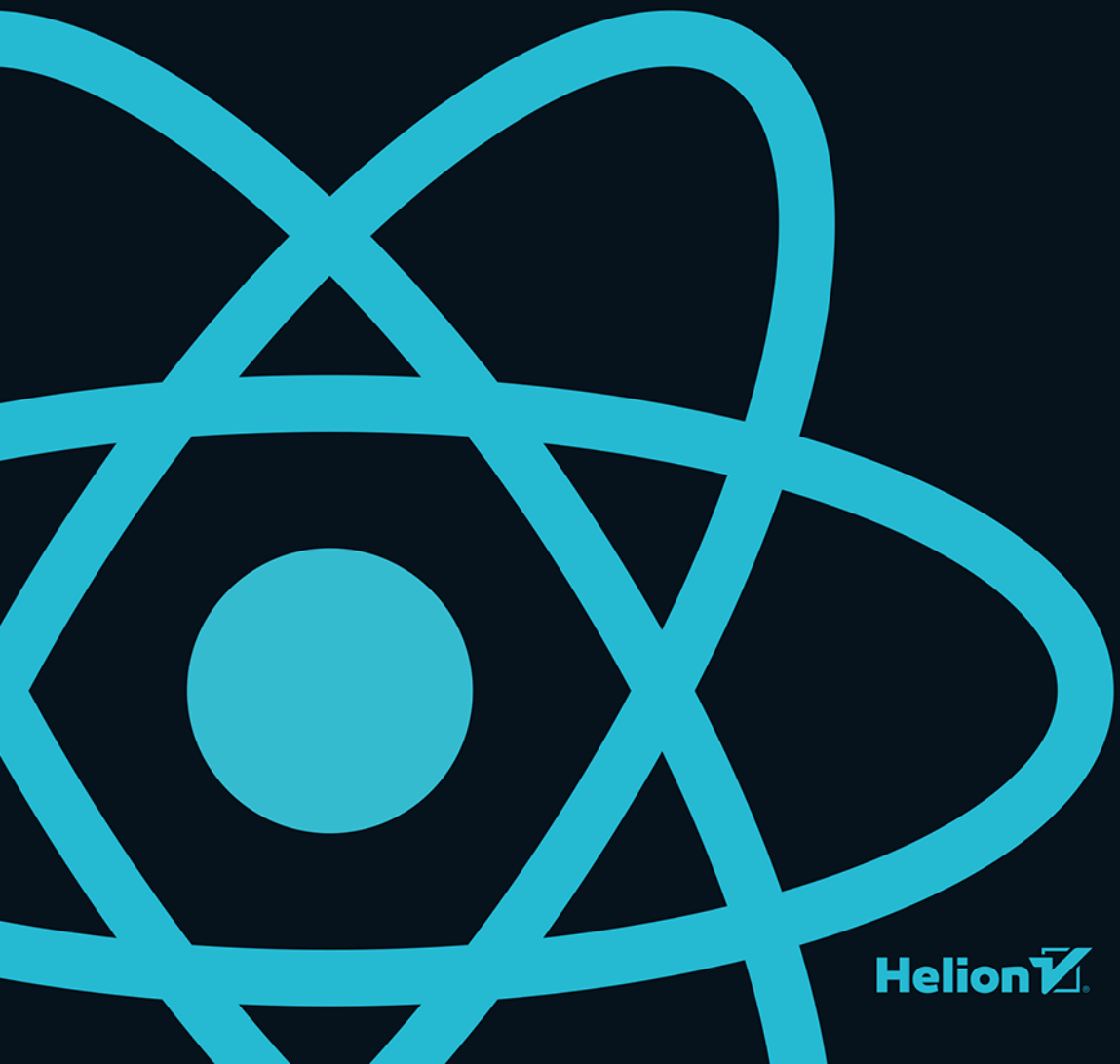
Paweł Kamiński

React

Wstęp do programowania



Helion

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki
Grafika na okładce została wykorzystana za zgodą Shutterstock.com

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

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

ISBN: 978-83-283-6850-7

Printed in Poland.

Spis treści

Rozdział 6.

Routing

Kolejnym z elementów, którym zajmiemy się podczas poznawania biblioteki React, jest możliwość tworzenia i wywoływania podstron serwisu. W typowej aplikacji klient–serwer, gdzie każde żądanie oznacza przeładowanie całego kodu przez serwer aplikacyjny, tworzenie podstron jest czymś naturalnym. Jak więc przełożyć to na język *Single Page Applications*? Z pewnością przejście pomiędzy podstronami serwisu musi charakteryzować się kilkoma warunkami.

- Nie może istnieć proces przeładowania witryny, całość ma się odbywać w sposób jak najbardziej intuicyjny.
- Przejście pomiędzy podstronami powinno być sygnalizowane aktualizacją paska adresu; co więcej, późniejsze wejście na stronę właśnie z tego adresu powinno zaowocować wyświetleniem podstrony zgodnej z adresem.
- W pasku adresu powinno być możliwe przesyłanie parametrów, które następnie mogłyby być odczytywane w kodzie.
- Przejście pomiędzy podstronami powinno odbywać się na zasadzie wyboru odpowiedniego elementu interfejsu (odnośnika) bądź też poprzez ręczne wpisanie adresu (czyli zgodnie z punktem drugim niniejszej listy).
- Niektóre z podstron powinny być zabezpieczone przed dostępem dla osób niezalogowanych.

Wszystkie powyższe cele łatwo zrealizujemy za pomocą biblioteki React-router. Jest to swoisty zbiór komponentów, które znacznie usprawniają prace związane z routingiem. Przejdźmy do konkretów, prace wykonywać będziemy na kodzie napisanym w poprzednim rozdziale, oznacza to, iż rozwiniemy aplikację logowania o nowe elementy. Dzięki temu będziemy mogli przećwiczyć również zabezpieczenia podstron przed nieupoważnionym dostępem. Zmiany rozpoczynamy od instalacji samej biblioteki, w terminalu wywołujemy więc polecenie:

```
npm install react-router-dom
```

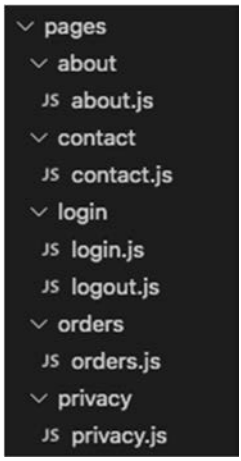
Jako że przy budowanym projekcie chcielibyśmy również dodać trochę własnego kodu kaskadowych arkuszy stylów, warto od razu doinstalować bibliotekę Node-sass, która w sposób automatyczny skompiluje pliki źródłowe SASS (ang. *Syntactically Awesome Style Sheets)*,

czyli pliki preprocesora języka do plików CSS. Instalacja jest tym razem również dość standardowa, gdyż ogranicza się do wywołania polecenia:

```
npm install node-sass
```

Dobrze, mamy już przygotowane niezbędne biblioteki, zapoznajmy się więc z podstawową strukturą aplikacji, która składać się będzie z kilku komponentów — każdy z nich będzie odpowiadał za jedną podstronę serwisu. Wszystkie będą umiejscowione w katalogu *src/pages*. Na rysunku 6.1 przedstawiono schemat plików umieszczonych w opisywanym miejscu.

RYSUNEK 6.1. *Struktura katalogów poszczególnych podstron*



Jak zostało już podkreślone, każdy z opisywanych wyżej plików to pojedynczy komponent. Intencjonalnie i trochę na wyrost są to komponenty klasowe, które — mimo że teraz nie przechowują żadnego stanu ani nie mają wyszukanej logiki biznesowej — jednak są przykładem zaślepek kodu, który w przyszłości może być rozbudowany i niezależny. Na listingu 6.1 przedstawiono przykład takiego komponentu w postaci kodu źródłowego pliku *src/pages/contact/contact.js*.

LISTING 6.1. *Kod źródłowy komponentu Contact*

```
import React, { Component } from "react";
import { Row, Col, Container } from "react-bootstrap";
class Contact extends React {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <Container>
        <Row>
          <Col>Kontakt</Col>{" "}
        </Row>
      </Container>
```

```
    );
  }
}
export { Contact as ContactPage };
```

Oczywiście nic nie stoi na przeszkodzie, by elementem docelowym reprezentującym wybraną podstronę był komponent funkcyjny. Taki przypadek został zaprezentowany na listingu 6.2.

LISTING 6.2. *Komponent funkcyjny reprezentujący podstronę serwisu*

```
import React from "react";
import { Row, Col, Container } from "react-bootstrap";
const AboutPage = () => (
  <Container>
    <Row>
      <Col>Strona o nas. </Col>
    </Row>
  </Container>
);
export { AboutPage };
```

Widzimy więc, iż komponent, który ma reprezentować podstronę, może być utworzony jako dowolny kod i realizować dowolną logikę oraz odwoływać się do dowolnych elementów, które zostały poznane dotychczas. Jak jednak sprawa wygląda w przypadku generowania samych odnośników?

Zacznijmy od modyfikacji samego źródła nawigacji — pliku *src/partials/navigation/navigation.js*, którego zawartość została zaprezentowana na listingu 6.3.

LISTING 6.3. *Plik źródłowy panelu nawigacji po zmianach*

```
import React, { Component } from "react";
import { connect } from "react-redux";
import { NavigationComponent } from "../../components/navigation/navigation";

const navigation = (props) => {
  let userInfo = "";
  const { isLogged } = props;
  const user = sessionStorage.getItem("account");
  if (user && isLogged) {
    userInfo = JSON.parse(JSON.parse(user).config.data).username;
  }
  const menus = {
      menuItems: [
        {
           title: "Zamówienia",
           link: "/orders"
        },
        {
           title: "Kontakt",
           link: "/contact"
        },
```

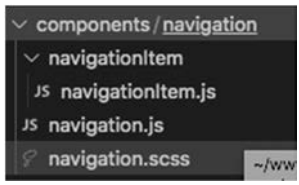
```
        {
           title: "Polityka bezpieczeństwa",
           link: "/privacy"
        },
        {
           title: "O nas",
           link: "/about"
        },
        {
           title: "Logowanie",
           link: "/login"
        }
      ]
    };

  return  (
    <div>
      <NavigationComponent items={menus.menuItems} />
      {isLogged && <h6>Witaj, {userInfo}</h6>}
    </div>
       );
    }
(...)
/** pozostała zawartość pliku bez zmian */
```

Oprócz znajomej już części generującej informacje o statusie zalogowanej osoby, w opisywanym powyżej skrypcie umieszczona została również stała przechowująca listę dostępnych w systemie odnośników. Każdy element listy to nazwa odnośnika wraz z odpowiadającym adresem. Cała lista jest natomiast przesyłana wprost do komponentu `NavigationComponent`, który — jak łatwo się domyślić — pełni funkcję wyświetlającą listę odnośników.

W tym miejscu warto przypatrzeć się opracowanej od zera prostej strukturze do zbudowania nawigacji, która składa się z następujących plików:

- *src/components/navigation/navigation.js* — komponent reprezentujący listę odnośników,
- *src/components/navigation/navigation.scss* — arkusz stylów opisujący pojedynczy element listy nawigacyjnej,
- *src/components/navigation/navigationItem/navigationItem.js* — pojedynczy odnośnik, element listy nawigacyjnej.

Na rysunku 6.2 przedstawiono zaprezentowaną strukturę w postaci drzewa katalogów.



RYSUNEK 6.2. *Struktura katalogów dla nawigacji*

Skoro zapoznaliśmy się z ogólnym położeniem plików, czas zajrzeć do ich środka. Zacznijmy od *src/components/navigation/navigation.js*, którego zawartość przedstawia listing 6.4.

LISTING 6.4. *Kod źródłowy komponentu listy nawigacyjnej*

```
import React from "react";
import NavigationItem from "./navigationItem/navigationItem";
import PropTypes from "prop-types";
const Navigation = props => {
  let items = [];
  if (props.items !== undefined) {
    items = props.items.map((item, key) => (
      <NavigationItem key={key} item={item} />
    ));
  }
  return (
    <nav className="navigation">
      <ul className="navigation__elements">{items}</ul>
    </nav>
  );
};
Navigation.propTypes = {
  items: PropTypes.array
};
export { Navigation as NavigationComponent };
```

Pierwsze linie kodu to klasyczne importy, które dotyczą pojedynczego elementu nawigacji (pojedynczego odnośnika) oraz znane już `PropTypes`. Kod samego komponentu to przede wszystkim iteracja po liście elementów nawigacji przekazanej w `props`, przy użyciu funkcji `map`. Dla każdego z elementów listy generowany jest komponent *NavigationItem*, do którego przekazywane są dane pojedynczego odnośnika. Całość umieszczona zostaje w znacznikach `nav` i `ul`.

W tym miejscu nasze kroki od razu kierujemy do kolejnego pliku — tym razem reprezentującego pojedynczy odnośnik, jest to oczywiście *src/components/navigation/navigationItem/navigationItem.js*, którego zawartość prezentuje listing 6.5.

LISTING 6.5. *Skrypt reprezentujący pojedynczy element nawigacji*

```
import React from "react";
import { Link } from "react-router-dom";
import PropTypes from "prop-types";

const NavigationItem = props => {
  return (
    <li className="navigation__element">
      <Link to={props.item.link} className="navigation__link">
        {props.item.title}
      </Link>
    </li>
  );
```

```
};
NavigationItem.propTypes = {
  item: PropTypes.object.isRequired
};
export default NavigationItem;
```

Co najważniejsze, w przypadku tworzenia odnośnika, czyli pojedynczego elementu nawigacji, z biblioteki react-router-dom importujemy komponent `Link`, za którego pomocą budujemy odnośnik. Konstrukcja:

```
<Link to={props.item.link} className="navigation__link">
  {props.item.title}
</Link>
```

oznacza, iż:

- odnośnik ma prowadzić do adresu podanego w `props.item.link`,
- tytuł odnośnika to `props.item.title`,
- klasa odnośnika to `navigation_class`.

Właśnie wspomniałem o klasach, dlatego warto byłoby w tym momencie przybliżyć możliwość dodawania własnych kaskadowych arkuszy stylów. W prezentowanych przykładach posłużymy się kodem napisanym za pomocą preprocesora *SASS* z użyciem metodologii *BEM* (ang. *Block Element Modifier*). Oba te zagadnienia wykraczają poza tematykę tej książki, dlatego szczerze zachęcam do zapoznania się z nimi, czy to na podstawie zasobów internetowych, czy innych pozycji książkowych. W skrócie można powiedzieć, iż *SASS* to skryptowy język służący do budowania stylów, interpretowany i kompilowany do klasycznych kaskadowych arkuszy, czyli plików CSS. W praktyce używa się dwóch różnych składni tego języka, przedstawię tu nowszą wersję nazwaną *SCSS* (ang. *Sassy CSS*). Pliki źródłowe tego języka zapisujemy w plikach z rozszerzeniem „scss".

Metodologia *BEM* to natomiast sposób, w jaki dobieramy nazewnictwo budowanych klas. Jej idea jest stosunkowo łatwa — budujemy kod stylów tak, by był on jak najłatwiejszy w ponownym użyciu, podzielony na komponenty, czytelny, przejrzysty i uporządkowany.

Tyle teorii, spróbujmy w praktyce zbudować kod arkusza stylów odpowiadający za nawigację — kod umieścimy w pliku *src/components/navigation/navigation.scss*, został on zaprezentowany na listingu 6.6.

LISTING 6.6. *Kod arkusza stylów dla komponentu navigation*

```
.navigation {
    color: var(--blue-500);
    &__elements {
        list-style-type: none;
        margin: 0;
        padding: 0;
        overflow: hidden;
```

```
        float:left;

    }
    &__element {
        float: left;
    }
    &__link {
        display: block;
        padding: 15px;
        text-decoration: none;
        font-size: 1.5em;
        &:hover {
          text-decoration: underline;
        }
    }
}
```

Powyżej zaprezentowany kod to oczywiście źródło języka skryptowego *SASS* w wersji składni *SCSS*. Elementem nadrzędnym dla wszystkich innych selektorów będzie klasa `navigation`. Za pomocą znaku `&` możemy doklejać inne pozostałe etykiety. W rezultacie po skompilowaniu powstaną poniższe selektory:

- `.navigation` z jedną regułą — ustawionym niebieskim kolorem,
- `.navigation__elements` z regułą ustawiającą typ listy, marginesy itp.; jak się łatwo domyślić, selektor ten będzie wykorzystywany do ostylowania listy nawigacyjnej w jednym rzędzie,
- `.navigation__element` — z jedną regułą „float:left", oczywiście jest to pojedynczy element listy,
- `.navigation__link` — style opisujące odnośnik,
- `.navvigation__link:hover` — styl dotyczący odnośnika, na który najechano myszą.

Istnieje wiele różnych teorii i sposobów na umiejscowienie kodu stylów, tu widzimy jeden z nich polegający na umieszczaniu źródeł preprocesora w katalogu, w którym znajduje się sam komponent. Dzięki temu jest on faktycznie całkowicie przenośny, wszystkie elementy go dotyczące składowane są w jednym miejscu.

Warto jednak sporządzić jedną listę wszystkich tego typu arkuszy komponentowych i umieścić ją w jednym pliku. W tym celu posłużymy się plikiem *src/assets/_all.scss* (listing 6.7), w którym za pomocą słowa kluczowego *@import* zaimportowano pojedynczy arkusz stylów komponentu.

LISTING 6.7. *Zawartość pliku src/assets/_all.scss*

```
@import '../components/navigation/navigation.scss';
```

Oczywiście plik ten może, a wręcz powinien być rozszerzany o nowe style przypisane do poszczególnych komponentów.

Nie pozostaje już nic innego, jak zmodyfikować plik główny aplikacji — *src/App.js* — i przygotować go do obsługi nowych funkcjonalności (listing 6.8).

LISTING 6.8. *Kod źródłowy pliku głównego aplikacji*

```
import React from "react";
import { BrowserRouter, Switch, Route } from "react-router-dom";
import { connect } from "react-redux";
import { LoginPage } from "./pages/login/login";
import { Navigation } from "./partials/navigation/navigation";
import PropTypes from "prop-types";
import { Row, Col, Container } from "react-bootstrap";

import { OrdersPage } from "./pages/orders/orders";
import { LogoutPage } from "./pages/login/logout";
import { PrivacyPage } from "./pages/privacy/privacy";
import { AboutPage } from "./pages/about/about";
import { ContactPage } from "./pages/contact/contact";
import "./assets/_all.scss";

class App extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
    const { message, type } = this.props;
    return (
      <div className="mt-3">
        <BrowserRouter>
          <Container>
            <Row>
              <Col md={12}>
                <Navigation />
                <div>
                  {message !== undefined && (
                    <div role="alert" className={`message ${type}`}>
                      {message}
                    </div>
                  )}
                </div>
              </Col>
            </Row>
          </Container>
          <Switch>
            <Route exact path="/" component={OrdersPage} />
            <Route exact path="/login" component={LoginPage} />
            <Route exact path="/logout" component={LogoutPage} />
            <Route exact path="/orders" component={OrdersPage} />
            <Route exact path="/contact" component={ContactPage} />
            <Route exact path="/about" component={AboutPage} />
            <Route exact path="/privacy" component={PrivacyPage} />
```

```
          </Switch>
        </BrowserRouter>
      </div>
    );
  }
  static propTypes = {
    dispatch: PropTypes.func.isRequired,
    message: PropTypes.string,
    type: PropTypes.string
  };
function mapStateToProps(state) {
  return {
    message: state.messageBagReducer.message,
    type: state.messageBagReducer.type
  };
}

export default connect(mapStateToProps, null)(App);
```

Na pierwszy rzut oka uwidocznia się bardzo długa lista importów, wśród nich wyróżnić można komponent nawigacji, komponenty związane z obsługą routingu — `BrowserRouter`, `Switch`, `Route`. Następnie wyraźnie oddzielone nową linią zostały komponenty będące poszczególnymi podstronami. Na końcu można także odnaleźć import głównego pliku kaskadowych arkuszy stylów.

Najważniejsze elementy tego kodu odnalazły swoje miejsce w funkcji `render()`, całość kodu została otoczona za pomocą komponentu `BrowserRouter`, a jako pierwsza uwidocznia się nawigacja — słusznie, jest to bowiem element, który powinien znaleźć się na samej górze strony. Pod nawigacją umiejscowiono panel informujący o komunikatach. W przypadku gdy `message` pobrane z Reduksa będzie zawierało treść, system powinien wyświetlić ją na ekranie. Dodatkowo za pomocą pola `type` przypisane zostanie dynamiczne ostylowanie zależne od rodzaju wiadomości.

Kolejny z elementów, który powinien nas szczególnie zainteresować, to komponent *Switch*. Można go potraktować jak przełącznik sprawdzający poszczególne wpisy i próbujący odnaleźć pasujący do wzorca adres. Możliwe do wywołania adresy umieszczone zostały w komponentach *Route*, w których `path` identyfikuje adres reprezentujący daną podstronę, natomiast `component` reprezentuje komponent, który będzie ją obsługiwał, ponadto `exact` oznacza, iż wzorzec musi być spełniony w 100%, np. adres `/privacy/id/1` nie będzie oznaczał tego samego wzorca co `/privacy`.

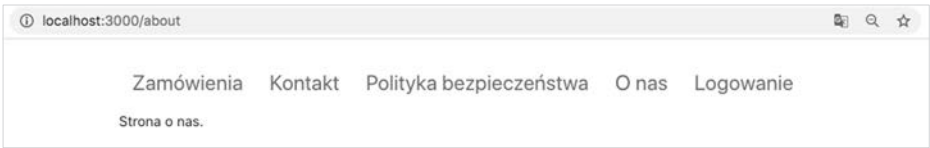
Analizując przykładowy poniższy wpis:

```
<Route exact path="/contact" component={ContactPage} />
```

możemy wnioskować, iż dotyczy dokładnie adresu `/contact`, który ma zostać obsłużony za pomocą komponentu `ContactPage`.

Spróbujmy teraz uruchomić aplikację i zobaczyć, czy wprowadzone zmiany przyniosły efekt.

Na rysunku 6.3 pokazano ekran, który powinniśmy zobaczyć.



RYSUNEK 6.3. *Ekran nawigacji*

Oczywiście wszystkie z podstron powinny działać, a po otwarciu podstrony „Logowanie" wyświetlony zostanie formularz logowania.

Wszystko to, co do tej pory udało się zrobić, dotyczy kwestii najbardziej oczywistych. Przyjrzymy się teraz sprawom bardziej specyficznym, jednak bardzo potrzebnym i ważnym.

6.1. Strona 404

Przykładem tego typu funkcjonalności jest wyświetlenie strony informującej o złym adresie URL. Tego typu element witryny jest czymś niezbędnym; dobra praktyka to poinformowanie gościa witryny o fakcie zboczenia z kursu — czy to wybraniu wygaśniętego odnośnika, czy ręcznego, złego wpisania adresu.

W celu wykonania tego elementu witryny musimy przejść do pliku *src/App.js* i dodać jeden wpis komponentu *Route*, w którym `path` przyjmie wartość „/**". Ponadto musimy oczywiście sprecyzować komponent, który zajmie się realizacją obsługi strony 404. W naszym przypadku posłuży do tego komponent *NotFoundPage*. Całość opisanej modyfikacji przedstawiona została na listingu 6.9.

LISTING 6.9. *Zmiany w pliku App.js*

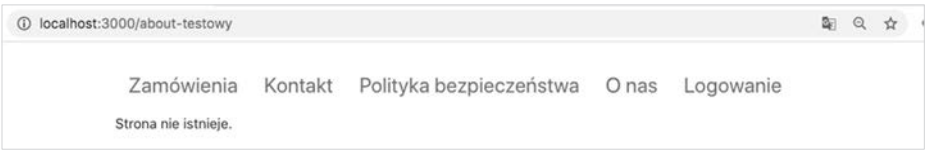
```
<Switch>
  <Route exact path="/" component={OrdersPage} />
  <Route exact path="/login" component={LoginPage} />
  <Route exact path="/logout" component={LogoutPage} />
  <Route exact path="/orders" component={OrdersPage} />
  <Route exact path="/contact" component={ContactPage} />
  <Route exact path="/about" component={AboutPage} />
  <Route exact path="/privacy" component={PrivacyPage} />
  <Route path="/**" component={NotFoundPage} />
</Switch>
```

Musimy również pamiętać o dodaniu samego komponentu, dlatego tworzymy nowy plik */pages/notFound/notFound.js* z zawartością przedstawioną na listingu 6.10.

LISTING 6.10. *Komponent reprezentujący brak znalezionego adresu*

```
import React from "react";
import { Row, Col, Container } from "react-bootstrap";
const NotFoundPage = () => (
  <Container>
    <Row>
      <Col>Strona nie istnieje.</Col>
    </Row>
  </Container>
);

export { NotFoundPage };
```

W rozpatrywanym przypadku, w momencie nieznalezienia pasującego adresu, aplikacja wyświetlić ma jedynie stosowny komunikat. Tak też się faktycznie dzieje, czego dowodem jest zrzut ekranu z rysunku 6.4 ilustrujący próbę wywołania adresu *about-testowy*, który w naszym systemie nie istnieje.



RYSUNEK 6.4. *Próba wywołania błędnego adresu podstrony*

Zaimplementowanie funkcjonalności obsługi złego adresu podstron nie jest może kluczowe, jednak ważne, by użytkownik wiedział, że popełnił błąd lub system nie zadziałał poprawnie.

6.2. Parametry

Często spotykaną funkcjonalnością dotyczącą szeroko pojętego routingu jest używanie sparametryzowanych adresów URL. Przykładem takiego adresu może być np. */osoba/100*. Wartości liczbowe często wykorzystywane są jako identyfikator zasobu, który ma zostać pobrany z bazy, czy też na którym mają zostać wykonane obliczenia matematyczne, czy logiczne. Cała idea przekazywania danych w parametrach wraz z uwzględnieniem ich ochrony to temat na osobną książkę. Dla nas w tej chwili najważniejsze jest, jak takie wartości odczytywać — samo tworzenie odnośników nie powinno być trudne, wszak jest to po prostu dodanie odpowiedniej wartości liczbowej.

Sam odczyt odbywa się, tu rozważamy sposób najprostszy, za pomocą tzw. hooka o nazwie `useParams()`. Ideę i technikę stosowania hooków przedstawię dalej, tu jedynie skorzystam z jednego z nich. Na teraz wystarczy stwierdzenie, iż jest to funkcja, która umożliwia to, czego potrzebujemy, czyli odczyt danych z parametrów. W celu przećwiczenia całego procesu dodamy nową podstronę, niech to będzie komponent *Order*. Przedstawia

on pojedyncze zamówienie w odróżnieniu do *Orders*, gdzie chcielibyśmy umieścić listę. Siłą rzeczy, w pojedynczym komponencie *Order* chcielibyśmy w pasku adresu przekazać identyfikator zamówienia, które ma zostać wyświetlone. Całość umieszczona zostaje więc w pliku *src/pages/orders/order.js*, którego zawartość widoczna jest na listingu 6.11.

LISTING 6.11. *Zawartość komponentu Order*

```
import React from "react";
import { Row, Col, Container } from "react-bootstrap";
import { useParams } from "react-router-dom";
const Order = () => {
  let { id } = useParams();
  return (
    <Container>
      <Row>
        <Col>Zamówienie numer {id}</Col>
      </Row>
    </Container>
  );
};
export { Order as OrderPage };
```

To, co nas najbardziej interesuje w powyższym kodzie, to linia szósta:

```
let { id } = useParams();
```

Widzimy, iż funkcja `useParams()`, zaimportowana z react-router-dom to wszystko, co musimy zrobić, gdyż za jej pomocą wyłuskamy wartość zapisaną pod etykietą `id`.

Jak jednak zadeklarować samą trasę routingu? W tym celu musimy wrócić do pliku *src/App.js* i dodać nowy import komponentu reprezentującego pojedyncze zamówienie. Musimy również dodać nową trasę — komponent *Route*, którego wartość `path` będzie wynosiła:

```
<Route exact path="/order/:id" component={OrderPage} />
```

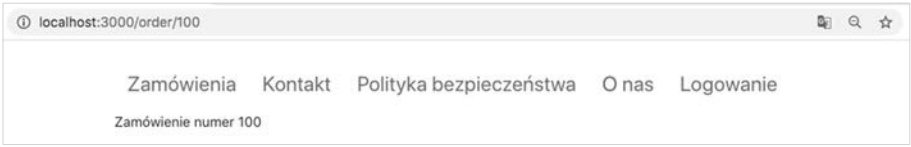
Jak się łatwo domyślić, zapis „:id” oznacza, iż właśnie pod tą nazwą przechowywać będziemy wartość liczbową.

Na listingu 6.12 zaprezentowano obie opisywane zmiany pliku *App.js*.

LISTING 6.12. *Dodanie trasy dla strony pojedynczego zamówienia*

```
import { OrderPage } from "./pages/orders/order";
(...)
<Route exact path="/order/:id" component={OrderPage} />
```

W tym momencie, gdy spróbujemy za pomocą przeglądarki internetowej wpisać adres `/order/100`, powinniśmy ujrzeć widok zbliżony do tego z rysunku 6.5.



RYSUNEK 6.5. *Widok pojedynczego zamówienia*

Oczywiście wartość zamówienia wypisywana na ekranie podawana jest w sposób dynamiczny, zmiana identyfikatora w pasku adresu zaowocuje aktualizacją wpisu. Jest to oczywiście dość trywialny przykład, którego jedyną ideą jest wyświetlanie numeru na ekranie, jednak może on być łatwo rozbudowany np. o wysłanie zapytania do API, które pobierze zasób zamówienia o identyfikatorze podanym w pasku. Tak jak napisałem wcześniej, kwestią do zastanowienia jest jawne podawanie tego typu numerów, gdyż same w sobie stanowią pewną wartość i informację o systemie. Technik zachowania poufności dla zasobów jest mnóstwo, można wykorzystać identyfikację przy użyciu *UUID* czy też inne techniki.

6.3. Query Parameters

Kolejną z technik przekazywania danych w pasku adresu są tzw. *Query Parameters*. W odróżnieniu do parametrów przekazywanych w ścieżce, a zaprezentowanych powyżej, służą one zazwyczaj do modyfikacji sposobu pobierania danych, np. poprzez ich filtrację czy sortowanie. Poprzednie wywołania sparametryzowane jawnie specyfikowały identyfikator zasobu, tutaj oczywiście tego nie mamy.

Podobnie jak wcześniej, tutaj też spróbujmy przećwiczyć tę funkcjonalność na przykładzie. Dodajmy więc do komponentu listy zamówień (*Orders*) możliwość precyzowania sortowania. W rezultacie dla przykładowego adresu:

```
/orders?sort=rosnaco
```

otrzymamy wartość `rosnaco`.

Przechodzimy więc do pliku */src/orders/orders.js* i modyfikujemy jego zawartość zgodnie z listingiem 6.13.

LISTING 6.13. *Modyfikacje listy zamówień*

```
import React from "react";
import { Row, Col, Container } from "react-bootstrap";
import { useLocation } from "react-router-dom";
class Orders extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
    let sort = new URLSearchParams(useLocation().search);
    return (
```

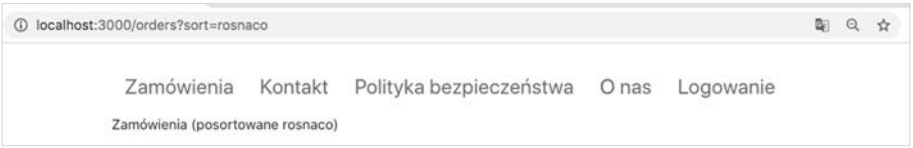
```
      <Container>
        <Row>
          <Col>Zamówienia (posortowane {sort.get("sorting")}</Col>{" "}
        </Row>
      </Container>
    );
  }
}
export { Orders as OrdersPage };
```

Tym razem w celu wyciągnięcia parametrów z paska adresu użyjemy kolejnego hooka udostępnionego przez react-router-dom. Będzie to `useLocation`, a samo wyodrębnianie rodzaju sortowania odbywa się w linii:

```
let sort = new URLSearchParams(useLocation().search);
```

Wartość pobrana z paska adresu zostanie pobrana i zapisana w zmiennej `sort`. Inaczej niż w przypadku pierwszej opcji, tym razem otrzymamy wartość będącą obiektem, toteż wyłuskanie z niego wartości odbywa się poprzez wywołanie funkcji `get` z odpowiednim parametrem będącym wartością klucza.

W takiej sytuacji nie musimy robić już nic więcej, w oknie przeglądarki przejdźmy do adresu `/orders?sort=rosnaco`, efekt zaprezentowano na rysunku 6.6.



RYSUNEK 6.6. *Efekt działania sortowania*

Analogicznie, samo wyświetlenie komunikatu o sposobie sortowania nie jest niczym niezwykłym. Chodzi tu jednak o sam fakt przekazania danych, natomiast ich wykorzystanie może przybrać dowolne formy, choć tu faktycznie najbardziej oczekiwana to posortowanie listy.

6.4. Zabezpieczanie podstron

Ostatnią z funkcjonalności związanych z routingiem opisanych w tej książce będzie zabezpieczenie podstron przed niepowołanym dostępem. I tu od razu należy podkreślić, iż zabezpieczenie to będzie czysto wizualne. Oznacza to, iż by mieć 100% pewności, iż zasób czy podstrona nie będą dostępne dla osób niepowołanych, musimy polegać na uwierzytelnianiu po stronie serwera. W przypadku opisanej poniżej aplikacji sprawdzanie dostępu do podstron ma charakter czysto praktyczny, pamiętajmy o tym, iż dostęp do danych wrażliwych, przechowywanych po stronie serwera, powinien być sprawdzany po jego stronie.

W przypadku prezentowanego kodu najpierw tworzymy nowy komponent o nazwie *PrivateRoute*. Jego idea i sposób działania są bardzo proste, jest to w praktyce komponent `Route` opisujący trasę do podstrony z pewną modyfikacją dotyczącą wyświetlania kodu JSX. Kiedy w *local storage* przeglądarki zapisany jest klucz `account`, to może zostać wyświetlona zawartość komponentu wraz z wszystkimi `props`.

W innym przypadku, gdy *local storage* nie zawiera wpisu o kluczu `account`, następuje przekierowanie do adresu `/login`.

Całość przedstawiona została na listingu 6.14 (plik *src/components/privateRoute.js*).

LISTING 6.14. *Zawartość komponentu PrivateRoute*

```
import React from "react";
import { Route, Redirect } from "react-router-dom";

export const PrivateRoute = ({ component: Component, ...rest }) => (
  <Route
    {...rest}
    render={props =>
      localStorage.getItem("account") ? (
        <Component {...props} />
      ) : (
        <Redirect
          to={{ pathname: "/login", state: { from: props.location } }}
        />
      )
    }
  />
);
```

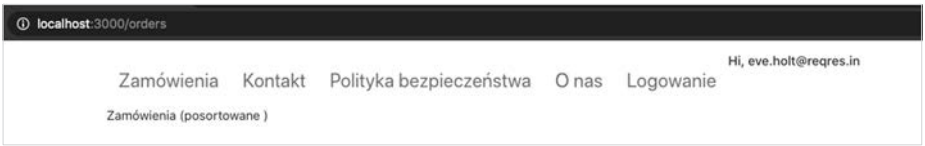
Pozostaje już tylko zadeklarować, który z komponentów powinien być traktowany jako ten, który wymaga zalogowania. W tym celu ponownie modyfikujemy plik *src/App.js*, w którym importujemy *PrivateRoute* i zamieniamy odpowiednie trasy.

Na listingu 6.15 przedstawiono kod, w którym komponenty związane z obsługą i wyświetleniem zamówień zostały zabezpieczone i wymagają logowania.

LISTING 6.15. *Zabezpieczone trasy*

```
<Switch>
  <Route exact path="/" component={OrdersPage} />
  <Route exact path="/login" component={LoginPage} />
  <Route exact path="/logout" component={LogoutPage} />
  <PrivateRoute exact path="/order/:id" component={OrderPage} />
  <PrivateRoute exact path="/orders" component={OrdersPage} />
  <Route exact path="/contact" component={ContactPage} />
  <Route exact path="/about" component={AboutPage} />
  <Route exact path="/privacy" component={PrivacyPage} />
  <Route path="/**" component={NotFoundPage} />
</Switch>
```

Rysunek 6.7 przedstawia dostęp do podstrony *orders*, gdzie uwidocznia się również fakt zalogowania do systemu przez użytkownika. Bez tego nie byłoby możliwości skorzystania z opisywanej podstrony.



RYSUNEK 6.7. *Dostęp do podstrony po zalogowaniu*

Podsumowując kończący się rozdział, warto jeszcze raz podkreślić, iż w procesie uwierzytelniania należy pamiętać o podziale na API i aplikację frontendową. To, że zapiszemy dane w *local storage* przeglądarki, jest tylko pomocą przy odwoływaniu się do serwera, gdzie każde z żądań powinno być sprawdzane pod kątem autentyczności tokena i czasu jego życia. Dotyczy to oczywiście opisywanych tu tras i podstron.

Sam routing jest narzędziem i techniką niezbędną przy tworzeniu nawet niedużych aplikacji, gdzie chcemy w logiczny sposób podzielić system na podstrony i moduły.

Reaguj z Reactem!

- **Poznaj React od podszewki**
- **Naucz się stosować tę bibliotekę w praktyce**
- **Twórz atrakcyjne interfejsy użytkownika**

React to bez wątpienia jedna z najpopularniejszych bibliotek służących do tworzenia interfejsów użytkownika. Zawdzięcza to dużej elastyczności, łatwości adaptacji i... reklamie, którą bibliotece zapewniło użycie jej przez kilka najbardziej rozpoznawalnych serwisów internetowych na świecie, takich jak Netflix, PayPal czy Imgur. Nie bez znaczenia są oczywiście możliwości Reacta, prostota jego zastosowania i czytelność kodu. Liczba ofert pracy dla programistów znających tę bibliotekę stale rośnie i nic nie zapowiada, aby w najbliższym czasie się to zmieniło.

Jeśli zatem marzy Ci się kariera frontendowca i stoisz przed wyborem technologii do nauki, bez wątpienia powinieneś się zainteresować Reactem! Sięgnij w tym celu po źródło wiedzy, które wprowadzi Cię w arkana tej biblioteki od strony praktycznej. Znajdziesz tu opis najważniejszych możliwości Reacta, nauczysz się tworzyć w nim aplikacje oraz pobierać i przechowywać dane. Poznasz też metodę uwierzytelniania i kontroli dostępu do aplikacji, a także dowiesz się, jak zapewnić właściwy routing i posługiwać się hookami. A wszystko to na podstawie praktycznych przykładów kodu.

- **Podstawy języka TypeScript**
- **Tworzenie aplikacji React**
- **Przegląd możliwości biblioteki**
- **Posługiwanie się danymi**
- **Uwierzytelnianie i routing**
- **Korzystanie z hooków**
- **Użyteczne przykłady kodu**
- **Praktyczne rozwiązania**

Stosuj bibliotekę React w praktyce!

Helion

helion.pl

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

ISBN 978-83-283-6850-7

Cena: 69,00 zł