What is Create React App?

Create React App is an officially supported CLI tool from Facebook to set up React apps, without having to deal with complicated configurations. It provides a default setup for tools like webpack and Babel that are useful in a modern development pipeline.

Gatsby is similar in that it can also help you set up an application and removes much of the configuration headache. However, Gatsby offers some additional advantages like performance optimizations with static rendering and a thriving ecosystem of plugins. Gatsby and Create React App even share nearly identical webpack and Babel configurations which makes porting less cumbersome. React's own docs run on Gatsby, and React even recommends Gatsby to users of Create React App!

What do you get by changing to Gatsby?

Both Create React App and Gatsby use React and allow users to build apps more quickly. However, there are some important differences.

Performance optimizations

Gatsby enables some useful performance optimizations for you, by default. Route-based code splitting and pre-loading of the *next* resources make your application lightning fast, without any additional effort! Further still, the following optimizations and techniques are also available to you:

- `gatsby-link` uses an intersection observer to preload linked pages when they appear in the viewport, making them feel like they load *instantly*
- `gatsby-plugin-image` will create optimized versions of your images in different sizes, loading a smaller, optimized version of an image and replacing it with a higher resolution version when loading has finished. It also uses an intersection observer to cheaply lazy load images. Check out the demo to see for yourself

These features and more work together to make your Gatsby site fast by default. If you'd like a deep dive on these and other performance optimizations, check out this blog post

Expansive ecosystem

Gatsby still works with all the `react` packages running in a Create React App project, but adds the ability to leverage hundreds of other Gatsby plugins to drop in additional features such as: adding SEO, responsive images, RSS feeds, and much, much more.

Unified GraphQL data layer

Plugins can also pull in data from any number of sources like APIs, CMSs, or the filesystem. That data is combined into a unified data layer that you can query with GraphQL throughout your app.

This data layer simplifies the process of pulling data from different sources and providing them in your pages and components. This combination of data from different sources stitched together in a modern workflow is referred to as the content mesh.

> ***Note**: GraphQL isn't required for managing data in a Gatsby app. Feel free to look over the guide on using Gatsby without GraphQL as well*

What changes need to be made?

In order to transition your codebase over to using Gatsby, a few things need to be taken care of to account for the differences between how the projects are set up.

In order to use Gatsby, you have to install it:

```
npm install gatsby
```

> ***Note**: rather than using the `gatsby new` command like you would initializing a new Gatsby site, this will install Gatsby as a dependency in your project*

After installation, the key things that need to change are:

1. ensuring any calls to browser-based APIs still work (if there are any)
2. converting routes into pages in the `/pages` directory

The following sections explain the above steps as well as other changes that you might need to make depending on the complexity of your app. A default Create React App project is able to run with the above steps.

Project structure

To show some of the differences of how your project structure could differ by moving to Gatsby, a default Create React App project looks something like this:

```
  my-create-react-app
  ├── .git
  ├── .gitignore
  ├── README.md
  ├── node_modules
  ├── package.json
  ├── src
  │   ├── App.css
  │   ├── App.js
  │   ├── App.test.js
  │   ├── index.css
  │   ├── index.js
  │   ├── logo.svg
  │   └── serviceWorker.js
  └── yarn.lock
```

Whereas a default Gatsby project will look something like this (files that are different between Create React App and Gatsby are highlighted):

```
  my-gatsby-site
  ├── .git
  ├── .gitignore
+ ├── .prettierrc
+ ├── LICENSE
  ├── README.md
+ ├── gatsby-browser.js
+ ├── gatsby-config.js
+ ├── gatsby-node.js
```

```
+ ├── gatsby-ssr.js
  ├── node_modules
  ├── package.json
  ├── src
+ │   ├── components
+ │   ├── images
+ │   └── pages
  └── yarn.lock
```

The structure of a Gatsby project adds some additional configuration files to hook into specific Gatsby APIs for the browser and for server-side rendering, though much of the project structure is similar enough to feel comfortable quickly.

An important difference to note is the `/pages` folder in the Gatsby project structure where components will automatically be turned into static pages by Gatsby. This is discussed more in the routing section below.

The `src/pages/index.js` file in Gatsby is a little different from the `src/index.js` file in Create React App where the root React element mounts, because it is a sibling to other pages in the app -- rather than a parent like Create React App. This means in order to share state or wrap multiple pages you should use the wrapRootElement Gatsby API.

Server-side rendering and browser APIs

Server-side rendering means pages and content are built out by the server, and then sent to a browser ready to go. It's like your pages are constructed before even being sent to the user. Gatsby is server-side rendered at build time, meaning that the code that gets to your browser has already been run to build pages and content, but this doesn't mean you can't still have dynamic pages.

Understanding the distinction between the client (or browser) and server will help you understand that key difference between Create React App and Gatsby. Create React App doesn't by default render your components with server-side rendering APIs when it is built like Gatsby does.

Check out Dustin Schau's blog post about Gatsby internals that explains the technical aspects of the build process in greater detail

The `gatsby build` command also won't be able to use browser APIs, so some code that relies on browser DOM APIs and globals will cause your build to break if it isn't protected. Learn more about using client-side only packages.

Some common globals that would need to be protected are:

- `window`
- `localStorage`
- `sessionStorage`
- `navigator`
- `document`

Additionally, some packages that depend on globals existing (e.g. `react-router-dom`) may need to be patched or migrated to other packages.

These are only a few examples, though all can be fixed in one of two ways:

1. wrapping the code in an `if` statement to check if whatever you are referencing is defined so builds won't try to reference something undefined. Then the browser will be able to deal with it fine when the condition matches:

```
if (typeof window !== `undefined`) {
  // code that references a browser global
  window.alert("Woohoo!")
}
```

2. For class components: moving references to browser globals into a `componentDidMount`

```
import React, { Component } from "react"

class MyComponment() extends Component {
  render() {
    window.alert("This will break the build")

    return (
      <div>
        <p>Component</p>
      </div>
    )
  }
}
```

Would be changed into:

```
import React, { Component } from "react"

class MyComponment() extends Component {
  componentDidMount() {
    // code that references the browser global
    window.alert("This won't break the build")
  }

  render() {
    return (
      <div>
        <p>Component</p>
      </div>
    )
  }
}
```

3. For function components: moving references to browser globals into a `useEffect` hook

```
import React from "react"

const Foo = () => {
  window.alert("This will break the build")
  return <span>Bar</span>
```

```
}

export default Foo
```

Would be changed to:

```
import React from "react"

const Foo = () => {
  React.useEffect(() => {
    window.alert("This won't break the build")
  })
  return <span>Bar</span>
}

export default Foo
```

If these browser globals aren't protected correctly, you'll see a webpack error like the one below when building your site:

```
WebpackError: ReferenceError: window is not defined
```

For more information about errors encountered during builds, see the doc on debugging HTML builds. For more information about React hooks, check out the React docs.

Routing

There are two possibilities of routes that you can set up: static and client-only. Gatsby automatically turns React components in the pages folder into static routes. This means that a statically rendered page directly corresponds to index.html files in the app's built assets, whereas a client-only route is rendered by the client and configured by the routing setup you define. Both types of pages can fetch data at runtime just like any other React app.

> ***Note**: An advantage to having pages in separate files like this is a defined way of automatically code splitting, whereas Create React App requires you to use the `import()` syntax to assign what elements should be loaded dynamically.*

For dynamic routes, you should implement routing with @reach/router, which is already included with Gatsby. Dynamic routes can be implemented the same way you would implement a router in Create React App (or any other React application). However, because these routes won't be represented as HTML files in the final build, if you want users to be able to visit the routes directly (like entering the URL in the search bar), you'll need to generate pages in the `gatsby-node.js` file which is demonstrated in the Building Apps with Gatsby guide.

```
import React from "react"
import { Router } from "@reach/router"

const App = () => (
  <Router>
    <Route path="/user/" component={Users} />
    <Route path="/user/:id" component={UserDetails} />
  </Router>
)
```

```
export default App
```

Gatsby provides a `<Link />` component and a `navigate` function to help you direct users through pages on your site. You can read about how to use each in the `gatsby-link` doc.

Handling state

Because Gatsby rehydrates into a regular React app, state can be handled inside of components in the same way it would in Create React App. If you use another library for state management and want to wrap your app in some sort of global state the section on context providers will be helpful.

Environment variables

Create React App requires you to create environment variables prefixed with `REACT_APP_`. Gatsby instead uses the `GATSBY_` prefix to make environment variables accessible in the browser context.

```
REACT_APP_API_URL=http://someapi.com
```

```
GATSBY_API_URL=http://someapi.com
```

Advanced customizations

Part of Gatsby's philosophy around tooling is progressively disclosing complexity; this simplifies the experience for a wider audience while still allowing the option to configure more advanced features for those that feel inclined. You won't have to "eject" your Gatsby app to edit more complex configurations.

In terms of levels of abstraction, Gatsby allows you to move up or down to tap into more sophisticated, lower-level APIs without needing to eject like you would in Create React App.

webpack

Create React App will require you to eject or rely on another workaround to edit the webpack configuration. Gatsby allows custom configuration of webpack via the `gatsby-node.js` file.

Context providers

React's context API allows you to share state from a higher component and distribute it to components below it in the component tree without having to deal with issues like prop drilling.

How do you share state across components like a theme without one top level `App.js` file? Gatsby has a `wrapRootElement` and a `wrapPageElement` API that allow you to wrap the root element or all pages of your Gatsby site with components you want.

In Create React App it could look like this:

```
import React from "react"

const defaultTheme = "light"
const ThemeContext = React.createContext(defaultTheme)

function App() {
```

```
  return (
    <ThemeContext.Provider value={defaultTheme}>
      {/* App, routing, and other components */}
    </ThemeContext.Provider>
  )
}

export default App
```

In Gatsby, if you want your providers to be global across pages you would move those providers to `gatsby-browser.js` and `gatsby-ssr.js` :

```
import React from "react"

const defaultTheme = "light"
export const ThemeContext = React.createContext(defaultTheme)

export const wrapRootElement = ({ element }) => {
  return (
    <ThemeContext.Provider value={defaultTheme}>
      {element}
    </ThemeContext.Provider>
  )
}
```

```
import React from "react"

const defaultTheme = "light"
export const ThemeContext = React.createContext(defaultTheme)

export const wrapRootElement = ({ element }) => {
  return (
    <ThemeContext.Provider value={defaultTheme}>
      {element}
    </ThemeContext.Provider>
  )
}
```