Ing. Luis Guillermo Molero Suárez

Visión general de pruebas

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”).

Esta sección de la documentación está enfocada en estrategias de prueba para el primer caso. Mientras las pruebas de tipo “end-to-end” pueden ser muy útiles para prever regresiones a flujos de trabajos importantes, estas pruebas no están relacionadas con los componentes React particularmente y están fuera del alcance de esta sección.

Concesiones

Cuando estás eligiendo las herramientas para realizar las pruebas, vale la pena considerar algunas Concesiones:

- Velocidad de iteración vs Entorno realista: Algunas herramientas ofrecen un ciclo de retroalimentación muy rápido entre hacer un cambio y ver el resultado, pero no modelan el comportamiento del navegador con precisión. Otras herramientas pueden usar un entorno de navegador real, pero reducen la velocidad de iteración y son menos confiables en un servidor de integración continua.
- Cuanto abarcar: Cuando pruebas componentes la diferencia entre Prueba Unitaria y Prueba de Integración puede ser borrosa. ¿Si estás probando un formulario, se deben probar los botones del formulario en esta prueba? ¿O el componente del botón debe tener su propia suit de pruebas? ¿Debería la refactorización del botón afectar el resultado de las pruebas del formulario?

Diferentes respuestas pueden funcionar para diferentes equipos y diferentes productos.

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 es una 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.

Recetas sobre pruebas

Patrones comunes de pruebas para componentes de React.

Acá, utilizaremos principalmente componentes de función. Sin embargo, estas estrategias de prueba no dependen de detalles de implementación y funcionan igualmente en componentes de clase.

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 de manera tal que siempre ejecuten y separen los efectos de la prueba misma:

```
import { unmountComponentAtNode } from "react-dom";

let container = null;
beforeEach(() => {
  // configurar un elemento del DOM como objetivo del renderizado
  container = document.createElement("div");
  document.body.appendChild(container);
});
```

```
afterEach(() => {
  // limpieza al salir
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});
```

Puedes usar un patrón diferente, pero ten en cuenta que queremos ejecutar la limpieza incluso si falla una prueba. De otro modo, las pruebas pueden tener "fugas" y una prueba puede cambiar el comportamiento de otra prueba. Eso dificulta la tarea de depurarlas.

act()

Cuando se escriben pruebas de interfaz de usuario, tareas como el renderizado, los eventos de usuario, o la obtención de datos pueden considerarse "unidades" de interacción con la interfaz de usuario. react-dom/test-utils proporciona una utilidad llamada act() que asegura que todas las actualizaciones relacionadas con estas "unidades" hayan sido procesadas y aplicadas al DOM antes de que hagas cualquier afirmación:

```
act(() => {
  // renderizar componentes
});
// hacer afirmaciones
```

Esto ayuda a que tus pruebas se ejecutan de una manera más cercana a la experiencia de un usuario real que usa tu aplicación. El resto de estos ejemplos utilizan act() para asegurar estas garantías.
Utilizar act() directamente puede parecerte demasiado verboso. Para evitar algo de este código repetitivo, puedes usar una biblioteca como React Testing Library, cuyas utilidades están envueltas con act().

Nota:

El nombre act viene del patrón Arrange-Act-Assert.

Renderizado

Comúnmente, te gustaría probar si un componente se renderiza correctamente para unas props dadas. Considera un componente simple que renderiza un mensaje basado en una prop:

```
// hello.js

import React from "react";

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

Podemos escribir una prueba para este componente:

```
// hello.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Hello from "./hello";

let container = null;
beforeEach(() => {
  // configurar un elemento del DOM como objetivo del renderizado
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // limpieza al salir
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("renderiza con o sin nombre", () => {
```

```
  act(() => {    render(<Hello />, container);  });
expect(container.textContent).toBe("Hey, stranger");
  act(() => {
    render(<Hello name="Jenny" />, container);
  });
  expect(container.textContent).toBe("Hello, Jenny!");

  act(() => {
    render(<Hello name="Margaret" />, container);
  });
  expect(container.textContent).toBe("Hello, Margaret!");
});
```

Obtención de datos

En lugar de llamar APIs reales en todas tus pruebas, puedes simular peticiones con datos falsos. Simular peticiones con datos “falsos” previene pruebas con resultados impredecibles debido a un backend no disponible y permite ejecutarlas más rápidamente. Nota: aún puedes querer ejecutar un subconjunto de pruebas usando un framework de “extremo a extremo” que te diga que toda tu aplicación está funcionando correctamente en su conjunto.

```
// user.js

import React, { useState, useEffect } from "react";

export default function User(props) {
  const [user, setUser] = useState(null);

  async function fetchUserData(id) {
    const response = await fetch("/" + id);
    setUser(await response.json());
  }

  useEffect(() => {
    fetchUserData(props.id);
  }, [props.id]);

  if (!user) {
    return "loading...";
  }

  return (
```

```
    <details>
      <summary>{user.name}</summary>
      <strong>{user.age}</strong> years old
      <br />
      lives in {user.address}
    </details>
  );
}
```

Podemos escribir pruebas para este componente:

```
// user.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";
import User from "./user";

let container = null;
beforeEach(() => {
  // configurar un elemento del DOM como objetivo del renderizado
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // limpieza al salir
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("renderiza datos de usuario", async () => {
  const fakeUser = {    name: "Joni Baez",    age: "32",
address: "123, Charming Avenue"  };  jest.spyOn(global,
"fetch").mockImplementation(() =>    Promise.resolve({      json:
() => Promise.resolve(fakeUser)    })  );
  // Usa la versión asíncrona de act para aplicar promesas
resueltas
  await act(async () => {
    render(<User id="123" />, container);
  });
```

```
expect(container.querySelector("summary").textContent).toBe(fakeUs
er.name);

expect(container.querySelector("strong").textContent).toBe(fakeUse
r.age);
  expect(container.textContent).toContain(fakeUser.address);

  // elimina la simulación para asegurar que las pruebas estén
completamente aisladas  global.fetch.mockRestore();});
```

Simulación de módulos

Algunos módulos puede que no funcionen bien dentro de un entorno de pruebas, o puede que no sean esenciales para la prueba misma. Simular estos módulos con reemplazos “de imitación” puede hacer más fácil la escritura de pruebas para tu propio código.
Considera un componente Contacto que incluye un componente GoogleMap de terceros:

```
// map.js

import React from "react";

import { LoadScript, GoogleMap } from "react-google-maps";
export default function Map(props) {
  return (
    <LoadScript id="script-loader"
googleMapsApiKey="YOUR_API_KEY">
      <GoogleMap id="example-map" center={props.center} />
    </LoadScript>
  );
}

// contact.js

import React from "react";
import Map from "./map";

export default function Contact(props) {
  return (
    <div>
      <address>
        Contact {props.name} via{" "}
        <a data-testid="email" href={"mailto:" + props.email}>
```

```
          email
        </a>
        or on their <a data-testid="site" href={props.site}>
          website
        </a>.
      </address>
      <Map center={props.center} />
    </div>
  );
}
```

Si no queremos cargar este componente en nuestras pruebas, podemos simular la dependencia misma como un componente simulado y correr nuestras pruebas:

```
// contact.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Contact from "./contact";
import MockedMap from "./map";

jest.mock("./map", () => {  return function DummyMap(props) {
return (      <div data-testid="map">
{props.center.lat}:{props.center.long}      </div>    );  };});
let container = null;
beforeEach(() => {
  // configurar un elemento del DOM como objetivo del renderizado
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // limpieza al salir
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("debe renderizar información de contacto", () => {
  const center = { lat: 0, long: 0 };
  act(() => {
```

```
    render(
      <Contact
        name="Joni Baez"
        email="test@example.com"
        site="http://test.com"
        center={center}
      />,
      container
    );
  });

  expect(
    container.querySelector("[data-
testid='email']").getAttribute("href")
  ).toEqual("mailto:test@example.com");

  expect(
    container.querySelector('[data-
testid="site"]').getAttribute("href")
  ).toEqual("http://test.com");

  expect(container.querySelector('[data-
testid="map"]').textContent).toEqual(
    "0:0"
  );
});
```

Eventos

Recomendamos enviar eventos reales del DOM en elementos del DOM y luego hacer afirmaciones sobre el resultado. Considera el componente Toggle:

```
// toggle.js

import React, { useState } from "react";

export default function Toggle(props) {
  const [state, setState] = useState(false);
  return (
    <button
      onClick={() => {
        setState(previousState => !previousState);
        props.onChange(!state);
```

```
      }}
      data-testid="toggle"
    >
      {state === true ? "Turn off" : "Turn on"}
    </button>
  );
}
```

Podríamos escribir pruebas para este componente:

```
// toggle.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Toggle from "./toggle";

let container = null;
beforeEach(() => {
  // configurar un elemento del DOM como objetivo del renderizado
  container = document.createElement("div");
  document.body.appendChild(container);});
afterEach(() => {
  // limpiar al salir
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("cambia el valor cuando se le hace clic", () => {
  const onChange = jest.fn();
  act(() => {
    render(<Toggle onChange={onChange} />, container);
  });

  // encuentra el elemento  del botón y dispara algunos clics en
él
  const button = document.querySelector("[data-testid=toggle]");
  expect(button.innerHTML).toBe("Turn on");

  act(() => {
```

```
    button.dispatchEvent(new MouseEvent("click", { bubbles: true
}));
  });
  expect(onChange).toHaveBeenCalledTimes(1);
  expect(button.innerHTML).toBe("Turn off");

  act(() => {
    for (let i = 0; i < 5; i++) {
      button.dispatchEvent(new MouseEvent("click", { bubbles: true
}));
    }  });

  expect(onChange).toHaveBeenCalledTimes(6);
  expect(button.innerHTML).toBe("Turn on");
});
```

Diferentes eventos del DOM y sus propiedades se describen en MDN. Nota que necesitas pasar { bubbles: true } en cada evento que creas para que llegue al agente de escucha (listener) de React, porque React automáticamente delega los eventos a la raíz.

Nota:

React Testing Library ofrece una utilidad más concisa para disparar eventos.

Temporizadores

Tu código puede usar funciones basadas en temporizadores como setTimeout para programar más trabajo en el futuro. En este ejemplo, un panel de selección múltiple espera por una selección y avanza, terminando si una selección no se ha hecho en 5 segundos:

```
// card.js

import React, { useEffect } from "react";

export default function Card(props) {
  useEffect(() => {
    const timeoutID = setTimeout(() => {
      props.onSelect(null);
    }, 5000);
    return () => {
      clearTimeout(timeoutID);
    };
```

```
  }, [props.onSelect]);

  return [1, 2, 3, 4].map(choice => (
    <button
      key={choice}
      data-testid={choice}
      onClick={() => props.onSelect(choice)}
    >
      {choice}
    </button>
  ));
}
```

Podemos escribir pruebas para este componente aprovechando las simulaciones de temporizadores de Jest, y probando los diferentes estados en que puede estar.

```
// card.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Card from "./card";
let container = null;
beforeEach(() => {
  // configurar un elemento del DOM como objetivo del renderizado
  container = document.createElement("div");
  document.body.appendChild(container);
  jest.useFakeTimers();
});

afterEach(() => {
  // limpiar al salir
  unmountComponentAtNode(container);
  container.remove();
  container = null;
  jest.useRealTimers();
});

it("debe seleccionar null después de acabarse el tiempo", () => {
  const onSelect = jest.fn();
  act(() => {
    render(<Card onSelect={onSelect} />, container);
```

```
  });

  // adelantarse 100ms en el tiempo  act(() => {
    jest.advanceTimersByTime(100);
  });
  expect(onSelect).not.toHaveBeenCalled();

  // y luego adelantarse 5 segundos  act(() => {
    jest.advanceTimersByTime(5000);
  });
  expect(onSelect).toHaveBeenCalledWith(null);
});

it("debe limpiar al eliminarse", () => {
  const onSelect = jest.fn();
  act(() => {
    render(<Card onSelect={onSelect} />, container);
  });
  act(() => {
    jest.advanceTimersByTime(100);
  });
  expect(onSelect).not.toHaveBeenCalled();

  // desmonta la aplicación
  act(() => {
    render(null, container);
  });
  act(() => {
    jest.advanceTimersByTime(5000);
  });
  expect(onSelect).not.toHaveBeenCalled();
});

it("debe aceptar selecciones", () => {
  const onSelect = jest.fn();
  act(() => {
    render(<Card onSelect={onSelect} />, container);
  });

  act(() => {
    container
      .querySelector("[data-testid='2']")
      .dispatchEvent(new MouseEvent("click", { bubbles: true }));
```

```
  });

  expect(onSelect).toHaveBeenCalledWith(2);
});
```

Puedes escribir temporizadores falsos solo para algunas pruebas. Arriba, los habilitamos llamando a jest.useFakeTimers(). La mayor ventaja que proporcionan es que tu prueba no tiene que esperar realmente cinco segundos para ejecutarse, y tampoco hay necesidad de hacer el código del componente más complejo solo para probarlo.

Pruebas de instantánea

Frameworks como Jest también permiten guardar “instantáneas” de los datos con toMatchSnapshot / toMatchInlineSnapshot. Con estas, podemos “guardar” el resultado del componente renderizado y asegurarnos que un cambio a él tiene que hacerse explícitamente como un cambio a la instantánea.

En este ejemplo, renderizamos un componente y formateamos el HTML renderizado con el paquete pretty, antes de guardarlo como una instantánea en línea:

```
// hello.test.js, again

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";
import pretty from "pretty";

import Hello from "./hello";

let container = null;
beforeEach(() => {
  // configurar un elemento del DOM como objetivo del renderizado
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // limpiar al salir
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});
```

```
it("debe renderizar un saludo", () => {
  act(() => {
    render(<Hello />, container);
  });

  expect(    pretty(container.innerHTML)
).toMatchInlineSnapshot(); /* ... jest lo llena automáticamente
... */
  act(() => {
    render(<Hello name="Jenny" />, container);
  });

  expect(
    pretty(container.innerHTML)
  ).toMatchInlineSnapshot(); /* ... jest lo llena automáticamente
... */

  act(() => {
    render(<Hello name="Margaret" />, container);
  });

  expect(
    pretty(container.innerHTML)
  ).toMatchInlineSnapshot(); /* ... jest lo llena automáticamente
... */
});
```

Generalmente es mejor hacer afirmaciones más específicas que usar instantáneas. Este tipo de pruebas incluyen detalles de implementación, por lo que pueden romperse con facilidad y los equipos pueden desensibilizarse ante las fallas de las instantáneas. Simular algunos componentes hijos de manera selectiva puede ayudar a reducir el tamaño de las instantáneas y mantenerlas más legibles para las revisiones de código.

Múltiples renderizadores

En casos poco comunes, puedes ejecutar una prueba en un componente que utiliza múltiples renderizadores. Por ejemplo, puedes ejecutar pruebas de instantánea en un componente con react-test-renderer, que internamente utiliza ReactDOM.render dentro de un componente hijo para renderizar algún contenido. En este escenario, puedes envolver las actualizaciones con los act()s correspondientes a sus renderizadores.

```
import { act as domAct } from "react-dom/test-utils";
import { act as testAct, create } from "react-test-renderer";
// ...
let root;
domAct(() => {
  testAct(() => {
    root = create(<App />);
  });
});
expect(root).toMatchSnapshot();
```

Entornos de Pruebas

Bibliotecas de ejecución de pruebas

- Bibliotecas para ejecución de pruebas como Jest, mocha, ava permiten escribir conjuntos de pruebas en JavasScript regular y correrlas como parte de tu proceso de desarrollo. Adicionalmente, los suits de pruebas son ejecutados como parte de integraciones continuas.
- Jest es altamente compatible con proyectos de React, soportando características como modulos simulados y temporizadores, y soporte jsdom. Si usas Create React App, Jest ya está incluido para usar fácilmente con una configuracion por defecto útil.
- Bibliotecas como mocha funcionan bien en un entorno de navegador real, y puede ayudar con pruebas que necesiten de ello explícitamente.
- Las pruebas “end-to-end” son usadas para probar flujos más largos a través de múltiples páginas y que requieren una configuración diferente.

Simulando una superficie de renderizado

Las pruebas usualmente son ejecutadas en un entorno sin acceso a una superficie de renderizado real como un navegador. Para estos entornos, recomendamos simular el navegador usando jsdom, una implementación de navegador que se ejecuta sobre Node.js.

En la mayoría de los casos, jsdom se comporta como lo haría un navegador normal, pero no tiene características como navegación y layout. Aún así es útil para la mayoría de las pruebas de componentes web, al correr más rápido por no tener que iniciar un navegador para cada prueba. También se ejecuta en el mismo proceso que tus pruebas, así que puedes escribir código para examinar y comprobar resultados en el DOM renderizado.

Tal como en un navegador real, jsdom nos permite simular interacciones del usuario; las pruebas pueden llamar eventos en nodos del DOM, y entonces observar y comprobar los efectos resultantes de estas acciones (ejemplo).
Una gran parte de pruebas a la interfaz gráfica pueden ser escritas con la configuración descrita más arriba: usando Jest como biblioteca de prueba, renderizando en jsdom y con interacciones especificas del usuario como una secuencia de eventos del navegador, iniciadas por la función act() (ejemplo). Por ejemplo, muchas de las propias pruebas de React están escritas con esta combinación.

Si estas escribiendo una biblioteca que prueba principalmente un comportamiento específico del navegador y requiere comportamiento nativo del navegador como el layout o inputs reales, puedes usar un framework como mocha.

En un entorno donde no puedes simular el DOM (por ejemplo, probando componentes de React Native en Node.js), podrías usar simuladores de eventos para simular interacciones con elementos. De manera alternativa, también puedes usar
el helper fireEvent de @testing-library/react-native.
Frameworks como Cypress, puppeteer y webdriver son útiles para ejecutar pruebas “end-to-end”.

Simulando funciones

Cuando se están escribiendo pruebas, nos gustaría simular partes de nuestro código que no tienen un equivalente en nuestro entorno de pruebas (por ejemplo revisar el estado de navigator.onLine dentro de Node.js). Las pruebas también podrían espiar algunas funciones y observar como otras partes de la prueba interactúan con ellas. Es entonces útil ser capaz de simular selectivamente estas funciones con versiones más amigables para las pruebas.

Esto es especialmente útil en los llamados para obtener datos. Es preferible usar datos “falsos” para estas pruebas para evitar la lentitud y lo engorroso de llamados a endpoints API reales (ejemplo). Esto ayuda a que las pruebas sean predecibles. Bibliotecas como Jest y sinon, entre otras, soportan funciones simuladas. Para pruebas “end-to-end”, simular una red puede ser más complicado, pero también podrías querer probar los endpoints API reales en ellos igualmente.

Simulando módulos

Algunos componentes tienen dependencias de módulos que quizás no funcionen bien en entornos de prueba, o no son necesarios para nuestras pruebas. Puede ser útil simular de manera selectiva estos módulos con reemplazos adecuados (example).
En Node.js, bibliotecas como Jest soportan la simulación de módulos. También podrías usar bibliotecas como mock-require.

Simulando temporizadores

Hay componentes que pudiesen estar usando funciones basadas en el tiempo como setTimeout, setInterval, or Date.now. En entornos de prueba, puede ser de ayuda simular estas funciones con reemplazos que te permitan “avanzar” manualmente el tiempo. ¡Esto es excelente para asegurar que tus pruebas se ejecuten rápidamente! Las pruebas que dependen de temporizadores aún podrian ser resueltas en orden, pero mas rápido (ejemplo). La mayoría de los frameworks, incluyendo Jest, sinon y lolex, te permiten simular temporizadores en tus pruebas.

Algunas veces, podrías no querer simular los temporizadores. Por ejemplo, quizás estás probando una animación, o interactuando con un endpoint que es sensitivo al tiempo (como una API limitadora de peticiones). Bibliotecas con simuladores de temporizadores te permite habilitar y deshabilitarlas en base a pruebas o suits, de forma que tú puedas elegir como estas pruebas serán ejecutadas.

Pruebas “end-to-end”

Las pruebas “end-to-end” son útiles para flujos más largos, especialmente si estos son críticos para tu negocio (como los pagos o registros). Para estas pruebas, probablemente quisieras probar cómo un navegador real renderiza toda la aplicación, solicita datos de un endpoint API real, usa sesiones y cookies, navega entre diferentes enlaces. Podrías también querer hacer comprobaciones no solamente en el estado del DOM sino también en los datos que usa (por ejemplo, para verificar si las actualizaciones persisten en la base de datos).

En este escenario, podrías usar un framework como Cypress o una biblioteca como puppeteer de forma que puedas navegar entre diferentes rutas y

comprobar los efectos no solo del navegador si no potencialmente del backend también.