Testowanie komponentów React (React Testing Library)

Testowanie komponentów w Vitest przebiega podobnie jak w przypadku Jesta – korzystamy z biblioteki **React Testing Library** do renderowania komponentów w środowisku testowym (JS DOM) i do interakcji jak użytkownik. W testach komponentów najpierw renderujemy komponent za pomocą funkcji `render` z `@testing-library/react`, następnie wyszukujemy elementy (np. poprzez `screen.getByText` lub inne query), a na koniec wykonujemy asercje za pomocą `expect`. Jeśli komponent reaguje na zdarzenia (kliknięcia, wpisywanie tekstu itp.), możemy użyć `userEvent` z `@testing-library/user-event` do symulacji interakcji użytkownika. Poniżej przykład prostego testu komponentu Button, który przyjmuje tekst etykiety oraz funkcję obsługującą kliknięcie:

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

describe('Button component', () => {
  it('renderuje poprawny tekst', () => {
    render(<Button label="Kliknij" onClick={() => {}} />);
    // Sprawdzenie, czy przycisk z podanym tekstem jest w dokumencie
    expect(screen.getByText('Kliknij')).toBeInTheDocument();
  });

  it('wywołuje akcję onClick po kliknięciu', async () => {
    const handleClick = vi.fn();
    render(<Button label="Kliknij" onClick={handleClick} />);
    await userEvent.click(screen.getByText('Kliknij'));
    // Oczekujemy, że funkcja obsługująca kliknięcie została wywołana raz
    expect(handleClick).toHaveBeenCalledTimes(1);
  });
});
```

W powyższym teście użyto kilku ważnych elementów: funkcja `render` renderuje komponent w środowisku testowym (tu wirtualny DOM przeglądarki dzięki jsdom), obiekt `screen` umożliwia wyszukiwanie elementów w wyrenderowanym DOM, `userEvent.click` symuluje kliknięcie w przycisk, zaś `vi.fn()` tworzy **mock** funkcję (spy), pozwalającą śledzić wywołania funkcji przekazanej do komponentu. Asercje wykorzystują wbudowane matchery z biblioteki `@testing-library/jest-dom` takie jak `toBeInTheDocument()` do sprawdzenia, czy element istnieje w DOM. Dzięki globalnej konfiguracji Vitest (o której niżej) te dodatkowe matchery są dostępne w `expect`.

Dobra praktyka: testując komponenty, staraj się sprawdzać ich zachowanie z perspektywy użytkownika (treść wyświetlana w UI, reakcje na zdarzenia) zamiast wewnętrznych stanów. React Testing Library ułatwia takie podejście, np. wyszukując elementy po tekście, rolach czy test-id, co czyni testy bardziej odpornymi na zmiany implementacyjne.

Testowanie hooków (własnych i wbudowanych)

Hooki niestandardowe (custom hooks): W przypadku własnych hooków Reacta, do ich testowania przydaje się specjalna funkcja `renderHook` (dostępna w paczce `@testing-library/react-hooks`). Pozwala ona wywołać hook w kontekście testowym i uzyskać obiekt wyniku z polem `result.current`, reprezentującym zwracane wartości hooka. Dzięki temu można pisać testy podobnie jak dla komponentów – wywołujemy hook za pomocą `renderHook`, a następnie asercjami sprawdzamy wartości zwrócone. Jeśli hook korzysta ze state lub efektów, należy pamiętać o korzystaniu z funkcji `act()` otaczającej operacje, które zmieniają stan (np. wywołania funkcji aktualizujących state), aby React prawidłowo przetworzył zmiany zanim wykonamy asercje. Przykład:

```
import { renderHook, act } from '@testing-library/react-hooks';
import { useCounter } from './useCounter';  // Przykładowy hook zliczający

describe('useCounter', () => {
  it('inicjalizuje licznik z podaną wartością początkową', () => {
    const { result } = renderHook(() => useCounter(5));
    expect(result.current.count).toBe(5);
  });

  it('inkrementuje licznik po wywołaniu funkcji increment', () => {
    const { result } = renderHook(() => useCounter());
    act(() => {
      result.current.increment();
    });
    expect(result.current.count).toBe(1);
  });
});
```

W powyższym teście hook `useCounter` zwraca obiekt z bieżącą wartością licznika (`count`) oraz funkcjami `increment` i `decrement`. `renderHook` uruchamia hook i pozwala odczytać jego wartość zwracaną przez `result.current`. Wywołanie `act()` jest potrzebne wokół `result.current.increment()`, aby aktualizacja stanu została prawidłowo przetworzona zanim sprawdzimy asercję. **Uwaga:** nie należy wyciągać (destrukturyzować) wartości z `result.current` poza blokiem `act`, ponieważ stracimy aktualną referencję – zawsze odwołuj się do `result.current` bezpośrednio podczas sprawdzania efektów wywołania hooka.

Hooki wbudowane: Nie testujemy bezpośrednio wbudowanych hooków typu `useState` czy `useEffect` – zamiast tego testujemy logikę, którą z ich pomocą zaimplementowaliśmy. Innymi słowy, jeśli komponent lub nasz custom hook używa np. `useEffect` do pobrania danych, to testujemy czy *efekt* tego pobierania jest zgodny z oczekiwaniami (np. czy po pewnym czasie stan jest zaktualizowany).

Hooki asynchroniczne: Gdy hook wykonuje operacje asynchroniczne (np. zapytanie HTTP w `useEffect`), możemy w teście zastosować podejście z *mockowaniem* wywołań sieci (patrz sekcja Mockowanie) oraz oczekiwaniem na wynik za pomocą helperów z Testing Library. Przykładowo, możemy użyć `waitFor` aby poczekać na zmianę stanu hooka na oczekiwany. W poniższym fragmencie testu hooka `useProducts` (który pobiera listę produktów) najpierw sprawdzamy, że początkowo `isLoading` jest `true`, a następnie **czekamy** na pojawienie się danych w stanie `products`:

```
const { result } = renderHook(() => useProducts());
expect(result.current.isLoading).toBe(true);
// Oczekiwanie na zmianę stanu wewnątrz hooka (np. po zakończeniu fetch)
await waitFor(() => expect(result.current.products.length).toEqual(1));
expect(result.current.isLoading).toBe(false);
```

Funkcja `waitFor` będzie próbowała wykonywać przekazaną asercję aż do skutku (albo upływu domyślnego timera), dzięki czemu możemy poczekać na zakończenie operacji asynchronicznej wewnątrz hooka. Pamiętaj przy tym, by wcześniej zamockować wywołanie sieci (np. `fetch`), aby test nie wykonywał prawdziwego zapytania.

Testowanie funkcji pomocniczych (utilities)

Testowanie czystych funkcji pomocniczych (tzw. utils) przebiega najprościej – piszemy klasyczne testy jednostkowe, wywołując funkcję z różnymi parametrami i sprawdzając za pomocą `expect`, czy zwraca oczekiwany wynik. Takie testy nie wymagają żadnej specjalnej konfiguracji ani biblioteki do renderowania. Można je umieszczać w plikach `.test.ts` / `.test.tsx` podobnie jak testy komponentów. Przykład prostej funkcji i testu:

```
// utils/math.ts
export function sum(a: number, b: number): number {
  return a + b;
}

// utils/math.test.ts
import { sum } from './math';

it('sum prawidłowo dodaje dwie liczby', () => {
  expect(sum(2, 3)).toBe(5);
});
```

W powyższym przykładzie test sprawdza elementarną funkcjonalność funkcji sum. W praktyce funkcje pomocnicze mogą być bardziej złożone – np. formatowanie tekstu, obliczenia na obiektach, itp. – ale schemat testowania pozostaje taki sam. Vitest obsługuje TypeScript, więc nasze funkcje i testy mogą być pisane w TS; ewentualne błędy typów również będą wyłapywane podczas kompilacji testów.

Testowanie Context API

React Context API służy do przekazywania danych przez drzewo komponentów bez konieczności używania props. Testowanie kodu wykorzystującego context polega głównie na upewnieniu się, że konsument contextu dostaje właściwe wartości w różnych sytuacjach. Możemy podejść do tego na dwa sposoby:

1. Rzeczywisty Context w teście (Provider): Jeśli komponent korzysta z contextu, najprostszym podejściem jest owrapowanie go odpowiednim Providerem podczas renderowania w teście, aby dostarczyć oczekiwane wartości. Na przykład, załóżmy że mamy Context `MyContext` z domyślną wartością `'domyślna'` oraz komponent `ConsumerComponent`, który wyświetla aktualną wartość contextu. Możemy napisać test sprawdzający domyślną wartość (bez providera) oraz wartość dostarczoną poprzez providera:

```
// Definicja contextu i komponentu dla przykładu:
const MyContext = React.createContext('domyślna');
const ConsumerComponent = () => {
  const value = React.useContext(MyContext);
  return <span>Dane: {value}</span>;
};

// Test contextu:
it('używa wartości domyślnej bez providera', () => {
  render(<ConsumerComponent />);  // brak Providera
  expect(screen.getByText('Dane: domyślna')).toBeInTheDocument();
});

it('używa wartości z providera jeśli dostarczony', () => {
  render(
    <MyContext.Provider value="testowa">
      <ConsumerComponent />
    </MyContext.Provider>
  );
  expect(screen.getByText('Dane: testowa')).toBeInTheDocument();
});
```

W pierwszym teście `ConsumerComponent` korzysta z domyślnej wartości contextu, więc oczekujemy tekstu "Dane: domyślna". W drugim teście opakowujemy komponent w `MyContext.Provider`, przekazując wartość `"testowa"`, co powinno skutkować wyświetleniem "Dane: testowa". Ten sposób pozwala przetestować zarówno zachowanie z wartościami domyślnymi, jak i z konkretnymi scenariuszami dostarczanymi przez providera.

2. Własna funkcja renderująca z providerem: Przy częstym testowaniu komponentów korzystających z tego samego Context API można napisać własny wrapper lub utility do renderowania, który automatycznie dostarczy kontekst. Testing Library sugeruje np. zdefiniowanie własnej funkcji `render` wykorzystującej kontekst. Przykład (bazując na dokumentacji Testing Library):

```
// customRender.js (przykładowa implementacja)
import { render } from '@testing-library/react';
import { MyContext } from '../src/MyContext';  // Twój context

const customRender = (ui, { providerProps, ...renderOptions }) => {
  return render(
    <MyContext.Provider {...providerProps}>{ui}</MyContext.Provider>,
```

```
    renderOptions
  );
};

export * from '@testing-library/react';
export { customRender as render };
```

Dzięki temu w testach możemy używać `customRender` zamiast zwykłego `render`, co uprości dostarczanie różnych wartości contextu. Na przykład:

```
customRender(<ConsumerComponent />, { providerProps: { value: 'ABC' } });
```

zapewni, że `ConsumerComponent` otrzyma wartość `'ABC'` z kontekstu. Taka abstrakcja bywa przydatna, ale nie jest konieczna – często wystarczy jawnie użyć providera w samym teście, jak pokazano wcześniej.

Testowanie logiki providera: Jeśli masz komponent Providera z własną logiką (np. modyfikującą przekazywaną wartość contextu), możesz przetestować go podobnie – renderując fragment drzewa z tym Providerem i np. sztucznym konsumentem sprawdzającym dostarczone wartości. W razie potrzeby, możesz też mockować kontekst (np. poprzez zamianę całego modułu z kontekstem na fake, choć to rzadko bywa konieczne). Zwykle jednak bardziej sensowne jest przetestowanie rzeczywistej interakcji komponentów z contextem niż mockowanie go.

Mockowanie (fetch/axios, modułów, hooków, contextu, React Router)

W trakcie testowania często zachodzi potrzeba odizolowania testowanego kodu od zewnętrznych zależności czy efektów ubocznych. **Mockowanie** polega na podmienieniu prawdziwej implementacji funkcji, modułu czy obiektu na kontrolowaną przez nas "atrapę" zwracającą z góry ustalone wartości. W Vitest API do mockowania wzorowane jest na Jest, więc używamy m.in. `vi.fn()`, `vi.spyOn()` oraz `vi.mock()`. Poniżej omówione są typowe scenariusze mockowania w aplikacji React.

Mockowanie wywołań sieciowych (fetch/axios)

Global `fetch`: Jeśli komponent lub hook wykonuje zapytanie przez `fetch`, w środowisku Node/jsdom domyślnie może nie być zaimplementowanej funkcji `fetch`. Najprostszym podejściem jest zdefiniowanie własnej implementacji na potrzeby testu. Możemy to zrobić za pomocą `global.fetch = vi.fn()`. Przykład zastąpienia `fetch` zwracającego z góry ustaloną odpowiedź:

```
const dummyResponse = {
  ok: true,
  status: 200,
  json: async () => ({ message: 'ok' })
};
global.fetch = vi.fn().mockResolvedValue(dummyResponse);

// ... wykonaj kod, który wywołuje fetch(...)
const res = await fetch('/api/data');
const data = await res.json();
expect(data).toEqual({ message: 'ok' });
```

W powyższym kodzie `global.fetch` został zamockowany tak, by zawsze zwracał obiekt `Response` z metodą `json()` zwracającą konkretny obiekt (tu `{ message: 'ok' }`). Gdy testowany kod wywoła fetch('/api/data'), otrzyma naszą atrapę odpowiedzi, a my możemy sprawdzić, że została prawidłowo przetworzona:contentReference[oaicite:11]{index=11}:contentReference[oaicite:12]{index=12}. Alternatywnie Vitest oferuje wygodę w postaci vi.stubGlobal('fetch', fakeFetch)`, co działa podobnie do powyższej techniki.

Biblioteka axios: Do mockowania `axios` – popularnej biblioteki do zapytań HTTP – również możemy użyć `vi.mock()`. Vitest pozwala automatycznie zamockować cały moduł. Najpierw warto utworzyć plik z mockiem (np. `__mocks__/axios.ts`), w którym zdefiniujemy imitację modułu axios. Przykładowo:

```
// __mocks__/axios.ts
import { vi } from 'vitest';
export default {
  get: vi.fn(() => Promise.resolve({ data: { /* ... */ } })),
  // możemy zamockować potrzebne metody (post, put, etc.) podobnie
};
```

Następnie w teście wywołujemy `vi.mock('axios')`, aby Vitest podmienił importy axiosa na naszą imitację. W samym teście możemy dodatkowo kontrolować zachowanie, np.:

```
import axios from 'axios';
vi.mock('axios');

test('powinien pobrać dane z API', async () => {
  axios.get.mockResolvedValue({ data: { id: 123, name: 'Test' } });
  const result = await fetchData(); // funkcja która wewnętrznie używa
axios.get
  expect(result.name).toBe('Test');
  expect(axios.get).toHaveBeenCalledWith('/endpoint');
});
```

W powyższym przykładzie zakładamy, że nasza funkcja `fetchData` wewnątrz wywołuje `axios.get('/endpoint')`. Dzięki mockowi nie wykonuje się prawdziwe zapytanie – zamiast tego `axios.get` zwraca obiekt `{ data: {...} }` zgodnie z definicją `mockResolvedValue`. Możemy też sprawdzić, że metoda została wywołana z oczekiwanym URL. Po takim teście dobrze jest przywrócić oryginalną implementację (np. poprzez `vi.restoreAllMocks()` w `afterEach`).

Mockowanie modułów i funkcji

vi.fn(): Podstawowym sposobem tworzenia mocków jest `vi.fn()`. Tworzy on funkcję, którą możemy przekazać zamiast prawdziwej – jak w poprzednich przykładach (np. `vi.fn()` użyty do `handleClick`). Funkcje utworzone przez `vi.fn` mogą śledzić liczbę wywołań, argumenty oraz pozwalają ustalać wartości zwracane (np. `vi.fn(() => 42)` zwróci 42). Możemy też użyć metody `mockResolvedValue` / `mockReturnValue` aby określić co mock ma zwracać (Promise lub wartość synchroniczna).

vi.spyOn(): Jeśli chcemy **podglądać** istniejącą metodę/ funkcję, nie całkowicie ją zastępując, używamy `vi.spyOn(obiekt, 'metoda')`. Spy pozwala sprawdzić, czy dana metoda została wywołana i ewentualnie z jakimi argumentami. Można również za pomocą spya zmienić implementację na czas testu (metodą `mockImplementation` lub prostszymi `mockReturnValue`, `mockResolvedValue` itp.). Np.:

```
const consoleSpy = vi.spyOn(console, 'error');
// ... wykonać kod który powinien wywołać console.error(...)
expect(consoleSpy).toHaveBeenCalledOnce();
// ewentualnie:
consoleSpy.mockImplementation(() => {})  // tymczasowo wyłącza logowanie
błędów
```

Po teście dobrze jest posprzątać po spy/mocks. Vitest udostępnia `vi.restoreAllMocks()`, które można wywołać w `afterEach()`, aby automatycznie przywrócić wszystkie zamockowane funkcje do ich oryginalnych implementacji po każdym teście. Dzięki temu unikniemy niepożądanych interakcji między testami.

vi.mock(): Pozwala zamockować cały moduł (pakiet lub własny plik). Użycie jest proste – na górze testu (lub nawet poza testem) wywołujemy `vi.mock('nazwa-modułu', () => ({ /* fake implementacja */ }))`. Vitest podobnie jak Jest może też korzystać z plików **mocks** (jak pokazano dla axiosa). Mockowanie modułów jest przydatne, gdy testowany kod korzysta z zewnętrznych modułów, których działania nie chcemy wywoływać (np. wywołań do bazy, logowania, zewnętrznych SDK itd.). **Uwaga:** Vitest (tak jak Jest) domyślnie hoistuje (`vi.mock`) na górę, więc takie wywołanie jest wykonywane przed załadowaniem modułów w teście. Pamiętaj, by importy i `vi.mock` były w odpowiedniej kolejności zgodnie z dokumentacją Vitest.

Mockowanie hooków i contextu

Własne hooki często stanowią warstwę pośrednią, np. używają context API lub wywołań sieci. Jeśli chcemy przetestować komponent korzystający z custom hooka *nie wchodząc* w jego wewnętrzną logikę, możemy zamockować sam hook. Skoro hook jest zwykłą funkcją eksportowaną z modułu, da się go zamockować podobnie jak każdą inną funkcję. Można użyć `vi.mock()` na cały moduł z hookiem lub skorzystać z `vi.spyOn`. Ten drugi sposób bywa wygodny, bo pozwala w teście ustawić różne zwroty hooka w różnych przypadkach. Przykład: załóżmy, że mamy hook `useAuth` zwracający obiekt `{ user, login }`. Możemy zamockować ten hook w teście komponentu, który z niego korzysta:

```
// Plik testu komponentu, który używa useAuth
import * as authHooks from '../hooks/useAuth';
```

```
describe('Komponent wymagający autoryzacji', () => {
  beforeEach(() => {
    // Spy na hook useAuth – zastąpienie jego implementacji na potrzeby
testu
    vi.spyOn(authHooks, 'useAuth').mockReturnValue({
      user: { name: 'Tester' },
      login: vi.fn()
    });
  });

  it('pokazuje nazwę zalogowanego użytkownika', () => {
    render(<MyProtectedComponent />);
    expect(screen.getByText('Witaj, Tester')).toBeInTheDocument();
  });
});
```

W powyższym kodzie importujemy *wszystkie* eksporty z modułu `useAuth` jako `authHooks`, następnie za pomocą `vi.spyOn` podmieniamy implementację `authHooks.useAuth`, by zwracała kontrolowane przez nas dane. W efekcie, gdy testowany komponent wywoła wewnętrznie `useAuth()`, otrzyma obiekt z użytkownikiem o nazwie "Tester". Dzięki temu możemy łatwo testować zachowanie komponentu dla różnych scenariuszy (użytkownik zalogowany/niezalogowany itp.), nie zależąc od prawdziwej logiki hooka. Po teście spy warto przywrócić (`authHooks.useAuth.mockRestore()` lub ogólnie `vi.restoreAllMocks()` w `afterEach`).

Podobną technikę można zastosować dla hooków korzystających z contextu – np. jeśli mamy custom hook `useFeatureFlag()` czerpiący dane z kontekstu, możemy go zamockować by zwracał wybrany stan flagi, zamiast konfigurować cały kontekst. **Alternatywą** jest oczywiście dostarczenie prawdziwego kontekstu (jak opisano w sekcji testowania Context API) – wybór podejścia zależy od tego, czy chcemy testować integrację z Contextem, czy tylko logikę wyższej warstwy.

Jeśli chodzi o **sam context** – w rzadkich przypadkach można pokusić się o zamockowanie np. React.useContext, ale zazwyczaj nie jest to potrzebne ani zalecane. Lepiej zamockować ewentualny *custom hook* opakowujący useContext (jak pokazano wyżej), albo po prostu użyć dostosowanego providera.

Mockowanie React Router

Aplikacje React często korzystają z `react-router-dom` do nawigacji. Komponenty używające Routera (np. linki, przyciski nawigujące, komponenty wyświetlające różne widoki zależnie od routingu) wymagają dodatkowego przygotowania w testach. Mamy dwa podejścia: **integracyjne** z rzeczywistym routerem w pamięci, lub czysto **modułowe** z mockowaniem.

Podejście integracyjne (MemoryRouter): Dla komponentów korzystających z Routera najprostszym sposobem jest owinąć je w testach w `<MemoryRouter>` dostarczany przez React Router. MemoryRouter to lekki router działający tylko w pamięci, idealny do testów. Pozwala on także ustawić początkową ścieżkę, aby przetestować np. co się stanie dla danego URL. Przykład użycia:

```
import { MemoryRouter } from 'react-router-dom';

test('nawigacja do strony About zmienia widok', async () => {
  render(<App />, { wrapper: MemoryRouter });  // render całej aplikacji z
routerem
  expect(screen.getByText(/you are home/i)).toBeInTheDocument();
  await userEvent.click(screen.getByText(/about/i));
  expect(screen.getByText(/you are on the about
page/i)).toBeInTheDocument();
});
```

W powyższym teście (zaczerpniętym z dokumentacji RTL) renderujemy cały komponent `<App>` wewnątrz routera, dzięki czemu linki i nawigacja działają. Najpierw sprawdzamy, że domyślnie wczytuje się strona główna, następnie symulujemy kliknięcie w link "About" i oczekujemy pojawienia się tekstu ze strony About. MemoryRouter domyślnie startuje na ścieżce `'/'`; można to zmienić poprzez parametr `initialEntries`, np.:

```
render(
  <MemoryRouter initialEntries={['/some/bad/route']}>
    <App />
  </MemoryRouter>
);
expect(screen.getByText(/No match/i)).toBeInTheDocument();  // 404 case
```

Takie podejście pozwala testować pełną integrację z routingiem, łącznie z tym jak komponenty Routes renderują odpowiednie komponenty dla ścieżek.

Podejście modułowe (mockowanie routera): Czasem chcemy przetestować komponent korzystający z np. `useNavigate` albo `useParams`, ale bez uruchamiania całego routera. Można wtedy zamockować te funkcje z `react-router-dom`. Vitest pozwala na częściowe mockowanie modułów – czyli możemy zachować resztę funkcjonalności, a nadpisać tylko wybrane eksporty. Przykładowo, aby zamockować `useNavigate` (tak by np. sprawdzić czy został wywołany z oczekiwanym argumentem), możemy zrobić:

```
const fakeNavigate = vi.fn();
vi.mock('react-router-dom', async () => {
  const actual = await vi.importActual<typeof import('react-router-
dom')>('react-router-dom');
  return {
    ...actual,
```

```
    useNavigate: () => fakeNavigate
  };
});
```

Powyższy kod wykorzystuje `vi.importActual` do zaimportowania prawdziwego modułu routera, następnie zwraca wszystkie jego rzeczywiste eksporty, podmieniając tylko `useNavigate` na naszą własną funkcję mockującą. Dzięki temu w testach każdy wywołanie `useNavigate()` zwróci nasz `fakeNavigate` (który jest `vi.fn()`), a więc możemy np. sprawdzić `expect(fakeNavigate).toHaveBeenCalledWith('/expected-path')` po wykonaniu akcji. Podobnie można by zamockować np. `useParams` czy inne hooki z Routera, ale należy robić to ostrożnie – łatwo w ten sposób "oszukać" logikę komponentu. W większości wypadków użycie MemoryRouter z odpowiednim kontekstem jest bardziej odpowiednią metodą testowania (bo testujemy realne zachowanie zamiast wewnętrznych wywołań). Niemniej, technika partial mock bywa przydatna do testowania np. czy poprawnie wywołujemy `navigate()` bez renderowania całego routera.

Na koniec, pamiętajmy aby po testach z mockowanym routerem przywrócić oryginalny stan (np. `vi.resetModules()` lub odświeżyć importy, jeśli kolejne testy mają używać prawdziwego routera).

Konfiguracja środowiska testowego (Vitest, setup, TypeScript)

Aby możliwe było wygodne testowanie Reacta, musimy odpowiednio skonfigurować Vitest i otoczenie. Załóżmy, że aplikacja została stworzona przy pomocy Vite (szablon React + TS). Kroki konfiguracji będą następujące:

Instalacja zależności:
Zacznij od dodania Vitest oraz biblioteki Testing Library:

```
npm install --save-dev vitest @testing-library/react @testing-library/user-event @testing-library/jest-dom
```

Pakiet `vitest` to framework testów, `@testing-library/react` to narzędzia do testowania komponentów, `@testing-library/user-event` ułatwia symulację działań użytkownika, zaś `@testing-library/jest-dom` dostarcza dodatkowe **matchery** (typu `.toBeInTheDocument()`) dla asercji na elementach DOM.

Plik konfiguracyjny Vitest:
W projekcie utwórz plik `vitest.config.ts` (lub użyj istniejącego `vite.config.ts` eksportując konfigurację testów). Trzeba poinformować Vitest, że testujemy kod przeglądarkowy (DOM) oraz włączyć pewne ułatwienia. Minimalna konfiguracja może wyglądać tak:

```
// vitest.config.ts
import { defineConfig } from 'vitest/config';
import react from '@vitejs/plugin-react';

export default defineConfig({
  plugins: [react()],
  test: {
    environment: 'jsdom',
    globals: true,
    setupFiles: './src/setupTests.ts'
  }
});
```

Wyjaśnienie najważniejszych opcji:

- `environment: 'jsdom'` – ustawia środowisko testowe symulujące przeglądarkę (jsdom). Dzięki temu możemy testować interakcje z DOM, a np. `document` czy `window` są dostępne.
- `globals: true` – dzięki temu nie musimy importować w każdym teście funkcji typu `describe`, `it`, `expect` z `vitest`; są one dostępne globalnie (tak jak w Jest).
- `setupFiles` – ścieżka do pliku, który zostanie uruchomiony przed każdym testem, służy do globalnej konfiguracji testów (odpowiednik `setupFilesAfterEnv` z Jesta).

Setup tests (setupTests.ts):
Stwórz plik `src/setupTests.ts` i zaimportuj w nim rozszerzenia matchers oraz ewentualnie dokonaj innych globalnych konfiguracji. Typowy `setupTests.ts` dla Reacta wygląda następująco:

```
// src/setupTests.ts
import '@testing-library/jest-dom';
import { expect } from 'vitest';
import matchers from '@testing-library/jest-dom/matchers';

// Rozszerzenie expect o dodatkowe matchery (toBeInTheDocument, itp.)
expect.extend(matchers);
```

Pierwszy import dodaje do środowiska testowego predefiniowane asercje z Jest-DOM (np. `toBeInTheDocument`). Dodatkowo, w powyższym przykładzie (alternatywnym) użyto `expect.extend` by jawnie zarejestrować te matchery – w zależności od wersji biblioteki i konfiguracji, samo zaimportowanie `jest-dom` może już rozszerzyć `expect` globalnie. Warto upewnić się, że konfiguracja działa – np. uruchamiając jeden test sprawdzający `expect(element).toBeInTheDocument()`.

Konfiguracja TypeScript (tsconfig):
Vitest dostarcza własne typy (dla globalnych zmiennych testowych), a także Testing Library ma typy dla matcherów. Aby TypeScript nie zgłaszał błędów, należy włączyć te definicje. W pliku `tsconfig.json` (lub osobnym `tsconfig.testing.json` jeżeli taki podział preferujesz) dodaj w sekcji `compilerOptions` pole `"types"` zawierające przynajmniej `"vitest/globals"` oraz `"@testing-library/jest-dom"`. Przykład:

```
{
  "compilerOptions": {
    // ... inne opcje ...
    "types": ["vitest/globals", "@testing-library/jest-dom"]
  }
}
```

To sprawi, że TypeScript będzie znał takie globalne nazwy jak `describe` czy `expect`, a także rozszerzenia matchera (`toBeInTheDocument` itp.), dzięki czemu autouzupełnianie i kompilacja nie będą zgłaszać błędów. Dodatkowo upewnij się, że w polu `"include"` jest uwzględniony ewentualnie plik setupTests (jeśli leży poza katalogiem źródeł) – choć umieszczenie go w `src` zazwyczaj wystarcza.

Skrypt testowy w package.json:
Dodaj w `package.json` skrót do uruchamiania testów, np.:

```
"scripts": {
  "test": "vitest",
  "test:watch": "vitest --watch"
}
```

Teraz możesz uruchomić wszystkie testy komendą `npm test`. Jeśli wolisz, możesz również użyć opcji `vitest --ui`, aby otworzyć interfejs graficzny Vitest (Vitest UI) w przeglądarce – jest to przydatne do obserwowania na bieżąco wyników testów podczas TDD.

Struktura testów i przykładowe techniki (describe, it/test, expect, beforeEach, afterAll itp.)

Struktura blokowa: Vitest używa tego samego mechanizmu definiowania testów co Jest/Mocha – testy można grupować w bloki `describe`, co ułatwia organizację i czytelność raportów. W każdym bloku `describe` możemy mieć wiele przypadków testowych definiowanych za pomocą `it` lub `test` (działają one identycznie). Przykład struktury:

```
describe('Grupa testów dla funkcji X', () => {
  beforeEach(() => {
    // Kod ten wykona się przed KAŻDYM testem z tej grupy
  });

  afterAll(() => {
    // Kod ten wykona się RAZ po wykonaniu WSZYSTKICH testów w tej grupie
  });

  it('powinien zwrócić wynik poprawny dla danych przypadek A', () => {
    // ... kod testu ...
    expect(...).toBe(...);
  });

  test('powinien rzucić wyjątek dla niepoprawnych danych', () => {
    // ... kod innego testu ...
    expect(() => funkcjaX(złeDane)).toThrow();
  });
});
```

Bloki `beforeEach`, `afterEach`, `beforeAll`, `afterAll` służą do wykonywania kodu przygotowawczego lub sprzątającego. `beforeEach` uruchamia się przed każdym testem w danym bloku (lub jego podblokach), co jest idealne do np. resetowania wspólnego stanu, tworzenia świeżych instancji komponentu, czyszczenia bazy danych testowej itp. `afterEach` analogicznie uruchamia się po każdym teście – często używa się go do przywracania zamockowanych funkcji (`vi.restoreAllMocks()`), czyszczenia efektów ubocznych, odmontowania komponentów itp.. Natomiast `beforeAll`/`afterAll` wykonują się raz na początku lub końcu całej grupy testów – można tam np. zainicjować wspólne zasoby (serwer testowy, połączenie z DB) i na końcu je zamknąć. W większości przypadków w testach frontendowych przeglądarki częściej korzysta się z `beforeEach`/`afterEach` dla izolacji testów.

Asercje (`expect`): Do weryfikacji wyników używamy funkcji globalnej `expect(value)`, którą łączymy z matcherami. Kilka popularnych matcherów:

- Liczby, stringi, boolean: `toBe` (porównanie ścisłe), `toEqual` (głębokie porównanie obiektów/tabl.), `toMatch` (dla regex lub string w stringu).
- Obiekty: `toHaveProperty('field', value)`, `toContain` (zawiera element w tablicy).
- Asynchroniczne: `resolves` / `rejects` we współpracy z expect (np. `await expect(Promise.reject('err')).rejects.toThrow('err')`).
- Błędy: `toThrow` (sprawdzenie, czy funkcja rzuca wyjątek).
- Elementy DOM (dzięki jest-dom): `toBeInTheDocument`, `toHaveTextContent`, `toHaveAttribute` itp.

- Wywołania funkcji (spies/mocks): `toHaveBeenCalled()`, `toHaveBeenCalledTimes(n)`, `toHaveBeenCalledWith(arg1, arg2, ...)` do weryfikacji interakcji.

Możliwości jest bardzo dużo – warto zajrzeć do dokumentacji Testing Library i Vitest/Jest po pełną listę matcherów. Warto też wspomnieć o **snapshot testing** – Vitest obsługuje snapshoty podobnie jak Jest (matcher `toMatchSnapshot()`), co bywa użyteczne przy testowaniu renderów komponentów (choć w dobie Testing Library zaleca się raczej sprawdzać konkretne elementy niż całe snapshoty DOM).

oznaczanie testów: Vitest pozwala oznaczać testy jako *tylko* lub *pomijane*. Używamy do tego `test.only`/`it.only` aby uruchomić tylko dany test (przydatne w trakcie pisania, by skupić się na jednym teście) lub `test.skip`/`it.skip` aby pominąć dany przypadek (np. gdy jest niesprawny lub zależny od środowiska). Istnieje też funkcja `test.todo` do oznaczenia "do zrobienia". Pamiętaj, żeby nie zostawiać `it.only` w kodzie produkcyjnym testów – zablokuje to wykonywanie reszty testów!

Na koniec, zwróć uwagę że **Vitest jest kompatybilny z API Jesta** – większość powyższych konstrukcji pochodzi wprost z Jesta, więc jeśli znasz Jesta, poczujesz się jak w domu. Dotyczy to również funkcji `vi.fn()`/`vi.spyOn` (odpowiednik `jest.fn()`/`jest.spyOn`) oraz np. timerów (`vi.useFakeTimers()` itp.).

Pokrycie testami (coverage) i integracja z CI/CD

Pokrycie testów (code coverage): Vitest potrafi generować raport pokrycia kodu testami. Wystarczy uruchomić testy z flagą `--coverage`. Domyślnie używany jest mechanizm c8 (oparty o Istanbul), który wygeneruje raport w folderze `coverage`. Raport zawiera m.in. pliki HTML z podświetleniem nieprzetestowanych linii oraz raport w formacie tekstowym i LCOV do dalszej analizy. Przykładowo, uruchomienie:

```
npm test -- --coverage    # jeśli "test" mapuje na "vitest"
```

sprawi, że po zakończeniu testów w konsoli zobaczymy podsumowanie pokrycia (ile procent linii/branchy/funcji zostało wykonanych w trakcie testów), a katalog `coverage/` będzie zawierał szczegółowe wyniki. Można dostosować formaty raportu w konfiguracji (parametr `test.coverage.reporter` w `vitest.config.ts`), np. do wygenerowania tylko `text` i `lcov`. Jeśli używasz serwisów takich jak **Codecov** lub **Coveralls**, będą one mogły wykorzystać plik `lcov.info` z raportu coverage.

Integracja z CI (Continuous Integration): Automatyzacja testów w CI/CD (np. GitHub Actions, GitLab CI, Jenkins) jest kluczowa, by zapewnić, że wszystkie testy przechodzą przed wprowadzeniem zmian na główną gałąź. Szczęśliwie, integracja Vitest nie różni się znacząco od integracji np. Jesta. Wystarczy dodać krok uruchamiający nasze testy. Poniżej uproszczony przykład konfiguracji GitHub Actions (workflow w pliku YAML):

```
name: CI Tests
on: [push, pull_request]

jobs:
  build-and-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: 18
          cache: 'npm'
      - run: npm ci
      - run: npm test -- --coverage
      - name: Upload coverage artifact
        uses: actions/upload-artifact@v3
        if: always()
        with:
          name: coverage-report
          path: coverage
```

Ten przykładowy workflow uruchamia się przy każdym pushu i pull requeście. W kroku **Install dependencies** instalowane są paczki (`npm ci`), następnie w kroku **Run tests** odpalane są testy wraz z generowaniem pokrycia. Krok **Upload coverage artifact** sprawia, że wygenerowany raport coverage zostanie zachowany jako artefakt do wglądu w interfejsie Actions (dzięki użyciu `if: always()` artefakt zapisze się nawet jeśli testy nie przeszły). Można ten raport wykorzystać np. lokalnie pobierając artefakt, bądź skonfigurować dodatkową akcję wysyłającą go do Codecov.

W CI warto również ustawić **wymagane statusy** – np. na GitHub można włączyć *branch protection rules* dla gałęzi głównej, tak by wymagać przejścia workflow testów przed zmerge'owaniem pull requestu. Dzięki temu żaden kod, który psuje testy, nie trafi do main. Dobrze jest też korzystać z macierzy buildów (np. testy na różnych wersjach Node) oraz cache'owania zależności, co przyspieszy przebieg.