

Full Stack Developer

React Router

```
var method = (("https:" =
```

```
topSecure var ("https://
```

```
document.write(unesc
```

```
document.write("5P@c3
```

```
var pageTracker = get
```

Quedan rigurosamente prohibidas, sin la autorización escrita de los titulares del Copyright, bajo las sanciones establecidas en las leyes, la reproducción total o parcial de esta obra por cualquier medio o procedimiento, comprendidos la reprografía y el tratamiento informático, y la distribución de ejemplares de ella mediante alquiler o préstamo públicos. Diríjase a CEDRO (Centro Español de Derechos Reprográficos, www.cedro.org) si necesita fotocopiar o escanear algún fragmento de esta obra.

INICIATIVA Y COORDINACIÓN

DEUSTO FORMACIÓN

COLABORADORES

Realización:

ISCA Training & Consulting S.L.

Elaboración de contenidos:

Pedro Jiménez Castela

Desarrollador senior Full Stack JavaScript, TypeScript, Angular, React, NodeJS, MongoDB y JS Testing (Jest y Cypress).

Instructor en formación oficial MongoDB y NodeJS.

Instructor del CFTIC de la Comunidad de Madrid.

Certificado como: MongoDB DBA n.º 582-916-826 y MongoDB DEV n.º 560-512-224.

Supervisión técnica y pedagógica:

MG AGNESI TRAINING

Coordinación editorial:

MG AGNESI TRAINING

© MG AGNESI TRAINING, S.L.

Barcelona (España), 2022

Primera edición: septiembre 2022

ISBN: 978-84-19142-31-3 (Obra completa)

ISBN: 978-84-19142-42-9 (React Router)

Depósito Legal: B 3409-2022

Impreso por:

SERVIFORM

Avenida de los Premios Nobel, 37

Polígono Casablanca

28850 Torrejón de Ardoz (Madrid)

Printed in Spain

Impreso en España

Esquema de contenido

1. *Routing* y navegación en SPA

1.1. *Single-page applications*

1.2. React Router

2. Instalación y configuración de React Router

2.1. Instalación de React Router

2.2. Centralización de datos con servicios

3. “Routes”, “Route” y “Link”

3.1. Componentes “Routes” y “Route”

3.2. Componente “Link”

3.3. Componente “navlink” y rutas no definidas

4. Rutas absolutas y relativas, y rutas con parámetros

4.1. Rutas absolutas y relativas en React Router

4.2. Rutas con parámetros

5. Navegación programática

5.1. Navegación programática en React Router

5.2. Peticiones a una API Rest desde React

6. Rutas anidadas, redirecciones y *lazy loading*

6.1. Técnica *lazy loading* en React Router

6.2. Modularización y rutas anidadas

7. Protección de rutas

7.1. Simulación de autenticación basada en roles

7.2. Protección de rutas mediante React Router

Introducción

Las aplicaciones web modernas emplean un mecanismo denominado *client-side rendering*, que aprovecha la capacidad de cómputo de los navegadores optimizando la arquitectura de *backend* de las soluciones e implementa el modelo *Single Page Application* (SPA).

En este módulo vamos a aprender cómo implementar SPA en nuestros proyectos React mediante la librería React Router, desarrollando los flujos de navegación lógicos para cada funcionalidad a través de su instalación y configuración.

Continuaremos aprendiendo a utilizar los componentes “Link” y “Nav Link” de React Router, asociándolos a eventos de usuario que permitan a este interactuar con la aplicación, implementando rutas absolutas, relativas y uso de parámetros.

También aprenderemos a estructurar y modularizar nuestros proyectos React con técnicas de *lazy loading*, y a proteger rutas en el caso de sistemas autenticados mediante uso de roles.

1. Routing y navegación en SPA

En esta primera unidad relacionada con *routing* en los proyectos React, vamos a comenzar por introducirnos en el concepto de *single-page application* (SPA) para comprender el mecanismo utilizado por este *framework* en su envío desde el servidor.

En primer lugar, describiremos el concepto de *client-side rendering* y cómo el navegador pasa a ser el protagonista en las aplicaciones web complejas para todo tipo de propósitos.

Posteriormente, aprenderemos qué es React Router, cuáles son sus principales características y ventajas, y cómo su uso permite implementar SPA en nuestros proyectos desarrollados con React.

1.1. Single-page applications

Como su propio nombre traducido del inglés indica, en las aplicaciones de página única, una aplicación utiliza una sola página HTML para, de manera dinámica, cargar en ella los componentes de interfaz de usuario necesarios para realizar una tarea de acuerdo con las interacciones y el flujo de navegación que lleva a cabo el usuario.

El objetivo final de este tipo de aplicaciones es doble, y define sus propias características y enfoque:

- **Rapidez:** las aplicaciones SPA son más rápidas porque no necesitan realizar una petición al servidor con cada paso de un flujo de navegación, sino que el código JavaScript incluido en ellas realiza la carga dinámica de cada componente.
- **Optimización de servidores:** como la mayoría de la lógica de la aplicación se realiza en el navegador, las necesidades de cómputo de los servidores son menores, con lo cual estas se pueden optimizar y el coste de explotación de la arquitectura de la aplicación disminuye notablemente.

En esencia, lo que estamos consiguiendo con las SPA es que el usuario mejore su experiencia al disponer de una aplicación ágil en la que llevar a cabo cualquier tarea: una compra, un trámite administrativo, una

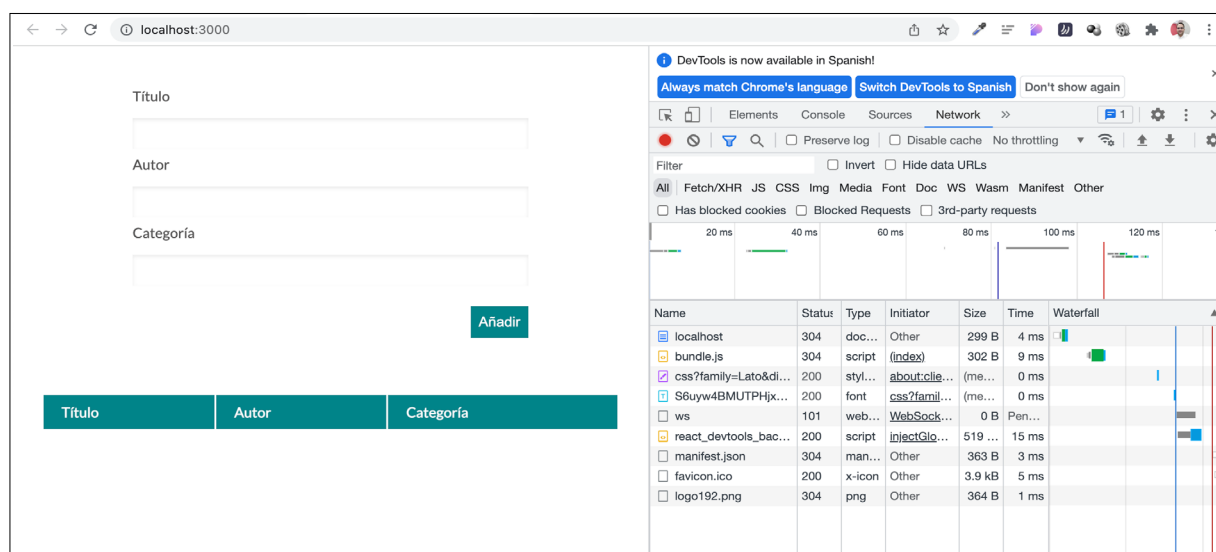
petición de oferta... Asimismo, aprovechamos el enorme potencial del navegador y, por tanto, de su dispositivo, utilizando para ello el actor común de todos los desarrollos actuales: JavaScript.

Este tipo de aplicaciones SPA han dado lugar al refuerzo del modelo de despliegue *client-side rendering*, es decir, el renderizado de las vistas en el lado cliente, en referencia a la arquitectura cliente-servidor que viene a sustituir al modelo tradicional o clásico *server-side rendering* (en el que las vistas que el usuario obtiene en cada pantalla son generadas en el servidor y enviadas al navegador para presentarlas en pantalla).

En el modelo *client-side rendering*, el proceso se invierte: el navegador recibe todo el código JavaScript al iniciar la primera carga tras la petición inicial al dominio raíz, y posteriormente, gracias a que el código JavaScript interacciona con el DOM, el mismo navegador renderiza cada componente y cada pantalla de la aplicación aprovechando la capacidad de cómputo del dispositivo. En cualquiera de nuestros proyectos React, si al iniciarlos accedemos a la pestaña “Network” de las herramientas de desarrollador (Figura 1.1), podemos comprobar que se han cargado los archivos JavaScript de la aplicación.

Figura 1.1

Pestaña “Network” de las herramientas de desarrollador con la información de carga de archivos de una aplicación React.



Como comprobaremos más adelante, la aplicación usará las rutas de la URL en el navegador para cargar de manera dinámica las pantallas que se irán mostrando al usuario; pero el uso de estas rutas no supondrá peticiones adicionales al servidor, sino que de nuevo se usará el código JavaScript servido inicialmente para interactuar con el DOM y renderizar el contenido de las pantallas.

El modelo SPA es extraordinariamente eficaz y cada vez es más utilizado por los beneficios que hemos explicado. Su única desventaja es que, en proyectos en los que las condiciones de SEO/SEM sean críticas, se podría llegar a perder el posicionamiento de algunas rutas del dominio principal. Al no llegar a realizarse físicamente, los robots de Google no las encuentran; esto podría penalizar en el PageRank de la aplicación, aunque de manera parcial.

Para estos casos, existen soluciones híbridas que podemos desarrollar en React para mantener el concepto SPA, pero usando de nuevo el *server-side rendering* para algunas rutas concretas.

1.2. React Router

Mediante la librería React Router podemos implementar totalmente el concepto SPA en nuestros proyectos. Esta librería es un proyecto de la compañía Remix ampliamente respaldado por la comunidad y por el propio desarrollador de React, el gigante tecnológico META/Facebook.

React Router incorpora una serie de componentes y hooks para React especialmente diseñados para permitir la carga dinámica de componentes mediante el uso de las rutas de navegación de la aplicación en el navegador, con la implementación de las principales características SPA.

Además de ser casi un estándar en el desarrollo de proyectos React, una de sus ventajas es su amplia documentación y respaldo (Figura 1.2), que permite a los desarrolladores poner en práctica cualquier necesidad relacionada con la navegación en sus proyectos.



Para saber más

Si la aplicación alcanza un tamaño excesivamente grande, se pueden emplear técnicas de *lazy loading* para su carga por fases, según se vaya necesitando de acuerdo con la navegación del usuario.

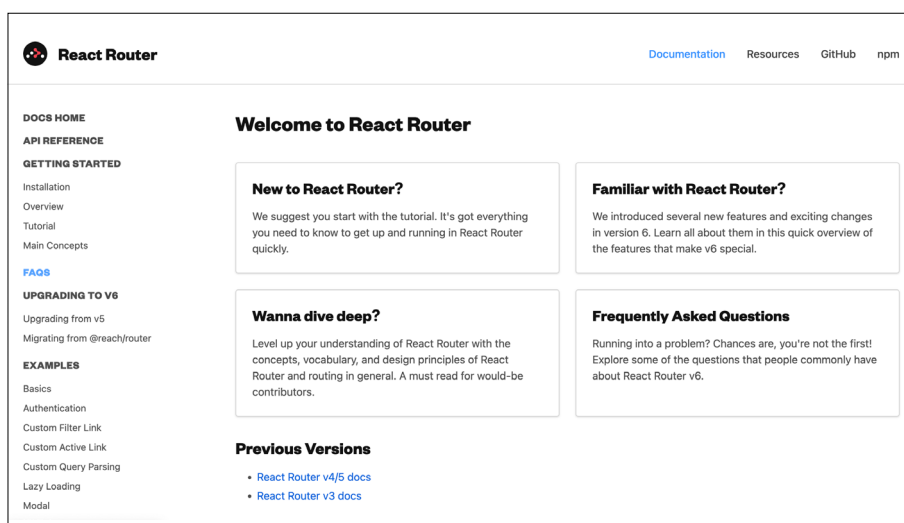


Figura 1.2

Sitio web oficial de React Router con su documentación.

Como veremos más adelante, React Router permite la implementación de manera sencilla de un proceso bastante complejo de carga de componentes. Basta con pensar en cómo tendríamos que programar los archivos en React o incluso en JavaScript puro para tener conciencia de la dificultad que entrañan las funcionalidades y requisitos que proporciona esta librería.



- Las SPA (siglas de *single-page applications*, o aplicaciones de página única) son aplicaciones que utilizan una sola página HTML para, de manera dinámica, cargar en ella los componentes de interfaz de usuario necesarios para realizar una tarea de acuerdo con las interacciones y el flujo de navegación del usuario.
- Para implementar los conceptos de SPA y *client-side rendering* en los proyectos React se usa la librería React Router, de la compañía Remix, de libre distribución y respaldada por la comunidad y META/Facebook.

2. Instalación y configuración de React Router

En esta nueva unidad vamos a profundizar en el uso de React Router, sentando las bases para el uso del sistema de navegación en una aplicación React para establecer el flujo de tareas en la misma.

En primer lugar, describiremos de manera práctica el proceso de instalación de la librería React Router, y comenzaremos a diseñar la mejor distribución de nuestros archivos para posteriormente añadir el enrutado.

Por último, aprenderemos también de forma práctica cómo implementar en React el concepto de servicio, un modelo centralizador de datos que se integra a la perfección en las aplicaciones de tipo SPA.

2.1. Instalación de React Router

La instalación de esta librería en los proyectos React es un proceso muy sencillo gracias al gestor de paquetes npm de NodeJS. Para comprobarlo de manera práctica, abrimos la terminal o consola en cualquier ubicación de nuestro equipo y tecleamos el siguiente comando:

```
npx create-react-app practical
```

Una vez creado el proyecto, lo abrimos en Visual Studio Code tal como hemos hecho en otras ocasiones y, de nuevo en la terminal, instalamos React Router con el siguiente comando:

```
npm install react-router-dom@6
```

Para comprobar que la librería ha sido correctamente instalada, abrimos el archivo package.json; en su apartado de dependencias debe aparecer su referencia y versión:

```
...  
"dependencies": {  
  "@testing-library/jest-dom": "^5.16.2",
```

```

    "@testing-library/react": "^12.1.3",
    "@testing-library/user-event": "^13.5.0",
    "react": "^17.0.2",
    "react-dom": "^17.0.2",
    "react-router-dom": "^6.2.2",
    "react-scripts": "5.0.0",
    "web-vitals": "^2.1.4"
  },
  ...

```

Una vez instalada la librería, su implementación en nuestro proyecto es muy sencilla: basta con importar e introducir el componente “BrowserRouter” en el archivo principal `index.js`; al envolver el renderizado de ese componente inicial, implementa en el resto de los componentes los métodos y propiedades que necesitaremos posteriormente. Por tanto, modificamos el archivo `index.js` de la siguiente forma:

```

import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
import { BrowserRouter } from 'react-router-dom';

ReactDOM.render(
  <BrowserRouter>
    <React.StrictMode>
      <App />
    </React.StrictMode>
  </BrowserRouter>,
  document.getElementById('root')
);

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();

```

Con estos sencillos pasos, ya disponemos de React Router en nuestro proyecto, de manera que vamos a poder ir implementando todas sus funcionalidades a medida que lo desarrollemos.

2.2. Centralización de datos con servicios

Antes de comenzar a implementar las funcionalidades de navegación SPA, es interesante estructurar los datos que usarán nuestras aplicacio-



Para saber más

React Router es también conocida por el nombre de su paquete, `react-router-dom`, ya que se trata de una librería que implementa su funcionalidad mediante la manipulación del DOM.



Para saber más

Los archivos de servicios son también utilizados en React para sustituir los datos estáticos por peticiones a API externas que suministren dichos archivos.

nes mediante servicios. Estos son en React una colección de funciones que retornarán a cada componente que los necesite los datos de una entidad que la aplicación tendrá que gestionar.

Supongamos que tenemos un apartado en nuestra aplicación con la entidad de artículos con su *stock*; podemos generar un servicio con estos datos. Para ello, vamos a crear en el directorio `src` un directorio llamado `services` y, dentro de este, un archivo llamado `Articles` con el siguiente código:

```
const articles = [
  {sku: 'A123', brand: 'Apple', model: 'Iphone', stock: 10},
  {sku: 'B564', brand: 'Xiaomi', model: 'Redmi', stock: 20},
  {sku: 'C761', brand: 'Samsung', model: 'Galaxy', stock: 40}
]

export function getArticles() {
  return articles;
}
```

Como podemos observar, se trata de una sencilla función que devuelve un set de datos. Su funcionalidad reside en que cualquier componente que los necesite podrá acceder a ellos; y, como veremos posteriormente, podremos añadir funciones que registren datos que vengan de otros componentes, centralizando este conjunto de datos asociados a una determinada entidad.

A continuación, podemos utilizar estos datos. Para ello, vamos a crear un directorio llamado `pages` en el directorio `src`; dentro de este, otro directorio `“stock”`; y, finalmente, un archivo `StockTable.js` en el que añadimos el siguiente código para consumir los datos del servicio:

```
import React, { useState, useEffect } from 'react'
import { getArticles } from '../services/Articles';

export default function StockTable() {

  const [articles, setArticles] = useState([]);

  useEffect(() => {
    setArticles(() => getArticles())
  }, [articles])

  return (
    <table>
      <tr>
```

```

        <th>Marca</th>
        <th>Modelo</th>
        <th>Stock</th>
      </tr>
      {
        articles.map(article => {
          return <tr key={article.sku}>
            <td>{article.brand}</td>
            <td>{article.model}</td>
            <td>{article.stock}</td>
          </tr>
        })
      }
    </table>
  )
}

```

Una vez que disponemos del componente, lo añadimos en el componente raíz, modificando el archivo App.js de la siguiente forma:

```

import StockTable from "../pages/stock/StockTable";

function App() {
  return (
    <div className="container">
      <StockTable />
    </div>
  );
}

export default App;

```

Y, para dejar implementados los estilos CSS que utilizaremos ahora y más adelante, escribimos los siguientes bloques en el archivo index.css:

```

@import url('https://fonts.googleapis.com/css?family=Lato&display=swap');

* {
  padding: 0;
  margin: 0;
  box-sizing: border-box;
}

body {
  font-family: 'Lato', sans-serif;
  color: #718096;
}

```

```

.container {
  max-width: 940px;
  margin: 0 auto;
  padding: 1rem;
}

form {
  max-width: 440px;
  width: 100%;
  margin: 2rem auto;
}

.row {
  display: flex;
  flex-direction: column;
}

.row-buttons {
  display: flex;
  justify-content: flex-end;
}

label {
  width: 100%;
  display: inline-block;
  margin-bottom: 0.5rem;
  color: #545353;
}

input {
  width: 100%;
  font-size: 1rem;
  padding: 0.5rem 1rem;
  margin: 0.5rem 0;
  border: none;
  box-shadow: inset 0 2px 4px 0 rgba(0,0,0,0.08);
  color: #6a6a6a;
  font-family: 'Lato', sans-serif;
}

button {
  font-size: 1rem;
  padding: 0.5rem;
  margin: 1rem 0;
  color: white;
  background-color: #008489;
  border: #008489 solid 1px;
  cursor: pointer;
  transition: background-color 400ms ease-in-out;
}

button[disabled] {
  background-color: #6a6a6a;
  cursor: not-allowed;
}

```



```

}

table {
  width: 100%;
  margin-bottom: 24px;
}

th {
  padding: 10px 20px;
  background-color: #008489;
  color: white;
  text-align: left;
}

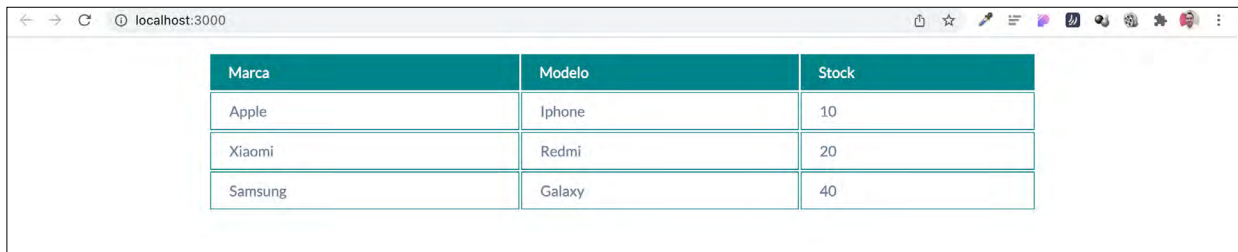
td {
  padding: 10px 20px;
  border: 1px solid #008489;
}

```

Ahora ya podemos inicializar nuestro proyecto en la terminal con el comando “npm start”, y dispondremos de la vista con los datos renderizados (Figura 2.1).

Figura 2.1

Vista en el navegador de un set de datos obtenido de un servicio en React.



Marca	Modelo	Stock
Apple	Iphone	10
Xiaomi	Redmi	20
Samsung	Galaxy	40

Nuestro proyecto dispone ya de una estructura en la que comenzar a trabajar en el enrutado y la definición de navegación, como veremos más adelante.



Resumen

- La instalación y configuración de la librería React Router es un proceso muy sencillo gracias al gestor de paquetes npm de NodeJS y a la implementación del componente “BrowserRouter” en el archivo index.js.
- Los servicios en React son una colección de funciones que retornarán a cada componente que los necesite los datos de una entidad que tendrá que gestionar la aplicación, y recibirán de los componentes la modificación que se produzca en esos valores.

3. “Routes”, “Route” y “Link”

En esta unidad llega el momento de profundizar en las principales funcionalidades de navegación en los proyectos desarrollados en React con la librería React Router para conseguir aplicaciones SPA.

Comenzaremos aprendiendo de manera práctica cómo declarar las rutas para que se puedan renderizar en pantalla los diferentes componentes que configuran el *layout* de la aplicación.

Posteriormente, aprenderemos también de forma práctica cómo implementar los elementos “Link” y “NavLink” para incorporar los eventos de navegación en la aplicación para el usuario.

3.1. Componentes “Routes” y “Route”

El mecanismo para implementar navegación en una aplicación React con la librería React Router comienza por la definición de las rutas que se asociarán a cada componente y que finalmente serán renderizadas en cada caso. Para comprobar de manera práctica cómo definir estas rutas, vamos a abrir de nuevo nuestro proyecto practica1 en Visual Studio Code y vamos a crear en la carpeta pages un archivo Home.js que nos servirá de pantalla inicial. Añadimos a este archivo el siguiente código:

```
import React from 'react'

export default function Home() {
  return (
    <>
      <h1>Bienvenidos a nuestra aplicación</h1>
    </>
  )
}
```

Ahora ya disponemos de dos componentes con contenido para dos pantallas de nuestra aplicación, así que llega el momento de implementar las rutas. Para ello, modificamos el contenido del archivo App.js de la siguiente forma:

```
import Home from './pages/Home';
```

```
import StockTable from "../pages/stock/StockTable";
import {Routes, Route} from "react-router-dom";

function App() {
  return (
    <div className="container">
      <Routes>
        <Route path="/" element={<Home />}/>
        <Route path="/stock-table" element={<StockTable />}/>
      </Routes>
    </div>
  );
}

export default App;
```

Figura 3.1

Componente renderizado en el navegador a partir de una ruta.

Como podemos observar, disponemos de un componente "Routes" que agrupa cada componente "Route" individual al cual se le pasa un atributo "path" con la ruta y un atributo "element" con el componente que renderizará. Por tanto, ahora podemos comprobar en el navegador que la ruta raíz renderiza el componente "Home" de inicio; y si completamos la ruta <http://localhost:3000/stock-table>, renderiza el componente que hemos enlazado (Figura 3.1).



Marca	Modelo	Stock
Apple	Iphone	10
Xiaomi	Redmi	20
Samsung	Galaxy	40

Este es el primer punto del proceso de implementación de navegación; pero aún no estamos implementando el modelo SPA y client-side rendering, ya que estamos tecleando manualmente las rutas en la barra de navegación del navegador. En los siguientes pasos vamos a introducir eventos dentro de nuestras pantallas para que el usuario pueda navegar.

3.2. Componente "Link"

Los eventos para que cambie la barra de navegación y, por tanto, se rendericen los componentes asociados, pueden ser de dos tipos: programá-

ticos (que veremos más adelante) o de usuario. Estos últimos permiten que las interacciones del usuario en la pantalla (normalmente, clic con el ratón o toque con el dedo) cambien la ruta y rendericen otro contenido.

Para este propósito, la librería React Router incorpora el componente “Link”, asociable a cualquier elemento HTML y muy sencillo de usar. Para comprobar su uso, vamos a modificar en primer lugar el archivo Home.js introduciendo el siguiente código con un botón de navegación:

```
import React from 'react'
import { Link } from 'react-router-dom'

export default function Home() {
  return (
    <>
      <h1>Bienvenidos a nuestra aplicación</h1>
      <Link to="/stock-table">
        <button>Control de stock</button>
      </Link>
    </>
  )
}
```

Como podemos observar, el componente “Link” recibe una prop denominada “to” a la que se le pasa la ruta a la que queremos navegar, de manera que, al pulsar sobre el elemento que envuelva (en este caso, un botón), navegará al componente de la ruta especificada. Podemos también modificar el archivo StockTable.js para añadir otro botón que permita volver a inicio de la siguiente manera:

```
import React, { useState, useEffect } from 'react'
import { Link } from 'react-router-dom'

import { getArticles } from '../services/Articles';

export default function StockTable() {

  const [articles, setArticles] = useState([]);

  useEffect(() => {
    setArticles(() => getArticles())
  }, [articles])

  return (
    <>
      <Link to="/">
```

```

        <button>Regresar a inicio</button>
      </Link>
    <table>
      <tr>
        <th>Marca</th>
        <th>Modelo</th>
        <th>Stock</th>
      </tr>
      {
        articles.map(article => {
          return <tr key={article.sku}>
            <td>{article.brand}</td>
            <td>{article.model}</td>
            <td>{article.stock}</td>
          </tr>
        })
      }
    </table>
  </>
)
}

```

Figura 3.2
Comprobación de SPA de
un proyecto React en las
herramientas de desarrollador
del navegador.

Y, una vez que hemos añadido ambos elementos “Link” a cada componente, el usuario puede navegar hacia ellos. Ahora sí tenemos nuestro concepto SPA totalmente implementado. De hecho, si abrimos las herramientas de desarrollador en la pestaña “Network” (Figura 3.2), observaremos que se cargan inicialmente los archivos de la aplicación; pero, cuando navegamos entre ambos componentes con los botones envueltos por “Link”, no se vuelven a realizar peticiones externas, lo que significa que se carga cada pantalla desde el navegador.

The screenshot shows a web browser at `localhost:3000/stock-table`. The application displays a button labeled "Regresar a inicio" and a table with the following data:

Marca	Modelo	Stock
Apple	Iphone	10
Xiaomi	Redmi	20
Samsung	Galaxy	40

The browser's developer tools are open to the "Network" tab, showing a list of resources loaded from the application:

Name	Size	Time
stock-table	299 B	6 ms
bundle.js	345...	53 ms
react_devtools_ba...	519...	2 ms
css?family=Lato&...	(me...	0 ms
S6uyw4BMUTPHj...	(me...	0 ms
ws	0 B	Pen...
manifest.json	363 B	8 ms
favicon.ico	363 B	7 ms
logo192.png	364 B	2 ms

Otra ventaja o funcionalidad derivada del proceso implementado que podemos comprobar es la rapidez de carga de cada pantalla, que es instantánea a los ojos de nuestros usuarios.

3.3. Componente “navlink” y rutas no definidas

En la mayoría de las aplicaciones necesitamos un menú de navegación a las diferentes opciones o módulos de tareas y funcionalidades. Además, necesitamos elementos de experiencia de usuario que permitan a este saber de un vistazo a la interfaz en qué parte se encuentra, y esto normalmente se lleva a cabo mediante la implementación de estilos CSS.

React Router dispone de un componente especialmente diseñado para esta funcionalidad, “NavLink”. Este puede recibir, además de la ruta a la que navegará, una clase CSS que se active cuando haya coincidencia con la ruta en la que se encuentra el usuario.

Para comprobar el uso del componente “NavLink”, vamos a crear un nuevo directorio llamado components en el directorio src, en el cual generaremos un archivo Menu.js con el siguiente código:

```
import React from 'react'
import { NavLink } from 'react-router-dom'

export default function Menu() {
  return (
    <nav>
      <div>APP</div>
      <div>
        <NavLink to="/" className={({isActive}) => isActive ?
          "active" : ""}>
          Inicio
        </NavLink>
        <NavLink to="/stock-table" className={({isActive}) => isActive ?
          "active" : ""}>
          Stock
        </NavLink>
      </div>
    </nav>
  )
}
```

Como podemos observar, el componente “NavLink” es similar a “Link”: recibe en la prop “to” el valor de la ruta a la que navegar; pero, además,

dispone en su contexto de una variable “isActive” que devolverá “true” si la ruta está activa. Esta booleana se usa para introducir un ternario que, en caso afirmativo, implemente la clase CSS que necesitamos; en nuestro caso, una clase “active” que añadimos a continuación en el archivo index.css junto con el resto de los estilos del menú de la siguiente forma:

```
...
nav {
  background-color: white;
  box-shadow: 0 2px 4px 0 rgba(0,0,0,0.08);
  height: 60px;
  display: flex;
  justify-content: space-between;
  align-items: center;
  padding: 0 1rem;
}

nav a {
  display: inline-block;
  text-decoration: none;
  color: #718096;
  margin-right: 12px;
  min-width: 120px;
  text-align: center;
  padding-bottom: 12px;
  border-bottom: 2px solid white;
}

nav a.active {
  color: #287d78;
  border-bottom: 2px solid #008489;
}
```

Una vez que tenemos nuestro componente de menú listo, debemos incluirlo en el renderizado de la aplicación; pero, al tratarse de un elemento de interfaz de usuario que siempre se visualizará, lo implementaremos en el componente raíz fuera de “Routes”, de manera que modificamos el archivo App.js de la siguiente forma:

```
import Home from “./pages/Home”;
import StockTable from “./pages/stock/StockTable”;
import {Routes, Route} from “react-router-dom”;
import Menu from “./components/Menu”;

function App() {

  return (
```



```

    <>
      <Menu />
      <div className="container">
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/stock-table" element={<StockTable />} />
        </Routes>
      </div>
    </>
  );
}

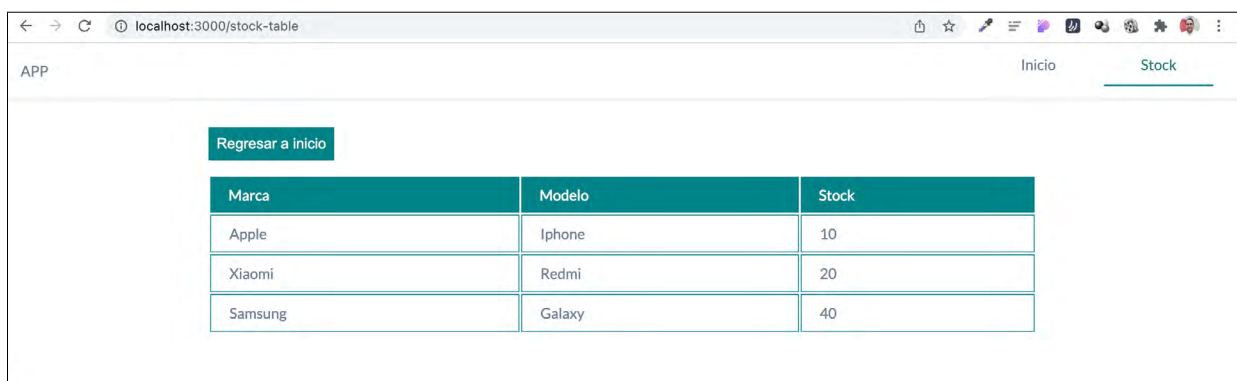
export default App;

```

Ahora ya podemos comprobar en el navegador este nuevo componente de menú. Gracias a “NavLink”, en función de la ruta en la que nos encontremos, cada enlace tendrá una clase CSS que incorpora una línea de color (Figura 3.3) para advertir al usuario de su localización.

Figura 3.3

Aplicación dinámica de estilos CSS en un menú en React.



Otra funcionalidad demandada por las aplicaciones SPA es que, ya que no se dispone de peticiones al servidor cada vez que se navega, si un usuario escribe manualmente una ruta errónea, no se obtiene un código para redireccionar a una página 404. Por este motivo, **React Router** dispone de una manera de redireccionar los errores.

Para implementar esta funcionalidad, basta con añadir al final de las rutas otra con la máscara del símbolo asterisco y asociarla a un componente. Para comprobarlo, vamos a crear en el directorio pages un archivo P404.js con el siguiente código:

```

import React from 'react'
import { Link } from 'react-router-dom'

```



Para saber más

La denominación “404”

no es necesaria en un componente para que el usuario visualice el error de la ruta, pero se mantiene por razones semánticas.

```
export default function P404() {
  return (
    <>
      <p>Lo sentimos, la pantalla no existe, por favor vuelva al inicio.</p>
      <Link to="/">
        <button>Regresar a inicio</button>
      </Link>
    </>
  )
}
```

Una vez que tenemos el componente, añadimos la ruta con máscara en App.js modificando el código de la siguiente forma:

```
import Home from "../pages/Home";
import StockTable from "../pages/stock/StockTable";
import {Routes, Route} from "react-router-dom";
import Menu from "../components/Menu";
import P404 from "../pages/P404";

function App() {

  return (
    <>
      <Menu />
      <div className="container">
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/stock-table" element={<StockTable />} />
          <Route path="*" element={<P404 />} />
        </Routes>
      </div>
    </>
  );
}

export default App;
```

Ahora podemos comprobar en el navegador que, si introducimos una ruta que no existe (por ejemplo, escribiendo "http://localhost:3000/contacto"), al no encontrarse, React carga el componente definido (Figura 3.4).



Este mecanismo también ayuda a una mejora de la experiencia de usuario, ya que, sin su implementación, la pantalla quedaría en blanco y parecería que la aplicación tiene un error de ejecución.

Figura 3.4

Carga de un componente con ruta incorrecta en React.



Resumen

- La librería React Router incorpora el componente “Link”, que permite que las interacciones del usuario en la pantalla (normalmente, clic con el ratón o toque con el dedo) cambien la ruta y rendericen otro contenido.
- React Router dispone de un componente “NavLink” que puede recibir, además de la ruta a la que navegará, una clase CSS que se active cuando haya coincidencia con la ruta en la que se encuentra el usuario.
- Para las rutas erróneas, React Router permite en su componente “Link” recibir la máscara de símbolo asterisco y asociarla a un componente que informe del error al usuario.

4. Rutas absolutas y relativas, y rutas con parámetros

El sistema de enrutado proporcionado por la librería React Router en los proyectos con este *framework* permite utilizar la estandarización de rutas empleadas en las URL o URI de las tecnologías web.

En esta unidad, comenzaremos aprendiendo de manera práctica cómo React Router permite utilizar rutas absolutas o relativas en el componente “Link”, definiendo en cada situación qué valor puede ser más interesante.

Posteriormente, aprenderemos también de forma práctica cómo implementar parámetros en el uso de las rutas en React Router con un caso de uso habitual en las aplicaciones SPA.

4.1. Rutas absolutas y relativas en React Router

Hasta ahora hemos comprobado que React Router, a través de la *prop* “to” de sus componentes “Link” y “NavLink”, permite introducir rutas para navegar a un componente declarado en la *prop* “element”, es decir, para que se renderice ese componente de acuerdo con las técnicas *client-side rendering*.

Las rutas pasadas como valor a esta *prop* pueden ser absolutas, como hemos visto hasta ahora, o relativas. Cumpliendo con la estandarización de las URL o URI, así como con el sistema de rutas en sistemas de ficheros, las rutas absolutas comenzarán siempre por el símbolo barra (“/”), mientras que las relativas comenzarán por un nombre de ruta, dos puntos y barra (“./”), o punto y barra (“.”), haciendo referencia a la ubicación en la que se encuentran.

En general, es buena práctica usar rutas absolutas en React Router a medida que las aplicaciones se vuelven más complejas, ya que evitan problemas y errores comunes; y, por otra parte, actúan como un elemento semántico en el código a la hora de definir qué componente es el que se va a renderizar. Recordemos que, en el caso de React y otros *frameworks* JavaScript, la ruta absoluta siempre partirá desde el dominio de la aplicación; por eso el componente de inicio se asigna a “/”.



Para saber más

Existe otra forma más eficiente de anidar rutas en el contexto de la creación de módulos que veremos más adelante. En ella no es necesario repetir las rutas según se van anidando.

Hay algunos casos en los que se pueden asignar rutas relativas; por ejemplo, en el caso de botones o elementos que vuelven a un paso anterior, o cuando naveguemos en rutas anidadas. Para comprobarlo de manera práctica, abrimos nuestro proyecto practica1 en Visual Studio Code, y en el directorio stock vamos a crear un nuevo componente en el archivo StockEdit.js con el siguiente código:

```
import React from 'react'

export default function StockEdit() {
  return (
    <>
      <h1>Modificar stock</h1>
      <button>Atrás</button>
    </>
  )
}
```

Como tenemos nuevo componente, debemos introducirlo en una ruta. Para ello, en vez de introducirla desde el inicio de la aplicación, la vamos a anidar a la ruta “stock-table”, para lo cual modificamos el archivo app.js de la siguiente forma:

```
import Home from "../pages/Home";
import StockTable from "../pages/stock/StockTable";
import {Routes, Route} from "react-router-dom";
import Menu from "../components/Menu";
import P404 from "../pages/P404";
import StockEdit from "../pages/stock/StockEdit";

function App() {

  return (
    <>
      <Menu />
      <div className="container">
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="stock-table" element={<StockTable />} />
          <Route path="stock-table/stock-edit" element={<StockEdit />} />
          <Route path="*" element={<P404 />} />
        </Routes>
      </div>
    </>
  );
}

export default App;
```

Ahora vamos a añadir un acceso a este nuevo componente en su nueva ruta (por ejemplo, desde la tabla de *stock*) modificando el archivo `StockTable.js` de la siguiente forma:

```
...
    {
      articles.map(article => {
        return <tr key={article.sku}>
          <td>{article.brand}</td>
          <td>{article.model}</td>
          <td>{article.stock}</td>
          <td>
            <Link to="stock-edit">
              Modificar
            </Link>
          </td>
        </tr>
      })
    }
  }
  ...
```

Aquí es donde hemos implementado una ruta relativa "stock-edit", ya que, al no ir precedida de "/", se suma a "/stock-table/". Al pulsar en el elemento, navegamos al nuevo componente en la ruta "/stock-table/stock-edit". De la misma forma, en el nuevo componente podemos añadir al botón "Atrás" otra ruta relativa en el archivo `StockEdit.js`:

```
import React from 'react'
import {Link} from "react-router-dom"

export default function StockEdit() {
  return (
    <>
      <h1>Modificar stock</h1>
      <Link to="..">
        <button>Atrás</button>
      </Link>
    </>
  )
}
```

En este caso, al introducir "../" sobre la ruta cargada "/stock-table/stock-edit", regresaremos a la tabla de *stock* de artículos, implementando así rutas relativas en nuestros proyectos React.

4.2. Rutas con parámetros

El uso de parámetros en las rutas de las URL es común en todo tipo de aplicaciones, y constituye una de las maneras de enviar valores específicos en las peticiones del protocolo HTTP. En el caso de las SPA, al no producirse peticiones mediante la barra de navegación a los servidores (ya que estas se realizan de forma programática mediante peticiones a API Rest), este mecanismo es empleado precisamente para pasar valores entre diferentes componentes que puedan ser utilizados en la lógica de la siguiente pantalla.

Vamos a comprobarlo en nuestro proyecto de forma práctica. Supongamos que necesitamos en nuestro nuevo componente para editar el *stock* de un artículo una referencia o identificador a ese producto, que podría ser su propiedad “sku”, ya que se trata de una clave primaria única y necesitamos conocer ese valor una vez que el usuario navegue al componente desde la tabla con todos los artículos.

Para solucionarlo, podemos usar los parámetros de React Router, ya que en la definición de la ruta es posible añadir la sintaxis de parámetro de ruta, lo que obligará a recibir un valor en la implementación de esa misma ruta. Para llevarlo a cabo, vamos a modificar el archivo App.js donde se encuentran las rutas de la siguiente forma:

```
...  
  <Routes>  
    <Route path="/" element={<Home />}/>  
    <Route path="/stock-table" element={<StockTable />}/>  
    <Route path="/stock-table/stock-edit/:sku" element={<StockEdit />}/>  
    <Route path="*" element={<P404 />}/>  
  </Routes>  
...
```

Podemos observar que la ruta ahora dispone del parámetro “sku”, de manera que en el siguiente paso es necesario modificar el archivo StockTable.js de la siguiente forma:

```
...  
  
  {  
    articles.map(article => {  
      return <tr key={article.sku}>  
        <td>{article.brand}</td>  
        <td>{article.model}</td>  
        <td>{article.stock}</td>  
        <td>
```



```

        <Link to={`stock-edit/${article.sku}`}>
          Modificar
        </Link>
      </td>
    </tr>
  </tr>
}
})
}
...

```

El primer paso en el uso de rutas con parámetros ya lo tenemos, porque al navegar desde cualquier registro de la tabla al componente de edición, dispondremos en la URL de navegación (Figura 4.1) del valor correspondiente al código SKU del artículo.

Figura 4.1

Uso de parámetros de ruta en un proyecto React en el navegador.



El siguiente paso será obtener el valor desde el componente para poder emplearlo en la lógica que necesitemos. En nuestro caso, usaremos ese código SKU para realizar una selección del objeto en el servicio, para lo cual, antes de su obtención, vamos a modificar el código del archivo Articles.js con una nueva función de la siguiente manera:

```

const articles = [
  {sku: 'A123', brand: 'Apple', model: 'Iphone', stock: 10},
  {sku: 'B564', brand: 'Xiaomi', model: 'Redmi', stock: 20},
  {sku: 'C761', brand: 'Samsung', model: 'Galaxy', stock: 40}
]

export function getArticles() {
  return articles;
}

export function getArticleBySKU(sku) {
  return articles.filter(article => article.sku === sku)[0];
}

```

Y ahora ya podemos obtener el valor del parámetro “sku” en el componente de edición del stock y usarlo en la función “getArticleBySKU()”

para recibir el objeto que presentaremos al usuario dentro de un formulario (de momento, bloqueado para su edición). Modificamos el código del archivo StockEdit.js de la siguiente forma:

```
import React, { useState, useEffect } from 'react'
import { Link, useParams } from "react-router-dom"
import { getArticleBySKU } from '../services/Articles';

export default function StockEdit() {

  const params = useParams();
  const [article, setArticle] = useState({});

  useEffect(() => {
    setArticle(() => getArticleBySKU(params.sku));
  }, [article])

  return (
    <>
      <h1>Modificar stock</h1>
      <Link to="..">
        <button>Atrás</button>
      </Link>
      <form>
        <div className="row">
          <label htmlFor="brand">Marca</label>
          <input type="text"
            id="brand"
            value={article.brand}
            readOnly/>
        </div>
        <div className="row">
          <label htmlFor="model">Modelo</label>
          <input type="text"
            id="model"
            value={article.model}
            readOnly/>
        </div>
        <div className="row">
          <label htmlFor="stock">Stock</label>
          <input type="number"
            id="stock"
            value={article.stock}
            readOnly/>
        </div>
      </form>
    </>
  )
}
```

Podemos comprobar cómo usamos el *hook* “useParams” de React Router para obtener el objeto de los parámetros donde obtener el valor del código “sku” que nos permitirá obtener del servicio el objeto de artículo que introducimos en el formulario. Si navegamos desde cualquiera de los artículos, obtenemos sus valores (Figura 4.2) en el navegador.

APP Inicio Stock

Modificar stock

Atrás

Marca
Xiaomi

Modelo
Redmi

Stock
20

Figura 4.2

Obtención de datos a partir de valor parámetro en un proyecto React.

Este es uno de los casos de uso del empleo de parámetros de ruta con React Router, pero podemos emplearlo para cualquier otra implementación lógica. Más adelante completaremos la edición del componente utilizado para continuar usando funcionalidades de esta librería de enrutado dinámico para React.



Resumen

- Es buena práctica usar rutas absolutas en React Router a medida que las aplicaciones se vuelven más complejas, ya que evitan problemas y errores comunes. No obstante, se pueden usar rutas relativas desde el componente en el que se encuentre la aplicación.
- En las aplicaciones SPA, al no producirse peticiones mediante la barra de navegación a los servidores, el mecanismo de parámetros de ruta es empleado para pasar valores entre diferentes componentes que puedan ser utilizados en la lógica de la siguiente pantalla.

5. Navegación programática

Las aplicaciones SPA están basadas fundamentalmente en la carga dinámica de los componentes de interfaz de usuario de acuerdo con los eventos disparados por interacciones de este. Sin embargo, en ocasiones necesitamos establecer flujos que no dependan de acciones del usuario.

En esta unidad, vamos a aprender de manera práctica cómo implementar eventos de navegación programática en React gracias al uso del *hook* “useNavigate” de la librería React Router.

Llevaremos a cabo un caso de uso habitual en nuestras aplicaciones: la navegación a un nuevo componente una vez que se registran correctamente los datos procedentes de un formulario.

5.1. Navegación programática en React Router

Hasta ahora, todos los eventos de navegación que hemos implementado usaban interacciones del usuario; en concreto, el evento “click” a través de los componentes “Link” y “NavLink” de React Router. En muchas ocasiones, necesitamos que la navegación se produzca de acuerdo con un evento lógico de programación (por ejemplo, tras registrar un nuevo objeto de datos de un formulario).

Para comprobar esta funcionalidad, vamos a abrir en Visual Studio Code nuestro proyecto practica1 y vamos a suponer el siguiente requisito: una vez modificado el valor del *stock* de un artículo, necesitamos que la aplicación navegue de nuevo a la tabla principal. Para su implementación, en primer lugar, necesitamos una función en nuestro servicio que registre el valor modificado del *stock*, para lo cual añadimos el siguiente código al archivo Articles.js:

```
export function setStock(sku, stock) {  
  articles.forEach(article => {  
    if(article.sku === sku){  
      article.stock = stock;  
    }  
  })  
}
```

Con esta función, podemos enviar desde cualquier componente el código SKU del artículo que hay que modificar y su nuevo *stock*. Como disponemos de un componente para edición del *stock* y ya dispone del formulario, vamos a introducir dos funciones manejadoras: una para obtener del elemento “input” el nuevo *stock* y otra para confirmar la modificación e invocar el servicio. Modificamos el archivo `StockEdit.js` de la siguiente forma:

```
import React, { useState, useEffect } from 'react'
import { Link, useParams } from "react-router-dom"
import { getArticleBySKU, setStock } from '../../services/Articles';

export default function StockEdit() {

  const params = useParams();
  const [article, setArticle] = useState({});

  useEffect(() => {
    setArticle(() => getArticleBySKU(params.sku));
  }, [article])

  const handleOnChange = e => {
    const newStock = e.target.value !== '' ? Number(e.target.value) : '';
    setArticle({
      ...article,
      stock: newStock
    })
  }

  const handleOnSubmit = e => {
    e.preventDefault();
    setStock(article.sku, article.stock);
  }

  return (
    <>
      <h1>Modificar stock</h1>
      <Link to="..">
        <button>Atrás</button>
      </Link>
      <form onSubmit={handleOnSubmit}>
        <div className="row">
          <label htmlFor="brand">Marca</label>
          <input type="text"
            id="brand"
            value={article.brand}
            readOnly/>
        </div>
        <div className="row">
```

```

        <label htmlFor="model">Modelo</label>
        <input type="text"
              id="model"
              value={article.model}
              readOnly/>
      </div>
      <div className="row">
        <label htmlFor="stock">Stock</label>
        <input type="number"
              id="stock"
              value={article.stock}
              name="stock"
              onChange={handleOnChange}/>
      </div>
      <div className="row-buttons">
        <button type="submit">Guardar cambios</button>
      </div>
    </form>
  </>
)
}

```

También hemos añadido un botón para modificar los cambios y hacer persistente el nuevo *stock* del artículo, pero aún no disponemos de la lógica para que navegue a la tabla de artículos. Para ello, de nuevo en el archivo *StockEdit.js* añadimos la importación del *hook* “*useNavigation*”, lo añadimos a una constante y, desde la función asociada al evento “*submit*”, navegamos a la tabla. Modificamos el archivo con los siguientes bloques:

```

...
import { Link, useParams, useNavigate } from "react-router-dom"
...

const navigate = useNavigate();
...

const handleOnSubmit = e => {
  e.preventDefault();
  setStock(article.sku, article.stock);
  navigate('/stock-table');
}
...

```

Ahora podemos llevar cabo una modificación del *stock* de un artículo (Figura 5.1) y comprobar que se actualizará el valor y la aplicación navegará a la tabla.

APP

Inicio

Stock

Regresar a inicio

Marca	Modelo	Stock	
Apple	Iphone	10	Modificar
Xiaomi	Redmi	5	Modificar
Samsung	Galaxy	40	Modificar

Figura 5.1
Implementación de navegación
programática en una aplicación
React.

En nuestro ejemplo, podíamos haber implementado la navegación en el botón “Guardar cambios” (ya que en este caso no existe un proceso de asincronía). Como veremos a continuación, la implementación de peticiones a una API Rest será el escenario idóneo para el uso de estas técnicas.

5.2. Peticiones a una API Rest desde React

Ya disponemos en nuestra aplicación de un sistema para navegar dinámicamente por las pantallas tanto para eventos de usuario como programáticos, pero las aplicaciones SPA necesitan datos para utilizarlos en su lógica y funcionalidad. Por ello, es necesario realizar peticiones a API Rest usando el protocolo HTTP.

La implementación de estas peticiones se puede llevar a cabo mediante el uso del método “fetch” nativo de JavaScript o bien en la librería Axios. Si disponemos de una API Rest con la entidad de artículos, podemos reconvertir el servicio de artículos incorporando las peticiones a la colección de funciones. Por ejemplo, podemos comentar el código existente del archivo Articles.js y modificarlo de la siguiente forma:

```
const endPoint = 'http://localhost:3000/articles/'

export function getArticles() {
  return fetch(endPoint);
}

export function getArticleBySKU(sku) {
  return fetch(endPoint + sku)
}

export function setStock(sku, stock) {
  const options = {method: 'PUT'};
  return fetch(endPoint + sku + '/' + stock, options);
}
```


Lógicamente, si no disponemos de una API que resuelva esas peticiones en el *endpoint* especificado, nuestra aplicación no funcionará; pero nos sirve para conocer la posible sintaxis de este servicio de peticiones HTTP. Si nos fijamos también en la implementación de “fetch”, es devuelta con la palabra “return”. Se realiza de esta forma porque “fetch” devuelve una promesa y esta es resuelta en el componente que use el servicio.

De esta forma, habría también que modificar los componentes que usan estas funciones. Por ejemplo, en el componente de tabla StockTable.js cambiaríamos el siguiente código para resolver la promesa:

```
...
  useEffect(() => {
    getArticles()
      .then(data => setArticles(data.articles))
      .catch(err => console.log(err));
  }, [articles])
...
```

Esta implementación mediante promesas es altamente beneficiosa, ya que permite introducir en la gestión de la asincronía la lógica que necesitamos para que el usuario reciba información durante la petición, por ejemplo, usando un elemento gráfico como un *spinner* para que sepa que tiene que esperar la respuesta del servidor de API. En el caso de que no dispongamos de una API Rest, devolveremos el código de la aplicación al estado en el que lo dejamos antes de comenzar este apartado.



Resumen

- React permite implementar eventos de navegación programática gracias al uso del *hook* “useNavigate” de la librería React Router, que proporciona la posibilidad de establecer una ruta dentro de cualquiera de las funciones manejadoras de un componente.
- La implementación de peticiones HTTP en React se puede llevar a cabo mediante el uso del método “fetch” nativo de JavaScript o bien con la librería Axios, utilizando en ambos casos servicios para realizar las peticiones que retornarán promesas que habrá que resolver en los componentes para obtener los datos.

6. Rutas anidadas, redirecciones y *lazy loading*

A medida que se incrementa la complejidad de una aplicación, crece tanto su tamaño como el número de archivos que necesitan los proyectos React para implementar las técnicas de *Single Page Application* (SPA).

En esta unidad, vamos a aprender de manera práctica cómo implementar la modularización de código en React, un proceso que emplea la técnica *lazy loading* para la carga de los archivos a demanda del usuario.

Posteriormente aprenderemos también cómo realizar rutas anidadas que nos permitan organizar los componentes de los diferentes módulos, aportando la estructura necesaria tanto para el desarrollo como para el futuro mantenimiento evolutivo de las aplicaciones.

6.1. Técnica *lazy loading* en React Router

Hasta ahora hemos podido comprobar que toda la carga de las aplicaciones SPA se produce en su inicio, es decir, cuando el usuario realiza la primera petición “get” desde su navegador al servidor, que devuelve el archivo HTML y, casi a la par, los archivos JavaScript y CSS.

En determinados escenarios (normalmente, cuando las aplicaciones son complejas y disponen de decenas o incluso centenares de componentes), el tamaño de los archivos generados por el comando “build” puede ser excesivamente grande, a pesar de estar compilados y minificados. Esto retrasará el primer renderizado de la aplicación y, por tanto, su rendimiento inicial.

Por otra parte, una cantidad grande de archivos para componentes o servicios puede provocar un caos en los equipos de desarrollo o mantenimiento. Una solución a ambos problemas puede ser la implementación de las técnicas de *lazy loading* en nuestros proyectos React. Esta medida consiste en modularizar los componentes agrupándolos por regla de negocio y que, posteriormente, en el proceso de *build*, generen varios paquetes que serán entregados por el servidor de acuerdo con la navegación del usuario.

Para comprobar esta técnica y funcionalidad, vamos a utilizar nuestro proyecto abriendo el directorio practica1 en Visual Studio Code y vamos a modularizar un conjunto de componentes de recursos humanos. Para ello, creamos en el directorio pages un nuevo directorio rh y, dentro de este, un nuevo archivo con el nombre RHHome.js y el siguiente código:

```
import React from 'react'

export default function RHHome() {
  return (
    <>
      <h1>Recursos Humanos</h1>
    </>
  )
}
```

Siguiendo nuestro procedimiento habitual, el siguiente paso sería crear una nueva ruta en el archivo App.js; sin embargo, en este caso se trata de una ruta que tiene una sintaxis diferente, la de *lazy loading*, que conllevará dos diferencias:

- Cuando se realiza el proceso de build para este nuevo componente y los que posteriormente añadamos como módulo, se genera un nuevo archivo JavaScript.
- La carga del código de estos componentes solo se realizará cuando el usuario navegue a ellos, lo que mejorará el rendimiento de la carga inicial.

Vamos a modificar el archivo App.js de la siguiente manera para añadir *lazy loading* a este nuevo componente:

```
import React from "react";
import Home from "../pages/Home";
import StockTable from "../pages/stock/StockTable";
import {Routes, Route} from "react-router-dom";
import Menu from "../components/Menu";
import P404 from "../pages/P404";
import StockEdit from "../pages/stock/StockEdit";

const RHHome = React.lazy(() => import('../pages/rh/RHHome'));

function App() {
```

```

return (
  <>
    <Menu />
    <div className="container">
      <Routes>
        <Route path="/" element={<Home />}/>
        <Route path="stock-table" element={<StockTable />} />
        <Route path="stock-table/stock-edit/:sku" element={<StockEdit />}/>
        <Route path="/recursos-humanos/*" element={
          <React.Suspense fallback={
            <p style={{textAlign: 'center'}}>Cargando...</p>
          }>
            <RHHome />
          </React.Suspense>
        } />
        <Route path="*" element={<P404 /> } />
      </Routes>
    </div>
  </>
);
}

export default App;

```

Podemos observar que estamos utilizando como enlace al nuevo componente el método “lazy()” de la librería React para cargarlo solo cuando se utilice la ruta asociada; y, como será un proceso con asincronía, se usa el componente “Suspense” de React, que recibe un elemento para renderizar mientras se produce la carga.

A continuación, vamos a añadir un “NavLink” en el menú modificando el archivo Menu.js de la siguiente forma:

```

import React from 'react'
import { NavLink } from 'react-router-dom'

export default function Menu() {
  return (
    <nav>
      <div>APP</div>
      <div>
        <NavLink to="/"
          className={({isActive}) => isActive ? "active" : ""}>
          Inicio
        </NavLink>
        <NavLink to="/stock-table" className={({isActive}) => isActive ? "active" : ""}>
          Stock
        </NavLink>
      </div>
    </nav>
  )
}

```

```

    <NavLink to="/recursos-humanos" className={({isActive}) => isActive ? "active" : ""}>
      Recursos humanos
    </NavLink>
  </div>
</nav>
)
}

```

Una vez implementada la ruta en el menú, si pulsamos en el acceso a “Recursos Humanos”, podemos comprobar en el navegador, en la pestaña “Network” (Figura 6.1), que se produce la carga del código JavaScript en ese instante.

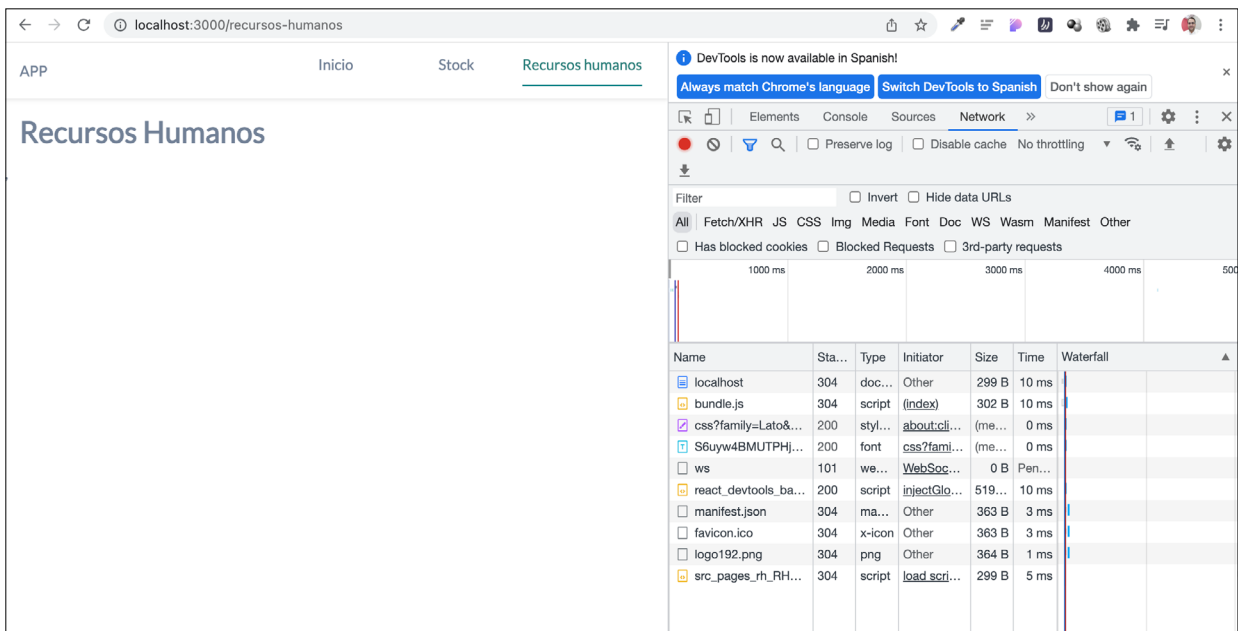


Figura 6.1
Comprobación de carga de un
componente mediante *lazy*
loading en las herramientas de
desarrollador.

La primera ventaja de la técnica de carga *lazy loading* es que nos permite aumentar el rendimiento de la aplicación, dividiendo el tamaño de los *bundles* y haciendo que los archivos se vayan descargando desde el servidor a demanda del usuario.

6.2. Modularización y rutas anidadas

En el paso anterior, y mediante el establecimiento de una ruta con carga *lazy loading*, implícitamente estamos modularizando nuestro código de la aplicación, de manera que podemos establecer el componente del archivo RHHome.js como punto de entrada de un nuevo módulo que

agrupará un conjunto de componentes para una entidad o regla de negocio; en este caso, para gestionar recursos humanos.

Sin embargo, para completar la modularización que tendremos en el directorio `rh` debemos generar por una parte los nuevos componentes y, una vez añadidos, crear rutas anidadas de manera que se asocien al mismo *bundle* final. Para comprobarlo de manera práctica, vamos a añadir en nuestro directorio `rh` un nuevo directorio llamado `employees` y, dentro de este, un nuevo archivo `EmployeeTable.js` con el siguiente código:

```
import React from 'react'

export default function EmployeesTable() {
  return (
    <>
      <h1>Empleados</h1>
    </>
  )
}
```

Dentro de este módulo de recursos humanos podríamos disponer también de otra entidad para tratamiento de datos de nóminas, así que vamos a crear otro directorio en `rh` con el nombre `payrolls`; y, dentro de este, un archivo `PayrollsTable.js` con el siguiente código:

```
import React from 'react'

export default function PayrollsTable() {
  return (
    <>
      <h1>Nóminas</h1>
    </>
  )
}
```

Como tenemos dos nuevos componentes, debemos crear dos nuevas rutas para cada uno de ellos; y también debemos mantener la carga *lazy loading*, así que modificaremos el archivo de inicio de este módulo, `RHHome.js`, de la siguiente forma:

```
import React from 'react'
import { Routes, Route, Outlet, Link } from "react-router-dom";
```

```

import EmployeesTable from './employees/EmployeesTable';
import PayrollsTable from './payrolls/PayrollsTable';

export default function RHHome() {
  return (
    <Routes>
      <Route path="/" element={<Outlet />}>
        <Route index element={<RHHomeLayout />} />
        <Route path="employees-table" element={<EmployeesTable />} />
        <Route path="payrolls-table" element={<PayrollsTable />} />
      </Route>
    </Routes>
  )
}

function RHHomeLayout() {

  return (
    <div className="container">
      <h1>Recursos Humanos</h1>
      <Link to="employees-table">
        <button>Empleados</button>
      </Link>
      <Link to="payrolls-table">
        <button>Nóminas</button>
      </Link>
    </div>
  )
}

```

El mecanismo que implementamos en esta nueva configuración de acceso al módulo es la definición en el componente “Routes” de una ruta raíz para el modulo con el valor “/”, la cual es una subruta de la definida en el archivo App.js, es decir “/recursos-humanos”. Esta ruta raíz es asociada a un componente de salida denominado “Outlet /” de la librería React Router que cargará cada una de las rutas anidadas a continuación.

Esas rutas anidadas se definen con la *prop* “index” para el propio componente, lo cual implica desarrollar en el mismo archivo RHHome.js una función que retorne el *layout* de ese componente, la cual definimos en “RRHomeLayout”. Además, como rutas anidadas se añaden también las de los nuevos componentes para nóminas y empleados.

Finalmente, en la función “RRHomeLayout” añadimos la llamada a las rutas relativas con “Link” para poder navegar a cada componente de

este módulo. Este proceso es totalmente transparente para el usuario final; así, desde el inicio del módulo (Figura 6.2) puede acceder a cada componente, pero internamente toda la carga del módulo se produce con los mismos archivos JavaScript.

Figura 6.2

Acceso a varios componentes de un módulo en React en el navegador.



La ventaja de esta modularización también se consigue a efectos de distribución de trabajo en desarrollo a los diferentes equipos y, de nuevo, en cuanto al mantenimiento evolutivo de la aplicación.



Resumen

- React permite la implementación de la carga *lazy loading* de los componentes para mejorar el rendimiento de la aplicación en soluciones complejas que podrían retrasar su inicio.
- A través de *lazy loading* y el anidado de rutas en React Router, podemos modularizar el código de nuestras aplicaciones con las ventajas significativas para las fases de desarrollo y mantenimiento evolutivo.

7. Protección de rutas

Uno de los requerimientos habituales de cualquier aplicación es un sistema de autenticación basado en roles para el acceso a las diferentes funcionalidades del sistema. Por ese motivo, las SPA necesitan un mecanismo para proteger el acceso mediante rutas.

En esta unidad, vamos a aprender de manera práctica cómo implementar la protección de rutas, para lo cual comenzaremos por simular un sistema de autenticación de usuarios.

Posteriormente aprenderemos, a partir del acceso basado en roles, cómo proteger las rutas para que solo los usuarios con un determinado rol puedan cargar en su navegador los componentes con la funcionalidad especificada.

7.1. Simulación de autenticación basada en roles

Un requisito común en muchas aplicaciones es el uso de la autenticación basada en roles, mediante la cual la parte *frontend* de la aplicación realiza una petición a una API o servicio enviando las credenciales obtenidas de un formulario de *login*; en caso de que sean correctas, se obtienen los permisos de acceso, incluyendo un rol de usuario.

Para simular ese comportamiento y poder emplear el valor de un rol en la protección de rutas, vamos a abrir nuestro proyecto practica1 en Visual Studio Code y, en primer lugar, vamos a generar en el componente raíz del archivo App.js un estado para los datos de usuario y dos funciones manejadoras para establecer unos datos de usuario con rol de recursos humanos o eliminarlos. Modificamos su código de la siguiente manera:

```
import React, {useState} from "react";
import Home from "../pages/Home";
import StockTable from "../pages/stock/StockTable";
import {Routes, Route, useNavigate} from "react-router-dom";
import Menu from "../components/Menu";
import P404 from "../pages/P404";
import StockEdit from "../pages/stock/StockEdit";

const RHHome = React.lazy(() => import('../pages/rh/RHHome'));
```

```

function App() {

  // Simulación o mock de un sistema de autenticación

  const [user, setUser] = useState(null);
  const navigate = useNavigate();

  const handleLogin = () => {
    setUser({name: 'Juan Pérez', role: 'rrhh'});
    navigate('/');
  }

  const handleLogout = () => {
    setUser(null);
    navigate('/');
  }

  return (
    <>
      <Menu user={user}
        handleLogin={handleLogin}
        handleLogout={handleLogout} />
      <div className="container">
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="stock-table" element={<StockTable />} />
          <Route path="stock-table/stock-edit/:sku" element={<StockEdit />} />
          <Route path="/recursos-humanos/*" element={<React.Suspense fallback={
            <p style={{textAlign: 'center'}}>Cargando...</p>
          }>
            <RHHome />
          </React.Suspense>
          } />
          <Route path="*" element={<P404 />} />
        </Routes>
      </div>
    </>
  );
}

export default App;

```

Como podemos observar, las funciones “handleLogin” y “handleLogout” establecen el valor de usuario y navegan al inicio de la aplicación, así que necesitamos añadirlas a algún evento que el usuario pueda llevar a cabo. Continuando como la simulación, hemos añadido ambas funciones y el estado al componente “Menú” como *props*.

Para poder utilizar estas *props* en el componente de “Menú”, las vamos a asociar a unos botones que mostraremos según el estado, para lo cual vamos a modificar el archivo Menu.js de la siguiente forma:

```
import React from 'react'
import { NavLink } from 'react-router-dom'

export default function Menu(props) {
  return (
    <nav>
      {
        props.user !== null ?
          <div>
            <button onClick={props.handleLogout}>
              Logout
            </button>
            &nbsp;{props.user.name}
          </div>
          :
          <button onClick={props.handleLogin}>
            Login
          </button>
        }
      <div>
        <NavLink to="/" className={({isActive}) => isActive ? "active" : ""}>
          Inicio
        </NavLink>
        <NavLink to="/stock-table" className={({isActive}) => isActive ? "active" : ""}>
          Stock
        </NavLink>
        <NavLink to="/recursos-humanos" className={({isActive}) => isActive ? "active" : ""}>
          Recursos humanos
        </NavLink>
      </div>
    </nav>
  )
}
```

Como podemos observar, hemos introducido ambos botones condicionando su visualización a la existencia de usuario en el estado “user” que recibimos como *prop* del componente padre “App”. Y, al poder accionar cada botón e invocar las funciones manejadoras, podemos llevar a cabo la simulación de autenticación. Si iniciamos la aplicación en el navegador, podemos comprobar su correcto funcionamiento y cómo se visualiza el usuario en la barra de menú (Figura 7.1).

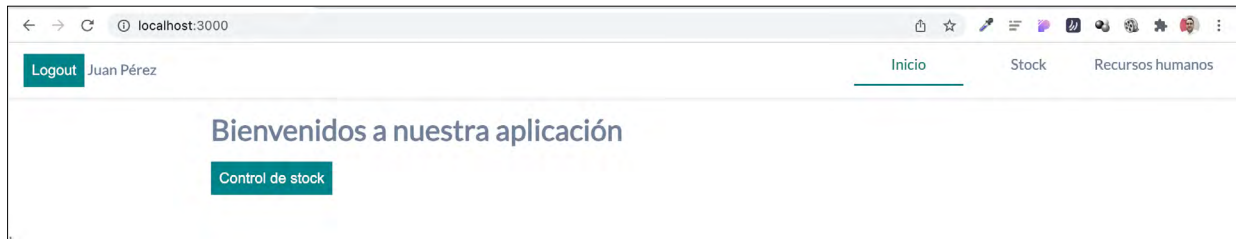


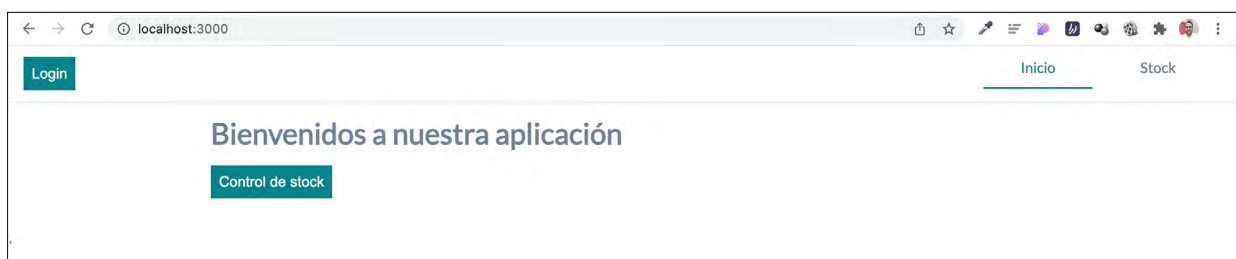
Figura 7.1
Implementación de un sistema
de simulación de autenticación
en React.

Ahora ya podemos usar el valor de la propiedad “role” del estado del usuario, por ejemplo, para limitar el acceso al módulo de “Recursos Humanos” de la aplicación. Así, de nuevo en el archivo Menu.js vamos a condicionar el enlace a ese módulo a que el usuario tenga el rol de recursos humanos. Modificamos de nuevo el código del componente “NavLink” de la siguiente forma:

```
...
{
  props?.user?.role === 'rrhh' ?
    <NavLink to="/recursos-humanos"
      className={({isActive}) => isActive ? "active" : "">
        Recursos humanos
      </NavLink>
    :
    null
}
...
```

Figura 7.2
Comprobación de la limitación
de accesos basada en roles en
el navegador..

Al grabar los cambios, como nuestro usuario está autenticado precisamente con ese rol, mantendremos la visualización del enlace y, por tanto, el acceso al módulo de recursos humanos; pero si pulsamos en el botón “Logout” simulando el cierre de sesión o cambiamos el rol en el archivo App.js, comprobaremos como el acceso desaparece (Figura 7.2).



De esta manera, limitamos en una primera aproximación los accesos e implementamos las técnicas de control de accesos basado en roles; pero,

como las aplicaciones React son *Single Page Application*, un usuario sin credenciales que conociera la ruta podría acceder al módulo. De hecho, podemos comprobarlo si completamos en el navegador `http://localhost:3000/recursos-humanos`; el componente será cargado y tendremos total acceso, aunque no hayamos iniciado sesión (Figura 7.3).

Figura 7.3

Acceso no permitido mediante ruta no protegida en el navegador.



Para impedir este comportamiento, necesitamos proteger las rutas con ayuda de la librería React Router, como veremos a continuación, una vez que ya tenemos implementada nuestra simulación de autenticación.

7.2. Protección de rutas mediante React Router

La protección de rutas se puede implementar con la librería React Router mediante varios mecanismos, pero uno de los más utilizados es crear funciones-componente de protección, denominadas “guard”, que comprueben una condición para, en caso de que no se cumplan, no cargar el componente y redirigir al usuario en su navegación.

Para comprobarlo de manera práctica, vamos a crear en el directorio `src` otro directorio denominado `guards`, en el que vamos a crear un nuevo archivo `ProtectRHRoute.js`, donde vamos a declarar y exportar una función-componente de estas características con el siguiente código:

```
import { Navigate } from "react-router-dom";

export default function ProtectRHRoute({user, children}) {

  if(user?.role !== 'rrhh') {
    return <Navigate to="/" replace/>
  }

  return children;
}
```

Podemos observar que la función-componente tiene como parámetro un objeto con los valores del usuario y “children”, una referencia en la que tendremos el componente que queremos cargar en la ruta. Así, si el rol de usuario en este caso no es el de recursos humanos, devolvemos esa referencia “children” con un componente de redirección al inicio de la aplicación.

Esta función de protección de la ruta debe protegerse ahora en la declaración de la ruta, para lo cual modificamos el archivo App.js de la siguiente forma:

```
import React, { useState } from "react";
import Home from "../pages/Home";
import StockTable from "../pages/stock/StockTable";
import { Routes, Route, useNavigate } from "react-router-dom";
import Menu from "../components/Menu";
import P404 from "../pages/P404";
import StockEdit from "../pages/stock/StockEdit";
import ProtectRHRoute from "../guards/ProtectRHRoute";

const RHHome = React.lazy(() => import('../pages/rh/RHHome'));

function App() {

  // Simulación o mock de un sistema de autenticación

  const [user, setUser] = useState(null);
  const navigate = useNavigate();

  const handleLogin = () => {
    setUser({name: 'Juan Pérez', role: 'rrhh'});
    navigate('/');
  }

  const handleLogout = () => {
    setUser(null);
    navigate('/');
  }

  return (
    <>
      <Menu user={user}
        handleLogin={handleLogin}
        handleLogout={handleLogout} />
    </>
  )
}
```



```

    <div className="container">
      <Routes>
        <Route path="/" element={<Home />}/>
        <Route path="stock-table" element={<StockTable />} />
        <Route path="stock-table/stock-edit/:sku" element={<StockEdit />}/>
        <Route path="/recursos-humanos/*" element={<ProtectRHRoute user={user}>
          <React.Suspense fallback={
            <p style={{textAlign: 'center'}}>Cargando...</p>
          }>
            <RHHome />
          </React.Suspense>
        </ProtectRHRoute>
      } />
        <Route path="*" element={<P404 /> } />
      </Routes>
    </div>
  </>
);
}

export default App;

```

Es decir, envolvemos la carga del componente “RHHome” con esta función-componente “guard” y le pasamos como *prop* los datos del usuario, de manera que redireccionará al origen si el usuario no tiene el rol definido. Si volvemos a acceder a la ruta <http://localhost:3000/recursos-humanos> en el navegador sin estar autenticados, podemos comprobar que el proyecto redirecciona al origen.

Además, podemos mejorar la experiencia de usuario creando un componente que le indique que no puede acceder a ese módulo. Vamos a añadir en el directorio `pages` un archivo `Forbidden.js` con el siguiente código:

```

import React from 'react'

export default function Forbidden() {
  return (
    <p>No dispone de permisos de acceso. Inicie sesión o contacte con su administrador</p>
  )
}

```

Seguidamente, y para este nuevo componente, añadimos una ruta en el archivo App.js insertando la siguiente línea dentro del componente “Routes”:

```
...  
    <Route path="/forbidden" element={<Forbidden /> } />  
...
```

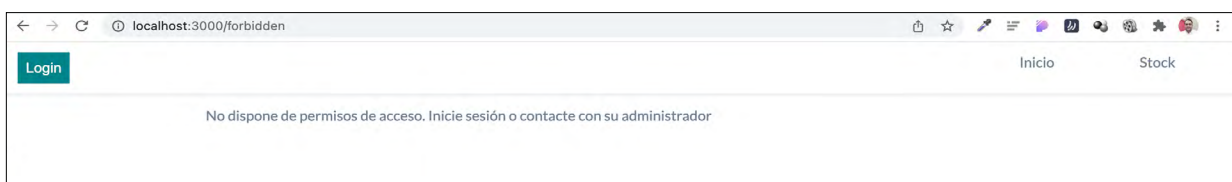
Y, una vez que tenemos la ruta, podemos modificar la función-componente “guard” del archivo ProtectRHRoute.js para que, si no dispone del rol de recursos humanos, cargue en pantalla este componente con información sobre el acceso. Modificamos el archivo con el siguiente código:

```
import { Navigate } from "react-router-dom";  
  
export default function ProtectRHRoute({user, children}) {  
  if(user?.role !== 'rrhh') {  
    return <Navigate to="/forbidden" replace />  
  }  
  
  return children;  
}
```

Figura 7.4

Renderizado de componente de acceso restringido en el navegador.

Ahora, al volver a intentar entrar mediante la ruta del navegador al módulo de recursos humanos sin estar autenticado, comprobamos que se carga el nuevo componente:



Con esta protección de rutas, aumentamos la robustez de nuestro proyecto en React manteniendo las ventajas del modelo SPA y las técnicas de *client-side rendering*.



- Uno de los requisitos habituales de las aplicaciones React es un sistema de autenticación basado en roles para el acceso a las diferentes funcionalidades del sistema; pero si las rutas no están protegidas, los usuarios sin permisos podrán acceder mediante la barra de navegación.
- El mecanismo de protección de rutas en React utiliza la librería React Router para redirigir la carga de componentes cuando no cumplan unas condiciones relacionadas con los roles de usuario que son evaluadas en un tipo de función-componente denominado “guard”.

Índice

Esquema de contenido	3
Introducción	5
1. Routing y navegación en SPA	7
1.1. <i>Single-page applications</i>	7
1.2. React Router	9
Resumen	11
2. Instalación y configuración de React Router	12
2.1. Instalación de React Router	12
2.2. Centralización de datos con servicios	13
Resumen	18
3. “Routes”, “Route” y “Link”	19
3.1. Componentes “Routes” y “Route”	19
3.2. Componente “Link”	20
3.3. Componente “navlink” y rutas no definidas	23
Resumen	28
4. Rutas absolutas y relativas, y rutas con parámetros	29
4.1. Rutas absolutas y relativas en React Router	29
4.2. Rutas con parámetros	32
Resumen	36
5. Navegación programática	37
5.1. Navegación programática en React Router	37
5.2. Peticiones a una API Rest desde React	40
Resumen	42
6. Rutas anidadas, redirecciones y lazy loading	43
6.1. Técnica <i>lazy loading</i> en React Router	43
6.2. Modularización y rutas anidadas	46
Resumen	50

7. Protección de rutas	51
7.1. Simulación de autenticación basada en roles	51
7.2. Protección de rutas mediante React Router	55
Resumen	59