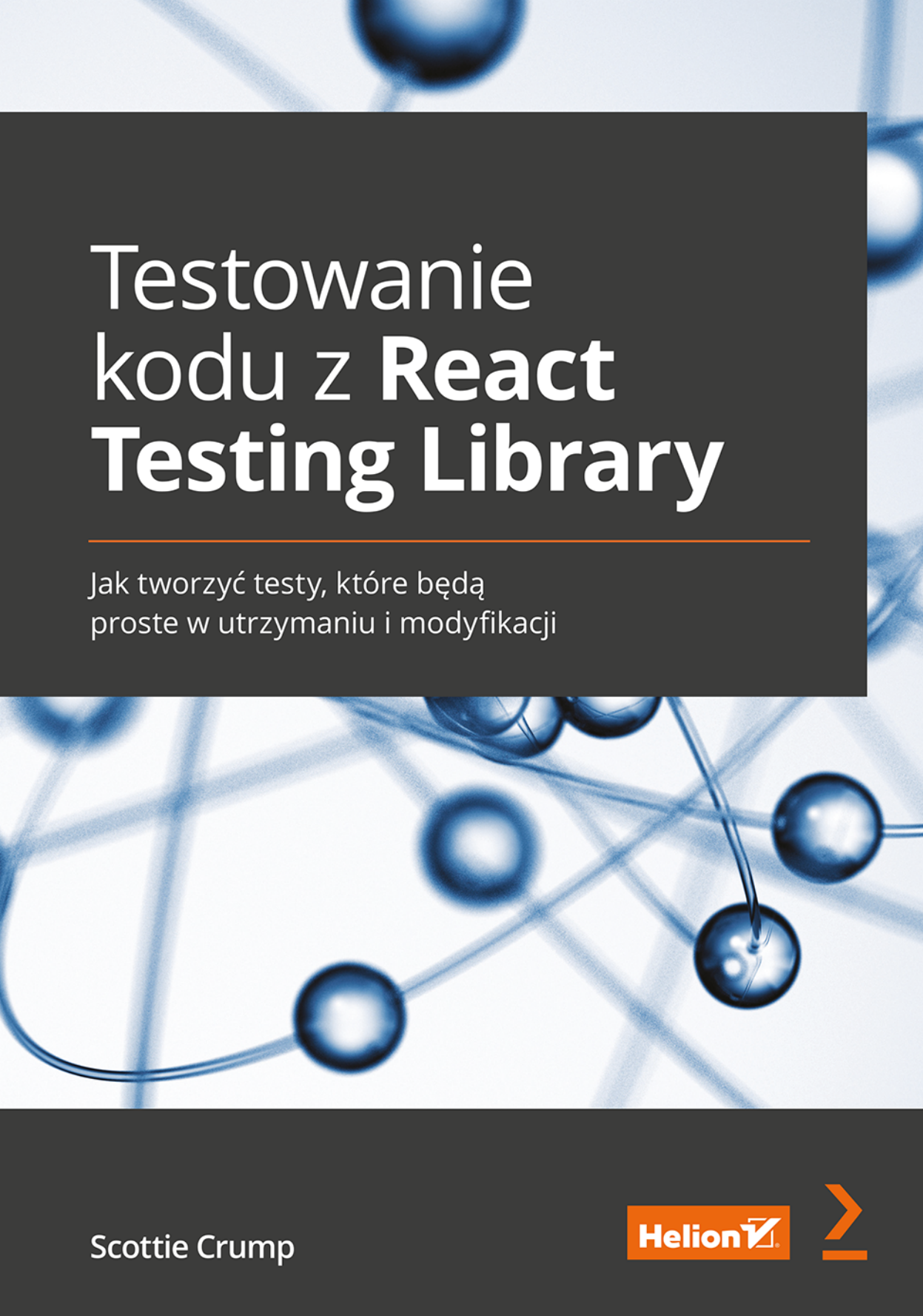


Testowanie kodu z **React Testing Library**

Jak tworzyć testy, które będą proste w utrzymaniu i modyfikacji

Scottie Crump

Helion

Tytuł oryginału: Simplify Testing with React Testing Library: Create maintainable tests using RTL that do not break with changes

Tłumaczenie: Katarzyna Bogusławska

ISBN: 978-83-283-8872-7

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/bibrea
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/bibrea.zip

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

ROZDZIAŁ 3

Testowanie złożonych komponentów przy użyciu React Testing Library

W rozdziale 2,. „Praca z React Testing Library”, nauczyłeś się testować komponenty warstwy prezentacji. Większość funkcjonalności projektuje się jednak z myślą o tym, by umożliwić użytkownikom działania, w konsekwencji których zmieniają się stan i dane wyjściowe. Przetestowanie jak największej liczby scenariuszy działań użytkowników jest kluczowe dla ograniczania ryzyka przed wdrożeniem kodu na środowisku produkcyjnym. Po zapoznaniu się z treścią tego rozdziału będziesz umiał symulować działania użytkownika dzięki modułom `fireEvent` oraz `user-event`. Dowiesz się, jak testować komponenty, które wchodzą w interakcje z interfejsami API usług sieciowych. Wreszcie poznasz technikę test-driven development jako sposób kierowania procesem wytwarzania funkcjonalności.

W tym rozdziale poruszę następujące zagadnienia:

- wykonywanie działań na komponentach przy użyciu modułu `fireEvent`;
- symulowanie zdarzeń obiektowego modelu dokumentu (DOM) dzięki modułowi `user-event`;
- testowanie komponentów, które wchodzą w interakcje z interfejsami API;
- implementowanie podejścia test-driven development z React Testing Library.

Umiejętności, które nabędziesz w tym rozdziale, dadzą Ci solidne zrozumienie testowania rezultatów zachowań użytkowników. Zyskasz także inną perspektywę na budowanie komponentów od początku do końca.

Wymagania techniczne

Aby uruchomić przykłady z tego rozdziału, będziesz musiał zainstalować na swoim komputerze Node.js. We wszystkich przykładach będę posługiwał się narzędziem terminalowym `create-react-app`. Jeśli jeszcze o nim nie słyszałeś, zapoznaj się z nim, zanim zaczniesz pracę w tym rozdziale. Choć nie jest to niezbędne, może się okazać, że łatwiej będzie Ci zrozumieć poruszane w tym rozdziale tematy po przypomnieniu sobie materiałów z dwóch pierwszych rozdziałów książki.

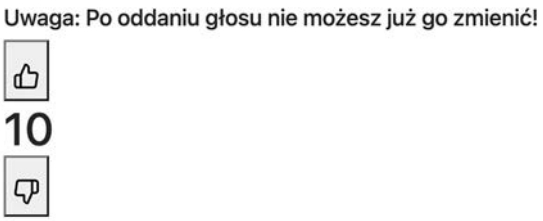
Zawarte tu przykłady możesz znaleźć pod adresem *https://ftp.helion.pl/przyklady/bibrea.zip*.

Testowanie zdarzeń użytkowników

Z tego podrozdziału dowiesz się, jak symulować zdarzenia użytkowników i testować ich wyniki. Aby testować interakcje między komponentami, podobnie jak między użytkownikami a komponentami, potrzebujesz metod symulowania zdarzeń DOM w testach. W drzewie DOM użytkownik może wyzwolić wiele zdarzeń. Przykładowo, użytkownik może wyzwolić zdarzenie przyciśnięcia klawisza przy wpisywaniu tekstu w odpowiednie pole, zdarzenie kliknięcia w interakcji z przyciskiem czy zdarzenie przesunięcia myszy po elemencie podczas przeglądania opcji listy rozwijanej. Biblioteka DOM Testing Library udostępnia dwie inne biblioteki do symulowania akcji użytkownika, tj. `fireEvent` oraz `user-event`. Obie omówię w kolejnych punktach.

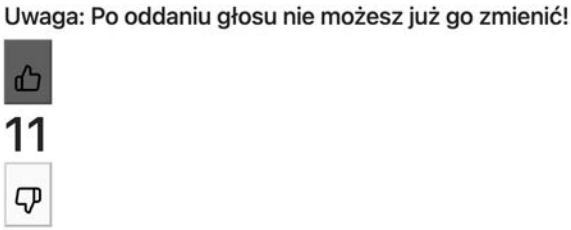
Symulowanie akcji użytkownika za pomocą fireEvent

Możesz skorzystać z modułu `fireEvent` do naśladowania akcji użytkownika na rezultatach działania komponentów w drzewie DOM. Przykładowo, możesz zbudować komponent wielokrotnego użytku `Vote`, który renderuje w drzewie DOM rezultat przedstawiony na rysunku 3.1.



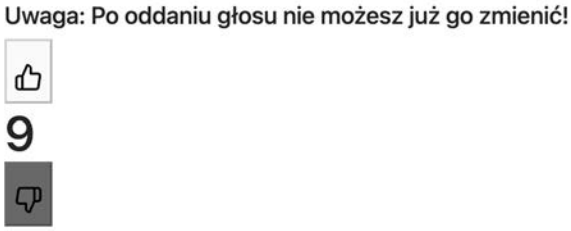
Rysunek 3.1. Komponent Vote

Na rysunku 3.1 liczba 10 przedstawia ranking polubień. Do dyspozycji użytkownika są dwa przyciski, które może klikać, by zagłosować i wpłynąć na liczbę polubień — z kciukiem w górę i kciukiem w dół. Wyświetlana jest także informacja, że użytkownik może zagłosować tylko raz. Po kliknięciu przycisku z kciukiem w górę użytkownik zobaczy widok pokazany na rysunku 3.2.



Rysunek 3.2. Głos „kciuk w górę"

Widać na ilustracji, że liczba polubień wzrosła z 10 do 11. Gdy natomiast użytkownik kliknie przycisk z kciukiem w dół, zobaczy nieco inny widok, przedstawiony na rysunku 3.3.



Rysunek 3.3. Głos „kciuk w dół"

Tym razem widać, że liczba polubień zmalała z 10 do 9. Kliknięcie przycisku to zdarzenie, które można zasymulować za pomocą `fireEvent`. W kodzie implementacji komponentu `Vote` wywoływana jest procedura obsługi zdarzenia z logiką nakazującą aktualizację liczby polubień, jaką użytkownik widzi na ekranie:

```
const handleLikeVote = () => dispatch({ type: 'LIKE' })
  const handleDislikeVote = () => dispatch({ type: 'DISLIKE' })

  return (
    <div className="d-flex d-inline-flex flex-column h1 m-2">
      <h5>Uwaga: Po oddaniu głosu nie możesz już go zmienić!</h5>
      <button
        onClick={handleLikeVote}
        disabled={hasVoted}
        style={clickedLike ? { background: 'green' } : null}
      >
        <img src={thumbsUp} alt="thumbs up" />
      </button>
```

W powyższym kodzie przycisk wiąże się z procedurą obsługi zdarzenia `onClick`. Po przyciśnięciu przycisku procedura wywołuje metodę `handleLikeVote`, która z kolei wywołuje inną metodę — `dispatch` — ta zaś aktualizuje liczbę polubień.

> Zapoznaj się z przykładowym kodem pobranym pod adresem *https://ftp.helion.pl/przyklady/bibrea.zip*, by zobaczyć cały komponent.

Możesz napisać test, który zweryfikuje rezultat głosowania:

```
import { fireEvent, render, screen } from '@testing-library/react'
import Vote from './Vote'
test('zwiększa liczbę polubień o jeden', () => {
  render(<Vote totalGlobalLikes={10} />)
```

W powyższym kodzie importuję `fireEvent`, `render` i `screen` z React Testing Library. Następnie importuję komponent `Vote` poddawany testom. Potem organizuję kod testowy w metodzie `test` i korzystam z metody `render` do wyrenderowania komponentu `Vote` z wartością właściwości `totalGlobalLikes` ustawioną na 10 i przekazaną do komponentu.

Ta właściwość wyraża liczbę polubień, którą początkowo widać na ekranie, gdy komponent zostaje wyrenderowany i oddaje stan całej aplikacji pod względem polubień. W pełnej, gotowej aplikacji wartość `totalGlobalLikes` byłaby przekazywana do komponentu `Vote` przez komponent-rodzica. Wracając do testu, powinienem teraz zająć się interakcją i asercją wyniku w wyrenderowanym komponencie:

```
  expect(screen.getByText(/10/i)).toBeInTheDocument()
  fireEvent.click(screen.getByRole('button', { name: /thumbs up/i }))
  expect(screen.getByText(11).toBeInTheDocument()
  expect(screen.getByRole('button', { name: /thumbs
up/i})).toHaveStyle('background: green')
})
```

W powyższym bloku kodu najpierw dokonuję weryfikacji, czy lokalna dla komponentu `Vote` wersja zmiennej `totalGlobalLikes` znajduje się w dokumencie i ma wartość 10. Następnie korzystam z metody `click` z `fireEvent`, aby kliknąć przycisk z nazwą `thumbs up`. Potem dokonuję asercji na wartości `totalGlobalLikes`, która powinna zostać zaktualizowana w dokumencie i wynosić 11. Wreszcie weryfikuję, czy kolor tła przycisku `thumbs up` zmienił się na zielony.

W wielu wypadkach użycie `fireEvent` jest zupełnie wystarczające. Niestety, moduł ten ma pewne ograniczenia. Przykładowo, gdy użytkownik wykonuje pewną akcję, taką jak wpisywanie tekstu w pole tekstowe, wyzwolonych może zostać wiele zdarzeń, np. `keydown` i `keyup`. Moduł `fireEvent` udostępnia metody wykonania wszystkich tych pojedynczych akcji, ale nie pozwala na obsłużenie ich wszystkich w sekwencji.

Z ograniczeniami modułu fireEvent radzi sobie jednak biblioteka `user-event`, której poświęcony będzie kolejny punkt.

Symulowanie działań użytkownika z user-event

Biblioteka `user-event` to rozbudowana wersja `fireEvent`. W poprzednim punkcie poznałeś metody `fireEvent` służące do symulowania różnych zdarzeń, jakie są wyzwalane, gdy użytkownik np. wpisuje tekst w pole tekstowe. Biblioteka `user-event` oferuje wiele metod takich jak `click` czy `type`, które automatycznie naśladują wszystkie zdarzenia, jakie zachodzą, gdy użytkownik dokonuje akcji w drzewie DOM. Zaletą metod `user-event` jest to, że dostarczają one większą wartość niż metody `fireEvent`.

Narzędzie `create-react-app` zawiera już w sobie instalację `user-event`. W projektach, w których nie posługujesz się `create-react-app`, będziesz musiał skorzystać z poniższego polecenia w celu instalacji:

```
npm install --save-dev @testing-library/user-event
```

Możesz teraz zaktualizować poprzednią wersję testu komponentu `Vote` z wykorzystaniem `user-event`:

```
import { render, screen } from '@testing-library/react'
import user from '@testing-library/user-event'
import Vote from './Vote'

test('zwiększa liczbę polubień o jeden', () => {
  render(<Vote totalGlobalLikes={10} />)

  expect(screen.getByText(/10/i)).toBeInTheDocument()
  user.click(screen.getByRole('button', { name: /thumbs up/i }))
  expect(screen.getByText(/11/i)).toBeInTheDocument()

  expect(screen.getByRole('button', { name: /thumbs up/i })).toHaveStyle(
    'background: green'
  )
})
```

W powyższym kodzie importuję bibliotekę `user-event` pod nazwą `user`. Potem korzystam z metody `click` z `user-event`, by kliknąć przycisk polubienia. Taki test jest bardziej wartościowy, ponieważ w większym stopniu odpowiada działaniom użytkownika w drzewie DOM. Zespół React Testing Library zaleca stosowanie `user-event` zawsze, gdy to możliwe, więc nie będę już korzystał z `fireEvent` w kolejnych przykładach w tej książce.

Gdy omawiałem komponent `Vote` w poprzednim punkcie, zaznaczałem, że użytkownik może oddać głos tylko raz. Można napisać test, który będzie weryfikował to ograniczenie:

```
test('Użytkownik może zagłosować tylko raz', () => {
  render(<Vote totalGlobalLikes={10} />)
  const thumbsUpBtn = screen.getByRole('button', { name: /thumbs up/i })
  const thumbsDownBtn = screen.getByRole('button', { name: /thumbs down/i })

  expect(screen.getByText(/10/i)).toBeInTheDocument()
  user.click(thumbsUpBtn)
  user.click(thumbsUpBtn)
  expect(screen.getByText(/11/i)).toBeInTheDocument()

  user.click(thumbsDownBtn)
  expect(screen.getByText(/11/i)).toBeInTheDocument()
})
```

W powyższym kodzie najpierw uzyskuję dostęp do przycisków `thumbs up` i `thumbs down`. Następnie weryfikuję, czy bieżąca liczba polubień to 10, i dwukrotnie klikam przycisk `thumbs up`. Po tym działaniu weryfikuję, czy liczba polubień wynosi 11. Wreszcie klikam przycisk `thumbs down` i dokonuję asercji na wartości polubień, która powinna nadal wynosić 11.

W kolejnym przypadku testowym mogę też zweryfikować, czy lokalna wersja `totalGlobalLikes` maleje, gdy użytkownik klika przycisk `thumbs down`.

```
test('zmniejsza liczbę polubień o jeden', () => {
  render(<Vote totalGlobalLikes={10} />)

  expect(screen.getByText(/10/i)).toBeInTheDocument()
  user.click(screen.getByRole('button', { name: /thumbs down/i }))
  expect(screen.getByText(/9/i)).toBeInTheDocument()

  expect(screen.getByRole('button', { name: /thumbs down/i })).toHaveStyle(
    'background: red'
  )
})
```

Klikam tu przycisk `thumbs down` i weryfikuję, czy łączna liczba polubień zmalała z 10 do 9, a kolor tła przycisku zmienił się na czerwony.

Po uruchomieniu wszystkich testów komponentu `Vote` zobaczysz widok podobny do tego poniżej, dzięki czemu będziesz wiedział, że wszystkie testy zakończyły się pomyślnie.

Na rysunku 3.4 widać, że testy „zwiększa liczbę polubień o jeden", „Użytkownik może zagłosować tylko raz" i „zmniejsza liczbę polubień o jeden" z pliku `Vote.test.js` zwróciły pozytywny wynik.

```
PASS src/Vote.test.js
  √ zwiększa liczbę polubień o jeden (222 ms)
  √ zmniejsza liczbę polubień o jeden (80 ms)
  √ Użytkownik może zagłosować tylko raz (60 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        4.909 s
Ran all test suites matching /Vote.test.js/i.

Active Filters: filename /Vote.test.js/
```

Rysunek 3.4. Wynik testów komponentu Vote

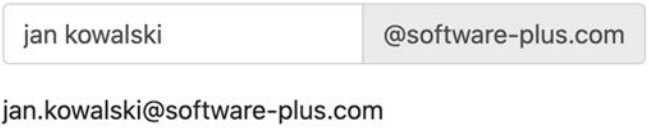
W innym przykładzie mogę stworzyć komponent z polem, w którym pracownicy mają wpisywać swój login (rysunek 3.5).

Rysunek 3.5. Pole przyjmujące login adresu e-mail pracownika

Gdy pracownik wpisuje swój login, komponent skleja go z nazwą strony internetowej firmy i wyświetla rezultat na ekranie (rysunek 3.6).

Rysunek 3.6. Uzupełnione pole przyjmujące login adresu e-mail pracownika

Gdy użytkownik poda swoje imię i nazwisko oddzielone spacją, komponent łączy je znakiem `.` (rysunek 3.7).



Rysunek 3.7. Złączone imię i nazwisko pracownika w polu adresu e-mail

W tej sytuacji mogę skorzystać z metody `type` z `user-event`, by zasymulować wpisywanie tekstu w pole adresu e-mail i dokonać asercji wyniku tego działania:

```
import { render, screen } from '@testing-library/react'
import user from '@testing-library/user-event'
import EmployeeEmail from './EmployeeEmail'

test('przyjmuje nazwę użytkownika i wyświetla ją na ekranie', () => {
  render(<EmployeeEmail />)
  const input = screen.getByRole('textbox', { name: /wpisz imię i nazwisko/i })

  user.type(input, 'anna nowak')

  expect(screen.getByText(/anna.nowak@software-plus.com/i)).toBeInTheDocument()
})
```

Importuję tu `render`, `screen` a także moduł `user-event`. Następnie importuję komponent `EmployeeEmail`. Renderuję go na ekranie. Potem znajduję pole do wpisywania adresu e-mail i przechowuję je w zmiennej. Wreszcie wywołuję metodę `type` z `user-event`, aby wpisać `anna nowak` we wspomniane pole. Na koniec weryfikuję, czy tekst `anna.nowak@software-plus.com` jest w drzewie DOM.

Po uruchomieniu takiego testu w oknie terminala widzę informacje świadczące o tym, że test przebiegł poprawnie (rysunek 3.8).

Rysunek 3.8. Wyniki testu komponentu EmployeeEmail

Powyższy zrzut ekranu udowadnia, że test `przyjmuje nazwę użytkownika i wyświetla ją na ekranie` z pliku `EmployeeEmail.test.js` zwrócił pozytywny wynik. Wiesz już teraz, jak naśladować akcje użytkownika za pomocą modułu `user-event`. Umiejętności zdobyte w tym podrozdziale są kluczowe, ponieważ większość testów zwykle obejmuje różnego rodzaju działania podejmowane w aplikacji przez użytkownika.

W kolejnym podrozdziale nauczysz się, jak testować komponenty, które wywołują procedury obsługi zdarzeń w izolacji.

Testowanie komponentów wywołujących procedury obsługi zdarzeń w izolacji

Bardzo powszechne jest tworzenie komponentów-dzieci, które wywołują metody przekazane im z komponentów-rodziców. W poprzednim podrozdziale widziałeś komponent `Vote`, który obejmował dwa przyciski w tym samym komponencie, co przedstawia także poniższy kod:

```
<button
  onClick={voteLike}
  disabled={hasVoted}
  style={clickedLike ? { background: 'green' } : null}
>
  <img src={thumbsUp} alt="thumbs up" />
</button>
<div>{totalLikes}</div>
<button
  onClick={voteDislike}
  disabled={hasVoted}
  style={clickedDislike ? { background: 'red' } : null}
>
  <img src={thumbsDown} alt="thumbs down" />
</button>
```

Mógłbyś zdecydować się jednak na przeniesienie kodu przycisków do oddzielnych plików, by stały się one niezależnymi komponentami wielokrotnego użytku:

```
const VoteBtn = props => {
  return (
    <button onClick={props.handleVote} disabled={props.hasVoted}>
      <img src={props.imgSrc} alt={props.altText} />
    </button>
  )
}
```

W powyższym kodzie przedstawiono komponent `VoteBtn`, który przyjmuje metodę `handleVote` oraz właściwości `hasVoted`, `imgSrc` i `altText` w obiekcie `props`. Pochodzi on z komponentu-rodzica. Na potrzeby tego podrozdziału skupię się na metodzie `handleVote`. Jest ona wywoływana, gdy wyzwolone zostanie zdarzenie `click` po kliknięciu przycisku. Kiedy metoda ta działa w ramach komponentu `Vote`, jej rezultatem jest zaktualizowanie lokalnej wersji `totalGlobalLikes`. Widok przycisku na ekranie pokazano na rysunku 3.9.

Rysunek 3.9. Przycisk głosowania

Widać tu komponent `Vote` z ikoną `thumbs up`. Aby przetestować komponent `VoteBtn` w odosobnieniu, muszę dostarczyć mu właściwości, ponieważ nie będzie on już obudowany komponentem, który przekazałby je automatycznie. Jest udostępnia funkcjonalności odgrywające rolę dublerów testowych, które zastępują rzeczywiste wersje metod na potrzeby testów.

Dubler testowy to ogólny termin określający to, co zastępuje rzeczywisty obiekt na potrzeby testu. Dubler testowy używany jako znak zastępczy zależności takiej jak interfejs API czy baza danych to **zaślepka**. Gdy jednak używa się go do dokonywania asercji, stosuje się określenie **atrapa**. Przykładowo, możemy skorzystać z funkcji `jest.fn`, by zastąpić w testach `handleVote`:

```
import { render, screen } from '@testing-library/react'
import user from '@testing-library/user-event'
import thumbsUp from './images/thumbs-up.svg'
import VoteBtn from './VoteBtn'
test('po kliknięciu przycisku wywołuje handleVote', () => {
    const mockHandleVote = jest.fn()
    render(
      <VoteBtn
        handleVote={mockHandleVote}
        hasVoted={false}
        imgSrc={stubThumbsUp}
        altText="vote like"
      />
    )
```

W powyższym kodzie zaczynam od zaimportowania metody `render` i obiektu `screen` z React Testing Library. Następnie importuję moduł `user-event`. Wreszcie kolej na obraz `thumbsUp` oraz komponent `VoteBtn`, który chcę poddać testom. Potem wewnątrz metody `test` tworzę funkcję jest, która korzysta z atrapy, i przypisuję ją do zmiennej `mockHandleVote`.

Następnie renderuję komponent `VoteBtn` w drzewie DOM i przekazuję komponent `mockHandle` ↪`Vote` oraz inne właściwości. Na tym etapie kod testu jest już zorganizowany i mogę wykonywać operacje oraz przeprowadzać weryfikacje:

```
    user.click(screen.getByRole('button', { name: /vote like/i }))

      expect(mockHandleVote).toHaveBeenCalled()
      expect(mockHandleVote).toHaveBeenCalledTimes(1)
    })
  })
```

W tym kodzie klikam przycisk o nazwie `vote like`. Na koniec dokonuję dwóch asercji. Pierwsza z nich weryfikuje, czy metoda `mockHandleVote` została wywołana, gdy użytkownik kliknął przycisk. Druga asercja sprawdza, czy metoda ta została wywołana dokładnie raz. Te dwie weryfikacje dotyczące `mockHandleVote` mogą być ważne, jeśli musisz być pewien, że funkcja jest używana poprawnie. Kiedy uruchomię ten test, zobaczę widok taki, jak na rysunku 3.10, informujący mnie, że test przeszedł z powodzeniem:

```
PASS src/VoteBtn.test.js
  Vote Button
    ✓ po kliknięciu przycisku wywołuje handleVote (34 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.734 s, estimated 1 s
Ran all test suites matching /src\/VoteBtn\.test\.js/i.
```

Rysunek 3.10. Wyniki testu komponentu VoteBtn

Na powyższym zrzucie ekranu widać, że test `wywołuje handleVote` z pliku `VoteBtn.test.js` zwrócił pozytywny wynik. Trzeba jednak zauważyć, że choć byłem w stanie zweryfikować, czy procedura obsługi zdarzenia została wywołana, nie byłem w stanie potwierdzić, że po kliknięciu stan przycisku zmienia się na nieaktywny. Musiałbym objąć testem także komponent-rodzica i napisać test integracyjny, by sprawdzić to zachowanie. Dowiesz się, jak poradzić sobie z takimi scenariuszami, z rozdziału 4., „Testy integracyjne i zewnętrzne biblioteki w Twojej aplikacji".

Wiesz już teraz, jak testować procedury obsługi zdarzeń z wykorzystaniem dublerów testowych. W tym podrozdziale poznałeś sposoby symulowania i testowania zachowań użytkownika. Miałeś okazję zobaczyć, jak naśladować zachowania za pomocą `fireEvent` oraz `user-event`. Nauczyłeś się także używać dublerów testowych do testowania procedur obsługi zdarzeń. Umiejętności zdobyte w tym podrozdziale przydadzą Ci się w kolejnej, w której będziesz testował komponenty współpracujące z interfejsami API.

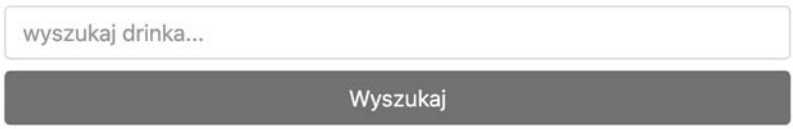
Testowanie komponentów, które współpracują z interfejsami API

Ten podrozdział będzie odwoływać się do wiedzy na temat procedur obsługi zdarzeń zdobytej w poprzednim podrozdziale podczas omawiania testowania komponentów, które wysyłają i otrzymują dane z interfejsów API. W testach jednostkowych komponentów można zmniejszyć ryzyko związane z wytwarzaniem aplikacji dzięki wysiłkom testowym opartym na narzędziach odgrywających rolę dublerów testowych tam, gdzie występują interfejsy API. Posługiwanie się dublerami zamiast prawdziwych API pozwala uniknąć polegania na wolnych połączeniach internetowych czy otrzymywaniu dynamicznych danych, przez co rezultaty testów stają się nieprzewidywalne.

Dowiesz się, jak zainstalować i wykorzystywać *Mock Service Worker* (MSW) jako dublera testowego, by przechwytywać żądania API wysyłane przez komponenty i zwracać dane. W tym podrozdziale będę testował komponent zaprojektowany po to, by użytkownicy mogli wyszukiwać dane na temat drinków z interfejsu API. Pokażę też, jak korzystać z MSW w charakterze serwera w procesie budowania oprogramowania. Pojęcia przedstawione w tym podrozdziale pozwolą Ci zrozumieć, jak weryfikować komunikację między warstwą frontendu a serwerami API.

Żądanie danych z API poprzez fetch

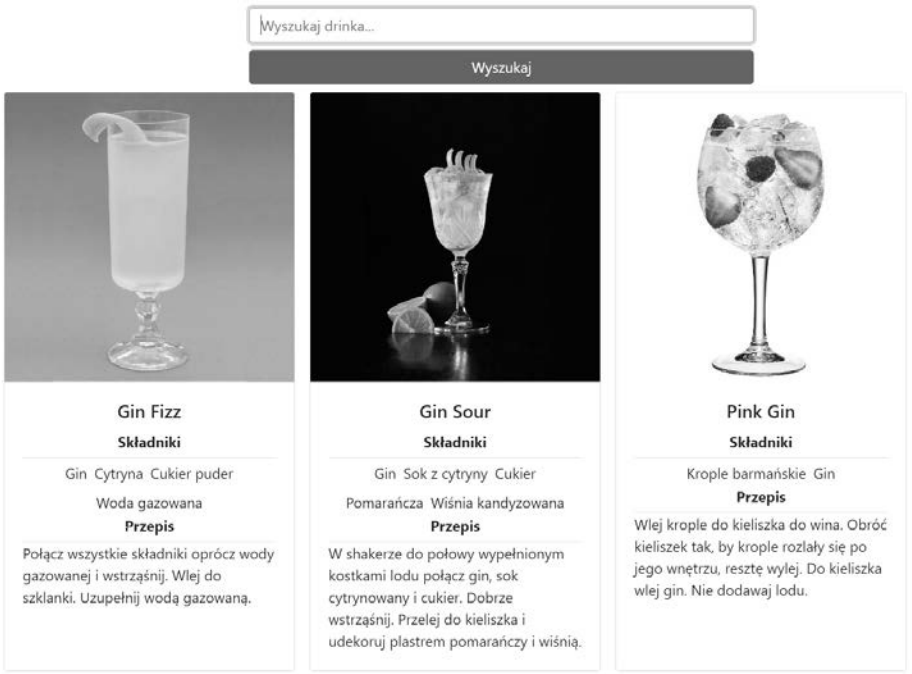
Mogę zbudować komponent, który pozwoli użytkownikowi wyszukiwać drinki w bazie The CockTailDB (*https://www.thecocktaildb.com*), ogólnodostępnej usłudze źródłowej, która będzie pełniła funkcję serwera backendowego. Mój komponent będzie kontaktował się z usługą i żądał danych. Po wyrenderowaniu się komponentu użytkownik będzie widział pole tekstowe i przycisk *Wyszukaj* (rysunek 3.11).



Rysunek 3.11. Komponent wyszukiwania drinków

Po tym, jak użytkownik wyszuka drinki, interfejs API zwróci dane podobne do tych przedstawionych na rysunku 3.12.

Na powyższym zrzucie ekranu widać, że użytkownika interesował gin i że zobaczył on tablicę wyników z API. Jeśli użytkownik szuka drinków, na temat których nie można znaleźć żadnych danych, na ekranie zobaczy komunikat `Nie znaleziono drinków` (rysunek 3.13).



Rysunek 3.12. Wynik wyszukiwania w komponencie Drink

Rysunek 3.13. Brak wyników wyszukiwania

Z kolei gdy użytkownik zleci wyszukiwanie, ale serwer API będzie niedostępny, pojawi się informacja `Usługa niedostępna` (rysunek 3.14).

Rysunek 3.14. Błąd żądania

Taki komponent będzie wykorzystywał moduł `HTTP request`, który został zaprojektowany z myślą o pobieraniu danych na temat drinków z interfejsu API poprzez metodę `fetch`, czyli narzędzie w przeglądarce, które pozwala na wykonywanie żądań HTTP:

```
const fetchDrinks = async drinkQuery => {
  const response = await fetch(
    `https://www.thecocktaildb.com/api/json/v1/1/search.php?s=${drinkQuery}`
  )

  const data = await response.json()
  return data.drinks
}
export default fetchDrinks
```

W powyższym bloku kodu metoda `fetchDrinks` przyjmuje parametr `drinkQuery`, który odpowiada danym wyszukiwania i sprawia, że żądanie API zwraca dane na ten konkretny temat.

Komponent `DrinkSearch` ma taką formę, że po kliknięciu przycisku wywołana zostanie metoda `handleDrinkQuery`, która w końcu wywoła moduł `request` w celu znalezienia danych na temat drinka:

```
<form className="form-group m-auto w-50 pt-2" onSubmit={handleDrinkQuery}>
  <input
    className="form-control"
    placeholder="szukaj drinka..."
    type="search"
    value={drinkQuery}
    onChange={event => setDrinkQuery(event.target.value)}
  />
  <button className="btn btn-primary mt-2 btn-block" type="submit">
    Search
  </button>
</form>
```

Gdy moduł `request` wyśle odpowiedź, która obejmuje tablicę drinków, komponent `DrinkSearch` wywoła metodę `drinkResult`, która wyrenderuje dane ze zmiennej `drinks` na ekranie:

```
{drinks && <div className="d-flex flex-wrap">{drinkResults()}</div>}
```

Jeśli odpowiedź nie zawiera żadnych drinków, renderowany jest kod związany z brakiem rezultatów:

```
{!drinks && <h5 className="text-center mt-5">🍸 Nie znaleziono drinków 🍸</h5>}
```

Z kolei jeśli nastąpi błąd w komunikacji z serwerem, renderowany jest kod związany z niedostępnością usługi:

```
{error && <h5 className="text-center mt-5">🔘 Usługa niedostępna 🔘</h5>}
```

Wiesz już teraz, jak zachowuje się komponent `DrinkSearch` w zależności od interakcji. Przejdę teraz do tworzenia zastępczych danych z API w celu przetestowania komponentu.

Tworzenie atrap danych API z MSW

MSW to narzędzie, którego możesz użyć, by przechwycić żądania API wykonywane przez Twoje komponenty i zwracać atrapy odpowiedzi. Kiedy frontend aplikacji React wykonuje żądanie HTTP do serwera API, MSW przechwyci je, zanim trafi ono do sieci i odpowie atrapą danych. Skorzystaj z poniższego polecenia, by zainstalować MSW w swoim projekcie:

```
npm install msw --save-dev
```

Aby zacząć korzystać z MSW, stworzę atrapę procedury kierowania odpowiedzi, by nadpisywała żądania do konkretnych adresów URL z moich komponentów:

```
import { rest } from 'msw'

export const handlers = [
  rest.get(
    'https://www.thecocktaildb.com/api/json/v1/1/search.php',
    (req, res, ctx) => {
      return res(
        ctx.status(200),
        ctx.json({
          drinks: [
            {
              idDrink: 1,
              strDrinkThumb: './images/thumbs-down.svg',
              strDrink: 'testowy drink',
              strInstructions: 'testowy przepis',
              strIngredient1: 'testowe składniki'
            }
          ]
        })
      )
    }
  )
]
```

W powyższym kodzie importuję `rest` z `msw`. Obiekt `rest` pozwala ustawić typ żądania `request` na `mock` (atrapę). Wewnątrz metody `get` określam przekierowanie, które zostanie nadpisane, gdy wywołane zostanie żądanie `GET`. W parametrze `callback` metody `get` przyjmowane są trzy parametry — parametr `req` dostarczający informacji na temat żądania, np. danych wysłanych w żądaniu. Parametr `res` to funkcja, która zostanie użyta do wykonania atrapy odpowiedzi. Parametr `ctx` odnosi się do kontekstu, jaki wyśle funkcja odpowiedzi.

W ramach `ctx` ustawiam status odpowiedzi na `200`, wskazujący na udane żądanie, a wreszcie buduję obiekt JSON z danych do zwrócenia, którymi będzie tablica drinków. Mogłeś zauważyć, że ścieżka żądania `GET` nie obejmuje całego adresu URL w żądaniu HTTP z poprzedniego punktu. MSW dokona dopasowania adresu URL do wzorca, co powoduje, że wskazywanie pełnego adresu nie jest potrzebne.

Następnie zajmę się stworzeniem atrapy serwera i przekazaniem mu procedury obsługi przekierowania odpowiedzi:

```
import { setupServer } from 'msw/node'
import { handlers } from './handlers'

export const mockServer = setupServer(...handlers)
```

W powyższym kodzie zaczynam do zaimportowania `setupServer` z `msw/node`, z czego będę korzystał do przechwytywania żądań wykonywanych do procedury obsługi przekierowań stworzonej wcześniej. Korzystam z `msw/node`, ponieważ mój kod testowy będzie funkcjonował w środowisku Node.js. Następnie importuję procedury obsługi przekierowań. Wreszcie przekazuję je do `setupServer` i eksportuję kod poprzez zmienną `mockServer`. Gdy kod serwera jest już gotowy, mogę napisać test komponentu `DrinkSearch`.

Testowanie komponentu DrinkSearch

Aby zacząć testowanie komponentu, muszę zaimportować niezbędny kod i uruchomić atrapę serwera:

```
import { render, screen } from '@testing-library/react'
import user from '@testing-library/user-event'
import DrinkSearch from './DrinkSearch'
import { mockServer } from './mocks/server.js'
```

Zaczynam od zaimportowania metody `render` i obiektu `screen` z React Testing Library. Potem importuję moduł `user-event`. Następnie — komponent `DrinkSearch`, który chcę przetestować. Wreszcie importuję `mockServer`, czyli moją atrapę serwera. Muszę go później uruchomić i ustawić tak, by wykonywał konkretne akcje na różnych etapach testu:

```
beforeAll(() => mockServer.listen())
afterEach(() => mockServer.resetHandlers())
afterAll(() => mockServer.close())
```

W tym bloku kodu konfiguruję atrapę serwera, by nasłuchiwała żądań HTTP przed uruchomieniem jakiegokolwiek testu. Następnie wskazuję, by atrapa serwera uruchamiała się ponownie po każdym teście, by uniknąć wpływu poprzedniego testu na kolejny w sekwencji. Wreszcie po wykonaniu wszystkich testów wyłączam serwer. Przejdę teraz do zapisania głównego kodu testów:

```
test('renderuje atrapę danych drinków', async () => {
  render(<DrinkSearch />)
  const searchInput = screen.getByRole('searchbox')

  user.type(searchInput, 'wódka, {enter}')
```

W powyższym bloku kodu renderuję komponent `DrinkSearch`. Następnie lokalizuję pole wyszukiwania i wpisuję w nie słowo `wódka`. Zapis `{enter}` po słowie `wódka` symuluje kliknięcie przycisku `Enter` na klawiaturze. Następnie dokonam asercji na rezultatach działań użytkownika:

```
    expect(
      await screen.findByRole('img', { name: /testowy drink/i })
    ).toBeInTheDocument()
    expect(
      screen.getByRole('heading', { name: /testowy drink/i })
    ).toBeInTheDocument()
    expect(screen.getByText(/testowe składniki/i)).toBeInTheDocument()
    expect(screen.getByText(/testowy przepis/i)).toBeInTheDocument()
  })
```

W powyższym kodzie korzystam z metody wyszukiwania `findByRole`, by znaleźć obrazek. W poprzednich przykładach posługiwałem się tylko wyszukiwaniem za pomocą `getBy*`. Mogą one być stosowane w większości sytuacji, w których spodziewasz się, że element będzie dostępny w bieżącym stanie drzewa DOM. W powyższym kodzie używam natomiast metody `findBy`, ponieważ proces komunikacji z API jest asynchroniczny, więc muszę dać aplikacji czas na odebranie odpowiedzi i zaktualizowanie drzewa DOM, zanim zacznę próbę zlokalizowania elementu.

Podczas korzystania z metod `getBy*` do wybierania elementów zostałby rzucony wyjątek, a test zwróciłby wynik negatywny z powodu nieznalezienia elementu w obecnym drzewie DOM (rysunek 3.15).

Rysunek 3.15. Negatywny wynik testu wyszukiwania drinka

Powyższy zrzut ekranu pokazuje, że test `renderuje atrapę danych drinków` z pliku `DrinkSearch.test.js` zwrócił negatywny wynik. Dane zwracane po teście dają także więcej kontekstu niepowodzenia, informując, że element o nazwie `drink testowy` nie został znaleziony. Metody `findBy*` też zgłoszą wyjątek, ale dopiero po kilku sekundach, umożliwiając elementom pojawienie się na ekranie z opóźnieniem.

Mogę także napisać test weryfikujący przypadek, gdy serwer API nie zwróci żadnych danych na temat mojego zapytania. Mogę zmodyfikować odpowiedź w serwerze MSW uruchamianym dla tego scenariusza:

```
test('renderuje brak rezultatów', async () => {
  mockServer.use(
    rest.get(
      'https://www.thecocktaildb.com/api/json/v1/1/search.php',
      (req, res, ctx) => {
        return res(
          ctx.status(200),
          ctx.json({
            drinks: null
          })
        )
      }
    )
  )
```

W tym kodzie korzystam z metody `use`, by nadpisać domyślne wartości atrapy tak, by zwracały `null`. Jak wspominałem w punkcie „Żądanie danych z API poprzez fetch", komponent będzie zwracał komunikat `Nie znaleziono drinków`, jeśli serwer nie odpowie tablicą drinków. Gdy ustawię już serwer tak, by zwracał właściwe dane, mogę napisać główny kod testu:

```
  render(<DrinkSearch />)
  const searchInput = screen.getByRole('searchbox')

  user.type(searchInput, 'wódka, {enter}')

  expect(
    await screen.findByRole('heading', { name: /nie znaleziono drinków/i })
  ).toBeInTheDocument()
})
```

Renderuję komponent `DrinkSearch` i wyszukuję hasło `wódka` tak, jak w poprzednim teście. Tym razem jednak, zamiast oczekiwać tablicy drinków, spodziewam się komunikatu `Nie znaleziono drinków`.

W kolejnym teście będę weryfikował zachowanie w przypadku niedostępności usługi. Podobnie jak w poprzednim przypadku, zmodyfikuję odpowiedź serwera MSW uruchamianego w scenariuszu testowym:

```
test('renderuje brak dostępności usługi', async () => {
  mockServer.use(
    rest.get(
      'https://www.thecocktaildb.com/api/json/v1/1/search.php',
      (req, res, ctx) => {
        return res(ctx.status(503))
      }
    )
  )
```

Nadpisuję domyślne wartości atrapy w powyższym kodzie tak, by odpowiadały statusem 503 świadczącym o niedostępności usługi. Jak wspominałem w punkcie „Żądanie danych z API poprzez fetch", komponent zwróci komunikat `Usługa niedostępna`, gdy serwer nie będzie odpowiadał. Po ustawieniu serwera tak, by zwracał właściwe dane, mogę napisać główny kod testu:

```
  render(<DrinkSearch />)
  const searchInput = screen.getByRole('searchbox')

  user.type(searchInput, 'wódka, {enter}')

  expect(
    await screen.findByRole('heading', { name: /usługa niedostępna/i })
  ).toBeInTheDocument()
})
```

Podobnie jak we wcześniejszym przypadku, renderuję komponent `DrinkSearch` i wyszukuję hasło `wódka`. W tej chwili oczekuję jednak w dokumencie informacji `Usługa niedostępna`, ponieważ serwer zwrócił status 503.

Ostatni test będzie weryfikował, czy żadne żądanie nie zostaje wysłane, gdy użytkownik próbuje podać puste hasło wyszukiwania:

```
test('zapobiega żądaniom GET, gdy pole wyszukiwania jest puste', async () => {
  render(<DrinkSearch />)
  const searchInput = screen.getByRole('searchbox')

  user.type(searchInput, '{enter}')

  expect(screen.queryByRole('heading')).not.toBeInTheDocument()
})
```

W powyższym kodzie wciskam klawisz *Enter*, nie podając wcześniej żadnego hasła wyszukiwania. Gdy aplikacja ładuje się po raz pierwszy, użytkownik widzi tylko pole wyszukiwania i przycisk. Aplikacja została zaprojektowana do wyświetlania dodatkowych treści, które obejmują nagłówki, po wysłaniu zapytania do interfejsu API. W tym scenariuszu spodziewamy się, że elementu o roli nagłówka nie będzie na ekranie przy wyszukiwaniu za pomocą metody `queryBy*`. To właśnie ona jest zalecana do weryfikowania, czy konkretnego elementu nie ma na ekranie.

W przeciwieństwie do metod `getBy*` oraz `findBy*`, metoda `queryBy*` nie zgłasza wyjątku i nie powoduje negatywnego zakończenia testu, gdy elementu nie ma. Gdy elementu brakuje, metoda `queryBy*` zwróci wartość `null`, dzięki czemu możesz dokonać asercji dotyczącej braku elementu w drzewie DOM i nie spowodować błędu testu. Po uruchomieniu wszystkich testów powinieneś zobaczyć w oknie terminala widok przedstawiony na rysunku 3.16, co da Ci pewność, że wszystkie testy zakończyły się z powodzeniem:

```
PASS src/DrinkSearch.test.js
  √ renderuje atrapę danych drinków (231 ms)
  √ renderuje brak rezultatów (90 ms)
  √ renderuje brak dostępności usługi (76 ms)
  √ zapobiega żądaniom  GET, gdy pole wyszukiwania jest puste (14 ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        4.052 s
Ran all test suites matching /DrinkSearch.test.js/i.

Active Filters: filename /DrinkSearch.test.js/
```

Rysunek 3.16. Pozytywny wynik zestawu testów komponentu DrinkSearch

Powyższy zrzut ekranu pokazuje, że testy `renderuje atrapę danych drinków`, `renderuje brak rezultatów`, `renderuje brak dostępności usługi` i `zapobiega żądaniom GET, gdy pole wyszukiwania jest puste` z pliku `DrinkSearch.test.js` zwróciły poprawne wyniki. Teraz wiesz już, jak stworzyć atrapę serwera za pomocą MSW i testować komponenty, które dokonują żądań API.

Z kolejnego punktu dowiesz się, jak korzystać z MSW na etapie pisania oprogramowania.

Zastosowanie MSW w budowaniu oprogramowania

Poza tym, że moduł MSW oferuje atrapy odpowiedzi HTTP na potrzeby testów, możesz używać go także do naśladowania odpowiedzi na etapie budowania oprogramowania. Korzyścią z posiadania atrapy serwera programistycznego jest fakt, że budowanie i testowanie frontendu możliwe jest nawet wtedy, gdy API serwera backendowego nie jest jeszcze w pełni gotowe. W takim przypadku wystarczy, że będziesz wiedział, jak będzie wyglądała komunikacja i wymiana danych między frontendem a backendem, by stworzyć właściwe atrapy odpowiedzi.

Po pierwsze trzeba dodać plik tzw. *service workera*, który będzie przechwytywał żądania HTTP wykonywane z frontendu aplikacji i odpowiadał atrapą danych. Dokumentacja MSW wskazuje, że taki plik należ zainstalować w publicznym folderze projektu. Wykonaj poniższe polecenie, znajdując się w głównym katalogu swojego projektu, aby zainstalować service workera:

```
npx msw init public/
```

Powyższe polecenie automatycznie pobierze plik service workera i zainstaluje go w folderze `public`. Jeśli korzystasz z `create-react-app` do budowania projektu, ten podkatalog znajduje się w głównym katalogu projektu. Po pobraniu z plikiem nie trzeba robić nic więcej. Następnie musisz zająć się stworzeniem pliku w katalogu `src/mocks`, aby skonfigurować i uruchomić service workera, podobnie jak robiłem to w punkcie „Tworzenie atrap danych API z MSW" wcześniej w tym rozdziale.

W przypadku korzystania z atrapy serwera programistycznego trzeba dokonać pewnych zmian w konfiguracji serwera:

```
import { rest, setupWorker } from 'msw'

const drinks = [
  {
    idDrink: '11457',
    strDrink: 'Gin Fizz',
    strInstructions:
      'Połącz wszystkie składniki oprócz wody gazowanej i wstrząśnij.
      ↪Wlej do szklanki. Uzupełnij wodą gazowaną.',
    strDrinkThumb:
      'https://www.thecocktaildb.com/images/media/drink/drtihp1606768397.jpg',
    strIngredient1: 'Gin',
    strIngredient2: 'Cytryna',
    strIngredient3: 'Cukier puder',
    strIngredient4: 'Woda gazowana'
  },
]
```

W powyższym kodzie importuję `rest` oraz `setupWorker` z `msw`. W punkcie „Tworzenie atrap danych API z MSW" w tym rozdziale importowałem moduły z msw/node, ponieważ testy uruchamiane były w środowisku Node.js. Atrapa serwera programistycznego będzie działała w przeglądarce, więc nie muszę importować wersji Node.js. Teraz przejdę do stworzenia tablicy danych drinków pod nazwą `drinks`. Następnie ustalam ścieżki i odpowiedzi serwera.

```
export const worker = setupWorker(
  rest.get(
    'https://www.thecocktaildb.com/api/json/v1/1/search.php',
    (req, res, ctx) => {
      return res(
        ctx.status(200),
        ctx.json({
          drinks
        })
      )
    }
  )
)
```

W powyższym kodzie tworzę procedurę obsługi ścieżek, aby obsługiwała żądania `GET` wysłane pod adres URL w nadziei na dostęp do API koktajli. Przekazuję tablicę drinków jako dane odpowiedzi. W punkcie „Tworzenie atrap danych API z MSW" w tym rozdziale dzieliłem kod ustawień serwera i procedury obsługi ścieżek na dwa pliki. W przypadku atrapy serwera programistycznego osiągnę jednak ten sam cel, zachowując cały kod ustawień serwera w jednym pliku. Ostatni krok to skonfigurowanie aplikacji tak, by korzystała w atrapy serwera w środowisku programistycznym:

```
if (process.env.NODE_ENV === 'development') {
  const { worker } = require('./mocks/browser')
  worker.start()
}
ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
)
```

W powyższym kodzie konfiguruję serwer tak, by uruchamiał się, gdy zmienna środowiskowa `NODE_ENV` jest ustawiona na `development`, zanim wyrenderuję komponent `App` w drzewie DOM. Aplikacje budowane z `create-react-app` od razu ustawiają zmienną `NODE_ENV` na `development`, więc wystarczy tylko uruchomić aplikację poprzez skrypt `npm start`, co jest typowe dla budowania aplikacji z `create-react-app`.

Wiesz już, jak zbudować atrapę serwera za pomocą MSW, aby testować komponenty, które wysyłają żądania danych z API. Stworzyłeś także serwer MSW odpowiadający spreparowanymi danymi w celu pracy nad wytwarzaniem aplikacji. Wiesz także, kiedy używać metod `findBy*` i `getBy*`.

W tym podrozdziale poznałeś sposoby instalacji i użycia MSW. Testowałeś komponent wykorzystywany do wyszukiwania drinków na podstawie interfejsu API. Zobaczyłeś też, jak używać MSW w charakterze serwera programistycznego. Z kolejnego podrozdziału dowiesz się, jak wykorzystać do pisania testów podejście *test-driven-development*.

Implementacja programowania sterowanego testami

Programowanie sterowane testami (ang. *test-driven-development* — TDD) zakłada, że najpierw pisze się testy jednostkowe, a potem buduje kod tak, by testy te zwracały poprawne wyniki. To podejście pozwala Ci mieć na uwadze, czy kod jest właściwy dla testów, jakie chcesz napisać. Proces ten daje perspektywę, w której dąży się do pisania jak najmniejszej ilości kodu pozwalającej na poprawny przebieg testu. Podejście TDD nazywa się też *Red*, *Green*, *Refactor* (ang. zielony, czerwony, refaktoryzacja). *Red* (z ang. czerwony) odpowiada testowi, który zwraca wynik negatywy, *Green* (zielony) — testowi, który zwraca pozytywny wynik, a *Refactor* (refaktoryzacja) oznacza zmiany w kodzie, które nie powodują, że testy przestaną przechodzić poprawnie. Zwykły przepływ pracy w TDD obejmuje:

1. napisanie testu;
2. wykonanie testu z założeniem, że się ono nie powiedzie;
3. napisanie minimalnej ilości kodu potrzebnej do tego, by test zwrócił pozytywny wynik;
4. ponowne wykonanie testu i weryfikacja założenia, że daje on pozytywny wynik;
5. refaktorowanie kodu w miarę potrzeb;
6. powtarzanie kroków od 2 do 5 w miarę potrzeb.

Można korzystać z React Testing Library do kierowania procesem wytwarzania komponentów React w podejściu TDD. Zacznę od zastosowania TDD do zbudowania komponentu `Vote`, który wprowadziłem już wcześniej w tym rozdziale. Następnie skorzystam z tego samego modelu do stworzenia komponentu `Registration`.

Budowanie komponentu Vote z wykorzystaniem TDD

W podrozdziale „Testowanie komponentów wywołujących procedury obsługi zdarzeń w izolacji" budowałem komponent `VoteButton`, zaczynając od komponentu i dodając testy później. W tym punkcie oprę się na podejściu TDD. Najpierw zaplanuję, jak komponent powinien wyglądać po wyrenderowaniu w drzewie DOM i jakie akcje powinny być dostępne dla użytkownika. Zdecydowałem, że komponent będzie klikalną grafiką. Komponent-rodzic powinien przekazywać źródło grafiki i alternatywny tekst obrazka komponentowi w polu `props`.

Komponent będzie także przyjmował wartość `Boolean` przekazywaną do właściwości `hasVoted`, by móc ustawić stan przycisku na `enabled` lub `disabled`. Jeśli `hasVoted` ma wartość `true`, użytkownik może kliknąć przycisk i wywołać metodę, która obsłuży aktualizację liczby głosów. Następnie piszę testy oparte na tym projekcie. Pierwszy test będzie dotyczył tego, czy komponent renderuje się na ekranie z przekazanymi w `props` wartościami:

```
test('po przekazaniu grafiki i statusu głosu renderuje przycisk na ekranie', () => {
    const stubHandleVote = jest.fn()
    const stubAltText = 'vote like'
```

```
    render(
      <VoteBtn
        handleVote={stubHandleVote}
        hasVoted={false}
        imgSrc={stubThumbsUp}
        altText={stubAltText}
      />
    )
    const image = screen.getByRole('img', { name: stubAltText })
    const button = screen.getByRole('button', { name: stubAltText })
```

W powyższym kodzie najpierw tworzę funkcje jest i przypisuję je do zmiennych `stubHandleVote` oraz `stubAltText`. Do nazw zmiennych dodaję przedrostek *stub*, ponieważ używam ich wyłącznie jako symboli zastępczych zależności w teście. Takie nazwy zmiennych zapewniają także dokładniejszy kontekst w środowisku testowym.

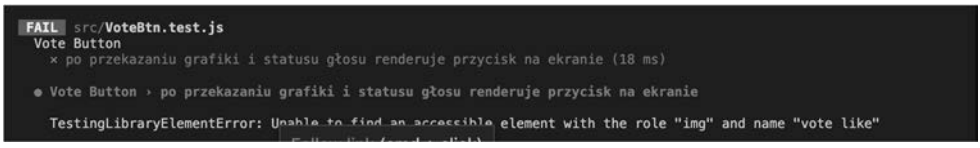
Następnie renderuję komponent z przekazanymi wartościami `props`. Potem znajduję elementy `image` oraz `button` i przypisuję do odpowiednich zmiennych. Wreszcie dokonuję asercji:

```
    expect(image).toBeInTheDocument()
    expect(button).toBeInTheDocument()
    expect(button).toBeEnabled()
  })
```

W tym kodzie dokonuję weryfikacji, czy elementy `image` oraz `button` znajdują się w drzewie DOM. Sprawdzam także, czy stan przycisku to `enabled`, co oznacza, że użytkownik może go kliknąć. Tworzę plik dla komponentu `VoteButton` w następujący sposób:

```
const VoteBtn = props => {
  return null
}
export default VoteBtn
```

W ten sposób tworzę komponent `VoteBtn`, który nie zwraca w tej chwili żadnego kodu do wyrenderowania w drzewie DOM. Eksportuję ten komponent, by móc używać go w innych plikach. Kiedy uruchomię test, zobaczę rezultat pokazany na rysunku 3.17.



Rysunek 3.17. Krok 1 testu TDD VoteButton

Na powyższym zrzucie ekranu widać, że test `po przekazaniu grafiki i statusu głosu renderuje przycisk na ekranie` zwrócił negatywny wynik. Komunikat błędu daje kontekst niepowodzenia, do którego doszło po tym, jak test nie znalazł elementu `image` z nazwą `vote like` w drzewie DOM. Ponieważ wiem, że grafika powinna być dzieckiem elementu `button`, w kolejnym kroku rozwiążę problem, tworząc element `button` z dzieckiem w postaci elementu `image`, a także przekażę im wymagane właściwości w pliku komponentu `VoteBtn`:

```
const VoteBtn = props => {
return (
  <button disabled={props.hasVoted}>
    <img src={props.imgSrc} alt={props.altText} />
  </button>
  )
}
export default VoteBtn
```

W powyższym kodzie tworzę element `button` z dzieckiem `image` i wymaganymi właściwościami `props` dla źródła obrazu, alternatywnym tekstem oraz atrybutem `disabled`. Jeśli uruchomię ten sam test ponownie, zobaczę rezultat przedstawiony na rysunku 3.18.

Rysunek 3.18. Krok 2 testu TDD VoteButton

Na powyższym zrzucie ekranu widać, że test `po przekazaniu grafiki i statusu głosu renderuje przycisk na ekranie` zwrócił pozytywny wynik. Dalszy etap pracy nad `VoteButon` będzie obejmował napisanie kodu pozwalającego użytkownikowi kliknąć przycisk, by wywołać metodę, która obsłuży aktualizację liczby polubień, gdy wartość `hasVoted` ma wartość `true`. Zacznę od napisania kolejnego testu, który będzie weryfikował właśnie tę funkcjonalność:

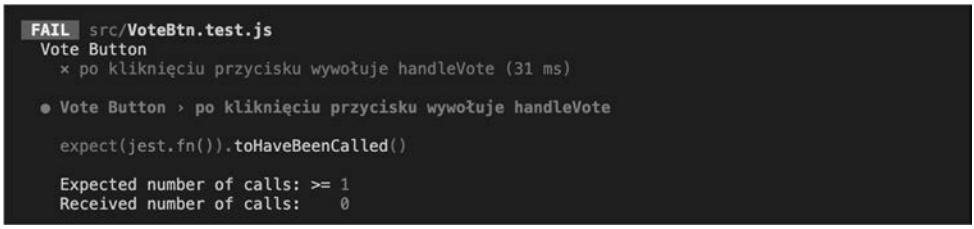
```
test('po kliknięciu przycisku wywołuje handleVote', () => {
  const mockHandleVote = jest.fn()
  render(
    <VoteBtn
      handleVote={mockHandleVote}
      hasVoted={false}
      imgSrc={stubThumbsUp}
      altText="vote like"
    />
  )
```

W powyższym kodzie najpierw tworzę funkcję jest i przypisuję ją do zmiennej o nazwie `mockHandleVote`. Do nazwy zmiennej dołączam przedrostek `mock`, ponieważ będę dokonywał asercji dotyczącej tej zmiennej w teście. Następnie, renderuję komponent `VoteBtn` w drzewie DOM i przekazuję mu wymagane właściwości. Zwróć uwagę, że przekazuję `mockHandleVote` zamiast `handleVote`. Potem klikam przycisk i dokonuję asercji:

```
user.click(screen.getByRole('button', { name: /vote like/i }))

    expect(mockHandleVote).toHaveBeenCalled()
    expect(mockHandleVote).toHaveBeenCalledTimes(1)
  })
})
```

Powyższy kod zaczyna się od kliknięcia przycisku wewnątrz komponentu. Potem weryfikuję, czy wywołana została metoda `mockHandleVote` i doszło do tego dokładnie jeden raz. Sprawdzenie czy i jak wywołana została metoda `mockHandleVote` jest kluczowe. Jeśli metoda ta nie została wywołana lub została wywołana więcej niż raz dla pojedynczego kliknięcia, będę wiedział, że komponent nie będzie komunikował się poprawnie po integracji z komponentem-rodzicem. Po uruchomieniu testu można zobaczyć wynik przedstawiony na rysunku 3.19.

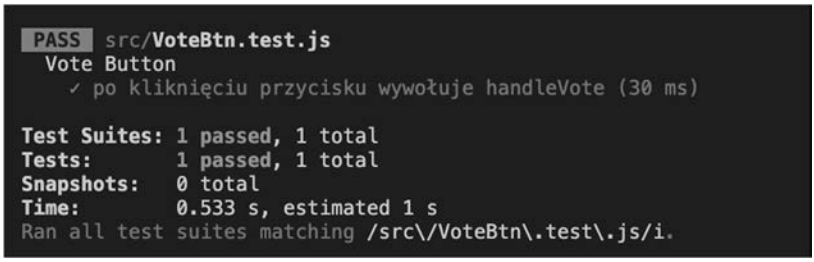


Rysunek 3.19. Krok 3 testu TDD VoteButton

Na powyższym zrzucie ekranu widać, że test `po kliknięciu przycisku wywołuje handleVote` zwraca wynik negatywny. Test oczekiwał, że funkcja jest przekazana do komponentu zostanie wywołana przynajmniej raz, ale nie została ona wywołana wcale. Rozwiążę ten problem poprzez dodanie implementacji do komponentu:

```
<button onClick={props.handleVote} disabled={props.hasVoted}>
```

W tym kodzie z kolei dodałem procedurę obsługi zdarzenia `onClick`, która wywoła metodę `handleVote` przekazaną komponentowi jako właściwość, gdy przycisk zostanie kliknięty. Gdy uruchomię test po tych zmianach, zobaczę komunikat przedstawiony na rysunku 3.20.



Rysunek 3.20. Krok 4 testu TDD VoteButton

Na powyższym zrzucie ekranu widać, że test `po kliknięciu przycisku wywołuje handleVote` zwraca wynik pozytywny. Teraz, gdy wszystkie elementy projektu komponentu `VoteBtn` zostały zaimplementowane i przetestowane, mogę zakończyć pracę nad funkcjonalnością w podejściu TDD.

W kolejnym punkcie użyję TDD do stworzenia komponentu rejestracji.

Budowanie formularza rejestracji z wykorzystaniem TDD

W poprzednim punkcie korzystałem z TDD, by zbudować komponent `Vote`. W tym punkcie nadal będę posługiwał się tym samym podejściem, by zbudować komponent służący użytkownikom do tworzenia kont na stronie internetowej. Po zbudowaniu minimalnej funkcjonalności pozwalającej na to, by testy zwróciły pozytywne wyniki, zajmę się refaktoryzacją implementacji komponentów i weryfikacją, czy nie odbiło się to negatywnie na stanie testów. Komponent będzie zawierał nagłówek *Zarejestruj się*, pola *Adres e-mail* oraz *Hasło* oraz przycisk *Załóż konto*. Kiedy formularz zostanie wysłany, wywołana powinna zostać metoda `handleSubmit`. Ostateczna wersja komponentu powinna wyglądać jak na rysunku 3.21.

Rysunek 3.21. Formularz rejestracji

Na powyższym zrzucie ekranu widać formularz pozwalający użytkownikom podać adres e-mail oraz hasło i zarejestrować swoje konto na stronie. Skoro wiem już, jak powinna wyglądać ostateczna wersja na ekranie, mogę napisać test opierający się na tym projekcie. Na potrzeby tego punktu będę weryfikował, czy po wysłaniu formularza wywoływana jest metoda `handleRegister`:

```
test('po wypełnieniu formularza wywołuje handleRegister', () => {
  const mockHandleRegister = jest.fn()
  const mockUser = {
    email: 'janek@mail.com',
    password: '123'
  }

  render(<Register handleRegister={mockHandleRegister} />)
```

W tym kodzie tworzę zmienne `mockHandleRegister` oraz `mockValues`. Te zmienne posłużą potem do asercji. Następnie renderuję komponent poddawany testom w drzewie DOM i przekazuję mu `mockHandleRegister`. Pozwoli to na przetestowanie komponentu `Register` w izolacji, bez zależności w postaci `handleRegister`. Potem przechodzę do podania wartości w polach formularza:

```
user.type(screen.getByLabelText('Adres e-mail'), mockUser.email)
    user.type(screen.getByLabelText('Hasło'), mockUser.password)
    user.click(screen.getByRole('button', { name: /załóż konto/i }))
```

W tym fragmencie kodu podaję wartości ze zmiennej `mockValues` w pola `email` oraz `password`. Zwróć uwagę na użycie łańcuchów znaków przekazywanych do metody `getByLabelText`. Łańcuchy znaków to jedna z możliwości, gdy nie chcesz posługiwać się wyrażeniami regularnymi. Następnie dokonuję asercji:

```
expect(mockHandleRegister).toHaveBeenCalledWith({
      email: mockUser.email,
      password: mockUser.password
    })
    expect(mockHandleRegister).toHaveBeenCalledTimes(1)
  })
})
```

W powyższym kodzie oczekuję, że metoda `mockHandleRegister` zostanie wywołana tylko raz. Wreszcie oczekuję, że wartości z obiektu `mockValues` stanowiły argumenty przy wywołaniu `mockHandleRegister`. Weryfikowanie argumentów przekazywanych do tej funkcji jest ważne, bo pomaga ograniczyć ryzyko, że wartości z formularza nie będą przekazywane prawdziwej metodzie `handleRegister`.

Następnie stworzę plik dla komponentu `Register`:

```
export default class Register extends React.Component {
  render() {
    return null
  }
}
```

Tworzę i eksportuję komponent `Register`, który nie zwraca w tej chwili żadnego kodu do wyrenderowania w drzewie DOM. Po uruchomieniu testu w oknie terminala zobaczę wynik przedstawiony na rysunku 3.22.

```
FAIL src/Register.test.js
Rejestracja
  × po wypełnieniu formularza wywołuje handleRegister (27 ms)

● Rejestracja › po wypełnieniu formularza wywołuje handleRegister

  TestingLibraryElementError: Unable to find a label with the text of: Adres e-mail
```

Rysunek 3.22. Krok 1 testu TDD rejestracji

Na powyższym zrzucie ekranu widać, że test `po wypełnieniu formularza wywołuje handleRegister` zwrócił wynik negatywny. Komunikat błędu daje kontekst niepowodzenia, do którego doszło po tym, jak test nie znalazł elementu `email` w drzewie DOM. Rozwiążę ten problem poprzez stworzenie tego pola. Dodam także pole *hasło* i przycisk *Załóż konto*:

```
state = {
  email: '',
  password: ''
}
```

```
  handleChange = event => {
    const { id, value } = event.target
    this.setState(prevState => {
      return {
        ...prevState,
        [id]: value
      }
    })
  }
```

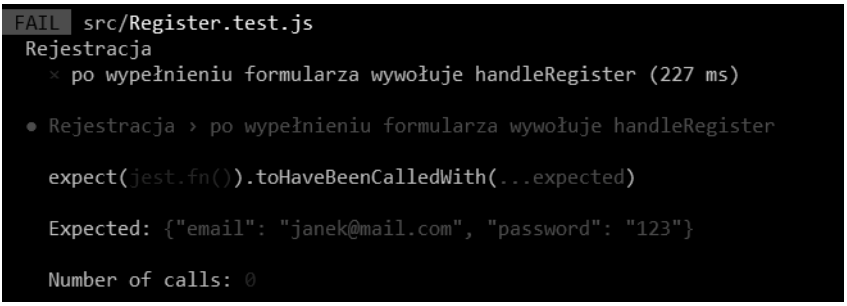
W powyższym kodzie najpierw tworzę obiekt `state` do przechowywania wartości wpisanych w pola `email` i `password`. Następnie piszę metodę `handleChange`, która będzie wywoływana zawsze, gdy użytkownik wpisze wartości w pola formularza. Metoda ta będzie aktualizować wartości w zmiennej `state` w zależności od tego, jak zmienią się pola formularza. Potem przechodzę do zbudowania elementu nagłówka oraz pola `email`:

```
<main>
  <h1>Zarejestruj się</h1>
  <form>
    <div className="form-group">
      <label htmlFor="email">Adres e-mail</label>
      <input
        value={values.email}
        onChange={handleChange}
        type="email"
        className="form-control"
        id="email"
      />
    </div>
```

W powyższym kodzie najpierw tworzę element `main`, który opakowuje elementy `heading` oraz `form`. Wewnątrz dodaję nagłówek `Zarejestruj się`. Potem buduję element `form` i umieszczam w nim pola, w których użytkownik może podać swój adres e-mail oraz hasło. Kiedy użytkownik wpisze tam wartości, procedura obsługi zdarzenia `onChange` zostanie wywołana, by z kolei uruchomić metodę `handleChange` w celu aktualizacji wartości w obiekcie `state`. Atrybut `value` w tych polach zawsze pokazuje bieżącą wartość przechowywaną w obiekcie stanu pod odpowiednim kluczem. Następnie tworzę pole, w którym użytkownicy będą mogli podać swoje hasło, a także przycisk `button` do wysłania formularza:

```
    <div>
      <label htmlFor='password'>Hasło</label>
      <input
        value={this.state.password}
        onChange={this.handleChange}
        type='password'
        id='password'
      />
    </div>
    <button type='submit'>Załóż konto</button>
  </form>
</main>
```

W powyższym kodzie najpierw tworzę pole `password`. Wiąże się z nim ta sama procedura obsługi zdarzenia, co w przypadku pola `email`. Wreszcie dodaję przycisk *Załóż konto*, by umożliwić użytkownikom wysłanie wartości wpisanych w formularzu. Gdy teraz uruchomię mój test, zobaczę dane jak na rysunku 3.23.



Rysunek 3.23. Krok 2 testu TDD rejestracji

Na powyższym zrzucie ekranu widać, że test nadal kończy się niepowodzeniem, ale z innego powodu. W tej chwili test jest w stanie uzupełnić wartości i wysłać formularz, ale metoda `mockHandleRegister` nie zostaje wywołana z właściwymi wartościami. Dzieje się tak, ponieważ nie zaimplementowałem jeszcze procedury obsługi zdarzenia `onSubmit` tak, by wywoływała metodę `mockHandleRegister` lub wykonywała jakiekolwiek inne pożądane po wysłaniu formularza działanie.

Rozwiążę ten problem poprzez dodanie procedury obsługi zdarzenia `onSubmit` w formularzu i sprawię, że będzie ona wywoływała metodę `handleSubmit`, którą dopiszę:

```
handleSubmit = event => {
  event.preventDefault()
  this.props.handleRegister(this.state)
}
```

W powyższym kodzie tworzę metodę `handleSubmit`. Gdy zostaje ona wywołana, przeglądarkowe zdarzenie, które ją wywołało, również jest przekazywane. Potem zapobiegam normalnemu zachowaniu przeglądarki, jakim jest odświeżenie strony po wysłaniu formularza dzięki skorzystaniu z metody `preventDefault`. Wreszcie wywołuję metodę `handleRegister`, przekazaną komponentowi w `props`, i przekazuję wartości z formularza przechowywane w obiekcie `state`. Następnie przypiszę metodę `handleSubmit` formularzowi:

```
<form onSubmit={this.handleSubmit}>
```

Jak widać, dodaję procedurę obsługi zdarzenia `onSubmit` i przekazuję `handleSubmit`. Po wysłaniu formularza zostanie wywołana metoda `handleSubmit`, co poskutkuje wywołaniem metody `handleRegister` z wartościami z formularza w ramach argumentów. Gdy uruchomię test po tych zmianach, zobaczę w oknie terminala widok jak na rysunku 3.24.

```
PASS src/Register.test.js
  Rejestracja
    √ po wypełnieniu formularza wywołuje handleRegister (170 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        3.062 s
Ran all test suites matching /Register.test.js/i.
```

Rysunek 3.24. Krok 3 testu TDD rejestracji

Na powyższym zrzucie ekranu widać, że test wreszcie zwraca pozytywny wynik. Technicznie mógłbym na tym poprzestać, bo kod, który napisałem, powoduje już, że test przechodzi bez zarzutu. Niemniej jednak mogę posprzątać nieco kod komponentu, zamieniając go z komponentu klasowego na komponentu funkcyjny. Dopóki zachowanie pozostaje takie samo, test powinien nadal zwracać pozytywny wynik. Mogę dokonać refaktoryzacji komponentu w następujący sposób:

```
const Register = props => {
  const [values, setValues] = React.useState({
    email: '',
    password: ''
  })
```

W powyższym kodzie zaczynam od zamiany klasy na funkcję. Następnie korzystam z hooka `useState` do zarządzania stanem wartości formularza. Następnie zmodyfikuję metody `handleChange` i `handleSubmit`:

```
const handleChange = event => {
  const { id, value } = event.target
  setValues({ ...values, [id]: value })
}
const handleSubmit = event => {
  event.preventDefault()
  props.handleRegister(values)
}
```

W powyższym kodzie `handleChange` i `handleSubmit` zostają zamienione na funkcje. Metoda `handleChange` wywołuje `setValues`, by zmienić stan dla każdej wartości wprowadzonej w formularzu. Implementacja `handleSubmit` jest dokładnie taka sama w obu wersjach. Dokonam także zmian w kodzie zwracanym do wyrenderowania jako HTML w przeglądarce:

```
<main className="m-3 d-flex flex-column">
  <h1>Zarejestruj się</h1>
  <form onSubmit={handleSubmit}>
    <div>
      <label htmlFor="email">Adres e-mail</label>
      <input
        value={values.email}
        onChange={handleChange}
// pozostały kod komponentu ...
```

Zaczynam od usunięcia metody `render` wymaganej w komponencie klasowym. Reszta kodu jest natomiast bardzo zbliżona do pierwotnej wersji, chociaż atrybut `value` korzysta z obiektu `values`, a metoda `handleChange` przekazywana do procedury obsługi zdarzenia `onChange` nie wymaga słowa kluczowego `this`. Po ponownym wykonaniu testu otrzymam wynik jak na rysunku 3.25.

```
PASS src/Register.test.js
  Rejestracja
    √ po wypełnieniu formularza wywołuje handleRegister (170 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        3.062 s
Ran all test suites matching /Register.test.js/i.
```

Rysunek 3.25. Krok 4 testu TDD rejestracji

Powyższy zrzut ekranu udowadnia, że test pomimo refaktoryzacji kodu komponentu nadal zwraca pozytywny wynik. Dzięki dokonanym zmianom kod komponentu jest za to znacznie czystszy. Teraz wiesz już, jak budować komponenty z wykorzystaniem podejścia TDD i biblioteki React Testing Library. W tym podrozdziale kierowałem się podejściem TDD w budowaniu funkcjonalności głosowania i rejestrowania konta. Wyniki testów w React Testing Library i informacja zwrotna, jakiej dostarczają, to przyjemne doświadczenie w procesie wytwarzania oprogramowania.

Podsumowanie

W tym rozdziale nauczyłeś się instalować i wykorzystywać moduły symulujące działania użytkowników na wyrenderowanych w drzewie DOM komponentach aplikacji. Umiesz teraz zainstalować i przetestować funkcjonalności związane z interakcjami z interfejsami API przy użyciu przyjaznych dla użytkownika narzędzi. Wiesz także, jak testować komponenty w izolacji od zależności w postaci procedur obsługi zdarzeń dzięki atrapom. Wreszcie poznałeś podejście TDD pozwalające budować funkcjonalności w połączeniu z React Testing Library.

W kolejnym rozdziale poznasz korzyści płynące z testów integracyjnych. Dowiesz się także, jak testować komponenty React, które wykorzystują popularne zewnętrzne biblioteki.

Pytania

1. Dlaczego należy korzystać z user-event, a nie fireEvent do symulowania akcji użytkownika w testach?

2. Wyjaśnij, jak MSW pozwala testować komponenty, które wykonują żądania do interfejsów API.
3. Czym jest atrapa funkcji?
4. Wyjaśnij, na czym polega ryzyko związane z testowaniem komponentów w izolacji z atrapami funkcji.
5. Opisz swoimi słowami przepływ pracy w podejściu TDD.
6. Wyjaśnij, kiedy używać `getBy*`, `findBy*` i `queryBy` do szukania elementów.

React: z łatwością napiszesz doskonałe komponenty testowe!

W ostatnich latach zdecydowanie wzrosła popularność frameworka React, pozwalającego w pełni skorzystać z możliwości nowoczesnych przeglądarek i urządzeń mobilnych. Nowym narzędziem służącym do testowania obiektowego modelu dokumentu (DOM) jest React Testing Library. Zostało ono zaprojektowane tak, aby ułatwiać projektantom pracę zgodną z podejściem *test-driven development* (TDD). Jest to bardzo cenna właściwość, gdyż prawidłowo napisane komponenty testowe znacznie obniżają ryzyko wystąpienia poważnych błędów aplikacji.

W tej książce przystępnie wyjaśniono, w jaki sposób używać nowoczesnego narzędzia, jakim jest React Testing Library (RTL), do testowania komponentów napisanych w React. Dzięki niej zrozumiesz główne aspekty działania tej biblioteki. Nauczysz się symulować interakcje użytkownika i zrozumiesz podejście TDD. Znajdziesz tu wskazówki, jak tworzyć testy jednostkowe komponentów, które wchodzą w interakcje z interfejsami API, a także jak testować komponenty za pomocą takich bibliotek jak GraphQL i Redux. Zapoznasz się też z technikami testowania aplikacji, które podległy poważnej refaktoryzacji. Na koniec dowiesz się, jak pisać całościowe testy funkcjonalne z wykorzystaniem biblioteki Cypress i które wzorce projektowe są najbardziej przydatne do ustrukturyzowania profesjonalnego zbioru testów.

W książce między innymi:

- wprowadzenie do RTL i jego zastosowań
- użycie *jest-dom* do rozbudowy testów opartych na RTL
- techniki tworzenia komponentów testowych łatwych do modyfikacji
- integracja zestawów testowych z Cucumber i Cypress
- podejście TDD

Scottie Crump jest starszym inżynierem testów. Od kilku lat doradza klientom z różnych branż, między innymi handlowej, motoryzacyjnej, telekomunikacyjnej, ochrony zdrowia. Bierze też udział w ważnych projektach z zakresu zapewniania jakości oprogramowania. Jego pasją jest popularyzowanie najlepszych praktyk branży i technik wspierania procesów decyzyjnych za pomocą danych.

Helion

helion.pl

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-8872-7

Cena: 59,00 zł

Packt>