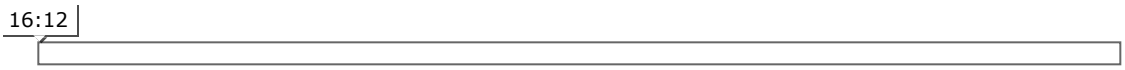
Watch: ⚠️ **Peligros de doblar el fetch**



Puedes encontrar el código de este vídeo en GitHub.

Con lo hablado en el anterior vídeo siempre damos por supuesto que va a existir un servicio que contacte con la API externa, pero no siempre es el caso.

¿Qué podemos hacer si nos encontramos código que realiza un `fetch` contra la API directamente desde un componente? Lo ideal sería añadir un servicio y mover la llamada ahí, pero no es buena idea refactorizar el código sin tener tests para asegurarnos de no romper nada.

Podemos mockear `fetch` en nuestros tests de la siguiente manera:

```
beforeAll(() => jest.spyOn(window, "fetch"));
```

```
it("should list posts", async () => {
  window.fetch.mockResolvedValueOnce({
    ok: true,
    json: async () =>  [
      // ...
    ]
  });
  // ...
});
```

Pero vemos varios problemas: tenemos que devolver esa función `json`, atándonos a la implementación. Tampoco podemos diferenciar las llamadas que se hacen, y tendríamos que usar `mockImplementation` para poder devolver un resultado u otro en función del path, de si la llamada es POST o GET...

Para solucionar estos problemas podemos usar la librería MSW. Esta nos permite interceptar las llamadas que haga `fetch` a nivel de *network*, y así no tenemos que doblar nada. Después de instalar la librería, creamos un fichero `server.js` en la carpeta `mocks`:

```
import { setupServer } from "msw/node";

import { handlers } from "./handlers";

export const server = setupServer(...handlers);
```

La variable `handlers` es un array que exportamos de otro archivo, que listará las posibles llamadas a la api, con su método y path. Por ejemplo, el handler para interceptar la llamada GET a la api sería:

```
rest.post(
  "https://jsonplaceholder.typicode.com/posts",
  (req, res, ctx) => {
    return res(ctx.status(200), ctx.json([
      // contenido de nuestros posts
    ]));
  }
)
```

Como vamos a querer cambiar el contenido de la response según el caso de nuestro test, exportaremos funciones para que nos devuelvan el handler con unos valores por defecto, que podremos modificar en nuestro test, como veremos más adelante. El archivo `handlers.js` quedaría de la siguiente manera:

```
import { rest } from "msw";

export function getPosts(response = [], status = 200) {
  return rest.get(
    "https://jsonplaceholder.typicode.com/posts",
    (req, res, ctx) => {
      return res(ctx.status(status), ctx.json(response));
    }
  );
}

export function createPost(response = {}, status = 200) {
  return rest.post(
    "https://jsonplaceholder.typicode.com/posts",
    (req, res, ctx) => {
      return res(ctx.status(status), ctx.json(response));
    }
  );
}

export const handlers = [createPost(), getPosts()];
```

Para terminar el setup de nuestro servidor fake, lo levantaremos al iniciar la ejecución de los tests y resetearemos entre cada test. Podemos hacerlo en el archivo `setupTests.js` para evitar añadirlo en cada test:

```
import "@testing-library/jest-dom";
import { server } from "./mocks/server";

beforeAll(() => server.listen());

afterEach(() => server.resetHandlers());

afterAll(() => server.close());
```

Ahora ya estamos preparados para empezar a testear. Para modificar la respuesta que hemos definido en `handlers.js`, que era un array vacío, podemos modificarlo en el test con `server.use`, pasando la función que exportamos anteriormente con el array de posts que queremos validar que se renderiza correctamente:

```
describe("when user loads the component", () => {
  it("should list posts", async () => {
    const posts = [
      {
        id: 1,
        title: "My post",
        body: "lorem ipsum",
      },
    ];

    server.use(getPosts(posts));

    render(<Posts />);

    const post = await screen.findByRole("heading", { name: "My
post" });

    expect(post).toBeInTheDocument();
  });
});
```

Vemos que MSW es una muy buena opción especialmente en aplicaciones legacy que tengan el fetch acoplado en el código, pero si cambia la api devolviendo un formato distinto, o cambiando el path de un endpoint, tendremos que modificar el archivo `handler.js`. Por eso pensamos que, si las circunstancias lo permiten, es más recomendable mover la lógica relacionada con la api a un servicio, para obtener un código más desacoplado y poder testear sin necesidad de librerías externas.

Links relacionados:

- MSW