ babel.config.js
\_ Dockerfile
\_ jest.config.js
\_ next.config.js
\_ package.json
\_ README.md
```

Standard directories

Most Next.JS apps (and React apps for that matter) will have these, or at least a set, of directories:

- **Pages, "pages/"**: You'll need to have a "pages/" directory, which behaves as a sort of one-to-one static router for your site. Each page will match a .js(x) file.
- **Components, "components/"**: A repository with all your individual components. Each in an individual directory. Meaning that, if you have a

Button component, you'll create a "**components/Button/index.js(x)**".

- **Store, "redux/"**: Let's assume you are connecting to a Redux store (it can easily be extrapolated to any other store software). In the root, I create a redux directory. In it, 4 more directories:
- **Store/Actions, "redux/actions/"**: For instance, "users" actions will prompt this structure: "**redux/store/actions/users/index.js**".
- **Store/Reducers, "redux/reducers/"**: Following up the previous example, for "users": "**redux/store/reducers/users/index.js**".
- **Store/Types, "redux/types/"**: Depending on the size of your project you can have a single file or a file per role. However, in most cases, I just create a single "**redux/store/types/index.js**".
- **Store, "redux/store/"**: This would be the actual store to be passed to your app: "**redux/store/index.js**".
- **Widgets, "widgets/"**: A widget will essentially encapsulate several components, they essentially work as containers. A Page Header widget could be created in this file: "**widgets/PageHeader/index.js(x)**".
- **Constants, "constants/"**: Regularly I prefer to enclose modules in separate directories, this is to keep related files in the same folder (tests, styles, storybook files) however since constants usually do not need

tests, styles or additional files, I just create a file per constant group in this directory, such as "**constants/themes.js**" or "**constants/ui.js**".

- **Static files, "public/"**: Any file placed inside this directory will be transferred to the root of the build, this usually works for static files, such as images, text files, etc.

Global resources

A few files might be used across the board, scripts or libraries. In other words globally.

- **Contexts, "contexts/"**: A React Context that may be accessed globally, for instance: "**contexts/Locale/index.js**".
- **Providers, "providers/"**: Similarly to Contexts, React Providers can also be accessed globally, especially by containers, such as in widgets or the full App too. For example, a Page provider would be created here: "**providers/Page/index.js(x)**".
- **Shared resources, "shared/"**: this folder will hold other resources, such as JSON files, CSS stylesheets, etc. Shared libraries can also be

stored in this directory in "**shared/libs/***".

Root directory files

At the root of the directory, there are a few configuration and setup files such as:

- **.eslintignore** and **.eslintrc**
- **Dockerfile**
- **.env** files
- **babel.config.js**
- **jest.config.js**
- **next.config.js**
- **package.json**
- **README.md**

What's Next and Further Reading

Next.Js provides great opportunities to expand and empower a React application. Please read further these articles on the series:

New features introduced in Next.JS 9.4 and real-world applications

Support for Environment variables, built-in Fetch support, integrated Web Vitals analysis, improved Import of packages

medium.com

Configuring Jest for Next.JS (React) and Babel from scratch

Construct your Jest, Babel and package.json configuration files for a working Next.JS environment

medium.com

Fetching and hydrating a Next.JS app using `getServerSideProps` and `getStaticProps`

Learn how and when to use `getStaticProps`, `getServerSideProps` and `getInitialProps` in a Next.JS project

medium.com

Dockerize your Next.JS application

Learn to use Docker for packaging your Next.JS app for testing or deployment for staging or production

medium.com

Further reading:

- Next.JS documentation

Thanks for reading!

Find me on LinkedIn, Medium, GitHub.

Nextjs React Architecture Design Components JavaScript