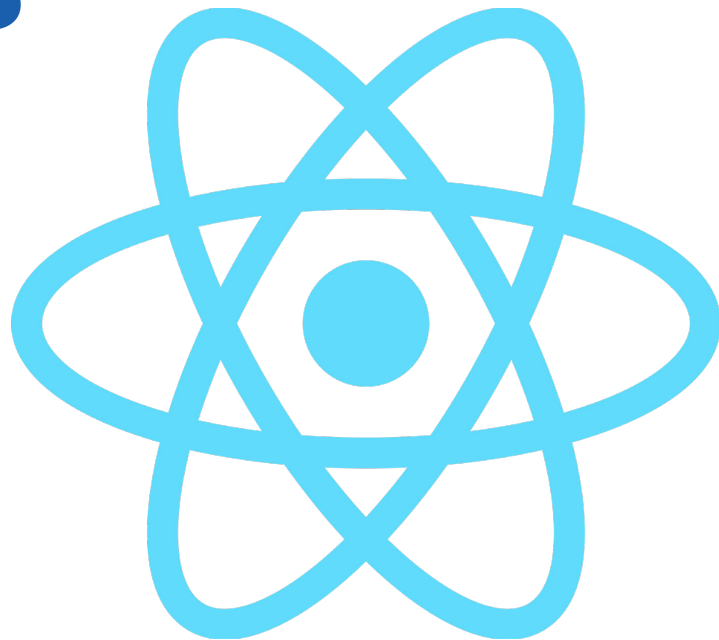


Clase 8

REACT



ALBAÑILESDIGITALES

Developing your new career



VeriDas

Temario

1. React
2. React Components and Props
3. Composing React Components and Stateful React Components
4. Events, conditional rendering
5. Lists and forms
6. React Router I
7. React Router II
8. **Debugging and Testing**

Testing:

Puedes probar un componente de React similar a como pruebas otro código JavaScript.

Hay varias formas de probar un componente React, la mayoría se agrupan en dos categorías:

- Renderizado del árbol de componentes en un entorno de prueba simplificado y comprobando sus salidas.
- Ejecutando la aplicación completa en un entorno de prueba más realista utilizando un navegador web (más conocido como pruebas “end-to-end”).

Herramientas recomendadas

- **Jest**: Es una biblioteca de JavaScript para ejecución de pruebas que permite acceder al DOM mediante [jsdom](#). Aunque jsdom solo se aproxima a como realmente los navegadores web trabajan, usualmente es suficiente para probar componentes de React. Jest brinda una gran velocidad de iteración combinada con potentes funcionalidades como simular [módulos](#) y [temporizadores](#), esto permite tener mayor control sobre cómo se ejecuta el código.
- **Biblioteca de pruebas para React**: biblioteca de utilidades que te ayudan a probar componentes React sin depender de los detalles de su implementación. Este enfoque simplifica la refactorización y también lo empuja hacia las mejores prácticas de accesibilidad, aunque no proporciona una forma de renderizar “superficialmente” un componente sin sus hijos, Jest te permite hacerlo gracias a su funcionalidad para [simular](#).

Testing: Configuración/Limpieza

Para cada prueba, usualmente queremos renderizar nuestro árbol de React en un elemento del DOM que esté asociado a `document`. Esto es importante para poder recibir eventos del DOM. Cuando la prueba termina, queremos “limpiar” y desmontar el árbol de `document`.

Una forma común de hacerlo es usar un par de bloques **beforeEach** y **afterEach**, eventos que se ejecutan al inicio y fin de la ejecución del test respectivamente. En nuestro caso podemos usar la función **cleanup** dentro del bloque **afterEach**:

```
import { cleanup } from "@testing-library/react";

afterEach(() => {
  cleanup();
});
```

Testing: Renderizado

Una de las pruebas comunes es probar si un componente se renderiza correctamente con unas props dadas. Vamos a considerar el componente **Hello** que renderiza un mensaje basado en una prop:

```
// src/hello.js
export default function Hello(props) {
  if (props.name) {
    return <h1>Hello, {props.name}!</h1>;
  } else {
    return <span>Hey, stranger</span>;
  }
}
```

Tenemos que probar que pasándole una prop **name** nos renderiza “**Hello, [name]**” y en otro caso nos renderiza “**Hey, stranger**”.

Podemos ejecutar los tests con: **npm test**

```
// hello.test.js
import {
  cleanup, render, screen
} from "@testing-library/react";
import Hello from "./hello";

afterEach(() => {
  cleanup();
});

test("renderiza sin nombre", () => {
  render(<Hello />);
  const textElement = screen.getByText(/Hey, stranger/i);
  expect(textElement).toBeInTheDocument();
});

test("renderiza con nombre", () => {
  render(<Hello name="Jenny" />);
  const textElement = screen.getByText(/Hello, Jenny!/i);
  expect(textElement).toBeInTheDocument();
});
```

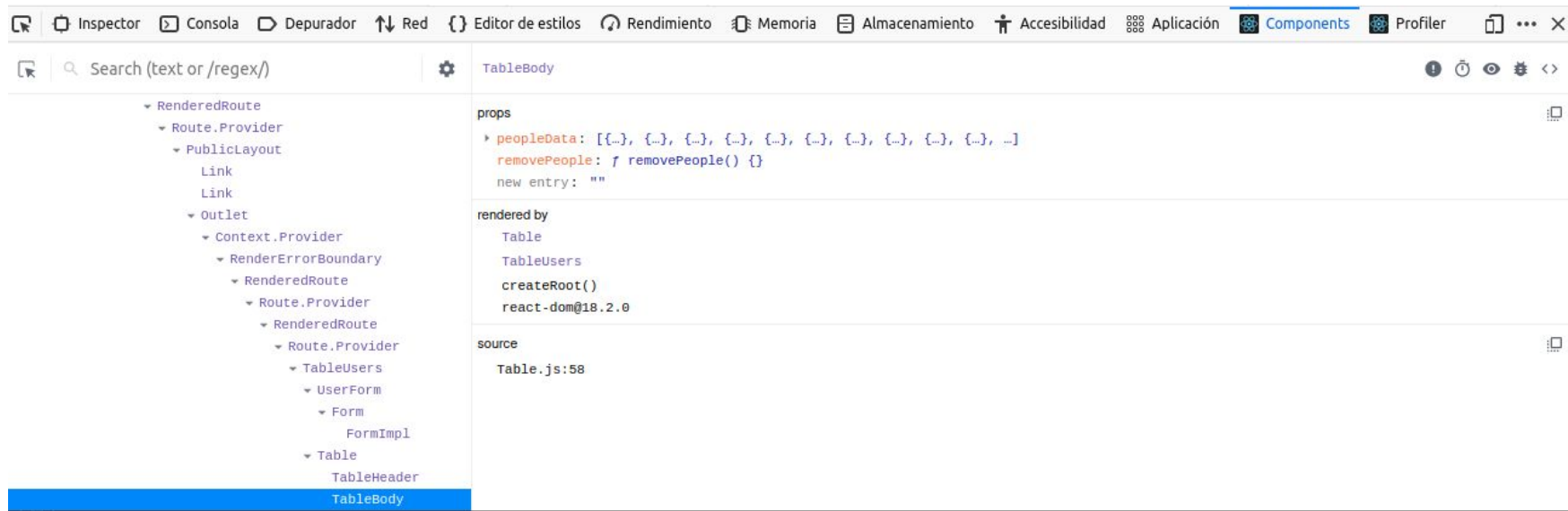
```
PASS src/hello.test.js
PASS src/App.test.js
```

```
Test Suites: 2 passed, 2 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        2.092 s
Ran all test suites related to changed files.
```

Debugging: React Dev Tools

Tal y como hemos visto en lecciones anteriores, existen plugins para inspeccionar los componentes de React en el navegador, son los denominados [React Dev Tools](#).

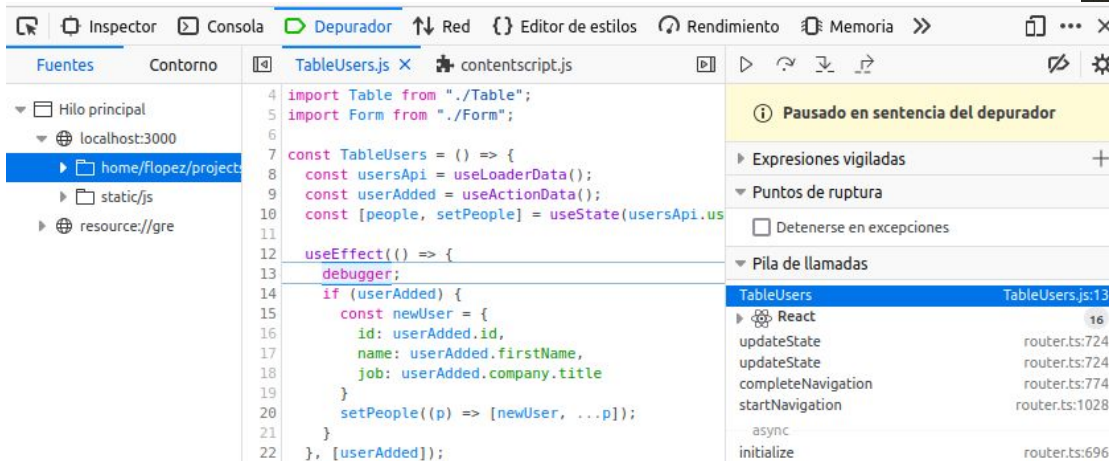
Tras instalarlo, podemos acceder a los componentes a través de las Dev Tools pulsando la tecla **F12** o haciendo clic derecho en la página y pulsando en **Inspeccionar**.



Debugging: debugger

Las React Dev Tools nos permiten visualizar el estado, las propiedades y el árbol de componentes, sin embargo, a veces es necesario poder acceder a las funciones e inspeccionar línea a línea para visualizar los datos de variables y funciones, entre otros.

Para ello, existe el comando **debugger**, que al colocarlo en el código, el navegador parará la ejecución en ese punto.



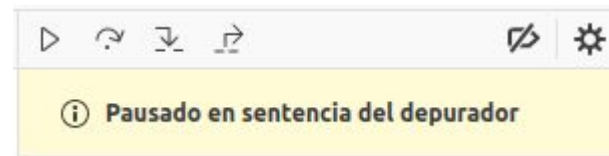
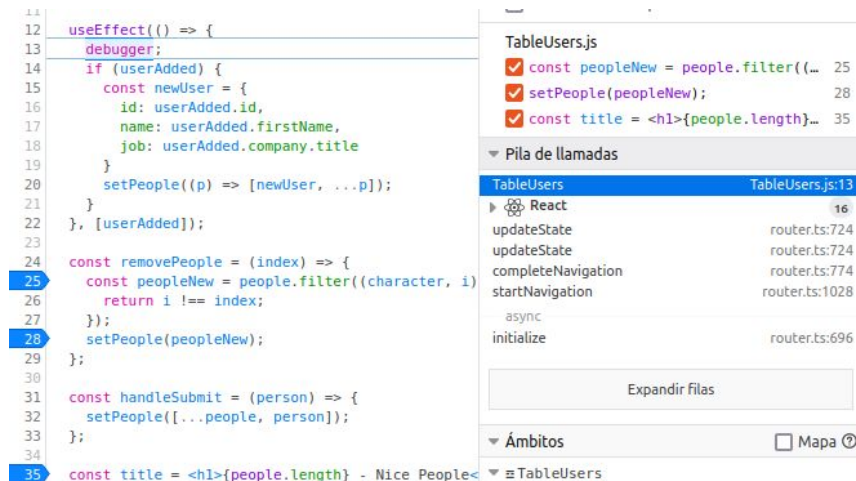
```

1 // src/routes/TableUsers.js
2 import { useState, useEffect } from "react";
3 import { useLoaderData, useActionData } from "react-router-dom";
4 import Table from "./Table";
5 import Form from "./Form";
6
7 const TableUsers = () => {
8   const usersApi = useLoaderData();
9   const userAdded = useActionData();
10   const [people, setPeople] = useState(usersApi.users);
11
12   useEffect(() => {
13     debugger;
14     if (userAdded) {
15       const newUser = {
16         id: userAdded.id,
17         name: userAdded.firstName,
18         job: userAdded.company.title
19       };
20       setPeople((p) => [newUser, ...p]);
21     }
22   }, [userAdded]);
  
```






Debugging: breakpoints

Además del comando **debugger**, podemos añadir otros puntos de parada en la ejecución, se llaman **breakpoints**. La funcionalidad es la misma que debugger, el navegador para la ejecución del código para poder inspeccionar los elementos de nuestro código.

Para activar un breakpoint basta con clicar en el número de línea, en el ejemplo hay un breakpoint en la línea 25, 28 y 35.



Para controlar la ejecución del código, disponemos de 5 botones:

-  Ejecutar hasta el siguiente breakpoint.
-  Continuar ejecutando la siguiente línea de código.
-  Entrar en la línea de código actual, por ejemplo entrar a una función o método.
-  Continuar con la ejecución del código línea a línea.
-  Desactivar o Activar todos los breakpoints.

Ejercicios

1. Replicar el código de clase:
 - App my-testing-app



ALBAÑILES DIGITALES

Developing your new career



VeriDas