# Opinionated Guide to React

It Depends

Sara Vieira

# Contents

### 0.1 Table of Contents

# 1 The Book

This Book is not supposed to ever serve as a teaching mechanism for react but more of a way to see react from the eyes of someone who has been using it for years and got sick of the “it depends”. All I will show you here is things either I use(d) or things developers I trust have used.

Also opinions will be shared that you may agree or not but to show the options and point of view is my objective with this book.

## 1.1 What will you learn?

Probably not how to use react from the basics but a clearer picture of how bigger react apps work, a bit of the tools they use, the ups and downs, their structure and also some knowledge on how to use these tools yourself. We will start with some things like folder and name structure and then go into packages, starter kits and many more.

## 1.2 Folder/ File Structuring

In this chapter we will be talking about structure, in simple terms how I usually structure applications in terms of folder position, file exports and some other small tidbits.

## 2 Folder

In apps and websites I have built with react I tend to have a similar structure that seems to work and that looks like so

```
.
├── src
|   ├── index.js
|   ├── components/
|   ├──── button/
|   ├────── index.js
|   ├────── elements.(js/css)
|   ├── pages/
|   ├──── homepage/
|   ├──── screens/
|   ├────── hero/
|   ├────── index.js
|   ├────── elements.(js/css)
|   ├──── index.js
|   ├──── elements.(js/css)
|   ├── index.js
|   ├── utils/
|   ├──── date.js
|   ├── assets/
|   ├──── icons/
|   └────  images/
```

In the core I have 4 main folders:

- `components` - This is where components used by more than one page or module get placed. These things usually don't quite belong in a design system. One example can be a `SaveButton` this will be an extension of the button with some differences that will be used in a lot of places. If no design system is in place basically anything that's used by more than one page or component like an `Alert`.
- `pages`- This is where your main pages will stay, this will have an `index.js` file that is where your route file be placed and where all these pages imported will be. Usually within a page you can have multiple sections like a hero, this won't be used anywhere else, but it's a crucial part

and should have both an `index.js` and a styles file so we can put this in a folder as well to minimize the size of our files, while also making it easier to find things.

- `assets` - This folder will contain all images and icons. I usually have both, so I find it easier to divide the folders since most of the times my icons will be in SVG and these will be translated into JSX and end up being also `JavaScript` files at their core.
- `utils` - This is where your overly complicated functions go to. This is kind of like hiding the shame but in a calculated way. Let's say you need to transform dates in a component and its a pretty heavy function. In my opinion this should be its own file maybe generalized to dates so it can export several functions for date manipulation - trust me, there is always date manipulation.

## 2.1 File naming

I always try to name my files `index.js` and let the folder name do the talking. This will allow me to have more freedom in the composition of that component or page, as more files may be added, and that way they all stay concise in that folder. So I may have something like:

`src/components/Alert/index.js`

Even though it's the index file, the way module resolution works in JavaScript you don't need to specify `index.js` so you can just import like you would a file:

```
import Alert from "./components/Alert";
```

This will look for the file and then if it doesn't find for the folder and an `index` file in that folder so don't worry about more typing.

## 2.2 Exporting Components

For many years, I used the good old `export default` even though react always yelled at me, I would export a components like so:

```
import React from "react";

export default ({ onClick }) => (
  <>
    <h1>Sup?</h1>
    <button onClick={onClick}>I am a button</button>
```

```
  </>
);
```

One of the drawbacks with this is how more INCREDIBLY hard it becomes to find anything in the devTools. For VSCode users, it also removes the autocomplete because you never named the component.

In the last years I have always exported the same component like so:

```
import React from "react";

const ButtonWrapper = ({ onClick }) => (
  <>
    <h1>Sup?</h1>
    <button onClick={onClick}>I am a button</button>
  </>
);

export default ButtonWrapper;
```

This has two main advantages over default:

- You can now see in the react devtools what the component name is making it for easier debugging and just overall cleaning of the devtools.
- Autocomplete in VSCode, even without TypeScript, VSCode is pretty smart to do a run down of your folders and see the components name and see what is what you want. It's not bullet proof without TypeScript but honestly, it's pretty impressive and more than enough for me to be productive

## 2.3 TypeScript

Let's talk about the elephant in the room *TypeScript*.

**Do you need it to build a react app?** Oh god no, even less in the start, I think TypeScript is one of those "pluck it in when you need it" type of tool. In the start it's definitely not needed, maybe your app will start feeling very prone to errors and it's a good idea but not in the start. Never in the start.

**Should I use it for my marketing page?** Honestly…why? It will add way more complexity without improving gains a lot, you don't have state, you don't have complicated things, it's a website and not an app, so in all honesty there is no need for something as heavy as that.

**Fine, when do I need it?** When honestly you can't manage state and you have no idea wtf is what anymore, and how many `isLoggedIn` states you have in your store, you need TypeScript when you would rather cry than manage state.

**What things should have Redux?** In my opinion, state and design systems are things where TypeScript is quite handy because these are things you use all the time and need to know what props you want to use, what they take and all of those fancy things.

**But what does a TypeScript react component look like?** Lets do one with state and props, let's take this simple components and make it all TypeScript compatible:

```
import React from "react";
import { AlertWrapper, Message, CloseButton } from "./elements";

const Alert = ({ onClose, type, children, neverClose }) => {
  const [open, setOpen] = useState(true);

  return open ? (
    <AlertWrapper type={type}>
      {!neverClose ? (
        <CloseButton
          onClick={e => {
            setOpen(false);
            onClose && onClose(e);
          }}
        >
          x
        </CloseButton>
      ) : null}
      <Message>{children}</Message>
    </AlertWrapper>
  ) : null;
};
```

In this case we have some props we may want to type, and looking at them we have:

- `onClose` - An optional function that returns nothing and takes the event to the parent component.
- `type` - The type of alert this is and in our case it can either be: `success`, `error` or `warning`.
- `children` - Any react nodes we want to pass as the message

- `neverClose` - An optional boolean attribute to check if want the user to be able to close it.

So let's transfer this into an interface in TypeScript:

```tsx
interface Props {
  onClick?: (event: React.MouseEvent) => void;
  type: "success" | "error" | "warning";
  children: React.ReactNode;
  neverClose?: boolean;
}
```

To apply this to the react component we do as such:

```tsx
import React from "react";
import { AlertWrapper, Message, CloseButton } from "./elements";

interface Props {
  onClick?: (event: React.MouseEvent) => void;
  type: "success" | "error" | "warning";
  children: React.ReactNode;
  neverClose?: boolean;
}

const Alert = ({ onClose, type, children, neverClose }: Props) => {
  const [open, setOpen] = useState(true);

  return open ? (
    <AlertWrapper type={type}>
      {!neverClose ? (
        <CloseButton
          onClick={e => {
            setOpen(false);
            onClose && onClose(e);
          }}
        >
          x
        </CloseButton>
      ) : null}
      <Message>{children}</Message>
```

```
    </AlertWrapper>
  ) : null;
};
```

There are many other types, but in general typing react components like these is not a though thing to do, but sometimes doing this will lead to more work like transpiling or debugging edge cases that is not always worth it.

I have very strong opinions on TypeScript as I think it creates a barrier for people to get into web development in a way as open and accessible as I did, and most of the times for no reason. I would say 50% or more of apps don't need TypeScript at all, more than 80% don't need TypeScript all over their pages and 100% don't need TypeScript in a marketing page with no state management.

If you want your designer to make changes, add JSX, fix CSS and overall do some code, please avoid using TypeScript and it's not something that they need to learn and consider if you yourself need it when making an open source project or if it's creating a barrier of entrance for people who want to help.

# 3 Project Starters

One of the main issues that existed when React started was that it was incredibly hard to get started, you had to mess with webpack and do a lot of really hard things just to get an hello world up and running. In the last couple of years that has gotten better we now have a lot of tools to help us get started writing React projects in no time but on the other hand we have so many and so good that sometimes its harder to know what starter to use. I will go through the three most used starters used right now to create different type of projects in react.

## 3.1 Create React App

**Link: https://create-react-app.dev/**

Create React App is the first and most famous one, its made and maintained by the React team themselves so you know all the choices are have the team stamp of approval.

Without a doubt CRA (create react app short name) is the fastest way to get started and have some react code show up on your page but the main issue is that CRA is not very extensible as you don't have access to the webpack or even babel config so its a tradeoff you must know from the start as sometimes the only way to add something is to `eject` and that will leave you with *ALL* the webpack config and no way to update your react scripts.

Let's look at the pros and cons:

Props:

- Quick to get started
- Supports most CSS preprocessors
- Supports PWA
- Easily updatable with new features
- Support for SVG as React Components

Cons:

- Not a lot of flexibility when it comes to changing the way it handles file types
- No Server Side render Support
- No decisions from the react team in terms of app building so all the router, state management etc will be up to you.

In my opinion create react app is a good starting point but if your application grows big enough it will also get out of hand and you will end up with a lot of webpack to handle either way.

## 3.2 Next

**Link: https://nextjs.org/**

Next is great, it comes prepared for a lot of things in your application, more than a starting point it's a guide to making server side rendered applications in react as that is supported out of the box and one of the biggest selling points of next.

It also makes some decisions for you like routing and styling but gives you the room to make your own and even extend their scripts by changing the babel configuration to support more things you may need.

Let's say you don't want to use their styled options but prefer to use styled components, in that case you can extend the babel config by adding the plugin like in a new `.babelrc` file

```
{
  "presets": ["next/babel"],
  "plugins": ["babel-plugin-styled-components"]
}
```

Make sure to leave the `next/babel` plugin as that adds a lot of functionality and a lot of babel presets and plugins you can read more about it here.

A big advantage of next is also that it has a simple way to get started with the cli, to make a new project you can simply run:

```
npx create-next-app
```

So let's look at the pros and cons of next in my opinion:

Pros:

- Server side render support
- Amazing docs with lessons to follow
- Production grade
- Customizing options

Cons:

- Steep learning curve, mostly for being SSR things are just harder.
- Harder to leave if it the `next` is not the best for your project
- **ZEIT PROPAGANDA ASK IVES ABOUT ZEIT NEXT THING**

## 3.3 Gatsby

**Link: https://www.gatsbyjs.org/**

I'll be honest, gatsby is basically my create react app, its what I use for basically everything, even the website for this book is made in gatsby just because.

Gatsby started a project by Kyle Mathews to create blogs but it grew into so much more and now it's a VC backed company and the product is way more than a blog creator, you can pull data from anywhere to make static sites.

You may ask what is so good about static sites? One of the main benefits is without a doubt the SEO, the app is HTML so everything gets read by google to better rank your site, another big benefit is the deployment, static HTML sites are waaaaay easier to deploy than server side applications.

Getting started is also quite easy as `gatsby` also has a CLI:

```
npx gatsby new gatsby-site
```

This will give you a new site with the default template, this one comes with some plugins and also two pages so you can get an idea how the routing works inside of gatsby.

The main strength of gatsby is to be able to source from basically anywhere and create HTML files from it with GraphQL so it is needed to know some GraphQL in order to get a full grasp of it's potential.

Let's start with a simple example using a plugin that will get data from https://randomuser.me/ and display it in our page.

First installing the plugin:

```
yarn add gatsby-source-randomuser
```

Now that we installed the plugin we have to add to our list of plugins that is located on our `gatsby-config.js`, in there we can add the plugin and tell it to get 25 people:

```
{
  resolve: "gatsby-source-randomuser",
  options: {
    results: 25,
  },
},
```

Now that we have this our site can be fed this date from GraphQL and to do this we need to add a query to our index page:

```
export const Query = graphql`
  query Users {
    allRandomUser {
      edges {
        node {
          id
          name {
            first
            last
          }
          picture {
            thumbnail
          }
        }
      }
    }
  }
`;
```

If you want to test this query and all of the graphql things you will do you do that at `http://localhost:8000/___` , this will give you a graphql playground to test your queries.

After this is done we can now pass this data to our component and render our humans:

```
const IndexPage = ({ data }) => (
  <Layout>
    <SEO title="Home" />
    <h1>Hi peeps :wave:</h1>
    <ul>
      {data.allRandomUser.edges.map(({ node }) => (
        <li>
          <img src={node.picture.thumbnail} alt={node.name.first} />
          {node.name.first} {node.name.last}
        </li>
      ))}
    </ul>
  </Layout>
);
```

There is waaay more you can do with this example, including creating a page for every user but as an example of the power I feel we leave in a good spot. For the code and preview you can go to CodeSandbox.

Now that you have an idea how much I like gatsby let's go over some pros and cons of using it.

Pros:

- Amazing documentation
- Flexibility to tweak gatsby internals to your needs
- Export to HTML
- Amazing community and team behind it
- Focus on performance and accessability

Cons:

- Steep learning curve for any advanced things
- Knowledge of GraphQL required to get started
- Not everything can be static

# 4 Packages

If you want something there is almost certainly a react package for that and in my opinion that is both a strength and a downfall when combined with the fact that react itself only provides your with a view layer and the rest is supposed to be figured out by yourself.

You may ask how I can see this as a downfall since the fact that there are so many packages and people out there making tools should only be an advantage? It is an advantage when there is direction and recommended ways, something React refuses to create so to anyone getting started it all looks like a sea of sameness. This part is a bit for people who feel the same way, like searching for a router is like going into a voyage. Here I will go through the packages I use for some parts of an app and how to use the basics of them so that you can take everything I say and then make a formed decision if you will use the same thing or continue on the quest.

Let's start.

## 4.1 Routing

**Winner: Reach Router**

For years I used React Router, the syntax and some API choices weren't really my type of coffee but it was the most used and supported so I kept using it until Reach Router came around that got all the little things I didn't really like about React Router and made them go away.

First thing I like is how to define routes, it gets rid of the `Route` component replacing it with the actual component handling that like so:

```
import { render } from "react-dom";
import React from "react";
import { Router, Link } from "@reach/router";

let Home = () => <div>Home</div>;
let User = () => <div>User</div>;

render(
  <Router>
    <Home path="/" />
    <User path="user/:id" />
  </Router>,
  document.getElementById("root")
);
```

Now let's say you wanna get that user id to fetch from database or from an api, you can just get it from the direct props instead of getting inside objects of objects:

```
import { render } from "react-dom";
import React from "react";
import { Router, Link } from "@reach/router";

let Home = () => <div>Home</div>;
let User = ({ id }) => <div>User: {id}</div>;

render(
  <Router>
    <Home path="/" />
    <User path="user/:id" />
  </Router>,
  document.getElementById("root")
);
```

Adding links is also has a pretty good user Developer experience :

```
import { render } from "react-dom";
import React from "react";
import { Router, Link } from "@reach/router";

const Home = () => (
  <div>
    <Link to="/user/random">Go to Random user</Link>
  </div>
);
const User = ({ id }) => <div>User: {id}</div>;

render(
  <Router>
    <Home path="/" />
    <User path="user/:id" />
  </Router>,
  document.getElementById("root")
);
```

One thing that really took me to Reach Router was the `navigate` function, this simple but powerful functions allows you navigate somewhere in your app without complication or components like so:

```
import { render } from "react-dom";
import React, { useState } from "react";
import { Router, Link, navigate } from "@reach/router";

const Home = () => {
  const [user, setUser] = useState("");
  return (
    <div>
      <Link to="user/random">Go to Random user</Link>
      <input value={user} onChange={e => setUser(e.target.value)} />
      <button disabled={!user} onClick={() => navigate(`/user/${user}`)}>
        Go to that user
      </button>
    </div>
  );
};
const User = ({ id }) => <div>User: {id}</div>;

render(
  <Router>
    <Home path="/" />
    <User path="user/:id" />
  </Router>,
  document.getElementById("root")
);
```

As soon as you fill in the input and also click the button you will be redirected to the new page with the value typed with no fuss.

Let's finish our "app" by adding a a 404 page so that our user can know he got lost:

```
import { render } from "react-dom";
import React, { useState } from "react";
import { Router, Link, navigate } from "@reach/router";

const Home = () => {
  const [user, setUser] = useState("");
  return (
    <div>
      <Link to="user/random">Go to Random user</Link>
      <input value={user} onChange={e => setUser(e.target.value)} />
      <button disabled={!user} onClick={() => navigate(`/user/${user}`)}>
```

```
        Go to that user
      </button>
    </div>
  );
};
const User = ({ id }) => <div>User: {id}</div>;
const NotFound = () => <p>Sorry, nothing here</p>;

render(
  <Router>
    <Home path="/" />
    <User path="user/:id" />
    <NotFound default />
  </Router>,
  document.getElementById("root")
);
```

We created a bunch of functionality in 27 lines. I think that's where Reach Router shines, it's simple experience as a developer and it's way of making himself more and more robust if needed.

Link to CodeSandbox

### 4.2 State Management

**Winner: Overmind**

### 4.3 Animation

**Winner: Framer Motion**

### 4.4 Styling

**Winner: Styled Components**