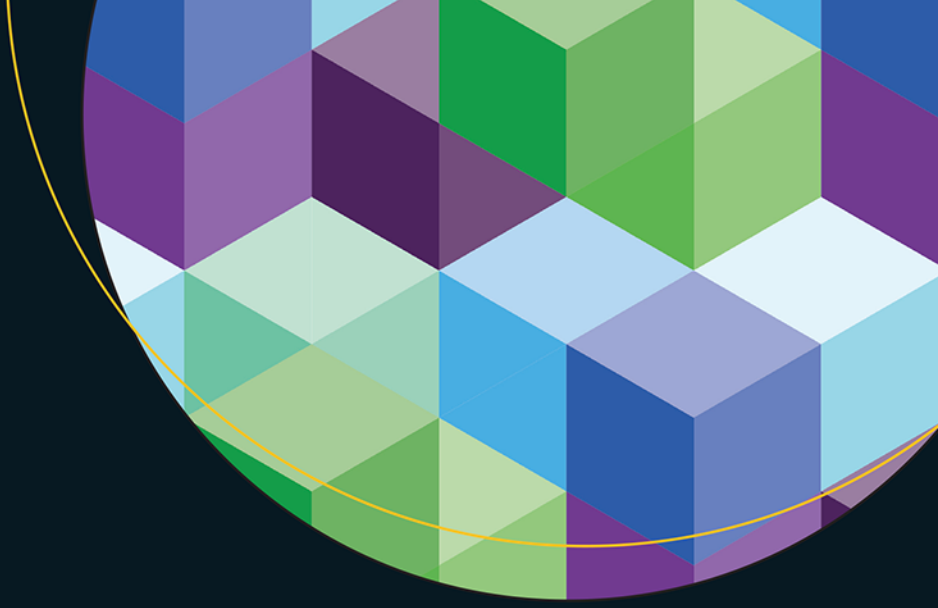


Angular

Profesjonalne techniki programowania

Wydanie II

—

Adam Freeman

Helion

Apress

Tytuł oryginału: Pro Angular, 2nd Edition

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-4231-6

Wydawnictwo HELION
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)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/angup2.xz

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/angup2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 7.

SportsStore — rzeczywista aplikacja

W rozdziale 2. utworzyliśmy niewielką i prostą aplikację Angular. Dzięki tego rodzaju przykładom mogliśmy skoncentrować się na konkretnych funkcjach Angular, choć były one pozbawione kontekstu. Teraz zbudujemy prostą, ale realistyczną aplikację typu e-commerce.

Budowana tutaj aplikacja o nazwie SportsStore będzie oparta na klasycznym podejściu stosowanym podczas tworzenia sklepów internetowych. Przygotujemy katalog produktów, które klienci będą mogli przeglądać według kategorii. Aplikacja będzie obsługiwała koszyk na zakupy, do którego klienci będą mogli dodawać produkty (lub usuwać je z niego). Oczywiście aplikacja będzie zawierać stronę pozwalającą klientom na finalizację zakupu i podanie danych potrzebnych do realizacji zamówienia. Opracujemy także obszar administracyjny, aby zapewnić sobie możliwość zarządzania katalogiem produktów (przeprowadzanie operacji typu CRUD). Wspomniany obszar będzie chroniony i tylko użytkownicy zalogowani jako administratorzy będą mogli wprowadzać zmiany. Na końcu pokażę, jak przygotować aplikację do wdrożenia, a następnie ją wdrożyć.

Celem przyświecającym mi w tym oraz kolejnych rozdziałach jest pokazanie na maksymalnie rzeczywistym przykładzie, jak faktycznie wygląda programowanie z wykorzystaniem Angular. Ponieważ koncentruję się na frameworku Angular, to uproszczona została integracja z zewnętrznymi systemami, takimi jak magazyn danych, a niektóre (na przykład przetwarzanie płatności) wręcz pominąłem.

Przykład aplikacji SportsStore wykorzystałem w kilku moich książkach, ponieważ pokazuje on, w jaki sposób można wykorzystać różne frameworki, języki i style programowania do osiągnięcia tego samego efektu. Nie musisz czytać innych moich książek, aby zrozumieć materiał przedstawiony w tym rozdziale. Jednak ich lektura może pokazać interesujące różnice w implementacji budowanej tutaj aplikacji.

Funkcje Angular wykorzystane w aplikacji SportsStore będą szczegółowo omówione w późniejszych rozdziałach książki. Zamiast powielać przedstawione treści, podam jedynie objaśnienia niezbędne do zrozumienia działania przykładowej aplikacji, a po szczegółowe informacje odeślę Cię do innych rozdziałów. Rozdziały, w których budujemy aplikację SportsStore, możesz czytać albo dokładnie od początku do końca, aby się dowiedzieć, jak działa Angular, albo też przechodzić od razu do wskazanych rozdziałów w celu uzyskania dokładniejszych informacji o danej funkcji. Niezależnie od przyjętego podejścia nie oczekuj, że wszystko od razu zrozumiesz. Angular to dość rozbudowany framework, a aplikacja SportsStore ma pokazać wiele jego możliwości, ale chwilowo bez zbytniego zagłębiania się w szczegóły, ponieważ będą one omówione w pozostałej części książki.

Utworzenie przykładowego projektu

W celu utworzenia projektu *SportsStore* przejdź do powłoki, wybierz katalog przeznaczony dla projektu, a następnie wydaj poniższe polecenie:

```
$ ng new SportsStore
```

Pakiet `angular-cli` wygeneruje nowy projekt aplikacji Angular wraz z plikami konfiguracyjnymi, miejscem zarezerwowanym na treść oraz z innymi narzędziami programistycznymi. Przygotowanie projektu wymaga nieco czasu, ponieważ w trakcie tego procesu pobierana jest i instalowana duża liczba pakietów npm.

Utworzenie struktury katalogów

Bardzo ważnym aspektem podczas przygotowywania każdej aplikacji Angular jest utworzenie struktury katalogów. Polecenie `ng new` powoduje wygenerowanie struktury projektu, w którym wszystkie pliki aplikacji zostają umieszczone w katalogu *src*, przy czym pliki Angular trafiają do katalogu *src/app*. W celu dodania dodatkowej struktury w projekcie utwórz kolejne katalogi wymienione w tabeli 7.1.

Tabela 7.1. *Dodatkowe katalogi dla aplikacji SportsStore*

Katalog	Opis
SportsStore/src/app/model	To jest katalog przeznaczony na kod modelu.
SportsStore/src/app/store	To jest katalog przeznaczony na kod podstawowej funkcjonalności sklepu internetowego.
SportsStore/src/app/admin	To jest katalog przeznaczony na kod funkcjonalności administracyjnej.

Instalowanie dodatkowych pakietów npm

Projekt *SportsStore* wymaga pewnych pakietów dodatkowych poza dodanymi standardowo przez `angular-cli`. Przeprowadź edycję pliku *package.json* w katalogu *SportsStore* i dodaj pakiety przedstawione na listingu 7.1.

Listing 7.1. *Zawartość pliku package.json w katalogu SportsStore*

```
{
  "name": "sports-store",
  "version": "0.0.0",
  "license": "MIT",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e",
    "json": "json-server data.js -p 3500 -m authMiddleware.js"
  },
  "private": true,
  "dependencies": {
    "@angular/common": "^4.0.0",
    "@angular/compiler": "^4.0.0",
    "@angular/core": "^4.0.0",
    "@angular/forms": "^4.0.0",
    "@angular/http": "^4.0.0",
    "@angular/platform-browser": "^4.0.0",
    "@angular/platform-browser-dynamic": "^4.0.0",
    "@angular/router": "^4.0.0",
    "core-js": "^2.4.1",
    "rxjs": "^5.1.0",
```

```
    "zone.js": "^0.8.4",
    "bootstrap": "4.0.0-alpha.4",
    "font-awesome": "4.7.0"
  },
  "devDependencies": {
    "@angular/cli": "1.0.0",
    "@angular/compiler-cli": "^4.0.0",
    "@types/jasmine": "2.5.38",
    "@types/node": "~6.0.60",
    "codelyzer": "~2.0.0",
    "jasmine-core": "~2.5.2",
    "jasmine-spec-reporter": "~3.2.0",
    "karma": "~1.4.1",
    "karma-chrome-launcher": "~2.0.0",
    "karma-cli": "~1.0.1",
    "karma-jasmine": "~1.1.0",
    "karma-jasmine-html-reporter": "^0.2.2",
    "karma-coverage-istanbul-reporter": "^0.2.0",
    "protractor": "~5.1.0",
    "ts-node": "~2.0.0",
    "tslint": "~4.5.0",
    "typescript": "~2.2.0",
    "json-server": "0.8.21",
    "jsonwebtoken": "7.1.9"
  }
}
```

■ **Ostrzeżenie** Aby otrzymać oczekiwane dane wyjściowe, we wszystkich przykładach przedstawionych w książce bardzo ważne jest użycie wersji pakietów podanych na listingach. Jeżeli napotkasz problemy z przykładami przedstawionymi w książce, spróbuj wykorzystać kod źródłowy przygotowanych przeze mnie aplikacji, który możesz pobrać ze strony *ftp://ftp.helion.pl/przyklady/angup2.xz*. Jeżeli dojdziesz do ściany i nie będziesz mógł sobie poradzić, napisz do mnie na adres *adam@adam-freeman.com*, a ja spróbuję Ci pomóc.

Aby pobrać i zainstalować pakiety wymagane podczas pracy nad tym projektem, zapisz plik, a następnie z poziomu katalogu *SportsStore* wydaj poniższe polecenie:

```
$ npm install
```

Menedżer npm wyświetli listę pakietów po zainstalowaniu wszystkich niezbędnych. W trakcie procesu instalacji zwykle pojawiają się pewne ostrzeżenia, które jednak można bezpiecznie zignorować.

Utworzenie usługi sieciowej typu RESTful

Aplikacja SportStore będzie używała asynchronicznych żądań HTTP w celu pobrania danych modelu dostarczanych przez usługę sieciową typu RESTful. REST, jak to dokładnie przedstawię w rozdziale 24., to po prostu podejście stosowane podczas projektowania usług sieciowych używających metod HTTP do określenia operacji oraz adresu URL do wskazania obiektów danych, w których będzie wykonana operacja.

W pliku *package.json* wymieniłem między innymi `json-server`, czyli doskonały pakiet pozwalający na szybkie utworzenie usługi sieciowej na podstawie danych JSON lub kodu JavaScript. Aby upewnić się o istnieniu pewnego stanu projektu, do którego zawsze będzie można powrócić, wykorzystam zaletę funkcji pozwalającej usłudze sieciowej typu RESTfull na dostarczanie danych za pomocą kodu JavaScript. Oznacza to, że ponowne uruchomienie usługi sieciowej wyzeruje dane aplikacji. W katalogu *SportStore* utwórz plik o nazwie *data.js* i umieść w nim kod przedstawiony na listingu 7.2.

Listing 7.2. *Zawartość pliku data.js w katalogu SportsStore*

```
module.exports = function () {
    return {
        products: [
            { id: 1, name: "Kajak", category: "Sporty wodne",
                description: "Łódka przeznaczona dla jednej osoby.", price: 275 },
            { id: 2, name: "Kamizelka ratunkowa", category: "Sporty wodne",
                description: "Chroni i dodaje uroku.", price: 48.95 },
            { id: 3, name: "Piłka", category: "Piłka nożna",
                description: "Zatwierdzone przez FIFA rozmiar i waga.", price: 19.50 },
            { id: 4, name: "Flagi narożne", category: "Piłka nożna",
                description: "Nadadzą twojemu boisku profesjonalny wygląd.",
                price: 34.95 },
            { id: 5, name: "Stadion", category: "Piłka nożna",
                description: "Składany stadion na 35 000 osób.", price: 79500 },
            { id: 6, name: "Czapka", category: "Szachy",
                description: "Zwiększa efektywność mózgu o 75%.", price: 16 },
            { id: 7, name: "Niestabilne krzesło", category: "Szachy",
                description: "Zmniejsza szanse przeciwnika.",
                price: 29.95 },
            { id: 8, name: "Ludzka szachownica", category: "Szachy",
                description: "Przyjemna gra dla całej rodziny.", price: 75 },
            { id: 9, name: "Błyszczący król", category: "Szachy",
                description: "Pokryty złotem i wysadzany diamentami król.", price: 1200 }
        ],
        orders: []
    }
}
```

■ **Wskazówka** Podczas tworzenia plików konfiguracyjnych trzeba koniecznie zwrócić uwagę na ich nazwy. Niektóre z nich mają rozszerzenie *.json*, czyli zawierają dane statyczne sformatowane w postaci JSON. Z kolei inne mają rozszerzenie *.js*, co oznacza, że zawierają kod JavaScript. Każde narzędzie niezbędne podczas programowania z użyciem frameworka Angular ma pewne wymagania dotyczące jego pliku konfiguracyjnego.

Powyższy fragment kodu definiuje dwie kolekcje danych dostarczane przez usługę sieciową typu RESTful. Pierwsza kolekcja, `products`, zawiera wszystkie produkty oferowane klientom w sklepie internetowym. Druga, `orders`, będzie zawierała zamówienia złożone przez klientów — ta kolekcja jest aktualnie pusta.

Dane przechowywane w usłudze sieciowej typu RESTful muszą być chronione, aby zwykły użytkownik nie mógł modyfikować katalogu produktów bądź stanu zamówień. Pakiet `json-server` nie oferuje żadnych wbudowanych funkcji uwierzytelniania, więc w katalogu *SportStore* utwórz plik o nazwie *authMiddleware.js* i umieść w nim kod przedstawiony na listingu 7.3.

Listing 7.3. *Zawartość pliku authMiddleware.js w katalogu SportsStore*

```
const jwt = require("jsonwebtoken");

const APP_SECRET = "appsekret";
const USERNAME = "admin";
const PASSWORD = "sekret";

module.exports = function (req, res, next) {
    if (req.url == "/login" && req.method == "POST") {
        if (req.body != null && req.body.name == USERNAME
                && req.body.password == PASSWORD) {
            let token = jwt.sign({ data: USERNAME, expiresIn: "1h" }, APP_SECRET);
```

```
            res.json({ success: true, token: token });
        } else {
            res.json({ success: false });
        }
        res.end();
        return;
    } else if ((req.url.startsWith("/products") && req.method != "GET")
                || (req.url.startsWith("/orders") && req.method != "POST")) {
         let token = req.headers["authorization"];
         if (token != null && token.startsWith("Bearer<")) {
            token = token.substring(7, token.length - 1);
            try {
                jwt.verify(token, APP_SECRET);
                next();
                return;
            } catch (err) {}
         }
        res.statusCode = 401;
        res.end();
        return;
    }
    next();
}
```

Ten kod analizuje żądanie HTTP wykonane do usługi sieciowej typu RESTful i implementuje pewne podstawowe funkcje bezpieczeństwa. Jest to działający po stronie serwera kod niepowiązany bezpośrednio z programowaniem z użyciem frameworka Angular. Dlatego też nie przejmuj się, jeśli jego przeznaczenie pozostaje dla Ciebie niezrozumiałe. Proces uwierzytelniania i autoryzacji omówię dokładnie w rozdziale 9., w którym dowiesz się między innymi, jak uwierzytelniać użytkowników w aplikacji Angular.

■ **Ostrzeżenie** Kodu przedstawionego na listingu 7.3 nie używaj w aplikacji innej niż SportsStore. Zawiera on słabe hasła na stałe zdefiniowane w kodzie. Takie rozwiązanie jest wystarczające w projekcie aplikacji SportsStore, ponieważ tutaj nacisk położyłem na użycie frameworka Angular podczas programowania po stronie klienta. Przedstawione podejście jest jednak niedopuszczalne w rzeczywistych projektach.

Utworzenie pliku HTML

Każda aplikacja internetowa Angular ma dokument HTML wczytywany przez przeglądarkę WWW i używany do uruchomienia danej aplikacji. W katalogu *SportsStore/src* przeprowadź edycję pliku o nazwie *index.html* w celu usunięcia istniejącego w nim kodu i umieszczenia kodu przedstawionego na listingu 7.4.

Listing 7.4. *Zawartość pliku index.html w katalogu SportsStore/src*

```
<!DOCTYPE html>
<html lang="pl">
<head>
    <base href="/" />
    <title>SportsStore</title>
    <meta charset="utf-8" />
    <base href="/">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link href="node_modules/bootstrap/dist/css/bootstrap.min.css"
        rel="stylesheet" />
</head>
<body class="m-a-1">
```

```
    <app>Miejsce na treść sklepu SportsStore.</app>
</body>
</html>
```

Ten dokument HTML zawiera element `<link>` wczytujący arkusze stylów Bootstrap CSS oraz element `<app>` działający w charakterze miejsca zarezerwowanego na funkcjonalność sklepu internetowego SportsStore.

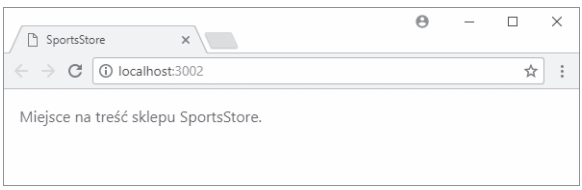
■ **Wskazówka** Przedstawiony dokument zawiera również element `<base>` wymagany przez funkcje routingu frameworka Angular, które dodam do projektu SportsStore w rozdziale 8.

Uruchomienie przykładowej aplikacji

Upewnij się, że zapisałeś wszystkie pliki, a następnie z poziomu katalogu *SportsStore* wydaj poniższe polecenie:

```
$ ng serve --port 3000 --open
```

To polecenie uruchomi łańcuch skonfigurowanych przez `angular-cli` narzędzi programistycznych, które przeprowadzą automatyczną kompilację kodu i plików z treścią umieszczonych w katalogu *src*. Kompilacja będzie się odbywała po wykryciu zmiany w dowolnym z wymienionych plików. Na ekranie zostanie otworzone nowe okno przeglądarki WWW wraz z treścią, jak pokazałem na rysunku 7.1.



Rysunek 7.1. *Efekt uruchomienia przykładowej aplikacji*

Programistyczny serwer HTTP będzie nasłuchiwał na porcie 3000, więc adres URL aplikacji ma postać *http://localhost:3000*. Nie ma potrzeby podawania nazwy dokumentu HTML, ponieważ *index.html* to nazwa domyślna, na którą odpowiada serwer.

Uruchomienie usługi sieciowej typu RESTful

W celu uruchomienia usługi sieciowej typu RESTful przejdź do nowego okna powłoki, następnie do katalogu *SportsStore* i wydaj poniższe polecenie:

```
$ npm run json
```

Usługa sieciowa typu RESTful została skonfigurowana do działania na porcie 3500. Aby przetestować żądanie do usługi sieciowej, w przeglądarce WWW wpisz adres URL *http://localhost:3500/products/1*. Przeglądarka WWW powinna wyświetlić informacje dotyczące jednego z produktów zdefiniowanych na listingu 7.2, jak pokazałem poniżej.

```
{
  "id": 1,
```

```
  "name": "Kajak",
  "category": "Sporty wodne",
  "description": "Łódka przeznaczona dla jednej osoby.",
  "price": 275
}
```

Przygotowanie funkcji projektu Angular

Każdy projekt Angular wymaga pewnych przygotowań w celu osiągnięcia stanu, w którym aplikacja może być wczytana i uruchomiona przez przeglądarkę WWW. W kolejnych sekcjach zaprezentuję podstawy, na których zostanie utworzona aplikacja SportsStore.

Uaktualnienie komponentu głównego

Pracę rozpoczynamy od komponentu głównego, który jest elementem konstrukcyjnym Angular przeznaczonym do zarządzania treścią elementu `<app>` zdefiniowanego w dokumencie HTML. Wprawdzie aplikacja może zawierać wiele komponentów, ale zawsze będzie komponent główny odpowiedzialny za treść najwyższego poziomu wyświetlaną użytkownikowi.

W katalogu *SportsStore/src/app* przeprowadź edycję pliku o nazwie *app.component.ts* i istniejący w nim kod zastąp przedstawionym na listingu 7.5.

Listing 7.5. *Zawartość pliku app.component.ts w katalogu SportsStore/src/app*

```
import { Component } from "@angular/core";

@Component({
    selector: "app",
    template: `<div class="bg-success p-a-1 text-xs-center">
                    To jest aplikacja SportsStore.
                </div>`
})
export class AppComponent {}
```

Dekorator `@Component` informuje Angular, że klasa `AppComponent` jest komponentem, a jej właściwości konfigurują sposób zastosowania tego komponentu. W rozdziale 17. dokładnie omówię pełny zbiór właściwości komponentu, trzy z nich użyte na listingu 7.9 to najprostsze i najczęściej stosowane. Właściwość `selector` wskazuje frameworkowi Angular, jak komponent ma być zastosowany w dokumencie HTML, natomiast właściwość `template` definiuje treść wyświetlaną przez ten komponent. Podobnie jak w omawianym tutaj przykładzie, komponent ma możliwość zdefiniowania osadzonego szablonu bądź też korzystania z zewnętrznych plików HTML, które mogą ułatwić zarządzanie skomplikowaną treścią.

Klasa `AppComponent` nie zawiera kodu, ponieważ komponent główny w projekcie Angular istnieje po prostu w celu zarządzania treścią wyświetlaną użytkownikowi. Początkowo będę ręcznie zarządzać treścią wyświetlaną przez komponent główny, natomiast w rozdziale 8. funkcji o nazwie **routing URL** użyję do automatycznego zarządzania treścią na podstawie akcji podejmowanych przez użytkownika.

Uaktualnienie modułu głównego

Mamy dwa rodzaje modułów Angular: funkcjonalny i główny. Moduł funkcjonalny jest używany w celu grupowania powiązanej funkcjonalności aplikacji, aby ułatwić zarządzanie nią. Tego rodzaju moduły utworzę w każdym z najważniejszych obszarów aplikacji, między innymi w modelu, w interfejsie sklepu wyświetlanym użytkownikom oraz w interfejsie administracyjnym.

Moduł główny jest używany do opisania aplikacji frameworkowi Angular. Wspomniany opis zawiera nazwy modułów zwykłych wymaganych do uruchomienia aplikacji, funkcji koniecznych do wczytania oraz nazwę komponentu głównego. Zgodnie z konwencją nazwa pliku komponentu głównego to *app.module.ts*. W katalogu *SportsStore/src/app* przeprowadź edycję pliku o nazwie *app.module.ts* i istniejący w nim kod zastąp przedstawionym na listingu 7.6.

Listing 7.6. *Zawartość pliku app.module.ts w katalogu SportsStore/app*

```
import { NgModule } from "@angular/core";
import { LOCALE_ID } from '@angular/core';
import { BrowserModule } from "@angular/platform-browser";
import { AppComponent } from "./app.component";

@NgModule({
    imports: [BrowserModule],
    declarations: [AppComponent],
    bootstrap: [AppComponent],
    providers: [{ provide: LOCALE_ID, useValue: "pl-PL" }]
})
export class AppModule {}
```

Podobnie jak w przypadku komponentu głównego, także klasa modułu głównego nie zawiera kodu. Ten moduł tak naprawdę istnieje jedynie w celu dostarczenia informacji za pomocą dekoratora `@NgModule`. Właściwość `imports` nakazuje frameworkowi Angular wczytanie funkcji `BrowserModule`, która zawiera całą podstawową funkcjonalność wymaganą przez aplikację internetową.

Właściwość `declarations` wskazuje Angular, że powinien być wczytany komponent główny, z kolei właściwość `bootstrap` wskazuje ten komponent główny — to jest klasa `AppModule`. Kolejne informacje do tych właściwości dekoratora będę dodawał później wraz z implementowaniem poszczególnych funkcjonalności aplikacji SportsStore. Ta podstawowa konfiguracja wystarcza do uruchomienia aplikacji.

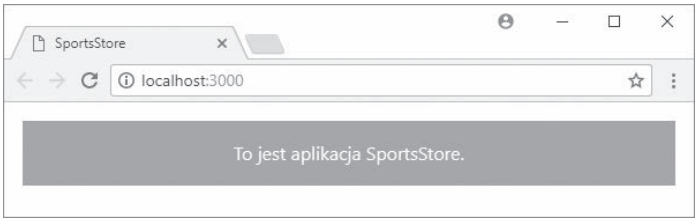
Analiza pliku typu bootstrap

Kolejnym krokiem jest utworzenie pliku typu bootstrap przeznaczonego do uruchamiania aplikacji. W tej książce koncentruję się na użyciu frameworka Angular do tworzenia aplikacji internetowych działających w przeglądarce WWW, ale platforma Angular może być wykorzystana do budowy aplikacji przeznaczonych również dla innych środowisk. Plik typu bootstrap używa platformy przeglądarki WWW w celu wczytania modułu głównego i uruchomienia aplikacji. Nie ma konieczności wprowadzania jakichkolwiek zmian w pliku o nazwie *main.ts* znajdującym się w katalogu *SportsStore/src*. Zawartość wymienionego pliku przedstawiłem na listingu 7.7.

Listing 7.7. *Zawartość pliku main.ts w katalogu SportsStore/src*

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Narzędzia programistyczne wykrywają wprowadzanie zmian w plikach projektu, co prowadzi do automatycznej kompilacji kodu, odświeżenia strony w przeglądarce WWW i wyświetlenia treści pokazanej na rysunku 7.2.



Rysunek 7.2. *Uruchomiona aplikacja SportsStore*

Jeżeli przeanalizujesz wygenerowany przez przeglądarkę WWW model DOM, wówczas zobaczysz, że treść pochodzącą z szablonu komponentu głównego framework Angular umieścił w elemencie `<app>`, jak pokazałem poniżej.

```
<body class="m-a-1">
    <app>
        <div class="bg-success p-a-1 text-xs-center">
            To jest aplikacja SportsStore.
        </div>
    </app>
</body>
```

Utworzenie danych modelu

Najlepszym miejscem do rozpoczęcia pracy nad nowym projektem jest jego model dostarczający dane. Chciałbym dotrzeć do punktu, w którym będziesz mógł zobaczyć w działaniu pewne funkcje frameworka Angular. Dlatego też zamiast definiować dane modelu od początku do końca, wykorzystam istniejącą funkcjonalność w celu dostarczenia fikcyjnych danych. Następnie będę używać tych danych do utworzenia funkcji przeznaczonych dla użytkownika. Do danych modelu jeszcze powrócę w rozdziale 8., w którym powiążę je z usługą sieciową typu RESTful.

Utworzenie klas modelu

Wszystkie dane modelu wymagają klas opisujących rodzaj danych, które będą znajdowały się w tym modelu. W przypadku aplikacji SportsStore oznacza to klasy opisujące produkty sprzedawane w sklepie oraz zamówienia składane przez klientów.

Na początku pracy z aplikacją SportsStore możliwość opisania produktów będzie wystarczająca. Kolejne klasy modelu przeznaczone do obsługi funkcjonalności sklepu będę tworzyć podczas implementacji wspomnianych funkcji. W katalogu *SportsStore/src/app/model* utwórz nowy plik o nazwie *product.model.ts* i umieść w nim kod przedstawiony na listingu 7.8.

Listing 7.8. *Zawartość pliku product.model.ts w katalogu SportsStore/src/app/model*

```
export class Product {
    constructor(
        public id?: number,
        public name?: string,
        public category?: string,
        public description?: string,
        public price?: number) {}
}
```

Klasa `Product` definiuje konstruktora akceptującego właściwości `id`, `name`, `category`, `description` i `price`, odpowiadające strukturze danych użytej podczas tworzenia usługi sieciowej typu RESTful we wcześniejszej części rozdziału. Znak zapytania umieszczony po nazwie parametru oznacza, że jest to parametr opcjonalny, który może być pominięty podczas tworzenia nowego egzemplarza obiektu na podstawie klasy `Product`. To jest użyteczna możliwość w trakcie budowy aplikacji, w której właściwość obiektu modelu ma przypisywaną wartość na podstawie danych pochodzących z formularza HTML.

Utworzenie fikcyjnego źródła danych

Aby przygotować się na przejście od danych fikcyjnych do rzeczywistych, zamierzam wykorzystać źródło danych. Pozostała część aplikacji nie wie, skąd pochodzą dane. Takie rozwiązanie umożliwi bezproblemowe przejście do pobierania danych za pomocą żądań HTTP.

W katalogu *SportsStore/src/app/model* utwórz nowy plik o nazwie *static.datasource.ts* i umieść w nim kod przedstawiony na listingu 7.9.

Listing 7.9. *Zawartość pliku static.datasource.ts w katalogu SportsStore/src/app/model*

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";
import { Observable } from "rxjs/Observable";
import "rxjs/add/observable/from";

@Injectable()
export class StaticDataSource {
    private products: Product[] = [
        new Product(1, "Produkt 1", "Kategoria  1", "Produkt 1 (Kategoria  1)", 100),
        new Product(2, "Produkt 2", "Kategoria  1", "Produkt 2 (Kategoria  1)", 100),
        new Product(3, "Produkt 3", "Kategoria  1", "Produkt 3 (Kategoria  1)", 100),
        new Product(4, "Produkt 4", "Kategoria  1", "Produkt 4 (Kategoria  1)", 100),
        new Product(5, "Produkt 5", "Kategoria  1", "Produkt 5 (Kategoria  1)", 100),
        new Product(6, "Produkt 6", "Kategoria  2", "Produkt 6 (Kategoria  2)", 100),
        new Product(7, "Produkt 7", "Kategoria  2", "Produkt 7 (Kategoria  2)", 100),
        new Product(8, "Produkt 8", "Kategoria  2", "Produkt 8 (Kategoria  2)", 100),
        new Product(9, "Produkt 9", "Kategoria  2", "Produkt 9 (Kategoria  2)", 100),
        new Product(10, "Produkt 10", "Kategoria  2", "Produkt 10 (Kategoria  2)", 100),
        new Product(11, "Produkt 11", "Kategoria  3", "Produkt 11 (Kategoria  3)", 100),
        new Product(12, "Produkt 12", "Kategoria  3", "Produkt 12 (Kategoria  3)", 100),
        new Product(13, "Produkt 13", "Kategoria  3", "Produkt 13 (Kategoria  3)", 100),
        new Product(14, "Produkt 14", "Kategoria  3", "Produkt 14 (Kategoria  3)", 100),
        new Product(15, "Produkt 15", "Kategoria  3", "Produkt 15 (Kategoria  3)", 100),
    ];

    getProducts(): Observable<Product[]> {
        return Observable.from([this.products]);
    }
}
```

Klasa `StaticDataSource` definiuje metodę o nazwie `getProducts()` zwracającą fikcyjne dane. Wynikiem wywołania tej metody jest `Observable<Product[]>`, czyli egzemplarz typu `Observable` tworzący tablicę obiektów `Product`.

Klasa `Observable` jest dostarczana przez pakiet biblioteki Reactive Extensions używanej przez framework Angular do obsługi zmiany stanu w aplikacji. W rozdziale 23. dokładnie omówię klasę `Observable`, natomiast w tym miejscu wystarczy wiedzieć, że tego rodzaju obiekt przypomina obietnicę JavaScript — reprezentuje zadanie asynchroniczne, które w przyszłości wygeneruje pewien wynik. Framework Angular wykorzystuje obiekty typu `Observable` dla niektórych funkcji, między innymi do wykonywania żądań HTTP, i dlatego metoda `getProducts()` zwraca `Observable<Product[]>`, zamiast po prostu asynchronicznie dostarczać dane lub wykorzystać obietnicę.

Dla klasy `StaticDataSource` został zastosowany dekorator `@Injectable`. Ten dekorator wskazuje frameworkowi Angular, że oznaczona nim klasa będzie używana w charakterze usługi. Dlatego też inne klasy będą mogły za pomocą mechanizmu wstrzykiwania zależności uzyskać dostęp do funkcjonalności oferowanej przez usługę. Mechanizm wstrzykiwania zależności dokładnie omówię w rozdziałach 19. i 20. Natomiast sposób działania usługi stanie się jasny, gdy aplikacja nabierze kształtu.

■ **Wskazówka** Zwróć uwagę na konieczność zaimportowania `Injectable` z modułu JavaScript `@angular/core`, aby można było zastosować dekorator `@Injectable`. Nie będę tutaj omawiał wszystkich klas Angular koniecznych do zaimportowania w budowanej aplikacji SportsStore. Więcej informacji na ich temat znajdziesz w rozdziałach, w których będę przedstawiał powiązane z nimi funkcje.

Utworzenie repozytorium modelu

Źródło danych jest odpowiedzialne za dostarczanie aplikacji wymaganych przez nią danych. Jednak dostęp do tych danych zwykle odbywa się za pomocą tak zwanego **repozytorium**, które staje się odpowiedzialne za dostarczanie danych poszczególnym komponentom aplikacji, nie ujawniając przy tym szczegółów związanych ze sposobem uzyskania danych. W katalogu *SportsStore/src/app/model* utwórz nowy plik o nazwie *product.repository.ts* i umieść w nim kod przedstawiony na listingu 7.10.

Listing 7.10. *Zawartość pliku product.repository.ts w katalogu SportsStore/src/app/model*

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";
import { StaticDataSource } from "./static.datasource";

@Injectable()
export class ProductRepository {
    private products: Product[] = [];
    private categories: string[] = [];

    constructor(private dataSource: StaticDataSource) {
        dataSource.getProducts().subscribe(data => {
            this.products = data;
            this.categories = data.map(p => p.category)
                .filter((c, index, array) => array.indexOf(c) == index).sort();
        });
    }

    getProducts(category: string = null): Product[] {
        return this.products
            .filter(p => category == null || category == p.category);
    }

    getProduct(id: number): Product {
        return this.products.find(p => p.id == id);
    }

    getCategories(): string[] {
        return this.categories;
    }
}
```

Gdy framework Angular będzie musiał utworzyć nowy egzemplarz repozytorium, przeanalizuje tę klasę i ustali, że w celu wywołania konstruktora `ProductRepository` i utworzenia nowego obiektu potrzebny jest obiekt `StaticDataSource`.

Konstruktor repozytorium wywołuje metodę `getProducts()` źródła danych, a następnie w celu otrzymania danych produktu używa metody `subscribe()` oferowanej przez zwrócony obiekt `Observable`. W rozdziale 23. dokładnie wyjaśnię sposób działania obiektu `Observable`.

Używanie prostych struktur danych

Do przechowywania danych modelu wykorzystałem tablicę, ponieważ zgodnie z ogólną regułą najprostsza z możliwych struktur danych daje najlepsze wyniki w aplikacji Angular. Podczas generowania treści w elemencie HTML framework Angular wykonuje wyrażenie w procesie dołączania danych. Oznacza to, że bardziej skomplikowane struktury — takie jak oparta na klasie `Map` i dostarczająca kolekcji typu klucz-wartość w JavaScript ES6 — muszą nieustannie przeprowadzać konwersje aż do chwili ustabilizowania się stanu aplikacji Angular. Dlatego też im prostsza struktura danych, tym mniejsza ilość pracy jest konieczna do wykonania, aby dostarczyć frameworkowi Angular wymagane przez niego dane.

Kolejnym powodem użycia prostych struktur danych są ograniczone możliwości w zakresie obsługi nowych funkcji JavaScript w starszych wersjach przeglądarek WWW. Na przykład w przypadku klasy `Map`, gdy kompilator jest używany do wygenerowania kodu JavaScript przeznaczonego do wykonania w starszych przeglądarkach WWW, TypeScript ogranicza sposób, w jaki treść map może być wykorzystana.

W efekcie staram się korzystać z prostych struktur danych, zwłaszcza tablic, i tworzyć nieco bardziej skomplikowane klasy przeznaczone do zarządzania danymi w tablicy. Przykład tego rodzaju podejścia zobaczysz, gdy w rozdziale 9 zacznę dodawać do klasy repozytorium produktu funkcje administracyjne. Wspomniane nowe funkcje będą musiały przeszukiwać tablicę, aby znaleźć obiekty, w których zostanie wykonana dana operacja. Takie rozwiązanie jest nieefektywne, ale te operacje są przeprowadzane znacznie rzadziej w porównaniu z częstotliwością, z jaką Angular wykonuje wyrażenia podczas procesu dołączania danych.

Utworzenie modułu funkcjonalnego

Przystępuję teraz do zdefiniowania modelu pozwalającego na łatwe wykorzystanie jego funkcjonalności w każdym miejscu aplikacji. W katalogu *SportsStore/src/app/model* utwórz nowy plik o nazwie *model.module.ts* i umieść w nim kod przedstawiony na listingu 7.11.

Listing 7.11. *Zawartość pliku model.module.ts w katalogu SportsStore/src/app/model*

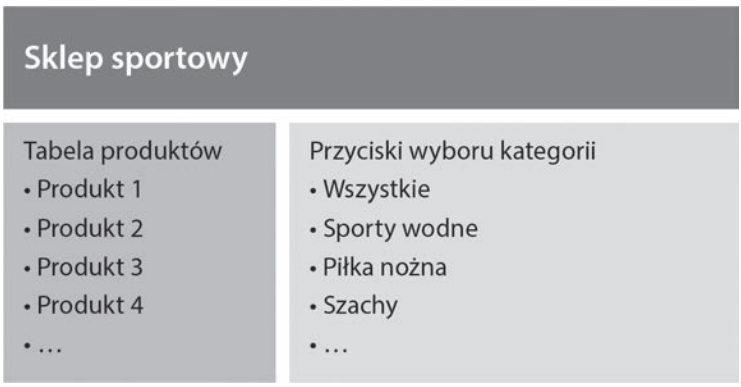
```
import { NgModule } from "@angular/core";
import { ProductRepository } from "./product.repository";
import { StaticDataSource } from "./static.datasource";

@NgModule({
    providers: [ProductRepository, StaticDataSource]
})
export class ModelModule {}
```

■ **Wskazówka** Nie przejmuj się, jeśli te wszystkie nazwy plików wydają Ci się podobne i mylące. Wraz z poznawaniem materiału przedstawionego w następnych rozdziałach książki przywykniesz do struktury aplikacji Angular. Wkrótce tylko spojrzysz na pliki znajdujące się w projekcie aplikacji Angular i od razu będziesz wiedział, do czego są przeznaczone.

Dekorator `@NgModule` jest używany do tworzenia zwykłych modułów, a jego właściwości wskazują frameworkowi Angular, jak moduł powinien być użyty. W przedstawionym powyżej module istnieje tylko jedna właściwość, `providers`, wskazująca Angular, które klasy powinny być używane w charakterze usług dla mechanizmu wstrzykiwania zależności, co zostanie dokładnie omówione w rozdziałach 19. i 20. Z kolei w rozdziale 21. omówię zwykłe moduły i dekorator `@NgModule`.

Rozpoczęcie pracy nad utworzeniem sklepu internetowego

Mając przygotowane dane modelu, można przystąpić do budowy funkcjonalności sklepu internetowego, która pozwoli użytkownikom na przeglądanie katalogu produktów oraz składanie zamówień. Podstawowa struktura sklepu internetowego to składający się z dwóch kolumn układ strony. Wyświetlane w lewej kolumnie przyciski kategorii pozwalają na filtrowanie listy produktów wyświetlanej w tabeli znajdującej się w prawej kolumnie, jak pokazałem na rysunku 7.3.



Rysunek 7.3. *Ogólna struktura strony w naszym sklepie internetowym*

W kolejnych sekcjach wykorzystam funkcje frameworka Angular i dane modelu w celu utworzenia układu pokazanego na rysunku 7.3.

Utworzenie szablonu i komponentu sklepu internetowego

Gdy lepiej poznasz framework Angular, zobaczysz, że można łączyć jego funkcje i ten sam problem rozwiązywać na wiele różnych sposobów. W celu zaprezentowania ważnych funkcji Angular spróbuję wprowadzić w projekcie SportsStore pewną odmienność. Jednak w tym momencie stawiam na prostotę, tak aby jak najszybciej dotrzeć do punktu pozwalającego na uruchomienie projektu.

Punktem wyjścia do opracowania funkcjonalności sklepu internetowego będzie nowy komponent. Ten komponent to po prostu klasa dostarczająca dane i logikę szablonowi HTML, który wykorzystuje mechanizm dołączania danych w celu dynamicznego generowania treści. W katalogu *SportsStore/src/app/store* utwórz nowy plik o nazwie *store.component.ts* i umieść w nim kod przedstawiony na listingu 7.12.

Listing 7.12. *Zawartość pliku store.component.ts w katalogu SportsStore/src/app/store*

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";

@Component({
```

```
    selector: "store",
    moduleId: module.id,
    templateUrl: "store.component.html"
})
export class StoreComponent {
    constructor(private repository: ProductRepository) {}

    get products(): Product[] {
        return this.repository.getProducts();
    }

    get categories(): string[] {
        return this.repository.getCategories();
    }
}
```

Dla klasy `StoreComponent` został zastosowany dekorator `@Component` wskazujący Angular, że mamy do czynienia z komponentem. Właściwości dekoratora określają frameworkowi Angular, jak zastosować ten komponent dla treści HTML (za pomocą elementu o nazwie `<store>`) i gdzie znajduje się szablon komponentu (w pliku o nazwie *store.component.html*).

Klasa `StoreComponent` dostarcza logikę przeznaczoną do obsługi treści umieszczonej w szablonie.

Konstruktor klasy otrzymuje argument w postaci obiektu `ProductRepository`, dostarczony za pomocą mechanizmu wstrzykiwania zależności, który dokładnie omówię w rozdziałach 19. i 20. W komponencie zostały zdefiniowane właściwości `products` i `categories` używane do wygenerowania w szablonie treści HTML na podstawie danych pochodzących z repozytorium.

Aby przygotować szablon dla budowanego tutaj komponentu, w katalogu *SportsStore/src/app/store* utwórz nowy plik o nazwie *store.component.html* i umieść w nim kod przedstawiony na listingu 7.13.

Listing 7.13. *Zawartość pliku store.component.html w katalogu SportsStore/src/app/store*

```
<div class="navbar navbar-inverse bg-inverse">
    <a class="navbar-brand">SKLEP SPORTOWY</a>
</div>
<div class="col-xs-3 bg-info p-a-1">
    {{categories.length}} kategorie
</div>
<div class="col-xs-9 bg-success p-a-1">
    {{products.length}} produktów
</div>
```

Ten szablon jest bardzo prosty i ma jedynie pozwolić na rozpoczęcie pracy. Większość elementów dostarcza strukturę dla układu sklepu internetowego i stosuje pewne style Bootstrap CSS. W tym momencie mamy jedynie dwie operacje dołączania danych wskazywane przez nawiasy {{ i }}. To są wiązania **interpolacji ciągu tekstowego**, które nakazują frameworkowi wykonanie wyrażenia wiązania i wstawienia jego wyniku do elementu. Wyrażenia w użytych wiązaniach powodują wyświetlenie liczby kategorii i produktów.

Utworzenie modułu funkcjonalnego dla sklepu

W tym momencie nie mamy jeszcze zbyt dużej funkcjonalności sklepu internetowego. Konieczne jest wykonanie jeszcze dodatkowej pracy i powiązania dostępnej funkcjonalności z pozostałą częścią aplikacji. Aby utworzyć moduł funkcjonalny dla sklepu internetowego, w katalogu *SportsStore/src/app/store* utwórz nowy plik o nazwie *store.module.ts* i umieść w nim kod przedstawiony na listingu 7.14.

Listing 7.14. *Zawartość pliku store.module.ts w katalogu SportsStore/src/app/store*

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
```

```
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { StoreComponent } from "./store.component";

@NgModule({
    imports: [ModelModule, BrowserModule, FormsModule],
    declarations: [StoreComponent],
    exports: [StoreComponent]
})
export class StoreModule {}
```

Dekorator `@NgModule` konfiguruje moduł i za pomocą właściwości `imports` wskazuje frameworkowi Angular, że działanie modułu sklepu internetowego zależy od modułu modelu, a także modułów `BrowserModule` i `FormsModule`, które zawierają standardowe funkcje Angular przeznaczone dla aplikacji internetowych i pracy z elementami formularza HTML. Ten dekorator używa właściwości `declarations` do poinformowania frameworka Angular o klasie `StoreComponent`, natomiast właściwość `exports` wskazuje na możliwość użycia tego modułu w innych miejscach aplikacji. To jest ważne, ponieważ wymieniony moduł będzie wykorzystany przez moduł główny.

Uaktualnienie komponentu i modułu głównego

Zastosowanie podstawowego modelu i funkcjonalności sklepu internetowego wymaga uaktualnienia modułu głównego aplikacji w celu zaimportowania dwóch przygotowanych wcześniej zwykłych modułów. Konieczne jest również uaktualnienie szablonu modułu głównego polegające na wskazaniu elementu HTML, do którego ma zastosowanie komponent w module sklepu internetowego. Na listingu 7.15 przedstawiłem zmianę niezbędną do wprowadzenia w komponencie głównym.

Listing 7.15. *Dodanie nowego elementu w pliku app.component.ts*

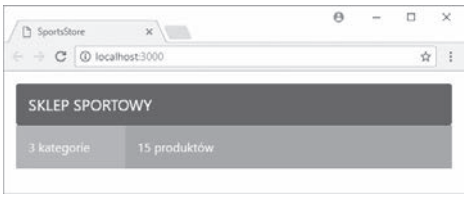
```
import { Component } from "@angular/core";

@Component({
    selector: "app",
    template: "<store></store>"
})
export class AppComponent {}
```

Element `<store>` zastępuje wcześniejszą treść szablonu komponentu głównego i odpowiada wartości właściwości `selector` dekoratora `@Component` na listingu 7.12. Na listingu 7.16 przedstawiłem zmiany konieczne do wprowadzenia w module głównym, aby framework Angular wczytał moduł zawierający funkcjonalność sklepu internetowego.

Listing 7.16. *Zaimportowanie modułów w pliku app.module.ts*

```
import { NgModule } from "@angular/core";
import { LOCALE_ID } from '@angular/core';
import { BrowserModule } from "@angular/platform-browser";
import { AppComponent } from "./app.component";
import { StoreModule } from "./store/store.module";

@NgModule({
    imports: [BrowserModule, StoreModule],
    declarations: [AppComponent],
    bootstrap: [AppComponent],
    providers: [{ provide: LOCALE_ID, useValue: "pl-PL" }]
})
export class AppModule {}
```

Po zapisaniu zmian w module głównym Angular ma wszystkie informacje niezbędne do wczytania aplikacji i wyświetlenia treści pochodzącej z modułu sklepu internetowego, jak pokazałem na rysunku 7.4.



Rysunek 7.4. *Podstawowe funkcje w aplikacji sklepu internetowego*

Wszystkie elementy konstrukcyjne przygotowane w poprzednich sekcjach współdziałają ze sobą i wyświetlają prostą treść. Na tym etapie otrzymujemy informacje o liczbie kategorii i produktów w sklepie.

Dodawanie funkcji związanych z produktem

Programowanie z użyciem frameworka Angular oznacza powolny start, w trakcie którego przygotowywane są fundamenty projektu i tworzone są podstawowe elementy konstrukcyjne. Gdy ta praca zostanie wykonana, dodawanie kolejnych funkcji będzie już względnie prostym zadaniem. W kolejnych sekcjach zajmę się dodawaniem do sklepu internetowego funkcji pozwalających użytkownikowi na przeglądanie katalogu dostępnych produktów.

Wyświetlanie szczegółów produktu

Wydaje się oczywiste, że pracę należy rozpocząć od wyświetlenia szczegółów produktów oferowanych klientowi. Na listingu 7.17 przedstawiłem elementy HTML, które trzeba umieścić w szablonie komponentu sklepu. Wyrażenia dołączania danych odpowiadają za wygenerowanie treści dla poszczególnych produktów.

Listing 7.17. *Dodawanie elementów w pliku store.component.html*

```
<div class="navbar navbar-inverse bg-inverse">
    <a class="navbar-brand">SKLEP SPORTOWY</a>
</div>
<div class="col-xs-3 bg-info p-a-1">
    {{categories.length}} kategorie
</div>
<div class="col-xs-9 p-a-1">
    <div *ngFor="let product of products" class="card card-outline-primary">
        <h4 class="card-header">
            {{product.name}}
            <span class="pull-xs-right tag tag-pill tag-primary">
                {{ product.price | currency:"PLN":true:"2.2-2" }}
            </span>
        </h4>
        <div class="card-text p-a-1">{{product.description}}</div>
    </div>
</div>
```

Większość elementów kontroluje układ i wygląd treści wyświetlanej na stronie. Najważniejszą zmianą jest dodanie wyrażenia dołączania danych Angular, jak pokazałem poniżej.

```
...
<div *ngFor="let product of products" class="card card-outline-primary">
...
```

To jest przykład dyrektywy przeprowadzającej transformację elementu HTML, do którego została zastosowana. W omawianym przykładzie mamy dyrektywę `ngFor`, której działanie polega na transformacji elementu `<div>` przez jego powielenie dla każdego obiektu zwróconego przez właściwość `products` komponentu. Angular oferuje wiele wbudowanych dyrektyw wykonujących najczęściej wymagane zadania. Więcej informacji na ten temat znajdziesz w rozdziale 13.

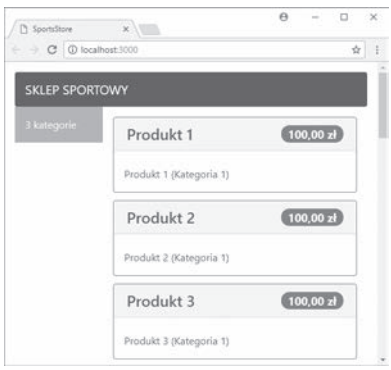
Podczas powielania elementu `<div>` bieżący obiekt zostaje przypisany zmiennej o nazwie `product`, do której można się łatwo odwoływać w innych wyrażeniach dołączania danych. Działanie przedstawionego tutaj wyrażenia powoduje wstawienie wartości właściwości `description` obiektu `product` jako treści elementu `<div>`, jak pokazałem poniżej.

```
...
<div class="card-text p-a-1">{{product.description}}</div>
...
```

Nie wszystkie dane znajdujące się w modelu aplikacji mogą być bezpośrednio wyświetlone użytkownikowi. Framework Angular oferuje funkcję o nazwie **potok**. Są to po prostu klasy używane do transformacji lub przygotowania wartości danych do użycia w procesie dołączania danych. Dostępnych jest wiele wbudowanych potoków, między innymi `currency`, który formatuje liczby jako wartości walutowe, jak pokazałem poniżej.

```
...
{{ product.price | currency:"PLN":true:"2.2-2" }}
...
```

Składnia zastosowania potoku może wydawać się nieco dziwna, ale wyrażenie w podanym poleceniu nakazuje Angular sformatowanie wartości właściwości `price` obiektu `product` z użyciem potoku `currency`, przy czym mają być zastosowane ustawienia polskiej waluty. Po zapisaniu zmian w szablonie zobaczysz długą listę produktów pobranych z modelu, jak pokazałem na rysunku 7.5.



Rysunek 7.5. *Wyświetlanie informacji o produkcie*

Dodawanie możliwości wyboru kategorii

Zaimplementowanie obsługi filtrowania listy produktów według kategorii wymaga odpowiedniego przygotowania komponentu sklepu internetowego, tak aby śledził on kategorię, którą chce wyświetlić użytkownik. Konieczna będzie więc zmiana sposobu pobierania danych dla danej kategorii, jak pokazałem na listingu 7.18.

Listing 7.18. *Zaimplementowanie w pliku store.component.ts filtrowania kategorii*

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";
```

```
@Component({
    selector: "store",
    moduleId: module.id,
    templateUrl: "store.component.html"
})
export class StoreComponent {
    public selectedCategory = null;

    constructor(private repository: ProductRepository) {}

    get products(): Product[] {
        return this.repository.getProducts(this.selectedCategory);
    }

    get categories(): string[] {
        return this.repository.getCategories();
    }

    changeCategory(newCategory?: string) {
        this.selectedCategory = newCategory;
    }
}
```

Zmiany są proste, ponieważ opierają się na fundamentach tak pieczołowicie budowanych na początku rozdziału. Właściwości `selectedCategory` zostaje przypisana kategoria wybrana przez użytkownika (przy czym wartość `null` oznacza wszystkie kategorie). Ta wartość jest używana w metodzie `updateData()` jako argument metody `getProducts()`, delegując do źródła danych operację filtrowania produktów. Metoda `changeCategory()` wykorzystuje oba elementy składowe w metodzie, która może być wywołana, gdy użytkownik wybierze kategorię.

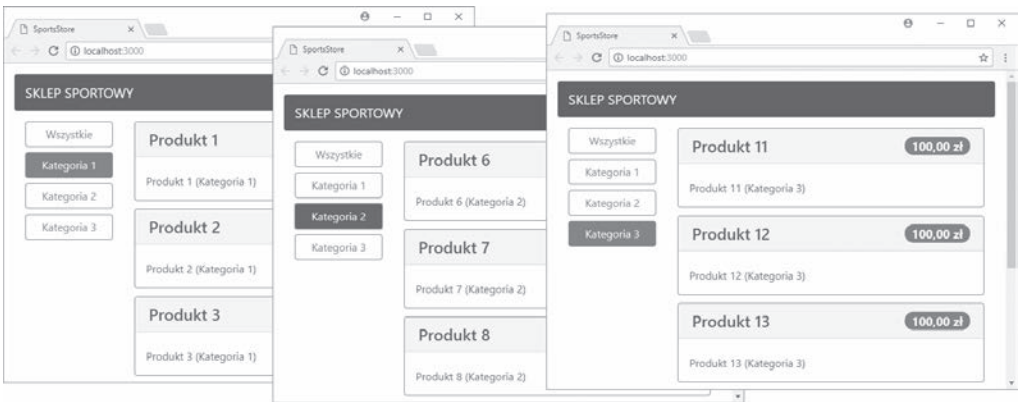
Na listingu 7.19 przedstawiłem zmiany konieczne do wprowadzenia w szablonie dokumentu, aby dostarczyć użytkownikowi zestaw przycisków pozwalających na zmianę kategorii i wyświetlić produkty w tej wybranej.

Listing 7.19. *Dodanie w pliku store.component.html przycisków pozwalających na wybór kategorii*

```
<div class="navbar navbar-inverse bg-inverse">
    <a class="navbar-brand">SKLEP SPORTOWY</a>
</div>
<div class="col-xs-3 p-a-1">
    <button class="btn btn-block btn-outline-primary" (click)="changeCategory()">
        Wszystkie
    </button>
    <button *ngFor="let cat of categories" class="btn btn-outline-primary btn-block"
            [class.active]="cat == selectedCategory" (click)="changeCategory(cat)">
        {{cat}}
    </button>
</div>
<div class="col-xs-9 p-a-1">
    <div *ngFor="let product of products" class="card card-outline-primary">
        <h4 class="card-header">
            {{product.name}}
            <span class="pull-xs-right tag tag-pill tag-primary">
                {{ product.price | currency:"PLN":true:"2.2-2" }}
            </span>
        </h4>
        <div class="card-text p-a-1">{{product.description}}</div>
    </div>
</div>
```

W szablonie pojawiły się dwa nowe elementy `<button>`. Pierwszy zatytułowany *Wszystkie* ma dołączone zdarzenie wywołujące metodę `changeCategory()` po kliknięciu przycisku. Metodzie nie jest przekazywany żaden argument, więc właściwość `selectedCategory` otrzymuje wartość `null`, co powoduje wyświetlenie wszystkich produktów.

Dyrektywa `ngFor` została zastosowana dla drugiego elementu `<button>`. Wyrażenie dyrektywy powoduje powielenie elementu dla każdej wartości w tablicy zwróconej przez właściwość `categories` komponentu. Element `<button>` ma przypisane zdarzenie `click`, którego wyrażenie wywołuje metodę `changeCategory()` w celu wybrania bieżącej kategorii, a tym samym wyświetlenia użytkownikowi jedynie produktów zaliczanych do danej kategorii. Mamy również wiązanie `class` dodające elementowi przycisku klasę `active`, gdy kategoria powiązana z danym przyciskiem jest wybraną kategorią. W ten sposób użytkownik otrzymuje wizualną informację o aktualnie wybranej kategorii, jak pokazałem na rysunku 7.6.



Rysunek 7.6. *Wybór kategorii produktów*

Dodawanie stronicowania produktów

Filtrowanie produktów według kategorii pomaga w łatwiejszym zarządzaniu listą produktów. Jednak znacznie częściej stosowane jest podejście polegające na podzieleniu listy na mniejsze sekcje i przedstawianie ich wraz z przyciskami nawigacyjnymi pozwalającymi na poruszanie się między stronami produktów.

Na listingu 7.20 przedstawiłem usprawnioną wersję komponentu sklepu internetowego monitorującą bieżącą stronę i liczbę elementów na stronie.

Listing 7.20*. Zaimplementowanie w pliku store.component.ts obsługi stronicowania*

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";

@Component({
    selector: "store",
    moduleId: module.id,
    templateUrl: "store.component.html"
})
export class StoreComponent {
    public selectedCategory = null;
    public productsPerPage = 4;
    public selectedPage = 1;

    constructor(private repository: ProductRepository) {}
```

```
    get products(): Product[] {
        let pageIndex = (this.selectedPage - 1) * this.productsPerPage
        return this.repository.getProducts(this.selectedCategory)
            .slice(pageIndex, pageIndex + this.productsPerPage);
    }

    get categories(): string[] {
        return this.repository.getCategories();
    }

    changeCategory(newCategory?: string) {
        this.selectedCategory = newCategory;
    }

    changePage(newPage: number) {
        this.selectedPage = newPage;
    }

    changePageSize(newSize: number) {
        this.productsPerPage = Number(newSize);
        this.changePage(1);
    }

    get pageNumbers(): number[] {
        return Array(Math.ceil(this.repository
            .getProducts(this.selectedCategory).length / this.productsPerPage))
                .fill(0).map((x, i) => i + 1);
    }
}
```

Na listingu pojawiły się dwie nowe funkcje. Pierwsza to możliwość pobrania strony produktów, natomiast druga to zmiana wielkości strony — dzięki temu użytkownik może zdecydować o liczbie produktów umieszczonych na stronie.

Mamy tu do czynienia z pewnym dziwactwem, które trzeba będzie obejść za pomocą komponentu. Wbudowana dyrektywa `ngFor` ma ograniczenie polegające na generowaniu treści jedynie dla obiektów pochodzących z tablicy lub kolekcji, a nie na podstawie wartości licznika. Skoro konieczne jest wygenerowanie ponumerowanych przycisków nawigacji między stronami, trzeba utworzyć tablicę zawierającą te liczby. Spójrz na poniższy fragment kodu.

```
...
return Array(Math.ceil(this.repository.getProducts(this.selectedCategory).length
    / this.productsPerPage)).fill(0).map((x, i) => i + 1);
...
```

To polecenie tworzy nową tablicę, wypełnia ją zerami, a następnie używa metody `map()` do wygenerowania nowej tablicy wraz z sekwencją liczb. Tego rodzaju podejście sprawdza się doskonale podczas implementacji funkcji stronicowania, ale wygląda okropnie. W następnej sekcji przedstawię znacznie lepsze podejście. Na listingu 7.21 pokazałem zmiany konieczne do wprowadzenia w szablonie sklepu internetowego, aby móc wykorzystać możliwość stronicowania produktów.

Listing 7.21. *Wykorzystanie stronicowania produktów w pliku store.component.html*

```
<div class="navbar navbar-inverse bg-inverse">
    <a class="navbar-brand">SKLEP SPORTOWY</a>
</div>
<div class="col-xs-3 p-a-1">
    <button class="btn btn-block btn-outline-primary" (click)="changeCategory()">
        Wszystkie
```

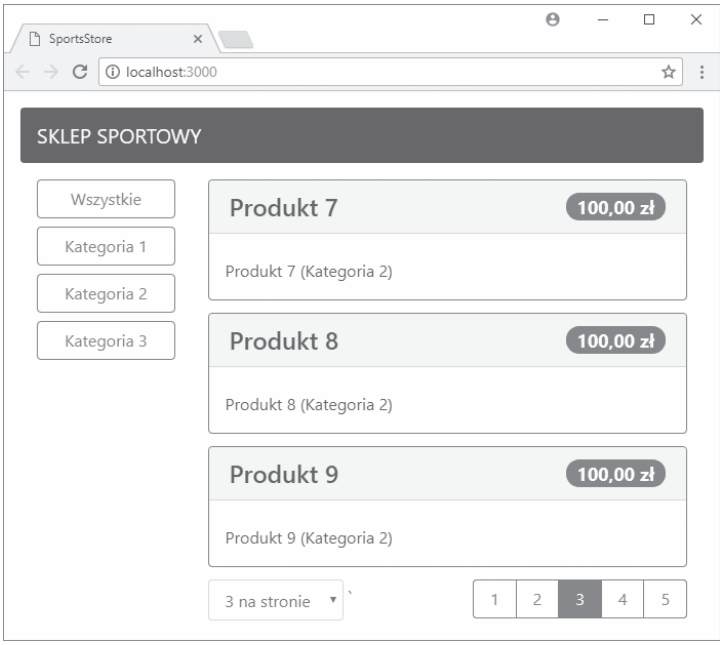
```
    </button>
    <button *ngFor="let cat of categories" class="btn btn-outline-primary btn-block"
            [class.active]="cat == selectedCategory" (click)="changeCategory(cat)">
        {{cat}}
    </button>
</div>
<div class="col-xs-9 p-a-1">
    <div *ngFor="let product of products" class="card card-outline-primary">
        <h4 class="card-header">
            {{product.name}}
            <span class="pull-xs-right tag tag-pill tag-primary">
                {{ product.price | currency:"PLN":true:"2.2-2" }}
            </span>
        </h4>
        <div class="card-text p-a-1">{{product.description}}</div>
    </div>
    <div class="form-inline pull-xs-left m-r-1">
        <select class="form-control" [value]="productsPerPage"
                (change)="changePageSize($event.target.value)">
            <option value="3">3 na stronie</option>
            <option value="4">4 na stronie</option>
            <option value="6">6 na stronie</option>
            <option value="8">8 na stronie</option>
        </select>
    </div>
    <div class="btn-group pull-xs-right">
        <button *ngFor="let page of pageNumbers" (click)="changePage(page)"
            class="btn btn-outline-primary" [class.active]="page == selectedPage">
                {{page}}
        </button>
    </div>
</div>
```

Nowy kod powoduje dodanie elementu `<select>` pozwalającego użytkownikowi na zmianę liczby produktów wyświetlanych na stronie oraz zestawu przycisków nawigacyjnych do poruszania się po stronach produktów. Te nowe elementy mają zdefiniowane wyrażenia dołączania danych łączące je z właściwościami i metodami dostarczanymi przez komponent. W efekcie wprowadzonych zmian otrzymujemy znacznie łatwiejszy w zarządzaniu zbiór produktów, jak pokazałem na rysunku 7.7.

■ **Wskazówka** Element `<select>` przedstawiony na listingu 7.21 zawiera elementy `<option>` zdefiniowane statycznie, a nie utworzone na podstawie danych komponentu. Dlatego też po przekazaniu wybranej wartości metodzie `changePageSize()` będzie ona typu `string`, stąd konieczność zmiany argumentu na typ `number` przed jego użyciem do zdefiniowania wielkości strony na listingu 7.20. Trzeba zachować szczególną ostrożność podczas używania wartości pochodzących z elementów formularza HTML i upewnić się, że są one oczekiwanego typu. Oferowane przez TypeScript adnotacje skryptu tutaj nie pomogą, ponieważ wyrażenie dołączania danych jest wykonywane w trakcie działania programu, czyli znacznie później po tym, gdy kompilator TypeScript wygenerował kod JavaScript niezawierający dodatkowych informacji o typie.

Utworzenie własnej dyrektywy

W tej sekcji zamierzam utworzyć własną dyrektywę, aby uniknąć konieczności generowania tablicy liczb potrzebnej do przygotowania przycisków nawigacji między stronami w trakcie stronicowania produktów. Framework Angular oferuje dość szeroką gamę wbudowanych dyrektyw. Prosty proces utworzenia własnej



Rysunek 7.7. *Zaimplementowane stronicowanie produktów*

dyrektywy rozwiązuje problem w naszej aplikacji, pozwala również na dodanie obsługi funkcji nieoferowanych przez wbudowane dyrektywy. W katalogu *SportsStore/src/app/store* utwórz nowy plik o nazwie *counter.directive.ts* i umieść w nim kod przedstawiony na listingu 7.22.

Listing 7.22. *Zawartość pliku counter.directive.ts w katalogu SportsStore/src/app/store*

```
import {
    Directive, ViewContainerRef, TemplateRef, Input, Attribute, SimpleChanges
} from "@angular/core";

@Directive({
    selector: "[counterOf]"
})
export class CounterDirective {
    constructor(private container: ViewContainerRef,
        private template: TemplateRef<Object>) {
    }

    @Input("counterOf")
    counter: number;

    ngOnChanges(changes: SimpleChanges) {
        this.container.clear();
        for (let i = 0; i < this.counter; i++) {
            this.container.createEmbeddedView(this.template,
                new CounterDirectiveContext(i + 1));
        }
    }
```

```
}

class CounterDirectiveContext {
    constructor(public $implicit: any) {}
}
```

To jest przykład dyrektywy strukturalnej, ten rodzaj dyrektyw omówię dokładnie w rozdziale 16. Dyrektywa będzie stosowana w elemencie za pomocą właściwości `counter`. Opiera ona swoje działanie na funkcjach specjalnych dostarczanych przez framework Angular na potrzeby powtarzającego się tworzenia treści, podobnie jak ma to miejsce w przypadku dyrektywy `ngFor`. W takim przypadku zamiast pobierać poszczególne obiekty z kolekcji, dyrektywa pobiera serię liczb, które następnie mogą być użyte do utworzenia przycisków nawigacji między stronami produktów.

■ **Wskazówka** W przypadku zmiany liczby stron ta dyrektywa powoduje usunięcie całej utworzonej przez siebie treści i rozpoczyna proces jej tworzenia zupełnie od początku. W bardziej skomplikowanych dyrektywach może to być dość kosztowny proces, dlatego w rozdziale 16. pokażę, jak można poprawić wydajność jego działania.

Aby móc użyć dyrektywy, należy ją dodać do właściwości `declarations` zwykłego modułu, jak pokazałem na listingu 7.23.

Listing 7.23. *Przykład rejestrowania własnej dyrektywy w pliku store.module.ts*

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { StoreComponent } from "./store.component";
import { CounterDirective } from "./counter.directive";

@NgModule({
    imports: [ModelModule, BrowserModule, FormsModule],
    declarations: [StoreComponent, CounterDirective],
    exports: [StoreComponent]
})
export class StoreModule {}
```

Po zarejestrowaniu dyrektywy można ją wykorzystać w szablonie komponentu sklepu internetowego i zastąpić nią wbudowaną dyrektywę `ngFor`, jak pokazałem na listingu 7.24.

Listing 7.24. *Zastąpienie w pliku store.component.html wbudowanej dyrektywy ngFor*

```
<div class="navbar navbar-inverse bg-inverse">
    <a class="navbar-brand">SKLEP SPORTOWY</a>
</div>
<div class="col-xs-3 p-a-1">
    <button class="btn btn-block btn-outline-primary" (click)="changeCategory()">
        Wszystkie
    </button>
    <button *ngFor="let cat of categories" class="btn btn-outline-primary btn-block"
            [class.active]="cat == selectedCategory" (click)="changeCategory(cat)">
        {{cat}}
    </button>
</div>
<div class="col-xs-9 p-a-1">
    <div *ngFor="let product of products" class="card card-outline-primary">
        <h4 class="card-header">
```

```
            {{product.name}}
            <span class="pull-xs-right tag tag-pill tag-primary">
                {{ product.price | currency:"PLN":true:"2.2-2" }}
            </span>
        </h4>
        <div class="card-text p-a-1">{{product.description}}</div>
    </div>
    <div class="form-inline pull-xs-left m-r-1">
        <select class="form-control" [value]="productsPerPage"
                (change)="changePageSize($event.target.value)">
            <option value="3">3 na stronie</option>
            <option value="4">4 na stronie</option>
            <option value="6">6 na stronie</option>
            <option value="8">8 na stronie</option>
        </select>
`   </div>
    <div class="btn-group pull-xs-right">
        <button *counter="let page of pageCount" (click)="changePage(page)"
            class="btn btn-outline-primary" [class.active]="page == selectedPage">
                {{page}}
        </button>
    </div>
</div>
```

Nowe wyrażenie dołączania danych opiera działanie na właściwości o nazwie `pageCount` konfigurującej niestandardową dyrektywę. Na listingu 7.25 pokazałem zastąpienie tablicy liczb prostym wywołaniem dostarczającym wartość dla wyrażenia.

Listing 7.25. *Przykład użycia w pliku store.component.ts własnej dyrektywy*

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";

@Component({
    selector: "store",
    moduleId: module.id,
    templateUrl: "store.component.html"
})
export class StoreComponent {
    public selectedCategory = null;
    public productsPerPage = 4;
    public selectedPage = 1;

    constructor(private repository: ProductRepository) {}

    get products(): Product[] {
        let pageIndex = (this.selectedPage - 1) * this.productsPerPage
        return this.repository.getProducts(this.selectedCategory)
            .slice(pageIndex, pageIndex + this.productsPerPage);
    }

    get categories(): string[] {
        return this.repository.getCategories();
    }

    changeCategory(newCategory?: string) {
        this.selectedCategory = newCategory;
    }
```

```
    changePage(newPage: number) {
        this.selectedPage = newPage;
    }

    changePageSize(newSize: number) {
        this.productsPerPage = Number(newSize);
        this.changePage(1);
    }

    get pageCount(): number {
        return Math.ceil(this.repository
            .getProducts(this.selectedCategory).length / this.productsPerPage)
    }

    // get pageNumbers(): number[] {
    //     return Array(Math.ceil(this.repository
    //         .getProducts(this.selectedCategory).length / this.productsPerPage))
    //             .fill(0).map((x, i) => i + 1);
    // }
}
```

Nie powinieneś zauważyć żadnej widocznej zmiany w sposobie działania aplikacji SportsStore, chociaż w tej sekcji pokazałem, jak wbudowaną funkcjonalność frameworka Angular można uzupełnić własnym kodem dopasowanym do potrzeb danego projektu.

Podsumowanie

W tym rozdziale rozpoczęliśmy proces budowy aplikacji SportsStore. Na początku rozdziału zająłem się przygotowaniem struktury projektu, między innymi zainstalowałem i skonfigurowałem narzędzia programistyczne, utworzyłem główne elementy konstrukcyjne aplikacji i rozpocząłem pracę nad poszczególnymi modułami. Po zakończeniu etapu przygotowań bardzo szybko byłem w stanie dodać funkcje pozwalające na wyświetlanie użytkownikowi fikcyjnych danych, a także zaimplementować stronicowanie oraz filtrowanie produktów według kategorii. Na końcu rozdziału pokazałem, jak można utworzyć własną dyrektywę oraz jak funkcje oferowane przez framework Angular uzupełnić własnym kodem. W następnym rozdziale będę kontynuował pracę nad aplikacją SportsStore.

Skorowidz

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

X

Z

Ź

Ż

Funkcjonalne narzędzie dla profesjonalisty

Framework Angular 2 jest dziś uważany za jeden z najwydajniejszych sposobów rozwijania średnich i dużych aplikacji internetowych. Działają one bez problemu niezależnie od platformy. Separacja kodu interfejsu od logiki aplikacji, prosta modularyzacja aplikacji, bezproblemowe przetwarzanie danych asynchronicznych, bogaty zestaw narzędzi i ogromne ułatwienia podczas projektowania interfejsu użytkownika — to tylko kilka z wielu zalet Angulara. Nawet jednak tak świetny framework wymaga od programisty wiedzy i umiejętności prawidłowego budowania kodu.

Książka rozpoczyna się od przedstawienia wzorca MVC i jego zalet. Ta wiedza następnie przydaje się do budowy własnego projektu za pomocą Angulara. Najpierw będzie to prosta, praktyczna aplikacja, a później zostaną zaprezentowane bardziej zaawansowane funkcje. Każdy temat jasno i spójnie wyjaśniono, ze szczegółami koniecznymi do efektywnej pracy. Przedstawiono sposoby wykorzystywania różnych pożytecznych narzędzi, w tym frameworka Bootstrap, biblioteki Reactive Extensions czy frameworka Jasmine. Nie zabrakło objaśnień najczęściej występujących problemów oraz sposobów ich rozwiązywania.

W tej książce między innymi:

- solidne podstawy koncepcji MVC
- obiektowy model dokumentu (DOM)
- usługi i dostawcy usług
- routing i animacja w Angularze
- testowanie aplikacji

Adam Freeman — jest wyjątkowo doświadczonym programistą i architektem. Doskonale rozumie wyzwania, jakie niesie ze sobą zapewnienie bezpieczeństwa dużym systemom informatycznym. Pracował w wielu firmach między innymi Netscape i Sun Microsystems, a ostatnio zajmował stanowisko dyrektora naczelnego w międzynarodowym banku. Obecnie jest na emeryturze. Swój czas dzieli między dwie pasje: pisanie i bieganie.

ISBN 978-83-283-4231-6
9 788328 342316
Cena: 119,00 zł
INFORMATYKA W NAJLEPSZYM WYDANIU
Apress