Sara Vieira

THE OPINIONATED GUIDE TO REACT

This book is dedicated to my wife Alex.

She gave the strength to finish it and relentlessly fixed my terrible grammar.

Table of Contents

- Create React App
- Next
- Gatsby

Packages

- Routing
- State Management
- Animation
- Styling
  - What about Emotion? Why not use that?
- Forms
- Dates
- GraphQL
- UI Toolkits
  - Unstyled
  - Styled
- Component Playgrounds
- Icons

The Hooks

- useEffect
- useLayoutEffect

- useState
- useContext
- useReducer
- useRef
- useMemo
- useCallback
- Make your own hook

Performance

- Add keys to your list elements
- Make small components
- React.memo
- Avoid mounting and unmounting components
- Virtualize long lists
- Take advantage of useMemo and useCallback

Deployments

- CRA
  - Netlify
  - Vercel
- Gatsby
- Next

> *You can read the book online at ReactGuide*

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 are things that either I use(d) or things developers I trust have used.

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

I hope you enjoy this book and have some fun writing web apps and making some amazing buttons.

About the author

I am never the best person to talk about myself, so I asked twitter, some of the answers were good, some bad and some just funny.

One of them was just amazing, and so it will be the one I use here:

> *It's my great honor to introduce you to my (spiritual) little sister, Sara Vieira. While I cannot paint a **complete** portrait of her in precious few words, I will attempt to highlight what I admire most about her.*
>
> *Despite her insistence otherwise, she's one of the smartest people you are ever going to meet. Sara possesses a profoundly unique talent to take difficult or mundane topics and infuse them with clarity and humor. When you meet her, you cannot help but be inspired and delighted by her affable demeanor and sharp wit. Sara is a fierce advocate for the underrepresented, the undervalued, and those would otherwise lack the same opportunities as her. She speaks English better than most Americans (present company included.) And while Sara quick to make fun of her native Portugal, she's even quicker to tell you to **** off for disparaging her pátria. She's a rare and respected voice for mental health in the tech industry that advocates for mental health and inclusivity. During the day, Sara works at https://codesandbox.io where she's a Berlin-base frontend engineer. When not saving the world one clever meme at a time, Sara is a renowned public speaker, an accomplished conference organizer, and an amazing spouse to her wife.*

> *And I'm lucky enough to call her my friend. I hope you enjoy her book; I know I will. You are supporting one of my favorite humans and doing yourself a huge favor by absorbing a bit of her wisdom. I hope you get to meet her some day; you too can see she's pretty indescribable.*
>
> *Brian Holt*
>
> *Senior Program Manager*
>
> *Visual Studio Code & JavaScript on Azure*
>
> *Microsoft*

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, also some knowledge on how to use these tools yourself and a lot of hooks. We will start with some things like folder and name structure and then go into packages, starter kits and much more.

What I hope is not that you follow everything I say blindly but that you consider these options and these packages as something to make use of in the future, my hope is that by the end the React world becomes easier to navigate.

Folder/ File Structuring

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

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/
| ├──── hooks/
```

```
| ├────── useLocalStorage.js
| ├─── date.js
| ├── assets/
| ├─── icons/
| └─── images/
```

In the core, I have four 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 could be a `SaveButton`: this will be an extension of the `Button` with some differences that will be used in a lot of places, but unlike the `Button` it doesn't quite belong in a design system. If no design system is in place, basically anything that's used by more than one page or component like an `Alert` should go here.
- `pages` - This is where your main pages will be. This folder will have an `index.js`, the place from which your routes file will import all the pages in your app. Usually, within a page you can have multiple sections like a hero. This hero component won't be used anywhere else, but it's a crucial part of this particular page. It should have both its own `index.js` and a styles file so we can put this in a separate folder to minimize the size of our files, while also making it easier to find things.
- `assets` - The assets folder will contain all your images and icons. I usually have both, so I find it easier to divide the folders: most of the time, my icons will be in SVG, and these get translated into `JSX`, so end up also being `JavaScript` files at their core. I did that so many times in random websites I even made a desktop application for it that you can download at svg-jsx.netlify.com and a Figma plugin to do the exact same thing and if you use Figma the plugin can be found in the figma community
- `utils` - This is where your overly complicated functions go. Making a utils folder is kind of like hiding the shame, but in a calculated way. Let's say you

need to transform dates in a component, and it's a pretty heavy function. In my opinion, this should be its file, maybe generalized to dates so it can export several functions for date manipulation - trust me, there will always be date manipulation. In this folder, I would also include any hooks you may create for this particular project.

This structure assumes I don't have a global state management solution, and in case I do, there are usually two options: Using just context and using Overmind(we will get into Overmind later).

In case I am only using context, I would have a folder like:

```
.
├── context
|  ├── userContext.js
|  ├── anotherContext.js
|  └── index.js
```

The idea is to separate your concerns and make each different context it's own file so you can easily change any of them and they are isolated.

I will not go through Overmind because it won't make much sense until we talk about it. Trust me it's great! It powers all of CodeSandbox.

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 is that 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 `Alert.js`, and then if it doesn't find that it looks for the folder and an `index` file within it, so don't worry about more typing.

Let's say this `Alert` component is weirdly complicated for an alert; it can also have components of itself. In many cases I see myself having a components folder inside a component like so:

```
.
├── Alert
| ├── index.js
| ├── elements.(js/css)
| ├── components/
| ├──── form/
| ├────── index.js
| ├────── elements.(js/css)
```

As everything in this book it's a preference I have, smaller files over less folders.

Exporting Components

For many years, I used the good old `export default` with an anonymous function even though the react eslint plugin always yelled at me in specific this

rule.

> *As an aside, I would like to clarify that the eslint-plugin-react is not in any way maintained by the react team.*

In the old days of my innocence, I would export a component like so:

```
import React from "react";

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

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

In recent 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 just exporting an anonymous function:

- You can now see in the React DevTools what the component name thus making it for easier debugging and just overall cleaning of the DevTools.
- Autocompletion in VSCode. Even without TypeScript, VSCode is pretty smart and can do a rundown of your folders, see the component's name and find the one you want. It's not bulletproof without TypeScript, but honestly, it's pretty impressive and more than enough for me to be productive.

Eslint

Let's talk about some eslint issues I have faced in the past, expanding on the previous chapter.

Some eslint plugins and configurations have too many opinions when it comes to stylistic issues, something that, in my opinion, is the purpose of prettier. Eslint is to enforce "good" code, not if you have space between your brackets because honestly, I do not care.

I find it way easier to code with something like eslint-config-react-app than with something that has too many ideas on how I should style my imports and wants them ordered.

With just configs like `eslint-config-react-app` you can get pretty far since, it saves you from all the common pitfalls without forcing you into a particular

style that may actually slow you down.

Without this, there is also no possibility you will get a very aggressive battle where you have conflicting configs in eslint and in prettier and every time you save code, a battle begins, and it all jumps.

In conclusion I do not use the Airbnb config because it cares too much about how my code is styled, and I always have prettier installed.

Remember the times before prettier? That was terrible.

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 "plug it in when you need it" type of tools. At the beginning, it's definitely not needed: maybe your app will start feeling very prone to errors, and it's a good idea then, but not at the start. Never at the beginning unless you know the app you are building will have the need for a complicated state.

SHOULD I USE IT FOR MY MARKETING PAGE?

Honestly...why? It will add way more complexity without gaining 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 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 TYPESCRIPT?

In my opinion, design systems are things where TypeScript is quite handy because you use them all the time, and you need to know what props you want to use, their types, and all of those fancy things.

If your app grows in complexity, you may find yourself in a position where it's also super useful to have typescript on your state management since it will help you make sense of what is going on.

BUT WHAT DOES A TYPESCRIPT REACT COMPONENT LOOK LIKE?

Let's do one with state and props, let's take this simple component 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 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 - 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 we want to show the close button.

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

```
interface Props {
  onClose?: (event: React.MouseEvent) => void;
```

```
  type: "success" | "error" | "warning";
  children: React.ReactNode;
  neverClose?: boolean;
}
```

To apply this to the React component, we do as follows:

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

interface Props {
  onClose?: (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);
          }}
```

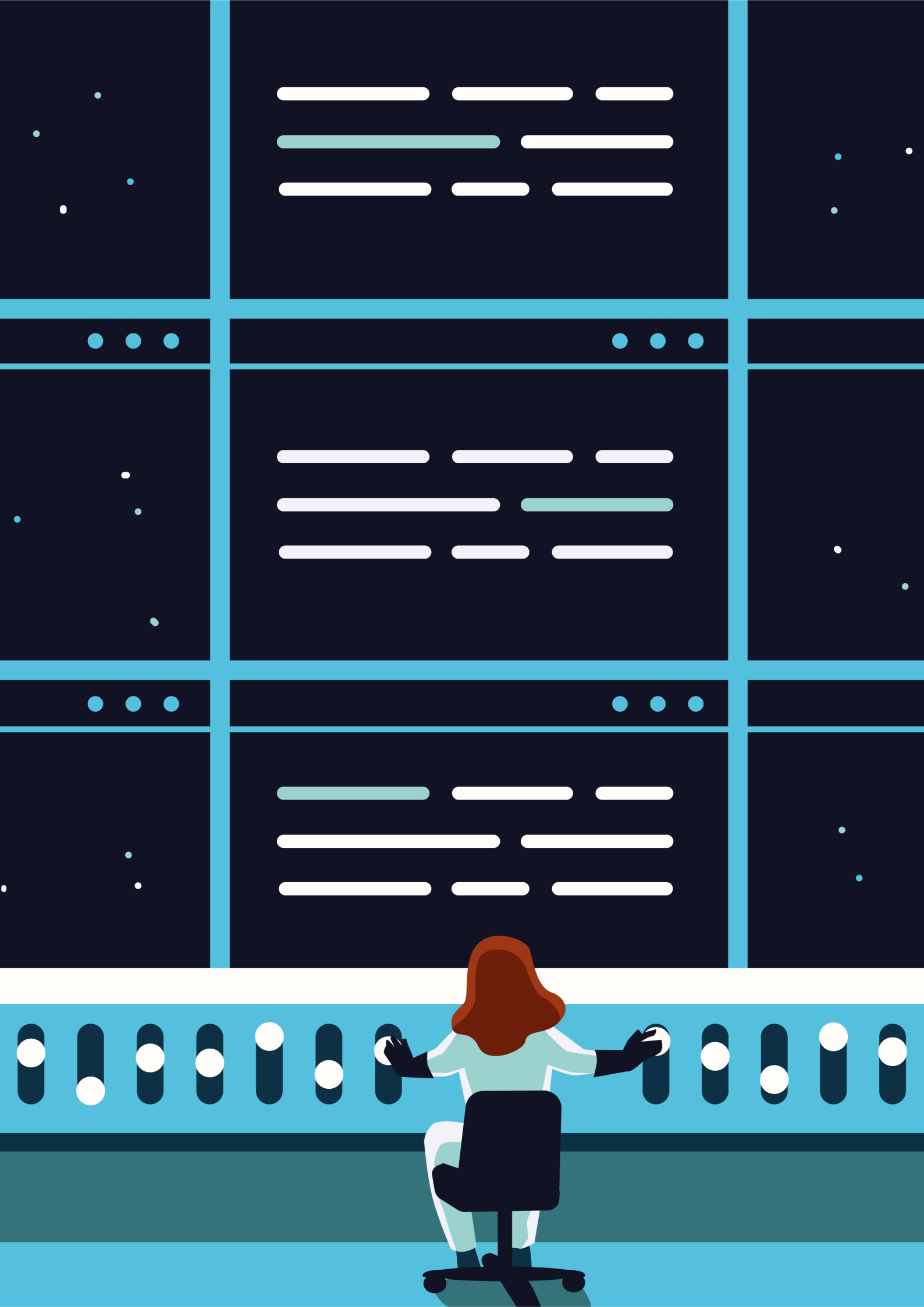
```
          >
            x
          </CloseButton>
        ) : null}
        <Message>{children}</Message>
      </AlertWrapper>
    ) : null;
};
```

There are many other types, but in general, typing React components like these is not a tough thing to do. However, sometimes doing this will lead to more work like transpiling or debugging edge cases that are 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 an open and accessible way as I did, and most of the time, for no reason. I would say 50% or more of the apps out there 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 generally write some code, please avoid using TypeScript. It's not something that they need to learn, and consider whether you yourself need it when making an open-source project or if it's creating a barrier of entry for people who want to help or it's actually helping you maintain it.

There is always a tradeoff.



State Management: Oh, all the options

State Management is tricky, and it may be the hardest thing after cache invalidation and naming we have in the React world, not only because there are plenty of libraries, but also because we have several different ways to handle state.

Let me explain; I see three different ways of handling state in React:

Using useState

`useState` can also be called local state, and if only one or two of your components need state, this state can usually be managed with `useState`.

Imagine that you have a simple app, and one page needs a form. This page is the only page that is not static, and that requires some state to keep track of all the inputs and errors.

There is no need to look any further, `useState` can handle this; it's just this component, and that's what `useState` was meant for, to manage state in a component level.

This all can look something like:

```
import React, { useState } from "react";
import "./styles.css";
```

```
export default function App() {
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");
  const [values, setValues] = useState("");

  const onSubmit = (e) => {
    e.preventDefault();
    setValues({
      name,
      email,
    });
  };

  return (
    <>
      <form onSubmit={onSubmit}>
        <label htmlFor="name">Name</label>
        <input
          name="name"
          id="name"
          value={name}
          required
          onChange={(e) => setName(e.target.value)}
        />
        <label htmlFor="name">Email</label>
        <input
          type="email"
          name="email"
          id="email"
          value={email}
          required
          onChange={(e) => setEmail(e.target.value)}
```

```
          />

          <button type="submit">Submit</button>
        </form>

        <pre>{JSON.stringify(values, null, 2)}</pre>
      </>
    );
}
```

In this case, using component state serves us very well. There will always be a few components like this that need only individual state without a need to share it with other components, and that's super okay. We pluck some local state, and all is good.

It's only when this starts getting complicated and local state is not enough that you should start looking for alternatives in state management.

You can find the code here

Global State with Context

Before thinking of any state management tool, we should consider a solution based on the pieces we have in React already. One of them is `Context`.

Context provides us with two components: a `Provider` and a `Consumer`. The `Provider` will hold the values, and the `Consumer` is the one that reads this value.

`Context` was created and set in beta for many years, and was mostly used by libraries, but now it is entirely stable. It comes with a hook called `useContext`,

that acts precisely as a `Consumer`.

Let's think of an example where you have a user, and you need to show their name everywhere in the application.

This example is something that `Context` can help with, avoiding setting up a whole library to manage state.

```
const Person = {
  name: "Sara",
  city: "Berlin",
  nationality: "Portugal",
};
```

We can create a context for Sara; This will be a call to `React.createContext`. We'll then use the `Provider` I mentioned before to pass this value to anything under it.

There is only one prop in this component, and that is the default value that in this case will be our `Person`:

```
const Person = {
  name: "Sara",
  city: "Berlin",
  nationality: "Portugal",
};

const PersonContext = React.createContext();

const App = () => {
  return (
```

```
    <PersonContext.Provider value={Person}>
      <>
        <Header />
        <Main />
      </>
    </PersonContext.Provider>
  );
};

export default App;
```

As you can see, we have two completely different components but using the `useContext` hook they both can access the same `Person` we have up top, like so:

```
const Header = () => {
  const person = useContext(PersonContext);
  return <p>Hello {person.name}</p>;
};

const Main = () => {
  const person = useContext(PersonContext);
  return (
    <p>
      I see with you are from {person.nationality} and
live in {person.city}
    </p>
  );
};
```

This is a straightforward example, but `Context` can go a long way, and we will see more about it in a later chapter.

I think context is perfect for many use cases where you need state spread around your application, but that state is never going to grow into a big ball of complicated sadness.

When it does, an alternative solution is a state management library.

CodeSandbox Link

Using a state management library

As I try to explain here, a lot of cases can be solved by one of the two previous methods, as it's essential to consider that most of the state management libraries are made with complex state in mind.

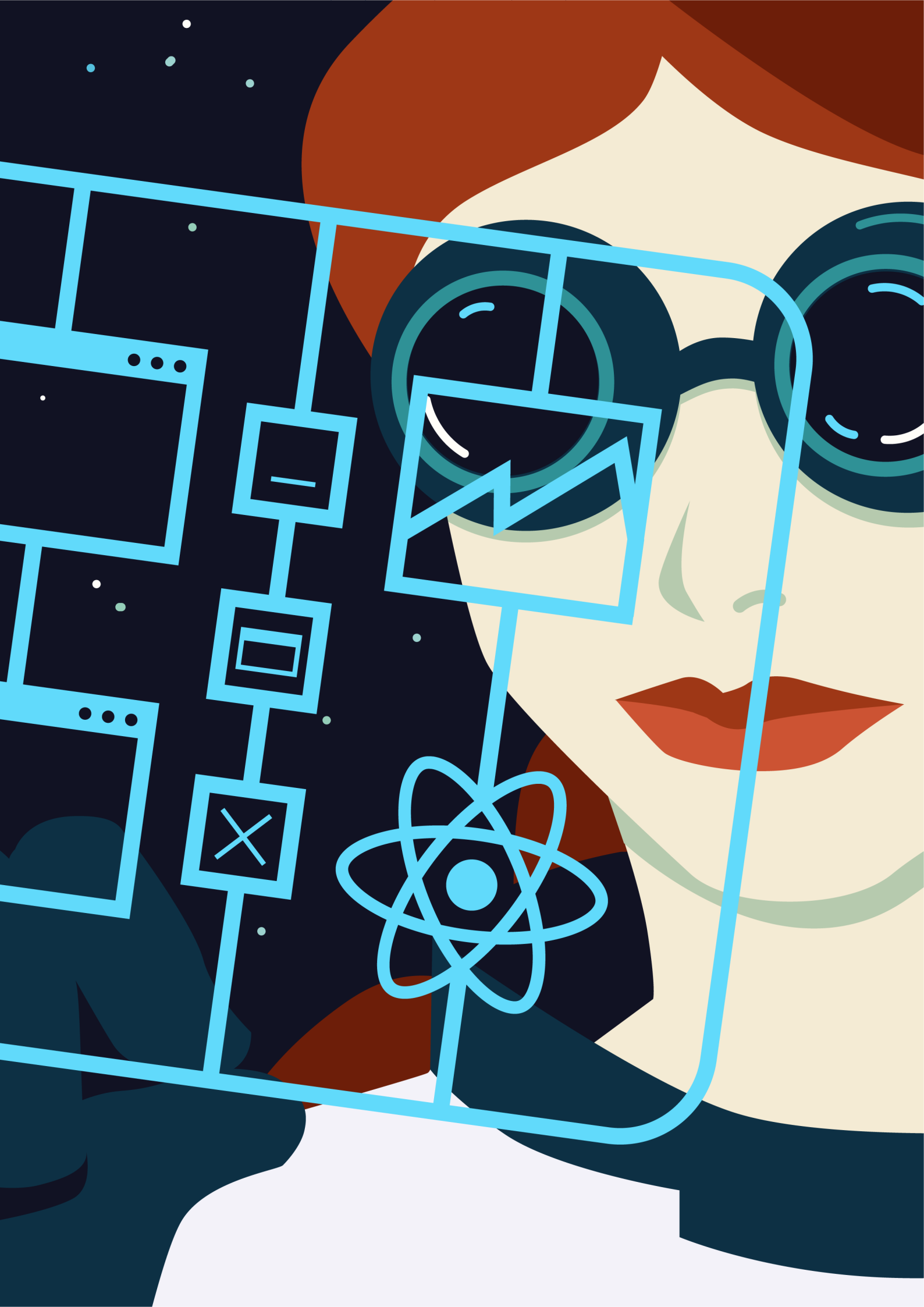
When you go into this journey, you will see that there are many options for managing state, mainly because when using one, you are stating you want to use this library's way of managing state. As humans, we all have very different perspectives on the best way to do this, and now because of this, we get to choose from 300 different libraries.

Later in the book I mention what library I usually use and prefer, but for now, I want you to be aware that using a state management library is not the only way to go, but if you do see that is the way to go in your project, I would tell you to take a look at the one I mention later in the book.

Why did we have this conversation so soon? I want to clarify that we all have the need to use tools that solve issues that React has maybe already solved for us. I would say that 80% of the apps I make for fun and work do not need a state management library and do just fine with local state or context.

One example I always think of is when `Redux` came out, it was used for EVERYTHING, from to-do apps to gigantic applications, and for some of these people, it may have been the perfect library to help them manage state, but it certainly wasn't for everyone.

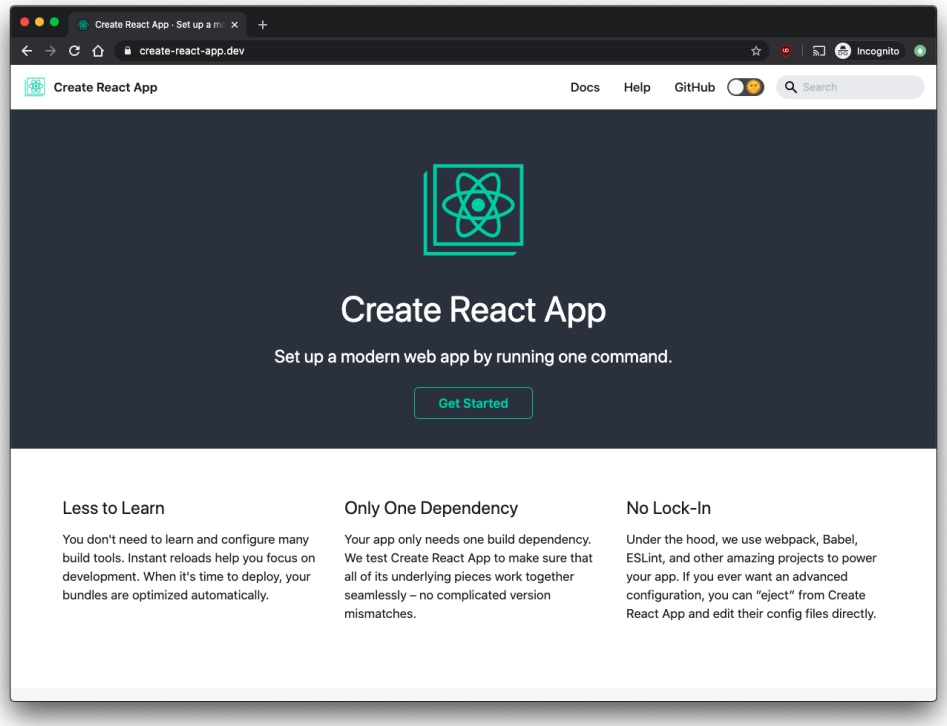
When you start the state management quest, think about this: React has thought about how it can help you make the state a friend in smaller projects.



Project Starters

One of the main issues we had in the first days of React was that it was incredibly hard to get started. You had to mess with webpack and do a lot of tough things to get a hello world up and running. In the last couple of years that has gotten better and 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 tools that are so good that sometimes it's harder to know what starter to use. I will go through the three most popular starters used right now to create different projects in React.

Create React App



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

Create React App (CRA) is the first and most famous one. It's made and maintained by the React team themselves, so you know it will be maintained and updated.

Without a doubt, CRA is the fastest way to get started and have some React code show up on your page. However, its main issue is that it's not very extensible, because you don't have access to the webpack or even babel config. It's a tradeoff you should be aware of from the start, as sometimes the only way to add something is to `eject` and that will leave you with a complex webpack config you need to manage by hand, including updating everything yourself, rather than upgrading the `react-scripts` package that CRA uses to hide that complexity.

Let's look at the pros and cons:

Pros:

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

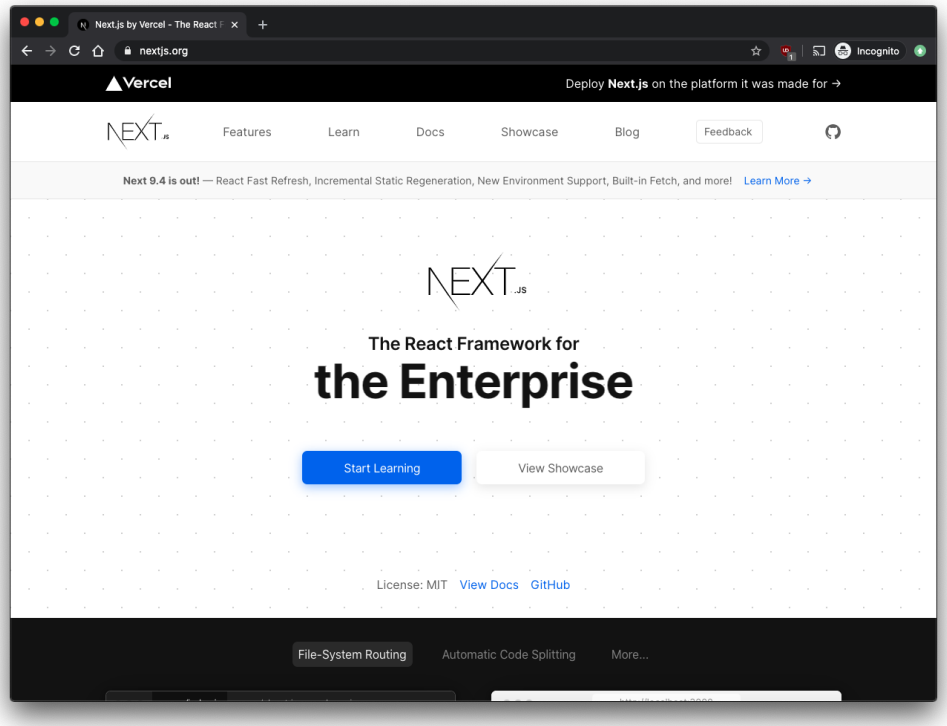
Cons:

- Not a lot of flexibility when it comes to changing the way it handles file types
- No Server Side Rendering (SSR) Support

- No decisions from the React team in terms of app building, so all the router, state management, and other choices 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 anyway, and you will end up with a lot of webpack to handle regardless.

Next



Link: https://nextjs.org/

Next is great, and it comes prepared for a lot of things in your application. More than just a starting point, it's a shortcut to making server-side rendered (SSR) applications in React: SSR is supported out of the box and one of the most significant selling points of Next.

It also makes some decisions for you like routing and styling but gives you the room to decide to use other options if you want to not stick with their choices. You can even change the babel config to support more things you may need.

Let's say you don't want to use their styling options but prefer to use styled-components. In that case, you can extend Next's babel config by adding the plugin in a `.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 in the Next Docs.

Another big advantage of Next is that it has a simple way to get started with the CLI. To create a new project, you can run:

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

The CLI tool will ask you if you want to use an example or start from the default and please choose `example` and then choose the `hello-world` template; this is the simplest one that gives us more leverage to get started.

When you open it, you can see that it has an `index.js` and some more pages, these other pages (`about.js` and a folder called `day`) can be deleted for what we are doing.

Like I said up top the biggest strength next has, in my opinion, is the ability do server side rendering out of the box, and this means that all our requests will first go through node – and like in the node API we have access to the `request` object that will give you a bunch of information on the user making the request, for example their IP address.

We will be using this to get the user's location.

The first thing we need to do is create our function that will get our server side values, that function is called `getServerSideProps`, and we want it to be async since we will be doing an HTTP request in it:

```
import Link from "next/link";

export async function getServerSideProps(context) {
  return {
    props: {},
  };
}

export default function IndexPage() {
  return (
    <div>
      Hello World.{" "}
      <Link href="/about">
        <a>About</a>
      </Link>
    </div>
  );
}
```

You can see that this function always returns an object that contains `props`. This is because this function's outcome will be to return some props to your frontend code so this can be used inside the React component.

The context that the function receives has several parameters, for now, the one we want to talk about is `req` as this is the request the user has made to our site and where their IP will reside, with this information we can get their IP like so:

```
import Link from "next/link";

export async function getServerSideProps({ req }) {
  const ip = req.headers["x-real-ip"];
  return {
    props: {},
  };
}

export default function IndexPage() {
  return (
    <div>
      Hello World.{" "}
      <Link href="/about">
        <a>About</a>
      </Link>
    </div>
  );
}
```

Cool, we have the IP, but we want to pass the full info about the user's location to the client, and for that, I will use an API called ip-api, this service allows us to make a request to `http://ip-api.com/json/${ip}` to get the users location.

Let's then get our location and pass it to the React component:

```
export async function getServerSideProps({ req }) {
  const ip = req.headers["x-real-ip"];

  const res = await fetch(`http://ip-api.com/json/${ip}`);
  const data = await res.json();

  return {
    props: {
      location: data,
    },
  };
}
```

> *As a side note this is an example, please do not ever use the IP to determine location and language, as someone who lives in Germany and doesn't know a lot of german this is the pain of my days.*

If now in your component you `console.log` the location prop, you can see that it gives you an object with all the users' location info we could get from the IP.

The next step we will do is show the user what country they are from. We could show more, but that would go into creepy territory, and we don't want that.

In addition to that, let's also add a link that will send them to a dynamic page we will create afterward:

```
import Link from "next/link";

export async function getServerSideProps({ req }) {
  const ip = req.headers["x-real-ip"];

  const res = await fetch(`http://ip-api.com/json/${ip}`);
  const data = await res.json();
  return {
    props: {
      location: data,
    },
  };
}

export default function IndexPage({ location }) {
  return (
    <div>
      <p>Hello User from {location.country} </p>
      <Link href={`/${location.country}`}>
        <a>{location.country} Page</a>
      </Link>
    </div>
  );
}
```

What we did here after saying hello to the user is that we created a `Link` that will send the user to `/Germany` in my case, these are called dynamic pages, and we can make them quite quickly as well with Next.

If you are wondering why we have an `a` inside the `Link` I don't have an answer for you, but it was a decision by the creators of Next and if you do not put it in it will not work, trust me I have tried and failed.

Let's make our country page, to do that in the `src` folder we add a `[country].js`, if we make a page name inside square brackets it tells Next this is a dynamic page and that it can have any title in the URL bar, kinda like what we do in routers:

```
<Route component={Country} path="/:country" />
```

In this case, we do the same, and the country is the param we can get when the user visits that page.

Inside that page, let's see what the props are like so:

```
export default function CountryPage(props) {
  return <pre>{JSON.stringify(props, null, 2)}</pre>;
}
```

As you can see we get something like:

```
{
  "url": {
    "query": {
      "country": "Germany"
    },
    "pathname": "/[country]",
    "asPath": "/Germany"
  }
}
```

I hope this helps make sense of the way the dynamic pages work in Next.

Let's then use another API, this time restcountries.eu to get more information about the country the user is coming from.

Like in the index page we will create a `getServerSideProps` function, and in this time we want the `query` from the context and pass the value of `country` to the API like so:

```
export async function getServerSideProps({ query }) {
  const url =
`https://restcountries.eu/rest/v2/name/${query.country}`;
  const res = await fetch(url);
  const data = await res.json();

  return {
    props: {
      countryData: data,
    },
  };
}
```

Cool, we now have the country data in our React component, and we can show it like so:

```
export async function getServerSideProps({ query }) {
  const url =
`https://restcountries.eu/rest/v2/name/${query.country}`;
  const res = await fetch(url);
  const data = await res.json();

  return {
    props: {
```

```
      countryData: data,
    },
  };
}

export default function CountryPage({ countryData }) {
  return <pre>{JSON.stringify(countryData, null, 2)}
</pre>;
}
```

I hope this helps you see the benefits of using something like Next to build your apps.

You can see the code and preview on CodeSandbox.

As a final thought, `Next` recently introduced ways of making your `Next` websites Statically Generated with functions like:

- `getStaticProps` - This is something you can use if you want props to be passed to the component on build instead of on server render, like in `getServerSideProps`.
- `getStaticPaths` - This function can programmatically create pre-rendered HTML pages for you depending on the paths you return from it. It uses templates that will be pre-filled to do this.

I am not diving into this because, in the next chapter, we will be talking about Gatsby, which is a static site generator that was created for these specific purposes with a lot of official and community plugins that help you in these tasks. Because of this, when something is static, I prefer to use `Gatsby`. When it's server rendered, I go with `Next`. I like to keep these things separated, and that's why I am not mentioning these functions, I've never used them.

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

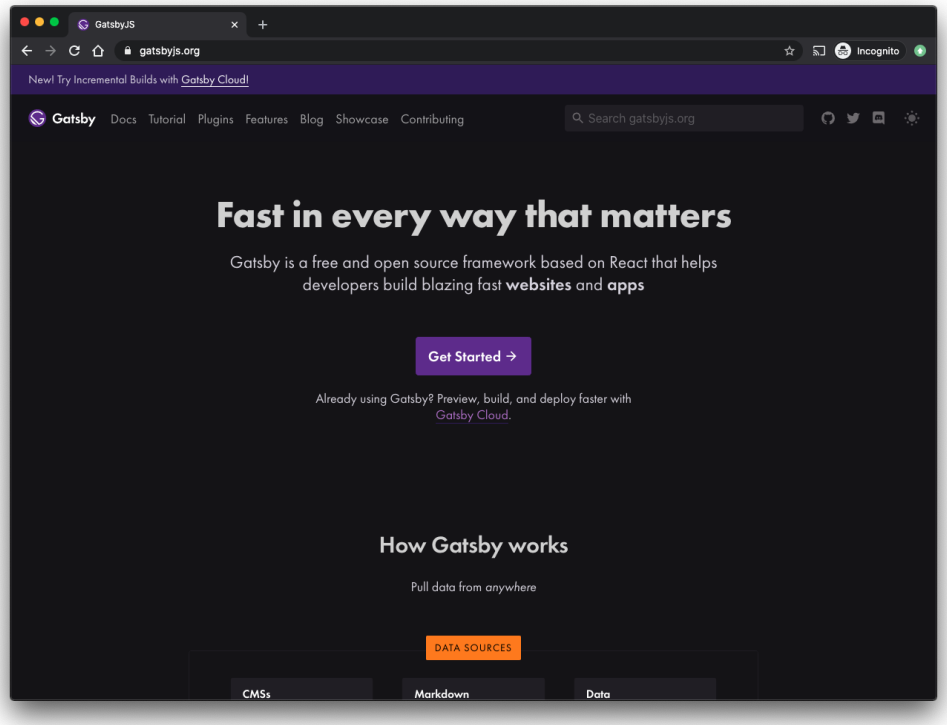
Pros:

- Server Side Rendering support
- Amazing docs with lessons to follow
- Production-grade
- Customization options

Cons:

- A steep learning curve, mostly because SSR things are just harder.
- Harder to leave if `Next` is not the best fit for your project

Gatsby



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

I'll be honest; Gatsby is basically my Create React App - it's what I use for basically everything. Even the website for this book is made in Gatsby, just because.

Gatsby was started as a project by Kyle Mathews to create blogs, but it grew into so much more. 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.

"What's so good about static sites?" you may ask. One of the main benefits is, without a doubt, the SEO: the app is pre-rendered HTML, so everything gets read by Google to rank your site better. Another significant advantage is the deployment: Static HTML sites are waaaaay more straightforward to deploy than server-side applications.

Getting started is also quite easy as Gatsby also has a CLI:

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

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

Gatsby's main strength is to be able to pull data from basically anywhere and create HTML files from it with GraphQL, so it's necessary to know some GraphQL to get a full grasp of its potential.

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

The first step is to install the plugin:

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

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

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

Now that we have the data, we're going to use some GraphQL now. GraphQL may look scary, but it's very similar to JSON as a way to structure data. In this case, Gatsby uses it to feed each page with the data it wants. The main benefit of this is that you can pick and choose what you want to fetch from a request, unlike REST most of the time.

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, or pretty much anything you want to achieve with GraphQL, open `http://localhost:8000/___graphql`. That 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>
);
```

What we have is already pretty cool, but we can still go further in this example. Let's now make a page for every single person we are getting from our GraphQL query.

For this, we will need to touch our `gatsby-node.js` file. This file allows us to make adjustments to our data layer, and what we get in each page.

We are looking for the `createPage` function, and it will allow us to create pages programmatically.

We now need to import `path`, since we need to get a `template`. In Gatsby's context, a `template` is a custom-made page, and it's called like that since they all have the same structure but get fed different data.

```
const path = require(`path`);

exports.createPages = async ({ graphql, actions }) => {
  const { createPage } = actions;
  const userTemplate =
path.resolve(`src/templates/user.js`);
};
```

If you try to restart Gatsby you will get an error, since this file does not exist yet. So let's create it with dummy content so we can restart the server:

```
import React from "react";

const User = (props) => {
  return <>Hello</>;
};

export default User;
```

Let's now add the `GraphQL` query that will get our data:

```
const path = require(`path`);

exports.createPages = ({ graphql, actions }) => {
```

```
  const { createPage } = actions;
  const userTemplate =
path.resolve(`src/templates/user.js`);

  const result = await graphql(
    `
      query Users {
        allRandomUser {
          edges {
            node {
              id
            }
          }
        }
      }
    `
  );
};
```

We need to do this because Gatsby needs to be fed this data to loop through all the random users and create the pages.

We can then go through every random person that comes back from the query and create a page for them:

```
const path = require(`path`);

exports.createPages = async ({ graphql, actions }) => {
  const { createPage } = actions;

  const result = await graphql(
    `
```

```
      query Users {
        allRandomUser {
          edges {
            node {
              id
            }
          }
        }
      }
    `
  );

  return result.data.allRandomUser.edges.forEach((edge) =>
{
    createPage({
      path: `/users/${edge.node.id}`,
      component: path.resolve(`src/templates/user.js`),
      context: {
        id: edge.node.id,
      },
    });
  });
};
```

Here, we are doing a `forEach` on `allRandomUser` to call `createPage` on each one of them.

The properties of the object that we need to pass to the `createPage` are:

- `path` - This is the path where we want to access this page. In this case, we want it to be `/users/the-user-id-here`.

- `component` - The path to our template component goes here, Gatsby knows this is the file from which it needs to generate the pages.
- `context`: In here, we can pass any additional data that the component may need. In this case, I will pass on the `id` so I can make a query for the correct person on the template itself.

Cool! We can now restart our server. To test that this works properly, we need to change our `pages/index.js` a bit, and add a link on each person there to send the user to the person page:

```
<Link to={`users/${node.id}`} key={node.id}>
  <li>
    <img src={node.picture.thumbnail} alt=
{node.name.first} />
    {node.name.first} {node.name.last}
  </li>
</Link>
```

We use the `Link` component from Gatsby to wrap each of our users in a `Link` to their personal page.

If we go back to the browser and click on a user, we will see that it sends us to a newly created page for this user. Too bad, it only says hello.

So let's get it to be more descriptive with the actual user data.

The first step is to create our `GraphQL` query that will fetch just one user based on the `id`:

```
export const query = graphql`
  query User($id: String) {
```

```
    allRandomUser(filter: { id: { eq: $id } }) {
      edges {
        node {
          name {
            first
          }
        }
      }
    }
  }
`
```

We are saying: give me all the users who have the id that equals the `id` that I'm passing.

How does it get this `id`? It's in the data what we passed in the context in `createPage`:

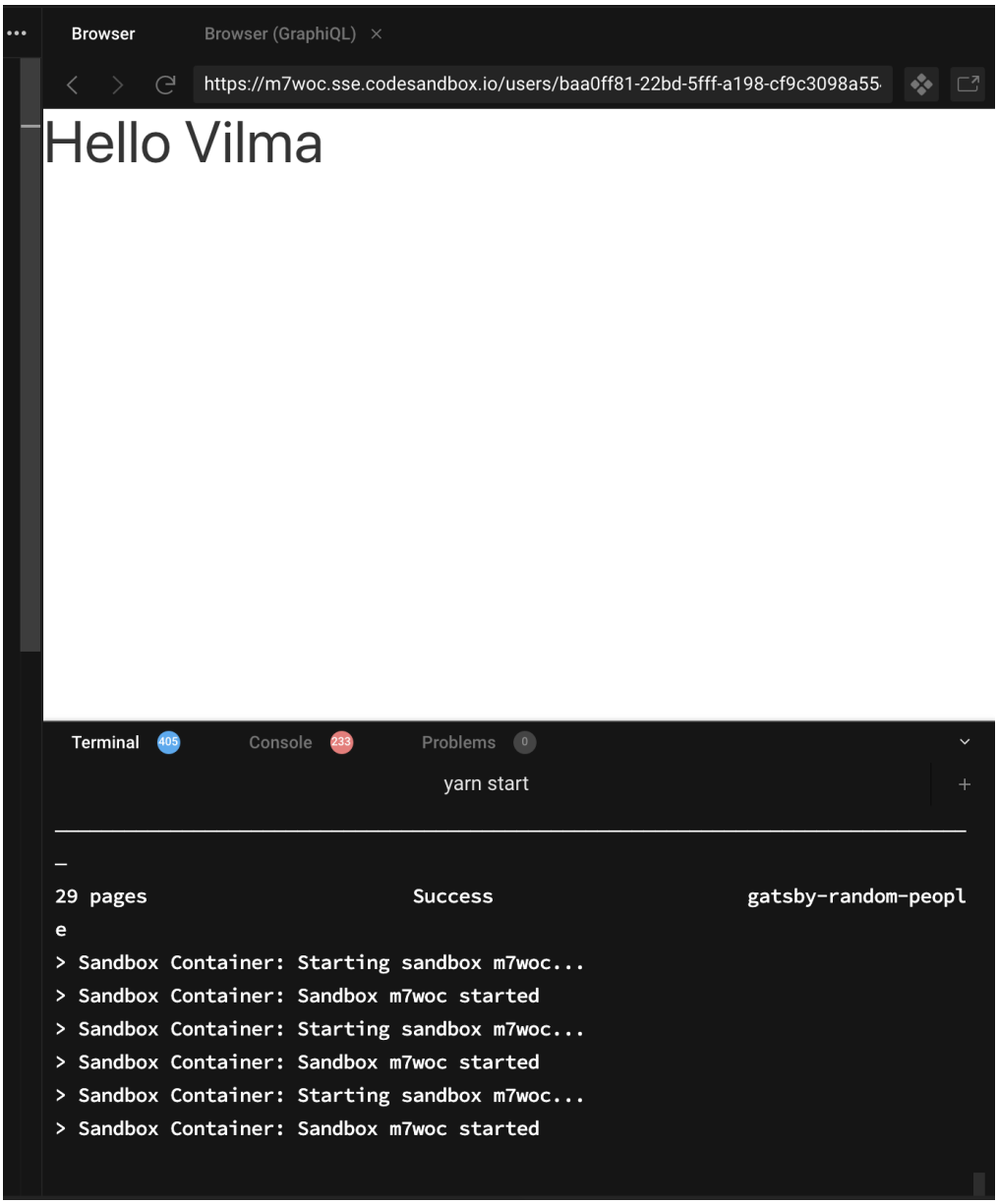
```
createPage({
  path: `/users/${edge.node.id}`,
  component: userTemplate,
  context: {
    id: edge.node.id,
  },
});
```

Now in our props, we will also get a `data` field like we do in our index, and from there, we can extract the user and show it.

```
const User = ({ data }) => {
  const user = data.allRandomUser.edges[0].node;
```

```
  return <h1>Hello {user.name.first}</h1>;
};
```

And it works!



We did a bunch of work with Gatsby, and I hope with this, you can see how powerful it is to create static websites.

You can see the code and preview on 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 accessibility

Cons:

- The steep learning curve for any advanced things
- Knowledge of GraphQL required to get started
- Not everything can be static
- Some errors only show up on build or deploy

Packages

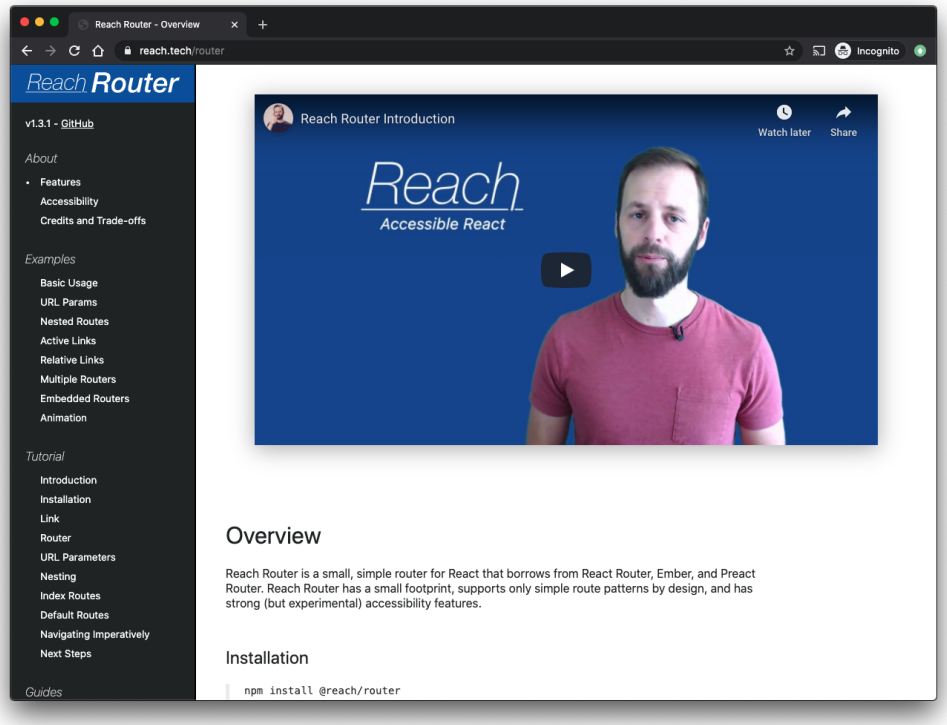
If you want something, there is almost certainly a React package for that. In my opinion, this is both a strength and a downfall when combined with the fact that React itself only provides you with a view layer, and the rest is supposed to be figured out by yourself.

You may ask how I can see this as a downfall - shouldn't the fact that there are so many packages and people out there making tools be an advantage? It's only an advantage when there is a direction and recommended ways; something React refuses to create, so to anyone getting started, it all looks like a sea of sameness. This section of the book is for people who feel the same way, for whom searching for a router is like going into a voyage.

Here I will go through the packages I use in parts of an app and how to use the basics of each. That way, you can take everything I say and then make an informed decision about whether you will use the same thing or continue on the quest.

Let's start.

Routing



Chosen: React Router 6

I want to start by saying this version of React Router is still in beta, so it may have some bugs. If you want my recommendation for something that is not in beta, then I would definitely say Reach Router, I used it for a long time, the main reason I will be talking about React Router instead is because Reach Router will not be getting any new features and the author is also working in React Router.

With this out of the way, let's create some routes!

The first step is to install the latest versions of `react-router-dom` and `history`:

```
yarn add history@5 react-router-dom@6
```

We'll start by making a homepage and call it using `react-router`. To do this, we'll create `Home.js`, for now with just the universal `Hello World` on it so that we know it works:

```
import React from "react";

const Home = () => `Hello World`;

export default Home;
```

Let's now import `react-router-dom` into our `index.js`, we'll show this new page on page load:

```
import { render } from "react-dom";
import React from "react";
import Home from "./Home";
import { BrowserRouter, Routes } from "react-router-dom";

render(
  <BrowserRouter>
    <Routes>
      <Home path="/" />
    </Routes>
  </BrowserRouter>,
  document.getElementById("root")
);
```

The first thing we do here is importing both `Routes` and `BrowserRouter`. Then we want to wrap our application in both.

You may notice that the route is written in a slightly different way, this is something that I started using back in my Preact days.

Instead of doing `<Route path="/" element={<Home />}/>` as it's recommended in the docs, we can also call it in a shorter version `<Home path="/" />`

This is ultimately a personal choice, the new React Router supports both, they call this shorter way "animal style" since its basically like a secret menu in their API.

Random fact: I've been told that these secret menu items called "animal style" are a thing in the US, specially at In-n-Out, where the "animal style" cheeseburger exists.

We can explore the traditional way adding a 404 error:

```
import { render } from "react-dom";
import React from "react";
import Home from "./Home";
import User from "./User";
import { BrowserRouter, Routes, Route } from "react-
router-dom";

render(
  <BrowserRouter>
    <Routes>
      <Home path="/" />
      <Route path="*" element={<p>Sorry, nothing here</p>}
/>
    </Routes>
  </BrowserRouter>,
```

```
  document.getElementById("root")
);
```

When testing this in the browser, you will see that the home page is now there. If you add anything else to the url, like my personal favorite: `/asdf`, we get the element we passed into the second route: an error.

Cool, we have a homepage now. But just for that, we don't really need a router right? No, we don't. Sorry that was a rhetorical question. Let's carry on.

The next step is adding a user page that will receive parameters from the path URL in our `index.js`:

```
import { render } from "react-dom";
import React from "react";
import Home from "./Home";
import User from "./User";
import { BrowserRouter, Routes, Route } from "react-
router-dom";

render(
  <BrowserRouter>
    <Routes>
      <Home path="/" />
      <User path="user/:id" />
      <Route path="*" element={<p>Sorry, nothing here</p>}
/>
    </Routes>
  </BrowserRouter>,
  document.getElementById("root")
);
```

As you can see in here, the path is passed as `user/:id`. This means that the parameter in the URL that comes after `user/` is matched as the id and will be passed to our component, thus the route will be rendered.

This also means that it *needs* an `:id` to be matched, if you try to go to `/user` you will get a 404.

We are importing a page that we have not created so let's create a `User.js`.

```
import React from "react";

const User = () => {
  return <div>Hello</div>;
};

export default User;
```

If you now navigate to `user/arandomuserhere`, you can see the browser says "hello" to us and that's pretty cool, we made it speak. But it would be great if it also knows our name, so let's teach it how to use it.

When we created our route, we said it should match the path `user/:id`.

This `id` is what is called a parameter in this route, to get these, React Router offers an excellent hook that makes parameters very straightforward to work with.

For that, we need to import the `useParams` hook:

```
import React from "react";
import { useParams } from "react-router-dom";
```

```
const User = () => {
  let { id } = useParams();
  return <div>Hello {id}</div>;
};

export default User;
```

Now the browser knows our name! What happens here, is that `useParams` will return an object containing all the "params" (or parameters) we have in the route along with their value, so that we can use them anywhere in our component.

We are getting places! But our homepage still says "hello world", and that's not the best, so let's change it.

Imagine we need to place an input where we can type the id of a user so we can get redirected to their page, and also a link to a user called `react-router`, because, why not?

Let's start by making our `<Link>`:

```
import React, { useState } from "react";
import { Link } from "react-router-dom";

const Home = () => {
  return <Link to="user/react-router">Go to the React
Router user</Link>;
};

export default Home;
```

To create links between pages in the same app, we can use `<Link>`. This renders the linked pages that we want to add navigation for without reloading the page.

There are times where we may also want to redirect a user without making them click on a link. For example, imagine a form being completed or a button being clicked.

For this purpose, we have another hook called `useNavigate`:

```
import React, { useState } from "react";
import { Link, useNavigate } from "react-router-dom";

const Home = () => {
  const navigate = useNavigate();
  const [user, setUser] = useState("");
  return (
    <>
      <Link to="user/react-router">Go to the React Router
user</Link>
      <form
        onSubmit={(e) => {
          e.preventDefault();
          navigate(`/user/${user}`);
        }}
      >
        <input
          placeholder="input a username"
          value={user}
          onChange={(e) => setUser(e.target.value)}
        />
        <button disabled={!user}>Go to that user</button>
```

```
      </form>
    </>
  );
};

export default Home;
```

The fact that I can just call `navigate` with a path and the user gets sent somewhere is the best thing about client routers because dealing with URL's is farthest away from fun in my mind... well maybe after dealing with Docker but it's super close.

Another thing I want to show you and that is the `Outlet` and that we can use to handle child routes. In our case I want the form to show up in both pages so that it's super easy to switch user even when we are already inside a user page. This is precisely what `Outlet` is meant to do, allow us to render child routes inside a main route.

To make this change and use `Outlet`, let's first change our `index.js` to actually make the `User` route a child of the `Home` route:

```
import { render } from "react-dom";
import React from "react";
import Home from "./Home";
import User from "./User";
import { BrowserRouter, Routes, Route } from "react-
router-dom";

render(
  <BrowserRouter>
    <Routes>
      <Home path="/">
```

```
        <User path="user/:id" />
      </Home>
      <Route path="*" element={<p>Sorry, nothing here</p>}
/>
    </Routes>
  </BrowserRouter>,
  document.getElementById("root")
);
```

With this in place you can see that now the user route is no longer available and that is because our `Outlet` is not yet placed and this is how react router knows where to render this child route.

Let's then place it after the form in our homepage:

```
import React, { useState } from "react";
import { Link, useNavigate, Outlet } from "react-router-
dom";

const Home = () => {
  const navigate = useNavigate();
  const [user, setUser] = useState("");
  return (
    <>
      <Link to="user/react-router">Go to the React Router
user</Link>
      <form
        onSubmit={(e) => {
          e.preventDefault();
          navigate(`/user/${user}`);
        }}
      >
```

```
        <input
          placeholder="input a username"
          value={user}
          onChange={(e) => setUser(e.target.value)}
        />
        <button disabled={!user}>Go to that user</button>
      </form>
      <Outlet />
    </>
  );
};

export default Home;
```

By reloading the page you can see this new route is rendered inside the main Homepage, it works great and just like we wanted. You can imagine how this would be super useful in a place like a dashboard where things like sidebars and headers are shared and only the content is changed, it makes life much easier.

Last thing before we move on to state management is the `NavLink` and this is a special type of `Link` component that knows when it's active and accepts an `activeStyle` or an `activeClassName` that will be activated when you are in the route that the link points to.

To test that let's make a link to our homepage:

```
import React, { useState } from "react";
import { Link, useNavigate, Outlet, NavLink } from "react-
router-dom";

const Home = () => {
  const navigate = useNavigate();
```

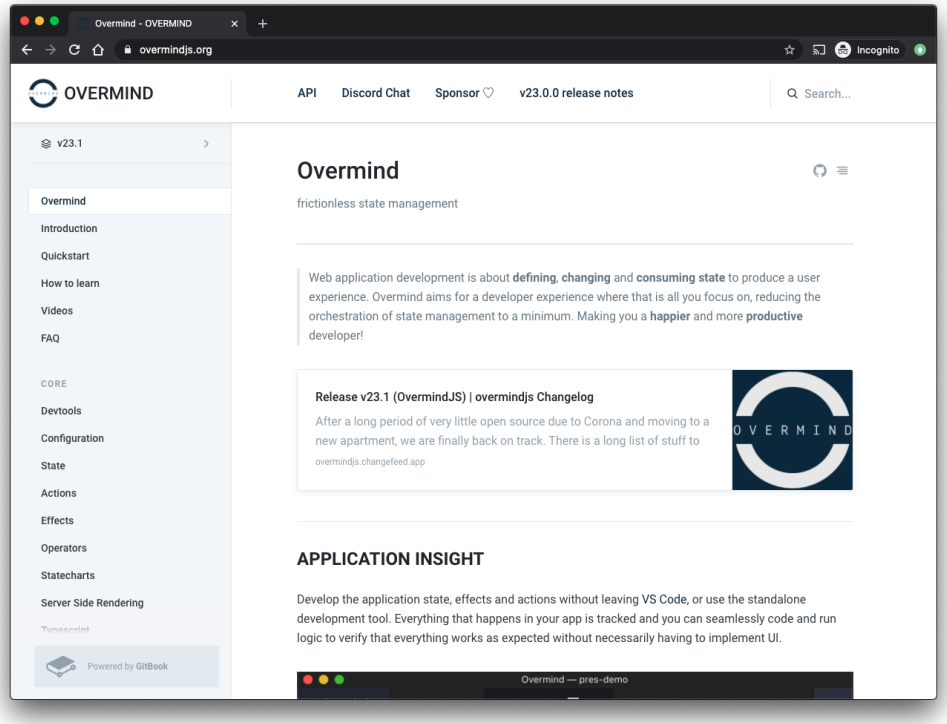
```
  const [user, setUser] = useState("");
  return (
    <>
      <NavLink activeStyle={{ color: "white" }} to="/">
        Go Home
      </NavLink>
      <Link to="user/react-router">Go to the React Router
user</Link>
      <form
        onSubmit={(e) => {
          e.preventDefault();
          navigate(`/user/${user}`);
        }}
      >
        <input
          placeholder="input a username"
          value={user}
          onChange={(e) => setUser(e.target.value)}
        />
        <button disabled={!user}>Go to that user</button>
      </form>
      <Outlet />
    </>
  );
};

export default Home;
```

And that's it, when we are in the `/` path we can see our link is now white and it's super easy to add custom styles for active links this way.

I hope you see what is so great about this latest release of react router and give it a try.

Link to CodeSandbox

State Management



Chosen: Overmind

At CodeSandbox, we have been using Overmind for a while, and we were using its predecessor `Cerebral` before that. It is honestly a breath of fresh air when it comes to state management: it's simpler than most but super powerful and super extensible. You can use Overmind in big applications with minimal boilerplate.

Beware that it is a mutable state management option. If this is a no go for you, I am sorry, but you should take a look anyway as it doesn't make your app slower or harder to follow. It also has complete TypeScript support: all your state gets

typed automatically, and as someone who has her doubts about TypeScript, even I can say it's impressive.

> *Back up!! What is the difference between mutable and immutable code?*

Glad you asked! You have probably heard this discussion a LOT when talking about React, with statements such as the fact that `setState` is immutable, among other things.

When dealing with values in JavaScript, there is usually both an immutable and a mutable way of updating these values.

Let's take the following array:

```
const bayernMunichChampionsLeagueWins = [1974, 1975, 1976, 2001, 2013];
```

After their latest win, we need to add the `2020` season to this array.

Let's explore the immutable approach first. In this scenario, we will not change the original array, we instead create a new variable that will have the old wins and also `2020` like so:

```
const bayernMunichChampionsLeagueWins = [1974, 1975, 1976, 2001, 2013];
const newWins = bayernMunichChampionsLeagueWins.concat(2020);

console.log(newWins);
```

In this case we didn't change the original array, so if we ever need to access it again, we can, the original variable keeps its original state.

On the other hand, we could also use `push`, which updates the existing array instead of creating a new one, like so:

```
const bayernMunichChampionsLeagueWins = [1974, 1975, 1976,
2001, 2013];
console.log(bayernMunichChampionsLeagueWins.push(2020));
```

This is the mutable way of doing it, it's called mutable because we can see the original value of the array has been changed.

There are advantages and disadvantages to using both.

On one hand, if you use mutable values, you may be deleting state that you may want to access again later. This has happened to me before when using push instead of concat.

This is less of an inconvenience with projects like Overmind, since they take care of keeping track of the changes you make to each value in your state.

Immutable code in the other hand, is usually harder to make sense of. You also tend to need more code to accomplish something that would be more straightforward with mutability.

I use immutable code in my views with `useState` as this is intentional in the design of React, but when it comes to global state, after using Redux and other immutable state management solutions out there, I think that the complexity it adds is not worth it because of tools like Overmind, which will keep track of all your mutations to the state making sure you know what's being changed and preventing unexpected side effects.

Let's now go back to the scheduled book.

Speaking is easier with code, so let's make a simple app to show how Overmind works.

The first thing we need to do is install `overmind` and `overmind-react`:

```
yarn add overmind overmind-react
```

Then we can create an `overmind/index.js` and start our Overmind setup:

```
import { createHook } from "overmind-react";

export const config = {
  state: {
    terms: ["SSR", "PWA"],
  },
  actions: {
    // anything to transform the state
  },
};

export const useOvermind = createHook();
```

Here we first import `createHook` from Overmind, which will allow us to use Overmind with a straightforward hook.

After that, we define our config that for now just holds our state with a couple of Front End-related terms.

Next, we need to pass this config to our React app:

```
import React from "react";
import { render } from "react-dom";
import { createOvermind } from "overmind";
import { Provider } from "overmind-react";
import { config } from "./overmind";
import App from "./components/App";

const overmind = createOvermind(config);

render(
  <Provider value={overmind}>
    <App />
  </Provider>,
  document.querySelector("#root")
);
```

So we created an Overmind instance and passed it to our App. That's it, that's all the boilerplate we need to do to get it to work.

We can now list our amazing terms on our page. In our `./components/App` you can add:

```
import React from "react";
import { useOvermind } from "../overmind";

const App = () => {
  const { state } = useOvermind();

  return (
```

```
    <>
      <ul>
        {state.terms.map((term, i) => (
          <li>{term}</li>
        ))}
      </ul>
    </>
  );
};

export default App;
```

If you now check back on our app, you can see that we can see all our terms on the page, and it was honestly pretty painless.

So far, we only display data, and usually, the hardest part is changing the state. But remember what I said at the top: Overmind is mutable, so changing the state is quite straightforward.

To do this, we create actions. Actions will also get the state as a parameter and have the ability to mutate it, so let's make an action to add a term, and one to delete a term:

```
import { createHook } from "overmind-react";

export const config = {
  state: {
    terms: ["SSR", "PWA"],
  },
  actions: {
    addTerm({ state }, term) {
      state.terms = [term, ...state.terms];
```

```
    },
    removeTerm({ state }, indexToDelete) {
      state.terms = state.terms.filter((_, index) =>
indexToDelete !== index);
    },
  },
};

export const useOvermind = createHook();
```

As you can see, all we do is reassign `state.terms` to a new value, which will update our components that are using Overmind and that part of the state magically.

Looking deeper into those actions, we can see that we get two parameters: the first one always comes from Overmind, and it includes `state` and other `actions` (and also a couple more things you may need in the future like `effects`), and our second parameter is anything we pass into the action when we call it.

Let's now attach it to our components by also getting the effects of our `useOvermind` hook:

```
import React, { useState } from "react";
import { useOvermind } from "../overmind";

const App = () => {
  const {
    state,
    actions: { addTerm, removeTerm },
  } = useOvermind();
  const [term, setTerm] = useState("");
```

```
  const onSubmit = (e) => {
    e.preventDefault();
    addTerm(term);
    setTerm("");
  };

  return (
    <>
      <h1>Add a term</h1>
      <form onSubmit={onSubmit}>
        <input
          type="text"
          value={term}
          onChange={(e) => setTerm(e.target.value)}
        />
        <button type="submit" disabled={!term}>
          Add
        </button>
      </form>
      <h1>Terms</h1>
      <ul>
        {state.terms.map((term, i) => (
          <li>
            {term} - <button onClick={() =>
removeTerm(i)}>x</button>
          </li>
        ))}
      </ul>
    </>
  );
};
```

```
export default App;
```

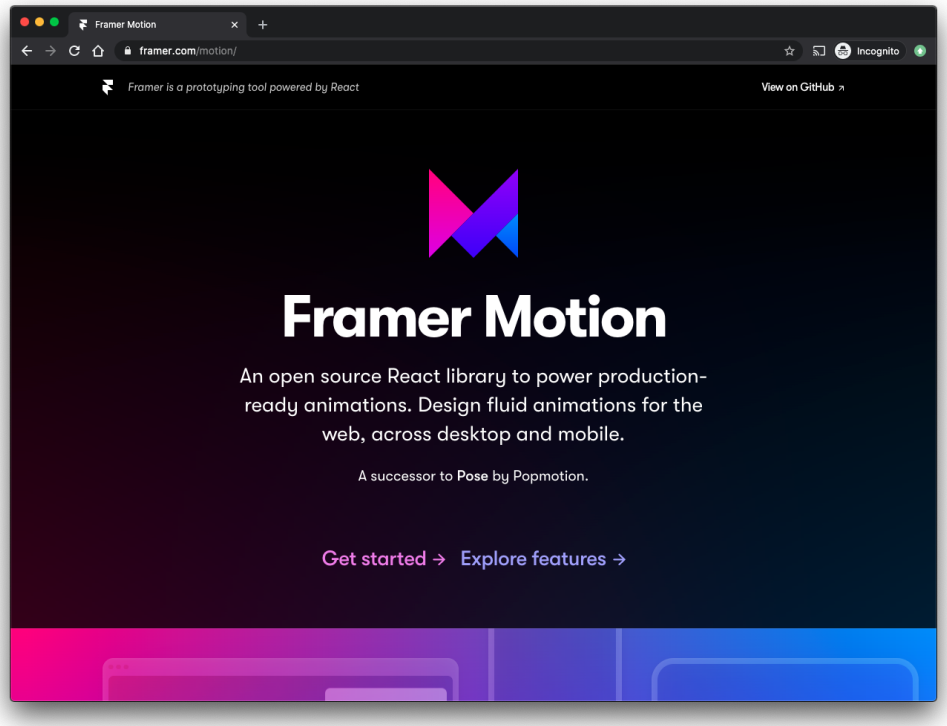
A fully connected form and it was pretty painless!

I am honestly in love with this way of state management. It's simple to get started, boilerplate-free and has a lot of room to grow. I feel like it ticks all the boxes in something you may want in a state management solution, including its own DevTools as a VSCode plugin.

I would say give it a try and let me and the creator know how you feel about it.

Link to CodeSandbox

Animation



Chosen: Framer Motion

Animation is hard - actually *super* hard - and I feel like in React, it ends up slightly harder because it's two different packages trying to control the DOM. For many years I tried several packages until Framer Motion came out. I find that it fits most of the needs I have for UI animation in React.

> *Beware that for very complex animation, you should still use gsap as that allows for things like timelines, which help when an animation needs a lot of entrances and exits.*

I will show a small example that will demonstrate how to animate something to `height: auto`, which makes us cry a lot when doing it in real life. Let's get started with installing Framer Motion:

```
yarn add framer-motion
```

The first thing you should know is that Framer Motion exports a whole set of components from their package, but today we will be focusing mostly on the `motion` component. This is the component where you can place all your animations and define the transition times and methods for them.

We will have a small app that will open an accordion-like so:

```
import React, { useState } from "react";
import Spectrum from "react-spectrum";
import ReactDOM from "react-dom";

const App = () => {
  const [isOpen, setIsOpen] = useState(false);

  return (
    <div class="wrapper">
```

```
      <button class="open" onClick={() =>
setIsOpen(!isOpen)}>
        {!isOpen ? "Open me" : "Close me"}
      </button>
      {isOpen && (
        <main style={{ overflow: "hidden" }}>
          <Spectrum
            linesPerParagraph={lines > 1 ? lines : 1}
            width={500}
            colors={["#757575", "#999999", "#0871F2",
"#BF5AF2"]}
          />
        </main>
      )}
    </div>
  );
};

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

> *The package react-spectrum is a simple package that generates colorful text placeholders, so we don't have to put in any ugly lorem ipsum text.*

If you click the button, you can see the text shows up, but it doesn't animate. For that, we need to replace the `main` component with a `motion.main`. The way to use the `motion` component is that everything after the `.` must be a valid HTML element. T will be the element that we use to wrap and animate your component, and in this case, I will be using the `main` element.

Let's now import `motion` and define our animations:

```
import React, { useState } from "react";
import Spectrum from "react-spectrum";
import ReactDOM from "react-dom";
import { motion } from "framer-motion";

const App = () => {
  const [isOpen, setIsOpen] = useState(false);

  return (
    <div class="wrapper">
      <button class="open" onClick={() =>
setIsOpen(!isOpen)}>
        {!isOpen ? "Open me" : "Close me"}
      </button>
      {isOpen && (
        <motion.main
          style={{ overflow: "hidden" }}
          initial={{ height: 0 }}
          animate={{ height: "auto" }}
        >
          <Spectrum
            linesPerParagraph={lines > 1 ? lines : 1}
            width={500}
            colors={["#757575", "#999999", "#0871F2",
"#BF5AF2"]}
          />
        </motion.main>
      )}
    </div>
  );
};
```

```
const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

Right now we have a motion `motion-main` with three props:

- `style` - Some style rules that will be applied in all states of the animation (can also be a `className`)
- `initial` - The starting state of our element
- `animate` - What values to animate to

If you click the button, it works on open, no problem (and it looks sweet), but no animation plays on close. This problem happens because in the way that we've built it, React unmounts the component as soon as the button is clicked, so there's no time left for an animation.

To solve this, there is a Framer Motion element called `AnimatePresence` that we'll wrap our `if` statement in, which will allow us to specify an `exit` prop on our `motion` component. This lets us define an animation to use when a component is unmounted.

Let's add that, then:

```
import React, { useState } from "react";
import Spectrum from "react-spectrum";
import ReactDOM from "react-dom";
import { motion, AnimatePresence } from "framer-motion";

const App = () => {
  const [isOpen, setIsOpen] = useState(false);

  return (
```

```
    <div class="wrapper">
      <button class="open" onClick={() =>
setIsOpen(!isOpen)}>
        {!isOpen ? "Open me" : "Close me"}
      </button>
      <AnimatePresence>
        {isOpen && (
          <motion.main
            style={{ overflow: "hidden" }}
            initial={{ height: 0 }}
            animate={{ height: "auto" }}
            exit={{ height: 0 }}
          >
            <Spectrum
              linesPerParagraph={lines > 1 ? lines : 1}
              width={500}
              colors={["#757575", "#999999", "#0871F2",
"#BF5AF2"]}
            />
          </motion.main>
        )}
      </AnimatePresence>
    </div>
  );
};

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

If you click the button now, you can see that the exit prop is used, and our animation works on enter *and* exit. This component is actual magic.

Other interesting props include the `transition` prop, which lets you define things like duration, ease, and delay. For example, to set the `duration` to `1` you can add to our `motion.main` the following prop:

```
<motion.main
  style={{ overflow: "hidden" }}
  initial={{ height: 0 }}
  animate={{ height: "auto" }}
  exit={{ height: 0 }}
  transition={{ duration: 1 }}
>
```

And now our animation will be one second long.

I hope this gets you excited about Framer Motion!

Link to CodeSandbox

Styling

Before starting this I want to say that when it comes to CSS, you can still write in standard CSS/SASS/LESS files as much as you want, I happen to prefer to use CSS-in-JS but don't think this is the only option, using simple classes like we always have also works and it will forever work.

Doing this:

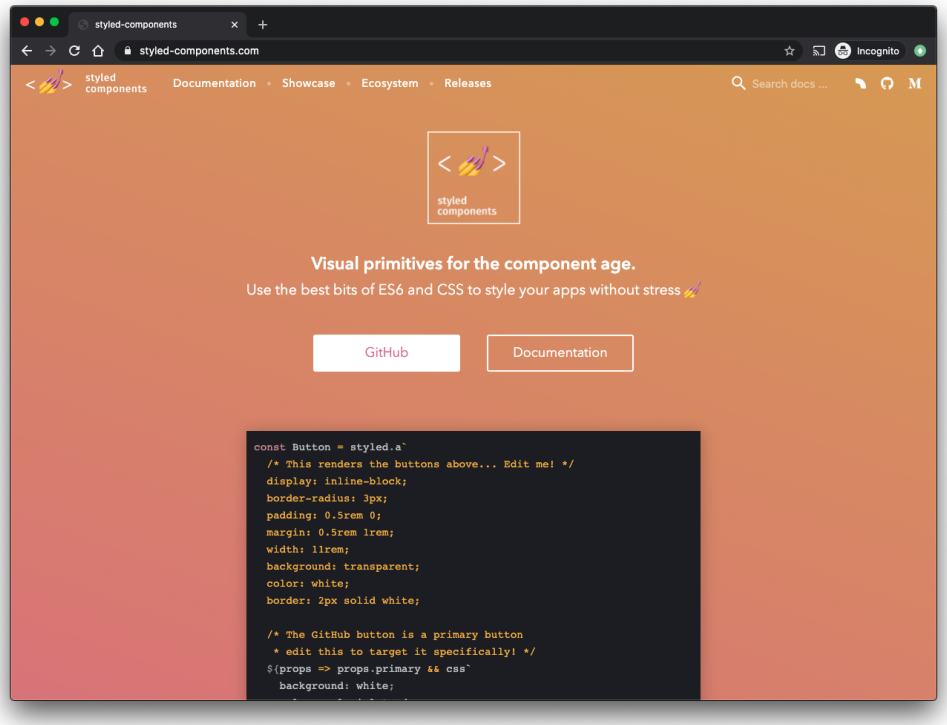
```
import React from "react";
import "./styles.css";
```

```
const App = () => {
  return (
    <div className="container">
      <h1>Hello</h1>
    </div>
  );
};

export default App;
```

```
.container {
  background: papayawhip;
  color: red;
}
```

Will work now and in the future, and as we continue, please don't forget that.

Let's now see some cursed CSS-in-JS.



Chosen: styled-components

I have been using styled-components basically since it came out. I think it's an amazing approach that addresses all my concerns with CSS-in-JS, as it has string interpolation, theming, SSR, and even global styles that are attached to the theme. It's almost the perfect solution because, also, if you don't like to use CSS in template strings, you can alternatively use styled-components in the object form, leaving the API up to you and your preferences.

All these next points will be using the template strings version, merely out of preference.

We will start from the top-down, beginning by creating a theme that we can use in all our components, as most applications usually use only a small set of colors.

Let's start by installing styled-components:

```
yarn add styled-components
```

After this, let's create a new file called `styles.js` in a `utils` folder and let's start our theme:

```
export const theme = {
  colors: {
    white: "#F1F2EB",
    grey: "#D8DAD3",
    black: "#4A4A48",
  },
  font:
    "-apple-system, BlinkMacSystemFont, 'Segoe UI',
Roboto, Oxygen, Ubuntu, Cantarell, 'Open Sans', 'Helvetica
Neue', sans-serif",
};
```

To use this theme let's now go to our `index.js` and import it along with the `ThemeProvider` from styled-components:

```
import React, { useState } from "react";
import ReactDOM from "react-dom";
import { ThemeProvider } from "styled-components";
import { theme } from "./utils/styles";

function App() {
  const [word, setWord] = useState("");
  return (
    <ThemeProvider theme={theme}>
      <div className="App">
        <h1>Write "React"</h1>
```

```
        <input value={word} onChange={(e) =>
setWord(e.target.value)} />
        <h2 react={word.toLowerCase() === "react"}>{word}
</h2>
      </div>
    </ThemeProvider>
  );
}

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

Awesome! We now have a theme. Let's start using it to declare some global styles, by which I mean styles that will apply everywhere. It's kind of like a general `styles.css`, but it also has access to our theme. For that, we can use the `createGlobalStyle` function from styled-components in our `utils/styles.js`.

Let's add our base styles:

```
export const Style = createGlobalStyle`
  body {
    text-align: center;
    margin: 0;
    background-color: ${(props) =>
props.theme.colors.black};
    color: ${(props) => props.theme.colors.white};
    font-family: ${(props) => props.theme.font}
  }
`;
```

The name has to start with an uppercase letter, as this will be a component on our DOM.

As you can see, we are constantly using `props` in almost all the rules, so we can also do something like:

```
const Style = createGlobalStyle`
  ${({ theme }) => `
    body {
      text-align: center;
      margin: 0;
      background-color: ${theme.colors.black};
      color: ${theme.colors.white};
      font-family: ${theme.font};
    }
  `}
`;
```

In the end, it's up to you how you want to do this and get the values from the props.

Let's now get our global styles and apply them to the body:

```
import React, { useState } from "react";
import ReactDOM from "react-dom";
import { createGlobalStyle, ThemeProvider } from "styled-
components";
import { theme, Style } from "./utils/styles";

function App() {
  const [word, setWord] = useState("");
```

```
  return (
    <ThemeProvider theme={theme}>
      <>
        <Style />
        <div className="App">
          <h1>Write "React"</h1>
          <input value={word} onChange={(e) =>
setWord(e.target.value)} />
          <h2 react={word.toLowerCase() === "react"}>
{word}</h2>
        </div>
      </>
    </ThemeProvider>
  );
}

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

If you check your page, you'll see the global styles applied. The idea is that now we have a styled base in which we'll style unique elements. Let's make an example with the button we have there.

The way styled-components works is that we have the `styled` function that, like in Framer Motion, gets an HTML element after it to return a styled version of that element like so:

```
const Heading = styled.h1`
  display: block;
`;
```

If you want to style another React component you can also do so by passing it as an argument to the `styled` function:

```
const Heading = styled(MyComponent)`
  display: block;
`;
```

Now that we know the basics we can create our styled input.

```
import React, { useState } from "react";
import ReactDOM from "react-dom";
import styled, { createGlobalStyle, ThemeProvider } from
"styled-components";
import { theme, Style } from "./utils/styles";

const Heading = styled.h2`
  background: black;
  padding: 8px 12px;
  color: ${(props) => props.theme.colors.white};
`;

function App() {
  const [word, setWord] = useState("");
  return (
    <ThemeProvider theme={theme}>
      <>
        <Styles />
        <div className="App">
          <h1>Write "React"</h1>
          <input value={word} onChange={(e) =>
setWord(e.target.value)} />
```

```
          <Heading react={word.toLowerCase() === "react"}>
{word}</h2>
        </div>
      </>
    </ThemeProvider>
  );
}

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

As you can see, we created a styled component with the `h2` tag, and we then use that element as we would use our `<h2>`.

Our h2 has some styles, but we also want to check that `react` prop we have and make some style changes if the value of the word equals "react". So for that, we get the `props` again and add some conditional logic based on the value of `react`, and our heading will look like this:

```
const Heading = styled.h2`
  background: black;
  padding: 8px 12px;
  color: ${(props) => props.theme.colors.white};

  ${(props) =>
    props.react &&
    css`
      background: ${props.theme.colors.white};
      color: black;
    `}
`;
```

We can also destructure our props inside the template string, like so:

```
const Heading = styled.h2`
  background: black;
  padding: 8px 12px;
  color: ${(props) => props.theme.colors.white};

  ${({ react, theme }) =>
    react &&
    css`
      background: ${theme.colors.white};
      color: black;
    `}
`;
```

As you can see, we have a new function: the `css` function. It takes strings and makes them into CSS values that styled-components can use to style our page.

There is also another way we can do it by having some ternary operators in the values of the CSS we have. In these operators we can check for the prop and pass the correct value like so:

```
const Button = styled.button`
  background: ${({ theme, react }) => (react ?
theme.colors.white : "black")};
  border: none;
  padding: 8px 12px;
  color: ${({ theme, react }) => (react ? "white" :
theme.colors.white)};
  transition: all 200ms ease;
`;
```

Both these options will return the same values in your final CSS, so choose the one you find best.

In my opinion, styled-components is the perfect combination of CSS's power and the power of JavaScript, creating an excellent way to manage our CSS.

What if you want to style just one element but don't really want to go through the trouble of adding a whole component for it, BUT you also know that inline styles are evil?

That's where the `css` props comes into play!

For this to work you will need to either install the babel plugin:

```
yarn add babel-plugin-styled-components --dev
```

Or if you are using something like `create-react-app` that does not allow you to change the babel config easily, you need to change your imports to be from `styled-components/macro`.

If we wanted to use it on our app, we would change the imports to:

```
import styled, { css } from "styled-components/macro";
```

Then in our elements, we could use the `css` props, and in there we can write any style and behind the scenes `styled-components` will turn that into classes that will be applied:

```
<input
  css={`
    border: none;
```

```
    padding: 8px;
  `}
  value={word}
  onChange={(e) => setWord(e.target.value)}
/>
```

If you prefer an object based approach this method supports both so you can also write it like this:

```
<input
  css={{
    border: "none",
    padding: 8,
  }}
  value={word}
  onChange={(e) => setWord(e.target.value)}
/>
```

I find myself using this a lot when I know an element only needs an adjustment, and it won't be added anywhere else.

It's incredible to have the ability to be this granular with ease and without polluting our DOM and ruining the CSS specificity of our site.

What about Emotion? Why not use that?

To be honest, I don't really have an answer here. Both styled-components and Emotion have the same API, to a point where you can actually interchange them easily.

Instead of importing:

```
import styled from "styled-components";
```

You do:

```
import styled from "@emotion/styled";
```

Also both support object based styling like so:

```
const Button = styled.button({
  color: "white",
});
```

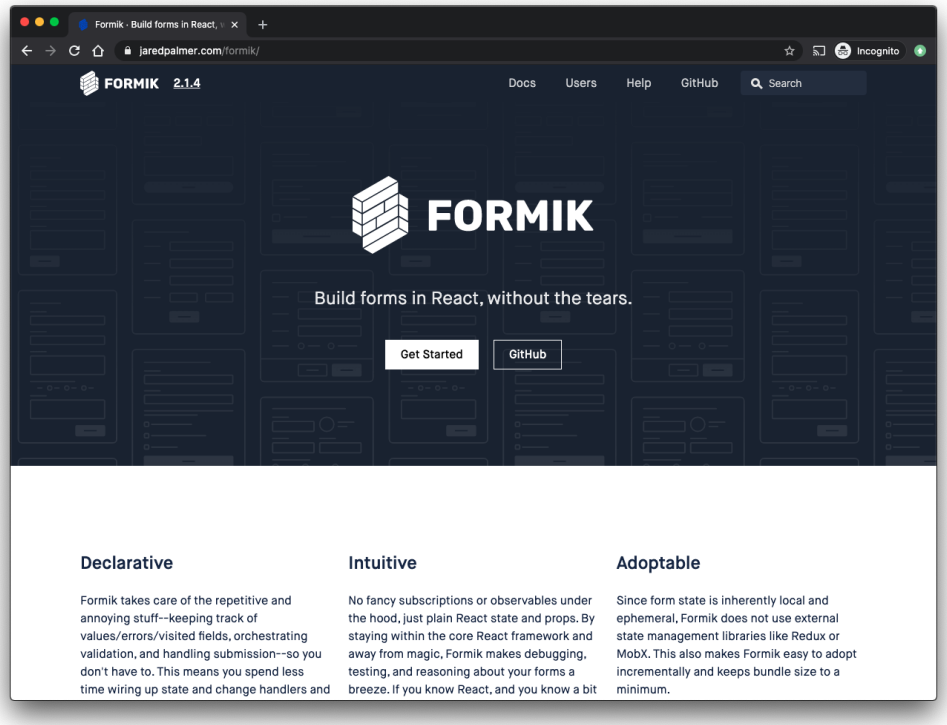
On top of this, both support the CSS prop that allows you to style individual elements without creating inline styles in the actual HTML element but instead creating classes.

You can take a look at Emotion in their docs.

As for a choice here, I merely went with styled-components because it's the one I started using, and also since I am more of string-based CSS person styled-components was the first one that supported it. At this time, they have the same API, so honestly, it's anyone's game.

Link to CodeSandbox

Forms



Chosen: Formik

Forms! No one likes to write them, not even to fill them, and honestly, React does not help us at all with forms, so a bunch of packages have sprung out to help us get forms done without crying. That's exactly what Formik promises, and it delivers really well.

For this example, to see how Formik works, we will be making a registration form with validation.

Let's start by adding Formik:

```
yarn add formik
```

Now that we have Formik, let's see how our JSX looks without any events added:

```
import React from "react";
import ReactDOM from "react-dom";
import "./styles.css";

const App = () => {
  return (
    <main className="App">
      <h1>Sign Up</h1>
      <form>
        <label htmlFor="name">Name</label>
        <input type="text" id="name" />
        <label htmlFor="email">Email</label>
        <input type="email" id="email" />
        <label htmlFor="password">Password</label>
        <input type="password" id="password" />
        <button type="submit">Sign Up</button>
      </form>
    </main>
  );
};

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

As you can see, we will have the standard form with name, email, and password fields that we want to hook up to Formik. We should start by importing the Formik hook and defining our base Formik state:

```
import React from "react";
import ReactDOM from "react-dom";
import { useFormik } from "formik";
```

```
import "./styles.css";

const App = () => {
  const formik = useFormik({
    initialValues: {
      name: "",
      email: "",
      password: "",
    },
    onSubmit: (values) => {
      alert(JSON.stringify(values, null, 2));
    },
  });
  return (
    <main className="App">
      <h1>Sign Up</h1>
      <form>
        <label htmlFor="name">Name</label>
        <input type="text" id="name" />
        <label htmlFor="email">Email</label>
        <input type="email" id="email" />
        <label htmlFor="password">Password</label>
        <input type="password" id="password" />
        <button type="submit">Sign Up</button>
      </form>
    </main>
  );
};

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

By adding this hook, Formik will now give us some functions for regular events like `onChange` and `onBlur`, and it will also keep track of our values.

The `initialValues` you see in the hook map to values of the same name that Formik will track when added to an input.

Better to show with code:

```
import React from "react";
import ReactDOM from "react-dom";
import { useFormik } from "formik";
import "./styles.css";

const App = () => {
  const formik = useFormik({
    initialValues: {
      name: "",
      email: "",
      password: "",
    },
    onSubmit: (values) => {
      alert(JSON.stringify(values, null, 2));
    },
  });

  const { values } = formik;
  return (
    <main className="App">
      <h1>Sign Up</h1>
      <form>
        <label htmlFor="name">Name</label>
        <input
```

```
          onChange={formik.handleChange}
          value={values.name}
          onBlur={formik.handleBlur}
          type="text"
          id="name"
        />
        <label htmlFor="email">Email</label>
        <input
          onChange={formik.handleChange}
          value={values.email}
          onBlur={formik.handleBlur}
          type="email"
          id="email"
        />
        <label htmlFor="password">Password</label>
        <input
          onChange={formik.handleChange}
          value={values.password}
          onBlur={formik.handleBlur}
          type="password"
          id="password"
        />
        <button type="submit">Sign Up</button>
      </form>
    </main>
  );
};

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

Our values are tracked, but that is a lot of boilerplate, so we clean it up with the `getFieldProps` functions that Formik will also give us. This function will

make our inputs looks like this:

```
import React from "react";
import ReactDOM from "react-dom";
import { useFormik } from "formik";
import "./styles.css";

const App = () => {
  const formik = useFormik({
    initialValues: {
      name: "",
      email: "",
      password: "",
    },
    onSubmit: (values) => {
      alert(JSON.stringify(values, null, 2));
    },
  });

  const { getFieldProps } = formik;
  return (
    <main className="App">
      <h1>Sign Up</h1>
      <form>
        <label htmlFor="name">Name</label>
        <input {...getFieldProps("name")} type="text"
id="name" />
        <label htmlFor="email">Email</label>
        <input {...getFieldProps("email")} type="email"
id="email" />
        <label htmlFor="password">Password</label>
        <input {...getFieldProps("password")}
```

```
type="password" id="password" />
        <button type="submit">Sign Up</button>
      </form>
    </main>
  );
};

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

This function attaches events like `onBlur`, `onChange`, and all the necessary hooks that Formik needs to track everything properly. With this done, let's add the submit function to our form and test it with the alert defined up top:

```
      <form onSubmit={formik.handleSubmit}>
```

You will now see the values you typed in the `alert` returned by the `onSubmit` event. You can use this to send your field data to an API.

So far, I think this is a pretty good developer experience for forms while also offering more complex functions, making Formik a very complete package.

The final thing I want to add is field validation. This `validate` function must return an `errors` object with the same values as the fields we have in our form, so if I want to validate our password field, I could add a function like this:

```
const validate = (values) => {
  const errors = {};
  if (!values.password) {
    errors.password = "Required";
  } else if (values.password.length < 5) {
```

```
    errors.password = "Password must be at least 5
characters long";
  }
  return errors;
};
```

Now, let's pass this function to `useFormik` like so:

```
const formik = useFormik({
  initialValues: {
    name: "",
    email: "",
    password: "",
  },
  validate,
  onSubmit: (values) => {
    alert(JSON.stringify(values, null, 2));
  },
});
```

You now have validation in your forms. The next step is showing errors on the page - for that we need to access the `errors` object and get the value we want like this:

```
<label htmlFor="name">Name</label>
<input type="text" id="name" {...getFieldProps("name")} />
{formik.errors.name && formik.touched.name ? <span>
{formik.errors.name}</span> : null}
```

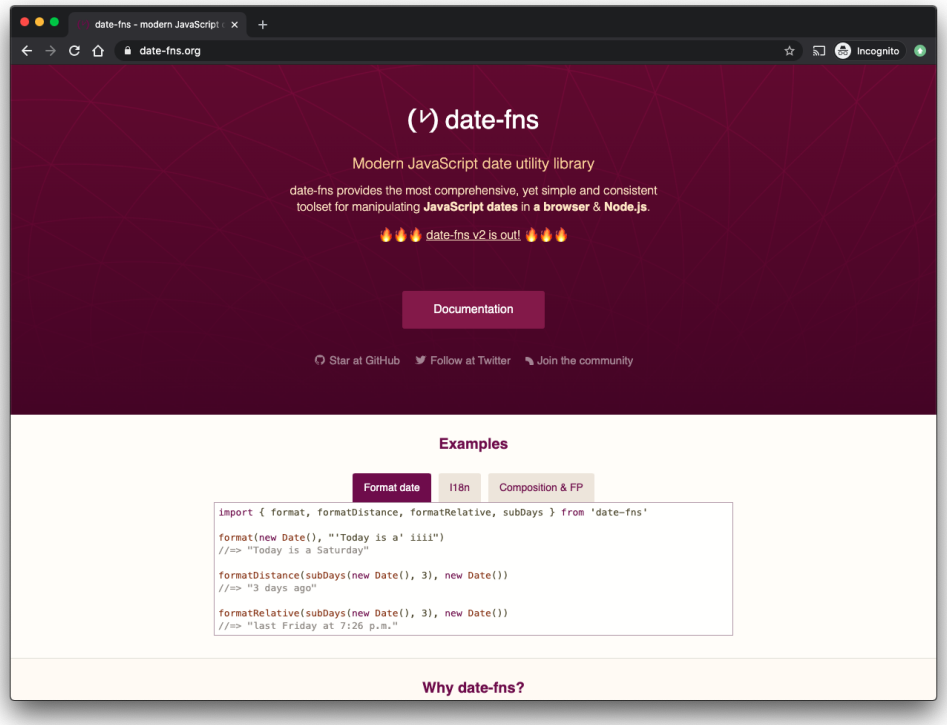
Here we check if there is an error and if the input has been touched. If both of these are true, we show an error to the user.

I like the fact that the `validate` function is pure JavaScript. You can also install libraries like Yup to help you in validating these strings.

And now a meme about bad forms:

Link to CodeSandbox

Dates



Chosen: date-fns

Dates are hard - dates are *really* hard - so a good date library goes a long way. At first, and for a long time, I used Moment.js. I stopped because moment is a massive package, and most of the time, I just wanted to format some dates, but the only way to use it was to import the entire package, you can't import individual functions.

Date-fns is a package with the same formatting and calculation options as moment, but one that is completely tree-shakeable and can be imported function by function.

The only thing it doesn't have as part of the core package is timezone support, so if this is something you need, it may be better to stick with moment.

Let's make a straightforward example where I format the current date and see how many days there are until a specific date.

Let's start by installing date-fns:

```
yarn add date-fns
```

Then I will create a variable to hold the date for this moment and call it `now`:

```
const now = new Date();
```

This is the date we will be manipulating and showing as we please, so let's now import the `format` function from `date-fns` and use it:

```
import React from "react";
import ReactDOM from "react-dom";
import { format } from "date-fns";

const now = new Date();

function App() {
  return (
    <div className="App">
      <p>Today is {format(now, "dd/MM/yyyy")}</p>
    </div>
  );
}

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

So what I do here is pass two parameters to the `format` function: the first one is the date that is to be formatted, and the second one is how we want to format

it. Here, `date-fns` uses a series of tokens that you can see in the Unicode date field symbols documentation.

In this case, I format it to look like `07/01/2020`: the `dd` means I want to get the day of the month as a number and, if smaller than 10, prepend a zero. Same idea with the month, where I say `MM` because one `M` will return me the month without a prepended zero.

The last part is the year and I want that in four digits. Between these, I placed `/` since this is how I want to separate the date parts, so here you could pass something else like `-` and the function would look like:

```
format(now, "dd-MM-yyyy");
```

It does sound confusing and like a lot to memorize, but in all honesty, I never remember these things and resort to the documentation in case of any doubts - they also have pretty good documentation.

Let's now see how many days there are until Christmas. First, let's create a new variable called `xmas`, and its value will be a new date that will resolve to Christmas Day.

```
const xmas = new Date(2020, 11, 25, 0, 0, 0);
```

Here we pass the year, month, day, hour, minute, and second, we want the new date object to have. The month's values are zero-based just to make it slightly more confusing, and that is why the month indicated is 11 instead of 12

To now see how many days are left until your presents arrive we need the `formatDistanceStrict` function from `date-fns`, and we can use it like so:

```
formatDistanceStrict(xmas, now);
```

Here the first parameter is the date you want to count, and the second one is the one to make the math against.

By default, this will give the time in months, and since we want to see this in days we can pass some options:

```
formatDistanceStrict(xmas, now, {
  unit: "day",
});
```

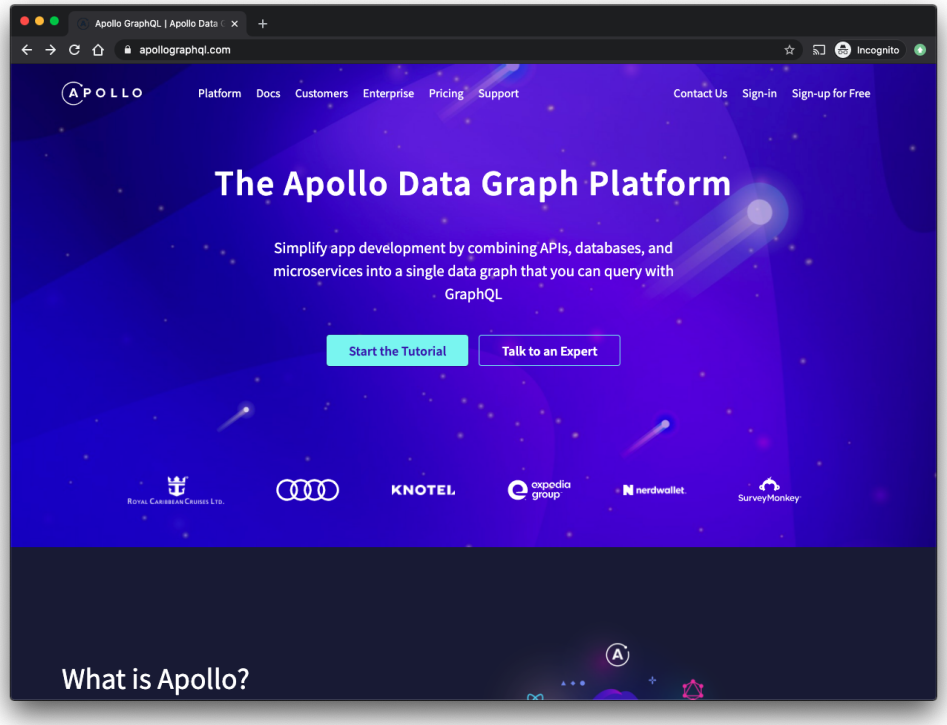
With this we can now add this function we created to our react code:

```
import React from "react";
import ReactDOM from "react-dom";
import { format, formatDistanceStrict } from "date-fns";

import "./styles.css";

const now = new Date();
const xmas = new Date(2020, 12, 25, 0, 0, 0);

function App() {
  return (
    <div className="App">
      <p>Today is {format(now, "dd/MM/yyyy")}</p>
      <p>
        Days until my Christmas:{" "}
        {formatDistanceStrict(xmas, now, {
          unit: "day",
```

```
        })}
      </p>
    </div>
  );
}

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

See the code on CodeSandbox

GraphQL



Chosen: React Apollo

GraphQL is amazing! It lets you build faster and more responsive UIs, and it allows the client-side to decide which data it wants to get instead of getting

everything at once (over fetching).

I use Apollo because their API is pretty solid to help you get started to implementing it in the frontend.

To demonstrate how the fetching of data in Apollo works, we will create a simple application that calls a GraphQL API where I have a list of names. You can see the code for that API on CodeSandbox.

Let's start by installing all the packages we need, and then we'll go over why we need each one:

```
yarn add apollo-boost @apollo/react-hooks graphql
```

- `apollo-boost`: Package that contains what you need to initialize the Apollo client and also some utilities.
- `@apollo/react-hooks`: The hooks that are used to do queries and mutations.
- `graphql`: To parse the GraphQL language strings, we will be using.

Now that we have all this installed, we first need to create an Apollo Client, pointing to our GraphQL API:

```
import React from "react";
import ReactDOM from "react-dom";
import ApolloClient from "apollo-boost";
import App from "./App";

const client = new ApolloClient({
```

```
  uri: "https://6qvmj.sse.codesandbox.io/",
});
```

One thing you need to know is that `apollo-boost` is framework agnostic, creating a standard Apollo client that can be plugged into any framework. To let Apollo knows we are using React, we need to import the `Provider` from `@apollo/react-hooks` and wrap it around our application like this:

```
import React from "react";
import ReactDOM from "react-dom";
import { ApolloProvider } from "@apollo/react-hooks";
import ApolloClient from "apollo-boost";
import App from "./App";

const client = new ApolloClient({
  uri: "https://6qvmj.sse.codesandbox.io/",
});

const Main = () => (
  <ApolloProvider client={client}>
    <App />
  </ApolloProvider>
);

const rootElement = document.getElementById("root");
ReactDOM.render(<Main />, rootElement);
```

Having this helps all our components know where to make the queries, mutations, and any other global configuration we may have for our application.

Let's now move to our `App.js` and write our first query:

```
query AllPeople {
  all {
    id
    firstName
  }
}
```

In this query we are getting all the movies, then we cherry-pick what we want to show, which is:

- `id` - To add keys to our react components
- `firstName` - First name of the person

If you copy this into the API on CodeSandbox, you will see it will give a list of names and an id.

Now that we have this query, we need to write it so that Apollo will understand: for that, we need `gql`, a JavaScript template literal tag that parses GraphQL query strings and transforms them into valid GraphQL.

```
import React from "react";
import { gql } from "apollo-boost";

const PEOPLE = gql`
  query AllPeople {
    all {
      id
      firstName
    }
  }
`;
```

This variable is now ready to be used in Apollo with the `useQuery` hook, so let's import it and use it:

```
import React from "react";
import { useQuery } from "@apollo/react-hooks";
import { gql } from "apollo-boost";

const PEOPLE = gql`
  query AllPeople {
    all {
      id
      firstName
    }
  }
`;

export default function App() {
  const { loading, data, error } = useQuery(PEOPLE);
  if (loading) return "loading";
  if (error) return "Oh no :(";
  return JSON.stringify(data, null, 2);
}
```

By looking at the browser, you can see we have a massive mess of objects and arrays, but we can see that our people are all inside an `all` object which we can `map` over to get the info about our people:

```
import React from "react";
import { useQuery } from "@apollo/react-hooks";
import { gql } from "apollo-boost";
```

```
const movies = gql`
  query AllMovies {
    all(filter: { sortBy: rating }) {
      id
      title
      rating
      info {
        poster_path
      }
    }
  }
`;

export default function App() {
  const { loading, data } = useQuery(PEOPLE);
  if (loading) return "loading";
  return (
    <>
      <h1>Random People</h1>
      <ul>
        {data.all.map((person) => (
          <li>{person.firstName}</li>
        ))}
      </ul>
    </>
  );
}
```

As you can see, getting this information was relatively painless after we dived a bit into the basics of GraphQL.

We can now get information from our backend, but we should also learn how to update information. For our example, let's see how we can add a person to our

list.

In GraphQL, any operation that changes the data is called a mutation. We can then add a mutation called `newPerson`, and this will take a first name and create a new person in our database.

Remember PHP? Well, in PHP, you wrote variables using `$` before the variable names, and that's also how the GraphQL query languages decided to set the syntax for variables.

Let's first look at our mutation to then go over it:

```
mutation newPerson($firstName: String) {
  add(firstName: $firstName) {
    id
  }
}
```

The first thing after the keyword mutation is the name of that mutation (`newPerson`). This name is useful for debugging purposes.

Then we open a parenthesis like if we were calling a JavaScript function, in here, we pass our parameters in a key-value notation that says which variables will be passed to our mutation, along with their type.

If you have used TypeScript, this may look very familiar, but don't forget that in GraphQL, the types are written with the first letter in uppercase, unlike in TypeScript.

We then call our mutation and say that the `firstName` it needs will come from the `$firstName` variable we will pass to it. This mutation returns the same fields as the `all` query, but in this case, I just want the `id`.

We are ready to create this mutation in our `App.js`:

```
const ADD_PERSON = gql`
  mutation newPerson($firstName: String) {
    add(firstName: $firstName) {
      id
    }
  }
`;
```

Let's now add a form that we will use to save the new first name to pass to the mutation:

```
export default function App() {
  const { loading, data } = useQuery(PEOPLE);
  const [name, setName] = useState("");

  const addPerson = (e) => {
    e.preventDefault();
    setName("");
  };

  if (loading) return "loading";
  return (
    <div>
      <h1>Add a person</h1>
      <form onSubmit={addPerson}>
        <input
          required
          onChange={(e) => setName(e.target.value)}
          value={name}
```

```
        />
        <button type="submit">Add</button>
      </form>
      <h1>Random People</h1>
      <ul>
        {data.all.map((person) => (
          <li>{person.firstName}</li>
        ))}
      </ul>
    </div>
  );
}
```

Now that we have the form set up, we need to call our mutation, for that we use the `useMutation` hook from Apollo:

```
const [AddPersonMutation] = useMutation(ADD_PERSON);
```

The `useMutation` hook returns its values in a similar way to the `useState` hook, as in it: returns an array that we can deconstruct.

The first item of the array is always going to be the function that will call the mutation itself, and in here we called it `AddPersonMutation`, after that we can also get the `context` and that is an object that contains:

- `data` - This will be what we get from the mutation. In our example, we asked for the ID of the new person created.
- `loading` - Loading will be `true` while our mutation is in progress.
- `error` - This will contain any errors we may have in the course of our mutation.

In our case, we will only use `loading`. Since a variable already exists with this name, we will need to rename it like so:

```
const [AddPersonMutation, { loading: loadingMutation }] =
useMutation(
  ADD_PERSON
);
```

We are now ready to call the `AddPersonMutation` mutation and send the variables it needs:

```
const addPerson = (e) => {
  e.preventDefault();
  AddPersonMutation({
    variables: {
      firstName: name,
    },
  });
  setName("");
};
```

As a small final touch, let's just have some UI feedback when the mutation is happening by using the `loadingMutation` variable to toggle the text in the button between `Add` and `Loading`, like so:

```
<button type="submit">{loadingMutation ? "Loading" :
"Add"}</button>
```

If you now try to write a name and then reload the page, you can see it does indeed work, and we have a new name added to the list, yet it's not the best user experience to have to reload the page to see our new name in the screen.

We can fix this by telling Apollo to fetch the query again once the mutation is finished so that when the query comes back, it already has the new name in it.

We can do that by passing a second parameter to `useMutation`. This second parameter is an `options` param, which is an object from which we want to get the `refetchQueries` property. And tell it to go and fetch the `AllPeople` query. `refetchQueries` runs after the mutation has been finished.

```
const [AddPersonMutation, { loading: loadingMutation }] =
useMutation(
  ADD_PERSON,
  {
    refetchQueries: ["AllPeople"],
  }
);
```

If you try to add a name again, you can see that it will fetch the new name right after the mutation is done, and our UI updates.

There is a very different way of handling the UI updates after a mutation, which involves updating the cache that Apollo creates, you can read about it in their docs.

I did not include it because I only use it when the query that needs to be refetched is quite big and can cause performance issues, in this particular case, I would go with updating the Apollo cache because that does not require another costly fetch call. For any other case, I try to avoid messing with the cache as it's very brittle since you're messing with something that's not even created by you: it's the magical Apollo cache.

If you want to learn more, I have a couple of free videos online where I taught GraphQL in Kyiv's workshop on YouTube, where I go over making the node

server and the frontend part.

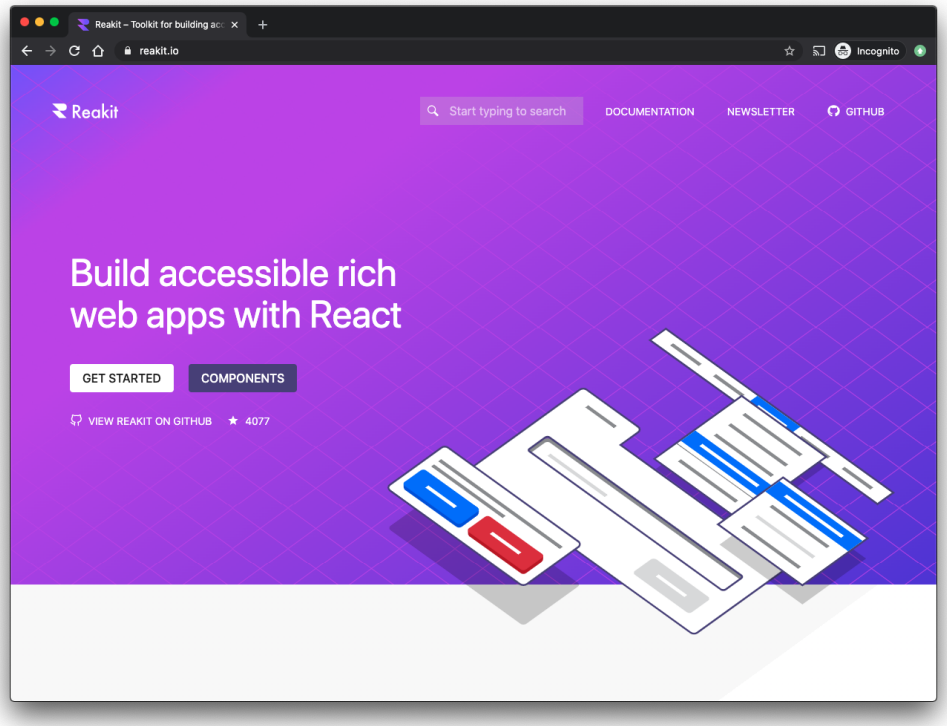
Link to CodeSandbox

UI Toolkits

UI Toolkits can be useful for various reasons: getting a website up and running faster, maintaining a consistent design, or even helping with accessibility.

For that reason, I decided to divide this part into two different categories: one for toolkits that are already styled, and one for toolkits that aren't and let you put all your styles on top of their components.

Unstyled



Chosen: Reakit

When it comes to unstyled toolkits, I have been using Reakit for a while as it helps me a lot with getting accessibility right in trickier things like modals and dropdown menus.

As our usual first step, let's start by installing `reakit`:

```
yarn add reakit
```

Because of this toolkit's simplicity by its lack of both styling and global state, there is no need to add a `Provider` component to take full advantage of it. We can start making some accessible components right away.

Let's start with a modal:

```
import React from "react";
import { useDialogState, Dialog, DialogDisclosure } from
"reakit/Dialog";

export default function App() {
  const dialog = useDialogState();

  return (
    <>
      <DialogDisclosure {...dialog}>Open
dialog</DialogDisclosure>
      <Dialog {...dialog} aria-label="Welcome Modal">
        I am a modal
      </Dialog>
    </>
  );
}
```

You can now see a button that, when clicked, opens a modal and traps the focus in it as specified by the accessibility guidelines. You can then style it as you please to fit with the design of your project.

Let's do one more example and show how to make some tabs using `reakit`:

```
import React from "react";
import { useTabState, Tab, TabList, TabPanel } from
"reakit/Tab";

export default function App() {
  const tab = useTabState();
  return (
    <>
      <TabList {...tab} aria-label="My tabs">
        <Tab {...tab} stopId="tab1">
          Tab 1
        </Tab>
        <Tab {...tab} stopId="tab2">
          Tab 2
        </Tab>
        <Tab {...tab} stopId="tab3">
          Tab 3
        </Tab>
      </TabList>
      <TabPanel {...tab} stopId="tab1">
        Tab 1
      </TabPanel>
      <TabPanel {...tab} stopId="tab2">
        Tab 2
      </TabPanel>
      <TabPanel {...tab} stopId="tab3">
```

```
          Tab 3
        </TabPanel>
      </>
    );
}
```

When you check the live version of this in your browser, you can see that you have three tabs, but none is selected by default, so the panel shows up empty. We can fix this by passing a parameter to the `useTabState` function:

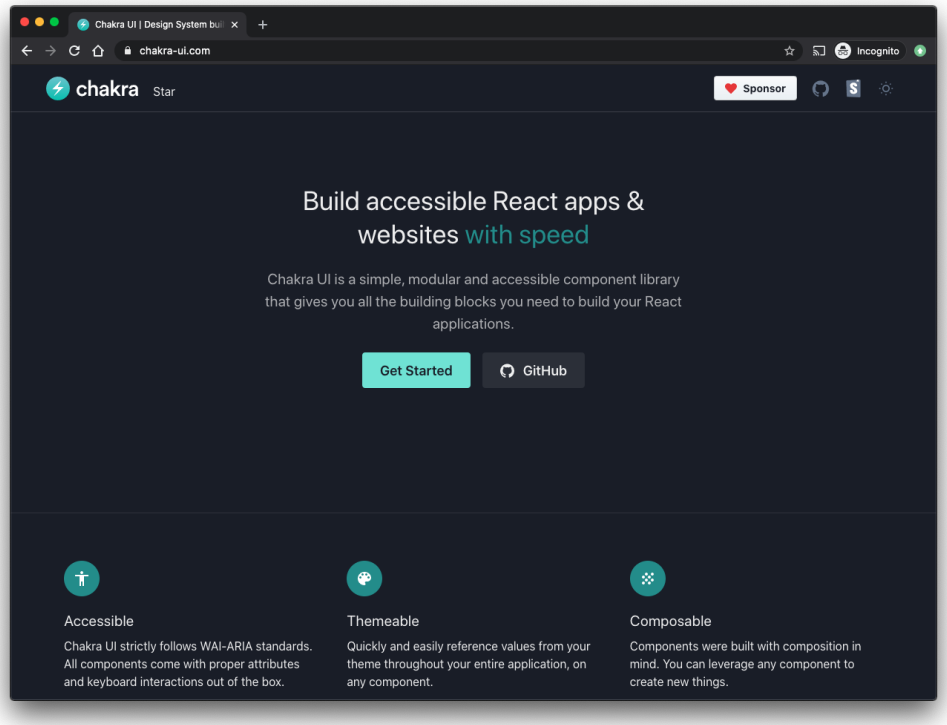
```
const tab = useTabState({ selectedId: "tab1" });
```

By doing this, you can now see that now the first tab shows up as selected by default on load, then you can swap the visible tab more intuitively.

If you are wondering how you can style these elements, `reakit` has a page with all the ways they can be styled in their documentation

Link to CodeSandbox

Styled



Chosen: Chakra UI

I have used plenty of styled options for UI libraries ever since the Bootstrap days, and I found `chakra-ui` to be the cleanest in terms of design, while also being the most composable one.

One thing I appreciate is their commitment to style props: any element you use from Chakra can be passed a massive variety of props to control properties like margin and background, which makes your JSX super understandable, and your styles become cleaner since they will be needed only in very specific cases.

This idea starts with their `Box` component, which is just a `div`, but you can do everything via its props.

For example, this is how that `Box` component can be styled as a red circle:

```
<Box
  width={200}
  height={200}
  margin="30px auto"
  backgroundColor="tomato"
  borderRadius="50%"
  borderColor="gray"
  borderWidth={10}
/>
```

Let's now start by making the same examples as before, creating a modal and a dropdown using `chakra-ui`.

First, we need to install all the dependencies and peer dependencies that it uses:

```
yarn add @chakra-ui/core @emotion/core @emotion/styled
emotion-theming
```

Don't get scared by the number of dependencies, emotion is used under the covers in `chakra-ui`, so we need to have it. You will only be touching the `@chakra-ui/core` package.

Let's start by adding the `ThemeProvider` with the default theme that comes bundled in `chakra-ui`.

```
import React from "react";
import { ThemeProvider } from "@chakra-ui/core";

export default function App() {
  return (
    <ThemeProvider>
```

```
      <h1>Hello CodeSandbox</h1>
      <h2>Start editing to see some magic happen!</h2>
    </ThemeProvider>
  );
}
```

Great, we just ruined the default CodeSandbox style, but we do have on trick up our sleeve that will reset everything. `chakra-ui` exports a `CSSReset` to make sure all their components will have the same styling no matter the browser, which is super handy to give us a baseline to keep working. Let's import it to use it:

```
import React from "react";
import { ThemeProvider, CSSReset } from "@chakra-ui/core";

export default function App() {
  return (
    <ThemeProvider>
      <CSSReset />
      <h1>Hello CodeSandbox</h1>
      <h2>Start editing to see some magic happen!</h2>
    </ThemeProvider>
  );
}
```

Looking better, we can work with this! Let's now add a container with the amazing `Box` component `chakra-ui` gives us to make everything centered:

```
import React from "react";
import { ThemeProvider, CSSReset, Box } from "@chakra-
ui/core";
```

```
export default function App() {
  return (
    <ThemeProvider>
      <CSSReset />
      <Box maxWidth="80%" width="600px" margin="50px auto"
textAlign="center">
        <h1>Hello CodeSandbox</h1>
        <h2>Start editing to see some magic happen!</h2>
      </Box>
    </ThemeProvider>
  );
}
```

We are getting somewhere; now we can start working on adding our first component, the dropdown.

This component gets imported from `chakra-ui` as the `Reactified` version of a `select` tag, and by that I mean it has a capital letter in the beginning, and it will be a `select` HTML element when used in the page.

> *Because `chakra-ui` has an extreme focus on accessibility, it uses the default styles for option tags, and honestly, there is nothing wrong with using those default styles. If someone says otherwise, please refer them to https://doineedacustomselect.com/.*

Let's use the Select now:

```
import React from "react";
import { ThemeProvider, CSSReset, Box, Select } from
"@chakra-ui/core";
```

```
export default function App() {
  return (
    <ThemeProvider>
      <CSSReset />
      <Box maxWidth="80%" width="600px" margin="50px auto"
textAlign="center">
        <Select
          placeholder="Berlin Teams"
          onChange={(e) => console.log(e.target.value)}
        >
          <option value="union">Union Berlin</option>
          <option value="hertha">Hertha Berlin</option>
        </Select>
      </Box>
    </ThemeProvider>
  );
}
```

As you can see, they made it work like a normal select would in pure HTML with the addition of a very useful `placeholder` tag that allows you to define the state of the select when it has no value.

Let's now build a modal using `chakra-ui` to finish our simple use case.

The idea of `chakra-ui` is to be super composable, so we will have to import many things, but don't be scared by this, all the possibilities are well explained in their docs.

We can now proceed with the actual modal and set it to be open by default, for that we will need to do this:

```
import React from "react";
import {
  Modal,
  ModalOverlay,
  ModalContent,
  ModalHeader,
  ModalFooter,
  ModalBody,
  ModalCloseButton,
} from "@chakra-ui/core";

export default function ModalComponent() {
  return (
    <>
      <Modal size="xs" isOpen>
        <ModalOverlay />
        <ModalContent>
          <ModalHeader>Hello</ModalHeader>
          <ModalCloseButton />
          <ModalBody>I am a modal</ModalBody>

          <ModalFooter>
            <Button variantColor="blue">Close</Button>
          </ModalFooter>
        </ModalContent>
      </Modal>
    </>
  );
}
```

As you can see, all these components together will make us a nice modal, but right now, it's still kinda useless to have this pretty element with some nice

buttons that don't have any behavior.

To make it behave like a modal, we can import hook called `useDisclosure`, which will help us handling common open, close, or toggle scenarios. This hook works with a bunch of other components in `chakra-ui`. In this case, it will allow us to control the modal and its states.

```
const { isOpen, onOpen, onClose } = useDisclosure();
```

Let's work with these by adding a button to control the modal:

```
import React from "react";
import {
  Modal,
  ModalOverlay,
  ModalContent,
  ModalHeader,
  ModalFooter,
  ModalBody,
  ModalCloseButton,
  Button,
  useDisclosure,
} from "@chakra-ui/core";

export default function ModalComponent() {
  const { isOpen, onOpen, onClose } = useDisclosure();
  return (
    <>
      <Button mt="3" onClick={onOpen}>
        Open Modal
      </Button>
```

```
      <Modal size="xs" isOpen={isOpen} onClose={onClose}>
        <ModalOverlay />
        <ModalContent>
          <ModalHeader>Hello</ModalHeader>
          <ModalCloseButton />
          <ModalBody>I am a modal</ModalBody>

          <ModalFooter>
            <Button variantColor="blue" onClick={onClose}>
              Close
            </Button>
          </ModalFooter>
        </ModalContent>
      </Modal>
    </>
  );
}
```

I believe that the real power of `chakra-ui` comes in the flexibility given by the `Box` component and all the ways in which you can manipulate it to position and style it in many ways.

The usage of `chakra-ui` becomes a very customizable experience, given the fact that it's designed to extend everything from this base component.

One final trick with `chakra-ui` that I wanna talk about is the magical `useColorMode`. It allows us to create a light/dark theme switch.

Let's implement it in our little app by adding `ColorModeProvider` to our `App.js`:

```
import React from "react";
import {
  ThemeProvider,
  CSSReset,
  Box,
  ColorModeProvider,
} from "@chakra-ui/core";
import Dropdown from "./Dropdown";
import Modal from "./Modal";

export default function App() {
  return (
    <ThemeProvider>
      <ColorModeProvider>
        <CSSReset />
        <Box
          width={200}
          height={200}
          margin="30px auto"
          backgroundColor="tomato"
          borderRadius="50%"
          borderColor="gray"
          borderWidth={10}
        />
        <Box maxWidth="80%" width="600px" margin="50px
auto" textAlign="center">
          <Dropdown />
          <Modal />
        </Box>
      </ColorModeProvider>
    </ThemeProvider>
```

```
  );
}
```

If you have your OS set to dark mode, you will see that all of our components default to it as well now.

That was easy, ah?

We can also add a button to toggle the color mode with the `useColorMode` hook, which returns a function to change both the theme and the variable that holds the current state of the theme being used.

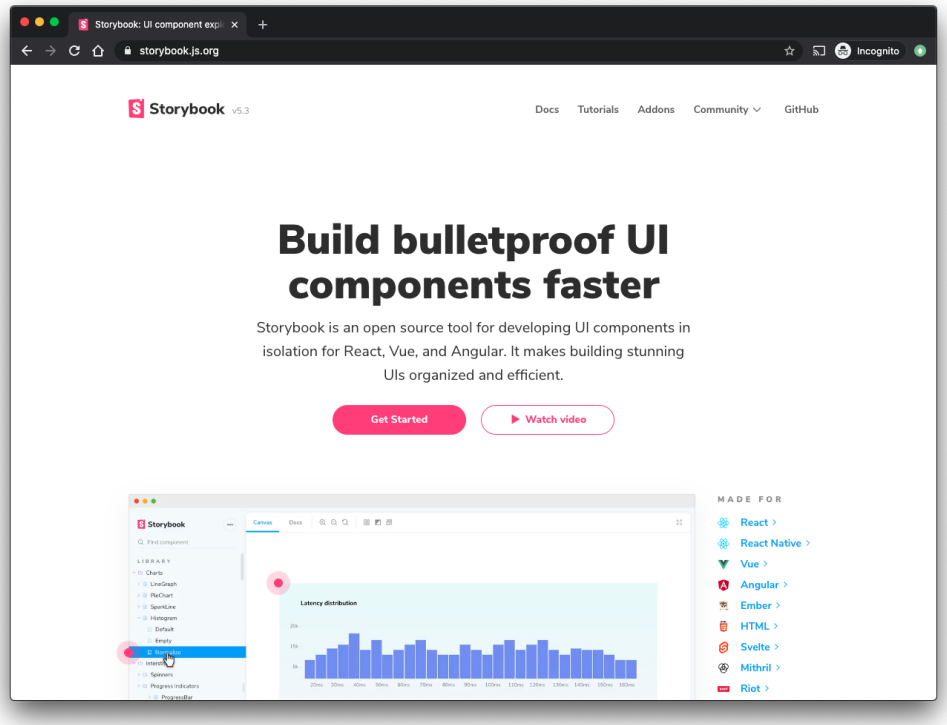
Let's add this button next to our open modal button:

```
const { colorMode, toggleColorMode } = useColorMode();

...

<Button mt="3" onClick={toggleColorMode}>
  Toggle {colorMode === "light" ? "Dark" : "Light"}
</Button>
```

Now by clicking this button, you can switch our theme, and honestly, it's amazing how enjoyable it is to work with `chakra-ui`. The tool stays out of your way and allows you to build apps with ease and beauty.

Link to CodeSandbox

Component Playgrounds



Chosen: Storybook

By now, Storybook has settled itself in the React community; it's a widely used tool to document and test your components in isolation.

The usefulness of Storybook comes precisely from that isolation. You can write your components, test them in Storybook, and then put them in your app. This way, you know that if your component works as intended in Storybook but not in the app, the problem is in the app. It also helps with heavy testing without reloading the page or navigating somewhere all the time. It's just a leaner way to test your components comprehensively.

We will use it in this example with CRA, and this means we will create what Storybook calls stories inside our app and use all the `react-scripts` magic to help us along the way.

Let's start by creating a new app using CRA:

```
create-react-app component-lib && cd component-lib/
```

Once this is done, we need some extra dependencies from Storybook, so let's install those:

```
yarn add @storybook/react @storybook/preset-create-react-app
```

The first one is the Storybook library for React, and the second one is a way to tell Storybook to use all the Webpack configs that come from CRA, so we don't need to write any.

To run Storybook, you can now add a script to your `package.json`:

```
  "start:storybook": "start-storybook -p 8080",
```

This script will start the Storybook application, running it in port `8080`.

We need one more configuration, and they will go into a new folder we need to create called `.storybook`. In here, let's create a file called `main.js`, which is where we tell Storybook what add-ons to use. In this case, we want to use the `@storybook/preset-create-react-app`.

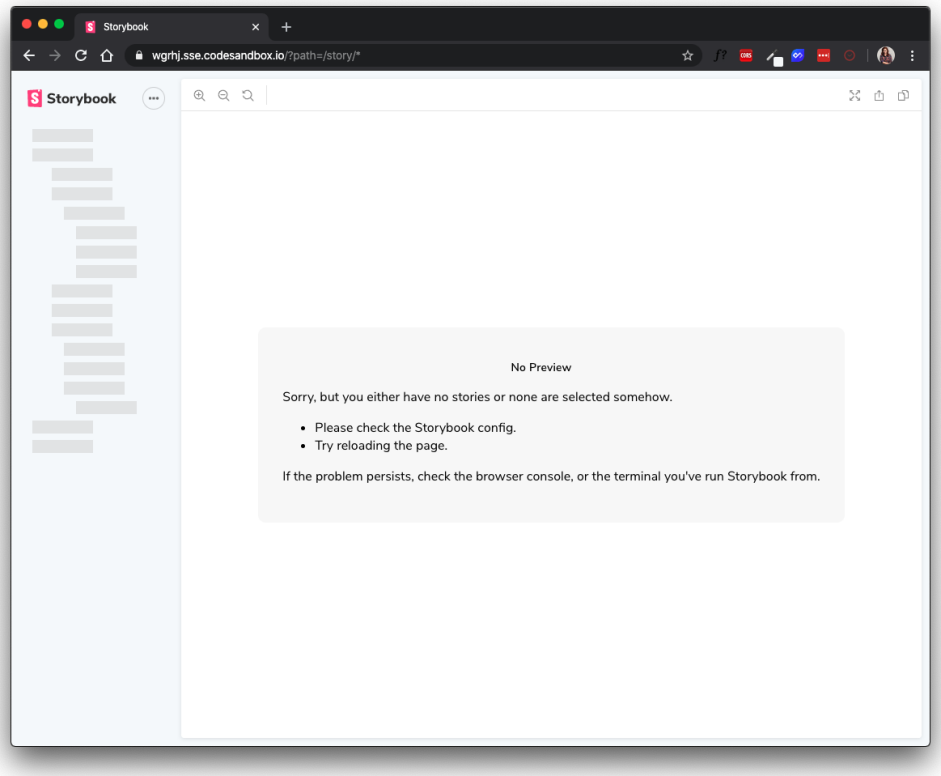
We also need to tell Storybook where to get our `stories`; these stories will be our component's renders.

Our `.storybook/main.js` will look like so:

```
module.exports = {
  addons: ["@storybook/preset-create-react-app"],
```

```
  stories: ["../src/**/*.stories.js"],
};
```

We are now ready to start Storybook. Even though it's empty, you should see something like this when running `yarn start:storybook`:



Let's write some components we can document. For this example, I built a `heading` and a `button` component.

I made the button in `src/components/button/index.js` and it looks like so:

```
import React from "react";
import styled, { css } from "styled-components";

const ButtonComponent = styled.button`
  padding: 8px 12px;
```

```
  border: none;
  background: #319795;
  color: white;

  ${(props) =>
    props.small &&
    css`
      padding: 4px 8px;
    `}
`;

const Button = ({ children, ...props }) => (
  <ButtonComponent {...props}>{children}</ButtonComponent>
);

export default Button;
```

Lastly the heading I wrote in `src/components/heading/index.js` like so:

```
import React from "react";
import styled from "styled-components";

const HeadingComponent = styled.h1`
  font-family: --apple-system, BlinkMacSystemFont, "Segoe
UI", Roboto, Oxygen,
    Ubuntu, Cantarell, "Open Sans", "Helvetica Neue",
sans-serif;
  color: #161515;
  padding: 0;
  margin: 0;
  color: #4f82b5;
`;
```

```
const Heading = ({ children, ...props }) => (
  <HeadingComponent {...props}>{children}
</HeadingComponent>
);

export default Heading;
```

We can now change our `index.js` to use these components to test them in the actual app:

```
import React from "react";
import ReactDOM from "react-dom";
import Heading from "./components/heading";
import Button from "./components/button/";
import styled from "styled-components";

const Main = styled.main`
  width: 1200px;
  max-width: 80%;
  margin: 50px auto;
`;

const App = () => {
  return (
    <Main>
      <Heading as="h2">Welcome to Storybook</Heading>
      <Button>I am a button</Button>
    </Main>
  );
};
```

```
const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

Our app works. That's not the most exciting thing ever because we still have no stories to show for it, and this is where the magic of Storybook starts. All we need to do for this is to create a file that ends with `stories.js` in one of the components folders, and we will already see something.

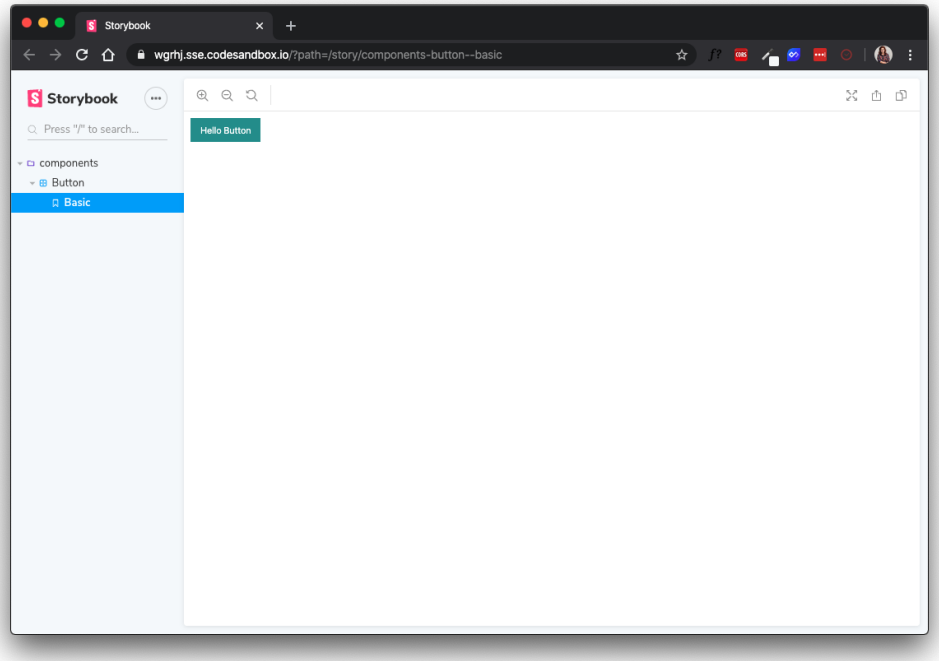
Let's start with the button by creating a file next to the index called `button.stories.js`. This is where we will create our `stories`, the first thing we need to do is import our button and tell Storybook what the title of our component will be in the Storybook UI and what is the component that we want to test. We can do it like so:

```
import React from "react";
import Button from "./";

export default {
  title: "components/Button",
  component: Button,
};
```

We still have no stories so let's add some. After this `export default` we can add:

```
export const basic = () => <Button>Hello Button</Button>;
```

And now we have a story, and our Storybook should look something like:



The idea is that now we can test our button in a completely clean environment. Let's add some more stories for our button to also test our small prop, and our `button.stories.js` should look like so:

```
import React from "react";
import Button from "./";

export default {
  title: "components/Button",
  component: Button,
};

export const basic = () => <Button>Hello Button</Button>;
export const emoji = () => (
  <Button>
    <span role="img" aria-label="so cool">
      😀 😎 👍 💯
    </span>
```

```
  </Button>
);
export const small = () => <Button small>Hello Small
Button</Button>;
```

We can follow the same idea for the heading and add a `heading.stories.js`:

```
import React from "react";
import Heading from "./";

export default {
  title: "components/Heading",
  component: Heading,
};

export const H1 = () => <Heading>Hello Heading</Heading>;
export const H2 = () => <Heading as="h2">Hello
Heading</Heading>;
export const H3 = () => <Heading as="h3">Hello
Heading</Heading>;
export const H4 = () => <Heading as="h4">Hello
Heading</Heading>;
export const H5 = () => <Heading as="h5">Hello
Heading</Heading>;
export const H6 = () => <Heading as="h6">Hello
Heading</Heading>;
```

This is pretty cool by itself, but the community behind Storybook is even more amazing, giving us some add-ons we can attach to Storybook to make our life easier.

One of those add-ons is called `@storybook/addon-storyshots`, which creates Jest snapshots for every story you create.

Let's add the add-on and let's start by installing it:

```
yarn add @storybook/addon-storyshots --dev
```

Once installed, we need to create a file that will match the test regex in Jest, so something that ends in `.test.js` is perfect. Next, we need to place it in the `src/` folder.

In this case, I made a `storyshots.test.js` file, which is where we need to configure the new add-ons we got like so:

```
import initStoryshots from "@storybook/addon-storyshots";

initStoryshots();
```

That's it, this will match the Jest regex so it will be run, and then the add-on looks for your stories and will make Jest snapshot tests around it.

One last thing before we leave Storybook behind is theming. The default theme for Storybook is light, and that's fine, but as a preference, I like all my themes to be dark so they can match my soul and turns out even that is entirely possible.

For that, we can create a `manager.js` file in the `.storybook.` folder, in there, we will use `@storybook/theming` to change our default theme:

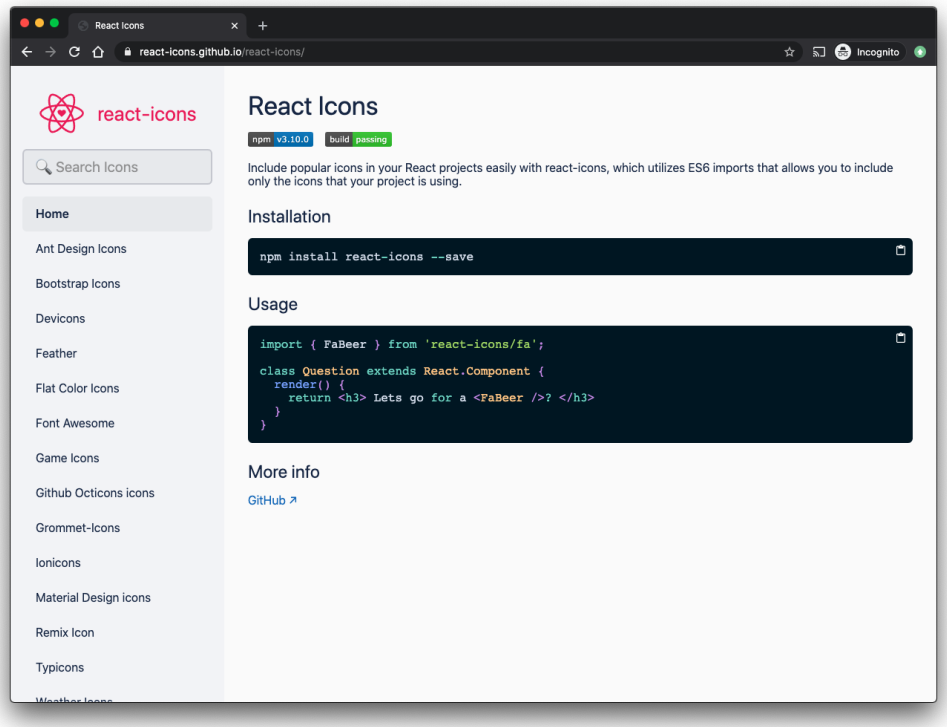
```
import { addons } from "@storybook/addons";
import { themes } from "@storybook/theming";
```

```
addons.setConfig({
  theme: themes.dark,
});
```

This sets a config for Storybook that tells it to use the default dark theme that comes with Storybook instead of the default light one. You can also make your own themes if you feel inclined to give it a go, there is some documentation for that in the storybook website.

Link to CodeSandbox

Icons



Chosen: React Icons

This one may seem like a very specific package that is not used in half of the projects you would create, but most applications and websites will require some sort of Icon package if you think about it.

The amazing thing about `react-icons` is that they are all pure SVG components instead of font icons; this makes them way more accessible and customizable than regular font icons.

If you want to read more about the issues with font icons, you can look at this article

Let's then install this magical package:

```
yarn add react-icons
```

We can see all the available icons in their website, and we can now search for React to import the React icon into our page.

Its name is `DiReact`, the `Di` prefix specifies the icon package it comes from and the folder from where we need to import it, which would look like this:

```
import { DiReact } from "react-icons/di";
```

It seems like a lot to memorize, but it's possible to get the hang of it thanks to the naming and the search box on their website, which is definitely a lifesaver.

To use this icon, we need to insert it into our page like so:

```
import React from "react";
import { DiReact } from "react-icons/di";
import "./styles.css";
```

```
export default function App() {
  return (
    <div className="App">
      <DiReact />
    </div>
  );
}
```

We now see a tiny SVG icon at the top, and there are many things we can customize in this SVG with its props. Let's start by making it bigger, and with a color that matches the React theme:

```
import React from "react";
import { DiReact } from "react-icons/di";
import "./styles.css";

export default function App() {
  return (
    <div className="App">
      <DiReact size={100} color="#61dafb" />
    </div>
  );
}
```

We now have a blue React logo that is `100x100`; this is where you can see this package's power.

Let's also animate it.

The first step to accomplish this is by adding a class to animate it using CSS.

```
import React from "react";
import { DiReact } from "react-icons/di";
import "./styles.css";

export default function App() {
  return (
    <div className="App">
      <DiReact size={100} color="#61dafb"
className="react-icon" />
    </div>
  );
}
```

And now, let's include the necessary CSS for the spinning animation:

```
@keyframes react-icon {
  100% {
    transform: rotate(900deg);
  }
}

.react-icon {
  animation: react-icon 2s infinite;
}
```

That is pretty impressive in terms of flexibility. Of course, it doesn't give you the ability to change everything, but it's still a pretty good amount of customization.

Let's say I also want to add the JS icon:

```
import React from "react";
import { DiReact, DiJsBadge } from "react-icons/di";
import "./styles.css";

export default function App() {
  return (
    <div className="App">
      <DiReact size={100} color="#61dafb"
className="react-icon" />
      <DiJsBadge color="#f7df1e" size={100} />
    </div>
  );
}
```

You may notice that both these icons have the same size, if we want all of our icons to match this size, we can pass a provider from `react-icons` to help us have some really good defaults:

```
import React from "react";
import ReactDOM from "react-dom";
import { IconContext } from "react-icons";

import App from "./App";

const rootElement = document.getElementById("root");
ReactDOM.render(
  <IconContext.Provider value={{ size: 100 }}>
    <App />
  </IconContext.Provider>,
  rootElement
);
```

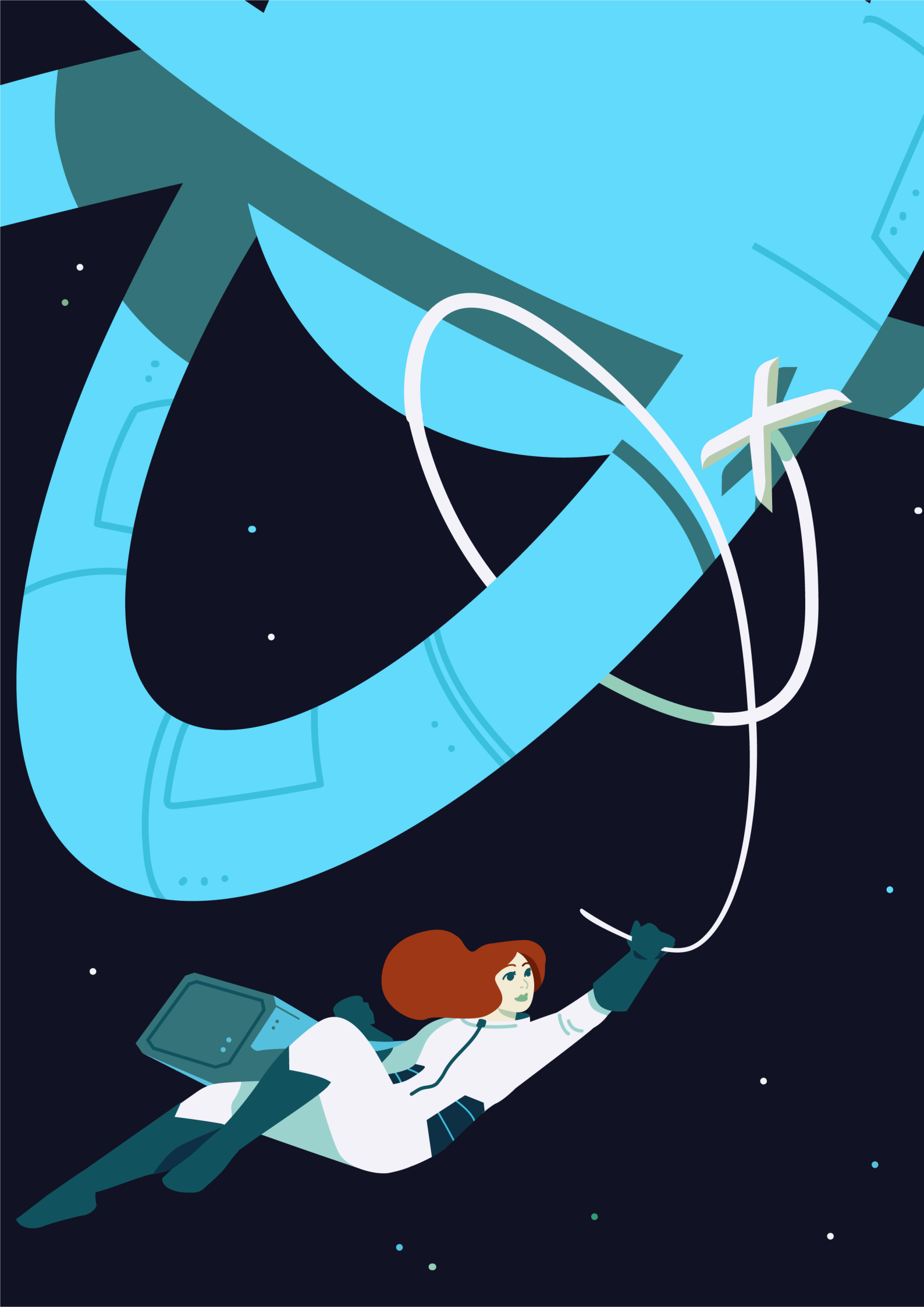
Here, we get the `IconContext` from `react-icons` and use its provider to specify that the default value of the size is 100. *Side note: if no unit is passed in the value, it will assume you mean px, just like in inline styles*

With this, we can clean up our app a bit more by removing the hardcoded 100 in both icons:

```
import React from "react";
import { DiReact, DiJsBadge } from "react-icons/di";
import "./styles.css";

export default function App() {
  return (
    <div className="App">
      <DiReact color="#61dafb" className="react-icon" />
      <DiJsBadge color="#f7df1e" />
    </div>
  );
}
```

This package is truly a lifesaver when it comes to icons, and it's definitely something that I use a LOT.

Link to CodeSandbox



The Hooks

Hooks are dope?! Right? RIGHT!?

Well, yes, but some of them can be quite confusing when starting with them or changing from class to function components. In this chapter, I will go through some of them.

useEffect

From my experience, `useEffect` is the most confusing of the bunch; In my opinion, this is mostly because it does everything imaginable. With just `useEffect`, you can replace all the lifecycle methods that exist in React.

Let's start with an example where we `console.log` whenever a prop changes with `useEffect`:

```
import React, { useEffect, useState } from "react";
import "./styles.css";

const Name = ({ name }) => {
  useEffect(() => {
    console.log("name changed");
  }, [name]);

  return <h1> Hello {name}</h1>;
};
```

```
const App = () => {
  const [name, setName] = useState("Jane Doe");
  return (
    <div className="App">
      <Name name={name} />
      <h2>Change the name</h2>
      <button onClick={() =>
setName("CodeSandbox")}>CodeSandbox</button>
      <button onClick={() => setName("John Doe")}>John
Doe</button>
    </div>
  );
};

export default App;
```

If you click any of the buttons, you can see that it does a `console.log` after the name changed, but it also does the `console.log` when it mounts, and this is where the confusion starts.

One very important thing about `useEffect` is that it will always render on mount unless told otherwise, because all props get changed on mount, and it's one of those situations where you look at it and think: "I guess!!!!".

Like in the previous hook, let's look at an example where the `useEffect` hook can be used in two different aspects, one on start and one when a specific state value changes.

We will be using the awesome Rick and Morty Api for these calls. Let's first get all the characters from Rick and Morty when the page loads:

```
import React, { useEffect, useState } from "react";

export default function App() {
  const [characters, setCharacters] = useState([]);
```

```
  const getCharacters = async () => {
    const data = await fetch(
      `https://rickandmortyapi.com/api/character/?page=1`
    );
    const characters = await data.json();
    return characters.results;
  };

  useEffect(() => {
    getCharacters().then((rsp) => setCharacters(rsp));
  }, []);

  return (
    <main className="App">
      <h1>Rick and Morty Characters</h1>
      <ul>
        {characters.map((character) => (
          <li key={character.id}>
            <img src={character.image} alt=
{character.name} />
            {character.name}
          </li>
        ))}
      </ul>
    </main>
  );
}
```

As you can see, we passed an empty array as the second parameter for the `useEffect` hook, which means that no matter how many times our page re-renders, this array will never change, so this effect will only run once, in the mounting of the component.

You can also pass a static value, and the idea will be the same. If you passed something like `["FANCY TEXT"]` as the dependency array, it would only run once since that text is static and not a prop or state value that can change. The empty array is just an easier way to write and make sense of it.

Now let's say I want to show some more info about a character when I click a button beneath them. We can do this by using our `useState` hook and setting the `id` of the character we want to show more info for.

```
import React, { useEffect, useState } from "react";
import "./styles.css";

export default function App() {
  const [characters, setCharacters] = useState([]);
  const [openId, setOpenId] = useState(null);
  const [info, setInfo] = useState(null);

  const getCharacters = async () => {
    const data = await fetch(
      `https://rickandmortyapi.com/api/character/?page=1`
    );
    const characters = await data.json();
    return characters.results;
  };

  const getCharacter = async (id) => {
    if (id) {
      const data = await fetch(
        `https://rickandmortyapi.com/api/character/${id}`
      );
      const character = await data.json();
      return character;
```

```
    }
  };

  useEffect(() => {
    getCharacters().then((rsp) => setCharacters(rsp));
  }, []);

  useEffect(() => {
    getCharacter(openId).then((rsp) => setInfo(rsp));
  }, [openId]);

  return (
    <main className="App">
      <h1>Rick and Morty Characters</h1>
      <ul>
        {characters.map((character) => (
          <li key={character.id}>
            <img src={character.image} alt=
{character.name} />
            {character.name}
            <button onClick={() =>
setOpenId(character.id)}>More Info</button>
          </li>
        ))}
      </ul>
    </main>
  );
}
```

Here, we add another `useEffect`, and this will also trigger when the state of `openId` changes, so it will run both in the mounting of the App and then when the `openId` changes.

In this effect, we set the JSON we get from their server as the info, so there are two moving parts:

- The `openId` - This is the `id` of the character we want to see more info.
- `info` - This is the info from the character whose `id` is currently set

> *This could be done in different ways, but I wanted to demonstrate how an effect can be triggered in different ways.*

We can now check if our `openId` is equal to the character we are rendering, and use the value of its info in our render function like this:

```
<main className="App">
  <h1>Rick and Morty Characters</h1>
  <ul>
    {characters.map((character) => (
      <li key={character.id}>
        <img src={character.image} alt={character.name} />
        {character.name}
        {openId === character.id && info ? (
          <div>
            <div>Status: {info.status}</div>
            <div>Species: {info.species}</div>
          </div>
        ) : null}
        <button onClick={() =>
setOpenId(character.id)}>More Info</button>
      </li>
    ))}
  </ul>
</main>
```

And it works! We now trigger the fetching of new character info using the `useEffect` hook.

This hook is very powerful, but it can also be slightly confusing. These are the things that might trick you out:

- Nothing passed as a second parameter means the `useEffect` hook will run on every re-render of the component, and your computer goes into hyperdrive.
- You cannot return `null` from an event, as it expects a runnable function to be returned.
- To clean up the effect, you can pass a function that will be called after the effect runs, like so:

```
useEffect(() => {
  window.addEventListener("mousemove", callback);
  return () => window.removeEventListener("mousemove",
callback);
});
```

After the event ran, we would clean after it by removing our event listener so that there are no weird performance issues.

- Passing a prop still means the effect will run on mount.
- To run something on unmount, you can use the example above but only with the return and an empty array as the second parameter like this:

```
useEffect(() => {
  return () => onUnmount();
}, []);
```

- Effect callbacks are synchronous to prevent race conditions, so you can not make it async but can call async functions from it by appending a `then` to it as we saw in the example. I know, it is pretty weird.

I think that is all, and I wish you the best of luck with this effect!

Link to CodeSandbox

useLayoutEffect

This effect is not a very common hook to use, but it can be very confusing when mentioned alongside `useEffect`.

I wanna start by saying that 99% of times you want to use `useEffect` but for cases like DOM manipulation you can use `useLayoutEffect` since the main the difference between these is that `useEffect` will run after all is rendered AND painted on the page and `useLayoutEffect` runs before the paint takes place, basically milliseconds before the user actually sees the changes.

It's handy in particular cases like when handling DOM, but in 99% of times, you will want to use `useEffect`.

useState

I am not going to go through a lot of `useState` because, well, we all have used it.

I just want to mention one small thing, and since I am boring person, I'm gonna mention it with a small counter example.

A lot of times I have used the state value inside of the `setter` function, like this:

```
export default function App() {
  const [counter, setCounter] = useState(0);

  const up = () => {
    setCounter(counter + 1);
  };

  return (
    <main className="App">
      <button onClick={up}>Up</button>
      <button onClick={() => setCounter(0)}>Reset</button>
      <p>{counter}</p>
    </main>
  );
}
```

This is usually fine, but let's add a weird `setTimeout` to this `up` function:

```
const up = () => {
  window.setTimeout(() => {
    setCounter(counter + 1);
  });
  setCounter(counter + 1);
};
```

When I click the button it should always increase the value of `counter` by two because I'm running it twice, right?

Well, no. When getting the state value like this, we can run into cases where the value we are accessing is actually not the latest one in the state.

To combat this, when dealing with updating the state depending on its current value, we should access it like so:

```
setCounter((counter) => counter + 1);
```

What you can see here is that the `setState` function can also take a function as an argument, and when this happens, the first argument of the passed function will always have the updated version of the state. This small adjustment can prevent some nasty bugs.

Let's then fix our code:

```
export default function App() {
  const [counter, setCounter] = useState(0);

  const up = () => {
    window.setTimeout(() => {
      setCounter((counter) => counter + 1);
    });
    setCounter((counter) => counter + 1);
  };

  return (
    <main className="App">
      <button onClick={up}>Up</button>
      <button onClick={() => setCounter(0)}>Reset</button>
      <p>{counter}</p>
    </main>
```

```
  );
}
```

If you now click the "Up" button, you can see it will increase the state value by two and we just fixed a bug.

Link to CodeSandbox

useContext

Context is something that the React community has been using for ages, even though it wasn't stable, as we all love living on the edge. Context allows you to store some data at the top of your application and access it anywhere.

Let's build an example of the ten worst airports in Europe and save this data in context:

Let's start by instantiating the context in a file called `AirportContext.js`

```
import React from "react";

const AirportContext = React.createContext();

export default AirportContext;
```

In here, we instantiate the state without a value as that will be passed via the Provider.

The provider is a part of the `AirportContext`, you can think of the `AirportContext` as `const { Provider, Consumer } = AirportContext` since it exports two properties out of it.

This part of the Context is used to store the value that will be passed down, and we can instantiate it like so:

```
import React from "react";
import ReactDOM from "react-dom";
import airports from "./data";
import App from "./App";
import AirportContext from "./AirportContext";

const rootElement = document.getElementById("root");
ReactDOM.render(
  <AirportContext.Provider value={airports}>
    <App />
  </AirportContext.Provider>,
  rootElement
);
```

We start by getting the data we have of the worst airports, then we pick up the `AirportContext` and use its Provider to wrap our whole application.

By doing this, we now have access to the airports anywhere in our app.

This is where we get to use the new `useContext` hook, and this hook will get the values from `AirportContext`, which looks like this:

```
import React from "react";
import AirportContext from "./AirportContext";

export default function App() {
  const airports = React.useContext(AirportContext);
  console.log(airports);
```

```
  return (
    <div className="App">
      <h1>Worst Airports in Europe</h1>
    </div>
  );
}
```

By taking a look in the console, you can see the values are exactly the ones we passed at the top of the application, which we can now show to the user:

```
import React from "react";
import AirportContext from "./AirportContext";

export default function App() {
  const airports = React.useContext(AirportContext);

  return (
    <div className="App">
      <h1>Worst Airports in Europe</h1>
      <ul>
        {airports.map((airport) => (
          <li>{airport.name}</li>
        ))}
      </ul>
    </div>
  );
}
```

This is where it gets interesting. If we can show data, we can also manipulate it. The value prop in the `Provider` of the `AirportContext` can also take functions that can manipulate the airport object.

Let's add a function that will allow us to remove an airport.

Instead of exporting everything static in `AirportContext.js`, we can make a component that can have state set. This component will make our context act like an elementary store that we can read and update.

```
import React, { useState, createContext } from "react";
import airportList from "./data";

export const AirportContext = createContext(null);

export default ({ children }) => {
  const [airports, setAirports] = useState(airportList);

  const store = {
    airports: airports,
    removeAirport: (name) =>
      setAirports((a) => a.filter((airport) =>
airport.name !== name)),
  };

  return (
    <AirportContext.Provider value={store}>{children}
</AirportContext.Provider>
  );
};
```

Here, we created a stateful component that returns our provider with a value set so that we can wrap it around our app.

The important thing is that instead of passing a static array to the `value` prop we now pass state that is returned from `useState`, and that means that every

time we update the state in this component, all of our components down the tree that use this context will update as well, creating a straightforward state management solution.

We can now clean up our `index.js` since no data is supposed to be passed to it anymore, and we have all of our state tucked away in one file:

```
import React from "react";
import ReactDOM from "react-dom";
import App from "./App";
import AirportContext from "./AirportContext";

const rootElement = document.getElementById("root");
ReactDOM.render(
  <AirportContext>
    <App />
  </AirportContext>,
  rootElement
);
```

The last step is updating a couple of things in our `App.js` to use the new `deleteAirport` function:

```
import React from "react";
import "./styles.css";
import { AirportContext } from "./AirportContext";

export default function App() {
  const { airports, removeAirport } =
React.useContext(AirportContext);
```

```
  return (
    <div className="App">
      <h1>Worst Airports in Europe</h1>
      <ul>
        {airports.map((airport) => (
          <li key={airport.name}>
            {airport.name}
            <button onClick={() =>
removeAirport(airport.name)}>x</button>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

There are two major differences. First, we are now deconstructing what we get from the context as it's an object containing the mutations to the state and the state itself. Then, we also deconstruct `AirportContext` as an import, since we now have two exports in that file instead of a default one.

With `useContext`, we can make plenty of small things like this, but be careful when doing it since you may fall into the trap of having way too much state attached to a simple `useContext`, and that can get out of hand pretty fast.

Link to CodeSandbox

useReducer

A component state can get complicated, resulting in too many actions and a lot of `if` statements. This is where `useReducer` comes in. This hook allows you

to manage better the mutations you make to the state by offering a consistent methodology to dispatch actions that will update the original state and return its updated version.

We will pick up our previous airport example with some changes, and I added id's and a `visited` toggle, you can see the new data in this GitHub gist.

Let's open up our `AirportContext.js` that now looks like this:

```
import React, { useState, createContext } from "react";
import airportList from "./data";

export const AirportContext = createContext(null);

export default ({ children }) => {
  const [airports, setAirports] = useState(airportList);

  const store = {
    airports: airports,
    removeAirport: (name) =>
      setAirports((a) => a.filter((airport) =>
airport.name !== name)),
  };

  return (
    <AirportContext.Provider value={store}>{children}
</AirportContext.Provider>
  );
};
```

We need to change this from `useState` to `useReducer`. The first thing we need to know is that `useReducer` takes two parameters:

- *The reducer function* - This is where you will update the state. This function usually `switches` depending on the type of action you send (don't worry, code will explain it better).
- *The initial state* - This is your initial state, as it was originally passed to the `useState` function in the code above

Let's then clean this code and start using this new hook:

```
import React, { createContext, useReducer } from "react";
import airportList from "./data";

export const AirportContext = createContext(null);

function reducer(state, action) {
  switch (action.type) {
    case "removeVisited":
      return state;
    case "addVisited":
      return state;
    default:
      return state;
  }
}

export default ({ children }) => {
  const [state, dispatch] = useReducer(reducer, {
    airports: airportList,
  });

  return (
    <AirportContext.Provider value={{ state, dispatch }}>
```

```
      {children}
    </AirportContext.Provider>
  );
};
```

We can see that we are now importing the `useReducer` hook and using it by passing both our `reducer` function, which we defined at the top and our default state with the airports from our data file.

Looking at our reducer function, we can see that it always gets two parameters: one is `state`, which holds the current state at the time the function is called, and the second one is `action`.

An action can be called from anywhere within the app using the `dispatch` function, and this is usually called with an object that contains `type` and `value` properties like this:

```
dispatch({ type: "removeVisited", value: airport.id });
```

You can pass anything you want to the `dispatch` function, and this is just the convention that is meant to make it easier to jump between projects holding the same object structure and action pattern.

In this case, we can read the action `type` and then return a different `state` depending on the action type. We can toggle on the `visited` value for a specific airport by mapping over the `airports` object in our state and setting the `visited` to `false` if the airport `id` matches the `id` we passed in the action:

```
case "removeVisited":
  return {
```

```
    airports: state.airports.map(airport => {
      if (airport.id === action.value) {
        return {
          ...airport,
          visited: false
        };
      }

      return airport;
    })
  };
```

We can now update the reducer for the two different types of actions we have, `removeVisited` and `addVisited`:

```
function reducer(state, action) {
  switch (action.type) {
    case "removeVisited":
      return {
        airports: state.airports.map((airport) => {
          if (airport.id === action.value) {
            return {
              ...airport,
              visited: false,
            };
          }

          return airport;
        }),
      };
    case "addVisited":
      return {
```

```
        airports: state.airports.map((airport) => {
          if (airport.id === action.value) {
            return {
              ...airport,
              visited: true,
            };
          }

          return airport;
        }),
      };
    default:
      return state;
  }
}
```

In our component, we call our `dispatch` function, which will trigger our reducer function and update the state:

```
import { AirportContext } from "./AirportContext";

export default function App() {
  const { state, dispatch } =
React.useContext(AirportContext);

  return (
    <div className="App">
      <h1>Worst Airports in Europe</h1>
      <ul>
        {state.airports.map((airport) => (
          <li key={airport.name}>
            {airport.name}
```

```
            {airport.visited ? (
              <button
                onClick={() =>
                  dispatch({ type: "removeVisited", value:
airport.id })
                }
              >
                Remove Visited
              </button>
            ) : (
              <button
                onClick={() =>
                  dispatch({ type: "addVisited", value:
airport.id })
                }
              >
                Mark as Visited
              </button>
            )}
            <img width="300" src={airport.photo} alt=
{airport.name} />
          </li>
        ))}
      </ul>
    </div>
  );
}
```

If you now click any of the buttons, you can see that the `visited` value is toggled to either `true` or `false`.

Now that we handled two different action types being passed through, we can clean up the code a bit and simplify using a single `toggleVisited` action type::

```
function reducer(state, action) {
  switch (action.type) {
    case "toggleVisited":
      return {
        airports: state.airports.map((airport) => {
          if (airport.id === action.value) {
            return {
              ...airport,
              visited: !airport.visited,
            };
          }

          return airport;
        }),
      };
    default:
      return state;
  }
}
```

Doing this makes it much cleaner and easier to add on to and trust me in the future this will not be the only action you will have.

The idea of `useReducer`, in my opinion, is to create a more scalable state for your components, it introduces a pattern and a more scalable way to handle state changes.

Link to CodeSandbox

useRef

Refs are something we already had in the previous versions of React, and the most common use case for them is to be able to access a DOM element and perform imperative actions on it, *jQuery style*.

This hook allows us to get values from a specific HTML element like its width, or even trigger actions in the HTML element. By using this, we can get access to the full DOM API as an escape hatch if we need to change the DOM without React intervening.

Let's import it and attach it somewhere so we can see how it works:

```
import React, { useRef, useEffect } from "react";

export default function App() {
  const button = useRef(null);

  useEffect(() => {
    console.log(button.current);
  }, []);

  return <button ref={button}>A button</button>;
}
```

You may be wondering why I `console.log` the `current` out of the button. That is because of the way React stores this. It always stores the last value of the `ref` in the current key of the `ref` object.

Your console should now have the HTML element as you would by using:

```
$("button");
```

And like you could in jQuery, we can click the button like so:

```
import React, { useRef, useEffect } from "react";

export default function App() {
  const button = useRef(null);

  useEffect(() => {
    button.current.click();
  }, []);

  return (
    <button ref={button} onClick={() => alert("I have been
clicked")}>
      A button
    </button>
  );
}
```

Like in the good ol' days, you will see an alert as soon the page is loaded.

In my last example, the button was clicked programmatically using the `ref` we placed on it.

We can take this even further by having two buttons and using `refs`. We can click on one button, which triggers a click on the other button.

Let's see this in code:

```
import React, { useRef, useState } from "react";
import "./styles.css";

export default function App() {
  const [count, setCount] = useState(0);
  const button = useRef(null);

  const fakeClick = () => {
    button.current.click();
  };

  return (
    <>
      <button onClick={fakeClick}>
        This button triggers a click somewhere else
      </button>
      Count: <span>{count}</span>
      <button ref={button} onClick={() => setCount(count +
1)}>
        +
      </button>
    </>
  );
}
```

If you click on the first button, you can see that it acts as if the increment button was clicked, and it will increase the counter.

You can also do some handy things with useRef like redirect user focus to where you want it to be, or get the dimensions of an element.

The idea of React is that it will handle DOM updates and manipulation for you, so please consider that when using `useRef` and use it with caution.

Link to CodeSandbox

As a final for hooks, let's also look at an interesting part about `useRef`: the fact that it's a mutable value that can hold anything in it's `current` property.

This idea may seem a bit confusing so let's test some code:

```
export default function App() {
  const count = useRef(0);
  const [countState, setCountState] = useState(0);

  useEffect(() => {
    count.current = 1;
    console.log("Ref " + count.current);
  }, []);

  useEffect(() => {
    setCountState(1);
    console.log("State " + countState);
  }, []);

  return <h1>Hello</h1>;
}
```

You can see we have two effects, one sets the value of `count.current` by accessing and mutating its value, and the next one uses `useState` to change that same value.

Both of these do a console.log after the value has been changed.

If we look at the console, we can see the following:

```
Ref 1
State 0
```

Interesting ah? :ponders-in-british:

The reason for that is that when you set setState in a react component, you trigger that component to run again with the new values it has, in this case, with the new state of `1`. So the `console.log` we have there doesn't actually get the new value since the effect doesn't run when that value changes. In the case of a `ref`, it's a mutable value, so it means that when you run, the value will be automagically updated with no re-render necessary, and that's why we can see `Ref 1` in the console.

You may be wondering what happens when we pass the `stateCount` as a dependency like so:

```
export default function App() {
  const count = useRef(0);
  const [countState, setCountState] = useState(0);

  useEffect(() => {
    count.current = 1;
    console.log("Ref " + count.current);
  }, []);

  useEffect(() => {
    setCountState(1);
    console.log("State " + countState);
  }, [countState]);
```

```
  return <>Hello</>;
}
```

In this case, we can see that what happens is that the component renders the same in the first run, but then since we have the dependency of the `countState`, the component re-renders, and only now can we see the new value.

I hope this makes sense and helps you understand a bit of the magic of `useRef`.

Link to CodeSandbox

useMemo

We are almost at the end of the hooks. We can do this.

We are now approaching the `memoization` hooks, I know that it's a very long word, it also took me some digging to figure it out because I had never heard this word before even though I had used the technique.

I think the best way to try and understand what `memoization` is to make a small function to create the same effect.

We have this function:

```
const multiplyNumberBy10 = (number) => {
  const value = number * 10;
  console.log(value);
};
```

```
[...Array(10)].map(() =>
multiplyNumberBy10(Math.round(Math.random() * 5)));
```

We create an array with ten elements, then for each, we map and call a function that multiplies a random number between 0 and 5 by 10.

If we take a look at our console, we will just see the results, nothing unusual in them, but what is interesting is that some of these are repeated, which means the function has already done this work.

We can save that in the function, so it doesn't have to repeat the same work twice. For that, we can create an object that saves the number passed as the key and the number it creates as the value, like so:

```
const cache = {};
const multiplyNumberBy10 = (number) => {
  if (cache[number]) return cache[number];
  const value = number * 5;
  console.log(value);
  cache[number] = value;
};
```

Sandbox Here

If we re-run this, we can see we don't see as many logs since the function has saved the values it had in the cache and then used those when it encountered the same parameters.

This is a simple example of `memoization` and also how to create a very simple cache for any function that will run a lot of times with repeated values.

Now that we know what `memoization` is, let's take a look at `useMemo`, its function is to return a memoized value.

As the second argument, you can pass to an array of dependencies like in `useEffect`, and only when any of those dependencies change will the value be calculated again.

To see this in action, let's create a small example that has two separate states and does a very intensive function as well:

```
import React, { useState } from "react";

function fibonacci(x) {
  console.log("here");
  if (x <= 0) return 0;
  if (x === 1) return 1;
  return fibonacci(x - 1) + fibonacci(x - 2);
}

const App = () => {
  const [number, setNumber] = useState(10);
  const [isGreen, setIsGreen] = useState(true);
  const fib = fibonacci(number);

  return (
    <>
      <button
        onClick={() => setIsGreen(!isGreen)}
        style={{ background: isGreen ? "#01FF70" :
"#B10DC9" }}
      >
        useMemo Example
```

```
      </button>
      <h2>
        Fibonacci of {number} is {fib}
      </h2>
      <button onClick={() => setNumber(number + 1)}>+
</button>
    </>
  );
};

export default App;
```

The `fibonacci` function is super badly implemented, and when we run it a lot of times, we can start to see our tab slowly getting stuck.

The main problem here is that if we click the button that sets the color, we can see the function is still called, and that's definitely not what we want.

We want to give it a cache so that it knows what numbers have been calculated and doesn't need to do that calculation again. That's precisely what `useMemo` is here for.

Let's wrap our function in a `useMemo`:

```
const fib = useMemo(() => fibonacci(number), [number]);
```

Awesome! Our function is only run again when the number changes and doesn't care about anything else changing in your app.

I hope this gets you to see how `memoization` can help when it comes to performance issues and bottlenecks.

Link to CodeSandbox

useCallback

Now that we know what `memoization` is, we can apply the same idea to `useCallback`, but instead of memoizing a value as it does in `useMemo`, it memoizes a function.

Let's create an example where we have a function that gets a number in the state and adds one to it:

```
import React, { useState, useEffect } from "react";

const App = () => {
  const [num, setNum] = useState(0);
  const [plusOne, setPlusOne] = useState(0);

  const add1 = (x) => {
    console.log("here", x);
    return x + 1;
  };

  useEffect(() => {
    setPlusOne(add1(num));
  }, [num, add1]);
  return (
    <>
      <h2>{num}</h2>
      <h3>{plusOne}</h3>
      <button onClick={() => setNum(num + 1)}>+</button>
    </>
```

```
  );
};

export default App;
```

If you now run this, you can see that the `add1` function runs twice every time the `num` changes, instead of once as we would expect.

The issue is that in JavaScript, no two functions are equal to each other, so our change gets triggered twice because the first time we call it, the function is re-created and triggering the effect to run again.

`useCallback` fixes this by saying the same thing as `useMemo`: give me the previously created function unless any of the dependencies I send you change.

If we wrap the `add1` function in `useCallback` like so:

```
const add1 = useCallback((x) => {
  console.log("here", x);
  return x + 1;
}, []);
```

We can see that now it's only rendering once as we always intended.

You can think of `useCallback` as `useMemo` but for functions.

Link to CodeSandbox

Make your own hook

We looked at a LOT of hooks already, and as cool as they are, the best thing about them is that you can actually make your own hook since they are just functions that take advantage of the React API.

Let's create a hook that will allow us to easily add `localStorage` saving to any app.

The first step would be to think of the API of this hook. That's just a fancy way of saying: "How I want to use it".

My idea is something like:

```
const [movies, setMovies] = useLocalStorage("movies");
//     values  settingNewValues
localStorageKey
```

> *Don't forget that hooks must start with `use`.*

The general concept that we want to implement is this:

You pass the `localStorage` key where you want your data to be stored to the function (hook), and it will return two things: all the values in that key already parsed and a function to set the new value for that `localStorage` key. Works a bit like `useState` but for the `localStorage`.

Let's then build it.

First, we need a file called `useLocalStorage.js`, there we can create the barebones of the function(hook):

```
import { useState, useEffect } from "react";

function useLocalStorage(key) {}

export default useLocalStorage;
```

Thinking in steps, the first thing we need to do here is to create an internal state in our hook, there we'll store the `localStorage` values that the user adds or updates with the hook, and we return those values back.

```
import { useState, useEffect } from "react";

function useLocalStorage(key) {
  const [storedValue, setStoredValue] = useState("");

  useEffect(() => {
    const item = window.localStorage.getItem(key);
    if (item !== null) {
      setStoredValue(JSON.parse(item));
    }
  }, [key]);

  return [storedValue];
}

export default useLocalStorage;
```

What we are doing here is calling an effect that will run when `useLocalStorage()` is first called and if the key changes. When `useEffect()` runs, we look for the `localStorage` value that corresponds to

the key that's being passed in `useLocalStorage()`. If that value is found, we save it to our internal state, and we return it.

> *The good thing about returning everything in an array format is that in the hook consumption side, the developer can call this `storedValue` anything they want.*

We got somewhere, we now get the `localStorage` value and save it, but we also need to update `localStorage`. For that, we can create a function called `updateLocalStorage` where we get a value as an argument and update the value for that `localStorage` key.

```
import { useState, useEffect } from "react";

function useLocalStorage(key) {
  const [storedValue, setStoredValue] = useState("");

  useEffect(() => {
    const item = window.localStorage.getItem(key);
    if (item !== null) {
      setStoredValue(JSON.parse(item));
    }
  }, [key]);

  const updateLocalStorage = (value) => {
    setStoredValue(value);
    window.localStorage.setItem(key,
JSON.stringify(value));
  };

  return [storedValue, updateLocalStorage];
}
```

```
export default useLocalStorage;
```

First, we need to keep our internal state up to speed on what we are changing in `localStorage`.

Then it's time to update our `localStorage` as well, for that we use the key, and we `stringify` the value, since `localStorage` only allows strings to be saved.

That's it! We can now test it.

For that I made a small movie app where you can add a list of movies you want to see:

```
import React, { useState } from "react";
import useLocalStorage from "./useLocalStorage";
import "./styles.css";

export default function App() {
  const [movies, setMovies] = useLocalStorage("movies");
  const [newMovie, setNewMovie] = useState("");

  const addMovie = (e) => {
    e.preventDefault();
    setMovies((movies || []).concat(newMovie));
    setNewMovie("");
  };

  return (
    <main>
      <h1>Movies to see</h1>
```

```
      <form onSubmit={addMovie}>
        <input
          placeholder="add a movie"
          onChange={(e) => setNewMovie(e.target.value)}
          value={newMovie}
        />
      </form>
      <ul>
        {movies ? (
          movies.map((movie) => <li key={movie}>{movie}
</li>)
        ) : (
          <small>There has to be something you want to
see</small>
        )}
      </ul>
    </main>
  );
}
```

You may notice a small issue here: You can't clear the list of movies after adding one or more, having people clear `localStorage` to reset something is not the best user experience, so let's add a function to clear `localStorage` in our hook:

```
import { useState, useEffect } from "react";

function useLocalStorage(key) {
  const [storedValue, setStoredValue] = useState("");

  // same as before
```

```
  const clearStorage = () => {
    setStoredValue("");
    window.localStorage.clear(key);
  };

  return [storedValue, updateLocalStorage, clearStorage];
}

export default useLocalStorage;
```

This part mostly shows you that you can return any number of values you want from a hook, and you are not constrained by only returning two like in `useState`.

When it comes to creating hooks and using them, the sky is the limit. You are welcomed to experiment and create the best API for your use case.

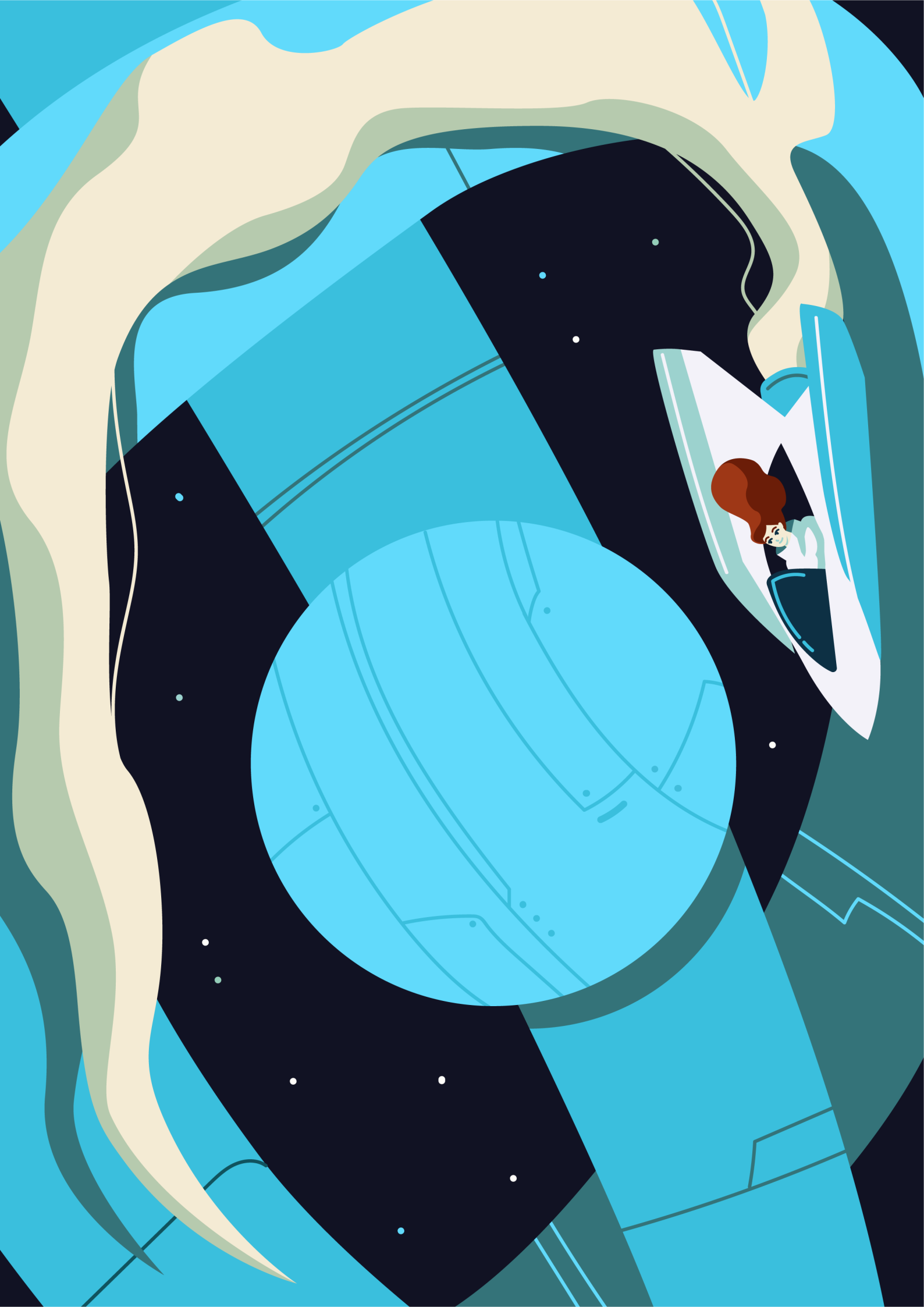
Let's now use the third function by adding a button to clear all the movies we have on the list:

```
import React, { useState } from "react";
import useLocalStorage from "./useLocalStorage";
import "./styles.css";

export default function App() {
  const [movies, setMovies, clearMovies] =
useLocalStorage("movies");
  const [newMovie, setNewMovie] = useState("");

  const addMovie = (e) => {
    e.preventDefault();
    setMovies((movies || []).concat(newMovie));
```

```
    setNewMovie("");
  };

  return (
    <main>
      <h1>Movies to see</h1>
      <form onSubmit={addMovie}>
        <input
          placeholder="add a movie"
          onChange={(e) => setNewMovie(e.target.value)}
          value={newMovie}
        />
      </form>
      <ul>
        {movies ? (
          movies.map((movie) => <li key={movie}>{movie}
</li>)
        ) : (
          <small>There has to be something you want to
see</small>
        )}
      </ul>

      <button onClick={clearMovies}>Remove all
Movies</button>
    </main>
  );
}
```

I hope this helped to make the idea of building your own hooks less scary, and also helped you see that in the end a custom hook is just a function that has access to React methods like `useState`, and gives you immense flexibility to create the developer experience you want to have.

Link to CodeSandbox



Performance

So your app is done, but it's kind of slow. It may also be really fast, but maybe you want to know a bit about performance in React.

In this small section, I am going to go through some of the things I have encountered that have helped me solve some performance issue.

Add keys to your list elements

This one may seem pretty straightforward and it is probably documented in every React document you have seen, but it's amazing how many times I forget this and then end up with gigantic performance losses.

This is because React keeps track of what elements were changed, removed or added with those keys, so if you don't add them and you have a dynamic list, React will feel lost and you will experience some weird stuff and laggy behavior.

Make small components

I said we were going to start easy. Let's imagine a component that has some state that changes quite often: what happens when you don't divide this component is that when the state changes, ALL your component gets updated, and that's usually not what you want.

React.memo

This is a super interesting high order function that exists in React, exactly for performance reasons. What it does is that it prevents an element from re-rendering if its props have not changed.

Let's take a small example:

```
import React, { useState, useEffect } from "react";

const ChildComponent = () => {
  console.log("rendered");
  return <p>HEEEY</p>;
};

export default function App() {
  const [quote, setQuote] = useState("");

  const getAQuote = async () => {
    const { quote } = await
fetch("https://api.kanye.rest").then((rsp) =>
      rsp.json()
    );

    setQuote(quote);
  };

  useEffect(() => {
    getAQuote();
  }, []);

  return (
    <main>
      <p>{quote}</p>
```

```
      <button onClick={getAQuote}>Get a new quote</button>
      <ChildComponent />
    </main>
  );
}
```

What happens here, is that every time I click the button to render the new quote, the `ChildComponent` also gets re-rendered, even though nothing has changed for it, no new props, no new state, it simply re-renders because its parent had to re-render.

One way we can fix this is to wrap our `ChildComponent` in `React.memo` like so:

```
const ChildComponent = React.memo(() => {
  console.log("rendered");
  return <p>HEEEY</p>;
});
```

By doing this we are telling React to not re-render this component unless its props change. By looking at the console, we can see this component was only rendered on mount and not when the parent got updated.

This is perfect for components that are completely presentational and just get props and show some UI.

Link to CodeSandbox

Avoid mounting and unmounting components

This comes a lot in play when we use animations or things that are shown only on click.

There are always two ways to approaching a situation like this, one is to mount the component like so:

```
import React, { useState } from "react";

const MountingComponent = () => {
  const [show, setShow] = useState(false);
  return (
    <main>
      <button onClick={() => setShow((show) =>
!show)}>Show the text</button>
      {show ? <p>I am the text</p> : null}
    </main>
  );
};
```

Or the CSS way:

```
const CSSWay = () => {
  const [show, setShow] = useState(false);
  return (
    <main>
      <button onClick={() => setShow((show) =>
!show)}>Show the text</button>
      <p style={{ opacity: show ? "1" : "0" }}>I am the
text</p>
    </main>
```

```
  );
};
```

If you run these two ways you can see that the UX the users gets is the same, but when it comes to performance, it's better for the browser and React to do this the CSS way as it doesn't need to create the component again and calculate everything twice.

Link to CodeSandbox

Virtualize long lists

Let's say you are rendering an enormous list of Simpsons characters with photos, if you just place everything in the DOM and try to scroll, you will have a bad time.

This is where virtualization plays a big role. What it does is that it only renders the part of the list that is visible at the time, and changes what is rendered as you scroll and another part becomes visible, it will give you an enormous performance boost when it comes to rendering big lists.

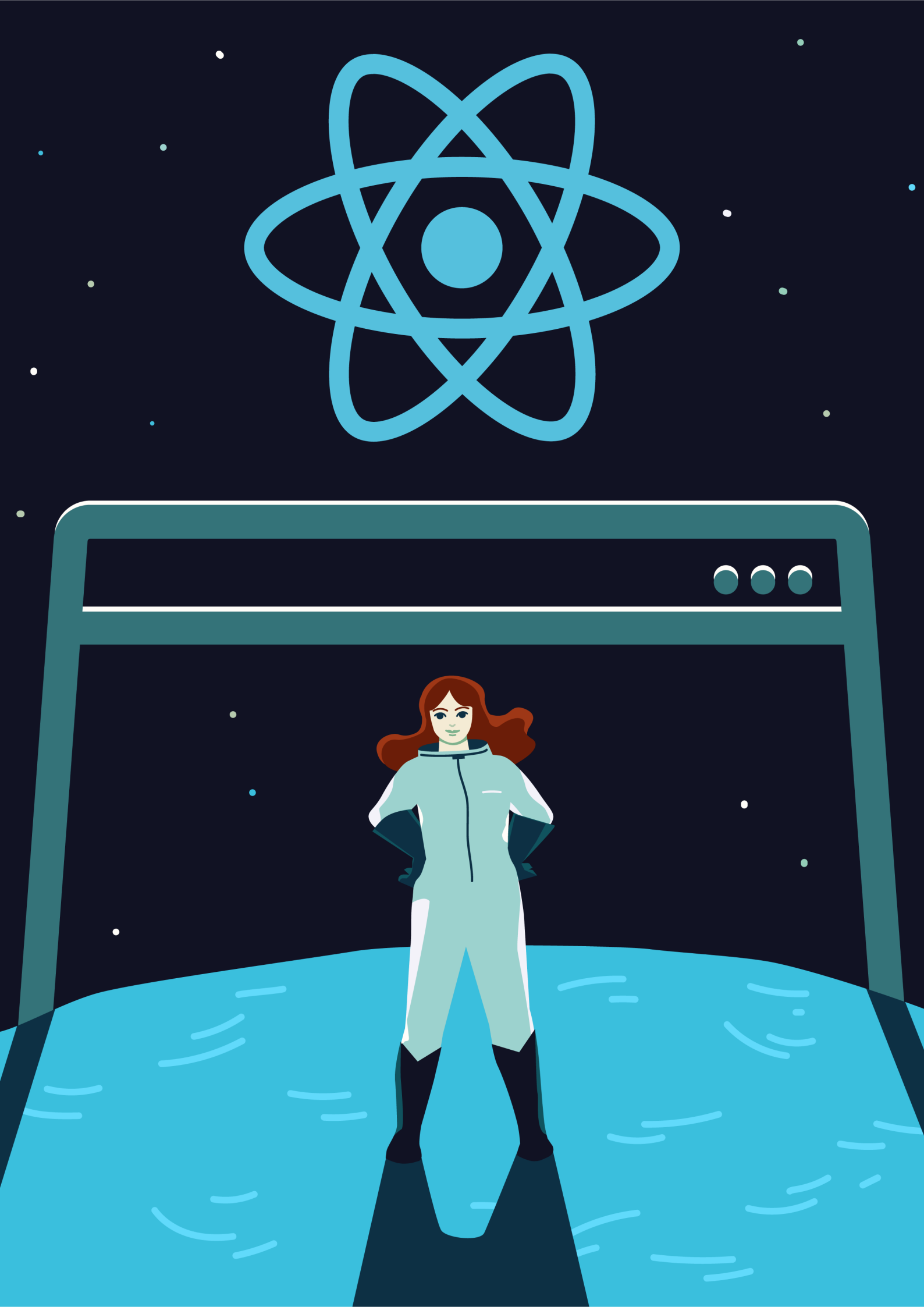
Two libraries that do this are react-window and react-virtualized. These libraries are not easy and changing their layout is pretty hard as well as they will place your items using position absolute but sometimes they truly need to be used.

Take advantage of useMemo and useCallback

These two functions are used for memoization so they are also very useful when it comes to helping your performance, we went over them at the top so imagining you have a function that may take the same values, its always a good idea to wrap in it a memo so you can basically have a free cache.

When it comes to `useCallback`, it works in the same way but for callback functions you may have in your files. Just these two can take you a long way.

I hope this small chapter can help you when it comes to fixing some performance issues.



Deployments

Cool, so you built yourself a pretty cool app, and you learned a bunch of things. The issue now is how to show it to the world.

Deployments have gotten much easier in the last couple of years, before Vercel and Netlify, I honestly didn't have anything on the internet, stuff like AWS always seemed pretty daunting to me, it still does to this day.

So let's take a look at how you could deploy the three tools we talked about (CRA, Gatsby, and Next) on both Netlify and Vercel.

CRA

Netlify

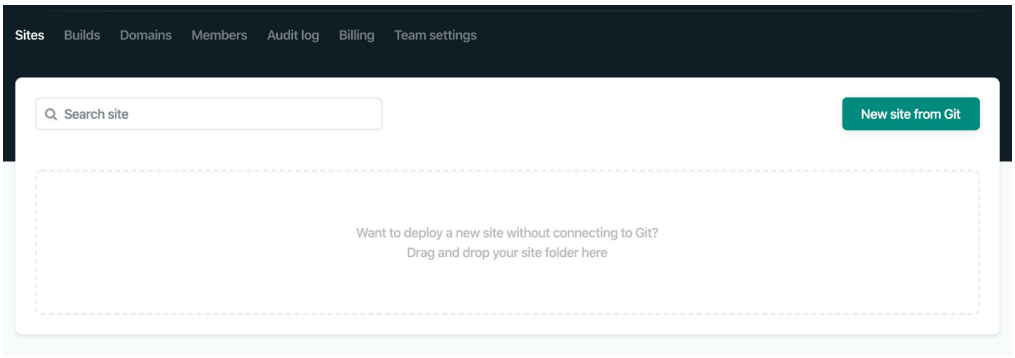
With Netlify, you have two options, you can either drag and drop your built files or import it from GitHub.

Let's start with the first one by creating a new app with this command:

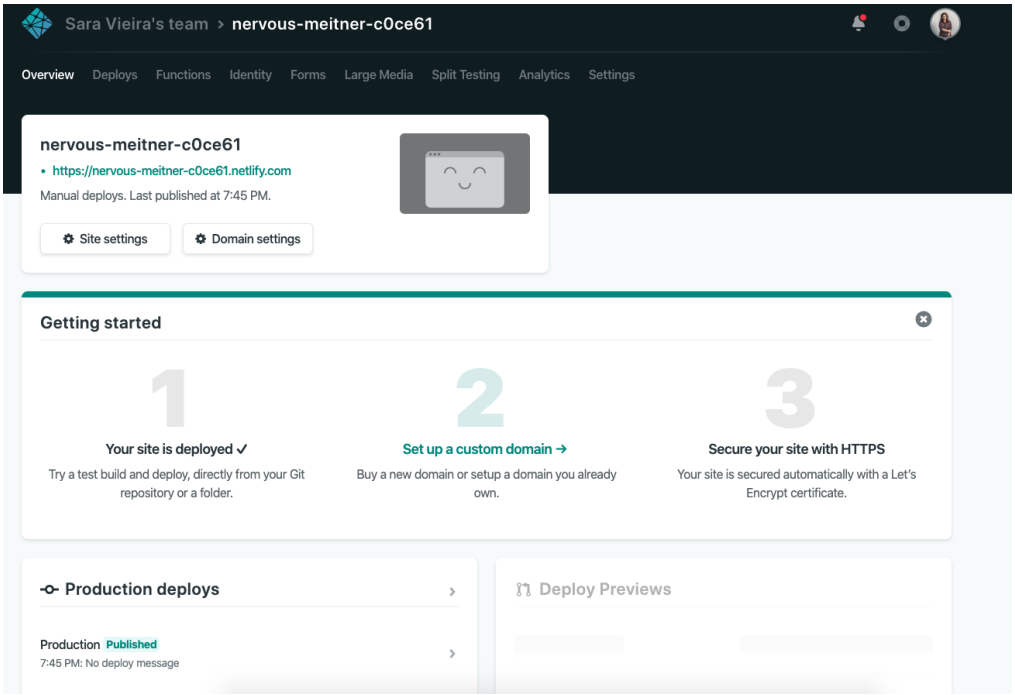
```
create-react-app example-netlify
```

After this, you need to `cd` into `example-netlify` and run `yarn build`, this will build all the static files you need in the `build` folder.

At this point, we can head over to Netlify and create an account. After doing so, you will be presented with a page that looks like this:



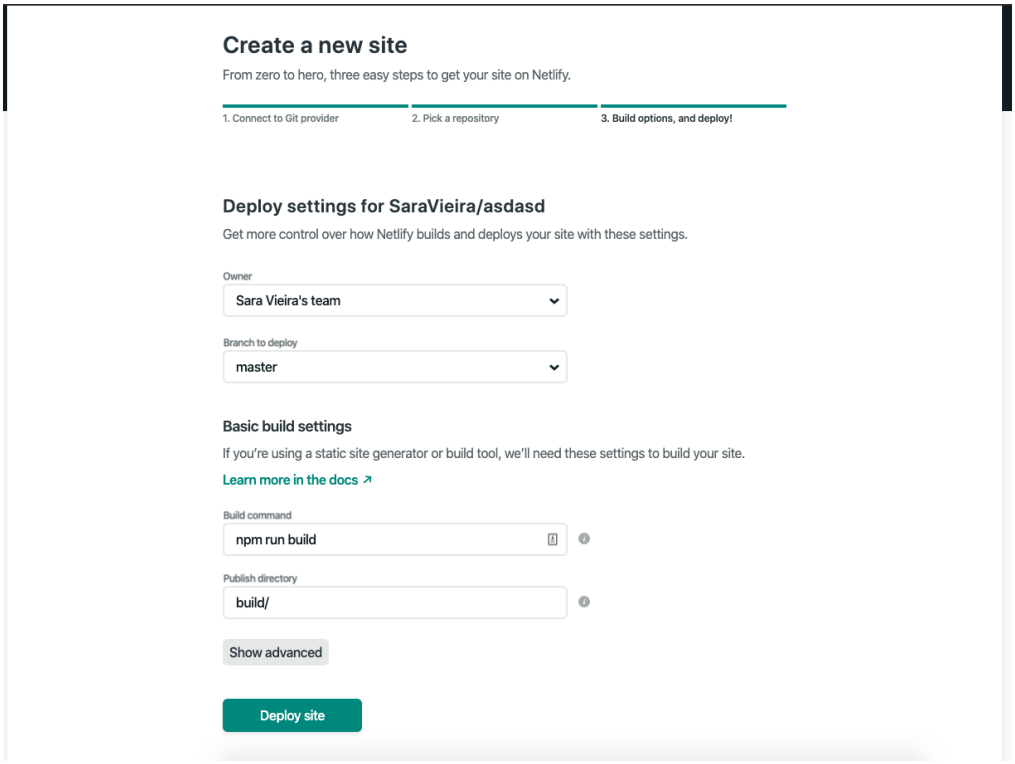
To deploy using drag and drop, you can drag your build folder into the bottom, and then you will be redirected to the page for that project, it should look something like:



In here, you can get the link at the top and go visit your newly created and deployed website. 🎉

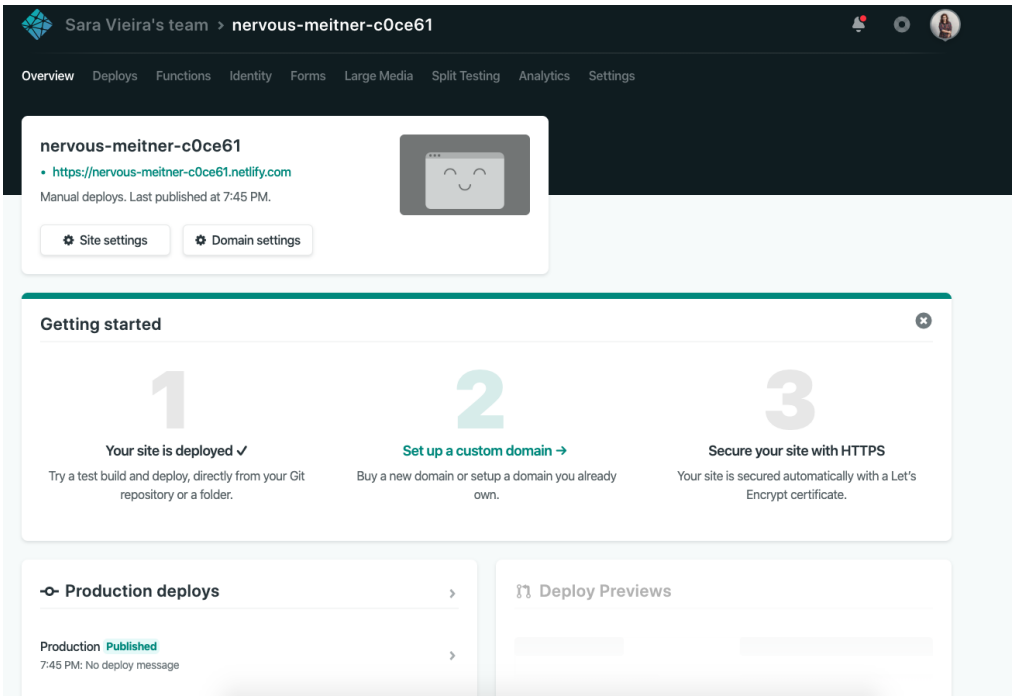
The drag and drop method is fine, but it can get quite cumbersome if you make a lot of changes, so another way of using Netlify is to connect a project with git.

To do that, you can head over to https://app.netlify.com and click the green button that says "New site from Git", with that you will be taken into a page where you can select what git provider you use. After selecting that you will see this:



Here you can search for your new project, then you will be taken to the last step to make adjustments to which `build` scripts will be run and where the built files go to.

When using `create-react-app`, all these things will be auto-filled for you, so you just got to click "Deploy Site."



And that's it! You now have a fully connected app that will be rebuilt and deployed every time you make a new commit. It will also trigger a deploy for each pull request on Github.

Vercel

Let's continue working with the same CRA folder for Vercel. First, we need to install their CLI, we can do that by running:

```
yarn global add vercel
```

The first step is to create an account with Vercel so you can have access to the platform. You can do that at their website.

After this is done, we can go to the terminal and type:

```
vercel login
```

It will ask for your email, the one you used to create the account. They will then send you an email to confirm and finish your login.

Once logged in you can deploy by typing:

```
vercel
```

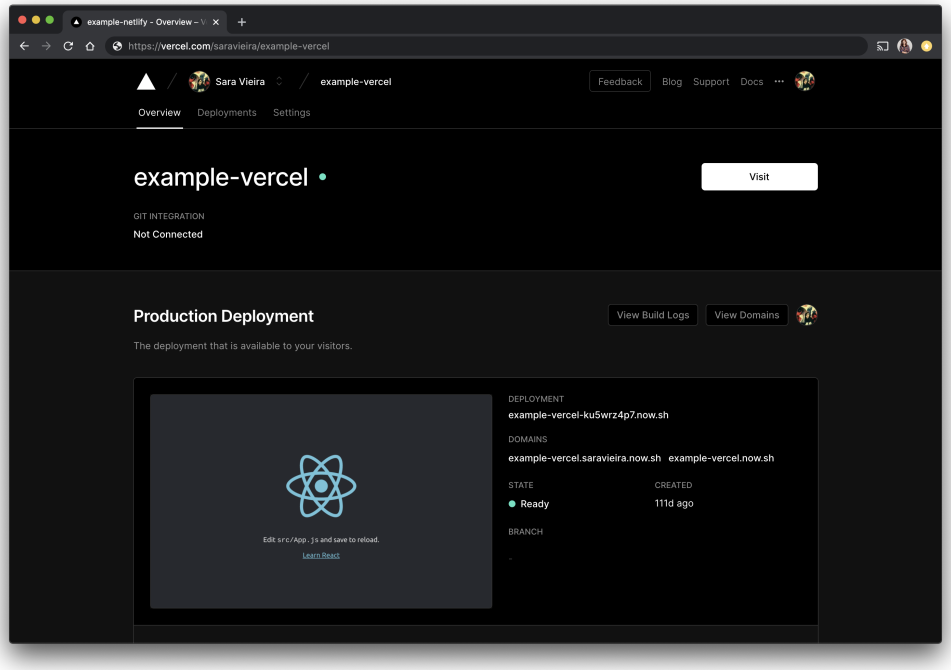
They will ask a couple of questions, you can press `enter` through all of them as we will be using the default settings.

You will see something like this:

```
Auto-detected project settings (Create React App):
- Build Command: `react-scripts build` or `build` from
`package.json`
- Output Directory: build
- Development Command: react-scripts start
? Want to override the settings? [y/N] y
? Which settings would you like to overwrite (select
multiple)? None
🔗  Linked to saravieira/example-vercel (created .vercel
and added it to .gitignore)
🔍  Inspect: https://vercel.com/saravieira/example-
vercel/ku5wrz4p7 [1s]
✅  Production: https://example-vercel.now.sh [copied to
clipboard] [55s]
📝  Deployed to production. Run `vercel --prod` to
overwrite later (https://vercel.com/docs/cli#getting-
started).
💡  To change the domain or build command, go to
https://vercel.com/saravieira/example-vercel/settings
```

Our project has been deployed to `https://example-vercel.now.sh`, and it's available online.

As we did with Netlify, it's also possible to link our deployment to your Github repo by going on the dashboard and selecting the project.



To add a GitHub repo, you can click on the `edit` button on the top left next to the Github logo and link your repository.

Gatsby

Gatsby deployment works in the same way as CRA deployment, for both Netlify and Vercel, as the settings are automatically figured out.

Next

Netlify

In the case of `Next`, we can't really use the out of the box methodologies in Netlify, since `Next` by default is a server-side rendered app and Netlify only supports static sites, in other words, Netlify needs an HTML file to be able to host your website.

In case you really need to host your `Next` site in Netlify, you can always statically export it with `next export`.

You can do this via the Netlify settings page, after importing your `Next` app, you need to add the following values:

- Build Command: `next build && next export`
- Publish directory: `out`

This will make your `Next` app build the static files into the `out` folder, which are the ones that can be deployed to Netlify.

If you are using the drag and drop method, you can add a new script to your `package.json` that looks like so:

```
"build:static": "next build && next export"
```

Once this is run, you can drag and drop the `out` directory into the Netlify UI for deployment.

Vercel

Next is a Vercel project, so you would expect this process to be pretty seamless, and I am not going to disappoint you.

If you have a `Next` app that you would like to deploy to Vercel, you can go to the command line and type:

```
vercel
```

That's it! It's in the cloud now. You have just deployed your website, and you didn't even cry once.

The Lingo Glossary

SSR

Server-Side Rendering

I use this term a lot when talking about things like `next`, and it means that your application will be rendered on the server before getting it fully rendered to the client. This helps a lot with both perceived performance and search engine optimization (SEO).

PWA

Progressive Web App

In simple terms, a PWA is a website/app that can be added to your phone's home screen and usually works offline by using service workers. They also have fancy loading screens.

CRA

Create React App

The fastest way to get started with React locally, it clones a project ready to go using `react-scripts`.

Monorepo

Basically one repo with folders for projects

At CodeSandbox, we use monorepos with `lerna`, which allows you to have all your mini-applications in one repo and in different `packages`. It makes it easier to run things in parallel and import packages that are meant to be used in the same application.

Hydration

Get that SSR data to JS

This refers to the JavaScript process of getting your Static HTML from the server and turning it into dynamic DOM that react can modify.

JAMSTACK

Who needs servers?

JAMSTACK stands for JavaScript, APIs, and Markup, and it's a way of building websites that is not reliant on a server. One of the most know frameworks for this is Gatsby, as it generates static HTML. You can read more about this in JAMSTACK website.

SSG

Static Site Generation

This a term mainly use in Gatsby and Next circles as it's the ability to get dynamic data and through a build pipeline make this into HTML pages. Like SSR this is very good for performance but also a very good technology to help keep our website as small and fast as possible.

Resources

I made a page with all the sandboxes that I mentioned in the book and you can find it at https://opinionatedreact.com/resources.

This page includes three starters for your projects:

- Next: A Next starter with Chakra UI, styled-components & Overmind, all working server side. This starter includes two client routes and one API route to demonstrate how you can make an API using only Next.
- Create React App: A Create React App starter with Chakra UI, styled-components, Overmind, and the latest version of react-router. This starter includes a small application that allows you to add movies to a list fetching the poster in the background.
- Gatsby: A Gatsby starter with Chakra UI, styled-components & Overmind, all working server side. This starter includes the same small application and also a page demonstrating how to get data from a markdown file.

Hope these examples and starters prove useful to you.

Conclusion

During the course of these short chapters, we've covered a lot of highly opinionated tricks, tips, and conventions around React that often are not as obvious on our first tries - if we get lucky, we'll start feeling familiar with do's, and don'ts after a few weeks.

Nothing that has been written here is an absolute truth - everything is highly subjective. I talk about the tools that have served me the best for the situations I've encountered, but by all means: please go, explore, and share your thoughts and ideas. This community needs all of us.

This book is also gonna grow organically as the React community and, therefore - plugins and alternatives - keep on evolving. Which if we've been paying attention to the frontend ecosphere, it's gonna happen every couple of months or so. Even during the time I was actively adding new stuff to this book before calling it "done" for the first time, I ended up going back and changing things - Reach Router was the original router alternative I recommended before it was merged with React Router, and Recoil hadn't made us all talk about how necessary a state management library is - or not - nowadays.

This book has also once again confirmed the power of learning through sharing - writing this book was a highly educational experience for me, one that has just started. Not only around the experience of writing and thinking beyond a blog post, but also about React itself, and as a bonus, I've learned a bit about thinking thoroughly which subjective things that I may not always be a fan of, are beneficial and helpful on the right context. For example, I don't get Docker; therefore, I don't like it, but that doesn't mean it's bad, it just means that I don't get it, and it makes me feel stupid. There's always a reason for the strong opinions we see in tech, a lot of the times, that has little to do with the

technology itself, it's either something within us, or it can be something completely foreign.

Beyond any opinion about implementation details and conventions that come and go, I would like to close this book - for today - by sharing the most important lesson the development world has gifted me with.

Tech comes and goes. Anyone can learn how to code if you put the hours and intention. But there's one thing that cannot be learned through tutorials or YouTube videos, and it's the most valuable asset that our community has. Be a decent human being. Care about your fellow developers and aim to leave this environment better than you found it, and this goes beyond adding types - or even, may the time permit, gasp: comments.

Being a decent human being is about caring for the people that will work with your code next. By remembering that you didn't always know the things you know now and that those things will likely be challenged by others' teachings. The most productive developer environments are not the ones that make a billionaire product that goes beyond series A, those environments are the ones that are full of people that bring out the best in you and care for each other.

Don't be a dick.

If you made it this far, thank you for taking the time to read these extra tidbits - cough cough, rants -. I appreciate our community and you, thanks for being you.

Until next time!