O'REILLY®

React Receptury

Poradnik dla zaawansowanych

David Griffiths
Dawn Griffiths

Helion

Tytuł oryginału: React Cookbook: Recipes for Mastering the React Framework

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-8763-8

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)

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

Printed in Poland.

Spis treści

ROZDZIAŁ 1.

Tworzenie aplikacji

React jest frameworkiem do tworzenia aplikacji cechującym się wyjątkowo dużymi możliwościami adaptacji do przeróżnych zastosowań. Programiści używają go zarówno do tworzenia dużych i złożonych *aplikacji jednostronicowych* (ang. *Single-Page Applications*, w skrócie *SPA*), jak również do tworzenie zadziwiająco małych wtyczek. Bez trudu można go wykorzystać do osadzania kodu wewnątrz aplikacji napisanych we frameworku Rails oraz do generowania aplikacji internetowych o bogatej treści.

W tym rozdziale przyjrzymy się różnym sposobom tworzenia aplikacji Reacta. Przedstawimy także wybrane przydatne narzędzia, które być może zdecydujesz się dodać do swojego cyklu pracy. Obecnie już mało kto tworzy projekty pisane w języku JavaScript w całości od podstaw. Takie rozwiązania są czasochłonne i wymagają tak wiele przygotowań i konfiguracji, że ich stosowanie jest niepraktyczne. Na szczęście istnieją narzędzia służące do generowania kodu, który będzie w stanie sprostać naszym oczekiwaniom niemal we wszystkich przypadkach.

Zróbmy zatem krótki przegląd różnych sposobów rozpoczynania przygody z Reactem, zaczynając od najbardziej popularnego narzędzia: `create-react-app`.

1.1. Generowanie prostej aplikacji

Problem

Tworzenie i konfigurowanie projektów aplikacji Reacta od podstaw jest dużym wyzwaniem. Wymaga podjęcia wielu decyzji projektowych — takich jak wybór dołączanych bibliotek, używanych narzędzi czy też uwzględnianych możliwości języka, a ponadto takie własnoręcznie tworzone aplikacje siłą rzeczy będą się od siebie różnić. Dziwactwa związane z różnymi projektami wpływają także na wydłużenie czasu koniecznego do osiągnięcia przez programistów wysokiej produktywności.

Rozwiązanie

`create-react-app` jest narzędziem do tworzenia aplikacji jednostronicowych o standardowej strukturze i określonym zestawie domyślnych opcji. W wygenerowanych projektach używa się do budowania, testowania i uruchamiania kodu biblioteki React Scripts. Projekty dysponują standardową konfiguracją Webpack oraz zestawem standardowo włączonych możliwości języka.

Każdy programista, który kiedyś pracował nad jedną aplikacją utworzoną przy użyciu narzędzia `create-react-app`, będzie czuł się jak w domu, gdy zacznie pracować nad jakąkolwiek inną. Będzie doskonale rozumiał strukturę projektu i wiedział, z których możliwości języka może korzystać. Narzędzie to jest proste w użyciu i zawiera wszystko to, czego może wymagać typowa aplikacja: zaczynając od konfiguracji narzędzia Babel, przez pliki wczytujące, a kończąc na bibliotekach do testowania i serwerze roboczym używanym podczas prowadzenia prac nad aplikacją.

Jeśli dopiero zaczynasz przygodę z Reactem lub jeśli musisz utworzyć aplikację jednostronicową ogólnego przeznaczenia i chcesz to zrobić możliwie jak najprościej, to zdecydowanie powinieneś rozważyć użycie `create-react-app`.

Polecenie `create-react-app` można zainstalować globalnie na komputerze, jednak takie rozwiązanie nie jest zalecane. Zamiast tego lepiej jest tworzyć nowy projekt, wywołując polecenie `create-react-app` za pośrednictwem programu `npx`. Zastosowanie `npx` zapewni, że do wygenerowania projektu aplikacji zostanie użyta najnowsza wersja `create-react-app`:

```
$ npx create-react-app moja-aplikacja
```

To polecenie tworzy nowy projekt aplikacji Reacta w katalogu *moja-aplikacja*. Domyślnie aplikacja będzie używać języka JavaScript. Jeśli jednak chcesz tworzyć aplikację w języku TypeScript, to narzędzie `create-react-app` zapewnia taką możliwość:

```
$ npx create-react-app --template typescript moja-aplikacja
```

Narzędzie `create-react-app` zostało opracowane przez Facebooka, więc nie powinno być żadnym zaskoczeniem, że jeśli na komputerze będzie zainstalowany menedżer pakietów `yarn`, to właśnie on będzie używany domyślnie. Aby zastosować menedżera pakietów `npm`, możesz użyć flagi `--use-npm` bądź też przejść do katalogu projektu, usunąć plik *yarn.lock*, a następnie ponownie zainstalować zależności za pomocą polecenia `npm`:

```
$ cd moja-aplikacja
$ rm yarn.lock
$ npm install
```

Aby uruchomić aplikację, wystarczy wykonać polecenie:

```
$ npm start     # lub yarn start
```

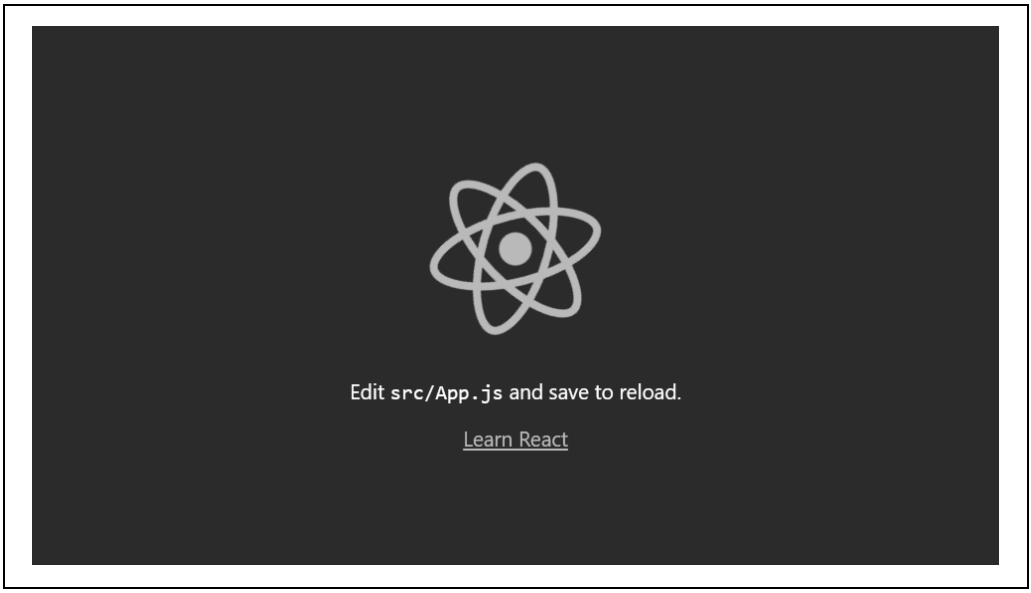
Wykonanie tego polecenia spowoduje uruchomienie serwera roboczego na porcie 3000, otwarcie przeglądarki i wyświetlenie w niej strony głównej aplikacji, która będzie wyglądać jak ta przedstawiona na rysunku 1.1.

Serwer udostępnia aplikację w formie jednej dużej wiązki kodu JavaScript. Kod aplikacji montuje wszystkie umieszczone w nim komponenty wewnątrz przedstawionego poniżej elementu `<div/>`, umieszczonego w pliku *public/index.html*:

```
<div id="root"></div>
```

Kod służący do generowania komponentów rozpoczyna się w pliku *src/index.js* (lub *src/index.tsx*, jeśli używamy języka TypeScript):

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
```



Rysunek 1.1. Wygenerowana strona główna aplikacji

```
import App from './App';
import reportWebVitals from './reportWebVitals';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

Ten plik nie robi wiele więcej oprócz wyrenderowania pojedynczego komponentu o nazwie `<App/>`, którego kod jest zapisany w tym samym katalogu, w pliku *App.js* (lub *App.tsx*):

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.tsx</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
```

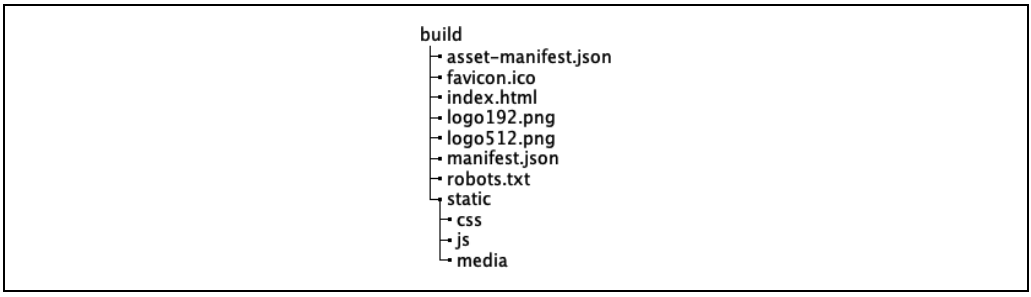
```
            target="_blank"
            rel="noopener noreferrer"
          >
            Learn React
          </a>
        </header>
      </div>
    );
  }

  export default App;
```

Jeśli zaczniesz modyfikować tę stronę, kiedy aplikacja będzie uruchomiona, to strona w przeglądarce będzie automatycznie aktualizowana.

Kiedy już będziemy gotowi przekazać kod aplikacji do środowiska produkcyjnego, konieczne będzie wygenerowanie zestawu statycznych plików umożliwiających wdrożenie na standardowym serwerze WWW. Do tego celu służy skrypt `build`:

```
$ npm run build
```

Wykonanie tego skryptu spowoduje utworzenie katalogu *build*, a następnie opublikowanie w nim zestawu statycznych plików (został on przedstawiony na rysunku 1.2).



Rysunek 1.2. Wygenerowane treści umieszczone w katalogu build

Kilka tych plików jest kopiowanych z katalogu *public/*. Kod aplikacji jest transpilowany do postaci kodu JavaScript, który będzie można wykonywać w przeglądarce, i zapisywany w jednym lub w kilku plikach w katalogu *static/js*. Arkusze stylów używane w aplikacji są łączone i umieszczane w katalogu *static/css*. W nazwach kilku wygenerowanych plików pojawiają się identyfikatory o postaci kodu mieszającego, dzięki czemu, po opublikowaniu aplikacji, przeglądarka pobierze najnowsze kody, a nie jakąś ich starszą wersję z pamięci podręcznej.

Analiza

`create-react-app` nie jest jedynie narzędziem do generowania nowych aplikacji; jest także platformą zapewniającą, że aplikacja będzie aktualna pod względem wykorzystania najnowszych narzędzi i bibliotek. Bibliotekę *react-scripts* możemy aktualizować jak każdą inną: wystarczy zmienić numer wersji i ponownie wykonać polecenie `npm install`. Nie musimy zarządzać listą pakietów narzędzia Babel, bibliotekami używanymi przez postcss ani zarządzać złożonym plikiem konfiguracyjnym *webpack.config.js*. Biblioteka *react-script* robi to wszystko za nas.

Oczywiście wszystkie te ustawienia konfiguracyjne istnieją, lecz zostały ukryte głęboko w katalogu react-scripts. *Można w nim znaleźć plik* webpack.config.js*, zawierający wszystkie ustawienia konfi*guracyjne narzędzia Babel, jak również wszystkie pliki wczytujące, których będzie używać aplikacja. A ponieważ React Scripts jest biblioteką, można ją aktualizować jak wszystkie inne zależności projektu.

Jeśli jednak później zdecydujemy ręcznie zarządzać wszystkimi tymi ustawieniami, możemy to zrobić. Wystarczy w tym celu wykonać polecenie `eject`, a wszystko wróci pod naszą kontrolę:

```
$ npm run eject
```

Jednak jest to operacja jednorazowa. Jej efektów nie można w żaden sposób cofnąć. Dlatego też użycie tej opcji należy dokładnie przemyśleć. Może się okazać, że ustawienie konfiguracyjne, którego potrzebujemy, już jest dostępne. Na przykład programiści często korzystali z polecenia `eject`, by zmienić używany język na TypeScript. Jednak obecnie dzięki opcji `--template typescript` nie jest to już konieczne.

Kolejnym popularnym powodem stosowania polecenia `eject` była chęć korzystania z usług pośredniczących. Aplikacje Reacta często muszą łączyć się z jakimś odrębnym API działającym na innym komputerze. Programiści robili to wcześniej, konfigurując Webpack w taki sposób, by w komunikacji ze zdalnym serwerem pośredniczył lokalny serwer używany do prac programistycznych. Obecnie można jednak uniknąć takich komplikacji, dodając poniższe ustawienie do pliku *package.json*:

```
"proxy": "http://mojserwerapi",
```

W razie jego użycia, jeśli kod spróbuje nawiązać połączenie z adresem URL, którego nie może znaleźć lokalnie (*/api/costam*), to biblioteka *react-scripts* automatycznie wygeneruje żądanie do *http://mojserwerapi/api/costam*.

Jeśli tylko możesz, unikaj stosowania polecenia `eject`. Przejrzyj dokumentację narzędzia `create-react-app` (*https://create-react-app.dev*), aby przekonać się, czy istnieje jakiś inny sposób wprowadzenia niezbędnych zmian.

Kody źródłowe do tej receptury możesz znaleźć w przykładach dołączonych do książki; w katalogu *r01-01-tworz-aplk-reacta* znajdują się kody aplikacji tworzonej w języku JavaScript, a w katalogu *r01-01-tworz-apl-reacta-ts* kody aplikacji tworzonej w języku TypeScript.

1.2. Tworzenie aplikacji o bogatych treściach z zastosowaniem narzędzia Gatsby

Problem

Witryny o bogatych treściach (ang. *content-rich sites*), takie jak blogi oraz internetowe sklepy, muszą w wydajny sposób udostępniać duże ilości złożonych treści. Narzędzia takie jak `create-react-app` nie nadają się do stosowania w takich aplikacjach, gdyż dostarczają wszystko w formie jednej, dużej wiązki kodu JavaScript, którą przeglądarka musi wczytać, zanim będzie mogła cokolwiek wyświetlić.

Rozwiązanie

Jeśli tworzysz witrynę o bogatych treściach, możesz rozważyć zastosowanie narzędzia o nazwie Gatsby.

Gatsby koncentruje się na wczytywaniu, przekształcaniu oraz dostarczaniu treści w możliwie jak najbardziej efektywny sposób. Narzędzie to pozwala na generowanie statycznych wersji stron WWW, co oznacza, że czasy odpowiedzi witryn tworzonych przy jego użyciu są niejednokrotnie znacząco dłuższe niż w aplikacjach tworzonych z wykorzystaniem `create-react-app`.

Gatsby posiada wiele wtyczek, które mogą efektywnie wczytywać i przekształcać lokalne dane statyczne, dane pochodzące ze źródeł GraphQL oraz systemów zarządzania treścią innych firm, takich jak WordPress.

Narzędzie Gatsby można zainstalować globalnie, lecz równie dobrze można je uruchamiać przy użyciu polecenia `npx`:

```
$ npx gatsby new moja-aplikacja
```

Polecenie `gatsby new` powoduje utworzenie nowego projektu w podanym katalogu (w powyższym przykładzie będzie to katalog *moja-aplikacja*). Podczas pierwszego uruchomienia tego polecenia zostaniesz zapytany o to, którego menedżera pakietów chcesz używać: `yarn` czy `npm`.

Aby uruchomić wygenerowaną aplikację w trybie do prowadzenia prac programistycznych, przejdź do utworzonego katalogu i wykonaj następujące polecenia:

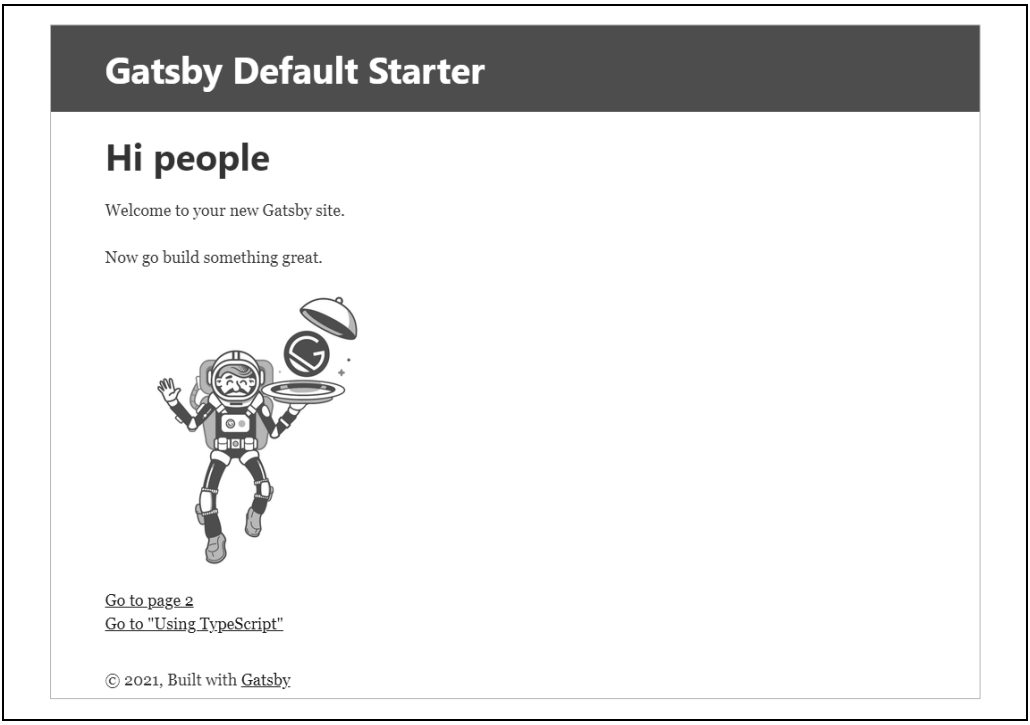
```
$ cd moja-aplikacja
$ npm run develop
```

Aby wyświetlić aplikację, wpisz w przeglądarce adres **`http://localhost:8000/`**. Domyślną stronę aplikacji wygenerowanej przy użyciu narzędzia Gatsby przedstawiłem na rysunku 1.3.

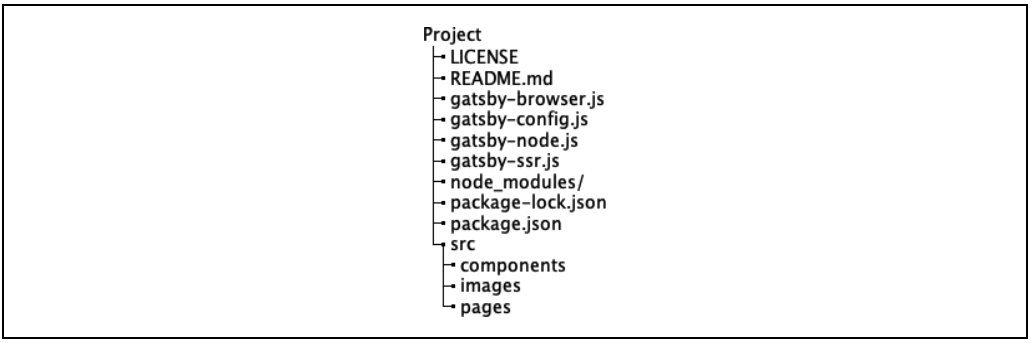
Projekty tworzone przy użyciu narzędzia Gatsby mają bardzo prostą strukturę, przedstawioną na rysunku 1.4.

Główna część aplikacji znajduje się w katalogu *src*. Każda jej strona posiada swój własny komponent Reacta. Poniższy listing przedstawia domyślną postać strony początkowej, której kod jest zapisany w pliku *index.js*:

```
import * as React from "react"
import { Link } from "gatsby"
import { StaticImage } from "gatsby-plugin-image"

import Layout from "../components/layout"
import Seo from "../components/seo"

const IndexPage = () => (
  <Layout>
    <Seo title="Home" />
    <h1>Hi people</h1>
    <p>Welcome to your new Gatsby site.</p>
    <p>Now go build something great.</p>
    <StaticImage
      src="../images/gatsby-astronaut.png"
      width={300}
      quality={95}
```



Rysunek 1.3. Domyślna strona aplikacji Gatsby'ego dostępna pod adresem http://localhost:8000



Rysunek 1.4. Struktura plików i katalogów aplikacji Gatsby'ego

```
        formats={["auto", "webp", "avif"]}
        alt="A Gatsby astronaut"
        style={{ marginBottom: `1.45rem` }}
      />
      <p>
        <Link to="/page-2/">Go to page 2</Link> <br />
        <Link to="/using-typescript/">Go to "Using TypeScript"</Link>
      </p>
    </Layout>
  )

  export default IndexPage
```

Nie na potrzeby tworzenia trasy pozwalającej wyświetlić stronę — każdemu komponentowi strony jest bowiem automatycznie przypisywana trasa. Na przykład strona zapisana w pliku *src/pages/using-typescript.tsx* będzie automatycznie dostępna pod adresem *using-typescript*[1]. Takie rozwiązanie ma wiele zalet. Przede wszystkim jeśli w aplikacji istnieje wiele stron, nie trzeba ręcznie zarządzać ich trasami. Poza tym oznacza to, że Gatsby może znacznie szybciej dostarczać treści. Aby przekonać się, dlaczego tak się dzieje, zobaczmy, jak można wygenerować wersję aplikacji Gatsby'ego przeznaczoną do zastosowań produkcyjnych.

Kiedy zatrzymasz serwer Gatsby'ego do prowadzenia prac programistycznych[2], będziesz mógł wygenerować produkcyjną wersję aplikacji, używając następującego polecenia:

```
$ npm run build
```

To polecenie odpowiada wykonaniu polecenia `gatsby build`, które tworzy katalog *public*. I to właśnie w tym katalogu kryje się cała magia Gatsby'ego. Dla każdej strony aplikacji są w nim umieszczone dwa pliki. Pierwszym jest wygenerowany plik JavaScript:

```
1389 06:48 component---src-pages-using-typescript-tsx-93b78cfadc08d7d203c6.js
```

W powyższym wierszu widać, że kod pliku *using-typescript.tsx* ma jedynie 1389 bajtów długości, co w połączeniu z jądrem frameworka w zupełności wystarcza do stworzenia strony. Jak widać, nie mamy tu do czynienia z tymi pojedynczymi, wielkimi plikami skryptów generowanymi w projektach tworzonych przy użyciu `create-react-app`.

Oprócz tego dla każdej strony tworzony jest katalog zawierający wygenerowany plik HTML. Na przykład dla strony *using-typescript.tsx* zostanie wygenerowany plik *src/using-typescript/index.html*, zawierający statyczną wersję strony. Ten plik zawiera kod HTML, który komponent *using-typescript.tsx* generuje dynamicznie. Na samym końcu tego pliku wczytywana jest strona w wersji kodu JavaScript, która generuje wszelkie dynamiczne treści.

Ta struktura plików oznacza, że strony witryny tworzonej przy użyciu Gatsby'ego są wczytywane tak szybko jak strony statyczne. Korzystając z dołączonej biblioteki `react-helmet`, można także wygenerować znaczniki `<meta/>` z dodatkowymi informacjami o witrynie. Obie te możliwości są niezwykle ważne i przydatne z punktu widzenia optymalizacji pod kątem wyszukiwarek internetowych (SEO).

Analiza

A w jaki sposób do aplikacji Gatsby'ego są dostarczane treści? Można w tym celu użyć „bezgłowego" (ang. *headless*) systemu zarządzania treścią, usługi GraphQL, statycznego źródła danych bądź też wielu innych dostępnych sposobów. Na szczęście Gatsby zapewnia wiele wtyczek, które pozwalają na podłączanie źródeł danych do aplikacji, a następnie przekształcanie pobieranych z nich treści w innych formatach do postaci kodu HTML.

Pełną listę wtyczek dostępnych w narzędziu Gatsby można znaleźć w witrynie poświęconej temu narzędziu, na stronie *https://www.gatsbyjs.com/docs/plugins/*.

[1] Owszem, to także oznacza, że Gatsby ma wbudowane wsparcie dla języka TypeScript.

[2] W większości systemów operacyjnych można to zrobić, naciskając kombinację klawiszy *Ctrl+C*.

W większości przypadków niezbędne wtyczki wybiera się podczas tworzenia projektu. Aby ułatwić rozpoczęcie pracy, Gatsby udostępnia tzw. *szablony startowe* (ang. *start templates*). Zawierają one początkową strukturę i konfigurację aplikacji. Aplikacja, którą zbudowaliśmy wcześniej w tej recepturze, korzystała z domyślnego szablonu startowego, który jest bardzo prosty. Wtyczki używane przez aplikację są skonfigurowane w pliku *gatsby-config.js*, umieszczonym w katalogu głównym aplikacji.

Dostępnych jest jednak naprawdę wiele szablonów startowych Gatsby'ego, skonfigurowanych wstępnie do budowania aplikacji korzystających z różnych źródeł danych, aplikacji posiadających określone ustawienia SEO, używających konkretnych arkuszy stylów, aplikacji korzystających z przechowywania danych pamięci podręcznej, tak by można ich było używać bez połączenia z serwerem, progresywnych aplikacji internetowych (PWA) i wielu innych. Niezależnie od tego, jakiego rodzaju aplikację tworzymy, na pewno znajdziemy szablon startowy zbliżony do naszych potrzeb.

W witrynie Gatsby'ego dostępna jest strona poświęcona szablonom startowym (*https://oreil.ly/vwUd8*), jak również ściągawka z informacjami o najbardziej użytecznych narzędziach (*https://www.gatsbyjs.com/docs/cheat-sheet/*).

Kody źródłowe do tej receptury są dostępne w przykładach dołączonych do książki, w katalogu *r01-02-gatsby*.

1.3. Tworzenie uniwersalnych aplikacji przy użyciu Razzle

Problem

Czasami, kiedy rozpoczynamy budowanie aplikacji, nie jest do końca jasne, jakie ważne decyzje projektowe będziemy musieli podjąć. Czy powinniśmy utworzyć aplikację jednostronicową — SPA? Jeśli wydajność ma krytyczne znaczenie, to czy do generowania danych po stronie serwera należy użyć języka R? Musimy zdecydować, jak będzie wyglądać platforma do prowadzenia prac programistycznych oraz czy kod aplikacji będzie pisany w języku JavaScript czy TypeScript.

Wiele narzędzi wymaga udzielania odpowiedzi na te pytania na wczesnych etapach prac nad projektem. Jeśli później zmienimy zdanie, to odpowiednia modyfikacja sposobu budowania i wdrażania aplikacji może być skomplikowana.

Rozwiązanie

Jeśli chcesz odłożyć decyzję dotyczącą sposobu budowania i wdrażania aplikacji na później, powinieneś rozważyć zastosowanie narzędzia Razzle (*https://github.com/jaredpalmer/razzle*).

Razzle jest narzędziem do budowania aplikacji uniwersalnych (*https://en.wikipedia.org/wiki/Isomorphic_JavaScript*), czyli aplikacji, których kod napisany w języku JavaScript może być na serwerze. Albo po stronie klienta, albo po stronie klienta i na serwerze.

Razzle używa architektury bazującej na wtyczkach, by zapewnić twórcom możliwość zmiany decyzji dotyczących sposobu budowania aplikacji. Pozwala nawet zmieniać zdanie co do tego, jakiego frameworka chcemy używać: Reacta, Preacta czy też jakiegoś zupełnie innego, na przykład Elm lub Vue.

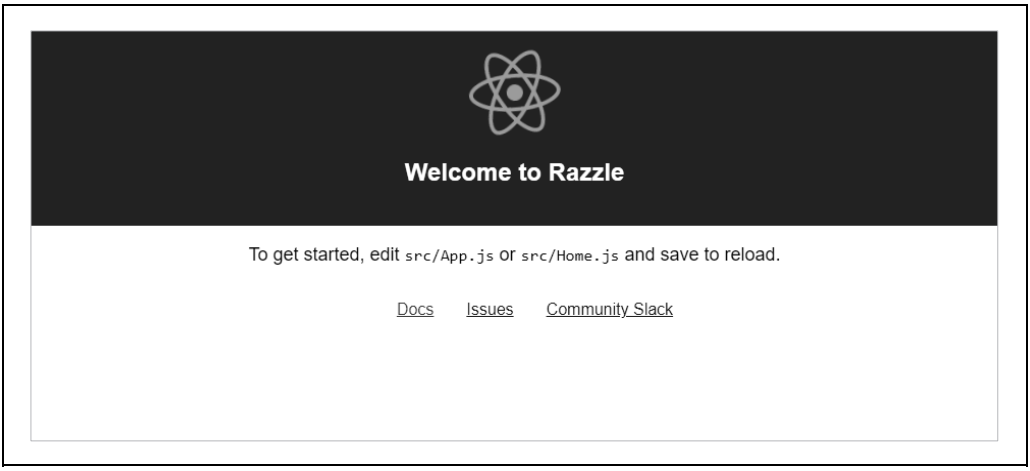
Aplikację Razzle można tworzyć przy użyciu polecenia `create-razzle-app`[3]:

```
$ npx create-razzle-app moja-aplikacja
```

Wykonanie tego polecenia spowoduje utworzenie nowego projektu aplikacji Razzle i zapisanie jej w katalogu *moja-aplikacja*. Aby rozpocząć prace nad nią, wystarczy wykonać skrypt `start`:

```
$ cd moja-aplikacja
$ npm run start
```

Skrypt `start` dynamicznie zbuduje zarówno kod wykonywany na serwerze, jak i ten działający po stronie klienta, a następnie uruchomi serwer na porcie 3000, jak pokazaliśmy na rysunku 1.5.



Rysunek 1.5. Strona początkowa aplikacji Razzle dostępna pod adresem http://localhost:3000

Kiedy będziesz chciał wygenerować produkcyjną wersję aplikacji, wystarczy skorzystać ze skryptu `build`:

```
$ npm run build
```

W odróżnieniu od narzędzia `create-react-app` polecenie `build` wygeneruje nie tylko kod kliencki, lecz także serwer Node. Razzle umieszcza generowane kody w katalogu *build*. Kod serwera w trakcie działania będzie generował statyczny kod kliencki. Serwer produkcyjny możesz uruchomić, wykonując plik *build/server.js* z wykorzystaniem do tego celu skryptu `start:prod`:

```
$ npm run start:prod
```

Serwer produkcyjny można wdrożyć na dowolnym systemie, w którym jest dostępne środowisko Node.

Zarówno serwer, jak i klient mogą wykonywać ten sam kod — to właśnie ta cecha sprawia, że aplikacje generowane przy użyciu narzędzia Razzle są określane jako *uniwersalne*. Ale dzięki czemu takie rozwiązanie jest możliwe?

[3] Podobieństwo do nazwy `create-react-app` jest jak najbardziej celowe. Twórca Razzle, Jared Palmer, wspomina o narzędziu `create-react-app` jako jednym ze źródeł inspiracji do utworzenia Razzle.

Klient oraz serwer mają różne punkty wejścia. Serwer uruchamia swój kod z użyciem pliku *src/server.js*, natomiast klient wykorzystuje plik *src/client.js*. Następnie oba te pliki, zarówno *server.js*, jak i *client.js*, renderują tę samą aplikację, używając przy tym pliku *src/App.js*.

Jeśli chcesz uruchomić aplikację jako aplikację jednostronicową, usuń plik *src/index.js* oraz *src/server.js*. Następnie w katalogu *public* utwórz plik *index.html* zawierający element `<div />` z identyfikatorem `root` i ponownie zbuduj aplikację, korzystając z następującego polecenia:

```
$ node_modules/.bin/razzle build --type=spa
```

Aby za każdym razem budować projekt jako aplikację jednostronicową, w pliku *package.json* dodaj do skryptów `start` i `build` parametr `--type=spa`.

To polecenie spowoduje zbudowanie kompletnej aplikacji jednostronicowej i zapisze ją w katalogu *build/public*; taką aplikację będziesz mógł wdrożyć na dowolnym serwerze WWW.

Analiza

Razzle jest tak elastyczny, gdyż składa się z zestawu wtyczek zapewniających ogromne możliwości konfiguracyjne. Każda wtyczka jest funkcją wyższego rzędu, do której jest przekazywana konfiguracja Webpacka i która zwraca jej zmodyfikowaną wersję. Jedna wtyczka może transpilować kod TypeScript, a inna scalać biblioteki Reacta.

Jeśli zechcesz zmienić framework stosowany w aplikacji na Vue, wystarczy zmienić używane przez nią wtyczki.

Kompletną listę dostępnych wtyczek narzędzia Razzle można znaleźć na stronie *https://github.com/jaredpalmer/razzle#plugins*.

Kod źródłowy tej receptury jest dostępny w przykładach dołączonych do książki, w katalogu *r01-03-razzle*.

1.4. Zarządzanie kodem klienta i serwera z wykorzystaniem Next.js

Problem

React generuje kod kliencki, choć robi to na serwerze. Czasami może się jednak zdarzyć, że będziemy dysponowali stosunkowo niewielkimi ilościami kodów interfejsów programowania aplikacji (API), którymi chcielibyśmy zarządzać z poziomu aplikacji Reacta.

Rozwiązanie

Next.js jest narzędziem do generowania aplikacji Reacta oraz kodu serwerowego. Punkty końcowe API oraz strony klienckie używają domyślnych konwencji routowania, dzięki czemu można je tworzyć

i wdrażać łatwiej niż w przypadku, gdybyśmy robili to ręcznie. Pełną dokumentację Next.js można znaleźć w jego witrynie — *https://nextjs.org*.

Aplikację Next.js możesz utworzyć, używając następującego polecenia:

```
$ npx create-next-app moja-aplikacja
```

Jeśli na wykorzystywanym komputerze jest zainstalowany menedżer pakietów `yarn`, to polecenie domyślnie go zastosuje. Możemy jednak wymusić zastosowanie menedżera `npm`, używając flagi `--use-npm`:

```
$ npx create-next-app --use-npm moja-aplikacja
```

Wykonanie polecenia `create-next-app` spowoduje utworzenie nowej aplikacji Next.js w podanym katalogu. Aby ją uruchomić, należy zastosować skrypt `dev` (patrz rysunek 1.6):

```
$ cd moja-aplikacja
$ npm run dev
```

Rysunek 1.6. Strona Next.js działająca na adresie http://localhost:3000

Next.js pozwala na tworzenie stron bez konieczności zarządzania informacjami o routowaniu. Jeśli tylko dodamy skrypt komponentu do katalogu *pages*, serwer od razu go udostępni. Na przykład za generowanie strony głównej domyślnej aplikacji odpowiada komponent *pages/index.js*.

To rozwiązanie jest nieco podobne do tego używanego w Gatsbym[4], jednak w Next.js zostało ono wzbogacone o dodawanie kodu serwerowego.

[4] Patrz receptura 1.2.

Aplikacje Next.js zazwyczaj zawierają jakiś kod serwerowy obsługujący API, co jest dość niezwykłe jak na aplikacje Reacta, które zazwyczaj są tworzone niezależnie od kodu serwerowego. Jeśli jednak zajrzymy do katalogu *pages/api*, to znajdziemy w nim przykładowy serwerowy punkt końcowy o nazwie *hello.js*:

```
// Next.js API route support: https://nextjs.org/docs/api-routes/introduction

export default function handler(req, res) {
  res.status(200).json({ name: 'John Doe' })
}
```

Operacja routowania, która kojarzy ten kod z punktem końcowym *api/hello*, jest wykonywana automatycznie.

Next.js transpiluje kod aplikacji do ukrytego katalogu o nazwie *.next*, który następnie może wdrożyć jako usługę na swojej własnej platformie o nazwie Vercel (*https://vercel.com*).

Aby wygenerować statyczną wersję aplikacji, użyj poniższego polecenia:

```
$ node_modules/.bin/next export
```

Polecenie `export` zbuduje kliencki kod aplikacji i zapisze go w katalogu *out*. Polecenie to skonwertuje każdą stronę do postaci statycznie wygenerowanego pliku HTML, który przeglądarki będą mogły szybko wczytywać i wyświetlać. Na końcu każdej strony będzie wczytywany jej odpowiednik zapisany w formie kodu JavaScript, odpowiadający za wczytywanie treści dynamicznych.

Jeśli utworzysz wyeksportowaną wersję aplikacji Next.js (używając polecenia `export`), nie będzie ona zawierać żadnych API działających po stronie serwera.

Next.js udostępnia kilka możliwości wczytywania danych, pozwalających na pobieranie ich ze źródeł statycznych czy też z „bezgłowych" systemów zarządzania treścią (CMS; patrz *https://nextjs.org/docs/basic-features/data-fetching*).

Analiza

Pod wieloma względami Next.js jest podobny do Gatsby'ego. Koncentruje się na szybkości dostarczania zawartości i nie wymaga rozbudowanej konfiguracji. Prawdopodobnie będzie najbardziej przydatny dla tych zespołów, które muszą używać stosunkowo niewiele kodu serwerowego.

Kody do tej receptury znajdziesz w przykładach dołączonych do książki, w katalogu *r01-04-nextjs*.

1.5. Tworzenie małych aplikacji przy użyciu Preacta

Problem

Aplikacje Reacta mogą być duże. Bez trudu można napisać prostą aplikację Reacta, która zostanie przekształcona na ogromny kod JavaScript o wielkości kilkuset kilobajtów. Być może będziesz chciał stworzyć aplikację, która pod względem cech i możliwości będzie przypominać aplikacje Reacta, lecz jednocześnie znacznie mniejszą.

Rozwiązanie

Jeśli chcesz dysponować możliwościami Reacta, a jednocześnie chciałbyś, by wygenerowany kod JavaScript aplikacji był mniejszy od tych, które normalnie tworzy React, to rozważ zastosowanie biblioteki Preact.

Preact to *nie* jest React. To zupełnie odrębna biblioteka zaprojektowana tak, by była możliwie jak najbardziej podobna do Reacta, a jednocześnie znacznie od niego mniejsza.

React jest tak duży ze względu na sposób, w jaki działa. Komponenty Reacta nie generują elementów bezpośrednio w obiektowym modelu dokumentu (ang. *Document Object Model*, w skrócie *DOM*) przeglądarki. Zamiast tego tworzą je w *wirtualnym DOM-ie*, a następnie, w krótkich odstępach czasu, aktualizują rzeczywisty DOM. Takie rozwiązanie sprawia, że proste operacje renderowania DOM-u mogą być bardzo szybkie, gdyż rzeczywisty DOM musi być aktualizowany wyłącznie w przypadkach, gdy faktycznie zajdą w nim jakieś zmiany. Jednak to rozwiązanie ma także wadę. Zagwarantowanie, że wirtualny DOM Reacta będzie aktualny, wymaga bardzo rozbudowanego kodu. Musi on bowiem obsługiwać kompletny, sztuczny model zdarzeń, dokładnie odpowiadający temu stosowanemu przez przeglądarkę. Z tego powodu framework React jest duży, a jego pobranie zajmuje trochę czasu.

Jednym z rozwiązań tego problemu jest stosowanie technik takich jak SSR[5], jednak korzystanie z SSR wymaga rozbudowanej konfiguracji[6]. Czasami chcemy pobierać niewielkie fragmenty kodu. I właśnie dlatego powstał Preact.

Biblioteka Preact, choć podobna do Reacta, jest bardzo mała. W czasie kiedy powstawała ta książka, wielkość głównej biblioteki Preact wynosiła około 4 KB. To na tyle mało, by pozwolić na dodawanie do stron WWW możliwości przypominających te, jakie daje React, z wykorzystaniem kodu niewiele większego od tego wymaganego do napisania rodzimego JavaScriptu.

Biblioteka Preact pozwala na wybór sposobu, w jaki będziemy jej używać: jako niewielkiej biblioteki JavaScriptu dodawanej do stron WWW (jest to tzw. podejście *beznarzędziowe*) bądź też jako kompletnej aplikacji JavaScriptu.

Ten pierwszy sposób jest bardzo prosty. Podstawowa biblioteka Preact nie obsługuje formatu JSX i nie zapewnia wsparcia dla narzędzia Babel, więc nie pozwala na stosowanie nowoczesnych możliwości JavaScriptu. Poniżej przedstawiona została przykładowa strona WWW korzystająca z biblioteki Preact w taki sposób:

```
<html>
  <head>
    <title>Aplikacja beznarzędziowa</title>
    <script src="https://unpkg.com/preact?umd"></script>
  </head>
  <body>
    <h1>Prosty sposób korzystania z biblioteki Preact!</h1>
    <div id="root"></div>
```

[5] *SSR* to skrót od angielskich słów *Server-Side Rendering*, oznaczających renderowanie po stronie serwera — *przyp. tłum.*

[6] Patrz receptury 1.2 i 1.3.

```
    <script>
      var h = window.preact.h;
      var render = window.preact.render;
      var mount = document.getElementById('root');
      render(
        h('button',
          {
            onClick: function() {
              render(h('div', null, 'Cześć!'), mount);
            }
          },
          'Kliknij!'),
        mount
      );
    </script>
  </body>
</html>
```

Ta aplikacja zostanie zamontowana w elemencie `<div/>` o identyfikatorze `root`, w którym wyświetli przycisk. Kliknięcie tego przycisku spowoduje zastąpienie zawartości elementu `div` o identyfikatorze `root` słowem `Cześć!`. To przykład tego, jak proste może być stosowanie biblioteki Preact.

Jednak rzadko kiedy będziemy pisać aplikacje w taki sposób. W praktycznych zastosowaniach utworzylibyśmy zapewne prosty łańcuch narzędzi, aby przynajmniej zapewnić sobie korzystanie z nowoczesnych możliwości języka JavaScript.

Preact obsługuje pełne spektrum aplikacji JavaScriptu. Po przeciwnej stronie skali złożoności jest tworzenie kompletnych aplikacji Preacta przy użyciu `preact-cli`.

`preact-cli` to narzędzie do tworzenia projektów korzystających z biblioteki Preact, stanowiące odpowiednik narzędzia `create-react-app`. Taką kompletną aplikację Preacta można utworzyć w następujący sposób:

```
$ npx preact-cli create default moja-aplikacja
```

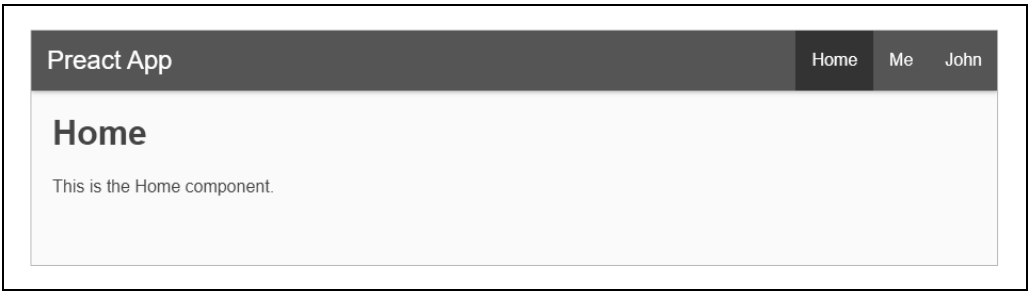
Powyższe polecenie korzysta ze standardowego szablonu aplikacji. Dostępne są także inne szablony służące do tworzenia projektów, na przykład korzystający z komponentów Material lub z języka TypeScript. Więcej informacji o nich można znaleźć na stronie biblioteki Preact w serwisie GitHub: *https://github.com/preactjs/preact-cli*.

To polecenie utworzy nową aplikację Preacta w katalogu *moja-aplikacja*. Aby ją uruchomić, wykonaj skrypt `dev`:

```
$ cd moja-aplikacja
$ npm run dev
```

Serwer zostanie uruchomiony na porcie 8080, a postać domyślnej strony aplikacji została przedstawiona na rysunku 1.7.

Serwer generuje stronę WWW, która z kolei odwołuje się do niego, by pobrać pakiet kodu JavaScript zapisany w pliku *src/index.js*.



Rysunek 1.7. Strona główna aplikacji Preacta

W ten sposób dysponujemy już kompletną aplikacją przypominającą aplikacje Reacta. Na przykład kod komponentu `Home`, zapisany w pliku *src/routes/home/index.js*, jest bardzo podobny do kodu normalnych komponentów Reacta i dysponuje pełnym wsparciem dla formatu JSX:

```
import { h } from 'preact';
import style from './style.css';

const Home = () => (
  <div class={style.home}>
         <h1>Home</h1>
         <p>This is the Home component.</p>
  </div>
);

export default Home;
```

W porównaniu ze standardowym komponentem Reacta jedyną znaczącą różnicą jest importowanie funkcji `h` z biblioteki `preact`, a nie `React` z biblioteki `react`.

Kod JSX umieszczony w komponentach Preacta będzie konwertowany na serię wywołań funkcji `h` i to stąd wynika konieczność jej zaimportowania. Z tego samego powodu aplikacje tworzone przy użyciu narzędzia `create-react-app` w wersji wcześniejszej od 17 wymagają zaimportowania obiektu `React`. Począwszy od wersji 17 w narzędziu `create-react-app` zaczęto stosować nowe przekształcenia JSX (*JSX Transform*; *https://reactjs.org/blog/2020/09/22/introducing-the-new-jsx-transform.html*), które eliminują konieczność każdorazowego importowania biblioteki `react`. Zawsze istnieje możliwość, że w przyszłych wersjach biblioteki Preact zostanie wprowadzona podobna zmiana.

Zwróćmy uwagę, że wielkość aplikacji wygenerowanej przy użyciu narzędzia `preact-cli` znacząco się zwiększyła: nieznacznie przekracza 300 KB. To całkiem dużo, choć pamiętajmy, że cały czas działamy w trybie do prowadzenia prac programistycznych nad aplikacją. Aby przekonać się o prawdziwej sile biblioteki Preact, zatrzymajmy serwer, naciskając kombinację klawiszy *Ctrl+C*, a następnie wykonajmy skrypt `build`:

```
$ npm run build
```

To polecenie wygeneruje statyczną wersję aplikacji i zapisze ją w katalogu *build*. Przede wszystkim jej zaletą będzie utworzenie statycznej kopii strony początkowej, którą przeglądarka będzie mogła

pobrać i wyświetlić bardzo szybko. Poza tym z tej wersji aplikacji zostanie usunięty cały nieużywany kod, a ten, który pozostanie, zostanie zminimalizowany. Jeśli udostępnisz tę wersję aplikacji na standardowym serwerze WWW, to po odwołaniu do niej przeglądarka będzie musiała pobrać jedynie między 50 a 60 KB kodu.

Analiza

Preact jest projektem godnym uwagi. Choć działa całkowicie inaczej niż React, biblioteka ta zapewnia niemal identyczne możliwości, choć jest wielokrotnie mniejsza. Fakt, że biblioteki Preact można używać do wszystkiego, zaczynając od najprostszego kodu, a na pełnych aplikacjach jednostronicowych kończąc, oznacza, że jeśli dla tworzonego projektu wielkość aplikacji ma kluczowe znaczenie, zdecydowanie warto rozważyć zastosowanie tej biblioteki.

Więcej informacji na temat biblioteki Preact możesz znaleźć na stronie *https://preactjs.com*.

Przykład prostego sposobu korzystania z biblioteki Preact znajdziesz w kodach źródłowych dołączonych do książki, w katalogu *r01-05-preact-beznarzedziowa*, z kolei większy przykład aplikacji korzystającej z biblioteki Preact jest dostępny w katalogu *r01-05-preact*.

Jeśli chcesz, by aplikacje tworzone przy użyciu biblioteki Preact w jeszcze większym stopniu przypominały aplikacje Reacta, zainteresuj się biblioteką *preact-compat* (*https://github.com/preactjs/preact-compat*).

I w końcu, być może zainteresuje Cię także inny projekt przypominający bibliotekę Preact — jest nim biblioteka InfernoJS (*https://infernojs.org*).

1.6. Tworzenie bibliotek z wykorzystaniem nwb

Problem

Duże firmy często w tym samym czasie pracują nad kilkoma aplikacjami Reacta. Jeśli pracujesz jako konsultant, to możesz tworzyć aplikacje dla wielu firm. Jeśli Twoja firma zajmuje się wytwarzaniem oprogramowania, to być może musicie budować różne aplikacja mające ten sam wygląd i sposób działania, więc zapewne będziecie chcieli tworzyć komponenty, które później będą używane w kilku różnych aplikacjach.

W przypadku prac nad projektem komponentów trzeba stworzyć strukturę katalogów, wybrać zestaw narzędzi, grupę używanych możliwości języka, przygotować proces budowania, który scali i zapisze te komponenty w formacie nadającym się do wdrażania. Cały ten proces może być równie trudny i męczący jak ręczne tworzenie projektu aplikacji Reacta.

Rozwiązanie

Pakiet narzędziowy `nwb` pozwala na tworzenie zarówno kompletnych aplikacji Reacta, jak i pojedynczych komponentów. Można go także używać do tworzenia komponentów, które później będą wykorzystywane w projektach Preacta lub InfernoJS; w tej recepturze skoncentrujemy się jednak na komponentach Reacta.

Aby utworzyć nowy projekt komponentów Reacta, w pierwszej kolejności musimy zainstalować pakiet `nwb` globalnie na komputerze:

```
$ npm install -g nwb
```

Kiedy to zrobimy, będziemy mogli utworzyć nowy projekt, używając do tego celu polecenia `nwb`:

```
$ nwb new react-component moj-komponent
```

Jeśli zamiast tworzyć jeden komponent, chcemy utworzyć całą aplikację `nwb`, to w powyższym poleceniu musimy zmienić `react-component` na `react-app`, `preact-app` lub `inferno-app`; w ten sposób utworzymy aplikację korzystającą z odpowiedniego frameworka. Możemy także zastosować opcję `vanilla-app`, aby utworzyć prosty projekt aplikacji JavaScriptu, która nie będzie używać żadnego frameworka.

Kiedy wykonamy to polecenie, będziemy musieli odpowiedzieć na kilka pytań dotyczących typu tworzonej biblioteki. Na przykład zostaniemy zapytani, czy chcemy utworzyć moduły ECMAScript:

```
Creating a react-component project...
? Do you want to create an ES modules build for use by compatible bundlers? (Y/n)
```

Ta opcja pozwala utworzyć kod zawierający instrukcje `export`, których Webpack może używać do określania, czy musi dodać komponent do aplikacji klienckiej. Zostaniemy także zapytani o to, czy chcemy używać UMD (Universal Module Definition):

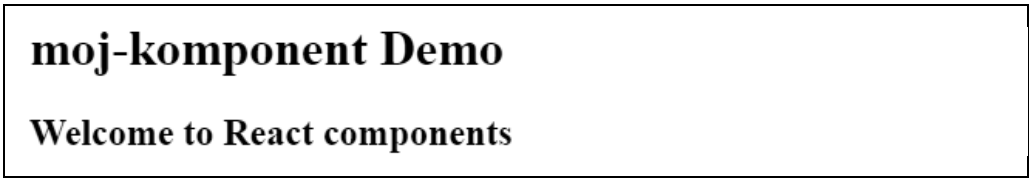
```
? Do you want to create a UMD build for global usage via <script> tag? (y/N)
```

Ta opcja jest przydatna, jeśli chcemy dołączać komponent do strony WWW przy użyciu znacznika `<script />`. W naszym przykładzie nie skorzystamy z tej możliwości.

Po podaniu odpowiedzi na te pytania narzędzie wygeneruje projekt komponentu Reacta i zapisze go w katalogu *moj-komponent*. W skład projektu będzie także wchodzić prosta aplikacja pozwalająca na sprawdzenie komponentu; możemy ją uruchomić przy użyciu skryptu `start`:

```
$ cd moj-komponent
$ npm run start
```

Aplikacja zostanie uruchomiona na porcie 3000, a jej postać została przedstawiona na rysunku 1.8.



Rysunek 1.8. Aplikacja prezentująca komponent nwb

Aplikacja będzie zawierać pojedynczy komponent zdefiniowany w pliku *src/index.js*:

```
import React, {Component} from 'react'

export default class extends Component {
  render() {
    return <div>
      <h2>Welcome to React components</h2>
```

```
      </div>
    }
  }
```

Teraz możemy już tworzyć komponent tak jak w każdym innym projekcie aplikacji Reacta. Kiedy już będziemy gotowi do tego, by przygotować wersję komponentu nadającą się do opublikowania, wystarczy wykonać polecenie:

```
$ npm run build
```

Zbudowany komponent zostanie umieszczony w pliku *lib/index.js*, który można umieścić w repozytorium, a następnie używać w innych projektach.

Analiza

Więcej informacji na temat tworzenia komponentów `nwb` można znaleźć w poradniku dotyczącym tworzenia komponentów i bibliotek, dostępnym na stronie *https://github.com/insin/nwb/blob/master/docs/guides/ReactComponents.md#developing-react-components-and-libraries-with-nwb*.

Kody źródłowe do tej receptury są dostępne w przykładach dołączonych do książki, w katalogu *r01-06-komponent-nwb*.

1.7. Dodawanie Reacta do aplikacji Rails za pomocą Webpackera

Problem

Framework Rails został utworzony, zanim interaktywne aplikacje budowane w języku JavaScript zyskały popularność. Aplikacje Rails są tworzone przy wykorzystaniu bardziej tradycyjnego modelu aplikacji internetowych, polegającego na generowaniu stron HTML na serwerze w odpowiedzi na żądania przesyłane przez przeglądarkę. Jednak czasami może się zdarzyć, że będziemy chcieli wzbogacić aplikację Rails o bardziej interaktywne możliwości.

Rozwiązanie

Można użyć biblioteki Webpacker, by wstawiać aplikacje Reacta do stron generowanych przez aplikację Rails. Aby przekonać się, jak działa to rozwiązanie, zacznijmy od wygenerowania aplikacji Rails, która będzie zawierać bibliotekę Webpacker:

```
$ rails new moja-aplikacja --webpack=react
```

To polecenie wygeneruje aplikację Rails i zapisze ją w katalogu *moja-aplikacja*. Aplikacja ta zostanie skonfigurowana do korzystania z serwera Webpacker. Zanim uruchomimy aplikację, należy przejść do jej katalogu i wygenerować przykładową stronę (kontroler):

```
$ cd moja-aplikacja
$ rails generate controller Example index
```

To polecenie wygeneruje poniższy szablon strony i zapisze go w pliku *app/views/example/index.html.erb*:

```
<h1>Example#index</h1>
<p>Find me in app/views/example/index.html.erb</p>
```

Kolejnym krokiem jest utworzenie prostej aplikacji Reacta, którą następnie wstawimy do wygenerowanej wcześniej strony. Rails wstawia aplikacje Webpackera jako tzw. *paczki* (ang. *packs*): niewielkie pakiety kodu JavaScript wchodzące w skład aplikacji Rails. W dalszej części tej receptury utworzymy nową paczkę, którą zapiszemy w pliku *app/javascript/packs/counter.js*; będzie ona zawierać prosty komponent licznika:

```
import React, { useState } from 'react'
import ReactDOM from 'react-dom'

const Counter = (props) => {
  const [count, setCount] = useState(0)
  return (
    <div className="Counter">
      Przycisk kliknięto {count} razy.
      <button onClick={() => setCount((c) => c + 1)}>Kliknij!</button>
    </div>
  )
}

document.addEventListener('DOMContentLoaded', () => {
  ReactDOM.render(
    <Counter />,
    document.body.appendChild(document.createElement('div'))
  )
})
```

Ta aplikacja aktualizuje licznik za każdym razem, kiedy użytkownik kliknie przycisk.

Teraz możemy już wstawić paczkę na stronę WWW; w tym celu wystarczy, że do szablonu strony dodamy jeden wiersz kodu:

```
<h1>Example#index</h1>
<p>Find me in app/views/example/index.html.erb</p>
<%= javascript_pack_tag 'counter' %>
```

Teraz możemy już uruchomić serwer Rails na porcie 3000:

```
$ rails server
```

W czasie kiedy przygotowywaliśmy tę książkę, uruchamianie serwera wymagało, by w systemie był zainstalowany menedżer pakietów `yarn`. Możemy to zrobić globalnie przy użyciu następującego polecenia: `npm install -g yarn`.

W przeglądarce zostanie wyświetlona strona *http://localhost:3000/example/index.html*, przedstawiona na rysunku 1.9.

Rysunek 1.9. Aplikacja Reacta osadzona na stronie http://localhost:3000/example/index.html

Analiza

Jak zapewne zgadłeś, za kulisami Webpacker przekształca aplikację z wykorzystaniem kopii narzędzia Webpack, którą można skonfigurować, używając pliku konfiguracyjnego *app/config/webpacker.yml*.

Webpacker jest używany raczej wraz z kodem aplikacji Rails, a nie stanowi jego zamiennika. Warto zastanowić się nad jego zastosowaniem, gdybyś chciał nieznacznie poprawić interaktywność tworzonej aplikacji Rails.

Więcej informacji na temat tego narzędzia można znaleźć na jego stronie w serwisie GitHub: *https://github.com/rails/webpacker*.

Kody źródłowe do tej receptury znajdują się w przykładach dołączonych do książki, w katalogu *r01-07-react-w-rails*.

1.8. Tworzenie niestandardowych elementów przy użyciu Preacta

Problem

Zdarzają się sytuacje, w których dodawanie kodu Reacta do istniejących treści stanowi duże wyzwanie. Na przykład w niektórych konfiguracjach systemów zarządzania treścią (CMS) użytkownicy nie mają możliwości dodawania do treści strony dodatkowego kodu JavaScript. W takich przypadkach bardzo przydatny może się okazać jakiś standaryzowany sposób bezpiecznego dodawania kodu JavaScript do stron.

Rozwiązanie

Własne elementy są standardowym sposobem tworzenia elementów HTML, które można dodawać do stron WWW. W efekcie poszerzają one standardowy język HTML, wzbogacając go o nowe znaczniki, z których mogą korzystać użytkownicy.

W tej recepturze pokażemy, jak można zastosować prosty framework, taki jak Preact, to tworzenia niestandardowych elementów, które następnie będzie można opublikować na serwerze WWW zarządzanym przez kogoś innego.

Zacznijmy od utworzenia aplikacji Preacta. Będzie ona udostępniać niestandardowy element, którego dzięki temu będziemy mogli użyć gdzieś indziej[7]:

```
$ preact create default moj-element
```

Następnie przejdźmy do katalogu utworzonego projektu i dodajmy do niego bibliotekę `preact-custom-element`:

```
$ cd moj-element
$ npm install preact-custom-element
```

Biblioteka `preact-custom-element` pozwoli zarejestrować nasz niestandardowy element HTML w przeglądarce.

Kolejnym krokiem jest modyfikacja pliku *src/index.js* w projekcie aplikacji Preacta w taki sposób, że zarejestruje on nowy, niestandardowy element zapisany w pliku *components/Converter/index.js*:

```
import register from 'preact-custom-element'
import Converter from './components/Converter'

register(Converter, 'x-converter', ['currency'])
```

Metoda `register` informuje przeglądarkę, że chcemy utworzyć nowy, niestandardowy element HTML o nazwie `<x-converter>`, który będzie miał jedną właściwość o nazwie `currency`, którą zdefiniujemy w pliku *src/components/Converter/index.js*:

```
import { h } from 'preact'
import { useEffect, useState } from 'preact/hooks'
import 'style/index.css'

const rates = { gbp: 0.81, eur: 0.92, jpy: 106.64 }

export default ({ currency = 'gbp' }) => {
  const [curr, setCurr] = useState(currency)
  const [amount, setAmount] = useState(0)

  useEffect(() => {
    setCurr(currency)
  }, [currency])

  return (
    <div className="Converter">
      <p>
        <label htmlFor="currency">Waluta: </label>
        <select
          name="currency"
          value={curr}
          onChange={(evt) => setCurr(evt.target.value)}
        >
          {Object.keys(rates).map((r) => (
            <option value={r}>{r.toUpperCase()}</option>
          ))}
        </select>
      </p>
      <p className="Converter-amount">
```

[7] Więcej informacji na temat tworzenia aplikacji przy użyciu frameworka Preact znajdziesz w recepturze 1.5.

```
        <label htmlFor="amount">Kwota: </label>
        <input
          name="amount"
          size={8}
          type="number"
          value={amount}
          onInput={(evt) => setAmount(parseFloat(evt.target.value))}
        />
      </p>
      <p>
        Wartość: 
        {((amount || 0) / rates[curr]).toLocaleString('pl-PL', {
          style: 'currency',
          currency: 'USD',
        })}
      </p>
    </div>
  )
}
```

W celu zachowania zgodności ze specyfikacją niestandardowych elementów jego nazwa musi zaczynać się od małej litery, nie może zawierać żadnych wielkich liter i musi zawierać znak minusa[8]. Ta konwencja gwarantuje, że nazwa elementu nie będzie kolidować z żadnymi nazwami standardowych elementów języka HTML.

Nasz komponent `Converter` służy do przeliczania walut, które w tym przykładzie mają ustalone, stałe kursy wymiany. Jeśli teraz uruchomisz serwer Preacta:

```
$ npm run dev
```

to kod JavaScript niestandardowego elementu będzie dostępny pod adresem *http://localhost:8080/bundle.js*.

Aby użyć tego nowego elementu, wystarczy gdzieś utworzyć statyczną stronę WWW zawierającą następujący kod HTML:

```
<html>
    <head>
        <script src="https://unpkg.com/babel-polyfill/dist/polyfill.min.js"></script>
        <script src="https://unpkg.com/@webcomponents/webcomponentsjs">
        </script>
        <!-- W tym elemencie podaj prawidłowy adres swojego niestandardowego elementu. -->
        <script type="text/javascript" src="http://localhost:8080/bundle.js"></script>
    </head>
    <body>
        <h1>Niestandardowy element HTML</h1>
        <div style="float: right; clear: both">
            <!-- Ten znacznik spowoduje wstawienie na stronę aplikacji Preacta -->
            <x-converter currency="jpy"/>
        </div>
        <p>Ta strona używa przykładowego, niestandardowego elementu
            HTML o nazwie <code>&lt;x-converter/&gt;</code>,
```

[8] Więcej informacji dotyczących niestandardowych elementów oraz stosowanych przy tym konwencji nazewniczych można znaleźć w specyfikacji WHATWG (dostępnej na stronie *https://html.spec.whatwg.org/multipage/custom-elements.html*).

```
            który jest udostępniany z innego serwera WWW.</p>
    </body>
</html>
```

Ta strona zawiera definicję niestandardowego elementu, która jest dodawana przez ostatni znacznik `<script/>` umieszczony w elemencie `<head/>`. Aby zapewnić, że nasz element będzie działał zarówno w nowych, jak i w starych przeglądarkach, dodaliśmy także kilka skryptów z witryny *unpkg.com*.

Skoro już umieściliśmy kod niestandardowego elementu w kodzie strony WWW, możemy umieścić w niej także znaczniki `<x-converter/>`, zupełnie tak samo, jak gdyby należały do standardu HTML. W naszym przykładzie do komponentu Preacta obsługującego niestandardowy element HTML przekazujemy także właściwość `currency`.

Nazwy właściwości niestandardowych elementów HTML przekazywane do komponentów, które je obsługują, są zapisywane małymi literami, niezależnie od tego, w jaki sposób zostaną zdefiniowane w kodzie HTML strony.

Tę stronę WWW możesz udostępnić na innym serwerze WWW niezależnym od serwera Preacta. Postać przykładowej strony po jej wyświetleniu w przeglądarce przedstawiliśmy na rysunku 1.10.

Rysunek 1.10. Niestandardowy element osadzony w statycznej stronie WWW

Analiza

Niestandardowy element nie musi być udostępniany przez ten sam serwer, na którym będą dostępne korzystające z niego strony WWW. Oznacza to, że możemy używać takich niestandardowych elementów w celu publikowania widżetów, które będą mogły być używane na dowolnych stronach WWW. Z tego względu warto zastanowić się nad sprawdzaniem nagłówka `Referer` dostępnego w odbieranych żądaniach do komponentu, by zabezpieczyć się przed niepożądanym użyciem elementu.

W przedstawionym przykładzie niestandardowy element jest udostępniany przez serwer Preacta służący do prowadzenia prac programistycznych. Do zastosowań produkcyjnych należałoby raczej wygenerować statyczną wersję komponentu, która będzie znacząco mniejsza[9].

[9] Więcej informacji na temat zmniejszania aplikacji Preacta znajdziesz w recepturze 1.5.

Kody źródłowe do tej receptury znajdziesz w przykładach dołączonych do książki, w katalogu *r01-08-niestandardowe-elementy*.

1.9. Tworzenie komponentów z zastosowaniem Storybooka

Problem

Komponenty są stabilnymi elementami konstrukcyjnymi używanymi do tworzenia aplikacji Reacta. Jeśli będziemy je tworzyć rozważnie i ostrożnie, będziemy mogli używać ich także w innych aplikacjach. Jednak podczas tworzenia komponentu sprawdzenie, jak będzie on działał we wszelkich sytuacjach, zajmuje sporo czasu. Na przykład w asynchronicznych aplikacjach może się zdarzyć, że kiedy React zacznie renderować komponent, część jego właściwości będzie mieć niezdefiniowane wartości. Czy w takiej sytuacji komponent wciąż będzie wyświetlany poprawnie? A może wystąpią jakieś błędy?

Jeśli będziemy tworzyć komponenty w ramach prac nad złożoną aplikacją, to odtworzenie wszystkich sytuacji, z którymi komponent będzie musiał sobie radzić, może być bardzo trudne.

Co więcej, jeśli w zespole będą pracować doświadczeni programiści zajmujący się zagadnieniami doświadczeń użytkownika (UX), to przechodzenie przez całą aplikację, by dotrzeć do jednego, tworzonego komponentu może być dla nich dużą stratą czasu.

Byłoby bardzo wygodne, gdyby można było uruchamiać komponent w izolowanym środowisku i jedynie przekazywać do niego przykładowe zestawy właściwości.

Rozwiązanie

Storybook jest narzędziem do wyświetlania bibliotek komponentów w różnych stanach. Można by je opisać jako galerię do prezentowania komponentów, ale to zapewne byłby niepełny opis. W rzeczywistości Storybook jest bowiem narzędziem do tworzenia komponentów.

W jaki sposób można dodać Storybooka do projektu? Zacznijmy od utworzenia aplikacji Reacta przy użyciu `create-react-app`:

```
$ npx create-react-app moja-aplikacja
$ cd moja-aplikacja
```

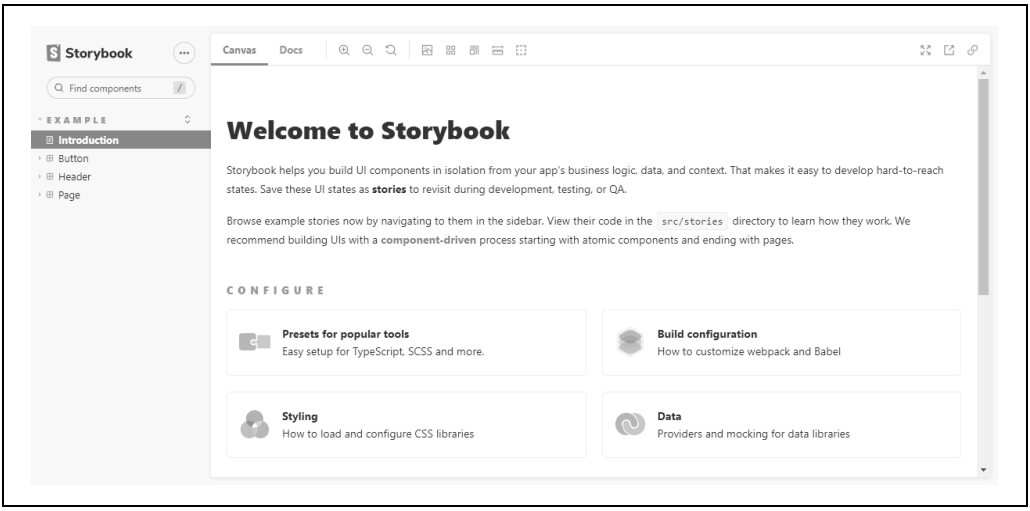
W kolejnym kroku dodamy Storybooka do tego projektu:

```
$ npx sb init
```

Teraz możemy już uruchomić serwer Storybooka, używając `yarn` lub `npm`:

```
$ npm run storybook
```

Storybook uruchamia odrębny serwer działający na porcie 9000; jego stronę początkową przedstawiliśmy na rysunku 1.11. W przypadku korzystania ze Storybooka nie ma potrzeby uruchamiania aplikacji Reacta.



Rysunek 1.11. Strona początkowa Storybooka

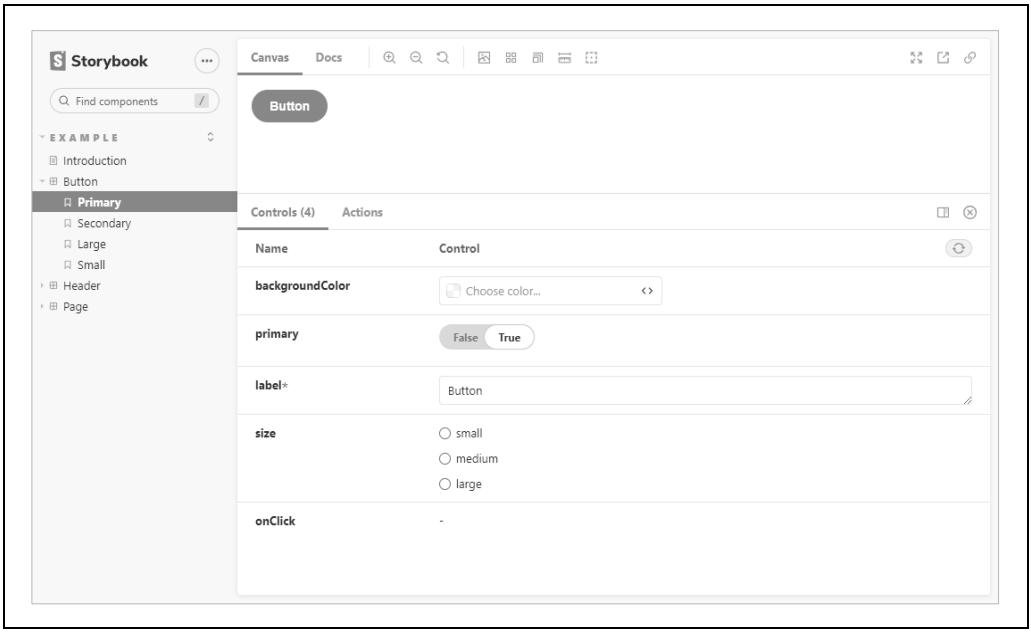
Storybook określa pojedynczy komponent renderowany z wykorzystaniem przykładowych właściwości jako *opowieść* (ang. *story*). Domyślnie Storybook instaluje kilka takich przykładowych opowieści, które są umieszczone w katalogu *src/stories* aplikacji. Przykład zamieszczony poniżej pochodzi z pliku *src/stories/Button.stories.js*:

```
import React from 'react';

import { Button } from './Button';

stories/introduction#default-export
export default {
  title: 'Example/Button',
  component: Button,
  argTypes: {
    backgroundColor: { control: 'color' },
  },
};

const Template = (args) => <Button {...args} />;

export const Primary = Template.bind({});
Primary.args = {
  primary: true,
  label: 'Button',
};

export const Secondary = Template.bind({});
Secondary.args = {
  label: 'Button',
};

export const Large = Template.bind({});
Large.args = {
  size: 'large',
  label: 'Button',
```

```
};

export const Small = Template.bind({});
Small.args = {
  size: 'small',
  label: 'Button',
};
```

Storybook poszukuje plików o nazwach pasujących do wzorca **.stories.js*, umieszczonych w katalogu z kodami źródłowymi aplikacji; nie zwraca jednak uwagi na to, gdzie w nim będą umieszczone, więc możemy je zapisywać w dowolnym miejscu. Jedno z często stosowanych rozwiązań polega na umieszczaniu plików *stories* w tym samym katalogu, gdzie są przechowywane komponenty, których te pliki dotyczą. W ten sposób, jeśli skopiujemy katalog do innej aplikacji, pliki historii zostaną skopiowane razem z nimi i będą stanowić rodzaj żywej dokumentacji komponentu.

Na rysunku 1.12 pokazaliśmy, jak wygląda opowieść *Button.stories.js* wyświetlona w Storybooku.



Rysunek 1.12. Przykładowa opowieść

Analiza

Niezależnie od swojego prostego wyglądu Storybook jest narzędziem programistycznym znacząco poprawiającym efektywność pracy. Pozwala koncentrować się na pojedynczych komponentach i pracować nad nimi. W sposób nieco zbliżony do wizualnych testów jednostkowych Storybook pozwala sprawdzać działanie komponentu w szeregu możliwych sytuacji, aby określić, czy funkcjonuje on prawidłowo.

Storybook dysponuje także wieloma dodatkami (patrz *https://storybook.js.org/addons*).

Dostępne dodatki pozwalają:

- sprawdzać występowanie problemów związanych z dostępnością (*addon-a11y*);
- dodawać interaktywne kontrolki służące do modyfikowania wartości właściwości (*Knobs*);
- dodawać dokumentację do opowieści (*Docs*);
- zapisywać migawki kodu HTML, by testować wpływ wprowadzanych zmian (*Storyshots*)

oraz na wiele więcej.

Dodatkowe informacje na temat Storybooka można znaleźć w jego witrynie — *https://storybook.js.org*.

Kody źródłowe do tej receptury są dostępne w przykładach dołączonych do książki, w katalogu *r01-09-stosowanie-storybooka*.

1.10. Testowanie kodu w przeglądarce z zastosowaniem Cypressa

Problem

Większość projektów Reacta zawiera biblioteki do testowania. Najczęściej używaną z nich jest prawdopodobnie `@testing-library/react`, która jest dołączana do aplikacji tworzonych przy użyciu `create-react-app` lub Enzyme, która z kolei jest używana we frameworku Preact.

Jednak niemal nic nie jest się w stanie równać z testowaniem kodu w rzeczywistych przeglądarkach, wraz z wszelkimi komplikacjami, jakie się z tym wiążą. Tradycyjnie już testowanie w przeglądarkach jest niestabilne i wymaga częstej pielęgnacji, gdyż każda aktualizacja przeglądarki pociąga za sobą konieczność zaktualizowania jej sterowników (takich jak Chrome Driver).

Jeśli dodamy do tego problemy związane z generowaniem danych testowych na serwerze, okaże się, że testowanie oprogramowania w przeglądarkach może być złożone, i to zarówno pod względem konfiguracji, jak i zarządzania.

Rozwiązanie

Framework testowy Cypress (*https://www.cypress.io*) pozwala uniknąć wielu wad, które tradycyjnie wiążą się z testowaniem w przeglądarkach. Cypress działa w przeglądarce, jednak nie zmusza do stosowania zewnętrznych narzędzi służących do sterowania jej pracą. Zamiast tego Cypress komunikuje się bezpośrednio z przeglądarką na określonym porcie, podobnie jak Chrome lub Electron, a następnie wstrzykuje kod JavaScript służący do wykonywania większości kodu testowego.

Aby przekonać się, jak używać frameworka Cypress do testowania kodu, zacznijmy od wygenerowania aplikacji przy użyciu `create-react-app`:

```
$ npx create-react-app --use-npm moja-aplikacja
```

Następnie przejdźmy do katalogu aplikacji i zainstalujmy framework Cypress:

```
$ cd moja-aplikacja
$ npm install cypress --save-dev
```

Zanim go uruchomimy, musimy go skonfigurować, by „wiedział", gdzie jest aplikacja. W tym celu w katalogu aplikacji utwórzmy plik *cypress.json*, a w nim podajmy adres URL aplikacji:

```
{
  "baseUrl": "http://localhost:3000/"
}
```

Kolejnym krokiem będzie uruchomienie aplikacji:

```
$ npm start
```

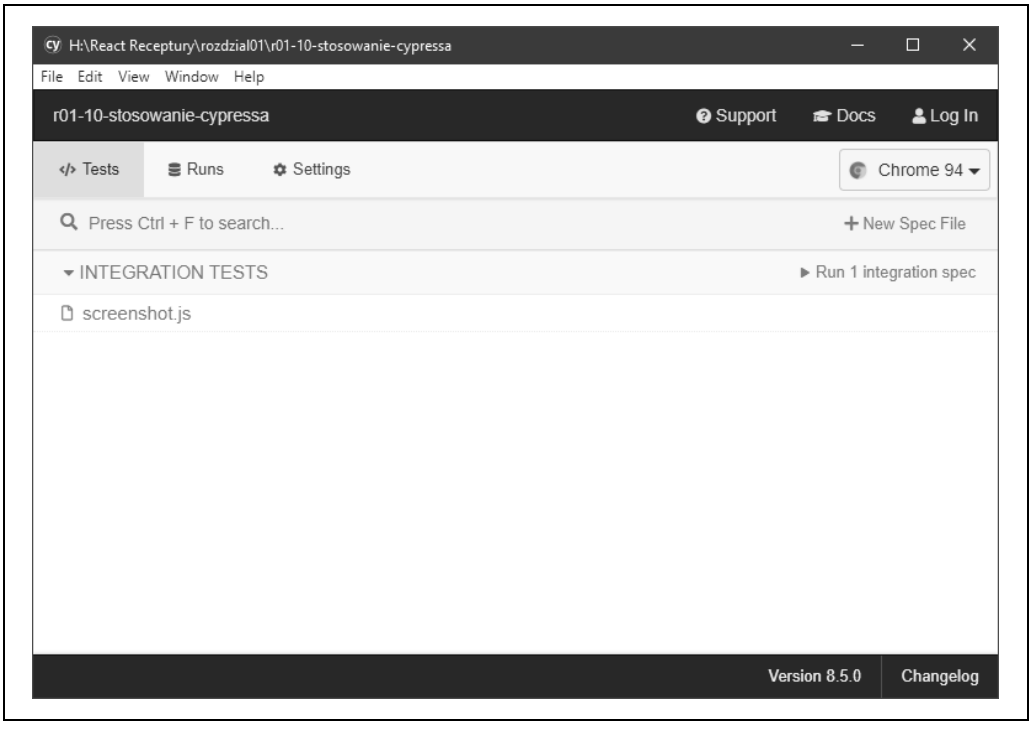
a następnym uruchomienie frameworka Cypress:

```
$ npx cypress open
```

W momencie pierwszego uruchomienia Cypress zainstaluje wszystkie niezbędne zależności. Teraz utworzymy test, który otwiera stronę domową aplikacji i robi jej zrzut ekranu; przedstawiony poniżej kod testu zapiszmy w pliku *screenshot.js* w katalogu *cypress/integrations*:

```
describe('screenshot', () => {
  it('można zrobić zrzut ekranu', () => {
    cy.visit('/');
    cy.screenshot('frontpage');
  });
});
```

Warto zauważyć, że ten test jest zapisany w formacie Jest. Kiedy go zapiszemy, plik *screenshots.js* zostanie wyświetlony w głównym oknie Cypressa, jak pokazaliśmy na rysunku 1.13.



Rysunek 1.13. Okno Cypressa

Kiedy dwukrotnie klikniemy test, Cypress uruchomi go w oknie przeglądarki. Uruchomienie testu spowoduje wyświetlenie głównej strony aplikacji, wykonanie jej zrzutu i zapisanie go w pliku *cypress/screenshots/screenshot.js/frontpage.png*.

Analiza

Poniżej przedstawiliśmy kilka przykładowych poleceń, które można wykonywać przy użyciu Cypressa:

Polecenie	Opis
`cy.contains('Franek')`	Znajduje element zawierający łańcuch *Franek*.
`cy.get('.Norman').click()`	Klika element należący do klasy *.Norman*.
`cy.get('input').type('Cześć!');`	Wpisuje łańcuch `'Cześć!'` w polu tekstowym.
`cy.get('h1').scrollIntoView()`	Przewija stronę tak, by był widoczny element `h1`.

To jedynie kilka przykładów poleceń pozwalających na prowadzenie interakcji ze stronami WWW. Jednak Cypress ma także w zanadrzu jeszcze inne sztuczki. Umożliwia modyfikowanie kodu w przeglądarce, by zmieniać w niej czas (`cy.clock()`), pozwala zarządzać ciasteczkami (`cy.cookie()`), czyścić magazyn lokalny (`cy.clearLocalStorage()`), a przede wszystkim imitować żądania przesyłane do API na zdalnym serwerze i odbierane odpowiedzi.

Cypress zapewnia te możliwości, modyfikując wbudowane w przeglądarkę funkcje sieciowe, dzięki czemu wywołanie:

```
cy.route("/api/server?*", [{coštam: 'Dane'}])
```

będzie przechwytywać wszystkie żądania kierowane do punktu końcowego */api/server?* i jako ich wynik zwracać tablicę zapisaną w formacie JSON: `[{coštam: 'Dane'}]`.

Symulowanie odpowiedzi na żądania sieciowe może całkowicie zmienić sposób, w jaki zespoły będą pracować nad aplikacjami, gdyż całkowicie oddziela prace nad aplikacją kliencką od pracy nad jej częścią serwerową. Testy wykonywane w przeglądarce mogą określać wszelkie niezbędne dane i to bez konieczności tworzenia faktycznego serwera oraz bazy danych.

Aby dowiedzieć się więcej na temat frameworka Cypress, zajrzyj do jego dokumentacji dostępnej na stronie *https://docs.cypress.io/api/table-of-contents*.

Kody źródłowe do tej receptury znajdują się w przykładach dołączonych do książki, w katalogu *r01-10-stosowanie-cypressa*.

React: zrozum, a napiszesz świetną aplikację w krótkim czasie!

JavaScript cieszy się opinią wszechstronnego i elastycznego języka programowania. Przy czym bywa dość wymagający dla programisty. Rozwiązaniem dla osób, którym zależy na pisaniu niezawodnych aplikacji w krótkim czasie, okazują się frameworki. Szczególnym uznaniem cieszy się React, gdyż pozwala na pełne wykorzystanie możliwości nowoczesnych przeglądarek i urządzeń mobilnych. Jest to jednak narzędzie, które na początku przysparza problemów ze zrozumieniem sposobu działania, a bez tego trudno o otrzymanie bezbłędnie pracującej aplikacji.

Dzięki tej książce błyskawicznie uzyskasz odpowiedzi na nurtujące programistów pytania o walidację danych w formularzach, testowanie kodu czy powiązanie kodu aplikacji z kodem serwerowym. Dowiesz się również, jak zapewnić sobie możliwość wielokrotnego wykorzystywania kodu i implementacji złożonych operacji w prosty sposób. Znajdziesz tu szereg przykładowych kodów, pogrupowanych tematycznie i dobranych tak, aby ułatwić Ci rozwiązywanie problemów najczęściej pojawiających się podczas pisania aplikacji Reacta. Poszczególne próbki kodu zostały gruntownie objaśnione, dzięki czemu szybko zrozumiesz, w jaki sposób współdziałają komponenty aplikacji Reacta i jego biblioteki. A wtedy w pełni docenisz zalety tego frameworku!

W książce między innymi:

- pisanie aplikacji jednostronicowych i progresywnych
- integracja aplikacji z usługami serwerowymi, takimi jak REST lub GraphQL
- automatyczne wykrywanie problemów z dostępnością
- zabezpieczanie i testowanie aplikacji
- unikanie powszechnych problemów funkcjonalnych i związanych z wydajnością

David Griffiths jest autorem książek, programistą i instruktorem. Jako piętnastolatek napisał implementację języka Logo. Od pięciu lat tworzy za pomocą Reacta aplikacje dla firm z różnych branż.

Dawn Griffiths jest znakomitą i doświadczoną programistką, a także autorką książek. Wraz z mężem Davidem opracowała animowany kurs wideo *The Agile Sketchpad*, uczący kluczowych pojęć i technik w sposób zapewniający aktywną pracę mózgu.

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

ISBN 978-83-283-8763-8