2021

ReactJS

CARLOS EDUARDO ARISMENDI SÁNCHEZ

Contenido

ReactJS: Demo tutorial básico

Introducción

En este tutorial veremos brevemente la sintáxis de ReactJS, los hooks, cómo crear componentes, rutas públicas, rutas protegidas, listados y cómo consumir API's.

Instalación

Para poder desarrollar en ReactJS, es necesario contar con:

- NodeJS: https://nodejs.org/es/
- Un IDE de nuestro agrado.

Crear proyecto

Para este paso, abriremos la consola de comandos de nuestro sistema y nos situaremos en el directorio en el cual queremos crear el proyecto. Una vez ahí, utilizamos el siguiente comando para crear el proyecto: *npx create-react-app myAppName*

Inicio rápido

Una vez creado el proyecto, desde la consola nos situamos en este y ejecutamos el comando: *npm start*. Esto iniciará el proyecto y abrirá una pestaña en nuestro navegador por defecto.

Plugins

Para darle estilos a la aplicación se ha utilizado el conocido framework Bootstrap. Además, se ha empleado Axios para realizar las llamadas a las API's.

Axios

Para instalarlo, hay que ejecutar el siguiente comando desde el directorio raíz de nuestro proyecto: *npm install axios*. Una vez instalado, es importante configurarlo para indicarle cuál es la ruta base del API que va a consumir y los headers necesarios. Adicionalmente, podemos definir un interceptor para las respuestas de Axios al consumir un API, lo que nos permitirá personalizar la respuesta que verá el usuario.

```
import axios from 'axios'
import UTILS from '../utils'

axios.defaults.baseURL = process.env.REACT_APP_API_URL
axios.defaults.headers.common['Access-Control-Allow-Origin'] = process.env.REACT_APP_API_URL

// Add a response interceptor
axios.interceptors.response.use((response) => {
  return response
}, function (error) {
  if (error.response.status === 401) {
    UTILS.user.removeLogin()
  } else {
    return Promise.reject(error)
  }
}
)

export default axios
```

En la anterior imagen, vemos la importación de Axios y UTILS (lo veremos más adelante). Posteriormente, le indicamos a Axios la ruta basde del API y el header '*Access-Control-Allow-Origin'*, el cual sirve para poder acceder a un API con CORS configurado. Finalmente, aparece un interceptor, el cual, al detectar el código de error http 401, llama a un método de UTILS que borra todos los posibles datos que pueda tener el usuario en el localStorage del navegador.

Una vez tenemos esta configuración, en próximos componentes de nuestra aplicación que necesiten utilizar Axios, en lugar de importarlo por defecto, importarán este archivo con la configuración.

Bootstrap

De forma similar a Axios, desde el directorio raíz de nuestro proyecto, ejecutamos: *npm install react-bootstrap bootstrap*. Para poder utilizarlo, debemos realizar la siguiente importación en App.js o index.js:

```
import 'bootstrap/dist/css/bootstrap.min.css';
```

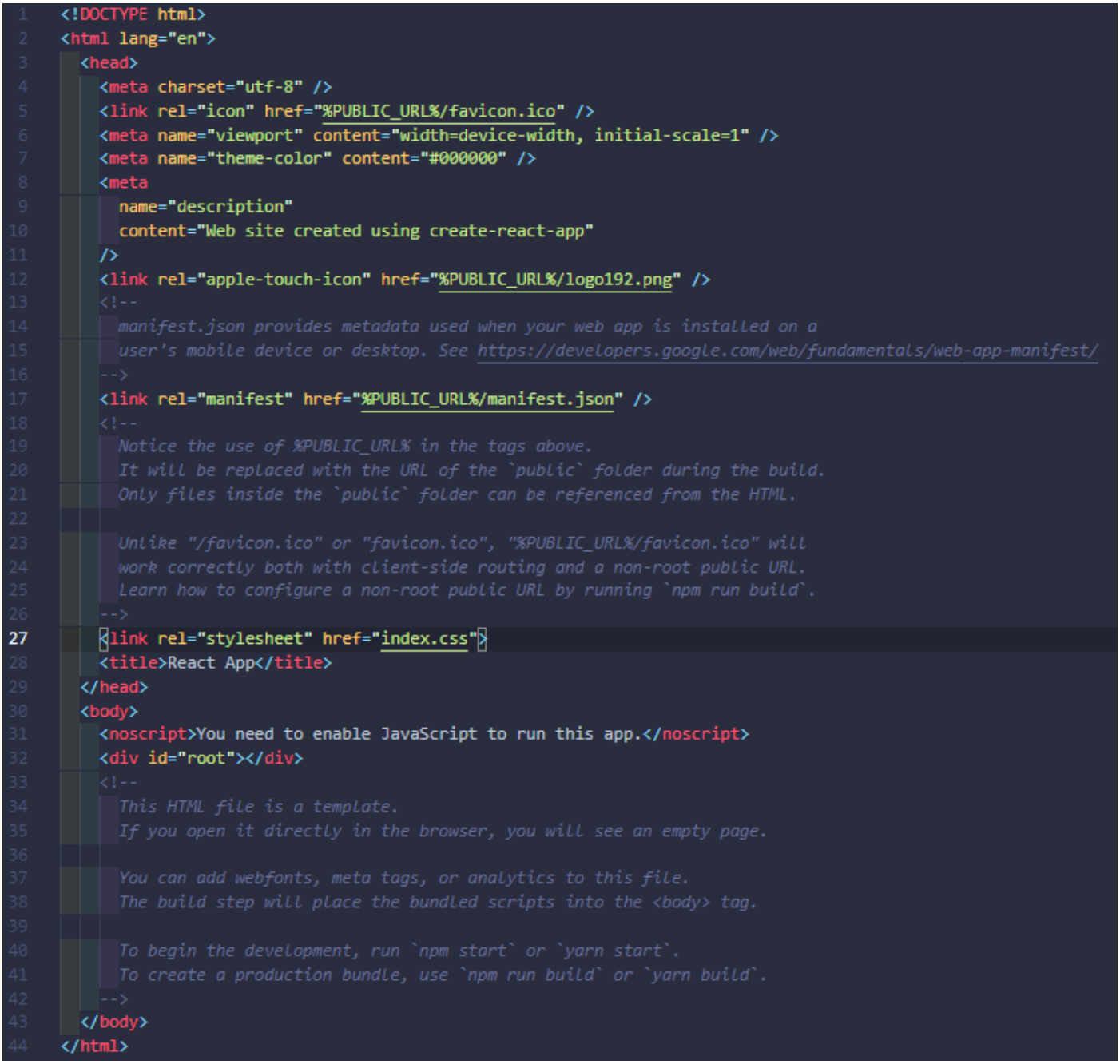
Esto nos incluirá de forma global a toda la aplicación, los estilos de Boostrap. Sin embargo, si queremos hacer uso de sus componentes, deberemos importar manualmente cada componente donde sea que lo vayamos a usar.

Utils

Implementa una serie de funciones para abstraer el almacenamiento y acceso de datos del usuario en el localStorage del navegador. Adicionalmente, importa el plugin Axios explicado anteriormente y configura el header '*Authorization*' con un jwt-token.

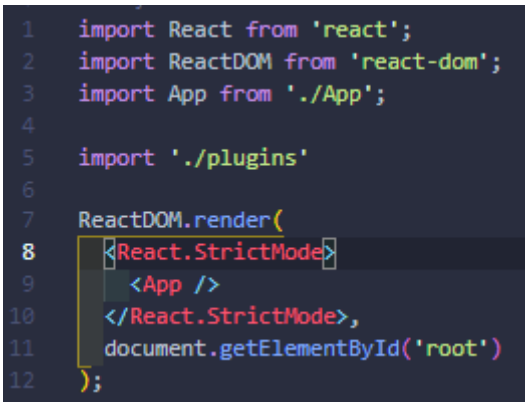
```
import axios from './plugins/axios'

const user = {
  setLogin: (token, idUser) => {
    localStorage.setItem('jwt-token', token);
    axios.defaults.headers.common['Authorization'] = `Bearer ${token}`

    localStorage.setItem('idUser', idUser)
  },
  removeLogin: () => {
    localStorage.removeItem('idUser')
    localStorage.removeItem('jwt-token')
    localStorage.removeItem('name')
    localStorage.removeItem('nickname')
    localStorage.removeItem('profileImage')
    axios.defaults.headers.common['Authorization'] = null
  },
  isAuthenticated: () => {
    let jwtToken = localStorage.getItem('jwt-token');

    if (jwtToken) {
      axios.defaults.headers.common['Authorization'] = `Bearer ${jwtToken}`
      return true
    }

    return false
  },
  getIdUser: () => {
    return localStorage.getItem('idUser')
  },
  getProfileImage: () => {
    return localStorage.getItem('profileImage')
  },
  getNickname: () => {
    return localStorage.getItem('nickname')
  },
  getName: () => {
    return localStorage.getItem('name')
  },
  setProfileImage: (image) => {
    localStorage.setItem('profileImage', image)
  },
  setNickname: (nickname) => {
    localStorage.setItem('nickname', nickname)
  },
  setName: (name) => {
    localStorage.setItem('name', name)
  }
}

const UTILS = {
  user: user
}

export default UTILS
```

index.js e index.html



```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Web site created using create-react-app"
    />
    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
    <!--
      manifest.json provides metadata used when your web app is installed on a
      user's mobile device or desktop. See https://developers.google.com/web/fundamentals/web-app-manifest/
    -->
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <!--
      Notice the use of %PUBLIC_URL% in the tags above.
      It will be replaced with the URL of the `public` folder during the build.
      Only files inside the `public` folder can be referenced from the HTML.

      Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC_URL%/favicon.ico" will
      work correctly both with client-side routing and a non-root public URL.
      Learn how to configure a non-root public URL by running `npm run build`.
    -->
    <link rel="stylesheet" href="index.css">
    <title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
    <!--
      This HTML file is a template.
      If you open it directly in the browser, you will see an empty page.

      You can add webfonts, meta tags, or analytics to this file.
      The build step will place the bundled scripts into the <body> tag.

      To begin the development, run `npm start` or `yarn start`.
      To create a production bundle, use `npm run build` or `yarn build`.
    -->
  </body>
</html>
```

En la anterior imagen vemos el index.html que se crea por defecto al crear una aplicación de ReactJS. Es importante destacar la línea 32: * *, pues ese contenedor es donde irá nuestra aplicación.



Este es el index.js, el archivo que se encarga de renderizar toda nuestra aplicación de ReactJS en el contenedor con id=”root” del html.

Componentes

Para crear un componente, en la carpeta /src/ podemos crear directamente un archivo Nombrecomponente.jsx, aunque recomiendo crear primero una carpeta /components/ y situar ahí nuestros componentes, para tener mejor organizado el proyecto.

PostFull

Este componente representa un post, similar a los que vemos en la página de inicio de Instagram.

```jsx
import React from 'react'
import { Card, Image } from 'react-bootstrap';

const PostFull = (props) => {
  const post = props.post

  const avatar = () => {
    const img = post.profileImage ? post.profileImage : '/logo192.png'
    return img
  }

  return (
    <div className="post-full text-center shadow-sm rounded m-auto">
      <Card>
        <div id="post-image" className="shadow-sm post-img">
          <Image alt="Vue logo" src={ post.image } fluid rounded/>
        </div>

        <div className="text-left py-1 px-2">
          <div id="post-creator">
            <div className="btn m-0 p-0">
              <Image className="avatar bg-dark" src={ avatar() } fluid roundedCircle />
              <span className="ml-1">@{ post.nickname }</span>
            </div>
          </div>

          <div id="post-text" className="mt-1">
            { post.description }
          </div>
        </div>
      </Card>
    </div>
  );
}

export default PostFull;
```

Podemos dividirlo en grandes bloques:

- Importaciones, primeras dos líneas.
- Definición e implementación del componente: desde la línea 4 hasta la 34.
- Exportar el componente para que pueda ser utilizado: línea 36.

Definición e implementación

El componente se define dentro de una función que puede recibir parámetros (props), la cual tendrá el nombre de nuestro componente. Dentro de esta función implementaremos todas las funciones que necesitemos, así como variables, hooks, etc., y deberá tener un return, el cual devolverá el jsx (html+Javascript) del componente que será renderizado.

Vemos que este componente espera recibir en el parámetro props, un objeto post con las variables image, nickname, description y profileImage. Para renderizar estas variables en el componente, debemos indicarlo entre el html mediante las llaves { variable }. Dentro de estas llaves, podremos poner también otro código javascript que consideremos necesario.

Es importante mencionar que para las clases CSS, debemos utilizar la palabra reservada className, en lugar de sólo class, como haríamos con Vue.js o html normal. Esto se debe a que ReactJS utiliza la palabra class para las clases de objetos de javascript.

ListPostFull

Normalmente, en las aplicaciones que creemos con ReactJS, tendremos múltiples componente, algunos de los cuales, utilizando otros componentes, por ello, en este ejemplo, veremos como hacer uso del componente PostFull que implementamos anteriormente:

```
1  import React from 'react'
2  import PostFull from '../posts/PostFull';
3
4  const ListPostsFull = (props) => {
5    const posts = props.posts
6    return (
7      <div className="container">
8        <div className="row">
9          <div className="col-sm-12 col-md-8 col-lg-6 m-auto">
10           {
11             posts.map(post =>
12               <div key={ post.idPost } className="mt-3">
13                 <PostFull post={ post } className="mt-4 rounded"/>
14               </div>
15             )
16           }
17         </div>
18       </div>
19     </div>
20   );
21 }
22
23 export default ListPostsFull;
```

Este componente, también recibe props, en este caso un array de objetos post.

Como usar otros componentes

En primer lugar, debemos importarlo como cualquier archivo, plugin o componente que importaríamos normalmente (línea 2) y, luego, para utilizarlo, simplemente debemos usarlo como una etiqueta html común:

<PostFull /> o <PostFull></PostFull> indistintamente, ya que en este caso son equivalentes porque no se espera código adicional dentro de las etiquetas del componente.

Ahora nos surge la siguiente pregunta: ¿cómo le paso el parámetro post que necesita este? La respuesta es bien sencilla: como vemos en la línea 13, se lo pasamos como un parámetro común de la etiqueta html <PostFull post={ datoQuePasamos } />. Es importante destacar que al utilizar variables/objetos javascript sobre el html, no utilizamos la notación normal de html con dobles comillas (<etiqueta variable="valor" />) sino que en su lugar, utilizamos las llaves (<etiqueta variable={ valor } />). Del mismo modo, si el componente utilizado recibiera múltiples parámetros en sus props, se pasarían todos y cada uno de forma independiente del mismo modo: (<etiqueta variable1="valor1" variable2="valor2" variable3="valo3" />).

Listado

Nos encontraremos también ocasiones como esta, en que de forma dinámica, necesitemos renderizar listados que a priori, no sabemos cuantos elementos tendremos. Para ello, tendremos todos los elementos en un array de javascript, y como vimos anteriormente, entre el html, con las llaves { // código javscript }, mostraremos con un bucle, función o como prefiramos -en el ejemplo mediante la función map del array-, los componentes que necesitemos listar (líneas de la 10 a 15).

Views, hooks, llamadas a API

A medida que nuestra aplicación va creciendo, tendremos más y más componentes, y también tendremos más y más páginas, las cuales llamaremos views y utilizan generalmente, un conjunto de componentes, variables, hooks y realizan llamadas a API's.

Hooks: useState

No son más que una forma de definir variables en React para mantener y usar el estado y otras características de React sin necesidad de escribir una clase. Estos son completamente opcionales, compatibles con versiones algo más antiguas de React y son más fáciles y cómodos de usar que clases , ya que resultan en código más limpio y entendible para los desarrolladores.

Para utilizarlos, es necesario primero hacer la importación de useState:

```
import React, { useState, useEffect } from 'react'
```

El modo de uso es similar a como trabajaríamos con una clase y sus variables, es decir, definimos la variable y la modificamos mediante un setter:

const [miVariable, setMiVariable] = useState(valorInicialDeMiVariable)

El código anterior es solo la inicialización, pero puede que necesitemos modificarla. Para ello, es tan sencillo como:

setMiVariable(nuevoValorDeMiVariable).

Es importante destacar que el anterior ejemplo sólo nos sirve si nuestra variable es de tipo primitivo (int, boolean…) o si es un objeto o array y queremos modificarlo al completo. Pero como todos sabemos, a veces necesitamos añadir un elemento nuevo a un array o modificar solo una propiedad de un objeto, por lo que surge la pregunta: ¿cómo lo haríamos?

Home.jsx

Utilizaremos un ejemplo real para ver los hooks en acciones, y resolver la pregunta final del apartado anterior:

```
1  import { Alert } from 'react-bootstrap'
2  import React, { useState, useEffect } from 'react'
3  import ListPostsFull from '../components/listposts/ListPostsFull'
4  import axios from '../plugins/axios'
5  import NavBarPrivate from '../components/navbars/NavBarPrivate'
6
7  const Home = () => {
8    const [postsError, setPostsError] = useState({ errorMessage: '', showError: false })
9    const [posts, setPosts] = useState([])
10   const [pagination, setPagination] = useState({
11     prev: null,
12     next: null,
13     limit: null,
14     count: 0,
15     totalRegs: 0,
16     nextPageBtn: true
17   })
```

Primero tenemos las diferentes importaciones y pasamos a la implementación de la vista Home (la cual sigue la misma estructura de un componente).

Hooks

En las líneas 8, 9 y 10-17, vemos la definición e inicialización de 3 hooks:

- postError: que almacenará un objeto con dos variables que indican si hubo un error en la llamada al API (la veremos luego).
- Posts: array de los posts que recibiremos del API y que renderizaremos con ayuda del componente ListPostFull explicado previamente.
- Pagination: objeto que almacena datos relacionados a la paginación de los posts.

Llamada al API

```
const getAllPosts = async () => {
  const params = {
    prev: pagination.prev,
    next: pagination.next,
    limit: pagination.limit
  }

  axios.get('posts/all', { params })
    .then(res => {
      if (res.data.status < 0) {
        setPostsError({
          ...postsError,
          errorMessage : res.data.error,
          showError : true
        })
      } else {
        setPosts([ ...posts, ...res.data.data.posts])
        setPagination({
          ...pagination,
          prev : res.data.data.prev,
          next : res.data.data.next,
          count : res.data.data.count,
          totalRegs : res.data.data.totalRegs,
          nextPageBtn : (res.data.data.count >= pagination.limit)
        })
      }
    })
    .catch(e => {
      console.log(e)
      setPostsError({
        ...postsError,
        errorMessage : e,
        showError : true
      })
    })
}
```

Para consumir el API, utilizamos el plugin Axios que detallamos en el apartado de Plugins. En concreto, utilizamos su método get, el cual recibe dos parámetros: la ruta (la cual se añade al final de la ruta base que tiene configurada) y los parámetros de la ruta. Este devolverá una promesa que debemos resolver, y si todo va según los previsto, ejecutaremos el .then. Aquí vemos como modificamos los hooks:

- Posts: queremos concatenar los posts que tuviéramos antes de la llamada al API con los nuevos que me devuelve el API, por eso, usamos el operador de propagación "…" sobre ambas variables y guardamos sus resultados en un array que luego será asignado al hook posts:

```
setPosts([ ...posts, ...res.data.data.posts])
```

  El operador de propagación se puede utilizar tanto en objetos como en arrays y lo que hace, a nivel conceptual es sacar todas sus variables o valores y ponerlas en un array en este caso de forma independiente. Así pues, si el valor de posts era [objetoPost1, objetoPost2, objetoPostn], con el operador de propagación, extraemos cada objetoPost para tenerlo de forma independiente, fuera del

array, todos a la vez en una única operación. Este mismo resultado podríamos obtenerlo haciendo uso de la función map del array o con una función que hagamos nosotros mismo con un bucle por ejemplo.

- Pagination y PostError: aquí, hacemos uso del mismo operador, y la documentación de React nos dice que para modificar una propiedad específica del objeto debemos hacer uso de la notación:

nombreVariable : valor

```
setPostsError({
  ...postsError,
  errorMessage : res.data.error,
  showError : true
})
```

UseEffect

La llamada al API, queremos que se ejecute automáticamente al renderizar la página, sin necesidad de que el usuario interactúe. Para ello, hacemos uso del hook useEffect, el cual le indica a React que el componente tiene que hacer algo después de renderizarse, en nuestro caso, llamar a la función que encapsula la llamada al API de los posts:

```
useEffect(() => {
  getAllPosts()
}, [])
```

Mencionar además, que el array al final sirve para optimizar el rendimiento. Ahí se incluiría una variable que React comprobaría cada vez que se renderiza el componente para decidir si ejecutar de nuevo o no el useEffect. En este caso, al estar vacío, solo se ejecutará una vez al renderizar el componente por primera vez, ya que no hay variable que comprobar.

Alert

En caso de que haya un error consumiendo el API, se le mostrará al usuario una alerta con un mensaje descriptivo del error:

```
64  const AlertDismissible = () => {
65    return (
66      <Alert
67        className="text-center"
68        variant="danger"
69        show={ postsError.showError }
70        onClose={ () => setPostsError({ ...postsError, showError : false }) }
71        dismissible
72      >
73        { postsError.errorMessage }
74      </Alert>
75    )
76  }
77
```

Esta alerta solo se mostrará cuando haya un error y el usuario podrá cerrarla (dismissible).

Renderizado de la vista completa

```
78    return (
79      <div>
80        <NavBarPrivate></NavBarPrivate>
81        <div className="container bg-white py-3">
82
83          { postsError.showError ? AlertDismissible() : ''}
84
85          <ListPostsFull posts={ posts } />
86          <div className="text-center">
87            <button
88              className="mt-4 btn btn-primary" onClick={ nextPage }>View more</button>
89          </div>
90        </div>
91      </div>
92    );
93  }
94
95  export default Home;
```

App.js

Es el componente “padre” de nuestra aplicación. Todas las vistas y componentes que renderizaremos serán renderizados dentro de este.

```
import React from 'react';

import {
  BrowserRouter as Router,
  Switch,
  Route,
  Redirect
} from "react-router-dom";

import UTILS from './utils'

import Home from './views/Home'
import SignIn from './views/SignIn'
import SignUp from './views/SignUp'

function App() {
  const ProtectedRoute = ({component, path, ...rest}) => {
    if(UTILS.user.isAuthenticated()){
      return <Route component={ component } path={ path } { ...rest } />
    }else{
      return <Redirect push to="/sign-in" { ...rest } />
    }
  }

  return (
    <Router>
      <div className="App">
        <Switch>
          <ProtectedRoute path='/' component={ Home } exact/>
          <Route path='/sign-in' component={ SignIn } exact/>
          <Route path='/sign-up' component={ SignUp } exact/>
        </Switch>
      </div>
    </Router>
  );
}

export default App;
```

Este contiene la definición de las rutas que componen nuestra aplicación haciendo uso de la librería “react-router-dom”. Deberemos encapsular todas las rutas dentro de un componente Router, que a su vez tendrá un componente Switch y ahí, ya pondremos de forma individual las rutas de nuestra aplicación.

Rutas públicas

Para estas hacemos uso directamente del componente Route, el cual mediante sus props path y component le indicamos que ruta debe comprobar y que componente renderizar cuando la ruta sea esa. Además, mediante el parámetro exact, le indicamos que la ruta debe ser exactamente igual a la indicada en el prop path.

```
<Route path='/sign-in' component={ SignIn } exact/>
<Route path='/sign-up' component={ SignUp } exact/>
```

Rutas protegidas

Es bastante probable que necesitemos proteger rutas, por ejemplo, para que solo puedan acceder autenticados. Para ello, creamos una función denominada ProtectedRoute, la cual recibirá mínimo dos parámetros: el componente a renderizar y el path. Esta función, haciendo uso de los métodos de UTILS, comprobará si el usuario está autenticado y en caso afirmativo, renderizará el componente de la ruta solicitada, y en caso contrario, lo redirigirá al login.

```
const ProtectedRoute = ({component, path, ...rest}) => {
  if(UTILS.user.isAuthenticated()){
    return <Route component={ component } path={ path } { ...rest } />
  }else{
    return <Redirect push to="/sign-in" { ...rest } />
  }
}
```

```
<ProtectedRoute path='/' component={ Home } exact/>
```

Observación

En este ejemplo, por mantener el ejemplo simple, hemos incluido las rutas dentro del App.js. Sin embargo, en un proyecto real, sería recomendable realizar su implementación en un archivo aparte, por ejemplo, llamado myrouter.js, que se incluiría y se utilizaría dentro de App.js. Así nuestra aplicación sería más modular.

Referencias

https://youtube.com/playlist?list=PLPl81lqbj-4KswGEN6o4lF0cscQalpycD

https://bluuweb.github.io/react-udemy/

https://es.reactjs.org/docs/hello-world.html