

React Router 2

Objetivos

- ☐ Conocer el componente NavLink Y la clase active
- ☐ Conocer el hook useNavigate()
- ☐ Conocer el hook useParams ()
- ☐ Proteger rutas

El componente NavLink

El componente NavLink es parte de la biblioteca React Router, que se utiliza para la navegación y la gestión de rutas en aplicaciones web de React. NavLink es una variante del componente Link y se utiliza específicamente para crear enlaces de navegación activos.

Cuando se utiliza NavLink, puedes definir un estilo o una clase CSS que se aplique automáticamente al enlace cuando la ruta actual coincide con la ruta especificada en el prop del componente NavLink. Esto es útil para resaltar visualmente el enlace activo en la barra de navegación de tu aplicación.


Nota: El prop to en un componente NavLink, se utiliza para especificar la ruta a la que debe dirigirse el enlace cuando se hace clic en él. Puedes pensar en el prop to como la dirección URL a la que apunta el enlace.

Usando el componente NavLink

Para usar NavLink simplemente debemos importarlo en el componente en el que lo vamos a usar:

```
import { NavLink } from "react-router-dom"; 8.6k (gzipped: 3.4k)
```

Una vez importado, podemos usarlo de la siguiente manera:

```
 <NavLink to="/home">Inicio</NavLink>
```

El componente NavLink tiene varias propiedades que podemos usar para personalizar su apariencia y comportamiento. Una de las propiedades más importantes es to, que especifica la ruta a la que navegará el enlace.

Otras propiedades útiles incluyen:

1. **activeClass**: La clase que se aplicará al enlace cuando el usuario se encuentre en la ruta correspondiente este nos devolverá true o false si la ruta está siendo consultada.
2. **exact**: Si esta propiedad se encuentra establecida en true, el enlace solo estará activo si la URL del navegador coincide exactamente con la ruta especificada en to.

La ActiveClass

La clase activa (activeClassName) en un componente NavLink de React Router es una propiedad que te permite especificar una clase CSS que se aplicará automáticamente al enlace cuando la ruta actual coincida con la ruta especificada en el prop to. Esta funcionalidad es útil para resaltar visualmente el enlace activo en la barra de navegación de tu aplicación.

Cuando el usuario navega a una ruta que coincide con la ruta especificada en el prop to, el componente NavLink aplicará la clase activa al enlace, lo que te permite darle un estilo diferente para indicar que está activo.

Para aplicar esta propiedad podemos hacer lo siguiente:

1. creamos una función esta toma la propiedad isActive, la cual retorna **true** cuando estamos en la ruta declarada en el componente NavLink y **false** cuando no lo estamos, luego validamos con un operador ternario y asignamos la clase que deseamos.

La función se debería ver de la siguiente manera:

```
const setActiveClass = ({ isActive }) => (isActive ? "text-warning mt-2 pe-2 text-decoration-none" : "text-white mt-2 pe-2 text-decoration-none");
```

En este caso estamos aplicando estilos de Bootstrap, pero también puedes usar CSS para dar el estilo que desees.

2. En el componente NavLink, procedemos a declarar en un className la función para que tome la clase que corresponda:

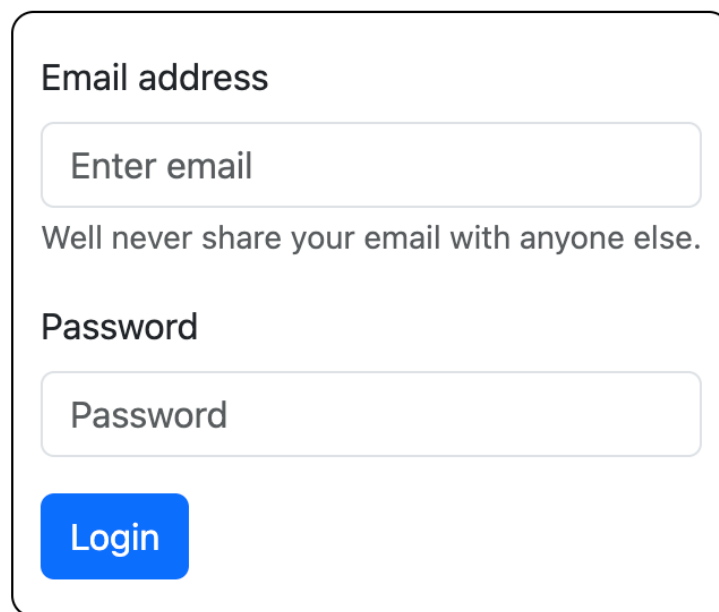
```
<NavLink  
  to="/about"  
  className={setActiveClass}  
>
```

El Hook UseNavigate

El hook useNavigate es un hook de React Router que permite navegar programáticamente dentro de tu app. Se introdujo en React Router v6 para reemplazar el hook useHistory.

Con navegar programáticamente nos referimos a redireccionar a nuestros usuarios a distintas partes de nuestra aplicación.

Para usar este hook debemos impórtalo de la misma manera que los hooks anteriores, pero antes crearemos un componente login para hacer una redirección si es que el usuario esta registrado en nuestro sistema.



Email address

Well never share your email with anyone else.

Password

Login

Una vez creado pasaremos a ver la lógica detrás de esto no sin antes crear la ruta para este componente.

1. En nuestro componente login importamos el hook useNavigate

```
import { useNavigate } from "react-router-dom"; 4.8k (gzipped: 2k)
```

2. Pasamos el hook a una variable esto necesario para obtener una referencia a la función `navigate()`, que se utiliza para navegar entre páginas así lo hacemos más legible, aunque también se puede usar sin esto.

```
const navigate = useNavigate();
```

3. Por último, usando `navigate` podemos redireccionar al usuario si inicio sesión con éxito, o redireccionarlo a otro componente en caso de que no esté registrado en nuestro proyecto.

```
27     if (email == "test@test.com" && password == "123456") {
28         Swal.fire({
29             title: "Success!",
30             text: "Bienvenido",
31             icon: "success",
32             timer: 1000,
33         });
34
35         navigate("/characters");
36     } else {
37         Swal.fire({
38             title: "Error!",
39             text: "Usuario no registrado seras derivado a la pagina de registro",
40             icon: "info",
41             timer: 1000,
42         });
43         navigate("/register");
44     }
```

Como logramos apreciar primero validamos que los datos ingresados coincidan, si están correctos redireccionamos al usuario a la ruta `characters` y si no es válido el usuario lo enviamos a la ruta de registro.

El hook `useParams`

El hook `useParams` es un hook de React Router que te permite acceder a los parámetros de la ruta actual desde un componente. Los parámetros de la ruta son los valores que se pasan a la ruta como parte de la URL. Por ejemplo, si tienes una ruta que se ve así:

```
22 <Route path="/profile/:id" element={<ProfileCharacter />} />
```

El parámetro `id` es un parámetro de la ruta. Cuando un usuario visita esta ruta, puede pasar un valor para el parámetro `id` como parte de la URL. Por ejemplo, si el usuario visita la ruta `/producto/1234`, el valor del parámetro `id` será `1234`.

El hook `useParams` te permite acceder al valor del parámetro de la ruta actual. Para hacerlo, simplemente importa el hook `useParams` y luego llámalo en el componente que quieras acceder a los parámetros de la ruta. El hook `useParams` devolverá un objeto con las claves y los valores de los parámetros de la ruta actual.

```
1 import { useParams } from "react-router-dom"; 484 (gzipped: 325)
2 import { useEffect, useState } from "react"; 4.2k (gzipped: 1.8k)
3 import axios from "axios"; 56.1k (gzipped: 20.5k)
4 import CardRick from "../CardRick";
5
6 const ProfileCharacter = () => {
7   const { id } = useParams();
8   const [character, setCharacter] = useState({});
9 }
```

El hook `useParams` es una herramienta útil para acceder a los parámetros de la ruta actual desde un componente. Si tu componente necesita acceder a los parámetros de la ruta, el hook `useParams` es una buena manera de hacerlo.

Aquí hay algunos ejemplos de cómo puedes utilizar el hook `useParams`:

- ☐ Lo puedes para mostrar información específica sobre un producto o servicio en función del valor del parámetro de la ruta.
- ☐ Para cargar datos específicos de la base de datos en función del valor del parámetro de la ruta.
- ☐ Para navegar a una nueva ruta en función del valor del parámetro de la ruta.

Protección de rutas en react

Las aplicaciones web construidas con React son muy versátiles y poderosas, pero también pueden presentar desafíos en términos de seguridad y control de acceso. Proteger las rutas en una aplicación React es un aspecto fundamental para garantizar la seguridad de los datos y la experiencia del usuario.

¿Qué significa proteger rutas en una aplicación React?

Proteger rutas en una aplicación React implica controlar quién tiene acceso a qué partes de la aplicación. Esto se logra asegurándose de que solo los usuarios autorizados puedan ver ciertas páginas o funcionalidades. La protección de rutas es esencial para aplicaciones que

manejan información sensible, requieren autenticación de usuarios o tienen diferentes roles de usuario con diferentes niveles de acceso.

Razones para proteger rutas en una aplicación React:

1. **Seguridad de Datos:** Algunas partes de la aplicación pueden contener información confidencial o privada, como perfiles de usuario, datos financieros o detalles de cuentas. Proteger estas rutas garantiza que solo los usuarios autorizados puedan acceder a esta información.
2. **Control de Acceso:** Las aplicaciones suelen tener varios roles de usuario, como usuarios normales, administradores, o editores. La protección de rutas permite establecer controles de acceso basados en roles para asegurarse de que cada usuario solo tenga acceso a las funcionalidades que le corresponden.
3. **Experiencia del Usuario:** Al ocultar rutas no autorizadas, se mejora la experiencia del usuario. Los usuarios no verán enlaces o botones que los lleven a páginas a las que no pueden acceder, lo que reduce la confusión y mejora la usabilidad.

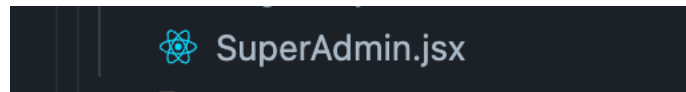
Cómo proteger rutas en una aplicación React:

1. **Configuración de Enrutamiento:** Utiliza una biblioteca de enrutamiento como React Router para definir y gestionar las rutas de tu aplicación.
2. **Autenticación:** Implementa un sistema de autenticación que verifique la identidad de los usuarios. Esto puede incluir inicio de sesión con contraseñas, autenticación basada en tokens, autenticación social, etc.
3. **Autorización:** Una vez autenticados, verifica los permisos del usuario para determinar si tienen acceso a una ruta específica. Esto puede basarse en roles de usuario, grupos de permisos o cualquier otro sistema de autorización que hayas configurado.
4. **Redireccionamiento:** Si un usuario intenta acceder a una ruta protegida sin autorización, redirígelos a una página de inicio de sesión o muestra un mensaje de error.

Ahora vamos al código, vamos a suponer que tenemos 2 tipos de usuarios un administrador y un usuario normal, además tendremos una vista de super admin a la cual solo podremos acceder si somos administradores y las vistas públicas a las cuales puede acceder cualquier usuario, por último, también crearemos un context, donde manejaremos el estado usuario a nivel global y así validar si este usuario es admin o no.

Vamos paso a paso a describir el cómo debemos proteger nuestra ruta super admin.

1. Primero crearemos nuestro componente super admin en la carpeta components



Nuestro componente super admin se utilizará para eliminar o editar usuarios quedando de la siguiente manera

```
import Table from "react-bootstrap/Table"; 2.6k (gzipped: 1.3k)
import users from "../../public/users.json";
import Button from "react-bootstrap/Button"; 4.1k (gzipped: 1.9k)
const SuperAdmin = () => {
  return (
    <>
      <Table striped bordered hover>
        <thead>
          <tr>
            <th>#</th>
            <th>Email</th>
            <th>Password</th>
            <th>Action</th>
          </tr>
        </thead>
        <tbody>
          {users.map((user) => {
            return (
              <tr key={user.id}>
                <td>{user.id}</td>
                <td>{user.email}</td>
                <td>{user.password}</td>
                <td>
                  <div className="d-flex justify-content-around">
                    <Button variant="danger">Danger</Button>
                    <Button variant="warning">Edit</Button>
                  </div>
                </td>
              </tr>
            );
          })}
        </tbody>
      </Table>
    </>
  );
};

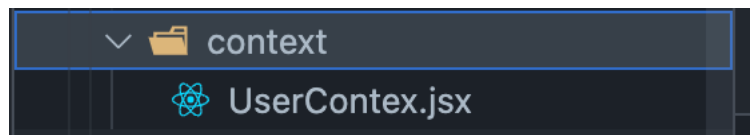
export default SuperAdmin;
```

React-Bootstrap Home About Login Superadmin			
#	Email	Password	Action
1	user1@example.com	password1	Danger Edit
2	user2@example.com	password2	Danger Edit
3	user3@example.com	password3	Danger Edit
4	user4@example.com	password4	Danger Edit
5	user5@example.com	password5	Danger Edit
6	user6@example.com	password6	Danger Edit
7	user7@example.com	password7	Danger Edit
8	user8@example.com	password8	Danger Edit
9	user9@example.com	password9	Danger Edit
10	user10@example.com	password10	Danger Edit

- En el componente App.jsx crearemos la ruta para nuestro super admin

```
27
28 <Route path="/super-admin" element={<SuperAdmin />} />
29
```

- Con nuestra ruta creada crearemos un context para el usuario en la carpeta del mismo nombre



```

1 import { createContext, useState } from "react"; 4.4k (gzipped: 1.9k)
2 export const UserContext = createContext();
3 const UserProvider = ({ children }) => {
4   const [user, setUser] = useState(null);
5   return (
6     <UserContext.Provider value={{ user, setUser }}>
7       {children}
8     </UserContext.Provider>
9   );
10 };
11 export default UserProvider;

```


4. Con nuestro context creado procedemos a envolver todos los componentes para poder validar sus rutas según el tipo de usuario esto lo podemos hacer en main.jsx.

```
1 import React from "react"; 6.9k (gzipped: 2.7k)
2 import ReactDOM from "react-dom/client"; 513 (gzipped: 319)
3 import App from "./App.jsx";
4 import "./index.css";
5 import "bootstrap/dist/css/bootstrap.min.css";
6 import { BrowserRouter } from "react-router-dom"; 5k (gzipped: 2.2k)
7 import UserProvider from "./context/UserContext.jsx";
8
9 ReactDOM.createRoot(document.getElementById("root")).render(
10   <BrowserRouter>
11     <React.StrictMode>
12       <UserProvider>
13         <App />
14       </UserProvider>
15     </React.StrictMode>
16   </BrowserRouter>
17 );
18
```

5. Ahora vamos a consumir nuestro context, en App.jsx para poder validar la presencia de este usuario con su tipo, de esta manera evaluaremos si es admin o no

```
1 import ProfileCharacter from "../components/ProfileCharacter";
2 import SuperAdmin from "../components/SuperAdmin";
3 import { useContext } from "react"; 4.1k (gzipped: 1.8k)
4
5 function App() {
6   const { user } = useContext(UserContext);
7   return (
8     <>
9
```

6. En la ruta que maneja el super admin vamos a realizar la siguiente validación usando el operador ternario quedando de la siguiente manera.

```
<Route
  path="/super-admin"
  element={user?.admin ? <SuperAdmin /> : <Navigate to="/login" />}
/>
```

Como podemos ver la validación se hace en el atributo element, que es donde indicamos que componente va a mostrar esa ruta, primero evalúa que exista el usuario, luego si es admin, si esto se cumple el usuario será redirigido a superAdmin, si no lo redireccionamos al componente login, esto también podría ser una alerta con un mensaje indicando que no tenemos permiso.

7. Por último, importaremos el context en el login con su variable de seteo, para guardar al usuario cuando inicie sesión, si bien los datos están pasados directamente es solo para recrear un registro de usuario.

```
7  import { useContext } from "../context/UserContext";
8  import { useState } from "react";
9
10 const Login = () => {
11   const { setUser } = useContext(UserContext);
12   const [email, setEmail] = useState("");
13   const [password, setPassword] = useState("");
14
15   const navigate = useNavigate();
16 }
```

```
if (email == "test@test.com" && password == "123456") {
  Swal.fire({
    title: "Success!",
    text: "Bienvenido",
    icon: "success",
    timer: 1000,
  });
  setUser({email: email, password: password, admin: true});
  navigate("/characters");
} else {
  Swal.fire({
    title: "Error!",
    text: "Usuario no registrado seras derivado a la pagina de registro",
    icon: "info",
    timer: 1000,
  });
  navigate("/register");
}
```

Como se puede ver en la validación del formulario, estamos guardando en el context un usuario con un email, password y un admin true, esto es lo que vamos a estar validando en la ruta para dar el acceso al usuario, al super admin o a las rutas que queramos proteger, Ahora puedes cambiar este dato a false y comprobar si puedes acceder al super admin.