

Cómo administrar la autenticación de usuarios con React JS

[#react#webdev#javascript#frontend](#)

La gestión del acceso de los usuarios es un aspecto fundamental del desarrollo de aplicaciones web. En esta guía completa, aprenderá y comprenderá el intrincado funcionamiento de la autorización y la autenticación en React.js. Este artículo está dirigido a lectores que poseen una comprensión fundamental de React, proporcionándoles información invaluable. Además, los ingenieros front-end experimentados descubrirán un flujo de trabajo detallado para la gestión de sesiones dentro del ámbito de esta guía.

Para empezar, tendrás que configurar un [proyecto de React](#) ejecutando el comando

```
npx create-react-app your-project-name
```

Además, puede ejecutar

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

Esto te ayudará a instalar [React Router Dom](#) para un sistema de enrutamiento eficiente.

Build Login Component

Comenzaremos con un componente de inicio de sesión simple que acepta dos entradas: un campo de correo electrónico y un campo de contraseña. Cuando se produce un evento onChange, estos dos campos de entrada desencadenan la función handleInput. La función también utiliza el atributo name de la entrada para establecer el estado de entrada. Este es un método eficaz para administrar campos de entrada en ReactJS. Si se presiona el botón, el elemento primario escucha un evento onSubmit e invoca el controlador handleSubmitEvent.

```
import { useState } from "react";
```

```
const Login = () => {  
  const [input, setInput] = useState({  
    username: "",  
    password: "",  
  });
```

```
  const handleSubmitEvent = (e) => {  
    e.preventDefault();  
    if (input.username !== "" && input.password !== "") {  
      //dispatch action from hooks  
    }  
    alert("please provide a valid input");  
  };
```

```
  const handleInput = (e) => {  
    const { name, value } = e.target;  
    setInput((prev) => ({  
      ...prev,  
      [name]: value,
```

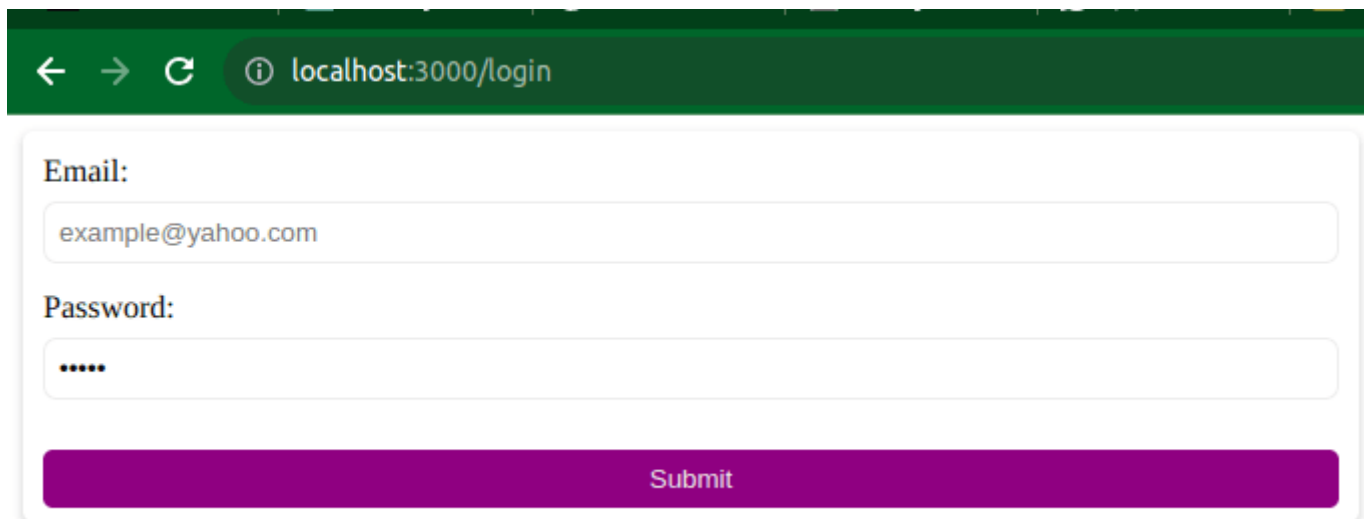
```

    });
};

return (
  <form onSubmit={handleSubmit}>
    <div className="form_control">
      <label htmlFor="user-email">Email:</label>
      <input
        type="email"
        id="user-email"
        name="email"
        placeholder="example@yahoo.com"
        aria-describedby="user-email"
        aria-invalid="false"
        onChange={handleInput}
      />
      <div id="user-email" className="sr-only">
        Please enter a valid username. It must contain at least 6 characters.
      </div>
    </div>
    <div className="form_control">
      <label htmlFor="password">Password:</label>
      <input
        type="password"
        id="password"
        name="password"
        aria-describedby="user-password"
        aria-invalid="false"
        onChange={handleInput}
      />
      <div id="user-password" className="sr-only">
        your password should be more than 6 character
      </div>
    </div>
    <button className="btn-submit">Submit</button>
  </form>
);
};

export default Login;

```



The image shows a web browser window with a dark green header bar. The address bar displays 'localhost:3000/login'. Below the header is a white login form. It contains two input fields: the first is labeled 'Email:' and contains the text 'example@yahoo.com'; the second is labeled 'Password:' and contains five dots. At the bottom of the form is a wide, purple button labeled 'Submit'.

Esta imagen muestra la salida de nuestro componente de inicio de sesión, que contiene dos campos de entrada y un botón.

Creación de AuthContext y AuthProvider

La [API de contexto](#) se usa generalmente para administrar los estados que se necesitarán en una aplicación. Por ejemplo, necesitamos nuestros datos de usuario o tokens que se devuelven como parte de la respuesta de inicio de sesión en los componentes del panel. Además, algunas partes de nuestra aplicación también necesitan datos de usuario, por lo que hacer uso de la API de contexto es más que resolver el problema para nosotros.

A continuación, en un archivo `AuthProvider.js`, cree un `AuthContext` para administrar el estado del usuario y un `AuthProvider` para consumir el contenido de nuestro contexto.

```
import { useContext, createContext } from "react";
const AuthContext = createContext();

const AuthProvider = ({ children }) => {
  return <AuthContext.Provider>{children}</AuthContext.Provider>;
};

export default AuthProvider;

export const useAuth = () => {
  return useContext(AuthContext);
};
```

El código configurado anteriormente se utiliza para crear el contexto de autenticación en React utilizando la API de contexto. Crea un `AuthContext` usando `createContext()` para administrar el estado de autenticación.

El componente `AuthProvider` está diseñado para encapsular la aplicación y proporcionar el contexto de autenticación a sus componentes secundarios mediante `AuthContext.Provider`.

El enlace personalizado [useAuth](#) utiliza `useContext` para acceder al contexto de autenticación desde dentro de los componentes, lo que les permite consumir el estado de autenticación y las funciones relacionadas almacenadas en el contexto. A medida que avance este artículo, agregaremos las lógicas de autenticación que controlan los procesos de inicio y cierre de sesión, pasándolas a través de

AuthProvider.Provider. A continuación, tendrá acceso a ellos y podrá utilizarlos cuando llame a la función useAuth.

A continuación, importe y ajuste AuthProvider alrededor del contenido de la aplicación en App.js.

```
import AuthProvider from "../hooks/AuthProvider";

function App() {
  return (
    <div className="App">
      <AuthProvider>{/* App content */}</AuthProvider>
    </div>
  );
}

export default App;
```

Ahora que el AuthProvider se ha envuelto alrededor de los componentes de la aplicación, podemos acceder a todos los valores de contexto en cualquiera de nuestras páginas o componentes dentro de la aplicación cuando agregamos todas las rutas necesarias al componente de la aplicación.

Creación de lógica de autenticación

A continuación, el componente AuthProvider se actualizará con funciones de inicio y cierre de sesión. Esta función se transmitirá a través de AuthContext.Provider y se podrá acceder a ella globalmente.

```
import { useContext, createContext, useState } from "react";
import { useNavigate } from "react-router-dom";

const AuthContext = createContext();

const AuthProvider = ({ children }) => {
  const [user, setUser] = useState(null);
  const [token, setToken] = useState(localStorage.getItem("site") || "");
  const navigate = useNavigate();
  const loginAction = async (data) => {
    try {
      const response = await fetch("your-api-endpoint/auth/login", {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify(data),
      });
      const res = await response.json();
      if (res.data) {
        setUser(res.data.user);
        setToken(res.token);
        localStorage.setItem("site", res.token);
        navigate("/dashboard");
        return;
      }
      throw new Error(res.message);
    } catch (err) {
```

```

    console.error(err);
  }
};

const logOut = () => {
  setUser(null);
  setToken("");
  localStorage.removeItem("site");
  navigate("/login");
};

return (
  <AuthContext.Provider value={{ token, user, loginAction, logOut }}>
    {children}
  </AuthContext.Provider>
);
};

export default AuthProvider;

export const useAuth = () => {
  return useContext(AuthContext);
};

```

En este código, el componente `AuthProvider` administra el estado de autenticación del usuario, proporcionando funcionalidades como inicio de sesión, cierre de sesión y almacenamiento de tokens mediante [enlaces useState](#).

La función `loginAction` controla el inicio de sesión del usuario mediante el envío de una [solicitud POST](#) a un punto de conexión de autenticación, la actualización del usuario y el estado del token tras una respuesta correcta y el almacenamiento del token en el [almacenamiento local](#).

La función `logOut` borra los datos del usuario y del token, quitando el token del almacenamiento local. `AuthContext.Provider` hace que el estado de autenticación y las funciones relacionadas estén disponibles para sus componentes secundarios, accesibles a través del enlace `useAuth`, lo que permite que los componentes consuman datos y acciones de autenticación dentro de la aplicación.

Proteja las rutas con autorización

A continuación, configuraremos un protector de ruta que proteja cualquier ruta que sea privada en la aplicación. Para lograr esto, se necesitará el gancho `useAuth` para acceder a nuestros datos de contexto. Aquí está el código sobre cómo trabajar a través de él.

```

import React from "react";
import { Navigate, Outlet } from "react-router-dom";
import { useAuth } from "../hooks/AuthProvider";

const PrivateRoute = () => {
  const user = useAuth();
  if (!user.token) return <Navigate to="/login" />;
  return <Outlet />;
};

```

```
export default PrivateRoute;
```

Este código define un componente `PrivateRoute` para controlar la autenticación. Utiliza el gancho `useAuth` de `AuthProvider` para acceder a los datos de autenticación de usuario. Si el usuario no posee un token, lo que indica que no ha iniciado sesión, el código desencadena una redirección a la ruta `/login` mediante el componente `<Navigate>`. De lo contrario, representa los componentes secundarios anidados dentro del componente `PrivateRoute` al que se accede a través de `<Outlet />`, lo que permite a los usuarios autenticados acceder a las rutas protegidas mientras redirige a los usuarios no autenticados a la página de inicio de sesión.

Agregar enrutamiento

A continuación, actualizaremos el componente `App.js` agregándole enrutamiento. Sirve como componente raíz, encerrando toda la aplicación. Dentro del `App.js`, utiliza el componente `Router` para configurar el mecanismo de enrutamiento.

```
import { BrowserRouter as Router, Route, Routes } from "react-router-dom";
importar Login desde "./components/Login";
importar el panel de control desde "./components/Dashboard";
importar AuthProvider desde "./hooks/AuthProvider";
importar PrivateRoute desde "./router/route";
```

```
function App() {
  return (
    <div className="App">
      <Router>
        <AuthProvider>
          <Routes>
            <Route path="/login" element={<Login />} />
            <Route element={<PrivateRoute />}>
              <Route path="/dashboard" element={<Dashboard />} />
            </Route>
            { /* Other routes */ }
          </Routes>
        </AuthProvider>
      </Router>
    </div>
  );
}
```

```
export default App;
```

El componente `Routes` establece la configuración de la ruta: la ruta `'/login'` se asigna al componente `Login` y se representa cuando la URL coincide. El componente `<PrivateRoute />` sirve como protección para proteger la ruta `/dashboard`. Cuando un usuario navega a `/dashboard`, `PrivateRoute` comprueba la autenticación mediante `AuthProvider`. Si el usuario está autenticado (tiene un token), representa el componente `Dashboar`; de lo contrario, redirige a la ruta `/login`, lo que garantiza un acceso protegido al panel.

Integración de API

Si has llegado a este punto, ¡buen trabajo! Nos acercamos al final de este artículo. Sin embargo, hay algunos pasos más que cubrir. Hemos creado correctamente un componente de inicio de sesión,

hemos establecido un AuthContext para administrar las sesiones de usuario y hemos configurado una protección de rutas. Ahora, el siguiente paso es activar la acción de inicio de sesión utilizando el gancho useAuth para acceder a la función. Este enfoque permite que la función handleSubmitEvent del componente de inicio de sesión active la solicitud de API. Tras una respuesta correcta, los tokens y los datos de usuario se guardarán y pasarán a través de AuthContext.

Este es el código actualizado para el componente de inicio de sesión.

```
import { useState } from "react";
import { useAuth } from "../hooks/AuthProvider";

const Login = () => {
  const [input, setInput] = useState({
    username: "",
    password: "",
  });

  const auth = useAuth();
  const handleSubmitEvent = (e) => {
    e.preventDefault();
    if (input.username !== "" && input.password !== "") {
      auth.loginAction(input);
      return;
    }
    alert("please provide a valid input");
  };

  return (
    <form onSubmit={handleSubmitEvent}>
      { /* Form inputs are provided in the above examples */ }
    </form>
  );
};

export default Login;
```

En el ejemplo anterior se muestra cómo distribuir loginAction mediante el enlace useAuth.

Botón Agregar cierre de sesión

A continuación, debemos agregar un botón para enviar la acción logOut para finalizar la sesión de usuario borrando el estado del usuario en el contexto y también aclarando el token localStorage. Ahora cree un componente de panel y agregue el código a continuación.

```
import React, { useEffect } from "react";
import { useAuth } from "../hooks/AuthProvider";

const Dashboard = () => {
  const auth = useAuth();
  return (
    <div className="container">
      <div>
        <h1>Welcome! {auth.user?.username}</h1>
        <button onClick={() => auth.logOut()} className="btn-submit">
```

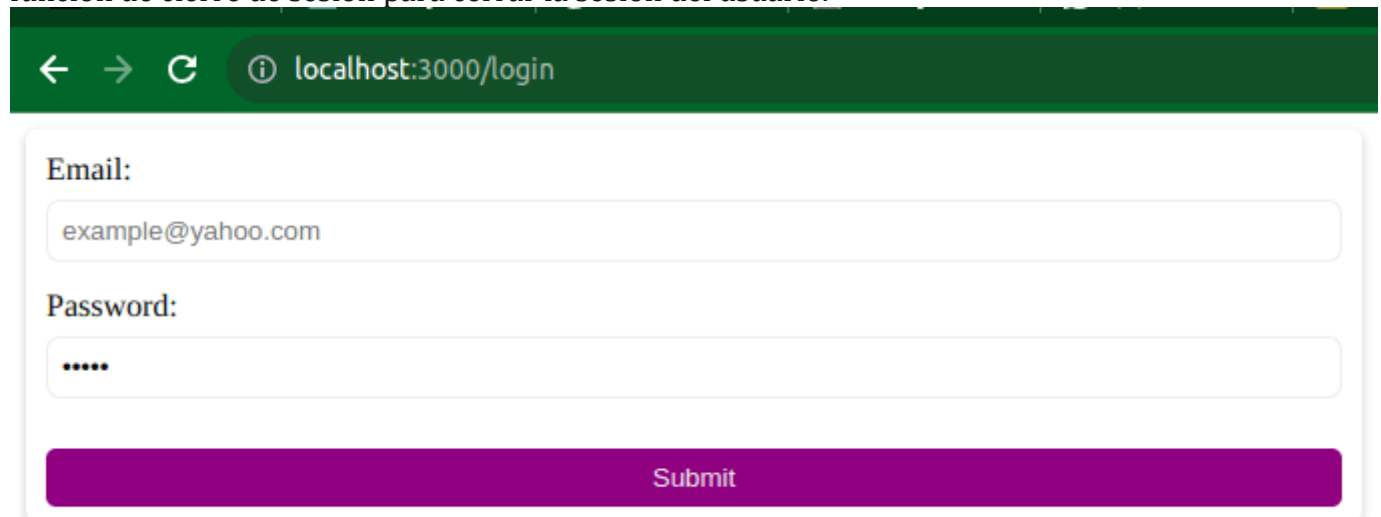
```

    logout
  </button>
</div>
</div>
);
};

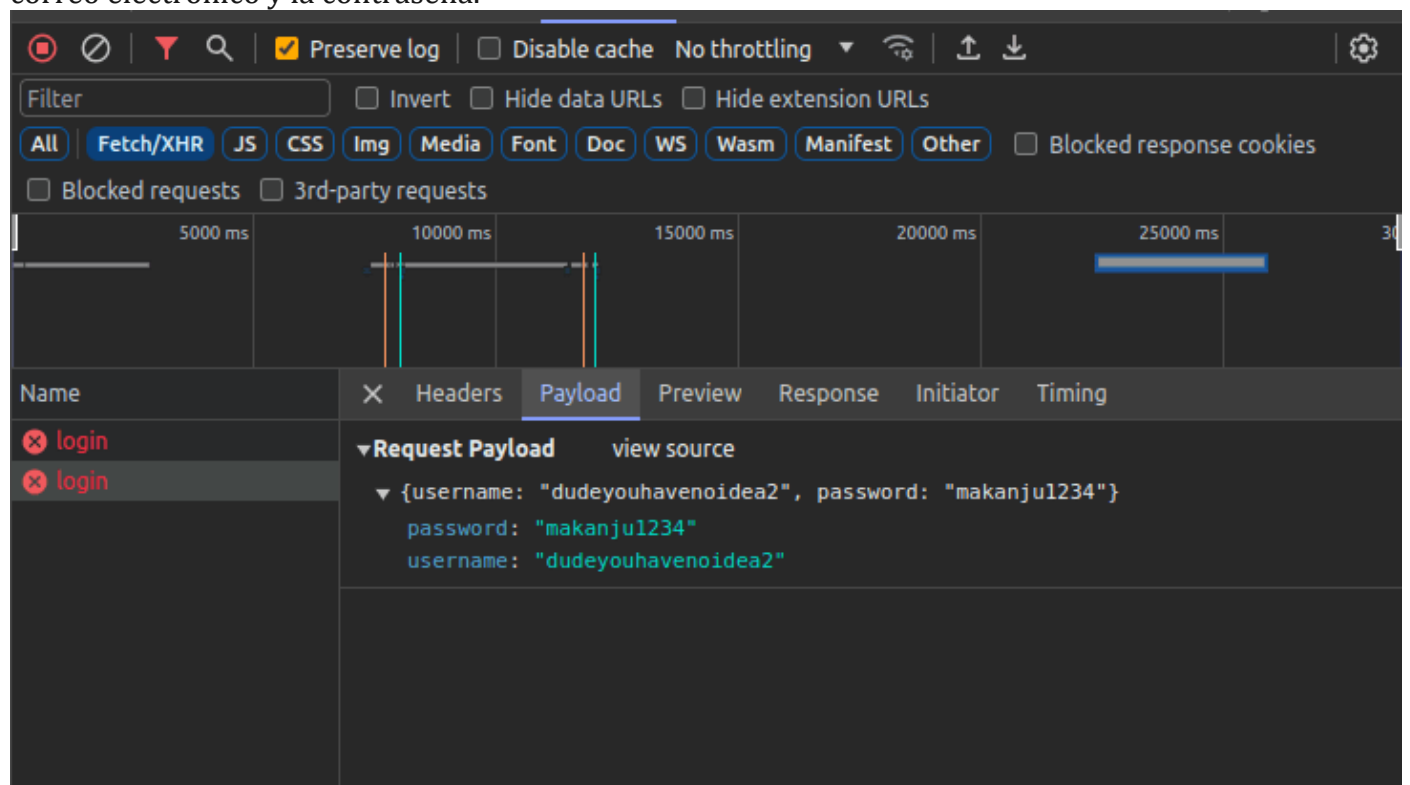
```

```
export default Dashboard;
```

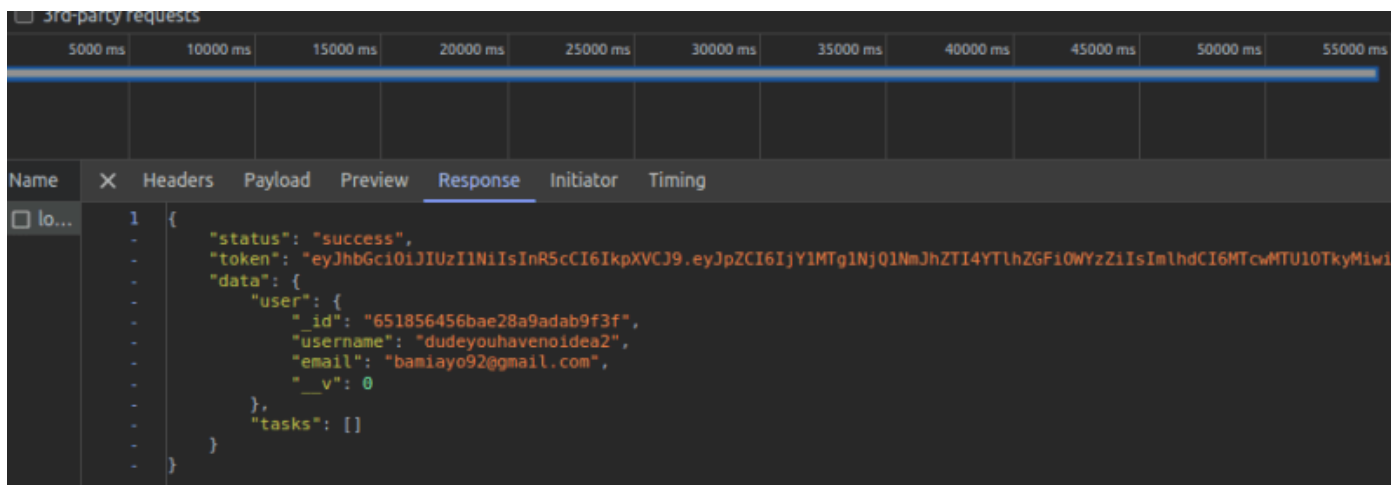
En el código anterior, el componente Dashboard utiliza el enlace useAuth de theAuthProvider para acceder a la información de autenticación. El componente muestra un mensaje de bienvenida con el nombre de usuario del usuario que ha iniciado sesión y un botón de cierre de sesión, que activa la función de cierre de sesión para cerrar la sesión del usuario.



La imagen de arriba muestra dos campos de entrada y un botón. El usuario introduce la dirección de correo electrónico y la contraseña.



Esta imagen ilustra los datos de entrada que se pasan como un objeto al backend después de que un usuario ingresa sus detalles.



Esta imagen muestra los datos de respuesta de la API de back-end, que incluyen datos de usuario, tareas de usuario y un token para administrar sesiones de usuario.

Welcome! dudeyouhavenoidea2

logout

Después de iniciar sesión correctamente, el usuario es redirigido al panel de control, donde se muestran los botones de nombre de usuario y cierre de sesión.

Las imágenes proporcionadas ilustran el proceso de inicio de sesión de nuestra implementación. Cuando un usuario ingresa la información correcta, la envía al servidor como una carga útil y el servidor verifica si el usuario sale. Si lo hacen, recibimos una respuesta con un objeto de usuario que contiene toda la información del usuario, como el nombre y la dirección de correo electrónico, y se le redirige a la página del panel. Además, recibimos un token, que se guarda en el almacenamiento local del navegador. Esto nos permite gestionar las sesiones de usuario de forma más eficaz. Cuando el usuario hace clic en la acción de cierre de sesión, se produce la última acción. Borra el almacenamiento local del usuario y lo redirige a la página de inicio de sesión.

Conclusión

Comprender cómo utilizar la API de contexto y React Router en los flujos de trabajo de autenticación es fundamental para administrar las sesiones de usuario en las aplicaciones de React. Desde la creación de contextos de autenticación hasta la creación de rutas protegidas y la habilitación de acciones de usuario como iniciar y cerrar sesión, esta guía completa equipa a los desarrolladores para manejar la autenticación de usuarios sin problemas y de forma segura dentro de sus proyectos de React.