<https://reactjs.org/docs/create-a-new-react-app.html>

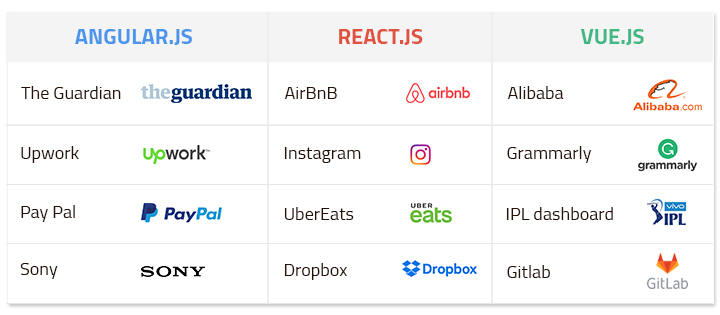
<https://github.com/facebook/create-react-app>

npx create-react-app my-app

npm init react-app my-app

yarn create react-app my-app

Grouping by features or routes vs Grouping by file type



## Una mejor manera de estructurar los proyectos de React

¡Hola a todos! Ya se ha derramado mucha tinta electrónica en las selecciones relativamente más fáciles de "Hacer X en React" o "Usar React con tecnología X".

Así que en su lugar, quiero hablar sobre las experiencias que he tenido construyendo frontends desde cero en [DelightChat](https://delightchat.io/) y en mis compañías anteriores.

Estos proyectos requieren una comprensión más profunda de React y un uso extendido en un entorno de producción.

Si desea ver una versión en video de este tutorial para complementar su lectura, [puede hacerlo aquí](https://www.youtube.com/watch?v=lViIdphWTwY).

**Introducción**

En pocas palabras, un proyecto complejo de React debería estructurarse así. Aunque uso NextJS en producción, esta estructura de archivos debería ser bastante útil en cualquier configuración de React. (Me falta ver el concepto de reducers y actions)

src

|---adapters

|---contexts

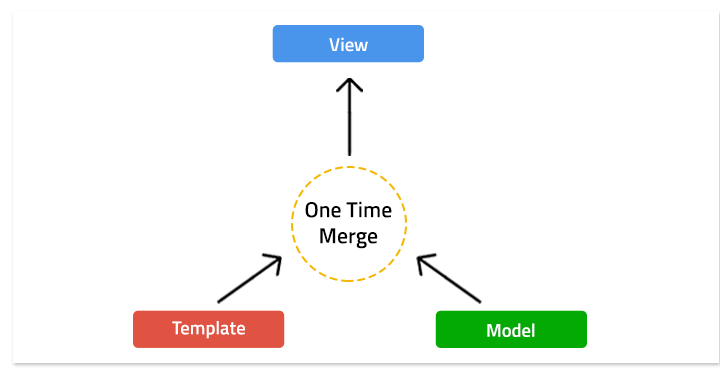
|---components

|---styles

|---pages

*Nota: En la estructura de archivos anterior, los activos o archivos estáticos deben colocarse en cualquiera que sea la variante de* *carpeta para su marco.*public \*

Para cada una de las carpetas anteriores, vamos a discutirlas en orden de precedencia.



* Virtual DOM ayuda a los desarrolladores de React a renderizar fácilmente cualquier cambio en la aplicación sin afectar las otras partes de la interfaz.
* React permite a los usuarios reutilizar componentes. Esto hace que el desarrollo de aplicaciones sea más fácil y eficiente.
* React utiliza [Redux](https://redux.js.org/basics/usage-with-react" \t "_blank), que simplifica el almacenamiento y la gestión de los estados de los componentes en grandes aplicaciones.
* El enlace de datos unidireccional hace que el código sea estable y admite el desarrollo futuro a medida que el flujo de datos se dirige hacia una dirección.

**1. Adaptadores**

Los adaptadores son los conectores de su aplicación con el mundo exterior. Cualquier forma de llamada API o interacción websocket que deba ocurrir, para compartir datos con un servicio o cliente externo, debe ocurrir dentro del propio adaptador.

Hay casos en los que algunos datos siempre se comparten entre todos los adaptadores, por ejemplo, el intercambio de cookies, URL base y encabezados en sus adaptadores AJAX (XHR). Estos pueden inicializarse en la carpeta xhr y luego importarse dentro de sus otros adaptadores para usarlos más.

Esta estructura se verá así:

adapters

|---xhr

|---page1Adapter

|---page2Adapter

En el caso de axios, puede usarlo para crear un adaptador base y exportar esta instancia inicializada o crear diferentes funciones para obtener, publicar, parchear y eliminar para abstraerlo aún más. Esto se vería así: axios.create

// adapters/xhr/index.tsx

import Axios from "axios";

function returnAxiosInstance() {

return Axios.create(initializers);

}

export function get(url){

const axios = returnAxiosInstance();

return axios.get(url);

}

export function post(url, requestData){

const axios = returnAxiosInstance();

return axios.post(url, requestData);

}

... and so on ...

Una vez que tengas listo el archivo base (o archivos), crea un archivo de adaptador independiente para cada página o cada conjunto de funcionalidades, según la complejidad de la aplicación. Una función bien nombrada hace que sea muy fácil entender lo que hace cada llamada a la API y lo que debe lograr.

// adapters/page1Adapter/index.tsx

import { get, post } from "adapters/xhr";

import socket from "socketio";

// well-named functions

export function getData(){

return get(someUrl);

}

export function setData(requestData){

return post(someUrl, requestData);

}

... and so on ...

Pero, ¿cómo serán útiles estos adaptadores? Averigüemos en la siguiente sección.

**2. Componentes**

Aunque en esta sección deberíamos hablar de contextos, quiero hablar primero de componentes. Esto es para entender por qué el contexto es requerido (y necesario) en aplicaciones complejas.

Los componentes son el alma de su aplicación. Tendrán la interfaz de usuario para su aplicación y, a veces, pueden contener la lógica empresarial y también cualquier estado que deba mantenerse.

En caso de que un componente se vuelva demasiado complejo para expresar Business Logic con su interfaz de usuario, es bueno poder dividirlo en un archivo bl.tsx separado, con su index.tsx raíz importando todas las funciones y controladores desde él.

Esta estructura se vería así:

components

|---page1Components

|--Component1

|--Component2

|---page2Component

|--Component1

|---index.tsx

|---bl.tsx

En esta estructura, cada página tiene su propia carpeta dentro de los componentes, de modo que es fácil averiguar qué componente afecta a qué.

También es importante limitar el alcance de un componente. Por lo tanto, un componente solo debe usar adaptadores para la obtención de datos, tener un archivo independiente para la lógica empresarial compleja y centrarse solo en la parte de la interfaz de usuario.

// components/page1Components/Component1/index.tsx

import businessLogic from "./bl.tsx";

export default function Component2() {

const { state and functions } = businessLogic();

return {

// JSX

}

}

Mientras que el archivo BL solo importa datos y los devuelve:

// components/page1Components/Component1/bl.tsx

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

import { adapters } from "adapters/path\_to\_adapter";

export default function Component1Bl(){

const [state, setState] = useState(initialState);

useEffect(() => {

fetchDataFromAdapter().then(updateState);

}, [])

}

Sin embargo, hay un problema que es común en todas las aplicaciones complejas. Administración de estado y cómo compartir el estado entre componentes distantes. Por ejemplo, considere la siguiente estructura de archivos:

components

|---page1Components

|--Component1

|---ComponentA

|---page2Component

|--ComponentB

Si algún estado tiene que ser compartido a través de ComponentA y B en el ejemplo anterior, tendrá que ser pasado a través de todos los componentes intermedios, y también a cualquier otro componente que desee interactuar con el estado.

Para resolver esto, hay varias soluciones que se pueden usar como Redux, Easy-Peasy y React Context, cada una de ellas con sus propios pros y contras. En general, React Context debe ser "lo suficientemente bueno" para resolver este problema. Almacenamos todos los archivos relacionados con el contexto en contextos.

**3. Contextos**

La carpeta de contextos es una carpeta mínima que solo contiene el estado que debe compartirse entre estos componentes. Cada página puede tener varios contextos anidados, y cada contexto solo pasa los datos hacia adelante en dirección descendente. Pero para evitar la complejidad, es mejor tener un solo archivo de contexto. Esta estructura se verá así:

contexts

|---page1Context

|---index.tsx (Exports consumers, providers, ...)

|---Context1.tsx (Contains part of the state)

|---Context2.tsx (Contains part of the state)

|---page2Context

|---index.tsx (Simple enough to also have state)

En el caso anterior, dado que puede ser un poco más complejo, permitimos un contexto anidado pasando el contexto secundario como un niño al padre. Sin embargo, generalmente un solo archivo que contenga el estado y exporte los archivos relevantes debería ser suficiente.page1index.tsx

No entraré en la parte de implementación de las bibliotecas de administración de estado de React, ya que cada una de ellas son sus propias bestias y tienen sus propias ventajas y desventajas. Por lo tanto, recomiendo pasar por el tutorial de lo que decida usar para aprender sus mejores prácticas.

El contexto puede importar desde adaptadores para buscar y reaccionar a efectos externos. En el caso de React Context, los proveedores se importan dentro de las páginas para compartir el estado en todos los componentes, y algo así se utiliza dentro de estos componentes para poder utilizar estos datos.useContext

Pasando a la última pieza importante del rompecabezas, las páginas.

**4. Páginas**

Quiero evitar estar sesgado a un marco para esta pieza, pero en general, tener una carpeta específica para colocar los componentes de nivel de ruta es una buena práctica.

Gatsby y NextJS obligan a tener todas las rutas en una carpeta llamada páginas. Esta es una forma bastante legible de definir componentes de nivel de ruta, e imitar esto en su aplicación generada por CRA también daría como resultado una mejor legibilidad del código.

Una ubicación centralizada para las rutas también le ayuda a utilizar la funcionalidad "Ir al archivo" de la mayoría de los IDE saltando a un archivo mediante (Cmd o Ctrl) + Haga clic en una importación.

Esto le ayuda a moverse a través del código rápidamente y con claridad de lo que pertenece a dónde. También establece una jerarquía clara de diferenciación entre páginas y componentes, donde una página puede importar un componente para mostrarlo y no hacer nada más, ni siquiera Business Logic.

Sin embargo, es posible importar proveedores de contexto dentro de su página para que los componentes secundarios puedan consumirla. O, en el caso de NextJS, escriba algún código del lado del servidor que pueda pasar datos a sus componentes usando getServerSideProps o getStaticProps.

**5. Estilos**

Finalmente, llegamos a los estilos. Aunque mi forma de ir es simplemente incrustar estilos dentro de la interfaz de usuario mediante el uso de una solución CSS-in-JS como Styled-Components, a veces es útil tener un conjunto global de estilos en un archivo CSS.

Un archivo CSS antiguo es más fácil de compartir entre proyectos y también puede afectar al CSS de los componentes a los que los componentes con estilo no pueden alcanzar (por ejemplo, componentes de terceros).

Por lo tanto, puede almacenar todos estos archivos CSS dentro de la carpeta de estilos e importarlos o vincularlos libremente desde donde desee.

Esos eran mis pensamientos. ¡Siéntase libre de enviarme un correo electrónico en caso de que quiera discutir algo o tener más aportes sobre cómo se puede mejorar esto!

Para más actualizaciones o discusiones, puedes seguirme en Twitter [aquí](https://twitter.com/thewritingdev).

Mi último artículo en freeCodeCamp fue escrito sobre cómo puedes comenzar con Deno construyendo un acortador de URL, que [puedes leer aquí](https://www.freecodecamp.org/news/build-a-url-shortener-in-deno/).