Certified Tech Developer
The Ultimate Degree

Front End III

Creando las páginas de la aplicación

Vamos a construir una aplicación que tendrá cinco páginas:

- Nuestra página principal se llamará /home.
- Tendremos una página llamada /main, y esta tendrá anidadas dos páginas: una llamada /blog y otra /vlog.
- Escogeremos /about como nombre de la página sobre nosotros.

Pero ¿qué pasaría si la página que se está intentando visitar no existe en nuestro sitio web?, ¿o si algún link ha quedado desactualizado? Para estos casos debemos contemplar visualizar una página especial. Las páginas son componentes renderizados en el momento, de forma muy similar a como venimos haciendo hasta ahora con React, así que simplemente crearemos un componente llamado NotFound para estos casos.

Escribamos los componentes para estas páginas. Serán todos componentes funcionales, no hace falta crear componentes de clase:

Home.jsx

```
import React from "react";
import styles from "./home.scss";

const Home = () => {
   return (
       <div className={styles.content}>
           <h1>Home</h1>
       </div>
   );
};
export default Home;
```

El componente <Home /> es muy simple, solo retorna un div con un elemento de encabezado de sección h1 que dice “Home”.

Main.jsx

```
import React from "react";
import styles from "./main.scss";

const Main = () => {
   return (
       <div className={styles.content}>
           <h1>Main</h1>
       </div>
   );
};
export default Main;
```

<Main /> por ahora será muy simple, solo retornará un elemento de encabezado de sección h1 que dice “Main”, todo dentro de un div. Más adelante trabajaremos con este componente para que anide a los componentes <Blog /> y <Vlog /> definidos más abajo.

Blog.jsx

```
import React from "react";
import styles from "./blog.scss";

const Blog = () => {
   return (
       <div className={styles.content}>
           <h1>Blog</h1>
       </div>
   );
};
export default Blog;
```

El componente <Blog /> también es sencillo, retorna un div con un h1 adentro que dice “Blog”. Este componente estará anidado dentro del componente <Main />.

Vlog.jsx

```
import React from "react";
import styles from "./vlog.scss";

const Vlog = () => {
   return (
       <div className={styles.content}>
           <h1>Vlog</h1>
       </div>
   );
};
export default Vlog;
```

Similar al componente <Blog />, el componente <Vlog /> retorna un div con un h1 adentro que dice “Vlog”. Este componente estará también anidado dentro del componente <Main />.

About.jsx

```
import React from "react";
import styles from "./about.scss";

const About = () => {
   return (
       <div className={styles.content}>
           <h1>About</h1>
       </div>
   );
};
export default About;
```

<About /> es similar a todos los demás. En el texto dice “About”.

NotFound.jsx

```
import React from "react";
import styles from "./notFound.scss";


const NotFound = () => {
   return (
       <div className={styles.content}>
           <h1>404 - Not found</h1>
       </div>
   );
};
export default NotFound;
```

<NotFound /> es nuestro componente para los casos en que la URL no coincide con el path de ninguno de nuestros componentes Route.

Hasta acá tenemos seis componentes: cinco de ellos se renderizarán en tres páginas porque tendremos dos páginas anidadas, y uno extra se renderizará como página de error 404.

Enrutando la aplicación

Ahora escribiremos otro componente funcional, al que llamaremos <App />, que será nuestro componente principal y contendrá a la aplicación.

En <App /> importamos la biblioteca React, luego nuestros componentes que funcionarán como páginas, y finalmente los componentes de React Router: BrowserRouter, Route, Switch y Link.

Veamos la estructura inicial de nuestro **componente <App />**:

App.jsx

```
import React from 'react';
import Home from "./Home.jsx";
import Main from "./Main-parameters-404.jsx";
import About from "./About.jsx";
import NotFound from "./NotFound.jsx";
import { BrowserRouter, Route, Switch, Link } from "react-router-dom";

function App () {
       return (
           <BrowserRouter>
               {null}
```

```
            </BrowserRouter>
        );
}
export default App;
```

React Router requiere que metamos la lógica de la aplicación dentro de un par <BrowserRouter></BrowserRouter>. Recordemos que, aunque podríamos trabajar con HashRouter, a menos que haya alguna razón muy particular, lo mejor es trabajar con BrowserRouter.

Ahora veamos la lógica de la aplicación

Cada componente Route tiene un atributo llamado path, y un componente children (porque está dentro). Cuando el atributo path coincide con la URL de la página, se renderiza el componente children. Para nuestro caso tendremos esto:

```
        <BrowserRouter>
            <Route exact path="/"><Home /></Route>
            <Route path="/main"><Main /></Route>
            <Route path="/about"><About /></Route>
        </BrowserRouter>
```

Ahora, recordemos qué sucede cuando una URL coincide con el path de algún componente Route: cualquier componente encerrado por Route será renderizado, es decir, que podríamos renderizar más de un componente al mismo tiempo.

Por esta razón, en el primer ítem hemos utilizado el atributo exact para indicarle a Route que la URL debe coincidir exactamente con el path /. De no utilizar exact, se renderizará al mismo tiempo tanto <Home /> como el componente de la página que estemos visitando porque todos empiezan con "/".

Route siempre renderizará algo, ya sea un componente, si la URL coincide con su path, o null.

Pero a veces, exact no es suficiente. Veremos un ejemplo de esto cuando manejemos el caso de página no encontrada más adelante.

Para evitar renderizar más de un componente bajo cualquier circunstancia, existe el componente Switch. Switch garantiza que solo el primer componente cuyo path coincida con la URL será renderizado, mientras que todos los demás cuyos paths coincidan también con la URL —si los hubiera— serán ignorados.

La forma de usar Switch es envolviendo a los componentes Route dentro de un par <Switch></Switch>. Una vez hecho esto, nuestro componente App es ahora completamente funcional:

App.jsx

```
import React from 'react';
import Home from "./Home.jsx";
import Main from "./Main-parameters-404.jsx";
import About from "./About.jsx";
import NotFound from "./NotFound.jsx";
import { BrowserRouter, Route, Switch, Link } from "react-router-dom";

function App () {
       return (
           <BrowserRouter>
               <Switch>
                   <Route exact path="/"> <Home /> </Route>
                   <Route path="/main"> <Main /> </Route>
                   <Route path="/about"> <About /> </Route>
               </Switch>
           </BrowserRouter>
       );
}
export default App;
```

Agregando las rutas de navegación

Ahora vamos a proveer navegación con enlaces (links) a las páginas que creamos. Utilizaremos el componente **Link** —y su atributo **to**— de React Router. Link es muy conveniente para una SPA porque no dispara una recarga de página como hacen las anclas (<a>).

Creemos una lista desordenada (
), ítems de listas (
), y el componente Link de React Router para crear los vínculos a las páginas:

```
<ul>
    <li>
        <Link to="/">Home</Link>
    </li>
    <li>
        <Link to="/main">Main</Link>
    </li>
    <li>
        <Link to="/about">About</Link>
    </li>
</ul>
```

El código de App, con links de navegación, luce así:

App.jsx

```
import React from 'react';

import Home from "./Home.jsx";

import Main from "./Main-parameters-404.jsx";

import About from "./About.jsx";

import NotFound from "./NotFound.jsx";

import { BrowserRouter, Route, Switch, Link } from "react-router-dom";


function App () {

       return (

           <BrowserRouter>

               <ul>

                   <li>

                       <Link to="/">Home</Link>

                   </li>

                   <li>

                       <Link to="/main">Main</Link>

                   </li>

                   <li>

                       <Link to="/about">About</Link>

                   </li>

               </ul>
```

```
                <Switch>
                    <Route exact path="/"><Home /></Route>
                    <Route path="/main"><Main /></Route>
                    <Route path="/about"><About /></Route>
                </Switch>
            </BrowserRouter>
        );
}
export default App;
```

Páginas anidadas y parámetros

Recordemos que con el enrutamiento dinámico ya no necesitamos un archivo con rutas predeterminadas, y que podemos usar componentes Route y Link según lo necesitemos en cualquier componente, siempre y cuando el componente esté dentro del árbol de BrouserRouter y ya sea parte de una ruta.

Ahora veremos cómo pasar parámetros a nuestras rutas. Supongamos que tenemos varios artículos para blogs y varios tópicos de videos. Podríamos tener una lista de nuestros contenidos con metainformación —como nombre, ID, descripción, etc.—. La forma en que accedamos a esta metainformación no es relevante para nuestro objetivo (podríamos leer la información desde una base de datos, desde un archivo JSON en el back end, un archivo XML, un archivo CSV, etc.), pero supongamos que disponemos de una forma de almacenar y consultar esta metainformación organizada como objetos JSON desde el back end.

En nuestro ejemplo, queremos que dentro de Main se visualicen Blog y Vlog como páginas, es decir, que cada uno tenga su URL. Además, que tanto Blog como Vlog acepten IDs como parámetros de la URL para poder conseguir la metainformación.

Entonces, dentro de Main —para indicarle a React Router que renderice las nuevas rutas—, usaremos Route (encerrando a los componentes Blog y Vlog) con los atributos paths apuntando a las URLs respectivas, tal como hemos venido haciendo, solo que ahora estas URLs están anidadas y, además, contienen los IDs como parámetros:

```
          <Route path={"/main/vlog/:id/:resourceId"}> <Vlog /> </Route>

          <Route path={"/main/blog/:id/:resourceId"}> <Blog /> </Route>
```

En el código de arriba estamos pasando dos identificadores en cada path: uno llamado **id** para el artículo (o tópico) y otro llamado **resourceId** para el recurso. Esa es la forma en que pasamos parámetros a una ruta con React Router.

Ahora, agreguemos algunos enlaces a las páginas /blog y /vlog dentro del componente Main:

```
<ul>
    <li key="R1-A1V">
        <Link to={"/main/vlog/A1V/R1-A1V"}>
            Recurso uno del artículo uno para video
        </Link>
    </li>
    <li key="R2-A1V">
        <Link to={"/main/vlog/A1V/R2-A1V"}>
            Recurso dos del artículo uno para video
        </Link>
    </li>
```

```
</ul>
```

Y:

```
<ul>
    <li key="R1-A1B">
        <Link to={"/main/blog/A1B/R1-A1B"}>
            Recurso uno del artículo uno para blog
        </Link>
    </li>
    <li key="R2-A1B">
        <Link to={"/main/blog/A1B/R2-A1B"}>
            Recurso dos del artículo uno para blog
        </Link>
    </li>
</ul>
```

Nuestro código luce así por ahora:

Main.jsx

```
import React from 'react';
import styles from "./main.scss";
import Blog from "./Blog.jsx";
import Vlog from "./Vlog.jsx";
import { Route, Switch, Link } from "react-router-dom";

function Main () {

   return (
       <div className={styles.content}>
           <h1>Main</h1>
           <h2>Vlog</h2>
           <ul>
               <li key="R1-A1V">
                   <Link to={"/main/vlog/A1V/R1-A1V"}>
                       Recurso uno del artículo uno para video
                   </Link>
               </li>
               <li key="R2-A1V">
```

```
                    <Link to={"/main/vlog/A1V/R2-A1V"}>
                        Recurso dos del artículo uno para video
                    </Link>
                </li>
            </ul>
            <h2>Blog</h2>
            <ul>
                <li key="R1-A1B">
                    <Link to={"/main/blog/A1B/R1-A1B"}>
                        Recurso uno del artículo uno para blog
                    </Link>
                </li>
                <li key="R2-A1B">
                    <Link to={"/main/blog/A1B/R2-A1B"}>
                        Recurso dos del artículo uno para blog
                    </Link>
                </li>
            </ul>
            <hr />
                <Route path={"/main/vlog/:id/:resourceId"}> <Vlog />
</Route>
                <Route path={"/main/blog/:id/:resourceId"}> <Blog />
</Route>
```

```
        </div>
    );
}
export default Main;
```

Ahora, cuando entremos a la página /main, se renderizará el componente Main y debajo tendremos los enlaces a las páginas /blog y /vlog. Al hacer clic en cualquiera de los enlaces, veremos el componente respectivo y su URL en la barra de direcciones.

Notemos que por ahora no estamos usando Switch. Esto producirá un renderizado incorrecto en algunas circunstancias, como veremos en la sección final.

Veamos los cambios necesarios en los componentes Vlog y Blog. A estos componentes les llegarán los parámetros desde la URL. Estos parámetros —que llamamos id y resourceId— los obtendremos usando la función useParams de react-router-dom. Veamos el código:

Vlog.jsx

```
import React from 'react';
import { useParams } from 'react-router-dom';
import styles from "./vlog.scss";

const Vlog = () => {
    let { id, resourceId } = useParams();
```

```
    return (
        <div className={styles.content}>
            <h1>Vlog</h1>
            <p>Tópico {id}</p>
            <p>Recurso {resourceId}</p>
        </div>
    );
};
export default Vlog;
```

Blog.jsx

```
import React from 'react';
import { useParams } from 'react-router-dom';
import styles from "./blog.scss";

const Blog = () => {
    let { id, resourceId } = useParams();

    return (
        <div className={styles.content}>
            <h1>Blog</h1>
```

```
            <p>Artículo {id} </p>
            <p>Recurso {resourceId}</p>
        </div>
    );
};
export default Blog;
```

La forma como obtenemos los IDs es la siguiente: la función useParams provista por react-router-dom nos devuelve un objeto con los pares clave-valor que hayamos pasado en la URL. Usando asignación por desestructuración (en inglés: destructuring), encerramos entre llaves los nombres de los elementos de ese objeto que deseamos "sacar" del objeto. Así, la sentencia **let { id, resourceId } = useParams()** significa: "obtener id y resourceId del objeto que devuelve useParams()". Una vez esos parámetros están en el componente, podemos hacer lo que necesitemos con ellos. En este caso, solamente los estamos renderizando entre elementos .

Páginas inexistentes

¿Qué sucederá cuando alguien trate de acceder a una página que no exista dentro de nuestro sitio web? Las personas que visitan nuestro sitio web, cuando tratan de acceder a una página inexistente, esperan ver algo que les indique que lo que están buscando no existe. Estas son las páginas 404, o 404 Error. Conocidas así porque el status de respuesta del servidor tiene el código HTTP 404.

Donde primero necesitaremos manejar ese escenario es en el componente App.jsx:

```
<Switch>
    <Route exact path="/"> <Home /> </Route>
    <Route path="/main"> <Main /> </Route>
```

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

      <Route path="*"> <NotFound /> </Route>

  </Switch>
```

Agregando al final **<Route path="*"><NotFound/></Route>**, garantizamos que cualquier ruta que no haya coincidido con algún path será delegada al componente que llamamos NotFound.

Habíamos dicho que manejar el escenario de página no encontrada era un caso en el que el uso de Switch era esencial. Sin Switch, la última ruta del código de arriba se renderizará siempre, incluso para el path con el atributo exact, porque usa un comodín (*wildcard*) que coincide con cualquier URL. La única forma de evitar que el componente NotFound se muestre siempre es usando Switch.

Insertemos un enlace roto a propósito:

```
            <li>

                <Link to="/brokenlink">Link roto</Link>

             </li>
```

Nuestro componente App luce así:

App.jsx

```
import React from 'react';
import Home from "./Home.jsx";
import Main from "./Main-parameters-404.jsx";
import About from "./About.jsx";
import NotFound from "./NotFound.jsx";
import { BrowserRouter, Route, Switch, Link } from "react-router-dom";

function App() {

   return (
       <BrowserRouter>
           <ul>
               <li>
                   <Link to="/">Home</Link>
               </li>
               <li>
                   <Link to="/main">Main</Link>
               </li>
               <li>
                   <Link to="/about">About</Link>
```

```
                </li>
                <li>
                    <Link to="/brokenlink">Link roto</Link>
                </li>
            </ul>

            <Switch>
                <Route exact path="/"> <Home /> </Route>
                <Route path="/main"> <Main /> </Route>
                <Route path="/about"> <About /> </Route>
                <Route path="*"> <NotFound /> </Route>
            </Switch>
        </BrowserRouter>
    );
}
export default App;
```

Páginas inexistentes en rutas anidadas

Por último, veamos qué sucede con las páginas inexistentes dentro una ruta anidada.

Esencialmente el escenario es similar al de rutas inexistentes en el componente principal. La diferencia es dónde metemos la lógica, y que ahora la ruta está anidada.

Empecemos por agregar un par de enlaces rotos en App.jsx:

```
            <li>
                <Link to="/main/vlog/brokenlink">
                    Link roto a video del Vlog con un solo parámetro
                </Link>
            </li>
            <li>
                <Link to="/main/blog/bro/ken/link">
                Link roto a artículo del Blog con más de dos parámetros
                </Link>
            </li>
```

Este es nuestro componente App.jsx final:

App.jsx

```
import React from 'react';
import Home from "./Home.jsx";
import Main from "./Main-parameters-404.jsx";
import About from "./About.jsx";
import NotFound from "./NotFound.jsx";
```

```
import { BrowserRouter, Route, Switch, Link } from "react-router-dom";


function App() {


   return (
       <BrowserRouter>
           <ul>
               <li>
                   <Link to="/">Home</Link>
               </li>
               <li>
                   <Link to="/main">Main</Link>
               </li>
               <li>
                   <Link to="/main/vlog/A1V/R1-A1V">
                       Primer video del Vlog
                   </Link>
               </li>
               <li>
                   <Link to="/main/blog/A1B/R1-A1B">
                       Primer artículo del Blog
                   </Link>
```

```
                </li>
                <li>
                    <Link to="/main/vlog/brokenlink">
                        Link roto a video del Vlog con un solo parámetro
                    </Link>
                </li>
                <li>
                    <Link to="/main/blog/bro/ken/link">
                        Link roto a artículo del Blog con más de dos
parámetros
                    </Link>
                </li>
                <li>
                    <Link to="/about">About</Link>
                </li>
                <li>
                    <Link to="/brokenlink">Link roto</Link>
                </li>
            </ul>

            <Switch>
                <Route exact path="/"> <Home /> </Route>
                <Route path="/main"> <Main /> </Route>
```

```
                <Route path="/about"> <About /> </Route>
                <Route path="*"> <NotFound /> </Route>
            </Switch>
        </BrowserRouter>
    );
}
export default App;
```

Analicemos los dos enlaces nuevos. El enlace **<Link to="/main/vlog/brokenlink">** apunta a la URL anidada /main/vlog con el parámetro brokenlink. Por su parte, el enlace **<Link to="/main/blog/bro/ken/link">** apunta la URL /main/blog con tres parámetros: bro, ken y link.

Recordemos cómo es la lógica de Main actualmente:

```
            <Route path="/main/vlog/:id/:resourceId"> <Vlog /> </Route>
            <Route path="/main/blog/:id/:resourceId"> <Blog /> </Route>
```

Tal y como está ahora, Main renderizará incorrectamente si se usan los enlaces rotos, por varias razones que explicaremos a continuación.

La URL **/main/vlog/brokenlink** del primer enlace roto no es manejada por ningún Route de Vlog, porque Vlog espera dos identificadores y estamos pasando uno solo. Por su parte, la URL **/main/blog/bro/ken/link** tampoco es manejada por ningún Route dentro de Blog, ya que espera dos identificadores y está recibiendo tres.

Entonces, necesitamos indicarle a React Router que —tanto para Vlog como para Blog— utilice NotFound para las rutas que no existen. Para esto necesitamos especificar que las URLs /main/vlog/* y /main/blog/* instanciarán a NotFound. De hecho, cualquier URL dentro de /main debería instanciar a NotFound si no coincide con los paths de Vlog y

Blog:

```
<Route path="/main/*"> <NotFound /> </Route>
```

Esto resuelve parcialmente el problema porque ahora nuestros enlaces funcionales también renderizarán a NotFound, ya que sus URLs coinciden con las rutas comodines. Incluso el enlace /main/blog/bro/ken/link también funcionará pasando los parámetros bro y ken al componente Blog.

La forma de solucionar esto es usando tanto Switch como el atributo exact. Ambos son necesarios porque el atributo exact garantiza que las rutas con el número incorrecto de parámetros no serán tomadas en los componentes y, por lo tanto, pasarán al final, donde serán tomadas por los componentes Route con los paths comodines. Por otro lado, Switch garantiza que cuando los enlaces sí existan, no se mostrarán los componentes NotFound.

Agreguemos un último enlace roto, dentro de Main:

```
<li key="R2-A1V">
     <Link to="/main/brokenlink">
         Recurso inexistente para video
     </Link>
 </li>
```

Nuestro componente Main finalmente queda así:

Main.jsx

```
```

```
import React from 'react';
import styles from "./main.scss";
import Blog from "./Blog.jsx";
import Vlog from "./Vlog.jsx";
import NotFound from "./NotFound.jsx";
import { blogArticles, vlogTopics } from "./articles.JSON";
import { Route, Switch, Link } from "react-router-dom";


function Main () {


   return (
       <div className={styles.content}>
           <h1>Main</h1>
           <h2>Vlog</h2>
           <ul>
               <li key="R1-A1V">
                   <Link to="/main/vlog/A1V/R1-A1V">
                       Recurso uno del artículo uno para video
                   </Link>
               </li>
               <li key="R2-A1V">
                   <Link to="/main/vlog/A1V/R2-A1V">
```

```
                Recurso dos del artículo uno para video
            </Link>
        </li>
        <li key="R2-A1V">
            <Link to="/main/brokenlink">
                Recurso inexistente para video
            </Link>
        </li>
    </ul>
    <h2>Blog</h2>
    <ul>
        <li key="R1-A1B">
            <Link to="/main/blog/A1B/R1-A1B">
                Recurso uno del artículo uno para blog
            </Link>
        </li>
        <li key="R2-A1B">
            <Link to="/main/blog/A1B/R2-A1B">
                Recurso dos del artículo uno para blog
            </Link>
        </li>
    </ul>
```

```
            <hr />
            <Switch>
                <Route exact path="/main/vlog/:id/:resourceId"> <Vlog />
</Route>
                <Route exact path="/main/blog/:id/:resourceId"> <Blog />
</Route>
                <Route path="/main/*"> <NotFound /> </Route>
            </Switch>
        </div>
    );
}
export default Main;
```

Por último, es importante tener en cuenta que si tenemos un enlace con una URL que sí existe, y con el número correcto de parámetros, React Router renderizará el componente independientemente de que los parámetros sean o no válidos. No hay forma de que React Router nos proteja de este tipo de errores. Cada componente debe implementar su lógica de validación de entradas del usuario y manejo de errores para lidiar con estos casos.

¡Hasta la próxima!