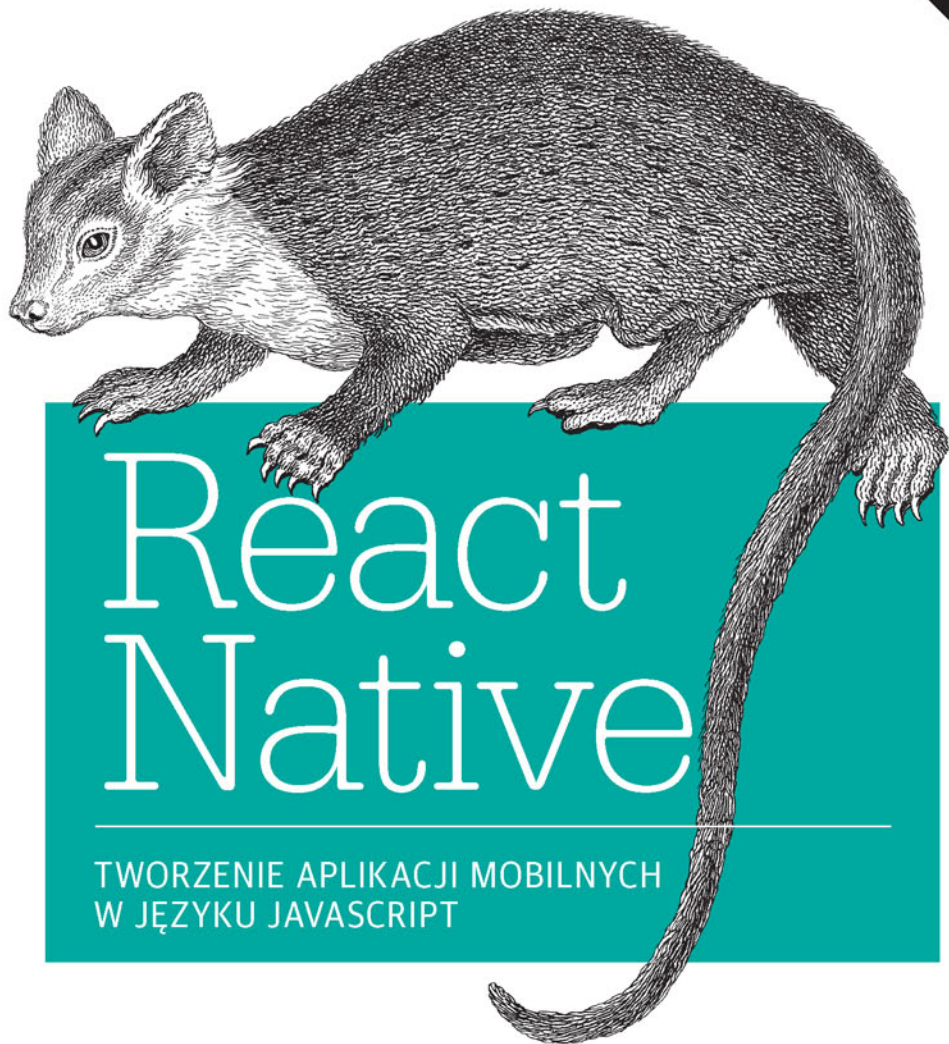


O'REILLY®

Wydanie II



# React Native

TWORZENIE APLIKACJI MOBILNYCH  
W JĘZYKU JAVASCRIPT

Helion 

Bonnie Eisenman

Tytuł oryginału: Learning React Native

Tłumaczenie: Patryk Wierzchoń

ISBN: 978-83-283-4424-2

© 2018 Helion SA

Authorized Polish translation of the English edition of Learning React Native ISBN 9781491989142 © 2018 Bonnie Eisenman. All rights reserved

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz HELION SA dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz HELION SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

HELION SA

ul. Kościuszki 1c, 44-100 GLIWICE

tel. 32 231 22 19, 32 230 98 63

e-mail: [helion@helion.pl](mailto:helion@helion.pl)

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:  
<ftp://ftp.helion.pl/przyklady/renaj2.zip>

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/renaj2>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzje.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

---

# Spis treści

<b>Przedmowa .....</b>	<b>9</b>
<b>1. Co to jest React Native? .....</b>	<b>13</b>
Zalety React Native	14
Doświadczenie programisty	15
Ponowne wykorzystanie kodu i dzielenie się wiedzą	16
Ryzyko i wady	16
Podsumowanie	17
<b>2. Praca z React Native .....</b>	<b>19</b>
Jak działa React Native?	19
Cykl renderowania	21
Tworzenie komponentów w React Native	22
Praca z widokami	22
Zastosowanie JSX	24
Style komponentów natywnych	25
API platform systemowych	26
Podsumowanie	27
<b>3. Tworzenie pierwszej aplikacji .....</b>	<b>29</b>
Konfiguracja środowiska	29
Konfiguracja środowiska — Create React Native App	30
Tworzenie pierwszej aplikacji za pomocą create-react-native-app	30
Podgląd aplikacji w iOS lub Androidzie	31

Konfiguracja środowiska — tradycyjne podejście	32
Tworzenie pierwszej aplikacji za pomocą react-native	33
Uruchamianie aplikacji w iOS	33
Uruchamianie aplikacji w Androidzie	35
Przegląd przykładowego kodu	35
Aplikacja Pogodynka	39
Obsługa wejścia użytkownika	40
Wyświetlanie danych	43
Pobieranie danych z sieci	46
Dodawanie obrazu w tle	50
Wszystko razem	52
Podsumowanie	54
<b>4. Komponenty w urządzeniach mobilnych .....</b>	<b>57</b>
Podobieństwa między elementami HTML i natywnymi	57
Komponent Text	58
Komponent Image	61
Obsługa dotyku i gestów	62
Podstawowe interakcje z komponentem Button	63
Komponent TouchableHighlight	63
PanResponder	66
Listy	73
Zastosowanie komponentu <FlatList>	74
Aktualizowanie zawartości <FlatList>	77
Wprowadzenie rzeczywistych danych	81
Zastosowanie komponentu <SectionList>	84
Nawigacja	88
Inne komponenty organizacyjne	88
Podsumowanie	89
<b>5. Style .....</b>	<b>91</b>
Deklaracja stylów i manipulowanie nimi	91
Style bezpośrednio w kodzie widoku	92
Style z wykorzystaniem obiektów	93
Zastosowanie StyleSheet.create	93
Łączenie stylów	94

Organizacja i dziedziczenie	95
Eksportowanie obiektów stylu	96
Przekazywanie stylów jako atrybutów	97
Ponowne wykorzystanie i współdzielenie stylów	97
Pozycjonowanie i projektowanie stylów	98
Zastosowanie flexboksa	98
Pozycjonowanie absolutne	102
Składanie wszystkiego razem	104
Podsumowanie	107
<b>6. API systemowe .....</b>	<b>109</b>
Korzystanie z geolokalizacji	110
Pobieranie lokalizacji użytkownika	110
Obsługa uprawnień	111
Testowanie geolokalizacji na emulatorach	112
Obserwowanie lokalizacji użytkownika	114
Ograniczenia	114
Modyfikacja Pogodynki	115
Korzystanie z kamery i obrazów użytkownika	118
Moduł CameraRoll	118
Pobieranie obrazów za pomocą GetPhotoParams	119
Renderowanie obrazu z rolki kamery	120
Wgranie obrazu na serwer	121
Przechowywanie trwałych danych za pomocą AsyncStorage	122
Aplikacja LepszaPogodynka	123
Komponent ProjektPogodynka	124
Komponent Prognoza	127
Komponent Button	128
Komponent PrzyciskLokalizacji	128
Komponent FotoTlo	129
Podsumowanie	131
<b>7. Moduły i kod natywny .....</b>	<b>133</b>
Instalacja bibliotek JavaScript za pomocą npm	133
Instalacja modułów natywnych	135
Zastosowanie komponentu Video	136

Anatomia modułu natywnego w języku Objective-C	137
Tworzenie modułu natywnego dla iOS w Objective-C	137
Implementacja RCTVideo	142
Moduły natywne w Javie	145
Tworzenie modułu natywnego dla Androida	145
Implementacja react-native-video w systemie Android	149
Wieloplatformowe komponenty natywne	151
Podsumowanie	152
<b>8. Kod dedykowany dla platformy .....</b>	<b>155</b>
Komponenty tylko dla jednej platformy	155
Komponenty z implementacjami dedykowanymi jednej platformie	156
Zastosowanie rozszerzeń plików z nazwą platformy	157
Zastosowanie komponentu Platform	159
Kiedy stosować komponenty dedykowane?	160
<b>9. Debugowanie i narzędzia programisty .....</b>	<b>161</b>
Metody debugowania JavaScript w przykładzie	161
Aktywacja opcji deweloperskich	161
Debugowanie z wykorzystaniem Console.log	162
Korzystanie z debugera JavaScript	165
Wykorzystanie narzędzi deweloperskich React	166
Narzędzia debugowania React Native	167
Sprawdzanie elementów	167
Czerwony ekran śmierci	168
Debugowanie poza kodem JavaScript	171
Częste problemy w środowisku deweloperskim	171
Częste problemy z Xcode	172
Częste problemy z Androidem	173
Packager React Native	174
Problemy związane z uruchamianiem aplikacji w iOS	174
Zachowanie symulatora	175
Testowanie kodu	176
Sprawdzanie typów za pomocą Flow	177
Testowanie za pomocą Jest	177
Testy migawkowe z użyciem Jest	178

Kiedy utkniesz	182
Podsumowanie	182
<b>10. Nawigacja i struktura w większych aplikacjach .....</b>	<b>183</b>
Aplikacja z fiskalami	183
Struktura projektu	186
Widoki aplikacji	187
Komponenty wielokrotnego użytku	193
Style	197
Modele danych	198
Zastosowanie biblioteki React Navigation	201
Tworzenie StackNavigatora	202
Zastosowanie navigation.navigate do przechodzenia między widokami	202
Konfigurowanie nagłówka za pomocą navigationOptions	205
Implementacja całej reszty	206
Podsumowanie	207
<b>11. Zarządzanie stanami w dużej aplikacji .....</b>	<b>209</b>
Zarządzanie stanami za pomocą Redux	209
Akcje	211
Reduktory	212
Połączenie biblioteki Redux z aplikacją	215
Zapisywanie danych za pomocą AsyncStorage	224
Podsumowanie i zadanie domowe	227
<b>Zakończenie .....</b>	<b>229</b>
<b>A Nowa składnia JavaScriptu .....</b>	<b>231</b>
<b>B Publikowanie aplikacji .....</b>	<b>237</b>
<b>C Praca z aplikacjami Expo .....</b>	<b>241</b>
<b>Skorowidz .....</b>	<b>242</b>





---

# Komponenty w urządzeniach mobilnych

W rozdziale 3. stworzyłeś prostą aplikację, Pogodynkę. W tym celu poznałeś podstawy tworzenia interfejsów użytkownika w React Native. W tym rozdziale przyjrzymy się bliżej komponentom dla urządzeń mobilnych w React Native i porównamy je z podstawowymi elementami HTML. Interfejsy mobilne opierają się na innych podstawowych elementach niż strony internetowe, więc musimy korzystać z innych komponentów.

W tym rozdziale zaczniemy od bardziej szczegółowego przeglądu najbardziej podstawowych komponentów: `<View>`, `<Image>` i `<Text>`. Następnie omówimy, jak dotyk i gesty wpływają na komponenty React Native oraz jak obsługiwać zdarzenia związane z dotykiem. Poznamy komponenty wyższego rzędu, takie jak zakładki, nawigatory i listy, które pozwalają na połączenie innych widoków w standardowe wzorce interfejsów urządzeń mobilnych.

## Podobieństwa między elementami HTML i natywnymi

Tworząc aplikacje internetowe, korzystamy z wielu różnych podstawowych elementów HTML. Zaliczamy do nich `<div>`, `<span>` i `<img>` oraz elementy porządkujące, takie jak `<ol>`, `<ul>` i `<table>` (moglibyśmy rozważyć też elementy takie jak `<audio>`, `<svg>`, `<canvas>` itd., ale w tej chwili je pominiemy).

Pracując z React Native, nie korzystamy z elementów HTML, ale korzystamy z niemalże analogicznych elementów (tabela 4.1).

Tabela 4.1. Podobne komponenty HTML i React Native

HTML	React Native
div	<View>
img	<Image>
span, p	<Text>
ul/ol, li	<FlatList>, elementy podrzędne

Ponieważ te elementy służą do podobnych celów, nie można ich zamienić. Przyjrzyjmy się, jak te komponenty działają na urządzeniach mobilnych z React Native i czym różnią się od ich przeglądarkowych odpowiedników.

### Czy mogę współdzielić kod React Native z aplikacjami sieciowymi?

Domyślnie React Native wspiera renderowanie wyłącznie dla iOS i Androida. Jeżeli chcesz renderować interfejsy kompatybilne ze stronami internetowymi, sprawdź `react-native-web` (<https://github.com/necolas/react-native-web>).

Jednak *możesz* przenosić każdy kod JavaScript, nawet zawierający komponenty React, jeśli nie renderuje żadnych elementów podstawowych. Możesz więc współdzielić kod, jeśli Twoja logika biznesowa oddzielona jest od renderowania widoku.

## Komponent Text

Renderowanie tekstu może wydawać się podstawową funkcjonalnością; praktycznie każda aplikacja musi gdzieś wyświetlać tekst. Jednak w React Native i programowaniu mobilnym działa to zupełnie inaczej niż w aplikacjach internetowych.

Pracując z tekstem w HTML, możesz wpisać tekst między wiele różnych znaczników. Dodatkowo możesz nadać im style, korzystając ze znaczników takich jak `<strong>` i `<em>`. Możesz więc mieć fragment HTML taki jak poniżej:

```
<p>Pchnąć w tę <em>łódź</em> jeża lub ośm skrzyń <strong>fig</strong>.</p>
```

W React Native tekst może być zadeklarowany tylko w komponencie `<Text>`, innymi słowy, poniższy kod jest nieprawidłowy:

```
<View>  
  Tak nie można!  
</View>
```

Zamiast tego opakuj go komponentem `<Text>`:

```
<View>
  <Text>Tak jest OK!</Text>
</View>
```

Korzystając z komponentu `<Text>`, nie możesz używać takich znaczników jak `<strong>` czy `<em>`. Możesz jednak skorzystać ze stylów w celu osiągnięcia podobnych efektów za pomocą atrybutów takich jak `fontWeight` czy `fontStyle`. Poniżej możesz zobaczyć, jak osiągnięto podobny efekt, używając stylów:

```
<Text>
  Pchnąć w tę <Text style={{fontStyle: "italic"}}>łódź</Text> jeża
  lub ośm skrzyń <Text style={{fontWeight: "bold"}}>fig</Text>.
</Text>
```

Takie podejście może jednak szybko okazać się rozwlekłe. Prawdopodobnie będziesz chciał tworzyć gotowe komponenty ze stylami tak jak w listingu 4.1, aby ułatwić sobie pracę z tekstem.

*Listing 4.1. Tworzenie komponentów wielokrotnego użytku do stylizowania tekstów*

```
import React, {Component} from "react";
import {StyleSheet, Text} from "react-native";
var styles = StyleSheet.create({
  pogrubiony: {
    fontWeight: "bold"
  },
  kursywa: {
    fontStyle: "italic"
  }
});

export class Strong extends Component {
  render() {
    return (
      <Text style={styles.pogrubiony}>
        {this.props.children}
      </Text>
    );
  }
}

export class Em extends Component {
  render() {
    return (
      <Text style={styles.kursywa}>
        {this.props.children}
      </Text>
    );
  }
}
```

```
    </Text>  
  );  
}
```

Gdy już utworzyłeś powyższe komponenty ze stylami, możesz ich swobodnie używać. Teraz nasz przykład w wersji React Native wygląda bardzo podobnie jak HTML (listing 4.2).

*Listing 4.2. Wykorzystanie komponentów ze stylami w renderowaniu tekstu*

```
<Text>  
  Pchnąć w tę <Em>łódź</Em> jeża  
  lub ośm skrzyń <Strong>fig</Strong>.  
</Text>
```

Analogicznie: pierwotnie React Native nie ma żadnej koncepcji nagłówków (h1, h2 itd.), ale z łatwością możesz stworzyć własne komponenty Text z odpowiednimi stylami i stosować je według potrzeb.

React Native wymusza zmianę podejścia do pracy ze sformatowanym tekstem. Ponieważ dziedziczenie jest ograniczone, nie masz możliwości domyślnego ustawienia czcionek we wszystkich tekstowych węzłach drzewa. Facebook zaleca rozwiązanie tego problemu poprzez użycie komponentów ze stylami:

Tracisz również możliwość ustawienia domyślnej czcionki dla całego drzewa podrzędnego. Zalecany sposób użycia spójnych czcionek w aplikacji to utworzenie komponentu MojTekst, który będzie je zawierał, i używanie go w aplikacji. Możesz za pomocą tego komponentu tworzyć dla innych rodzajów tekstu bardziej specyficzne komponenty, takie jak MojTekstNaglowka.

Dokumentacja komponentu <Text> (<http://bit.ly/1SVQxU3>) zawiera więcej szczegółów.

Zauważyłeś pewnie prawidłowość, że React Native woli ponowne wykorzystywanie komponentów ze stylami niż dziedziczenie lub ponowne stosowanie samych stylów. Chociaż jest to bardziej czasochłonne na początku, takie podejście pozwala lepiej wyizolować komponenty, dzięki czemu uzyskasz ten sam efekt, gdy wyświetlisz komponent w dowolnym miejscu aplikacji. Omówimy to zagadnienie szerzej w kolejnym rozdziale.

## Komponent Image

Jeśli elementy tekstowe to najbardziej podstawowe elementy w aplikacji, to obrazy są tuż za nimi pod względem powszechności zarówno w zastosowaniach mobilnych, jak i internetowych. Pisząc w HTML i CSS, załączamy obrazy na wiele różnych sposobów. Czasami używamy tagów `<img>`, czasami załączamy obraz przez CSS, np. gdy używamy atrybutu `background-image`. W React Native korzystamy z podobnego komponentu `<Image>`, zachowuje się on jednak inaczej.

Podstawowe użycie komponentu jest bardzo proste. Po prostu ustaw atrybut `source`:

```
<Image source={require("./pieski.png")} />
```

Ścieżka do obrazka rozwiązywana jest tak samo jak ścieżki do modułów JavaScript. Dlatego w powyższym przypadku plik *pieski.png* powinien znajdować się w tym samym folderze, co komponent, który go wykorzystuje.

Działa tu również pewna „magia” związana z nazwami plików. Jeżeli utworzysz pliki *pieski.ios.png* i *pieski.android.png*, zostaną one zaimportowane w zależności od systemu operacyjnego. Analogicznie: tworząc odpowiednie pliki z przyrostkiem `@2x` i `@3x`, umożliwisz menedżerowi pakietów React Native zastosowanie odpowiedniego pliku dla odpowiedniej rozdzielczości.

Warto również wspomnieć, że do swoich projektów możesz dołączać zasoby znajdujące się w sieci, zamiast pakować je do aplikacji. W aplikacji UIExplorer Facebook zamieścił taki przykład:

```
<Image source={{uri: 'https://facebook.github.io/react/img/logo_og.png'}}
  style={{width: 400, height: 400}} />
```

Korzystając z obrazków znajdujących się w internecie, musisz ręcznie określić rozmiary.

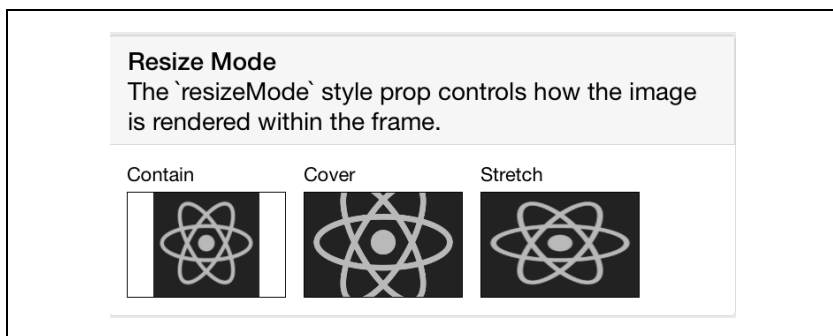
Pobieranie obrazów z internetu zamiast dołączania ich do zasobów ma pewne zalety. Podczas tworzenia aplikacji może być łatwiej skorzystać z tego podejścia, zamiast importować wszystkie zasoby z wyprzedzeniem. Zmniejsza się też rozmiar paczki z aplikacją, ponieważ użytkownicy nie muszą pobierać wszystkich zasobów. Oznacza to jednak, że Twoja aplikacja będzie uzależniona od transmisji danych u użytkownika. W większości przypadków będziesz unikał takiego stosowania zasobów.

Jeśli zastanawiasz się, jak korzystać z obrazów użytkownika, sprawdź omówienie obsługi kamery w rozdziale 6.

Ponieważ React Native kładzie nacisk na podejście oparte na komponentach, obrazy *muszą* być dołączane w komponentach `<Image>`, a nie w stylach. Przypomnij sobie rozdział 3., w którym chcieliśmy wstawić obrazek w tle naszej Pogodynki. W HTML i CSS użyłbyś w tym celu atrybutu `background-image`. Natomiast w React Native musisz użyć komponentu `<ImageBackground>` jako kontenera:

```
<ImageBackground source={require("./pieski.png")}>
  {/* Twój kod tutaj... */}
</ImageBackground>
```

Nadawanie stylów obrazom jest bardzo proste. Dodatkowo za pomocą określonych właściwości możesz określać sposób, w jaki renderowany jest obraz. Często będziesz korzystał z atrybutu `resizeMode`, który może przyjmować wartości `resize`, `cover` albo `contain`. Jest to dobrze przedstawione w aplikacji UIExplorer (rysunek 4.1).



Rysunek 4.1. Różnice między `resize`, `cover` i `contain`

Komponent `<Image>` jest łatwy w obsłudze i elastyczny. Będziesz go stosował bardzo często w swoich aplikacjach.

## Obsługa dotyku i gestów

Interfejsy dla przeglądarek internetowych są zazwyczaj zaprojektowane pod kątem obsługi myszy. By stworzyć uczucie interaktywności, używamy elementów wykrywających najechanie na element kursorem. W aplikacjach mobilnych

liczy się dotyk. Platformy mobilne mają własne normy dotyczące interakcji, które będziesz chciał wykorzystywać. Różnią się one w zależności od rodzaju platformy: iOS różni się od Androida, a ten z kolei różni się od Windows Phone.

React Native posiada różne biblioteki dające Ci przewagę w tworzeniu interfejsów dotykowych. W tej sekcji przyjrzymy się komponentowi `<Button>` oraz kontenerowi `<TouchableHighlight>`, a także niskopoziomym API dającym bezpośredni dostęp do zdarzeń dotyku.

## Podstawowe interakcje z komponentem Button

Jeżeli dopiero zaczynasz i chciałbyś mieć interaktywny przycisk, komponent `<Button>` to wszystko, czego potrzebujesz. Posiada on podstawowy interfejs pozwalający na zdefiniowanie koloru, tekstu i akcji, która wykona się po naciśnięciu.

```
<Button
  onPress={this._onPress}
  title="Naciśnij mnie"
  color="#841584"
  accessibilityLabel="Naciśnij przycisk"
/>
```

Ten komponent jest idealny na początek. Prawdopodobnie będziesz chciał stworzyć własne komponenty interaktywne. W tym celu musimy wykorzystać `<TouchableHighlight>`.

## Komponent TouchableHighlight

Wszystkie elementy, które powinny odpowiadać na dotyk użytkownika (przyciski, elementy sterujące itd.), powinny zazwyczaj być opakowane w komponent `<TouchableHighlight>`, który tworzy nakładkę na komponencie pokazującą reakcję widoczną dla użytkownika. W momencie dotknięcia zmieniony zostaje kolor komponentu. Jest to jedna z kluczowych interakcji, które powodują, że aplikacja mobilna wygląda natywnie, w przeciwieństwie do strony zoptymalizowanej pod kątem urządzeń mobilnych, gdzie obsługa dotyku jest ograniczona. Generalnie powinieneś trzymać się zasady, że komponentu `<TouchableHighlight>` należy używać wszędzie tam, gdzie pojawia się przycisk lub odnośnik do strony internetowej.

W najprostszym przypadku wystarczy, że opakujesz swój komponent w `<TouchableHighlight>`, dodając prosty efekt wizualny przy naciśnięciu. Komponent

ten umożliwia wykorzystanie w Twoich aplikacjach takich zdarzeń jak `onPressIn`, `onPressOut`, `onLongPress` i tym podobnych.

Na listingu 4.3 zademonstrowano, jak opakować komponent w `<TouchableHighlight>`, aby stworzyć interakcję z użytkownikiem.

*Listing 4.3. Zastosowanie komponentu `<TouchableHighlight>`*

```
<TouchableHighlight
  onPressIn={this._onPressIn}
  onPressOut={this._onPressOut}
  style={styles.dotyk}>
  <View style={styles.przycisk}>
    <Text style={styles.witaj}>
      {this.state.pressing ? 'IIK!' : 'NACIŚNIJ MNIE'}
    </Text>
  </View>
</TouchableHighlight>
```

Gdy użytkownik naciśnie przycisk, pojawia się na nim nakładka, a tekst ulega zmianie (rysunek 4.2).



*Rysunek 4.2. Zastosowanie `<TouchableHighlight>` do stworzenia interaktywnego wyglądu — stan przed naciśnięciem (po lewej) i podświetlony po naciśnięciu (po prawej)*



Jest to mało przydatny przykład, ale ilustruje podstawowe interakcje nadające przyciskowi wrażenie dotykalności. Nakładka jest kluczową częścią informacji zwracanej do użytkownika, która mówi mu, że przycisk może być użyty. Zauważ, że w tym przypadku nie musimy tworzyć żadnej logiki dla naszych stylów, komponent `<TouchableHighlight>` zrobi to za nas.

Listing 4.4 zawiera cały kod tego komponentu.

*Listing 4.4. DotykDemo.js ilustruje zastosowanie `<TouchableHighlight>`*

```
import React, { Component } from 'react';
import {
  StyleSheet,
  Text,
  TouchableHighlight,
  View
} from 'react-native';

export default class Button extends Component {
  constructor(props) {
    super(props);
    this.state = { pressing: false };
  }

  _onPressIn = () => {
    this.setState({ pressing: true });
  };

  _onPressOut = () => {
    this.setState({ pressing: false });
  };

  render() {
    return (
      <View style={styles.kontener}>
        <TouchableHighlight
          onPressIn={this._onPressIn}
          onPressOut={this._onPressOut}
          style={styles.dotyk}
        >

          <View style={styles.przycisk}>
            <Text style={styles.witaj}>
              {this.state.pressing ? "IIK!" : "NACIŚNIJ MNIE"}
            </Text>
          </View>

        </TouchableHighlight>
      </View>
    );
  }
}
```

```

    });
  }
}

const styles = StyleSheet.create({
  kontener: {
    flex: 1,
    justifyContent: "center",
    alignItems: "center",
    backgroundColor: "#F5FCFF"
  },
  witaj: { fontSize: 20, textAlign: "center", margin: 10, color: "FFFFFF" },
  dotyk: { borderRadius: 100 },
  przycisk: {
    backgroundColor: "#FF0000",
    borderRadius: 100,
    height: 200,
    width: 200,
    justifyContent: "center"
  }
});

```

Możesz spróbować zmodyfikować powyższy kod tak, aby komponent reagował na zdarzenia `onPress` i `onLongPress`. Najlepszą metodą sprawdzenia, jak te zdarzenia wpływają na interakcję z użytkownikiem, jest eksperymentowanie na urządzeniu fizycznym.

## PanResponder

`PanResponder` — w przeciwieństwie do `<TouchableHighlight>` — nie jest komponentem, a klasą React Native. Obiekt `gestureState` umożliwia dostęp do surowych danych o współrzędnych oraz informacji o szybkości i łącznej długości dotknięcia.

Aby wykorzystać klasę `PanResponder` w komponencie, musimy stworzyć obiekt typu `PanResponder` i dodać go do komponentu w metodzie `render`.

Utworzenie instancji `PanResponder` wymaga określenia odpowiednich metod obsługujących zdarzenia (listing 4.5).

*Listing 4.5. Utworzenie obiektu `PanResponder` wymaga przekazania kilku odwołań*

```

this._panResponder = PanResponder.create({
  onStartShouldSetPanResponder: this._obslugaStartUstawPanResponder,
  onMoveShouldSetPanResponder: this._obslugaRuchUstawPanResponder,
  onPanResponderGrant: this._obslugaPanResponderPrzyznano,
  onPanResponderMove: this._obslugaPanResponderRuch,
});

```

```

    onPanResponderRelease: this._obsługaPanResponderKoniec,
    onPanResponderTerminate: this._obsługaPanResponderKoniec,
  });

```

Te sześć funkcji umożliwia dostęp do całego cyklu życia zdarzenia dotyku. Funkcje `onStartShouldSetPanResponder` i `onMoveShouldSetPanResponder` określają, czy aplikacja powinna zareagować na dotyk. Funkcja `onPanResponderGrant` zostanie wywołana, gdy zdarzenie dotyku rozpocznie się, a `onPanResponderRelease` i `onPanResponderTerminate` będą wywołane pod koniec zdarzenia. `onPanResponderMove` pozwala na dostęp do danych w trakcie trwania zdarzenia.

Następnie używamy składni **spread**, żeby dodać obiekt `PanResponder` do komponentu w metodzie `render` (listing 4.6).

*Listing 4.6. Dodawanie obiektu `PanResponder` przy użyciu składni `spread`*

```

render: function() {
  return (
    <View
      {...this._panResponder.panHandlers}>
      { /* Treść tutaj */ }
    </View>
  );
}

```

Po wykonaniu tego kodu funkcje obsługi zdarzeń przekazane do `PanResponder`. `create` będą wywoływane podczas odpowiednich zdarzeń, jeśli dotyk ma swój początek w danym widoku.

Rysunek 4.3 pokazuje działanie przykładu. Wyświetlane jest koło, które możesz przesuwać po ekranie, a aplikacja wyświetli jego współrzędne.

Aby to zaimplementować, musimy wgryźć się w strukturę funkcji obsługujących zdarzenia. Początek jest prosty: implementując `_obsługaStartUstawPanResponder` i `_obsługaRuchUstawPanResponder`, określamy, czy chcemy, aby dany komponent otrzymywał zdarzenia związane z dotykiem (zobacz listing 4.7).

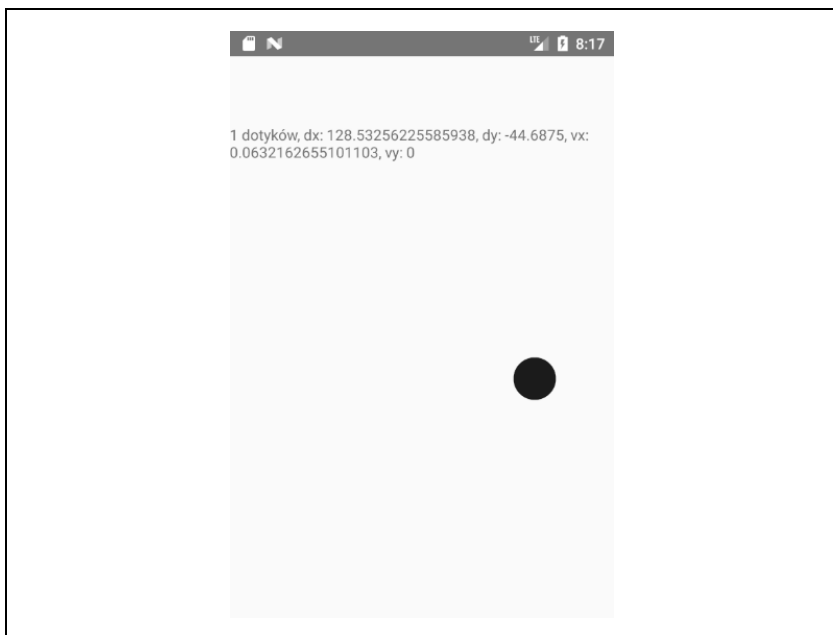
*Listing 4.7. Dwie pierwsze funkcje zwracające `true`*

```

_obsługaStartUstawPanResponder = (event, stanGestu) => {
  // Czy obsługa zdarzeń powinna być aktywna, jeśli użytkownik nacisnął koło?
  return true;
};

_obsługaRuchUstawPanResponder = (event, stanGestu) => {
  // Czy obsługa zdarzeń powinna być aktywna, jeśli użytkownik przesunie palcem nad kołem?
  return true;
};

```



Rysunek 4.3. Demo biblioteki PanResponder

Następnie chcemy w `_obsługaPanResponderRuch` wykorzystać współrzędne do zaktualizowania położenia koła (zobacz listing 4.8).

Listing 4.8. Aktualizowanie pozycji koła za pomocą `_obsługaPanResponderRuch`

```
_obsługaPanResponderRuch = (event, stanGestu) => {
  this.setState({
    idStanu: stanGestu.stateID,
    ruchX: stanGestu.moveX,
    ruchY: stanGestu.moveY,
    x0: stanGestu.x0,
    y0: stanGestu.y0,
    dx: stanGestu.dx,
    dy: stanGestu.dy,
    vx: stanGestu.vx,
    vy: stanGestu.vy,
    aktywneDotyki: stanGestu.numberActiveTouches
  });

  // Obliczanie bieżącej pozycji za pomocą delt
  this._styleKol.style.left = this._poprzedniLewo + stanGestu.dx;
  this._styleKol.style.top = this._poprzedniGora + stanGestu.dy;
  this._zaktualizujPozycje();
};
```

```

};
_zaktualizujPozycje = () => {
  this.kolo && this.kolo.setNativeProps(this._styleKolo);
};

```

Zauważ, że aktualizacja pozycji odbywa się poprzez metodę `setNativeProps`.



Pracując z animacjami, możesz wykorzystać `setNativeProps`, aby zmodyfikować komponent bezpośrednio, bez korzystania z `props` i `state`. Pozwala to ominąć hierarchię renderowania komponentów. Korzystaj z tego rozwiązania oszczędnie.

Zaimplementujmy teraz `_obsługaPanResponderPrzyznano` i `_obsługaPanResponderKoniec` tak, aby koło zmieniało kolor, gdy zostanie dotknięte (zobacz listing 4.9).

#### Listing 4.9. Implementacja podświetlenia

```

_podswietl = () => {
  this.kolo &&
    this.kolo.setNativeProps({
      style: { backgroundColor: KOLOR_PODSWIETLENIA_KOLA }
    });
};

_niePodswietl = () => {
  this.kolo &&
    this.kolo.setNativeProps({ style: { backgroundColor: KOLOR_KOLA } });
};

_obsługaPanResponderPrzyznano = (event, stanGestu) => {
  this._podswietl();
};

_obsługaPanResponderKoniec = (event, stanGestu) => {
  this._niePodswietl();
  this._poprzedniLewo += stanGestu.dx;
  this._poprzedniGora += stanGestu.dy;
};

```

Na listingu 4.10 połączono wszystkie te elementy razem, aby stworzyć interaktywne demo `PanRespondera`.

#### Listing 4.10. Aplikacja `Dotyk/PanDemo.js` ilustruje zastosowanie biblioteki `PanResponder`

```

// Zaadaptowano z https://github.com/facebook/react-native/blob/master/Examples/UIExplorer/PanResponderExample.js

"use strict";

```

```

import React, { Component } from "react";
import { StyleSheet, PanResponder, View, Text } from "react-native";

const RÓZMIAR_KOLA = 40;
const KOLOR_KOLA = "blue";
const KOLOR_PODSWIETLENIA_KOLA = "green";

export default class PanResponderPrzyklad extends Component {
  // Ustaw wartości początkowe.
  _panResponder = {};
  _poprzedniLewo = 0;
  _poprzedniGora = 0;
  _styleKol = {};
  kolo = null;

  constructor(props) {
    super(props);
    this.state = {
      aktywneDotyki: 0,
      ruchX: 0,
      ruchY: 0,
      x0: 0,
      y0: 0,
      dx: 0,
      dy: 0,
      vx: 0,
      vy: 0
    };
  }

  componentWillMount() {
    this._panResponder = PanResponder.create({
      onStartShouldSetPanResponder: this._obsługaStartUstawPanResponder,
      onMoveShouldSetPanResponder: this._obsługaRuchUstawPanResponder,
      onPanResponderGrant: this._obsługaPanResponderPrzyznano,
      onPanResponderMove: this._obsługaPanResponderRuch,
      onPanResponderRelease: this._obsługaPanResponderKoniec,
      onPanResponderTerminate: this._obsługaPanResponderKoniec
    });
    this._poprzedniLewo = 20;
    this._poprzedniGora = 84;
    this._styleKol = {
      style: { left: this._poprzedniLewo, top: this._poprzedniGora }
    };
  }

  componentDidMount() {
    this._zaktualizujPozycje();
  }
}

```

```

render() {
  return (
    <View style={styles.kontener}>
      <View
        ref={kolo => {
          this.kolo = kolo;
        }}
        style={styles.kolo}
        {...this._panResponder.panHandlers}
      />
      <Text>
        {this.state.aktywneDotyki} dotyków,
        dx: {this.state.dx},
        dy: {this.state.dy},
        vx: {this.state.vx},
        vy: {this.state.vy}
      </Text>
    </View>
  );
}

// _podswietl i _niePodswietl wywoływane są przez metody PanRespondera,
// zapewniając użytkownikowi informację graficzną.
_podswietl = () => {
  this.kolo &&
  this.kolo.setNativeProps({
    style: { backgroundColor: KOLOR_PODSWIETLENIA_KOLA }
  });
};

_niePodswietl = () => {
  this.kolo &&
  this.kolo.setNativeProps({ style: { backgroundColor: KOLOR_KOLA } });
};

// Za pomocą setNativeProps kontrolujemy właściwości koła.
_zaktualizujPozycje = () => {
  this.kolo && this.kolo.setNativeProps(this._styleKolo);
};

_obslugaStartUstawPanResponder = (event, stanGestu) => {
  // Czy obsługa zdarzeń powinna być aktywna, jeśli użytkownik nacisnął koło?
  return true;
};

_obslugaRuchUstawPanResponder = (event, stanGestu) => {
  // Czy obsługa zdarzeń powinna być aktywna, jeśli użytkownik przesunie palcem nad kołem?
  return true;
};

```

```

    _obsługaPanResponderPrzyznano = (event, stanGestu) => {
      this._podswietl();
    };

    _obsługaPanResponderRuch = (event, stanGestu) => {
      this.setState({
        idStanu: stanGestu.stateID,
        ruchX: stanGestu.moveX,
        ruchY: stanGestu.moveY,
        x0: stanGestu.x0,
        y0: stanGestu.y0,
        dx: stanGestu.dx,
        dy: stanGestu.dy,
        vx: stanGestu.vx,
        vy: stanGestu.vy,
        aktywneDotyki: stanGestu.numberActiveTouches
      });

      // Obliczanie bieżącej pozycji za pomocą delt
      this._styleKol.style.left = this._poprzedniLewo + stanGestu.dx;
      this._styleKol.style.top = this._poprzedniGora + stanGestu.dy;
      this._zaktualizujPozycje();
    };

    _obsługaPanResponderKoniec = (event, stanGestu) => {
      this._niePodswietl();
      this._poprzedniLewo += stanGestu.dx;
      this._poprzedniGora += stanGestu.dy;
    };
  }

  const styles = StyleSheet.create({
    koło: {
      width: ROZMIAR_KOLA,
      height: ROZMIAR_KOLA,
      borderRadius: ROZMIAR_KOLA / 2,
      backgroundColor: KOLOR_KOLA,
      position: "absolute",
      left: 0,
      top: 0
    },
    kontener: { flex: 1, paddingTop: 64 }
  });

```

Jeżeli planujesz implementować własne komponenty rozpoznające gesty, proponuję, żebyś poeksperymentował z tą aplikacją na urządzeniu fizycznym. Dzięki temu nabierzesz wyczucia co do działania konkretnych ustawień. Rysunek 4.3 przedstawia zrzut ekranu, ale powinieneś spróbować samemu na prawdziwym urządzeniu.



## Wybór sposobu obsługi dotyku

Jak zdecydować, które API do obsługi dotyku i gestów wybrać w swojej aplikacji? Zależy, co chcesz stworzyć.

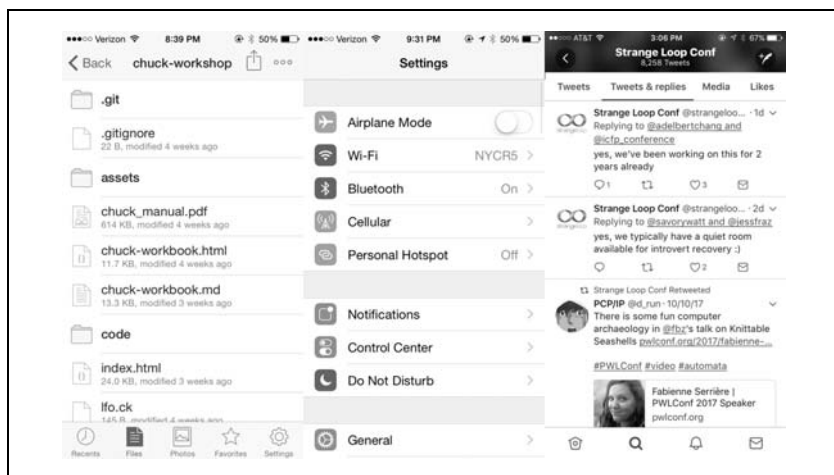
Aby zapewnić użytkownikowi podstawową informację zwrotną i wskazać, że element reaguje na dotyk, użyj komponentu `<TouchableHighlight>`.

Jeśli chcesz zaimplementować własne interfejsy dotykowe, wykorzystaj bibliotekę `PanResponder`. Jeśli projektujesz grę lub aplikację z nietypowym interfejsem, będziesz musiał poświęcić trochę czasu na stworzenie własnych interakcji przy użyciu tych narzędzi.

W wielu aplikacjach nie będzie konieczności implementowania własnej obsługi dotyku. W kolejnej sekcji przyjrzymy się komponentom wysokopoziomym, które za Ciebie implementują różne wzorce interfejsów użytkownika.

## Listy

Zauważ, że wielu użytkowników postrzeże listy jako centralny element aplikacji. Możesz przyrzeć się temu wzorcowi interakcji w aplikacjach Dropbox, Twitter oraz ustawieniach (Settings) systemu iOS (rysunek 4.4). Ich sercem jest przesuwany kontener zawierający inne widoki. Ten bardzo prosty wzorec jest nieodłącznym elementem wielu interfejsów graficznych.



Rysunek 4.4. Wykorzystanie list przez aplikacje Dropbox, Twitter i Settings iOS

React Native posiada dwa komponenty list z wygodnymi interfejsami. Komponent `<FlatList>` przeznaczony jest do pracy z długimi listami zmieniających się, lecz podobnie ustrukturyzowanych danych. Jest on zoptymalizowany pod kątem wydajności. Drugi z komponentów, `<SectionList>`, jest dedykowany dla danych rozbitych na kilka sekcji, zazwyczaj posiadających nagłówki. Jest on analogiczny do elementu systemu iOS `UITableView`. Powyższe komponenty mogą być stosowane w przeważającej większości przypadków. Jeśli jednak musisz zajrzeć pod maskę i dodać własne sposoby obsługi list, skorzystaj z komponentu `<VirtualizedList>`.



Optymalizacja wydajności renderowania listy jest często problematyczna, ponieważ różne przypadki użycia wymagają różnych podejść. Czy użytkownik przesuwa listę kontaktów szybko, aby znaleźć dany kontakt, czy może powoli przegląda listę obrazów? Czy lista jest jednorodna, czy każdy jej element wyświetlany jest inaczej? Jeżeli zajmujesz się problemami z wydajnością aplikacji, zwróć szczególną uwagę na listy.

W tej sekcji stworzymy aplikację wyświetlającą listę bestsellerów „New York Timesa” i umożliwiającą przeglądanie danych o każdej książce, tak jak pokazano na rysunku 4.5. Stworzymy dwie wersje, jedną z wykorzystaniem `<FlatList>`, a drugą — z `<SectionList>`.

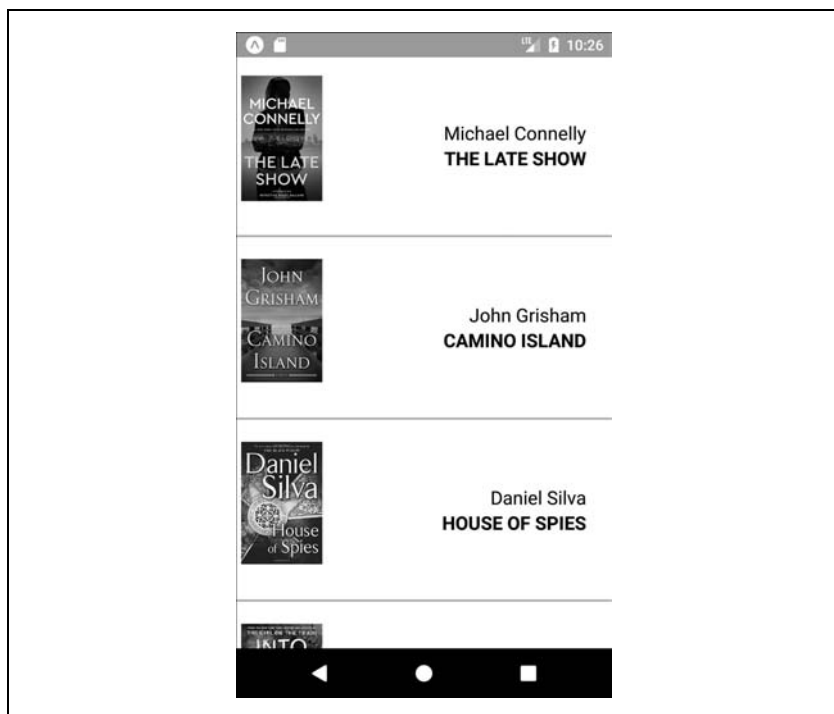
Jeśli chcesz, możesz pobrać swój klucz API ze strony „New York Timesa” (<http://developer.nytimes.com/apps/mykeys>). W przeciwnym razie skorzystaj z klucza umieszczonego w kodzie przykładowym.

## Zastosowanie komponentu `<FlatList>`

Zacniemy od komponentu `<FlatList>`, który posiada dwa atrybuty: `data` i `renderItem`.

```
<FlatList
  data={this.state.data}
  renderItem={this._renderItem} />
```

Atrybut `data` to dane renderowane przez `<FlatList>`. Powinna to być tablica, w której każdy z elementów ma unikatowy klucz, oraz inne atrybuty, które uznasz za użyteczne.



Rysunek 4.5. Aplikacja ListaKsiazek, którą będziemy tworzyć

Atrybut `renderItem` powinien przyjmować funkcję, która zwróci komponent utworzony na podstawie danych z jednego elementu tablicy `data`.

Przykład użycia komponentu `<FlatList>` został umieszczony w poniższym listingu.

#### Listing 4.11. *Bestseller/ProstaLista.js*

```
import React, { Component } from "react";
import { StyleSheet, Text, View, FlatList } from "react-native";

export default class ProstaLista extends Component {
  constructor(props) {
    super(props);
    this.state = {
      dane: [
        { key: "a" },
        { key: "b" },
      ],
    };
  }
}
```

```

        { key: "c" },
        { key: "d" },
        { key: "dłuższy przykład" },
        { key: "e" },
        { key: "f" },
        { key: "g" },
        { key: "h" },
        { key: "i" },
        { key: "j" },
        { key: "k" },
        { key: "l" },
        { key: "m" },
        { key: "n" },
        { key: "o" },
        { key: "p" }
    ]
  };
}

_renderujElement = dane => {
  return <Text style={styles.wiersz}>{dane.item.key}</Text>;
};

render() {
  return (
    <View style={styles.kontener}>
      <FlatList data={this.state.dane}
renderItem={this._renderujElement} />
    </View>
  );
}
}

const styles = StyleSheet.create({
  kontener: {
    flex: 1,
    justifyContent: "center",
    alignItems: "center",
    backgroundColor: "#F5FCFF"
  },
  wiersz: { fontSize: 24, padding: 42, borderWidth: 1, borderColor:
↳ "#DDDDDD" }
});

```

Jednym z haczyków w zastosowaniu tego komponentu jest przekazywanie obiektu z danymi do wyświetlenia w funkcji `renderItem` poprzez atrybut `item`.

```

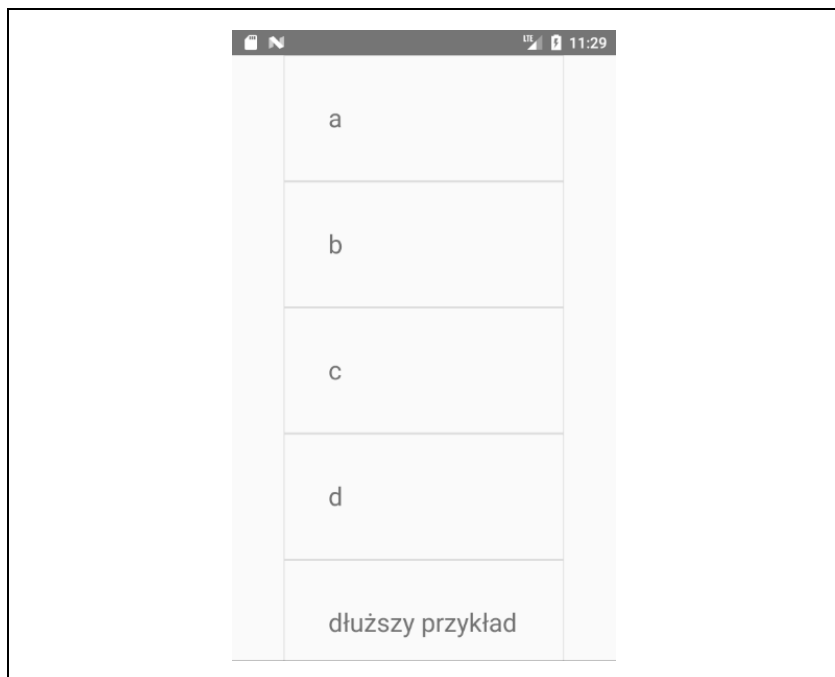
_renderujElement = dane => {
  return <Text style={styles.wiersz}>{dane.item.key}</Text>;
};

```

Możemy to jeszcze bardziej uprościć poprzez zastosowanie poniższej składni:

```
_renderujElement = ({item}) => {  
  return <Text style={styles.wiersz}>{item.key}</Text>;  
};
```

Aplikacja powinna wyglądać tak jak na rysunku 4.6.



Rysunek 4.6. Komponent *ProstaLista* demonstruje podstawowe zastosowanie `<FlatList>`

## Aktualizowanie zawartości `<FlatList>`

Co, jeśli chcemy zrobić coś więcej? Stwórzmy `<FlatList>` z bardziej złożonymi danymi. Skorzystamy z API „New York Times” do stworzenia prostej aplikacji renderującej listę bestsellerów „New York Timesa”.

Na początek sami utworzymy przykładowe dane przypominające odpowiedź z API „New York Timesa” — jak na listingu 4.12.

*Listing 4.12. Testowe dane obrazujące odpowiedź API*

```
const testoweKsiazki = [  
  {  
    rank: 1,  
    title: "GATHERING PREY",  
    author: "John Sandford",  
    book_image: "http://du.ec2.nytimes.com.s3.amazonaws.com/prd/books/  
↳9780399168796.jpg"  
  },  
  {  
    rank: 2,  
    title: "MEMORY MAN",  
    author: "David Baldacci",  
    book_image: "http://du.ec2.nytimes.com.s3.amazonaws.com/prd/books/  
↳9781455586387.jpg"  
  }  
];
```

Następnie musimy dodać komponent, który będzie renderował te dane. Komponent `<Ksiazka>` pokazany na listingu 4.13 wykorzystuje połączenie komponentów `<View>`, `<Text>` i `<Image>` do wyświetlania podstawowych informacji dotyczących książki.

*Listing 4.13. Bestseller/Ksiazka.js*

```
import React, { Component } from "react";  
  
import { StyleSheet, Text, View, Image, ListView } from "react-native";  
  
const styles = StyleSheet.create({  
  ksiazka: {  
    flexDirection: "row",  
    backgroundColor: "#FFFFFF",  
    borderBottomColor: "#AAAAAA",  
    borderBottomWidth: 2,  
    padding: 5,  
    height: 175  
  },  
  okladka: { flex: 1, height: 150, resizeMode: "contain" },  
  info: {  
    flex: 3,  
    alignItems: "flex-end",  
    flexDirection: "column",  
    alignSelf: "center",  
    padding: 20  
  },  
  autor: { fontSize: 18 },  
  tytul: { fontSize: 18, fontWeight: "bold" }  
});
```

```

export default class Ksiazka extends Component {
  render() {
    return (
      <View style={styles.ksiazka}>
        <Image style={styles.okladka} source={{ uri: this.props.okladkaURL }} />
        <View style={styles.info}>
          <Text style={styles.autor}>{this.props.autor}</Text>
          <Text style={styles.tytul}>{this.props.tytul}</Text>
        </View>
      </View>
    );
  }
}

```

Jeśli chcemy wykorzystać ten komponent, musimy zaktualizować funkcję `_renderujElement`. Komponent `<Ksiazka>` wymaga uzupełnienia trzech atrybutów: `okladkaURL`, `autor`, `tytul`.

```

_renderujElement = ({ item }) => {
  return (
    <Ksiazka
      okladkaURL={item.book_image}
      tytul={item.key}
      autor={item.author}
    />
  );
};

```

Pamiętaj, że każdy element we `<FlatList>` musi mieć unikatowy klucz. Listing 4.14 przedstawia funkcję pomocniczą, która dodaje klucze do elementów w tablicy.

*Listing 4.14. Metoda `_dodajKluczeDoKsiazek` dodaje klucz do każdego elementu w tablicy książki*

```

_dodajKluczeDoKsiazek = ksiazki => {
  return ksiazki.map(ksiazka => {
    return Object.assign(ksiazka, { key: ksiazka.title });
  });
};

```

Mając metodę pomocniczą, możemy zaktualizować stan początkowy za pomocą danych testowych, jak na listingu 4.12:

```

constructor(props) {
  super(props);
  this.state = { dane: this._dodajKluczeDoKsiazek(testoweKsiazki) };
}

```

Jeżeli połączymy te wszystkie elementy, kod naszej aplikacji z bestsellerami uzupełniony danymi testowymi będzie wyglądał jak na listingu 4.15 i da nam efekt jak na rysunku 4.7.

*Listing 4.15. Bestsellery/ListaNKsiazekTest.js*

```
import React, { Component } from "react";
import { StyleSheet, Text, View, Image, FlatList } from "react-native";
import Ksiazka from "./Ksiazka";

const testoweKsiazki = [
  {
    rank: 1,
    title: "GATHERING PREY",
    author: "John Sandford",
    book_image: "http://du.ec2.nytimes.com.s3.amazonaws.com/prd/books/
↳9780399168796.jpg"
  },
  {
    rank: 2,
    title: "MEMORY MAN",
    author: "David Baldacci",
    book_image: "http://du.ec2.nytimes.com.s3.amazonaws.com/prd/books/
↳9781455586387.jpg"
  }
];

export default class ListaKsiazekTest extends Component {
  constructor(props) {
    super(props);
    this.state = { dane: this._dodajKluczeDoKsiazek(testoweKsiazki) };
  }

  _renderujElement = ({ item }) => {
    return (
      <Ksiazka
        okladkaURL={item.book_image}
        tytul={item.key}
        autor={item.author}
      />
    );
  };

  _dodajKluczeDoKsiazek = ksiazki => {
    return ksiazki.map(ksiazka => {
      return Object.assign(ksiazka, { key: ksiazka.title });
    });
  };
};
```



```

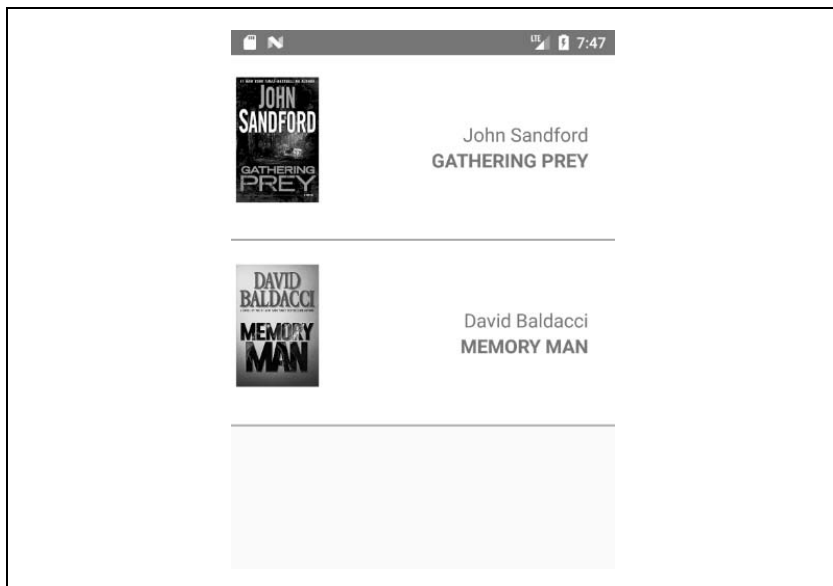
render() {
  return <FlatList data={this.state.dane} renderItem={this._renderujElement} />;
}

```

```

const styles = StyleSheet.create({ kontener: { flex: 1, paddingTop: 22 } });

```



Rysunek 4.7. Dane testowe wyświetlane za pomocą <FlatList>

## Wprowadzenie rzeczywistych danych

Aplikacja z danymi testowymi wygląda przyzwoicie. Najwyższy czas popracować z prawdziwymi danymi. Kod pozwalający na dostęp do API znajduje się na listingu 4.16.

Listing 4.16. *Bestsellery/ NYT.js*

```

const API_KEY = "73b19491b83909c7e07016f4bb4644f9:2:60667290";
const LIST_NAME = "hardcover-fiction";
const API_STEM = "https://api.nytimes.com/svc/books/v3/lists";
function pobierzKsiazki(list_name = LIST_NAME) {
  let url = `${API_STEM}/${LIST_NAME}?response-format=
  ↪json&api-key=${API_KEY}`;
  return fetch(url)
}

```

```

        .then(response => response.json())
        .then(responseJson => {
            return responseJson.results.books;
        })
        .catch(error => {
            console.error(error);
        });
    });
}

export default { pobierzKsiazki: pobierzKsiazki };

```

Zaimportujmy tę bibliotekę do naszego komponentu.

```
import NYT from "./NYT";
```

Teraz dodajmy metodę `_odswiezDane`, która zaimportuje dane z API.

```

_odswiezDane = () => {
    NYT.pobierzKsiazki().then(ksiazki => {
        this.setState({ dane: this._dodajKluczeDoKsiazek(ksiazki) });
    });
};

```

Na końcu musimy ustawić pustą tabelę jako stan początkowy, a metodę `_odswiezDane` umieścić w `componentDidMount`. Gdy to zrobisz, aplikacja wyświetli prawdziwe dane z listy bestsellerów „New York Timesa”! Cały kod znajduje się na listingu 4.17, a wygląd aktualnej aplikacji pokazano na rysunku 4.8.

*Listing 4.17. Bestsellery/ListaKsiazek.js*

```

import React, { Component } from "react";

import { StyleSheet, Text, View, Image, FlatList } from "react-native";

import Ksiazka from "./Ksiazka";
import NYT from "./NYT";

export default class ListaKsiazek extends Component {
    constructor(props) {
        super(props);
        this.state = { dane: [] };
    }

    componentDidMount() {
        this._odswiezDane();
    }

    _renderujElement = ({ item }) => {
        return (
            <Ksiazka
                okladkaURL={item.book_image}
                tytul={item.key}
                autor={item.author}
            />
        );
    };
}

```

```

    />
  );
};

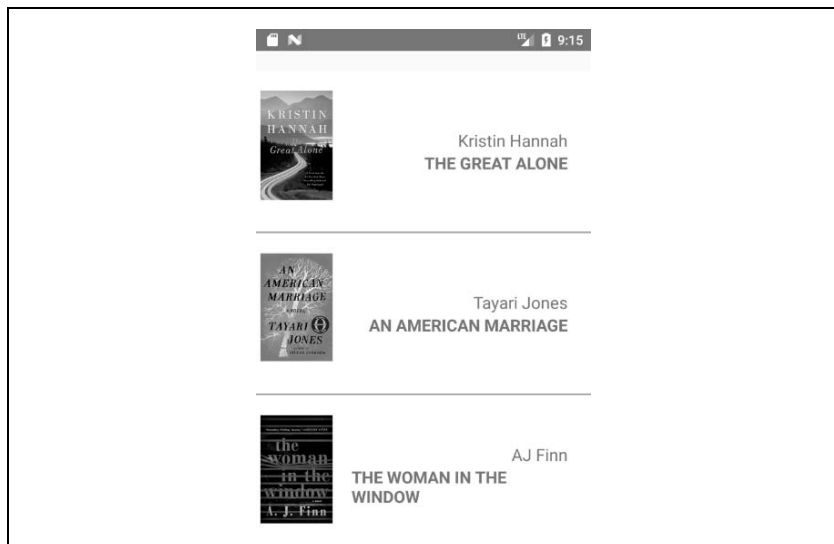
_dodajKluczeDoKsiazek = ksiazki => {
  // Odbiera odpowiedź z API NYTimes i dodaje klucz do obiektu do celów renderowania
  return ksiazki.map(ksiazka => {
    return Object.assign(ksiazka, { key: ksiazka.title });
  });
};

_odswiezDane = () => {
  NYT.pobierzKsiazki().then(ksiazki => {
    this.setState({ dane: this._dodajKluczeDoKsiazek(ksiazki) });
  });
};

render() {
  return (
    <View style={styles.kontener}>
      <FlatList data={this.state.dane} renderItem={this._renderujElement} />
    </View>
  );
}
}

```

```
const styles = StyleSheet.create({ kontener: { flex: 1, paddingTop: 22 } });
```



Rysunek 4.8. Przeglądanie bestsellerów w komponencie <FlatList>

Jak możesz zauważyć, praca z komponentem `<FlatList>` jest prosta, dopóki pamiętasz o odpowiednim ustrukturyzowaniu danych. Oprócz obsługi przewijania i interakcji dotykowych komponent ten posiada również szereg optymalizacji wydajności związanych z renderowaniem elementów.

## Zastosowanie komponentu `<SectionList>`

Komponent `<SectionList>` jest dedykowany dla zestawów danych, które w większości są jednorodne, jednak mogą mieć nagłówki sekcji. Jeżeli chciałbyś wyświetlać kilka różnych gatunków bestsellerów z nagłówkami rozdzielającymi je, `<SectionList>` to idealne rozwiązanie.

`<SectionList>` posiada atrybuty `sections`, `renderItem` i `renderSectionHeader`. Zaczniemy od `sections`, czyli tablicy, w której każdy element zawiera dane sekcji. Każda sekcja powinna posiadać atrybuty `title` (tytuł) i `data` (dane). Dane powinny być w tej samej formie jak dla komponentu `<FlatList>`: powinna to być tablica, w której każdy element będzie posiadał unikatowy klucz.

Zaktualizujmy metodę `_renderujDane`, aby pobierała zarówno książki z gatunku fikcji, jak i literatury faktu, oraz odpowiednio uzupełniała zawartość komponentu.

```
_odswiezDane = () => {
  Promise
    .all([
      NYT.pobierzKsiazki("hardcover-fiction"),
      NYT.pobierzKsiazki("hardcover-nonfiction")
    ])
    .then(results => {
      if (results.length !== 2) {
        console.error("Nieoczekiwane wyniki");
      }

      this.setState({
        sekcje: [
          {
            title: "Fikcja w Twardej Okładce",
            data: this._dodajKluczeDoKsiazek(results[0])
          },
          {
            title: "Literatura Faktu w Twardej Okładce",
            data: this._dodajKluczeDoKsiazek(results[1])
          }
        ]
      })
    })
}
```

```

    });
  });
};

```

Nie musimy zmieniać metody `_renderujElement`. Należy jednak dodać nową metodę `_renderujNaglowek`. Zróbmy to.

```

_renderujNaglowek = ({ section }) => {
  return (
    <Text style={styles.tekstNaglowka}>
      {section.title}
    </Text>
  );
};

```

Na koniec musimy w metodzie `render` zastąpić komponent `<FlatList>` komponentem `<SectionList>`.

```

render() {
  return (
    <View style={styles.kontener}>
      <SectionList
        sections={this.state.sekcje}
        renderItem={this._renderujElement}
        renderSectionHeader={this._renderujNaglowek}
      />
    </View>
  );
}

```

Jeżeli złączymy wszystko razem, nasz kod z zastosowaniem `<SectionList>` będzie wyglądał jak na listingu 4.18, co w efekcie powinno dać aplikację taką jak na rysunku 4.9.

*Listing 4.18. Bestseller/ListaKsiazekSekcje.js*

```

import React, { Component } from "react";
import { StyleSheet, Text, View, Image, SectionList } from "react-native";
import Ksiazka from "./Ksiazka";
import NYT from "./NYT";

export default class ListaKsiazekSekcje extends Component {
  constructor(props) {
    super(props);
    this.state = { sekcje: [] };
  }

  componentDidMount() {
    this._odswiezDane();
  }
}

```

```

    _dodajKluczeDoKsiazek = ksiazki => {
      // Odbiera odpowiedź z API NYTimes i dodaje klucz do obiektu do celów renderowania
      return ksiazki.map(ksiazka => {
        return Object.assign(ksiazka, { key: ksiazka.title });
      });
    };

    _odswiezDane = () => {
      Promise
        .all([
          NYT.pobierzKsiazki("hardcover-fiction"),
          NYT.pobierzKsiazki("hardcover-nonfiction")
        ])
        .then(results => {
          if (results.length !== 2) {
            console.error("Nieoczekiwane wyniki");
          }

          this.setState({
            sekcje: [
              {
                title: "Fikcja w Twardej Okładce",
                data: this._dodajKluczeDoKsiazek(results[0])
              },
              {
                title: "Literatura Faktu w Twardej Okładce",
                data: this._dodajKluczeDoKsiazek(results[1])
              }
            ]
          });
        });
    };

    _renderujElement = ({ item }) => {
      return (
        <Ksiazka
          okladkaURL={item.book_image}
          tytul={item.key}
          autor={item.author}
        />
      );
    };

    _renderujNaglowek = ({ section }) => {
      return (
        <Text style={styles.tekstNaglowka}>
          {section.title}
        </Text>
      );
    };

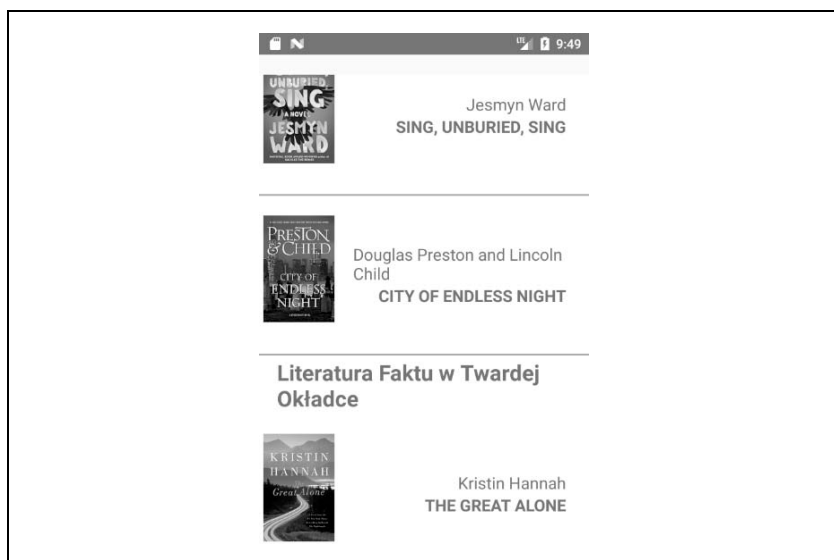
```

```

render() {
  return (
    <View style={styles.kontener}>
      <SectionList
        sections={this.state.sekcje}
        renderItem={this._renderujElement}
        renderSectionHeader={this._renderujNaglowek}
      />
    </View>
  );
}

const styles = StyleSheet.create({
  kontener: { flex: 1, paddingTop: 22 },
  tekstNaglowka: {
    fontSize: 24,
    alignSelf: "center",
    backgroundColor: "#FFF",
    fontWeight: "bold",
    paddingLeft: 20,
    paddingRight: 20,
    paddingTop: 2,
    paddingBottom: 2
  }
});

```



Rysunek 4.9. Przeglądanie aktualnych bestsellerów przy użyciu <SectionList>

# Nawigacja

Nawigacja w aplikacjach mobilnych odnosi się do kodu, który pozwala użytkownikom przechodzić między ekranami. W aplikacjach internetowych jest to część interfejsu `window.history`, który pozwala na przechodzenie „do przodu” i „wstecz”.

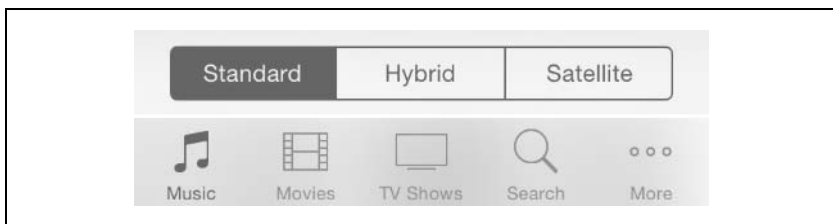
Powszechnie stosowane komponenty nawigacyjne React Native to wbudowane `<Navigator>` i `<NavigatorIOS>`, jak również rozwiązania stworzone przez społeczność użytkowników, np. `<StackNavigator>` (znajdujący się w bibliotece `react-navigation`).

Logika nawigacji jest niezbędna do przechodzenia między poszczególnymi widokami aplikacji. Umożliwia również przechodzenie z adresu URL prosto do konkretnego widoku aplikacji.

Nawigację omówimy szerzej w rozdziale 10.

## Inne komponenty organizacyjne

Istnieje wiele innych komponentów organizacyjnych. Kilka użytecznych to między innymi `<TabBarIOS>`, `<SegmentedControlIOS>` (pokazane na rysunku 4.10) oraz `<DrawerLayoutAndroid>` i `<ToolbarAndroid>` (pokazane na rysunku 4.11).

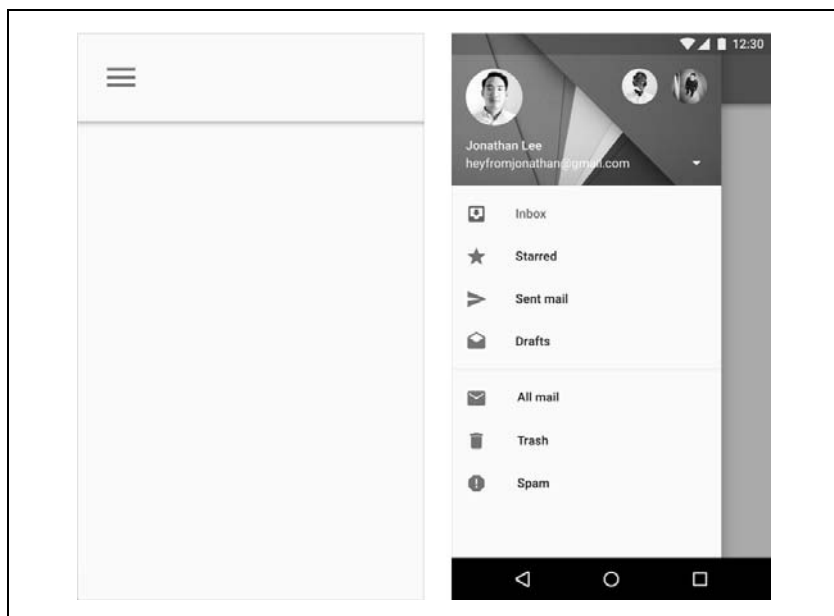


Rysunek 4.10. Komponenty `<SegmentedControlIOS>` (górną) i `<TabBarIOS>` (dół)

Zwróć uwagę, że mają one nazwy z końcówkami charakterystycznymi dla danego systemu. Jest tak, ponieważ korzystają one z natywnych API komponentów charakterystycznych dla danych platform.

Komponenty te są bardzo przydatne w organizowaniu wielu ekranów w jednej aplikacji. `<TabBarIOS>` i `<DrawerLayoutAndroid>` dają możliwość łatwego przełączania się między różnymi trybami i funkcjami. `<SegmentedControlIOS>` i `<ToolbarAndroid>` są bardziej adekwatne w przypadku drobnych kontrolek.





Rysunek 4.11. Komponenty <DrawerLayoutAndroid> (lewo) i <ToolbarAndroid> (pravo)

Jak najlepiej wykorzystać te komponenty, dowiesz się z wytycznych projektowych dla danej platformy:

- wytyczne projektowe dla systemów Android ([http://bit.ly/android\\_design\\_guide](http://bit.ly/android_design_guide)),
- wytyczne dla interfejsów człowiek – maszyna w systemie iOS ([http://bit.ly/designing\\_for\\_ios](http://bit.ly/designing_for_ios)).

Komponenty dedykowane dla danego systemu operacyjnego omówimy w rozdziale 7.

## Podsumowanie

W tym rozdziale poznaliśmy charakterystykę i różnorodność najważniejszych komponentów React Native. Wyjaśniono, jak używać podstawowych komponentów niskopoziomowych, takich jak <Text> i <View>, oraz komponentów wyższego poziomu, takich jak <ListView>, <Navigator> i <TabBarIOS>. Przyjrzeliliśmy się również zastosowaniu różnych API i komponentów obsługujących

dotyk, na wypadek gdybyś chciał stworzyć własną obsługę dotyku. Na końcu zobaczyles, jak wykorzystać w aplikacji komponenty dedykowane.

W tym momencie powinieneś mieć wystarczającą wiedzę, aby zbudować prostą i funkcjonalną aplikację przy użyciu React Native. Gdy znasz komponenty omówione w tym rozdziale, budowanie aplikacji przy ich użyciu powinno wydać się podobne do tworzenia aplikacji w sieci.

Oczywiście tworzenie podstawowych i funkcjonalnych aplikacji w React Native to tylko część drogi. W następnym rozdziale skupimy się na stylach i nadawaniu aplikacjom pożądanego przez nas wyglądu za pomocą implementacji stylów w React Native.

## A

akcja, 211, 212

animacja, 69

API

AVPlayer, 142

Fetch, 46

Geofencing, 114

MediaPlayer, 149

obietnic, 235

OpenWeatherMap, 39, 46, 116

sieciowe, 46

sprzętowe, 27

stylów, 92

systemowe, 26, 109

aplikacja

DevTools, 166

Expo, *Patrz:* Expo

mobilna, 13, 201

Pogodynka, 39

publikowanie, 237, 238, 239

RNTester, 23

stan, 210, 211

struktura, 35, 186

testowanie, 174, 238

tworzenie, 33, 183

treści, 184

uruchamianie, 31

Android, 35

iOS, 33

wersja produkcyjna, 238

widok, *Patrz:* widok

wieloplatformowa, 20, 183

z fiskalami, 183

AsyncStorage, 224

atrybut

alignItems, 102

flex, 100

flexDirection, 100, 104

navigation, 202

renderItem, 75, 84

renderSectionHeader, 84

resizeMode, 62

sections, 84

AVD, 173

## B

Babel, 231

biblioteka

AsyncStorage, 122

CameraRoll, 118

Flow, 177

instalowanie, 133, 135

Jest, 177, 178

loadsh, 134, 135

navigator.geolocation, 110

PanResponder, 73

RCTVideo, 142

React, 13

- React Navigation, 201, 202
- react-navigation, 88
- react-redux, 217
- Redux, 209, 210, 215
  - akcja, *Patrz*: akcja
  - AsyncStorage, 224, 225
  - instalowanie, 210
  - łączenie z aplikacją, 215, 217
  - łączenie z komponentem, 218
- StyleSheet, 38
- XHR, 121
- błąd
  - importowania komponentu, 168
  - niezadeklarowanej zmiennej, 169
- No visible interface for
  - RCTRootView, 172
- rozmiarów zasobów, 172
- składni, 168
- stylu, 169
- typowania, 177
- uruchamiania AVD, 173
- zależności Androida, 173

## C

- Chrome, 15
- Chrome Developer Tools, 161, 162
- Create React Native App, 29, 30
- CSS, 25, 91

## D

- dane
  - geolokalizacyjne, *Patrz*: geolokalizacja
  - pobieranie z sieci, 46
  - wyświetlanie, 43
- debuger, 161, 165, 166
- debugowanie, 161, 162, 165, 167
  - poza kodem JavaScript, 171
  - pułapka, 165
- destrukuryzacja, 232
- DOM, 19, 20, 135
- dotyk, 62, 73

## E

- ekran
  - rozdzielczość, 50, 61
  - śmierci czerwony, 168
- ES5, 231, 234
- ES6, 234
- Expo, 31, 162, 241

## F

- flexbox, 38, 98, 99
- funkcja
  - anonimowa, 234
  - AppRegistry, 37
  - deklaracja, 233
  - onMoveShouldSetPanResponder, 67
  - onPanResponderGrant, 67
  - onPanResponderRelease, 67
  - onPanResponderTerminate, 67
  - onStartShouldSetPanResponder, 67
  - render, 20
  - shuffle, 134
  - strzałkowa, 42, 233
  - StyleSheet.create, 93, 95
  - Stylesheet, 37

## G

- geolokalizacja, 110
  - pozycja, 114
  - testowanie na emulatorach, 112
  - uprawnienia, 111
- gest, 62, 73

## I

- interfejs
  - dotykowy, 63
  - mobilny, 57
  - typów, 177

interfejs  
użytkownika  
Android, 160  
iOS, 160  
wątek, *Patrz:* wątek interfejs  
użytkownika

## J

JavaScript, 231  
ECMAScript 5, *Patrz:* ES5  
język  
JavaScript, 13  
JSX, 13, 24  
LESS, *Patrz:* LESS  
SASS, *Patrz:* SASS  
XML, 13

## K

klasa  
PanResponder, 66, 73  
ReactVideoViewManager, 150  
ViewManager, 150  
klucz, 122  
NSLocationWhenInUseUsage  
→Description, 111  
kod  
dedykowany, 155, 160  
testowanie, 176  
kompilator  
Babel, *Patrz:* Babel  
kompilowanie, 238  
komponent  
dedykowany dla platformy, 155, 156, 160  
natywny, 148, 150  
wieloplatformowy, 151, 156  
Prognoza, 127  
React Native, *Patrz:* React Native  
komponent  
widoku, 143

konsola, 161, 162  
JavaScript, 110  
przeglądarki, 164  
Xcode, 163  
konstruktor, 147

## L

LESS, 91, 92  
lista, 73  
optymalizacja wydajności, 74

## M

menedżer pakietów Node.js, *Patrz:* npm  
metoda  
CameraRoll.getPhotos, 119  
createNativeModules, 148  
createStore, 215  
createViewManagers, 148  
getCurrentPosition, 110  
getItem, 122  
getName, 147, 151  
getPackages, 148  
getPhotos, 118  
setItem, 122  
setNativeProps, 69  
view, 144  
moduł  
natywny, 135, 137, 141, 142  
Android, 149, 150  
iOS, 137  
Java, 145

## N

narzędzia  
programisty, 161  
przeglądarkowe, 164, 167  
nawigator  
DrawerNavigator, 202  
TabNavigator, 202

Node.js menedżer pakietów, *Patrz:* npm  
npm, 30, 133, 171

## O

obiekt

- getPhotoParams, 119
- PanResponder, 66, 67
- position, 110
- stylu, 26, 38, 92, 93

obietnica, 46, 122, 235

obraz, 129

- atribut resizeMode, 62
- rolka kamery, 118
- renderowanie, 120

styl, *Patrz:* styl obrazu  
w tle, 50

wgranie na serwer, 121

## P

packager React Native, *Patrz:* React  
Native *packager*

plik

- .flowconfig, 177
- AndroidManifest.xml, 111, 112
- build.gradle, 135
- index.android.js, 152, 157
- index.ios.js, 152, 157
- Info.plist, 111
- ios/RCTVideo.h, 142
- MainActivity.java, 146
- MainApplication.java, 135
- nagłówkowy, 137
- package.json, 134
- RCTVideo.m, 143
- ReactVideoPackade.java, 149
- ReactVideoView.java, 149
- ReactVideoViewManager.java, 149
- rozszerzenie
  - .flowconfig, 177
  - .h, 138
  - .m, 137, 138

settings.gradle, 135

style.js, 96, 98

pole tekstowe, 40

polecenie

bind, 233

console.error, 165

console.log, 162, 165

console.warn, 165

logcat, 163

pozycjonowanie, 98

absolutne, 102, 103, 104

protokół RCTBridgeModule, 137, 142

przypisanie destrukuryzujące,  
*Patrz:* destrukuryzacja

## R

React Developer Tools, 166

React Native, 13

instalacja, 32

komponent, 22, 24, 193

Button, 63, 95, 128, 193

DatePickerIOS, 23

dla urządzeń mobilnych, 57

DrawerLayoutAndroid, 88

FlatList, 58, 74, 77, 81

hierarchia, 167

Image, 22, 58, 61, 62, 118

ImageBackground, 51

importowanie, 37

komunikacja, 209, 210

lista, 74

ListView, 22

Navigator, 88

NavigatorIOS, 88

Platform, 159

Prognoza, 43, 44

react-native-video, 135

SectionList, 74, 84

SegmentedControlIOS, 88, 89

StackNavigator, 88, 202, 205, 206

TabBarIOS, 88

- React Native
  - komponent
    - Text, 22, 41, 58, 59
    - TextInput, 41
    - ToolBarAndroid, 88, 89
    - TouchableHighlight, 63, 73
    - tworzenie, 22, 41
    - Video, 136
    - View, 22, 58, 168
    - VirtualizedList, 74
  - most, 20, 22
  - packager, 174, 175
  - środowisko programistyczne,  
*Patrz:* środowisko
- React Style, 91
- reduktor, 212
  - implementowanie, 213
  - sygnatura, 215
  - tworzenie, 212
- renderowanie, 21
  - interfejsu kompatybilnego ze  
stronami internetowymi, 58
  - natywne, 13

## S

- Safari, 15
- SASS, 91, 92
- składnia spread, 67
- społeczność użytkowników, 182
- styl, 59, 91, 197
  - błąd, 169
  - flexDirection, 51
  - jako atrybut, 97
  - lokalny, 26
  - łączenie, 94, 95
  - obrazu, 62
  - przeglądanie, 167
  - w kodzie widoku, 92
  - współdzielenie, 97

- system
  - Android, 35, 149, 150, 238
  - iOS, 33, 137
    - symulator, 112
  - Linux, 32
  - macOS, 32
  - symulator, 175
  - Windows, 32

## Ś

- środowisko
  - Android, 238
  - Android SDK, 33
  - Android Studio, 33
  - JDK, 33
  - konfiguracja, 29
  - Xcode, 33

## T

- tekst HTML, 58
  - React Native, 58, 60, *Patrz też:*  
React Native komponent Text
- test migawkowy, 178

## U

- układ flexbox, *Patrz:* flexbox

## W

- wątek interfejsu użytkownika, 14
- widok
  - nawigacja, 201, 203
  - wyszukiwanie, 202

## X

- Xcode
  - Issues, 172
  - konsola, *Patrz:* konsola Xcode

## Z

zależności, 171

    Androida, 173

zdarzenie, 67

zmienna tablicowa, 134

## Ż

żądanie XHR, 122



# PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



- 1. ZAREJESTRUJ SIĘ**
- 2. PREZENTUJ KSIĄŻKI**
- 3. ZBIERAJ PROWIZJĘ**

Zmień swoją stronę WWW  
w działający bankomat!

**Dowiedz się więcej i dołącz już dzisiaj!**

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA



**Helion SA**

# React Native: oto narzędzie do budowy kapitalnych aplikacji mobilnych!

React Native to framework JavaScript służący do budowania interfejsów użytkownika. Udostępniany przez Facebooka na licencji open source od samego początku cieszy się uznaniem programistów. React Native bowiem pozwala na proste tworzenie w pełni funkcjonalnych aplikacji mobilnych natywnie renderowanych na iOS i Androida. Umożliwia też bezproblemowe korzystanie z zasobów platform mobilnych, takich jak kamera, lokalizacja czy pamięć lokalna.

Jeśli znasz już podstawy biblioteki React i chcesz pisać natywne aplikacje na iOS i Androida, ta książka jest dla Ciebie. Szybko przypomnisz sobie zasady działania React Native, skonfigurujesz środowisko pracy i poznasz kolejne etapy tworzenia funkcjonalnej aplikacji mobilnej. Dzięki licznym przykładowym blokom kodu, krok po kroku nauczysz się tworzenia i nadawania stylów interfejsom graficznym, korzystania z komponentów mobilnych, a także debugowania i wdrażania aplikacji mobilnych. Dowiesz się też, jak rozszerzać możliwości React Native: używać zewnętrznych bibliotek, a nawet tworzyć własne biblioteki w językach Java i Objective-C.

## Bonnie Eisenman

– inżynier oprogramowania. Obecnie pracuje dla Twittera, a wcześniej zdobywała doświadczenie w Codecademy, Fog Creek Software i Google. Występuje na konferencjach, wygłaszając referaty o różnej tematyce: począwszy od ReactJS, poprzez programowanie instrumentów muzycznych, a na Arduino skończywszy.

W tej książce omówiono:

- tworzenie interfejsów dla komponentów natywnych
- opracowywanie własnych aplikacji i komponentów
- interfejsy API oraz moduły tworzone przez społeczność użytkowników React
- zarządzanie stanami w dużej aplikacji za pomocą biblioteki Redux

**Helion** 

 **helion.pl**

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

Sprawdź nasze szkolenia!

**SZKOLENIA**



**AKADEMIA IT & BUSINESS**

[WWW.SZKOLENIA.HELION.PL](http://WWW.SZKOLENIA.HELION.PL)

**KOD KORZYŚCI**  
Slegnij po więcej! ▶



ISBN 978-83-283-4424-2



9 788328 344242