User Interaction Testing

user-event ile gerçekçi etkileşim simülasyonu, tıklamayı ve yazmayı test etmek, form gönderimini vi.fn() ile doğrulamak, ve asenkron UI güncellemelerini findBy* ile test etmek -- basit örneklerle.

Orta 11 dk

Son güncelleme: 6 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/user-interaction-testing

---

User Interaction Testing

Component Testing dersinde, bir component'in DOĞRU render olduğunu test etmeyi öğrendik -- ama React uygulamalarının çoğu, kullanıcı bir şeye TIKLAYANA, bir şey YAZANA ya da bir form GÖNDERENE kadar "durağan"dır. Bu ders, kullanıcı ETKİLEŞİMLERİNİ test etmeyi anlatıyor.

user-event ile Gerçekçi Etkileşim Simülasyonu

React Testing Library ile birlikte, kullanıcı etkileşimlerini simüle etmek için `@testing-library/user-event` paketi kullanılır:

```
npm install -D @testing-library/user-event
```

RTL'in kendi `fireEvent` API'si de bir tıklama/yazma tetikleyebilir, ama `fireEvent` tek bir DOM olayını (`click` gibi) doğrudan gönderir. `user-event` ise gerçek bir kullanıcının tıklarken /yazarken tetiklediği ARA adımları da (hover, focus, pointer olayları) simüle eder -- bu yüzden RTL'in resmî dokümantasyonu artık `fireEvent` yerine `user-event`'i ÖNERİYOR.

Tıklamayı Test Etmek

State ve Event'ler kategorisindeki `Counter` component'ini, bu kez gerçek bir tıklama simüle ederek test edelim:

```
import { useState } from "react";
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";
import { describe, it, expect } from "vitest";

function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}

describe("Counter interaction", () => {
  it("increments the count when the button is clicked", async () => {
    // userEvent.setup() creates a "user" object that behaves closer to a
    // real user's click than fireEvent (it also simulates intermediate steps
    // like hover and focus). That's why RTL now RECOMMENDS userEvent.
    const user = userEvent.setup();
    render(<Counter />);

    expect(screen.getByText("Count: 0")).toBeInTheDocument();

    // userEvent's methods are ASYNCHRONOUS -- they must always be awaited.
    await user.click(screen.getByRole("button", { name: /increment/i }));

    expect(screen.getByText("Count: 1")).toBeInTheDocument();
  });
});
```

`userEvent.setup()`, bir "kullanıcı" nesnesi oluşturur. Bu nesnenin metotları (`click`, `type` gibi) HER ZAMAN asenkrondur ve `await` edilmelidir -- unutursak test, tıklama tamamlanmadan bir sonraki satıra geçer ve yanlış (eski) bir DOM durumunu kontrol eder.

Yazmayı Test Etmek

Forms dersindeki controlled input deseni, `user.type` ile test edilir:

```
import { useState } from "react";
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";
import { describe, it, expect } from "vitest";

function NameInput() {
  const [name, setName] = useState("");
  return (
    <div>
      <label htmlFor="name">Name</label>
      <input id="name" value={name} onChange={(e) => setName(e.target.value)} />
      <p>You typed: {name}</p>
    </div>
  );
}

describe("NameInput interaction", () => {
  it("updates the displayed text as the user types", async () => {
    const user = userEvent.setup();
    render(<NameInput />);

    const input = screen.getByLabelText("Name");

    // user.type types the given text CHARACTER BY CHARACTER -- each
    // keystroke triggers the onChange on the controlled component in a way
    // that closely resembles real typing.
    await user.type(input, "Ada");

    expect(screen.getByText("You typed: Ada")).toBeInTheDocument();
    expect(input).toHaveValue("Ada");
  });
});
```

`user.type(input, "Ada")`, verilen metni HARF HARF yazar -- her tuş vuruşu, controlled component'teki `onChange`'i gerçek bir klavyeyle yazmaya çok benzer şekilde tetikler. Testin sonunda hem ekrandaki metni (`getByText`) hem input'un kendi değerini (`toHaveValue`) kontrol ediyoruz.

Form Gönderimini Test Etmek

Bir formu doldurup göndermek, `user.type` ve `user.click`'in birlikte kullanıldığı en yaygın senaryodur:

```
import { useState } from "react";
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";
import { describe, it, expect, vi } from "vitest";

function SignupForm({ onSubmitted }) {
  const [email, setEmail] = useState("");

  function handleSubmit(e) {
    e.preventDefault();
    onSubmitted(email);
  }

  return (
    <form onSubmit={handleSubmit}>
      <label htmlFor="email">Email</label>
      <input id="email" value={email} onChange={(e) => setEmail(e.target.value)} />
      <button type="submit">Sign Up</button>
    </form>
  );
}

describe("SignupForm submission", () => {
  it("calls onSubmitted with the typed email when the form is submitted", async () => {
    const user = userEvent.setup();

    // vi.fn() is a FAKE function that stands in for a real prop -- we can
    // later query what arguments it was called with and how many times.
    const handleSubmitted = vi.fn();
    render(<SignupForm onSubmitted={handleSubmitted} />);

    await user.type(screen.getByLabelText("Email"), "ada@example.com");
    await user.click(screen.getByRole("button", { name: /sign up/i }));

    expect(handleSubmitted).toHaveBeenCalledWith("ada@example.com");
    expect(handleSubmitted).toHaveBeenCalledTimes(1);
  });
});
```

`vi.fn()`, gerçek bir prop yerine geçen SAHTE bir fonksiyon oluşturur -- component'in dışına hiçbir gerçek istek gitmeden, bu fonksiyonun hangi ARGÜMANLARLA, KAÇ KEZ çağrıldığını doğrulayabiliriz. `toHaveBeenCalledWith(...)` ve `toHaveBeenCalledTimes(...)`, bu sahte fonksiyonlara özel matcher'lardır.

Asenkron UI Güncellemelerini Test Etmek

Hooks dersindeki `useEffect` deseniyle, bir component zaman içinde KENDİLİĞİNDEN güncellenebilir (bir fetch isteğinin tamamlanması gibi). Bu tür güncellemeleri test etmek için `findBy*` sorguları kullanılır:

```
import { useEffect, useState } from "react";
import { render, screen } from "@testing-library/react";
import { describe, it, expect } from "vitest";

// The useEffect pattern from the Hooks lesson -- the component updates its
// own state a while after mounting (in a real app this would be a fetch
// request completing; here we use setTimeout to keep it simple).
function DelayedGreeting() {
  const [ready, setReady] = useState(false);

  useEffect(() => {
    const timer = setTimeout(() => setReady(true), 50);
    return () => clearTimeout(timer);
  }, []);

  if (!ready) return <p>Loading...</p>;
  return <p>Welcome!</p>;
}

describe("DelayedGreeting async update", () => {
  it("shows loading first, then the greeting once ready", async () => {
    render(<DelayedGreeting />);

    // On the first render, "Loading..." is still shown.
    expect(screen.getByText("Loading...")).toBeInTheDocument();

    // findByText is the ASYNC version of getByText: if the element isn't
    // there right away, it doesn't throw immediately -- it keeps retrying
    // for a given period (1000ms by default). This is the right way to test
    // anything in the DOM that changes over time (fetch, timers, after an
    // animation) -- waitFor can be used for the same purpose.
    const greeting = await screen.findByText("Welcome!");
    expect(greeting).toBeInTheDocument();
  });
});
```

`getByText` (ve `queryByText`), DOM'u SADECE O AN kontrol eder -- eleman henüz yoksa test başarısız olur. `findByText` ise ASENKRON'dur: eleman hemen yoksa hata fırlatmaz, belirli bir süre boyunca (varsayılan 1000ms) tekrar tekrar dener ve eleman görününce test'e devam eder. DOM'u zamanla değişen (fetch, timer, animasyon sonrası) her şeyi test etmenin doğru yolu budur; aynı amaç için `waitFor(...)` de kullanılabilir.

Özet ve Terimler Sözlüğü

`@testing-library/user-event`, `fireEvent`'ten daha GERÇEKÇİ bir etkileşim simülasyonu sunar; `userEvent.setup()`'tan dönen nesnenin `click` / `type` gibi metotları her zaman `await` edilmelidir. `vi.fn()` ile oluşturulan sahte fonksiyonlar, bir callback prop'un doğru argümanlarla çağrıldığını doğrulamak için kullanılır. Zamanla değişen (asenkron) DOM güncellemeleri, `getByText` yerine `findByText` / `waitFor` ile test edilir.

Terimler Sözlüğü

user-event — Kullanıcı etkileşimlerini (tıklama, yazma), gerçek tarayıcı davranışına yakın şekilde simüle eden kütüphane.

Mock Fonksiyon — `vi.fn()` ile oluşturulan, gerçek bir fonksiyonun yerine geçip çağrılma bilgisini (argümanlar, sayı) kaydeden sahte fonksiyon.

Asenkron Sorgu — `findBy*` gibi, elemanın DOM'da görünmesini bir süre BEKLEYEN sorgu türü.

Pratik Proje

Bu kategoride (Component Testing, User Interaction Testing) öğrendiğimiz kavramları bir arada kullanan, gerçek ve çalıştırılabilir bir örnek proje var: **Testing Demo** -- önceki kategorilerden tanıdık, arama yapılabilen bir kurs listesi ve bir kayıt formu, ama bu kez odak uygulamanın kendisinden çok onu doğrulayan gerçek Vitest testlerinde.

Proje; `SearchBar`, `CourseList` ve `EnrollForm` component'lerinin her birini kendi `.test.jsx` dosyasında (`getByLabelText` + `userEvent.type` , `getByText` / `queryByText` ile filtreleme, `vi.fn()` + `findByText` ile form gönderimi) AYRI AYRI test etmeyi, ve bunların hepsini App içinde lifting state up ile bağlayan bütünü tek bir integration testiyle (`App.test.jsx`) doğrulamayı gösteriyor. Bilgisayarına indirip çalıştırabilir, testleri satır satır inceleyebilirsin:

```
git clone https://github.com/cdurgun/react-course-projects.git
cd react-course-projects
npm install
cd projects/testing
npm test
```

`react-course-projects` deposu **npm workspaces** kullanır -- `npm install` yalnızca bir kez, depo kökünde çalıştırılır ve tüm proje klasörleri ortak bağımlılıkları paylaşır (her klasörde ayrı bir `node_modules` oluşmaz). Kök dizinde `npm install`'ı daha önce çalıştırdıysan, doğrudan `cd react-course-projects/projects/testing` yapıp `npm test` demen yeterli.