Front End III

useFetch

Una de las partes fundamentales de cualquier aplicación moderna consiste en la posibilidad de comunicarse con una API para obtener la información que desea mostrarse en la pantalla según el caso. En un componente React, es común realizar este tipo de solicitudes luego de que dicho componente se monte.

Para cumplir este objetivo, hacemos uso del Hook **useEffect** de una manera similar a la siguiente:

```
useEffect(
    () => {
        fetch(https://rickandmortyapi.com/api/character/)
        .then(response => {
            if (!response.ok) {
                throw Error(response.statusText);
            }
            return response.json()
        })
        .then(data => setElements(data))
        .catch(
            error => {}
        )
    },
    []
);
```

Si tenemos varios componentes que interactúan con una API, es posible que este bloque de código se repita en cada uno de ellos casi sin modificaciones (quizás cambiando el

endpoint o algún parámetro, pero nada más). Es por ello, que para este caso tiene sentido crear un Hook personalizado basado en la API fetch para lograr código más declarativo y reutilizable.

Escribamos una aplicación que consulta la API de Rick and Morty y muestra una lista de los personajes de la serie. Al igual que en el ejemplo anterior, lo haremos primero sin usar un Hook personalizado.

Supongamos que en el componente **App** tenemos los componentes para el encabezado, pie de página y listado:

```
import React from "react";
import { Encabezado } from './Encabezado.jsx';
import { PieDePagina } from './PieDePagina.jsx';
import { Lista } from './Lista.jsx';
import "./app.scss";

const App = () => {

   return (
       <div className="app">
           <Encabezado />

           <PieDePagina />
       </div>
   );
};
export default App;
```

Ahora veamos cómo obtener y mostrar el listado de los personajes.

La consulta al endpoint se debe hacer una vez que el componente **App** se monta. Desde que se inicia la consulta hasta que la misma termina, nuestro componente pasa por varios estados. Primero tendremos un estado de espera, seguido por un estado de éxito o error. Necesitamos poder manejar estos estados dentro de la aplicación, para indicar a la persona en qué momento nos encontramos.

Recordemos que para manejar el estado en un componente funcional necesitamos usar el Hook useState. Como queremos una lista de elementos, definamos nuestro estado de esta manera:

```
const [elements, setElements] = useState(null);
```

La consulta al endpoint con fetch podría ser entonces así:

```
fetch(https://rickandmortyapi.com/api/character/)
 .then(response => {
     if (!response.ok) {
         throw Error(response.statusText);
     }
     return response.json()
 })
 .then(data => setElements(data))
 .catch(
     error => {}
 )
```

Como la consulta al endpoint se debe hacer una vez que el componente **App** se haya montado, el código de arriba debe ir dentro de **useEffect.** Eso garantizará que el componente se habrá renderizado para cuando se ejecute ese Hook, y evitará que se

consulte al endpoint cada vez que se actualice el componente, por ejemplo, si alguna prop cambia. Le pasaremos un array vacío al Hook useEffect como segundo argumento:

```
  useEffect(
      () => {
          fetch(https://rickandmortyapi.com/api/character/)
          .then(response => {
              if (!response.ok) {
                  throw Error(response.statusText);
              }
              return response.json()
          })
          .then(data => setElements(data))
          .catch(
              error => {}
          )
      },
      []
  );
```

El código completo luce así:

```
import React, { useState, useEffect } from "react";
import { Lista } from './Lista.jsx';
import { Encabezado } from './Encabezado.jsx';
import { PieDePagina } from './PieDePagina.jsx';
import "./app.scss";

const App = () => {
```

```
const [elements, setElements] = useState(null);

useEffect(
    () => {
        fetch(https://rickandmortyapi.com/api/character/)
        .then(response => {
            if (!response.ok) {
                throw Error(response.statusText);
            }
            return response.json()
        })
        .then(data => setElements(data))
        .catch(
            error => {}
        )
    },
    []
);

const Component = (
    elements !== null ?
     <Lista elementos={data} />
       : <p>Loading...</p>
);

return (
    <div className="app">
        <Encabezado />
```

```
            { Component }
            <PieDePagina />
        </div>
    );
};

export default App;
```

Ahora veamos cómo podemos restarle complejidad a este componente, encapsulando y reutilizando el código que efectúa el pedido a la API y maneja el estado de la respuesta.

Lo primero que podemos hacer es crear un archivo con el nombre del Hook (**useFetch.jsx**), mover los Hooks **useState** y **useEffect** a ese archivo, y definir la función de esta manera:

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

export function useFetch (URL, options) {

    const [data, setData] = useState(null);

    useEffect(
        () => {
            fetch(URL, options)
            .then(response => {
                if (!response.ok) {
                    throw Error(response.statusText);
                }
                return response.json()
            })
```

```
            .then(data => setData(data))
        },
        []
    );

    return { data };
}
```

Hagamos algo más con respecto al manejo de estado. Agreguemos un objeto con los pares clave/valor para los posibles estados:

```
export const statuses = {
    LOADING: "Loading...",
    OK: "OK",
    ERROR: "Error"
};
```

Agreguemos una variable de estado para manejar el estatus:

```
    const [status, setStatus] = useState(null);
```

Y, dentro de **useEffect**, agreguemos código que maneja el estatus de la consulta y un catch al final de la cadena de promesas:

```
    useEffect(
        () => {
            setStatus(statuses.LOADING);

            fetch(URL, options)
```

```
            .then(response => {
                if (!response.ok) {
                    throw Error(response.statusText);
                }
                return response.json()
            })
            .then(data => setData(data))
            .then(() => setStatus(statuses.OK))
            .catch(
                error => setStatus(statuses.ERROR)
            )
        },
        []
    );
```

Ahora podemos retornar no solo la data recuperada del endpoint sino el estatus de la consulta:

```
    return { data, status };
```

El código completo luce así:

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

export const statuses = {
    LOADING: "Loading...",
    OK: "OK",
    ERROR: "Error"
};
```

```
export function useFetch (URL, options) {

   const [data, setData] = useState(null);
   const [status, setStatus] = useState(null);

   useEffect(
       () => {
           setStatus(statuses.LOADING);

           fetch(URL, options)
           .then(response => {
               if (!response.ok) {
                   throw Error(response.statusText);
               }
               return response.json()
           })
           .then(data => setData(data))
           .then(() => setStatus(statuses.OK))
           .catch(
               error => setStatus(statuses.ERROR)
           )
       },
       []
   );

   return { data, status };
}
```

El código de **useFetch** encapsula todo lo que se necesita para consultar cualquier API y manejar el estado de la consulta. De esta forma, ahora puede ser utilizado por cualquier componente y no únicamente por el componente **App**.

Para utilizarlo, el componente **App** lo consume de esta manera:

- Lo importa desde el archivo **useFetch.jsx**:

```
import { useFetch, statuses } from "./useFetch.jsx";
```

- Invoca al Hook personalizado **useFetch**:

```
  const { data, status } = useFetch(URL_API, {});
```

- Y según el estado de la consulta puede decidir qué renderizar:

```
  const Component = (
      status !== statuses.ERROR ?
      <>
          { status === statuses.LOADING && <p>Loading...</p> }
          { data && <Lista elementos={data} /> }
      </>
      : <p>Network Error</p>
  );
```

Estamos renderizando etiquetas ** ** en los casos de carga y error, pero bien podrían ser componentes o animaciones.

Cuando el status no es error, renderizamos el mensaje de que estamos cargando los datos o la lista de elementos renderizada desde el componente **Lista**.

El componente **App** final luce así:

```
import React from 'react';
import { Lista } from './Lista.jsx';
import { Encabezado } from "./Encabezado.jsx";
import { PieDePagina } from "./PieDePagina.jsx";
import { useFetch, statuses } from "./useFetch.jsx";
import "./app.scss";

const App = () => {
   const URL_API = 'https://rickandmortyapi.com/api/character/';

   const { data, status } = useFetch(URL_API, {});

   const Component = (
       status !== statuses.ERROR ?
       <>
           { status === statuses.LOADING && <p>Loading...</p> }
           { data && <Lista elementos={data} /> }
       </>
       : <p>Network Error</p>
   );

   return (
       <div className="app">
           <Encabezado />
           { Component }
           <PieDePagina />
```

```
        </div>
    );
};

export default App;
```

Ahora que hemos extraído la lógica a un Hook personalizado, podremos hacer uso del mismo en cualquier componente en donde necesitemos realizar una petición a la API. De esta manera, evitamos repeticiones innecesarias al tiempo que aumentamos la flexibilidad y escalabilidad de nuestra aplicación.