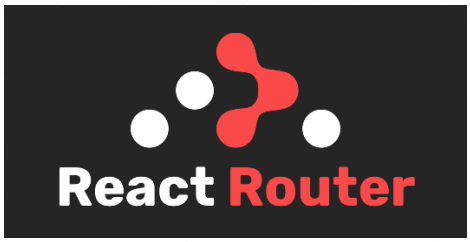


Ing. Luis Guillermo Molero Suárez

React Router

React Router es la biblioteca de enrutamiento estándar de facto para React. Cuando necesite navegar a través de una aplicación React con múltiples vistas, necesitará un enrutador para administrar las URL. React Router se encarga de eso, manteniendo la interfaz de usuario de la aplicación y la URL sincronizadas.

Introducción

React es una biblioteca popular para crear Single Page Application (SPA) que se representan en el lado del cliente. Un SPA puede tener varias vistas (también conocidas como páginas) y, a diferencia de las aplicaciones convencionales de varias páginas, navegar a través de estas vistas no debería provocar que se vuelva a cargar la página completa. En su lugar, queremos que las vistas se representen en línea dentro de la página actual. El usuario final, que está acostumbrado a las aplicaciones de varias páginas, espera que las siguientes funciones estén presentes en un SPA:

- Cada vista debe tener una URL que especifique de manera única esa vista. Esto es para que el usuario pueda marcar la URL como referencia en un momento posterior.
- El botón de avance y retroceso del navegador debería funcionar como se esperaba.
- Las vistas anidadas generadas dinámicamente también deben tener preferiblemente una URL propia.

El enrutamiento es el proceso de mantener la URL del navegador sincronizada con lo que se muestra en la página.

Para ampliar la información, visitar: https://reactrouter.com/

Para conocer los `templates` de React Router y poder desplegar mayores funcionalidades, visitar: https://reactrouter.com/web/guides/quick-start

React Router le permite manejar el enrutamiento de forma declarativa. El enfoque de enrutamiento declarativo le permite controlar el flujo de datos en su aplicación, diciendo "la ruta debería verse así":

```
<Route path="/about">
  <About />
</Route>
```

Puede colocar su componente <Route> en cualquier lugar donde desee que se represente su ruta. Dado que <Route>, <Link> y todas las demás API de React Router con las que nos ocuparemos son solo componentes, puede comenzar fácilmente con el enrutamiento en React.

"Existe la idea errónea de que React Router es una solución de enrutamiento oficial desarrollada por Facebook. En realidad, es una biblioteca de terceros que es muy popular por su diseño y simplicidad."

Configuración de React Router

Para iniciar, se necesita una versión reciente de Node Js instalada, si este no es el caso, visité la página de inicio de Node Js y descargue los binarios correctos para su sistema. https://nodejs.org/en/download/

Asimismo, Node Js viene con npm, un administrador de paquetes para JavaScript, con el que se instalarán algunas de las bibliotecas a utilizar. Puede obtener más información sobre el uso de npm visite la siguiente página:

https://docs.npmjs.com/

Puede comprobar que ambos están instalados correctamente emitiendo los siguientes comandos desde la línea de comandos:

```
node -v
npm -v
```

Una vez hecho esto, se inicia creando un nuevo proyecto React con la herramienta `Create React App`. Para ello, cree una carpeta de proyecto, habrá el CMD y ejecute:

```
npx create-react-app react-router-demo
```

Cuando esto haya terminado, es necesario cambiar al directorio recién creado:

```
cd react-router-demo
```

Por otra parte, la biblioteca React Router comprende tres paquetes:

- `react-router`
- `react-router-dom`
- `react-router-native`.

El paquete principal del enrutador es `react-router`, mientras que los otros dos son específicos del entorno. Debe usar `react-router-dom` si está creando un sitio web, y `react-router-native` si se encuentra en un entorno de desarrollo de aplicaciones móviles con React Native.

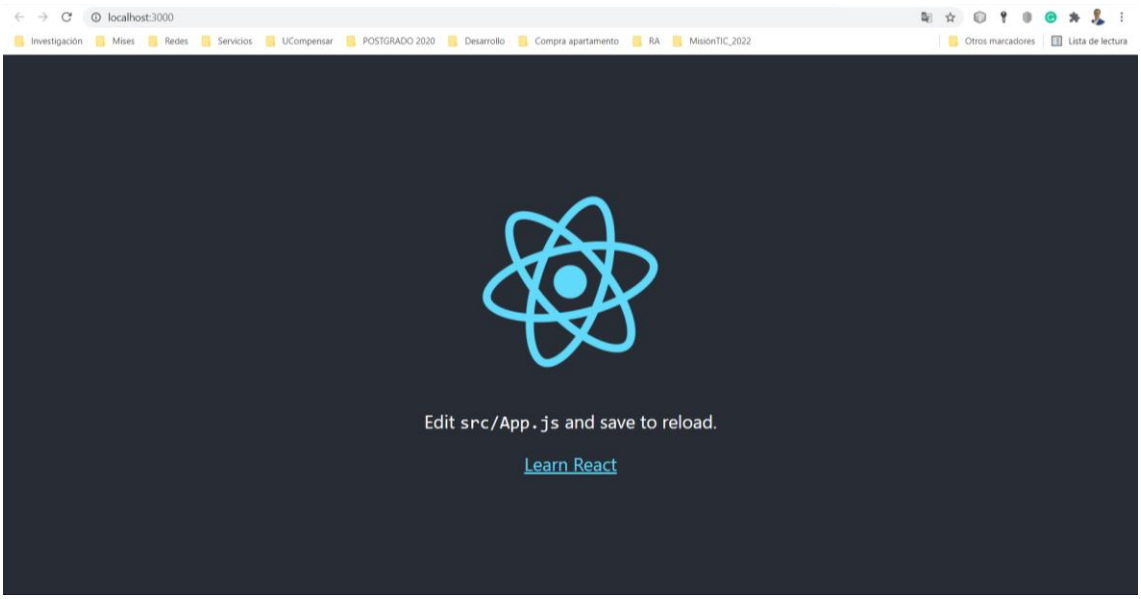
Utilice npm para instalar `react-router-dom` de forma local:

```
npm install react-router-dom
```

Luego inicie el servidor de desarrollo con esto:

```
yarn start
```

Observará la siguiente SPA:



Ahora tiene una aplicación react en funcionamiento con react router instalado.

Ahora, como primer paso, insertamos el siguiente código en los <meta> del archivo `index.html` que se encuentra en la carpeta `/public`, de la forma:

```
<link
      id="external-css"
      rel="stylesheet"
      type="text/css"
      href="https://bootswatch.com/3/paper/bootstrap.min.css"
      media="all"
    />
```

Conceptos básicos de react router

Para empezar a conocer acerca de react router, se creará una aplicación con tres vistas separadas:

- Inicio
- Categoría
- Productos.

El componente `Router`

Lo primero, es envolver nuestro componente <App> en un componente <`Router`> (proporcionado por React Router). Dado que se está creando una aplicación basada en navegador, se pueden usar dos tipos de enrutadores de la API React Router:

- `BrowserRouter`
- `HashRouter`

La principal diferencia entre ellos es evidente en las URL que crean:

```
// <BrowserRouter>
http://example.com/about

// <HashRouter>
http://example.com/#/about
```

<`BrowserRouter`> es el más popular de los dos porque usa la API de historial de HTML5 para mantener su interfaz de usuario sincronizada con la URL, mientras que <`HashRouter`> usa la parte hash de la URL (`window.location.hash`).

"Si necesita admitir navegadores heredados que no admiten la API de historial, debe usar <HashRouter>. De lo contrario, <BrowserRouter> es la mejor opción para la mayoría de los casos de uso. "

Puedes leer más sobre las diferencias visitando la siguiente página:

https://stackoverflow.com/questions/51974369/what-is-the-difference-between-hashrouter-and-browserrouter-in-react

Entonces, importamos el componente `BrowserRouter` y envolvemos en el componente de la aplicación, de la forma:

```
// src/index.js

import React from "react";
import ReactDOM from "react-dom";
import App from "./App";
import { BrowserRouter } from "react-router-dom";

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

El código anterior crea una instancia de `history` para todo nuestro componente <App>.

¿Qué significa?... Acerca de `history`

La librería `history` le permite administrar fácilmente el historial de sesiones en cualquier lugar donde se ejecute JavaScript.

Un objeto `history` abstrae las diferencias en varios entornos y proporciona una API mínima que le permite administrar la pila de `history`, navegar y conservar el estado entre sesiones.

Cada componente <Router> crea un objeto history que realiza un seguimiento de la ubicación actual (`history.location`) y también de las ubicaciones anteriores en una pila. Cuando cambia la ubicación actual, la vista se vuelve a renderizar y tiene una sensación de navegación.

¿Cómo cambia la ubicación actual?

El objeto `history` tiene métodos como `history.push` e `history.replace` para encargarse de eso. El método `history.push` se invoca cuando hace clic en un componente <Link>, y se llama a `history.replace` cuando usa un <Redirect>. Otros métodos, como `history.goBack` e `history.goForward`, se utilizan para navegar a través de la pila de `history` retrocediendo o adelantando una página.

Componentes Link y Route

El componente <Route> es el componente más importante en React Router. Muestra (renderiza) la IU si la ubicación actual coincide con la ruta del path. Idealmente, un componente <Route> debería tener una prop llamada path, y si el nombre de la ruta coincide con la ubicación actual, se renderiza (se muestra).

El componente <Link>, por otro lado, se utiliza para navegar entre páginas. Es comparable al elemento ancla HTML. Sin embargo, el uso de enlaces de anclaje daría como resultado una actualización de página completa, lo que no queremos. Entonces, en su lugar, podemos usar <Link> para navegar a una URL en particular y hacer que la vista vuelva a renderizarse sin una actualización.

Ahora, se ha cubierto todo lo que necesario para que la aplicación funcione.

Actualizar src/App.js de la siguiente manera:

```
// src/App.js
import React from "react";
import { Link, Route, Switch } from "react-router-dom";
const Home = () => (
  <div>
    <h2>Home</h2>
  </div>
);
const Category = () => (
  <div>
    <h2>Category</h2>
  </div>
);
const Products = () => (
  <div>
    <h2>Products</h2>
  </div>
);
export default function App() {
  return (
    <div>
      <nav className="navbar navbar-light">
        <ul className="nav navbar-nav">
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/category">Category</Link>
```

```
          </li>
          <li>
            <Link to="/products">Products</Link>
          </li>
        </ul>
      </nav>
      {/* Los componentes de la ruta se representan si la propieda
d de la ruta coincide con la URL actual */}
      <Route path="/">
        <Home />
      </Route>
      <Route path="/category">
        <Category />
      </Route>
      <Route path="/products">
        <Products />
      </Route>
    </div>
  );
}
```

Aquí, se declararon los componentes de Home, `Category` y `Products` dentro de App.js.

"Cuando un componente comienza a crecer, es mejor tener un archivo separado para cada componente."

Dentro del componente App, se ha escrito la lógica para el enrutamiento. El `path` de `<Route>` coincide con la ubicación actual y se renderiza un componente. Anteriormente, el componente que debería renderizarse se pasaba como un segundo `prop`. Sin embargo, las versiones recientes de React Router han introducido un nuevo patrón de renderizado de `path`, por el cual los componentes a renderizar son hijos de `<Route>`.

Aquí "/" coincide con "/" y "`/Category`", por lo tanto, ambas rutas coinciden y se renderizan.

¿Cómo se evita eso? Se debe pasar el `prop` exacto a la `<Route>` con `path = '/'`:

```
// src/App.js

{/* Los componentes de la ruta se representan si la propiedad de l
a ruta coincide con la URL actual */}

      <Route path="/"><Home /></Route>
      <Route path="/category"><Category /></Route>
      <Route path="/products"><Products /></Route>
```

"Si desea que solo una ruta sea renderizada si el path es exactamente el mismo, debe usar el exact prop"

Enrutamiento anidado

Para crear rutas anidadas, se necesita comprender mejor cómo funciona <Route>.

Como se puede leer en los documentos de React Router, el método recomendado para renderizar algo con una <Route> es usar elementos children, como se muestra arriba. Sin embargo, hay algunos otros métodos que puede usar para renderizar algo con una <Route>.

Estos se proporcionan principalmente para admitir aplicaciones que se crearon con versiones anteriores del enrutador antes de que se introdujeran los hooks:

- component: cuando la URL coincide, el enrutador crea un elemento React a partir del componente dado usando React.createElement.
- render: útil para renderizado en línea. El render prop espera una función que devuelva un elemento cuando la ubicación coincide con el path de la ruta.
- children: esto es similar a render, ya que espera una función que devuelva un componente React. Sin embargo, children se representan independientemente de si el path coincide con la ubicación o no.

Path y Match

El prop de path se utiliza para identificar la parte de la URL que el enrutador debe coincidir. Para ello, utiliza la biblioteca Path-to-RegExp para convertir una cadena de ruta en una expresión regular. Luego se comparará con la ubicación actual.

Si el path del enrutador y la ubicación coinciden correctamente, se crea un objeto que se denomina match Object, que contiene más información sobre la URL y el path. Esta información es accesible a través de sus props, que se enumeran a continuación:

- match.url: una cadena que devuelve la parte correspondiente de la URL. Esto es particularmente útil para construir componentes <Link> anidados.
- match.path: un string que devuelve la cadena del path, es decir, <Route path = "">. Usaremos esto para construir componentes <Route> anidados.
- match.isExact: un booleano que devuelve verdadero si la coincidencia fue exacta (sin caracteres finales).
- match.params: un objeto que contiene pares de key/value de la URL analizada (parsed) por el paquete Path-to-RegExp.

Pase implícito de props

Tenga en cuenta que cuando se utiliza el componente prop para representar una ruta, los `props` de `match`, `location` y `history` se pasan implícitamente al componente. Cuando se utiliza el patrón de representación de ruta más reciente, este no es el caso.

Por ejemplo, tome este componente:

```
// src/App.js
…

const Home = (props) => {
  console.log(props);

  return (
    <div>
      <h2>Home</h2>
    </div>
  );
};
```

Ahora, renderice la ruta de esta forma:

```
// src/App.js
…


      <Route exact path="/" component={Home} />
```

Esto registrará lo siguiente:

{history: {…}, location: {…}, match: {…}, staticContext: undefined}

Esto puede parecer una desventaja al principio, pero no lo es, ya que React v5.1 introdujo varios hooks para ayudarlo a acceder a lo que necesita, donde lo necesite. Estos hooks,

brindan nuevas formas de administrar el estado de nuestros enrutamientos y ayudan bastante a ordenar nuestros componentes.

"Consulte la siguiente página https://reacttraining.com/blog/react-router-v5-1/ para saber más al respecto"

El componente Switch

Cuando se utilizan varias <Route> juntas, todas las rutas que coinciden se representan de forma inclusiva. A continuación, se agregará una nueva ruta para demostrar por qué <Switch> es útil:

```
// src/App.js
…

      <Route exact path="/"><Home /></Route>
      <Route path="/category"><Category /></Route>
      <Route path="/products"><Products /></Route>
      <Route path="/:id">
        <p>This text will render for any route other than '/'</p>
      </Route>
```

Si la URL es /Products, se renderizan todas las rutas que coinciden con la ubicación /Products. Entonces, la <Route> con path /:id se renderiza junto con el componente < Products >. Esto es por diseño, sin embargo, si este no es el comportamiento que espera, debe agregar el componente <Switch> a sus rutas. Con <Switch>, solo se renderiza el primer hijo <Route> que coincide con la ubicación:

```
      <Switch>
        <Route exact path="/"><Home /></Route>
        <Route path="/categoria"><Categoria /></Route>
        <Route path="/productos"><Productos /></Route>
        <Route path="/:id">
          <p>This text will render for any route other than those
defined above</p>
        </Route>
      </Switch>
```

La parte :id del path se utiliza para el enrutamiento dinámico. Coincidirá con cualquier cosa después de la barra y hará que este valor esté disponible en el componente.

Ahora que sabemos todo sobre los componentes <Route> y <Switch>, agreguemos rutas anidadas a nuestra demostración.

Enrutamiento anidado dinámico

Anteriormente, creamos rutas para /, `/category` y `/Products`, sin embargo, si quisiéramos una URL en forma de `/category/shoes`?

Actualizando `src/App.js` de la siguiente manera:

```
// src/App.js
import React from "react";
import { Link, Route, Switch } from "react-router-dom";
import Category from "./Category";

const Home = () => (
  <div>
    <h2>Home</h2>
  </div>
);
const Products = () => (
  <div>
    <h2>Products</h2>
  </div>
);
export default function App() {
  return (
    <div>
      <nav className="navbar navbar-light">
        <ul className="nav navbar-nav">
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/category">Category</Link>
          </li>
          <li>
            <Link to="/products">Products</Link>
          </li>
        </ul>
      </nav>
      <Switch>
        <Route exact path="/">
          <Home />
        </Route>
        <Route path="/category">
          <Category />
        </Route>
```

```
        <Route path="/products">
          <Products />
        </Route>
      </Switch>
    </div>
  );
}
```

Notará que se ha movido `Category` a su propio componente. Aquí es donde deben ir nuestras rutas anidadas.

Creemos `Category.js` ahora:

```
// src/Category.js

import React from "react";
import { Link, Route, useParams, useRouteMatch } from "react-
router-dom";

const Item = () => {
  const { name } = useParams();

  return (
    <div>
      <h3>{name}</h3>
    </div>
  );
}

const Category = () => {
  const { url, path } = useRouteMatch();

  return (
    <div>
      <ul>
        <li>
          <Link to={`${url}/shoes`}>Shoes</Link>
        </li>
        <li>
          <Link to={`${url}/boots`}>Boots</Link>
        </li>
        <li>
          <Link to={`${url}/footwear`}>Footwear</Link>
        </li>
      </ul>
      <Route path={`${path}/:name`}>
```

```
        <Item />
      </Route>
    </div>
  );
};

export default Category;
```

Aquí, usamos el hook `useRouteMatch` para obtener acceso al `match object`. Como se mencionó anteriormente, `match.url` se utilizará para construir enlaces anidados y `match.path` para rutas anidadas. Si tiene problemas para comprender el concepto de `math`, `console.log (useRouteMatch ())` proporciona información útil que puede ayudar a aclararlo.

```
<Route path={`${path}/:name`}>
        <Item />
</Route>
```

Este es el primer intento adecuado de enrutamiento dinámico. En lugar de codificar las rutas, se usará una variable dentro de la `prop path` llamada `:name` que es un parámetro de `path` y captura todo después de `category/` hasta que se encuentra otra barra inclinada (/). Entonces, un nombre de ruta como `products/running-shoes` creará un `params object` de la siguiente manera:

```
{
    name: "running-shoes";
}
```

Para acceder a este valor dentro del componente `<Item>`, usamos el `hook useParams`, que devuelve un objeto de pares `key/value` de parámetros de URL.

Pruebe esto en su navegador. La sección de `Category` debería tener ahora tres subsecciones, cada una con su propia ruta.

Si presenta el siguiente ERROR, `[HMR] Waiting for update signal from WDS...` LA SOLUCIÓN:

https://stackoverflow.com/questions/59695102/reactjs-console-error-hmr-waiting-for-update-signal-from-wds

https://github.com/facebook/create-react-app/issues/8153

Enrutamiento anidado con parámetros de ruta

Un enrutador del mundo real tendrá que lidiar con datos y mostrarlos dinámicamente. Por ende, ahora vamos a incrustar el siguiente código a archivo `src/Products.js` y crear el `path` para las siguientes rutas:

- `/products`: debe mostrar una lista de productos.
- `/products/:productId`: si existe un producto con :`productId`, debería mostrar los datos del producto, y si no, debería mostrar un mensaje de error.

En consecuencia, creemos el nuevo archivo `src/Products.js` y agregue lo siguiente:

```
// src/Products.js

import React from "react";
import { Link, Route, useRouteMatch } from "react-router-dom";
import Product from "./Product";

const Products = ({ match }) => {
    const productData = [
        {
          id: 1,
          name: "NIKE Liteforce Blue Sneakers",
          description:
            "Lorem ipsum dolor sit amet, consectetur adipiscing el
it. Proin molestie.",
          status: "Available",
        },
        {
          id: 2,
          name: "Stylised Flip Flops and Slippers",
          description:
            "Mauris finibus, massa eu tempor volutpat, magna dolor
 euismod dolor.",
          status: "Out of Stock",
        },
        {
          id: 3,
          name: "ADIDAS Adispree Running Shoes",
          description:
            "Maecenas condimentum porttitor auctor. Maecenas viver
ra fringilla felis, eu pretium.",
          status: "Available",
        },
        {
```

```
          id: 4,
          name: "ADIDAS Mid Sneakers",
          description:
            "Ut hendrerit venenatis lacus, vel lacinia ipsum ferme
ntum vel. Cras.",
          status: "Out of Stock",
        },
      ];

  const { url } = useRouteMatch();

  /* Create an array of `<li>` items for each product */
  const linkList = productData.map((product) => {
    return (
      <li key={product.id}>
        <Link to={`${url}/${product.id}`}>{product.name}</Link>
      </li>
    );
  });

  return (
    <div>
      <div>
        <div>
          <h3>Products</h3>
          <ul>{linkList}</ul>
        </div>
      </div>

      <Route path={`${url}/:productId`}>
        <Product data={productData} />
      </Route>
      <Route exact path={url}>
        <p>Please select a product.</p>
      </Route>
    </div>
  );
};
export default Products;
```

Primero, usamos el hook `useRouteMatch` para tomar la URL del `match object`. Luego construimos una lista de componentes <`Links`> usando la propiedad `id` de cada uno de nuestros productos, que almacenamos en una variable `linkList`.

La primera ruta usa una variable en el prop `path` que corresponde a la del ID del producto. Cuando coincide, renderizamos el componente <Product> (que definirá pronto) y le pasamos nuestros datos de producto:

```
<Route path={`${url}/:productId`}>
        <Product data={productData} />
      </Route>
```

La segunda ruta tiene un prop `exatc`, por lo que solo se mostrará cuando la URL sea `/productos` y no se seleccione nada.

Ahora, aquí está el código para el componente <Product>. Deberá crear este archivo en: `src/Product.js`

```
// src/Product.js

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

const Product = ({ data }) => {
  const { productId } = useParams();
  const product = data.find(p => p.id === Number(productId));
  let productData;

  if (product) {
    productData = (
      <div>
        <h3> {product.name} </h3>
        <p>{product.description}</p>
        <hr />
        <h4>{product.status}</h4>
      </div>
    );
  } else {
    productData = <h2> Sorry. Product doesn't exist </h2>;
  }

  return (
    <div>
      <div>{productData}</div>
    </div>
  );
};

export default Product;
```

El método `find` se utiliza para buscar en la matriz un objeto con una propiedad de ID que sea igual a `match.params.productId`. Si el producto existe, se muestra `productData`. De lo contrario, se muestra el mensaje "El producto no existe".

Finalmente, actualice su componente <App> de la siguiente manera:
Elimine el `const Products` e importe `Products`, a saber:

```
import Products from "./Products";
```

Ahora, cuando visite la aplicación en el navegador y seleccione " `Products` ", verá un submenú renderizado, que a su vez muestra los datos del producto.

Protección de rutas

Un requisito común para muchas aplicaciones web modernas es asegurarse de que solo los usuarios que hayan iniciado sesión puedan acceder a ciertas partes del sitio. En la siguiente sección, veremos cómo implementar una ruta protegida, de modo que, si alguien intenta acceder a `/admin`, se le pedirá que inicie sesión.

Sin embargo, hay un par de aspectos de React Router que se debe cubrir primero.

El componente `Redirect`

Al igual que con las redirecciones del lado del servidor, el componente `Redirect` de React Router reemplazará la ubicación actual en la pila del `history` con una nueva ubicación. La nueva ubicación la especifica el `prop to`. Así es como usaremos `<Redirect>`:

```
<Redirect to={{pathname: '/login', state: { from: location }}}
```

Por lo tanto, si alguien intenta acceder a la ruta `/admin` mientras está desconectado, será redirigido a la ruta `/login`. La información sobre la ubicación actual se transmite a través del `prop state`, de modo que, si la autenticación es exitosa, el usuario puede ser redirigido a la página a la que originalmente intentaba acceder.

Rutas personalizadas

Una ruta personalizada es una forma elegante de describir una ruta anidada dentro de un componente. Si necesitamos tomar una decisión sobre si se debe renderizar una ruta o no, escribir una ruta personalizada es el camino a seguir.

Cree un nuevo archivo en `scr/PrivateRoute.js` y agregue el siguiente contenido:

```
// src/PrivateRoute.js

import React from "react";
import { Redirect, Route, useLocation } from "react-router-dom";
import { fakeAuth } from './Login';

const PrivateRoute = ({ component: Component, ...rest }) => {
  const location = useLocation();

  return (
    <Route {...rest}>
      {fakeAuth.isAuthenticated === true ?
        <Component />
      :
        <Redirect to={{ pathname: "/login", state: { from: locatio
n } }} />
      }
    </Route>
  );
};

export default PrivateRoute;
```

Como puede ver, en la definición de la función estamos desestructurando los accesorios que recibimos en un prop `Component` y en un prop `rest`. La propiedad `Component` contendrá cualquier componente que esté protegiendo nuestra `<PrivateRoute>` (en nuestro caso, `Admin`). El prop `rest` contendrá cualquier otro prop que se nos haya pasado.

Luego devolvemos un componente `<Route>`, que representa el componente protegido o nos redirige a nuestra ruta `/login`, dependiendo de si el usuario ha iniciado sesión o no. Esto se determina aquí por una propiedad `fakeAuth.isAuthenticated`, que se importa del componente `<Login>`.

Lo bueno de este enfoque es que evidentemente es más declarativo y `<PrivateRoute>` es reutilizable.

Aviso de seguridad importante

En una aplicación del mundo real, **debe validar cualquier solicitud de un recurso protegido en su servidor**. Esto se debe a que cualquier cosa que se ejecute en el cliente puede potencialmente someterse a ingeniería inversa y manipularse. Por ejemplo, en el

código anterior, uno puede simplemente abrir las herramientas de desarrollo de React y cambiar el valor de `isAuthenticated`, obteniendo así acceso al área protegida.

La autenticación en una aplicación React es digna de un tutorial propio, pero una forma de implementarla sería usando JSON Web Tokens. Por ejemplo, podría tener un punto final en su servidor que acepte una combinación de nombre de usuario y contraseña. Cuando los recibe (a través de Ajax), comprueba si las credenciales son válidas. Si es así, responde con un JWT, que la aplicación React guarda (por ejemplo, en `sessionStorage`), y si no, envía una respuesta `401 no autorizada` al cliente.

Suponiendo un inicio de sesión exitoso, el cliente enviaría el JWT como un encabezado junto con cualquier solicitud de un recurso protegido. Esto luego sería validado por el servidor antes de enviar una respuesta.

Al almacenar contraseñas, **el servidor no las almacenaría en texto sin formato**. Más bien, los cifraría, por ejemplo, usando bcryptjs.

Implementación de la ruta protegida

Ahora implementemos nuestra ruta protegida. Cambie `src/App.js` así:

```
// src/App.js

import React from "react";
import { Link, Route, Switch } from "react-router-dom";
import Category from "./Category";
import Products from "./Products";
import Login from './Login';
import PrivateRoute from "./PrivateRoute";

const Home = () => (
  <div>
    <h2>Home</h2>
  </div>
);

const Admin = () => (
  <div>
    <h2>Welcome admin!</h2>
  </div>
);

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

```
      <nav className="navbar navbar-light">
        <ul className="nav navbar-nav">
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/category">Category</Link>
          </li>
          <li>
            <Link to="/products">Products</Link>
          </li>
          <li>
            <Link to="/admin">Admin area</Link>
          </li>
        </ul>
      </nav>

      <Switch>
        <Route exact path="/"><Home /></Route>
        <Route path="/category"><Category /></Route>
        <Route path="/products"><Products /></Route>
        <Route path="/login"><Login /></Route>
        <PrivateRoute path="/admin" component={Admin} />
      </Switch>
    </div>
  );
};
```

Como puede ver, hemos agregado un componente `<Admin>` en la parte superior del archivo y estamos incluyendo nuestra `<PrivateRoute>` dentro del componente `<Switch>`. Como se mencionó anteriormente, esta ruta personalizada representa el componente `<Admin>` si el usuario está conectado. De lo contrario, el usuario es redirigido a `/login`.

Por último, cree un archivo llamado `Login.js` en la carpeta `/scr` y copie el código para el componente de inicio de sesión:

```
// src/Login.js

import React, { useState } from "react";
import { Redirect, useLocation } from "react-router-dom";

export default function Login() {
  const { state } = useLocation();
  const { from } = state || { from: { pathname: "/" } };
  const [redirectToReferrer, setRedirectToReferrer] = useState(fal
se);

  const login = () => {
    fakeAuth.authenticate(() => {
      setRedirectToReferrer(true);
    });
  };

  if (redirectToReferrer) {
    return <Redirect to={from} />;
  }

  return (
    <div>
      <p>You must log in to view the page at {from.pathname}</p>
      <button onClick={login}>Log in</button>
    </div>
  );
}

/* A fake authentication function */
export const fakeAuth = {
  isAuthenticated: false,
  authenticate(cb) {
    this.isAuthenticated = true;
    setTimeout(cb, 100);
  }
};
```

A estas alturas, es de esperar que no haya nada complicado aquí. Se utilizó el hook `useLocation` para acceder al prop `location` del enrutador, del cual tomamos la propiedad `state`. Luego usamos la desestructuración de objetos para obtener un valor

para la URL a la que el usuario intentaba acceder antes de que se le pidiera que iniciara sesión. Si no está presente, lo configuramos en `{pathname: "/"}`.

Luego usamos el hook `useState` de React para inicializar una propiedad `redirectToReferrer` a `false`. Dependiendo del valor de esta propiedad, el usuario es redirigido al lugar al que se dirigía (es decir, el usuario ha iniciado sesión) o se le presenta un botón para iniciar sesión.

Una vez que se hace clic en el botón, se ejecuta el método `fakeAuth.authenticate`, que establece `fakeAuth.isAuthenticated` en `true` y (en una función de devolución de llamada) actualiza el valor de `redirectToReferrer` en `true`. Esto hace que el componente vuelva a renderizarse y el usuario sea redirigido.