



Yakov Fain & Anton Moiseev

Angular 2

Programowanie z użyciem języka **TypeScript**

Helion 

Tytuł oryginału: Angular 2 Development with TypeScript

Tłumaczenie: Lech Lachowski

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdriowski, Zaprucki

ISBN: 978-83-283-3638-4

Original edition copyright © 2017 by Manning Publications Co.
All rights reserved.

Polish edition copyright © 2017 by HELION SA.
All rights reserved.

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.

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

Autor oraz Wydawnictwo HELION 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 Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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/ang2ty.zip>

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/ang2ty>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Spis treści

Przedmowa	9
Podziękowania	13
O książce	15
O autorach	19
Rozdział 1. Wprowadzenie do frameworku Angular 2	21
1.1. Przegląd frameworków i bibliotek JavaScript	22
1.1.1. Frameworki w pełni funkcjonalne	22
1.1.2. Lekkie frameworki	22
1.1.3. Biblioteki	23
1.1.4. Czym jest node.js?	24
1.2. Ogólny przegląd frameworku AngularJS	25
1.3. Ogólny przegląd frameworku Angular	28
1.3.1. Uproszczenie kodu	28
1.3.2. Poprawa wydajności	34
1.4. Zestaw narzędzi programisty Angular	35
1.5. Jak działa Angular?	39
1.6. Wprowadzenie do przykładu aplikacji aukcji internetowych	40
1.7. Podsumowanie	41
Rozdział 2. Zaczynamy pracę z frameworkiem Angular	43
2.1. Pierwsza aplikacja Angular	44
2.1.1. Witaj, świecie w języku TypeScript	44
2.1.2. Witaj, świecie w ES5	47
2.1.3. Witaj, świecie w ES6	49
2.1.4. Uruchamianie aplikacji	50
2.2. Elementy konstrukcyjne aplikacji Angular	51
2.2.1. Moduły	51
2.2.2. Komponenty	52
2.2.3. Dyrektywy	54
2.2.4. Krótkie wprowadzenie do wiązania danych	55
2.3. Uniwersalna ładowarka modułów SystemJS	55
2.3.1. Przegląd ładowarek modułów	56
2.3.2. Porównanie ładowarek modułów i znaczników <code><script></code>	56
2.3.3. Pierwsze kroki z SystemJS	57
2.4. Wybór menedżera pakietów	62
2.4.1. Porównanie <code>npm</code> i <code>jspm</code>	64
2.4.2. Rozpoczynanie projektu Angular za pomocą <code>npm</code>	65

2.5.	Część praktyczna: rozpoczynamy tworzenie aplikacji aukcji internetowych	70
2.5.1.	Wstępna konfiguracja projektu	71
2.5.2.	Tworzenie strony głównej	74
2.5.3.	Uruchomienie aplikacji aukcji internetowych	82
2.6.	Podsumowanie	82
Rozdział 3. Nawigacja za pomocą routera Angular		85
3.1.	Podstawy routingu	86
3.1.1.	Strategie lokalizacji	87
3.1.2.	Bloki konstrukcyjne nawigacji po stronie klienta	89
3.1.3.	Nawigacja do tras za pomocą metody <code>navigate()</code>	94
3.2.	Przekazywanie danych do tras	97
3.2.1.	Wydodrębnianie parametrów z <code>ActivatedRoute</code>	97
3.2.2.	Przekazywanie do trasy statycznych danych	100
3.3.	Trasy podrzędne	101
3.4.	Strzeżenie tras	107
3.5.	Tworzenie aplikacji SPA z wieloma outletami routera	112
3.6.	Dzielenie aplikacji na moduły	115
3.7.	Moduły leniwie ładowane	117
3.8.	Część praktyczna: dodanie nawigacji do aplikacji aukcji internetowych	119
3.8.1.	Tworzenie komponentu <code>ProductDetailComponent</code>	120
3.8.2.	Tworzenie komponentu <code>HomeComponent</code> i refaktoryzacja kodu	121
3.8.3.	Uproszczenie komponentu <code>ApplicationComponent</code>	122
3.8.4.	Dodawanie dyrektywy <code>RouterLink</code> do <code>ProductItemComponent</code>	123
3.8.5.	Modyfikacja modułu głównego w celu dodania routingu	125
3.8.6.	Uruchomienie aplikacji aukcji internetowych	126
3.9.	Podsumowanie	126
Rozdział 4. Wstrzykiwanie zależności		129
4.1.	Wzorce Wstrzykiwanie Zależności i Odwrócenie Sterowania	130
4.1.1.	Wzorzec Wstrzykiwanie Zależności	130
4.1.2.	Wzorzec Odwrócenie Sterowania	131
4.1.3.	Korzyści płynące ze wstrzykiwania zależności	131
4.2.	Wstrzykiwacze i dostawcy	133
4.2.1.	Jak zadeklarować dostawcę?	135
4.3.	Przykładowa aplikacja ze wstrzykiwaniem zależności frameworku Angular	137
4.3.1.	Wstrzyknięcie usługi produktowej	137
4.3.2.	Wstrzyknięcie usługi <code>Http</code>	140
4.4.	Ułatwione przełączanie wstrzykiwaczy	141
4.4.1.	Deklarowanie dostawców za pomocą właściwości <code>useFactory</code> i <code>useValue</code>	144
4.4.2.	Korzystanie z klasy <code>OpaqueToken</code>	147
4.5.	Hierarchia wstrzykiwaczy	148
4.5.1.	Właściwość <code>viewProviders</code>	150
4.6.	Część praktyczna: użycie mechanizmu DI w aplikacji aukcji internetowych	151
4.6.1.	Zmiana kodu w celu przekazania identyfikatora produktu jako parametru	154
4.6.2.	Modyfikacja komponentu <code>ProductDetailComponent</code>	154
4.7.	Podsumowanie	159

Rozdział 5. Wiązania, strumienie obserwowalne i potoki	161
5.1. Wiązanie danych	162
5.1.1. Wiązanie ze zdarzeniami	163
5.1.2. Wiązanie z właściwościami i atrybutami	164
5.1.3. Wiązanie w szablonach	168
5.1.4. Dwukierunkowe wiązanie danych	171
5.2. Programowanie reaktywne i strumienie obserwowalne	174
5.2.1. Czym są strumienie obserwowalne i obserwatory?	174
5.2.2. Obserwowalne strumienie zdarzeń	176
5.2.3. Anulowanie strumieni obserwowalnych	181
5.3. Potoki	184
5.3.1. Potoki niestandardowe	185
5.4. Część praktyczna: filtrowanie produktów w aplikacji aukcji internetowych	187
5.5. Podsumowanie	191
Rozdział 6. Implementowanie komunikacji komponentów	193
6.1. Komunikacja między komponentami	194
6.1.1. Właściwości wejściowe i wyjściowe	194
6.1.2. Wzorzec Mediator	201
6.1.3. Zmiana szablonów podczas pracy za pomocą dyrektywy <code>ngContent</code>	205
6.2. Cykl życia komponentów	210
6.2.1. Korzystanie z metody <code>ngOnChanges</code>	212
6.3. Ogólny przegląd działania mechanizmu wykrywania zmian	217
6.4. Udostępnianie interfejsu API komponentu potomnego	219
6.5. Część praktyczna: dodanie funkcjonalności oceniania do aplikacji aukcji internetowych	221
6.6. Podsumowanie	228
Rozdział 7. Praca z formularzami	231
7.1. Przegląd formularzy HTML	232
7.1.1. Standardowe funkcje przeglądarki	232
7.1.2. Interfejs <code>Forms API</code> frameworku Angular	234
7.2. Formularze oparte na szablonach	235
7.2.1. Przegląd dyrektyw	236
7.2.2. Wzbogacanie formularza HTML	238
7.3. Formularze reaktywne	240
7.3.1. Model formularza	240
7.3.2. Dyrektywy formularzy	241
7.3.3. Refaktoryzacja przykładowego formularza	245
7.3.4. Korzystanie z klasy <code>FormBuilder</code>	246
7.4. Walidacja formularza	247
7.4.1. Walidacja formularzy reaktywnych	247
7.5. Część praktyczna: dodanie walidacji do formularza wyszukiwania	256
7.5.1. Modyfikacja modułu głównego w celu dodania obsługi interfejsu <code>Forms API</code>	257
7.5.2. Dodawanie listy kategorii do <code>SearchComponent</code>	257
7.5.3. Tworzenie modelu formularza	258

7.5.4.	<i>Refaktoryzacja szablonu</i>	259
7.5.5.	<i>Implementacja metody onSearch()</i>	260
7.5.6.	<i>Uruchomienie aplikacji aukcji internetowych</i>	260
7.6.	<i>Podsumowanie</i>	261
Rozdział 8. Interakcja z serwerami przy użyciu protokołów HTTP i WebSocket		263
8.1.	Krótkie omówienie interfejsu API obiektu Http	264
8.2.	Tworzenie serwera WWW za pomocą frameworku Node i języka TypeScript	266
8.2.1.	<i>Tworzenie prostego serwera WWW</i>	267
8.2.2.	<i>Serwowanie danych w formacie JSON</i>	269
8.2.3.	<i>Rekompilacja TypeScriptu na żywo i ponowne załadowanie kodu</i>	271
8.2.4.	<i>Dodawanie interfejsu RESTful API dla serwowania produktów</i>	272
8.3.	Łączenie frameworku Angular i serwera Node	273
8.3.1.	<i>Zasoby statyczne na serwerze</i>	274
8.3.2.	<i>Wykonywanie żądań GET za pomocą obiektu Http</i>	276
8.3.3.	<i>Odpakowywanie obiektów obserwowalnych w szablonach za pomocą AsyncPipe</i>	278
8.3.4.	<i>Wstrzyknięcie HTTP do usługi</i>	280
8.4.	Komunikacja klient-serwer poprzez protokół WebSocket	283
8.4.1.	<i>Wysyłanie danych z serwera Node</i>	284
8.4.2.	<i>Zamiana obiektu WebSocket w strumień obserwowalny</i>	287
8.5.	Część praktyczna: implementacja wyszukiwania produktów i powiadomień o ofertach	294
8.5.1.	<i>Implementowanie wyszukiwania produktów przy użyciu protokołu HTTP</i>	295
8.5.2.	<i>Rozgłaszanie ofert aukcji za pomocą WebSocket</i>	299
8.6.	<i>Podsumowanie</i>	302
Rozdział 9. Testy jednostkowe aplikacji Angular		305
9.1.	Wprowadzenie do Jasmine	306
9.1.1.	<i>Co należy testować?</i>	309
9.1.2.	<i>Jak zainstalować Jasmine?</i>	309
9.2.	Co zawiera biblioteka testowa Angular?	310
9.2.1.	<i>Testowanie usług</i>	312
9.2.2.	<i>Testowanie nawigacji routera</i>	312
9.2.3.	<i>Testowanie komponentów</i>	313
9.3.	Testowanie przykładowej aplikacji pogodowej	314
9.3.1.	<i>Konfigurowanie ładowarki SystemJS</i>	316
9.3.2.	<i>Testowanie routera pogody</i>	316
9.3.3.	<i>Testowanie usługi pogodowej</i>	319
9.3.4.	<i>Testowanie komponentu pogodowego</i>	321
9.4.	Uruchamianie testów za pomocą narzędzia Karma	325
9.5.	Część praktyczna: testy jednostkowe aplikacji aukcji internetowych	328
9.5.1.	<i>Testowanie komponentu AppComponent</i>	329
9.5.2.	<i>Testowanie usługi ProductService</i>	330
9.5.3.	<i>Testowanie komponentu StarsComponent</i>	331
9.5.4.	<i>Uruchomienie testów</i>	333
9.6.	<i>Podsumowanie</i>	334

Rozdział 10. Tworzenie paczek i wdrażanie aplikacji za pomocą narzędzia Webpack	335
10.1. Poznajemy Webpack	338
10.1.1. Witaj, świecie z zastosowaniem bundlera Webpack	339
10.1.2. Jak używać ładowarek?	343
10.1.3. Jak używać wtyczek?	347
10.2. Tworzenie podstawowej konfiguracji Webpack dla frameworku Angular	348
10.2.1. Uruchomienie <code>npm run build</code>	351
10.2.2. Uruchomienie <code>npm start</code>	352
10.3. Tworzenie konfiguracji programistycznych i produkcyjnych	353
10.3.1. Konfiguracja programistyczna	353
10.3.2. Konfiguracja produkcyjna	354
10.3.3. Niestandardowy plik definicji typów	357
10.4. Co to jest Angular CLI?	358
10.4.1. Rozpoczęcie nowego projektu za pomocą Angular CLI	359
10.4.2. Polecenia CLI	361
10.5. Część praktyczna: wdrożenie aplikacji aukcji internetowych za pomocą bundlera Webpack	362
10.5.1. Uruchamianie serwera Node	362
10.5.2. Uruchomienie klienta aplikacji aukcji internetowych	364
10.5.3. Uruchomienie testów za pomocą narzędzia Karma	367
10.6. Podsumowanie	370
Dodatek A. Przegląd specyfikacji ECMAScript 6	371
A.1. Jak uruchamiać przykłady kodu?	372
A.2. Literały szablonów	372
A.2.1. Wieloliniowe łańcuchy znaków	373
A.2.2. Oznaczone łańcuchy znaków szablonów	374
A.3. Parametry opcjonalne i wartości domyślne	375
A.4. Zakres zmiennych	376
A.4.1. Wynoszenie zmiennych	376
A.4.2. Tworzenie zakresu bloku za pomocą słów kluczowych <code>let</code> i <code>const</code>	378
A.4.3. Zakres bloku dla funkcji	380
A.5. Wyrażenia funkcji strzałkowych, <code>this</code> i <code>that</code>	380
A.5.1. Operatory reszty i rozwijania	383
A.5.2. Generatory	385
A.5.3. Destrukturyzacja	387
A.6. Iterowanie za pomocą <code>forEach()</code> , <code>for-in</code> i <code>for-of</code>	391
A.6.1. Korzystanie z metody <code>forEach()</code>	391
A.6.2. Korzystanie z pętli <code>for-in</code>	391
A.6.3. Korzystanie z pętli <code>for-of</code>	392
A.7. Klasy i dziedziczenie	393
A.7.1. Konstruktory	394
A.7.2. Zmienne statyczne	395
A.7.3. Metody pobierające, ustawiające i definicje metod	396
A.7.4. Słowo kluczowe <code>super</code> i funkcja <code>super</code>	397

A.8.	Przetwarzanie asynchroniczne z wykorzystaniem obietnic	398
A.8.1.	<i>Koszmar wywołań zwrotnych</i>	399
A.8.2.	<i>Obietnice ES6</i>	399
A.8.3.	<i>Rozwiązanie kilku obietnic naraz</i>	402
A.9.	Moduły	403
A.9.1.	<i>Słowa kluczowe import i export</i>	404
A.9.2.	<i>Ładowanie modułów dynamicznie za pomocą ładowarki modułów ES6</i>	406

Dodatek B. TypeScript jako język dla aplikacji Angular 411

B.1.	Dlaczego pisać aplikacje Angular w języku TypeScript?	412
B.2.	Rola transkompilatorów	413
B.3.	Pierwsze kroki z językiem TypeScript	413
B.3.1.	<i>Instalacja i używanie kompilatora TypeScriptu</i>	414
B.4.	TypeScript jako nadzbiór JavaScriptu	417
B.5.	Typy opcjonalne	417
B.5.1.	<i>Funkcje</i>	419
B.5.2.	<i>Parametry domyślne</i>	420
B.5.3.	<i>Parametry opcjonalne</i>	420
B.5.4.	<i>Wyrażenia funkcji strzałkowych</i>	421
B.6.	Klasy	423
B.6.1.	<i>Modyfikatory dostępu</i>	425
B.6.2.	<i>Metody</i>	427
B.6.3.	<i>Dziedziczenie</i>	428
B.7.	Typy sparametryzowane	430
B.8.	Interfejsy	433
B.8.1.	<i>Deklarowanie typów niestandardowych z interfejsami</i>	433
B.8.2.	<i>Używanie słowa kluczowego implements</i>	434
B.8.3.	<i>Korzystanie z interfejsów wywoływalnych</i>	436
B.9.	Dodawanie metadanych klas za pomocą adnotacji	438
B.10.	Pliki definicji typów	439
B.10.1.	<i>Instalowanie plików definicji typów</i>	440
B.11.	Przegląd procesu tworzenia aplikacji TypeScript-Angular	441

Skorowidz	443
-----------	-----

Zaczynamy pracę z frameworkiem Angular

W tym rozdziale:

- napiszesz swoją pierwszą aplikację Angular;
- zapoznasz się z uniwersalną ładowarką modułów SystemJS;
- poznasz rolę menedżerów pakietów;
- opracujesz pierwszą wersję aplikacji aukcji internetowych.

W tym rozdziale rozpoczniemy omawianie sposobu tworzenia aplikacji Angular z wykorzystaniem nowoczesnych narzędzi i technologii internetowych, takich jak adnotacje, moduły ES6 i ładowarki modułów. Framework Angular zmienia sposób tworzenia aplikacji JavaScript. Napiszemy trzy wersje aplikacji *Witaj, świecie* oraz krótko omówimy menedżery pakietów i uniwersalną ładowarkę modułów SystemJS.

Potem utworzymy niewielki projekt, który może służyć jako szablon do tworzenia własnych projektów Angular. Następnie omówimy główne elementy konstrukcyjne aplikacji Angular, takie jak komponenty i widoki, oraz krótko opiszemy wstrzykiwanie zależności i wiązanie danych. Na końcu rozdziału przyjrzymy się aplikacji aukcji internetowych, którą będziemy rozwijać w całej książce.

UWAGA Wszystkie przykłady kodu zamieszczone w tej książce są oparte na wersji ostatecznej Angular 2.0.0.

WSKAZÓWKA Jeśli nie znasz składni TypeScriptu i ECMAScriptu 6, sugerujemy, żebyś przed kontynuowaniem lektury tego rozdziału przeczytał dodatki A i B.

2.1. Pierwsza aplikacja Angular

W tym podrozdziale pokażemy trzy wersje aplikacji *Witaj, świecie* napisane w TypeScriptie, ES5 i ES6. To jedyny podrozdział, w którym możesz zobaczyć aplikacje Angular zapisane w ES5 i ES6 — wszystkie pozostałe przykłady kodu zostały zapisane w języku TypeScript.

2.1.1. Witaj, świecie w języku TypeScript

Ta pierwsza aplikacja będzie dość minimalistyczna, aby szybko rozpocząć programowanie z frameworkiem Angular. Będzie składać się z dwóch plików:

```
└── index.html
└── main.ts
```

Oba pliki znajdują się w katalogu *hello-world-ts* w towarzyszących książce przykładach kodu, które można pobrać z serwera wydawnictwa Helion (<ftp://ftp.helion.pl/przyklady/ang2ty.zip>). Plik *index.html* jest punktem wejścia dla aplikacji. Znajdują się w nim referencje do frameworku Angular, jego zależności oraz do pliku *main.ts*, który zawiera kod do załadowania aplikacji. Niektóre z tych referencji można umieścić w pliku konfiguracyjnym modułu ładowarki (w tej książce będziemy używać ładowarek SystemJS i Webpack).

ŁADOWANIE FRAMEWORKU ANGULAR W PLIKU HTML

Kod frameworku Angular składa się z modułów (jeden plik na moduł) połączonych w biblioteki, które są logicznie zgrupowane w pakiety, takie jak @angular/core, @angular/common i tak dalej. Twoja aplikacja musi załadować wymagane pakiety przed kodem aplikacji.

Utwórzmy plik *index.html*, który rozpocznie się od załadowania wymaganych skryptów Angular, kompilatora TypeScriptu i modułu ładowarki SystemJS. Poniższy kod z listingu 2.1 łączy te skrypty z sieci dostarczania treści (ang. *content delivery network* — CDN) <https://unpkg.com>.

Listing 2.1. Plik *index.html* aplikacji TypeScript

```
<!DOCTYPE html>
<html>
<head>
  <script src="//unpkg.com/zone.js@0.6.12"></script>
  <script src="//unpkg.com/typescript@2.0.0"></script>
  <script src="//unpkg.com/systemjs@0.19.37/dist/system.src.js"></script>
  <script src="//unpkg.com/core-js/client/shim.min.js"></script>
  <script>
    System.config({
      // ...
```

Zone.js jest biblioteką, która umożliwia działanie mechanizmu wykrywania zmian.

Kompilator TypeScriptu transkompiluje kod źródłowy na JavaScript bezpośrednio w przeglądarce.

Biblioteka SystemJS dynamicznie ładuje kod aplikacji do przeglądarki. Omówimy SystemJS w dalszej części rozdziału.

Konfiguruje ładowarkę SystemJS w celu ładowania i transkompilowania kodu TypeScript.

```

transpiler: 'typescript',
typescriptOptions: {emitDecoratorMetadata: true},
map: {
  'rxjs': 'https://unpkg.com/rxjs@5.0.0-beta.12',
  '@angular/core'      : 'https://unpkg.com/@angular/core@2.0.0',
  '@angular/common'    : 'https://unpkg.com/@angular/common@2.0.0',
  '@angular/compiler'   : 'https://unpkg.com/@angular/compiler@2.0.0',
  '@angular/platform-browser' : 'https://unpkg.com/@angular/
    ↳platform-browser@2.0.0',
  '@angular/platform-browser-dynamic': 'https://unpkg.com/@angular/
    ↳platform-browser-dynamic@2.0.0'
},
packages: {
  '@angular/core'      : {main: 'index.js'},
  '@angular/common'    : {main: 'index.js'},
  '@angular/compiler'   : {main: 'index.js'},
  '@angular/platform-browser' : {main: 'index.js'},
  '@angular/platform-browser-dynamic': {main: 'index.js'}
}
});
System.import('main.ts');
</script>
</head>
<body>
  <hello-world></hello-world>
</body>
</html>

```

Mapuje nazwy modułów Angular na ich lokalizację CDN.

Określa skrypt główny dla każdego modułu Angular.

Instruuje SystemJS, aby ładować moduł główny z pliku main.ts.

Niestandardowy element HTML <hello-world></hello-world> reprezentuje komponent, który jest zaimplementowany w pliku main.ts.

Po uruchomieniu aplikacji znacznik <hello-world> zostanie zastąpiony zawartością szablonu z adnotacji @Component pokazanej w listingu 2.2.

WSKAZÓWKA Jeśli używasz przeglądarki Internet Explorer, konieczne może być pobranie dodatkowego skryptu *system-polyfills.js*.

Sieci dostarczania treści (CDN)

Sieć unpkg (<https://unpkg.com>) to sieć CDN dla pakietów opublikowanych w rejestrze menedżera pakietów npm (<https://www.npmjs.com/>). Wejdź na stronę <https://www.npmjs.com>, aby znaleźć najnowszą wersję konkretnego pakietu. Jeśli chcesz zobaczyć, jakie inne wersje pakietu są dostępne, uruchom polecenie `npm info nazwa_pakietu`.

Wygenerowane pliki nie są umieszczane w systemie kontroli wersji, a Angular 2 nie zawiera gotowych do użycia paczek w swoim repozytorium Git. Są one generowane w locie i publikowane wraz z pakietem npm (<https://www.npmjs.com/~angular>), dzięki czemu można używać unpkg, aby bezpośrednio odwoływać się do gotowych do użycia w środowisku produkcyjnym paczek w plikach HTML. Zamiast tego wolimy używać lokalnej instalacji frameworku Angular i jego zależności, więc zainstaluj je za pomocą npm w punkcie 2.4.2. Wszystko, co zostanie zainstalowane przez npm, będzie przechowywane w katalogu `node_modules` w każdym projekcie.

PLIK TYPESCRIPT

Utwórzmy teraz plik *main.ts*, który zawiera kod TypeScript/Angular i dzieli się na trzy części:

1. zadeklarowanie komponentu HelloWorld,
2. opakowanie go w moduł,
3. załadowanie modułu.

W dalszej części rozdziału zaimplementujemy te części w trzech oddzielnych plikach, ale tutaj dla uproszczenia będziemy przechowywać cały kod tej niewielkiej aplikacji w jednym pliku, który został pokazany w listingu 2.2.

Listing 2.2. Plik main.ts aplikacji TypeScript

```

import {Component} from '@angular/core';
import { NgModule }    from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

// Komponent
@Component({
  selector: 'hello-world',
  template: '<h1>Witaj, {{ name }}!</h1>'
})
class HelloWorldComponent {
  name: string;
  constructor() {
    this.name = 'Angular 2';
  }
}

// Moduł
@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ HelloWorldComponent ],
  bootstrap:    [ HelloWorldComponent ]
})
export class AppModule {}

// Funkcja inicjująca aplikację
platformBrowserDynamic().bootstrapModule(AppModule);
```

Importuje metodę bootstrap i adnotację @component z odpowiednich pakietów Angular, dzięki czemu stają się one dostępne dla kodu aplikacji.

Adnotacja @Component umieszczona nad klasą HelloWorldComponent zamienia ją w komponent frameworku Angular.

Właściwość template definiuje znaczniki HTML dla renderowania tego komponentu.

Właściwość name jest używana w wyrażeniu wiązania danych w szablonie komponentu.

Adnotowana klasa HelloWorldComponent reprezentuje komponent.

Wewnątrz konstruktora inicjujemy właściwość name z wartością Angular 2 dowiązaną do szablonu.

Deklaruje zawartość modułu.

Deklaruje klasę reprezentującą moduł.

Ładuje moduł.

Adnotacje @Component i @NgModule wprowadzimy w podrozdziale 2.2.

Każdy komponent aplikacji można umieścić w pliku HTML (lub szablonie innego komponentu), używając znacznika, który odpowiada nazwie komponentu we właściwości selector adnotacji @Component. Selektory komponentów są podobne do selektorów CSS, więc jeśli mamy dany selektor 'hello-world', będziemy renderować ten komponent na stronie HTML z elementem o nazwie <hello-world>. Angular przekonwertuje tę linię na `document.querySelectorAll(selector)`.

Zwróć uwagę, że w listingu 2.2 cały szablon jest opakowany w znaki apostrofu, aby zamienić ten szablon w łańcuch znaków. W ten sposób można używać pojedynczych i podwójnych cudzysłówów wewnątrz szablonu i łączyć go na wiele linii dla lepszego

Czym są metadane?

Ogólnie rzecz biorąc, metadane to dodatkowe informacje na temat danych. Przykładowo w pliku MP3 dźwięk to dane, a nazwa wykonawcy, tytuł utworu i okładka albumu to metadane. Odtwarzacz MP3 zawiera procesor metadanych, który czyta metadane i wyświetla niektóre z nich podczas odtwarzania utworu.

W przypadku klas metadane stanowią dodatkowe informacje o klasie. Przykładowo dekorator `@Component` (czyli adnotacja) informuje framework Angular (procesor metadanych), że nie jest to zwykła klasa, ale komponent. Angular generuje dodatkowy kod JavaScript oparty na informacjach dostarczonych we właściwości dekoratora `@Component`.

W przypadku właściwości klasy dekorator `@Input` informuje Angular, że ta właściwość klasy powinna obsługiwać wiązanie i być w stanie odbierać dane z komponentu nadrzędnego.

Dekorator można również potraktować jak funkcję, która dołącza pewne dane do elementu opatrzonego dekoratorem. Dekorator `@Component` nie zmienia udekorowanej klasy, ale dodaje pewne dane opisujące tę klasę, dzięki czemu kompilator Angular może właściwie wygenerować końcowy kod komponentu w pamięci przeglądarki (kompilacja dynamiczna) lub w pliku na dysku (kompilacja statyczna).

formatowania. Ten szablon zawiera wyrażenie wiązania danych `{{name}}` i w czasie wykonywania Angular znajdzie właściwość `name` w komponentcie oraz zastąpi wyrażenie wiązania danych umieszczone w nawiasach klamrowych konkretną wartością.

We wszystkich przykładach kodu w tej książce będziemy używać języka TypeScript. Wyjątek stanowią dwie wersje aplikacji *Witaj, świecie*, które pokażemy za chwilę. Pierwszy przykład to zademonstrowanie wersji ES5, a drugi został napisany w ES6.

2.1.2. Witaj, świecie w ES5

Aby utworzyć aplikację w ES5, należy użyć specjalnej paczki Angular dystrybuowanej w formacie Universal Module Definition (UMD) — zwróć uwagę na fragment *umd* w adresach URL. Publikuje ona wszystkie interfejsy API frameworku w globalnym obiekcie `ng`. Plik HTML aplikacji Angular *Witaj, świecie* napisanej w ES5 może wyglądać tak, jak pokazano w listingu 2.3 (zobacz folder *hello-world-es5*).

Listing 2.3. Plik `index.html` aplikacji ES5

```
<!DOCTYPE html>
<html>
<head>
  <script src="//unpkg.com/zone.js@0.6.12/dist/zone.js"></script>
  <script src="//unpkg.com/rxjs@5.0.0-beta.11/bundles/Rx.umd.js"></script>
  <script src="//unpkg.com/core-js/client/shim.min.js"></script>
  <script src="//unpkg.com/@angular/core@2.0.0/bundles/core.umd.js"></script>
  <script src="//unpkg.com/@angular/common@2.0.0/bundles/common.umd.js"></script>
  <script src="//unpkg.com/@angular/compiler@2.0.0/bundles/compiler.umd.js"></script>
  <script src="//unpkg.com/@angular/platform-browser@2.0.0/bundles/
    ↳platform-browser.umd.js"></script>
  <script src="//unpkg.com/@angular/platform-browser-dynamic@2.0.0/bundles/
    ↳platform-browser-dynamic.umd.js"></script>
</head>
<body>
<hello-world></hello-world>
```

```
<script src="main.js"></script>
</body>
</html>
```

Ponieważ ES5 nie obsługuje składni adnotacji i nie ma natywnego systemu modułów, plik *main.js* powinien być napisany inaczej niż jego wersja TypeScript. Pokazano go w listingu 2.4.

Listing 2.4. Plik *main.js* aplikacji ES5

```
// Komponent
(function(app) {
  app.HelloWorldComponent =
    ng.core.Component({
      selector: 'hello-world',
      template: '<h1>Witaj, {{name}}!</h1>'
    })
    .Class({
      constructor: function() {
        this.name = 'Angular 2';
      }
    });
})(window.app || (window.app = {}));

// Modul
(function(app) {
  app.AppModule =
    ng.core.NgModule({
      imports: [ ng.platformBrowser.BrowserModule ],
      declarations: [ app.HelloWorldComponent ],
      bootstrap: [ app.HelloWorldComponent ]
    })
    .Class({
      constructor: function() {}
    });
})(window.app || (window.app = {}));

// Funkcja inicjująca aplikację
(function(app) {
  document.addEventListener('DOMContentLoaded', function() {
    ng.platformBrowserDynamic
      .platformBrowserDynamic()
      .bootstrapModule(app.AppModule);
  });
})(window.app || (window.app = {}));
```

Pierwsze natychmiastowo wywoływane wyrażenie funkcyjne (ang. *immediately invoked function expression* — IIFE) wywołuje metody `Component()` i `Class` w globalnej, podstawowej przestrzeni nazw Angular, `ng.core`. Definiujemy obiekt `HelloWorldComponent`, a metoda `Component` przyłącza metadane definiujące jego selektor i szablon. W ten sposób przekształcamy obiekt JavaScript w komponent wizualny.

Logika biznesowa komponentu jest zakodowana wewnątrz metody `Class`. W tym przypadku deklarujemy i inicjujemy właściwość `name` związaną z szablonem komponentu.

Drugie wyrażenie IIFE wywołuje metodę `NgModule`, aby utworzyć moduł, który deklaruje komponent `HelloWorldComponent` i określa go jako komponent główny poprzez przypisanie jego nazwy do właściwości `bootstrap`. Wreszcie trzecie wyrażenie IIFE uruchamia aplikację, wywołując metodę `bootstrapModule()`, która ładuje moduł, tworzy instancję `HelloWorldComponent` i dołącza ją do struktury DOM przeglądarki.

2.1.3. Witaj, świecie w ES6

Wersja ES6 aplikacji *Witaj, świecie* wygląda bardzo podobnie do wersji TypeScript, ale wykorzystuje Traceur jako transpiler dla SystemJS. Plik `index.html` wygląda tak, jak pokazano w listingu 2.5.

Listing 2.5. Plik `index.html` aplikacji ES6

```
<!DOCTYPE html>
<html>
<head>
  <script src="//unpkg.com/zone.js@0.6.21"></script>
  <script src="//unpkg.com/reflect-metadata@0.1.3"></script>
  <script src="//unpkg.com/traceur@0.0.111/bin/traceur.js"></script>
  <script src="//unpkg.com/systemjs@0.19.37/dist/system.src.js"></script>
  <script>
    System.config({
      transpiler: 'traceur',
      traceurOptions: {annotations: true},
      map: {
        'rxjs': 'https://unpkg.com/rxjs@5.0.0-beta.12',

        '@angular/core'           : 'https://unpkg.com/@angular/core@2.0.0',
        '@angular/common'        : 'https://unpkg.com/@angular/common@2.0.0',
        '@angular/compiler'       : 'https://unpkg.com/@angular/compiler@2.0.0',
        '@angular/platform-browser' : 'https://unpkg.com/@angular/platform-
        ↪browser@2.0.0',
        '@angular/platform-browser-dynamic': 'https://unpkg.com/@angular/platform-browser-
        ↪dynamic@2.0.0'
      },
      packages: {
        '@angular/core'           : {main: 'index.js'},
        '@angular/common'        : {main: 'index.js'},
        '@angular/compiler'       : {main: 'index.js'},
        '@angular/platform-browser' : {main: 'index.js'},
        '@angular/platform-browser-dynamic': {main: 'index.js'}
      }
    });
    System.import('main.js');
  </script>
</head>
<body>
  <hello-world></hello-world>
</body>
</html>
```

ES6 nie jest w pełni obsługiwany przez wszystkie przeglądarki i wykorzystujemy Traceur do transkompilowania (w przeglądarce) kodu ES6 na wersję ES5.

Rozszerzeniem pliku skryptu jest teraz .js.

Jedyną różnicą między plikiem *main.js* ES6 pokazanym w listingu 2.6 a plikiem *main.ts* TypeScript jest to, że teraz nie mamy predeklarowanej składowej klasy *name*.

Listing 2.6. Plik *main.js* aplikacji ES6

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

// Komponent
@Component({
  selector: 'hello-world',
  template: '<h1>Witaj, {{ name }}!</h1>'
})
class HelloWorldComponent {

  constructor() {
    this.name = 'Angular 2';
  }
}

// Modul
@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ HelloWorldComponent ],
  bootstrap:    [ HelloWorldComponent ]
})
export class AppModule { }

// Funkcja inicjująca aplikację
platformBrowserDynamic().bootstrapModule(AppModule);
```

2.1.4. Uruchamianie aplikacji

Aby uruchomić dowolną aplikację internetową, potrzebujemy podstawowego serwera HTTP, takiego jak *http-server* lub *live-server*. Ten ostatni odświeża stronę internetową na bieżąco, gdy tylko zmodyfikujesz kod i zapiszesz plik uruchomionej aplikacji.

Aby zainstalować *http-server*, użyj następującego polecenia *npm*:

```
npm install http-server -g
```

Aby uruchomić serwer z poziomu wiersza poleceń w głównym katalogu projektu, użyj polecenia:

```
http-server
```

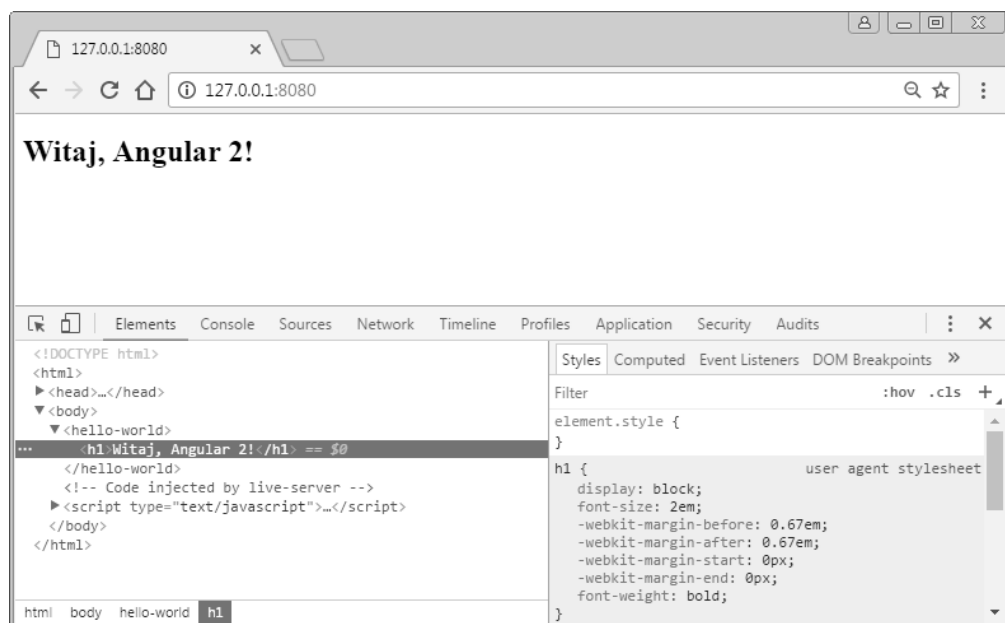
Wolimy odświeżanie okna przeglądarki na bieżąco, więc zainstaluj i uruchom *live-server* przy użyciu podobnej procedury:

```
npm install live-server -g
```

```
live-server
```

Jeśli używasz serwera `http-server`, musisz ręcznie otworzyć przeglądarkę internetową i wpisać adres URL `http://localhost:8080`, podczas gdy `live-server` otworzy przeglądarkę za Ciebie.

Aby uruchomić aplikację *Witaj, świecie*, uruchom serwer `live-server` w katalogu głównym projektu. W przeglądarce zostanie załadowany plik `index.html`. Powinieneś zobaczyć wyświetlony na stronie komunikat *Witaj, Angular 2!* (zobacz rysunek 2.1). W panelu narzędzi dla programistów przeglądarki możesz zobaczyć, że szablon określony dla `HelloWorldComponent` staje się zawartością elementu `<hello-world>`, a wyrażenie wiązania danych jest zastępowane rzeczywistą wartością użytą do zainicjowania właściwości `name` w konstruktorze komponentu.



Rysunek 2.1. Uruchomienie aplikacji Witaj, świecie

2.2. Elementy konstrukcyjne aplikacji Angular

W tym podrozdziale przedstawimy ogólny przegląd najważniejszych elementów konstrukcyjnych aplikacji Angular, żebyś mógł czytać kod Angular ze zrozumieniem. W kolejnych rozdziałach omówimy każdy z tych tematów szczegółowo.

2.2.1. Moduły

Moduł frameworku Angular to kontener dla grupy powiązanych komponentów, usług, dyrektyw i innych. Można potraktować moduł jako bibliotekę komponentów i usług implementujących określoną funkcjonalność z dziedziny biznesowej, której dotyczy aplikacja, taką jak moduł spedycyjny lub moduł rozliczeniowy. Wszystkie elementy niewielkiej aplikacji można umieścić w jednym module (module głównym), podczas gdy większe

aplikacje mogą mieć więcej niż jeden moduł (moduły funkcyjne). Wszystkie aplikacje muszą mieć co najmniej moduł główny, który jest pobierany podczas uruchamiania aplikacji.

UWAGA Moduły ES6 umożliwiają ukrycie i ochronę funkcji lub zmiennych oraz tworzenie skryptów, które można załadować. Z kolei moduły Angular są używane do spakowywania powiązanych ze sobą funkcjonalności aplikacji.

Z perspektywy składni moduł jest klasą adnotowaną dekoratorem `NgModule` i może zawierać inne zasoby. W podrozdziale 2.1 używaliśmy już modułu, który wyglądał następująco:

```
@NgModule({
  imports: [ BrowserModule ],
  declarations: [ HelloWorldComponent ],
  bootstrap: [ HelloWorldComponent ]
})
export class AppModule { }
```

Każda aplikacja przeglądarkowa musi importować BrowserModule i może, jeśli trzeba, importować inne moduły, takie jak FormsModule.

Deklaruje, że HelloWorldComponent należy do AppModule. Każdą składową modułu musi być tu wymieniona.

Podczas uruchamiania aplikacji ten moduł renderuje komponent główny, który jest przypisany do właściwości bootstrap dekoratora @NgModule.

Importowanie `BrowserModule` jest w module głównym koniecznością, ale jeśli Twoja aplikacja będzie składać się z modułu głównego i modułów funkcyjnych, te drugie będą musiały zamiast tego importować `CommonModule`. Składowe wszystkich importowanych modułów (takie jak `FormsModule` i `RouterModule`) są dostępne dla wszystkich komponentów modułu.

Aby załadować i skompilować moduł podczas uruchamiania aplikacji, należy wywołać moduł `bootstrapModule`:

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

Moduły aplikacji można łączyć natychmiast (gorliwie), tak jak w poprzednim fragmencie kodu, lub leniwie (jeśli trzeba) przez router (zobacz rozdział 3.). W każdym rozdziale książki będziemy używać `@NgModule`, więc będziesz miał okazję zobaczyć, jak deklarować moduły z wieloma składowymi. Szczegółowy opis modułów frameworku Angular można znaleźć w dokumentacji umieszczonej na stronie <https://angular.io/docs/ts/latest/guide/ngmodule.html>.

2.2.2. Komponenty

Głównym blokiem konstrukcyjnym aplikacji Angular jest **komponent**. Każdy komponent składa się z dwóch części — z widoku definiującego interfejs użytkownika i z klasy implementującej logikę, która kryje się za widokiem.

Każda aplikacja Angular reprezentuje hierarchię komponentów spakowanych w moduły. Aplikacja musi mieć co najmniej jeden moduł i jeden komponent, nazywany **komponentem głównym**. W komponencie głównym w porównaniu z innymi komponentami nie ma nic szczególnego. Każdy komponent przypisany do właściwości `bootstrap` modułu staje się komponentem głównym.

Aby utworzyć komponent, zadeklaruj klasę i dołącz do niej adnotację `@Component`:

```
@Component({
  selector: 'app-component',
  template: '<h1>Witaj !</h1>'
})
class HelloWorldComponent {}
```

Każda adnotacja `@Component` musi definiować właściwości `selector` i `template` (lub `templateUrl`), które determinują sposób wykrywania i renderowania komponentu na stronie.

Właściwość `selector` jest podobna do selektora CSS. Każdy element HTML, który odpowiada selektorowi, jest renderowany jako komponent Angular. Dekorator `@Component` można potraktować jako funkcję konfiguracyjną, która uzupełnia klasę. Jeśli przyjrzymy się transkompilowanemu kodowi pliku *main.ts* z listingu 2.2, zobaczysz, co kompilator Angular zrobił z dekoratorem `@Component`:

```
var core_1;
var HelloWorldComponent;

HelloWorldComponent = (function () {
  function HelloWorldComponent() {
    this.name = 'Angular 2';
  }
  HelloWorldComponent = __decorate([
    core_1.Component({
      selector: 'hello-world',
      template: '<h1>Witaj, {{ name }}!</h1>'
    }),
    __metadata('design:paramtypes', [])
  ], HelloWorldComponent);
  return HelloWorldComponent;
})();
```

Każdy komponent musi definiować widok, który jest określony we właściwości `template` lub `templateUrl` dekoratora `@Component`:

```
@Component({
  selector: 'app-component',
  template: '<h1>Komponent aplikacji</h1>' })
class AppComponent {}
```

W przypadku aplikacji internetowych `template` zawiera znaczniki HTML. Do renderowania natywnych aplikacji mobilnych można również użyć innego języka znaczników dostarczonego przez zewnętrzne frameworki. Jeśli kod znaczników składa się z nie więcej niż kilkudziesięciu linii, możemy utrzymywać go lokalnie przy użyciu właściwości `template`. W poprzednim przykładzie nie korzystaliśmy ze znaków grawis, ponieważ była to pojedyncza linia znaczników i nie zawierała pojedynczych lub podwójnych cudzysłówów. Obszerniejszy kod znaczników HTML powinien znajdować się w osobnym pliku HTML, do którego odwołuje się `templateUrl`.

Komponenty są stylizowane za pomocą standardowego kodu CSS. Można użyć właściwości `styles` dla lokalnego kodu CSS i właściwości `styleUrls` dla zewnętrznego pliku ze stylami. Pliki zewnętrzne pozwalają projektantom aplikacji internetowych pracować nad stylami bez modyfikowania kodu aplikacji. Ostatecznie decyzja dotycząca miejsca przechowywania kodu HTML lub CSS należy do Ciebie.

Widok można potraktować jako wynik scalenia układu interfejsu użytkownika z danymi. Fragment kodu AppComponent nie zawiera żadnych danych do scalenia, ale wersja TypeScript aplikacji *Witaj, świecie* (zobacz plik *main.ts* w listingu 2.2) będzie scalać znaczniki HTML z wartością zmiennej *name* w celu utworzenia widoku.

UWAGA We frameworku Angular renderowanie widoku jest oddzielone od komponentów, więc szablon może reprezentować charakterystyczny dla platformy natywny interfejs użytkownika, na przykład NativeScript (<https://www.nativescript.org>) lub React Native (<https://facebook.github.io/react-native>).

2.2.3. Dyrektywy

Dekorator @Directive umożliwia przypisanie niestandardowego zachowania do elementu HTML (można na przykład dodać funkcję autouzupełniania do elementu <input>). Każdy komponent jest zasadniczo dyrektywą z powiązaniem widokiem, ale w przeciwieństwie do komponentu dyrektywa nie ma własnego widoku.

Poniższy przykład przedstawia dyrektywę, która może być dołączona do elementu wejściowego w celu rejestrowania wartości wejściowej w konsoli przeglądarki, gdy tylko ta wartość zostanie zmieniona:

```
@Directive({
  selector: 'input[log-directive]',
  host: {
    '(input)': 'onInput($event)'
  }
})
class LogDirective {
  onInput(event) {
    console.log(event.target.value);
  }
}
```

Ten selektor wymaga, żeby docelowy element HTML miał element input i atrybut log-directive.

Element host jest tym, do którego dołączamy dyrektywę.

Procedura obsługi dla elementu <input> rejestruje jego wartość w konsoli.

Aby powiązać zdarzenia z procedurami obsługi zdarzeń, nazwę danego zdarzenia trzeba umieścić w nawiasach. Kiedy wystąpi zdarzenie input w elemencie host, wywołana zostanie procedura obsługi zdarzeń onInput(), a obiekt zdarzenia zostanie przekazany do tej metody jako argument.

Oto przykład sposobu dołączania dyrektywy do elementu HTML:

```
<input type="text" log-directive/>
```

Następny przykład przedstawia dyrektywę, która zmienia kolor tła dołączonego elementu na niebieski:

```
import { Directive, ElementRef, Renderer } from '@angular/core';

@Directive({ selector: '[highlight]' })

export class HighlightDirective {
  constructor(renderer: Renderer, el: ElementRef) {
    renderer.setStyle(el.nativeElement, 'backgroundColor', 'blue');
  }
}
```

Ta dyrektywa może być dołączona do różnych elementów HTML, a konstruktor tej dyrektywy uzyskuje referencje do renderera i elementu interfejsu użytkownika wstrzykniętego przez Angular. Oto sposób, w jaki można dołączyć tę dyrektywę do elementu `<h1>` HTML:

```
<h1 highlight>Witaj, świecie</h1>
```

Wszystkie dyrektywy, które są używane w module, muszą być dodane do deklaracji właściwości dekoratora `@NgModule`, tak jak w tym przykładzie:

```
@NgModule({
  imports: [ BrowserModule ],
  declarations: [ HelloWorldComponent,
                  HighlightDirective ],
  bootstrap: [ HelloWorldComponent ]
})
```

2.2.4. Krótkie wprowadzenie do wiązania danych

Framework Angular ma mechanizm zwany **wiązaniem danych** (ang. *data binding*), który pozwala utrzymywać synchronizację właściwości komponentu z widokiem. Ten mechanizm jest dość wyrafinowany i szczegółowo omówimy go w rozdziale 5. W tym punkcie opiszemy tylko najczęstsze formy składni wiązania danych.

Aby wyświetlić w szablonie wartość jako łańcuch znaków, skorzystaj z podwójnych nawiasów klamrowych:

```
<h1>Witaj, {{ name }}!</h1>
```

Jeśli chcesz powiązać właściwość elementu HTML z wartością, użyj nawiasów kwadratowych:

```
<span [hidden]="isValid">To pole jest wymagane</span>
```

Żeby powiązać procedurę obsługi zdarzeń ze zdarzeniem elementu, zastosuj nawiasy:

```
<button (click)="placeBid()">Złóż ofertę</button>
```

Kiedy chcesz odwołać się do właściwości obiektu DOM w szablonie, dodaj lokalną zmienną szablonu (jej nazwa musi zaczynać się od znaku #), która automatycznie zapisze odwołanie do odpowiedniego obiektu DOM, i użyj notacji kropkowej:

```
<input #title type="text" />
<span>{{ title.value }}</span>
```

Ponieważ wiesz już, jak napisać prostą aplikację Angular, zobaczmy, w jaki sposób można załadować kod do przeglądarki za pomocą biblioteki SystemJS.

2.3. Uniwersalna ładowarka modułów SystemJS

Większość istniejących aplikacji internetowych ładuje pliki JavaScript do strony HTML przy użyciu znaczników `<script>`. Chociaż można dodać kod Angular do strony w ten sam sposób, zalecanym sposobem jest jednak załadowanie kodu przy użyciu biblioteki SystemJS. Framework Angular używa również SystemJS wewnętrznie.

W tym podrozdziale omówimy pokrótce SystemJS, żebyś mógł zacząć tworzyć aplikacje Angular. Szczegółowy przewodnik dla SystemJS znajdziesz na stronie GitHub tej biblioteki: <https://github.com/systemjs/systemjs>.

2.3.1. Przegląd ładowarek modułów

Ostateczna wersja specyfikacji ES6 wprowadza moduły i obejmuje ich składnię oraz semantykę (<http://mng.bz/ri01>). Wczesne projekty specyfikacji zawierały definicję globalnego obiektu `System` odpowiedzialnego za ładowanie modułów do środowiska wykonawczego bez względu na to, czy jest to przeglądarka internetowa, czy samodzielny proces. Jednak definicja obiektu `System` została usunięta z finalnej wersji specyfikacji ES6 i jest obecnie śledzona przez Grupę Roboczą ds. Technologii Hipertekstowych Aplikacji Sieciowych (WHATWG — zobacz <http://whatwg.github.io/loader>). Obiekt `System` może stać się częścią specyfikacji ES8.

Polyfill ES6 Module Loader (<https://github.com/ModuleLoader/es6-module-loader>) oferuje obecnie jeden sposób na użycie obiektu `System` (bez oczekiwania na przyszłe specyfikacje ECMAScript). Stara się dopasować do przyszłych standardów, ale ten polyfill obsługuje tylko moduły ES6.

Ponieważ specyfikacja ES6 jest dość nowa, większość zewnętrznych pakietów przechowywanych w rejestrze NPM nie używa jeszcze modułów ES6. W pierwszych dziesięciu rozdziałach tej książki używamy ładowarki SystemJS, która zawiera nie tylko ES6 Module Loader, ale umożliwia również ładowanie modułów napisanych w AMD, CommonJS, UMD i globalnych formatach modułów. Obsługa tych formatów jest całkowicie przezroczysta dla użytkownika SystemJS, ponieważ ładowarka ta automatycznie rozpoznaje, z jakiego formatu modułów korzysta skrypt docelowy. W rozdziale 10. użyjemy innego modułu ładowarki o nazwie Webpack.

2.3.2. Porównanie ładowarek modułów i znaczników `<script>`

Po co w ogóle używać ładowarek modułów? Można ładować JavaScript za pomocą znacznika `<script>`? Znaczniki `<script>` są problematyczne.

- Programista jest odpowiedzialny za utrzymywanie znaczników `<script>` w pliku HTML. Z czasem niektóre z nich mogą stać się zbędne, ale jeśli zapomnisz je wyczyścić, wciąż będą ładowane przez przeglądarkę, zwiększając czas ładowania i marnując przepustowość sieci.
- Często kolejność ładowania skryptów ma znaczenie. Przeglądarki mogą zagwarantować kolejność wykonywania skryptów tylko wtedy, jeśli umieścisz znaczniki `<script>` w sekcji `<head>` dokumentu HTML. Jednak umieszczanie wszystkich skryptów w sekcji `<head>` jest uważane za złą praktykę, ponieważ uniemożliwia renderowanie strony, dopóki nie zostaną pobrane wszystkie skrypty.

Weźmy pod uwagę zalety korzystania z ładowarek modułów zarówno podczas rozwijania aplikacji, jak i w trakcie przygotowywania jej wersji produkcyjnej.

- W środowiskach programowania kod jest zazwyczaj podzielony na wiele plików, a każdy plik reprezentuje moduł. Za każdym razem, gdy importujemy moduł w kodzie, ładowarka będzie dopasowywać nazwę modułu do odpowiedniego pliku, pobierać go do przeglądarki, a następnie wykonywać resztę kodu. Moduły pozwalają utrzymać dobrą organizację projektów. Ładowarka modułów automatycznie składa wszystko razem w przeglądarce podczas uruchamiania aplikacji. Jeśli moduł zawiera zależności od innych modułów, wszystkie te moduły zostaną załadowane.
- Gdy przygotowujemy wersję produkcyjną aplikacji, ładowarka modułów pobiera plik główny, przechodzi przez drzewo wszystkich modułów osiągalnych z tego pliku i łączy je wszystkie w pojedynczą paczkę. W ten sposób paczka zawiera tylko kod faktycznie używany przez aplikację. Rozwiązuje również problem z kolejnością ładowania skryptów i referencjami cyklicznymi.

Korzyści te odnoszą się nie tylko do kodu aplikacji, ale również do pakietów zewnętrznych (takich jak Angular).

UWAGA W tej książce używamy terminów *moduł* i *plik* zamiennie. Moduł nie może obejmować wielu plików. Paczka jest zwykle reprezentowana przez pojedynczy plik i zawiera wiele zarejestrowanych w nim modułów. Kiedy różnica między *modulem* i *plikiem* będzie istotna, wyraźnie to zaznaczymy.

2.3.3. Pierwsze kroki z SystemJS

Kiedy używamy SystemJS na stronie HTML, biblioteka ta staje się dostępna jako globalny obiekt `System`, który ma wiele metod statycznych. Dwie podstawowe metody, których będziemy używać, to `System.import()` i `System.config()`.

Aby załadować moduł, użyj metody `System.import()`, która przyjmuje nazwę modułu jako argument. Nazwa modułu może być ścieżką do pliku lub nazwą logiczną zmapowaną na ścieżkę do pliku:

```
System.import('./mój-moduł.js'); ← Ścieżka do pliku.  
System.import('@angular2/core'); ← Nazwa logiczna.
```

Jeśli nazwa modułu zaczyna się od znaków `./`, jest to ścieżka do pliku nawet wtedy, jeśli rozszerzenie nazwy zostanie pominięte. SystemJS najpierw próbuje dopasować nazwę modułu do skonfigurowanego mapowania dostarczonego jako argument metody `System.config()` lub w pliku (na przykład *systemjs.config.js*). Jeśli mapowanie dla nazwy nie zostanie znalezione, rozważana jest ścieżka do pliku.

UWAGA W tej książce do wskazywania plików do załadowania będziemy używać zarówno prefiksu `./`, jak i konfiguracji mapowania. Jeśli zobaczysz metodę `System.import('app')` i nie będziesz mógł odnaleźć pliku o nazwie *app.ts*, sprawdź konfigurację mapowania projektu.

Metoda `System.import()` natychmiast zwraca obiekt obietnicy Promise (zobacz dodatek A). Kiedy obietnica zostanie rozwiązana na obiekt modułu, podczas ładowania modułu

wywoływane jest wywołanie zwrotne `then()`. Jeśli obietnica zostanie odrzucona, błędy są obsługiwane w metodzie `catch()`.

Obiekt modułu ES6 zawiera właściwość dla każdej wyeksportowanej wartości w załadowanym module. Poniższy fragment kodu z dwóch plików pokazuje, jak można wyeksportować zmienną z modułu i użyć jej w innym skrypcie:

```
// lib.js
export let foo = 'foo';

// main.js
System.import('./lib.js').then(libModule => {
  libModule.foo === 'foo'; // true
});
```

Używamy tutaj metody `then()` w celu określenia funkcji wywołania zwrotnego, która ma być wywołana, gdy załadowany zostanie *lib.js*. Załadowany obiekt jest przekazywany jako argument do wyrażenia strzałkowego (ang. *fat arrow expression*).

W skryptach ES5 używamy metody `System.import()` do załadowania kodu gorliwie lub leniwie (dynamicznie). Jeśli Twoją witrynę internetową przegląda na przykład anonimowy użytkownik, możesz nie potrzebować modułu, który implementuje funkcjonalność profilu użytkownika. Gdy jednak użytkownik zaloguje się, możesz dynamicznie załadować moduł profilu. W ten sposób zmniejszasz początkowy rozmiar i czas ładowania strony.

A co z instrukcjami `import` ES6? W naszej pierwszej aplikacji Angular użyliśmy metody `System.import()` w pliku *index.html*, aby załadować główny moduł aplikacji, czyli *main.ts*. Z kolei skrypt *main.ts* importował moduły Angular przy użyciu własnej instrukcji `import`.

Gdy SystemJS łączy moduł *main.ts*, automatycznie transkompiluje go na kod kompatybilny z ES5, więc nie ma w kodzie instrukcji `import`, które wykonują przeglądarki. W przyszłości, gdy moduły ES6 będą natywnie obsługiwane przez najważniejsze przeglądarki, ten krok nie będzie wymagany, a instrukcje `import` będą działać podobnie do metody `System.import()` z tym wyjątkiem, że nie będą kontrolować momentu ładowania modułu.

UWAGA Gdy SystemJS transkompiluje pliki, automatycznie generuje mapę źródeł dla każdego pliku *.js*, co pozwala debugować kod TypeScript w przeglądarce.

APLIKACJA DEMO

Weźmy pod uwagę aplikację, która musi załadować skrypty ES5 i ES6. Aplikacja będzie składać się z trzech plików (zobacz folder *systemjs-demo*):

```
├── index.html
├── es6module.js
└── es5module.js
```

W typowej aplikacji internetowej plik *index.html* zawierałby znaczniki `<script>` odwołujące się zarówno do *es6module.js*, jak i *es5module.js*. Każdy z tych plików byłby automatycznie ładowany i wykonywany przez przeglądarkę. Jednak to podejście powoduje

kilka problemów, które omówiliśmy w punkcie 2.3.2. Zobaczmy, jak rozwiązać te problemy za pomocą ładowarki SystemJS w aplikacji demo.

Używamy instrukcji `export ES6`, aby udostępnić nazwę modułu *es6module.js* poza skryptem. Obecność instrukcji `export` automatycznie zmienia plik w moduł ES6:

```
export let name = 'ES6';
```

```
console.log('Ładowany jest moduł ES6');
```

Plik *es5module.js* nie zawiera żadnej składni ES6 i używa formatu modułu CommonJS, aby wyeksportować nazwę modułu. Zasadniczo do obiektu `exports` dołączamy zmienne, które mają być widoczne poza modulem:

```
exports.name = 'ES5';
```

```
console.log('Ładowany jest moduł ES5');
```

Plik *index.html* z listingu 2.7 bez problemu importuje moduły CommonJS i ES6 za pomocą ładowarki SystemJS.

Listing 2.7. Plik *index.html* z SystemJS

```
<!DOCTYPE html>
<html>
<head>
  <script src="//unpkg.com/es6-promise@3.0.2/dist/es6-promise.js"></script>
  <script src="//unpkg.com/traceur@0.0.111/bin/traceur.js"></script>
  <script src="//unpkg.com/systemjs@0.19.37/dist/system.src.js"></script>
</script>

  Promise.all([
    System.import('./es6module.js'),
    System.import('./es5module.js')
  ]).then(function (modules) {
    var moduleNames = modules
      .map(function (m) { return m.name; })
      .join(', ');
    console.log('Załadowane zostały następujące moduły: ' + moduleNames);
  });
</script>
</head>
<body></body>
</html>
```

Metoda Promise.all() ES6 zwraca obiekt Promise, który zostaje rozwiązany (lub odrzucony), gdy ukończone zostaną wszystkie iterowalne argumenty.

Używamy tu ścieżki względnej do pliku es6module.js, który wykorzystuje składnię modułów ES6.

Ładuje es5module.js podobny do poprzedniego, ale tym razem SystemJS używa formatu CommonJS format.

Nie używamy tutaj funkcji strzałkowej ES6, ponieważ sam plik index.html nie jest przetwarzany przez SystemJS, więc kod nie byłby transkompilowany i nie działałby we wszystkich przeglądarkach.

Argumenty Promise.all() po załadowaniu są dostarczane do metody then() jako tablica modules.

Ta metoda map() wywołuje funkcję, która przekształca wynik poprzez wyodrębnienie właściwości name eksportowanej z każdego

Metoda join() łączy nazwy wszystkich modułów w łańcuch znaków rozdzielany przecinkami.

Ponieważ metoda `System.import()` zwraca obiekt `Promise`, można rozpocząć ładowanie wielu modułów jednocześnie i wykonać jakiś inny kod, gdy wszystkie moduły zostaną załadowane.

Po uruchomieniu aplikacji w konsoli przeglądarki zostanie wyświetlony następujący wynik (aby go zobaczyć, musisz mieć otwarty panel narzędzi dla programistów):

```
Live reload enabled.
Ładowany jest moduł ES6
Ładowany jest moduł ES5
Załadowane zostały następujące moduły: ES6, ES5
```

Pierwsza linia pochodzi z serwera `live-server`, a nie z aplikacji. Gdy tylko któryś z modułów zostanie załadowany, natychmiast wyświetla swój komunikat dziennika. Po załadowaniu wszystkich modułów wykonywana jest funkcja wywołania zwrotnego i wyświetlany ostatni komunikat dziennika.

KONFIGURACJA SYSTEMJS

Dotychczas używaliśmy domyślnej konfiguracji ładowarki `SystemJS`, ale można skonfigurować niemal każdy aspekt jej działania przy użyciu metody `System.config()`, która przyjmuje jako argument obiekt konfiguracyjny. Metoda `System.config()` może być wywoływana wielokrotnie z różnymi obiektami konfiguracyjnymi. Jeśli ta sama opcja jest ustawiana więcej niż raz, stosowana jest ostatnia wartość. Skrypt z `System.config()` można wstawić w pliku HTML lokalnie przy użyciu znacznika `<script>` (zobacz podrozdział 2.1) lub zapisać kod dla `System.config()` w osobnym pliku (takim jak *systemjs.config.js*) i załączyć go do pliku HTML za pomocą znacznika `<script>`.

Pełna lista opcji konfiguracyjnych `SystemJS` jest dostępna na stronie GitHub (<http://mng.bz/8N60>). Krótko omówimy niektóre z opcji konfiguracyjnych używanych w tej książce.

OPCJA BASEURL

Wszystkie moduły są ładowane względem tego adresu URL, chyba że nazwa modułu reprezentuje bezwzględny lub względny adres URL:

```
System.config({ baseURL: '/app' });
System.import('es6module.js'); // GET /app/es6module.js
System.import('./es6module.js'); // GET /es6module.js
System.import('http://example.com/es6module.js'); // GET http://example.com/es6module.js
```

OPCJA DEFAULTJSEXTENSIONS

Jeśli `defaultJSExtensions` ma wartość `true`, rozszerzenie `.js` będzie automatycznie dodawane do wszystkich ścieżek plików. Jeśli nazwa modułu zawiera już rozszerzenie inne niż `.js`, rozszerzenie `.js` i tak zostanie dodane:

```
System.config({ defaultJSExtensions: true });
System.import('./es6module'); // GET /es6module.js
System.import('./es6module.js'); // GET /es6module.js
System.import('./es6module.ts'); // GET /es6module.ts.js
```

OSTRZEŻENIE Właściwość `defaultJSExtensions` istnieje dla kompatybilności wstecznej i zostanie wycofana w przyszłych wersjach SystemJS.

OPCJA MAP

Opcja `map` umożliwia tworzenie aliasu dla nazwy modułu. Podczas importowania modułu jego nazwa zostanie zastąpiona przez powiązaną wartość, chyba że oryginalna nazwa modułu reprezentuje dowolną ścieżkę (bezwzględną lub względną). Parametr `map` jest stosowany przed `baseURL`:

```
System.config({ map: { 'es6module.js': 'esSixModule.js' } });
System.import('es6module.js'); // GET /esSixModule.js
System.import('./es6module.js'); // GET /es6Module.js
```

Oto inny przykład `map`:

```
System.config({
  baseURL: '/app',
  map: { 'es6module': 'esSixModule.js' }
});
System.import('es6module'); // GET /app/esSixModule.js
```

OPCJA PACKAGES

Opcja `packages` zapewnia wygodny sposób ustawiania metadanych i mapowania konfiguracji charakterystycznej dla typowej ścieżki. Przykładowo poniższy fragment kodu poprzez podanie jedynie nazwy pliku i domyślnego rozszerzenia `ts` dla TypeScript instruuje SystemJS, że metoda `System.import('app')` powinna załadować moduł zlokalizowany w pliku `main_router_sample.ts`:

```
System.config({
  packages: {
    app: {
      defaultExtension: "ts",
      main: "main_router_sample"
    }
  }
});
System.import('app');
```

OPCJA PATHS

Opcja `paths` jest podobna do `map`, ale obsługuje znaki wieloznaczne. Jest stosowana po `map`, ale przed `baseURL` (zobacz listing 2.6). Możesz używać obu opcji, `map` i `paths`, ale pamiętaj, że `paths` jest częścią specyfikacji Loader (zobacz <http://whatwg.github.io/loader>) i implementacji ES6 Module Loader (zobacz <https://github.com/ModuleLoader/es-module-loader>), a `map` jest rozpoznawalna tylko przez SystemJS:

```
System.config({
  baseURL: '/app',
  map: { 'es6module': 'esSixModule.js' },
  paths: { '*': 'lib/*' }
});

System.import('es6module'); // GET /app/lib/esSixModule.js
```

W wielu przykładach kodu w tej książce znajdziesz metodę `System.import('app')`, która otwiera plik z inną nazwą (czyli inną niż *app*), ponieważ skonfigurowana została właściwość `map` lub `packages`. Kiedy zobaczysz coś takiego jak `import {Component} from '@angular/core'`, to `@angular` będzie odnosiło się do nazwy zmapowanej na rzeczywisty katalog, w którym znajduje się framework Angular; natomiast `core` jest podkatalogiem, a główny plik w tym podkatalogu jest określany w konfiguracji `SystemJS`, tak jak w tym przykładzie:

```
packages: {
  '@angular/core' : {main: 'index.js'}
}
```

OPCJA TRANSPILER

Opcja `transpiler` umożliwia określenie nazwy modułu transkompilatora, który powinien być używany podczas ładowania modułów aplikacji. Jeśli plik nie zawiera co najmniej jednej instrukcji `import` lub `export`, nie będzie transkompilowany. Opcja `transpiler` może zawierać jedną z następujących wartości: `typescript`, `traceur` i `babel`.

```
System.config({
  transpiler: 'traceur',
  map: {
    traceur: '///unpkg.com/traceur@0.0.108/bin/traceur.js'
  }
});
```

OPCJA TYPESCRIPTOPTIONS

Opcja `typescriptOptions` umożliwia ustawienie opcji kompilatora TypeScriptu. Listę wszystkich dostępnych opcji można znaleźć w dokumentacji języka TypeScript <http://mng.bz/rf14>.

2.4. Wybór menedżera pakietów

Jest mało prawdopodobne, że będziesz pisać aplikację internetową bez użycia żadnych bibliotek. W tej książce będziemy korzystać z kilku różnych. Frameworku Angular będziemy używać w większości przykładów kodu, a w aplikacji aukcji internetowych będziemy korzystać również z biblioteki Twittera o nazwie `Bootstrap`, która ma `jQuery` jako zależność. Twoja aplikacja może wymagać konkretnych wersji tych zależności.

Ładowaniem bibliotek, frameworków i ich zależności zarządza menedżer pakietów i musisz zdecydować, którego z kilku popularnych menedżerów będziesz używać. Programiści JavaScript mogą być przytłoczeni różnorodnością dostępnych menedżerów pakietów, do których należą między innymi `npm`, `Bower`, `jspm`, `Jam` i `Duo`.

Typowy projekt zawiera plik konfiguracyjny z listą nazw i wersji wymaganych bibliotek i frameworków. Oto fragment z pliku konfiguracji `npm` *package.json*, którego będziemy używać w aplikacji aukcji internetowych:

```
"scripts": {
  "start": "live-server"
},
```

```

"dependencies": {
  "@angular/common": "2.0.0",
  "@angular/compiler": "2.0.0",
  "@angular/core": "2.0.0",
  "@angular/forms": "2.0.0",
  "@angular/http": "2.0.0",
  "@angular/platform-browser": "2.0.0",
  "@angular/platform-browser-dynamic": "2.0.0",
  "@angular/router": "3.0.0",

  "core-js": "^2.4.0",
  "rxjs": "5.0.0-beta.12",
  "systemjs": "0.19.37",
  "zone.js": "0.6.21",

  "bootstrap": "^3.3.6",
  "jquery": "^2.2.2"
},
"devDependencies": {
  "live-server": "0.8.2",
  "typescript": "^2.0.0"
}

```

Sekcja `scripts` określa polecenie, które zostanie uruchomione po wpisaniu `npm start` w wierszu poleceń. W tym przypadku chcemy uruchomić serwer `live-server`. Sekcja `dependencies` wymienia wszystkie zewnętrzne biblioteki i narzędzia wymagane przez środowisko uruchomieniowe, w którym aplikacja zostanie wdrożona.

Sekcja `devDependencies` dodaje narzędzia, które muszą być obecne na komputerze. Nie będziemy na przykład używać serwera `live-server` w środowisku produkcyjnym, ponieważ jest to dość prosty serwer, wystarczający tylko do celów rozwijania aplikacji. Powyższa konfiguracja wskazuje również, że kompilator TypeScriptu jest wymagany tylko podczas fazy rozwoju, a można się domyślić, że w czasie wdrażania cały kod TypeScript zostanie transkompilowany na JavaScript.

Powyższa konfiguracja zawiera także numery wersji. Jeśli przed numerem wersji umieszczony jest znak `^`, wskazuje on, że projekt wymaga określonej lub nowszej wersji pomocniczej tej biblioteki lub tego pakietu. Kiedy używaliśmy wersji beta frameworku Angular, chcieliśmy określić dokładną wersję pakietu, ponieważ nowsze wersje mogłyby zawierać pewne przełomowe zmiany.

Gdy zaczęliśmy współpracować z frameworkiem Angular, wiedzieliśmy, że będziemy korzystać z ładowarki modułów SystemJS. Potem dowiedzieliśmy się, że autor SystemJS (Guy Bedford) utworzył również menedżer pakietów `jspm`, który wewnętrznie wykorzystuje SystemJS, więc postanowiliśmy użyć `jspm`. Przez pewien czas używamy `npm` do instalowania narzędzi i `jspm` dla zależności aplikacji. Ta konfiguracja działała, ale w przypadku `jspm` przeglądarka internetowa wysyłała ponad 400 żądań do serwera, aby wyświetlić pierwszą stronę dość prostej aplikacji. Oczekiwanie 3,5 sekundy jedynie na uruchomienie aplikacji na lokalnej maszynie to trochę za długo.

Zdecydowaliśmy się wypróbować `npm` do zarządzania zależnościami podczas fazy rozwoju aplikacji. Wyniki były znacznie lepsze: tylko 30 żądań serwera i 1,5 sekundy na uruchomienie tej samej aplikacji.

Tak czy inaczej omówimy pokrótce oba menedżery pakietów i pokażemy, jak rozpocząć nowy projekt za pomocą każdego z nich. Menedżer `jspm` jest bardzo młody i z czasem może zostać ulepszony, ale w naszych projektach Angular zdecydowaliśmy się na użycie `npm`.

2.4.1. Porównanie `npm` i `jspm`

Menedżer `npm` jest menedżerem pakietów dla Node.js. Został utworzony do zarządzania modułami Node.js, które są zapisywane w formacie CommonJS. Format ten nie był przeznaczony dla aplikacji internetowych, ponieważ moduły miały być ładowane synchronicznie. Rozważmy następujący fragment kodu:

```
var x = require('module1');
var y = require('module2');
var z = require('module3');
```

Ładowanie modułu `module2` nie rozpocznie się, dopóki nie zostanie załadowany `module1`, a ładowanie `module3` poczeka na załadowanie modułu `module2`. W przypadku aplikacji desktopowych napisanych w Node.js jest to w porządku, ponieważ ładowanie odbywa się z lokalnego komputera, ale taki proces synchronicznego ładowania spowolniłby pobieranie aplikacji.

Kolejnym słabym punktem `npm` było to, że historycznie używał *zagnieżdżonych* zależności. Gdyby pakiety A i B zależały od pakietu C, każdy z nich przechowywałby kopię C w swoim katalogu, ponieważ A i B mogą zależeć od różnych wersji C. Chociaż było to w porządku w przypadku aplikacji Node.js, nie sprawdzało się za dobrze dla aplikacji ładowanych do przeglądarki internetowej. Problemy może powodować nawet dwukrotne ładowanie tej samej wersji biblioteki w przeglądarce. Jeśli ładowane są dwie różne wersje, szanse na przerwanie działania aplikacji są jeszcze wyższe.

Kwestia zagnieżdżonych zależności została uwzględniona w `npm 3`, ale problem został tylko częściowo rozwiązany. Domyślnie `npm` próbuje zainstalować pakiet C w tym samym katalogu, co A i B, więc pojedyncza kopia C jest współdzielona między A i B. Jeśli jednak A i B wymagają sprzecznych wersji C, `npm` wraca do podejścia opartego na zagnieżdżonych zależnościach. Biblioteki tworzone dla aplikacji po stronie klienta zawierają zwykle w pakietach `npm` skompilowane wersje (paczki z jednym plikiem). Te paczki nie zawierają zewnętrznych zależności, więc należy je załadować ręcznie na stronę. Pomaga to uniknąć problemu zagnieżdżonych zależności.

Menedżer `jspm` jest menedżerem pakietów utworzonym z uwzględnieniem modułów ES6 i ładowarek modułów. Sam nie hostuje żadnych pakietów. Ma koncepcję rejestrów, która pozwala tworzyć niestandardowe lokalizacje źródeł dla pakietów. Od ręki `jspm` umożliwia instalowanie pakietów z rejestru `npm` lub bezpośrednio z repozytoriów GitHub.

Jest przeznaczony do współpracy z SystemJS. Podczas inicjowania nowego projektu lub instalowania pakietu za pomocą `jspm` automatycznie tworzy konfigurację dla SystemJS w celu ładowania modułów. W przeciwieństwie do `npm` stosuje podejście oparte na *płaskich* zależnościach, więc zawsze jest tylko jedna kopia biblioteki w projekcie. Umożliwia to używanie instrukcji `import` nawet do ładowania zewnętrznego kodu. Rozwiązuje to

problem z kolejnością ładowania skryptów i zapewnia, że aplikacja będzie ładować tylko te moduły, których faktycznie używa.

Pakiety `jspm` zazwyczaj nie zawierają paczek. Zamiast tego zachowują oryginalną strukturę i pliki projektu, dzięki czemu każdy moduł może być ładowany indywidualnie. Chociaż posiadanie oryginalnej wersji pliku może poprawić debugowanie, w praktyce się nie opłaca. Importowanie każdego modułu indywidualnie prowadzi do ładowania setek plików w przeglądarce przed uruchomieniem aplikacji. Spowalnia to proces rozwoju aplikacji i nie jest odpowiednie dla wdrożenia do środowiska produkcyjnego.

Innym słabym punktem `jspm` jest to, że niekoniecznie można od ręki użyć dowolnego pakietu `npm` lub repozytorium `GitHub` jako pakietu `jspm`. Mogą one wymagać dodatkowej konfiguracji, aby `jspm` mógł prawidłowo ustawić `SystemJS` w celu załadowania modułów z pakietu. W czasie pisania tego rozdziału w rejestrze `jspm` jest niecałe 500 pakietów gotowych do współpracy z `SystemJS`, w porównaniu z 250 000 pakietów hostowanych przez `npm`.

2.4.2. Rozpoczynanie projektu Angular za pomocą npm

Aby uruchomić nowy projekt zarządzany przez `npm`, utwórz nowy katalog (na przykład *angular-seed*) i otwórz w nim wiersz poleceń. Następnie uruchom polecenie `npm init -y`, które utworzy początkową wersję pliku konfiguracyjnego *package.json*. Normalnie `npm init` zadaje kilka pytań podczas tworzenia pliku, ale flaga `-y` sprawia, że przyjmuje domyślne wartości dla wszystkich opcji. Poniższy przykład pokazuje wykonanie tego polecenia w pustym katalogu *angular-seed*.

```
$ npm init -y
Wrote to /Users/username/angular-seed/package.json:
{
  "name": "angular-seed",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Większa część wygenerowanej konfiguracji jest potrzebna do opublikowania projektu w rejestrze `npm` lub podczas instalacji pakietu jako zależności dla innego projektu. Menedżera `npm` będziemy używać tylko do zarządzania zależnościami projektu i automatyzacji procesów tworzenia i kompilowania.

Ponieważ nie będziemy publikować tego projektu w rejestrze `npm`, powinniśmy usunąć wszystkie właściwości z wyjątkiem `name`, `description` i `scripts`. Trzeba dodać również właściwość `"private": true`, ponieważ nie jest tworzona domyślnie. Zapobiegnie to przypadkowemu opublikowaniu pakietu w rejestrze `npm`. Plik *package.json* powinien wyglądać tak, jak widać w listingu 2.8.

Listing 2.8. Plik package.json

```
{
  "name": "angular-seed",
  "description": "Wstępny projekt zarządzany przez npm dla rozdziału 2.",
  "private": true,
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  }
}
```

Konfiguracja `scripts` umożliwia określenie poleceń, które mogą być uruchamiane w oknie wiersza poleceń. Domyślnie npm `init` tworzy polecenie `test`, które można uruchomić, wpisując `npm test`. Zastąpmy je poleceniem `start`, którego będziemy używać do uruchamiania serwera `live-server` zainstalowanego w punkcie 2.4.1. Oto konfiguracja właściwości `scripts`:

```
{
  ...
  "scripts": {
    "start": "live-server"
  }
}
```

Możesz uruchomić dowolne polecenie npm z sekcji `scripts` za pomocą składni `npm run moje_polecenie`, na przykład `npm run start`. Można również użyć skróconej wersji polecenia `npm start` zamiast `npm run start`. Składnia skrótowa jest dostępna tylko dla predefiniowanych skryptów npm (zapoznaj się z dokumentacją npm na stronie <https://docs.npmjs.com/misc/scripts>).

Teraz chcemy, żeby npm pobrał Angular dla tego projektu jako zależność. W wersji TypeScript aplikacji *Witaj, świecie* używaliśmy kodu Angular umieszczonego na serwerze `unpkg` CDN, ale tutaj chcemy pobrać go do katalogu projektu. Chcemy mieć także lokalne wersje ładowarki `SystemJS`, serwera `live-server` i kompilatora TypeScriptu.

Pakiety npm często składają się ze zoptymalizowanych do wykorzystania w środowisku produkcyjnym paczek, które nie zawierają kodu źródłowego bibliotek. Dodajmy do pliku `package.json` sekcję, która używa kodu źródłowego (a nie zoptymalizowanych paczek) konkretnych pakietów. Dodaj tę sekcję zaraz po sekcji `scripts` (zaktualizuj wersje zależności, żeby być na bieżąco) zgodnie z listingiem 2.9.

Listing 2.9. Używanie kodu źródłowego pakietów w pliku package.json

```
"dependencies": {
  "@angular/common": "2.0.0",
  "@angular/compiler": "2.0.0",
  "@angular/core": "2.0.0",
  "@angular/forms": "2.0.0",
  "@angular/http": "2.0.0",
  "@angular/platform-browser": "2.0.0",
  "@angular/platform-browser-dynamic": "2.0.0",
  "@angular/router": "3.0.0",
}
```

```

"core-js": "^2.4.0",
"rxjs": "5.0.0-beta.12",
"systemjs": "0.19.37",
"zone.js": "0.6.21"
},
"devDependencies": {
  "live-server": "0.8.2",
  "typescript": "^2.0.0"
}

```

Teraz w wierszu poleceń w katalogu, w którym znajduje się plik *package.json*, uruchom polecenie `npm install`, a npm rozpocznie pobieranie powyższych pakietów i ich zależności do folderu *node_modules*. Po zakończeniu tego procesu zobaczysz w folderze *node_modules* dziesiątki podkatalogów, w tym *@angular*, *systemjs*, *live-server* i *typescript*.

```

angular-seed
├── index.html
├── package.json
├── app
│   └── app.ts
├── node_modules
│   ├── @angular
│   ├── systemjs
│   ├── typescript
│   ├── live-server
│   └── ...

```

W folderze *angular-seed* utwórzmy nieco zmodyfikowaną wersję pliku *index.html* z zawartością pokazaną w listingu 2.10.

Listing 2.10. Plik *index.html*

```

<!DOCTYPE html>
<html>
<head>
  <title>Angular seed project</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">

  <script src="node_modules/typescript/lib/typescript.js"></script>
  <script src="node_modules/core-js/client/shim.min.js"></script>
  <script src="node_modules/zone.js/dist/zone.js"></script>
  <script src="node_modules/systemjs/dist/system.src.js"></script>
  <script src="systemjs.config.js"></script>
  <script>
    System.import('app').catch(function(err){ console.error(err); });
  </script>
</head>

<body>
<app>Ładuję...</app>
</body>
</html>

```

Zwróć uwagę, że znaczniki script ładują teraz wymagane zależności z lokalnego katalogu *node_modules*. Zrobimy to samo z plikiem konfiguracyjnym *systemjs.config.js* ładowarki SystemJS pokazanym w listingu 2.11.

Listing 2.11. Plik *systemjs.config.js*

```
System.config({
  transpiler: 'typescript',
  typescriptOptions: {emitDecoratorMetadata: true},
  map: {
    '@angular': 'node_modules/@angular',
    'rxjs'      : 'node_modules/rxjs'
  },
  paths: {
    'node_modules/@angular/*': 'node_modules/@angular/*/bundles'
  },
  meta: {
    '@angular/*': {'format': 'cjs'}
  },
  packages: {
    'app'                : {main: 'main', defaultExtension: 'ts'},
    'rxjs'               : {main: 'Rx'},
    '@angular/core'      : {main: 'core.umd.min.js'},
    '@angular/common'    : {main: 'common.umd.min.js'},
    '@angular/compiler'  : {main: 'compiler.umd.min.js'},
    '@angular/platform-browser' : {main: 'platform-browser.umd.min.js'},
    '@angular/platform-browser-dynamic': {main: 'platform-browser-dynamic.umd.min.js'}
  }
});
```

Powyższa konfiguracja SystemJS różni się nieco od pokazanej w listingu 2.1. Tym razem nie używamy kodu źródłowego pakietów Angular. Zamiast tego używamy ich spakowanych i zminimalizowanych wersji. Spowoduje to zminimalizowanie liczby żądań sieciowych wymaganych do załadowania frameworku Angular, a ta wersja frameworku będzie mniejsza. Każdy pakiet Angular jest dostarczany z katalogiem o nazwie *bundles*, który zawiera zminimalizowany kod. W sekcji *packages* pliku konfiguracyjnego SystemJS mapujemy nazwę *app* na główny skrypt znajdujący się w *main.ts*, więc kiedy napiszemy *System.import(app)* w pliku *index.html*, załadowany zostanie moduł *main.ts*.

W katalogu głównym projektu dodajmy jeszcze jeden plik konfiguracyjny, w którym określimy opcje kompilatora *tsc*, tak jak w listingu 2.12.

Listing 2.12. Plik *tsconfig.json*

```
{
  "compilerOptions": {
    "target": "ES5",
    "module": "commonjs",
    "experimentalDecorators": true,
    "noImplicitAny": true
  }
}
```

Jeśli TypeScript jest dla Ciebie nowym językiem, przeczytaj dodatek B, który wyjaśnia, że w celu uruchomienia kodu TypeScript trzeba go najpierw transkompilować na kod JavaScript za pomocą kompilatora `tsc` TypeScriptu. Przykłady kodu w rozdziałach od 1. do 7. działają bez bezpośredniego uruchomienia `tsc`, ponieważ SystemJS używa `tsc` wewnętrznie do transkompilacji TypeScriptu na kod JavaScript w locie podczas ładowania pliku skryptu. Mimo to, będziemy przechowywać plik `tsconfig.json` w głównym katalogu projektu, ponieważ wykorzystują go niektóre środowiska IDE.

UWAGA Jeśli kod Angular jest dynamicznie kompilowany w przeglądarce (nie mylić z transkompilacją), nazywa się to kompilacją *just-in-time* (JIT). Jeśli kod jest wstępnie kompilowany za pomocą specjalnego kompilatora `ngc`, nazywa się to kompilacją *ahead-of-time* (AoT). W tym rozdziale opiszemy aplikację z kompilacją JIT.

Kod aplikacji będzie składał się z trzech plików:

- `app.component.ts` — jedyny komponent aplikacji;
- `app.module.ts` — deklaracja modułu, który będzie zawierał komponent;
- `main.ts` — plik inicjujący moduł.

W punkcie 2.3.3 zmapowaliśmy nazwę `app` na `main.ts`, utwórzmy więc katalog o nazwie `app`, zawierający plik `app.component.ts` z zawartością pokazaną w listingu 2.13.

Listing 2.13. Plik `app.component.ts`

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

@Component({
  selector: 'app',
  template: `<h1>Witaj, {{ name }}!</h1>`
})
export class AppComponent {
  name: string;

  constructor() {
    this.name = 'Angular 2';
  }
}
```

Teraz musisz utworzyć moduł, który będzie zawierać `AppComponent`, tak jak widać w listingu 2.14. Umieścimy ten kod w pliku `app.module.ts`.

Listing 2.14. Plik `app.module.ts`

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

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent ],
```

```
bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Ten plik zawiera po prostu definicję modułu Angular. Klasa jest adnotowana dekoratorem `@NgModule` zawierającym `BrowserModule`, który każda przeglądarka musi importować. Ponieważ ten moduł zawiera tylko jedną klasę, należy ją podać we właściwości `declarations` i wymienić jako klasę inicjującą:

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';
```

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

Uruchom aplikację, wykonując polecenie `npm start` w oknie wiersza poleceń otwartego w folderze *angular-seed*. Otwarte zostanie okno przeglądarki wyświetlające przez ułamek sekundy komunikat *Ładowanie...*, a następnie komunikat *Witaj, Angular 2!*. Rysunek 2.2 pokazuje, jak wygląda ta aplikacja w przeglądarce Chrome. Przedstawia on przeglądarkę z otwartym panelem narzędzi dla programistów i wybraną zakładką *Network*, dzięki czemu można zobaczyć fragment tego, co zostało pobrane przez przeglądarkę i jak długo to trwało.

Nie przejmuj się rozmiarem pobierania. Zoptymalizujemy to w rozdziale 10. Ponieważ używamy serwera `live-server`, zaraz po zmodyfikowaniu i zapisaniu kodu tej aplikacji nastąpi odświeżenie strony w przeglądarce i załadowana zostanie najnowsza wersja kodu. Zastosujmy teraz to, czego się nauczyłeś, w aplikacji, która jest bardziej złożona niż *Witaj, świecie*.

2.5. Część praktyczna: rozpoczynamy tworzenie aplikacji aukcji internetowych

Od tego momentu każdy rozdział będzie kończył się częścią praktyczną zawierającą instrukcję opracowywania konkretnego aspektu aukcji internetowych, gdzie użytkownicy mogą zobaczyć listę oferowanych produktów, wyświetlić szczegółowe informacje dotyczące określonego produktu, przeprowadzić wyszukiwanie produktów i monitorować oferty składane przez innych użytkowników. Stopniowo będziemy dodawać kod do tej aplikacji, aby przećwiczyć to, czego nauczysz się w każdym rozdziale. Kod źródłowy dołączony do tej książki zawiera w folderze *auction* każdego rozdziału kompletną wersję części praktycznej, ale zachęcamy do samodzielnego wykonywania tych ćwiczeń.

W tym ćwiczeniu skonfigurujemy środowisko programistyczne i utworzymy wstępny układ projektu aukcji. Utworzymy stronę główną, podzielimy ją na komponenty Angular i utworzymy usługę pobierania produktów. Jeśli zastosujesz się do wszystkich instrukcji w tym podrozdziale, strona główna aukcji powinna wyglądać tak, jak pokazano na rysunku 2.3.

Użyjemy szarych prostokątów dostarczonych przez wygodną usługę `Placeholder.it` (<http://placeholder.it>), która generuje elementy zastępcze o określonych rozmiarach. Aby

Name	Status	Type	Initiator	Size	Time	Waterfall
127.0.0.1	200	document	127.0.0.1/42	2.0 KB	8 ms	
typescript.js	200	script	(index)	4.4 MB	85 ms	
shim.min.js	200	script	(index)	77.5 KB	15 ms	
zone.js	200	script	(index)	52.6 KB	14 ms	
system.src.js	200	script	(index)	164 KB	19 ms	
system.js.config.js	200	script	(index)	1.1 KB	12 ms	
main.ts	200	xhr	zone.js:1241	429 B	3 ms	
ws	101	websocket	(index):40	0 B	Pending	
platform-browser-dynamic.umd.min.js	200	xhr	zone.js:1241	4.9 KB	2 ms	
app.module.ts	200	xhr	zone.js:1241	562 B	4 ms	
compiler.umd.min.js	200	xhr	zone.js:1241	505 KB	41 ms	
core.umd.min.js	200	xhr	zone.js:1241	181 KB	9 ms	
platform-browser.umd.min.js	200	xhr	zone.js:1241	70.5 KB	5 ms	
app.component.ts	200	xhr	zone.js:1241	485 B	2 ms	
common.umd.min.js	200	xhr	zone.js:1241	56.6 KB	4 ms	
Subject.js	200	xhr	zone.js:1241	5.5 KB	3 ms	
Observable.js	200	xhr	zone.js:1241	6.2 KB	3 ms	
Subscriber.js	200	xhr	zone.js:1241	8.9 KB	5 ms	
Subscription.js	200	xhr	zone.js:1241	6.1 KB	5 ms	
ObjectUnsubscribedError.js	200	xhr	zone.js:1241	1.2 KB	5 ms	
SubjectSubscription.js	200	xhr	zone.js:1241	1.6 KB	9 ms	
rxSubscriber.js	200	xhr	zone.js:1241	538 B	8 ms	
root.js	200	xhr	zone.js:1241	739 B	4 ms	
toSubscriber.js	200	xhr	zone.js:1241	975 B	9 ms	
observable.js	200	xhr	zone.js:1241	899 B	9 ms	
isFunction.js	200	xhr	zone.js:1241	415 B	6 ms	

32 requests | 5.6 MB transferred | Finish: 559 ms | DOMContentLoaded: 312 ms | Load: 309 ms

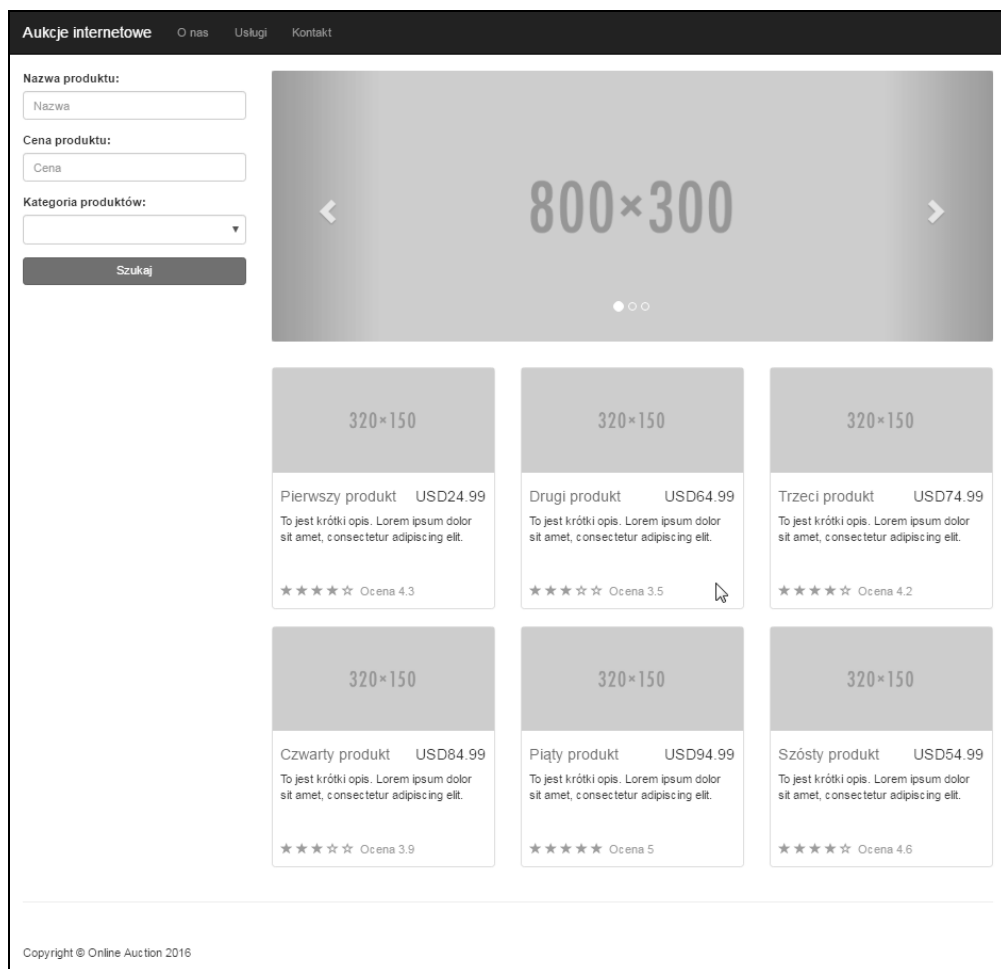
Rysunek 2.2. Uruchomienie aplikacji z projektu zarządzanego przez npm

zobaczyć te wygenerowane obrazy, musisz być podłączony do internetu podczas uruchamiania tej aplikacji. Kolejne punkty podrozdziału zawierają instrukcje, które należy wypełnić, aby ukończyć ćwiczenia praktyczne.

UWAGA Jeśli wolisz po prostu czytać kod działającej wersji aplikacji aukcji internetowych, użyj kodu z katalogu *auction* z przykładów dla tego rozdziału. Aby uruchomić dostarczony kod, przejdź do katalogu *auction*, uruchom polecenie `npm install` w celu zainstalowania wszystkich wymaganych zależności w katalogu *node_modules* i uruchom aplikację, wpisując polecenie `npm start`.

2.5.1. Wstępna konfiguracja projektu

Aby skonfigurować projekt, najpierw skopiuj katalog *angular-seed* z całą zawartością do osobnej lokalizacji i zmień jego nazwę na *auction*. Następnie zmodyfikuj pola *name* (nazwa) i *description* (opis) w pliku *package.json*. Otwórz wiersz poleceń i przejdź do nowo utworzonego katalogu *auction*. Uruchom polecenie `npm install`, które spowoduje utworzenie katalogu *node_modules* z zależnościami określonymi w pliku *package.json*.



Rysunek 2.3. Strona główna aplikacji aukcji internetowych

W tym projekcie będziemy używać biblioteki Bootstrap Twittera jako frameworku CSS oraz biblioteki komponentów responsywnych interfejsu użytkownika. Responsywne projektowanie stron internetowych to podejście umożliwiające tworzenie witryn zmieniających układ na podstawie szerokości rzutni urządzenia użytkownika. Określenie *komponenty responsywne* oznacza, że układ komponentów może dostosowywać się do rozmiarów ekranu.

Ponieważ biblioteka Bootstrap jest zbudowana na bazie jQuery, należy uruchomić następujące polecenia, aby zainstalować obie biblioteki:

```
npm install bootstrap --save
npm install jquery --save
```

Instalowanie biblioteki Bootstrap. Opcja --save
doda tę zależność do pliku package.json.

Pakiet Bootstrap nie określa jQuery jako zależności,
więc trzeba zainstalować ją osobno. Takie zależności
są również zwane zależnościami równorzędnymi.

WSKAZÓWKA Zalecamy użycie środowiska IDE, takiego jak WebStorm lub Visual Studio Code. Większość kroków niezbędnych do ukończenia tego praktycznego projektu można wykonać wewnątrz IDE. WebStorm pozwala nawet otworzyć okno terminala wewnątrz IDE.

Teraz utworzymy plik *systemjs.config.js* pokazany w listingu 2.15, aby zapisać konfigurację SystemJS. Załączymy ten plik w znaczniku `<script>` w pliku *index.html*.

Listing 2.15. Plik *systemjs.config.js*

```
System.config({
  transpiler: 'typescript',
  typescriptOptions: {emitDecoratorMetadata: true,
    target: "ES5",
    module: "commonjs"},
  map: {
    '@angular': 'node_modules/@angular',
    'rxjs' : 'node_modules/rxjs'
  },
  paths: {
    'node_modules/@angular/*': 'node_modules/@angular/*/bundles'
  },
  meta: {
    '@angular/*': {'format': 'cjs'}
  },
  packages: {
    'app' : {main: 'main', defaultExtension: 'ts'},
    'rxjs' : {main: 'Rx'},
    '@angular/core' : {main: 'core.umd.min.js'},
    '@angular/common' : {main: 'common.umd.min.js'},
    '@angular/compiler' : {main: 'compiler.umd.min.js'},
    '@angular/platform-browser' : {main: 'platform-browser.umd.min.js'},
    '@angular/platform-browser-dynamic' : {main: 'platform-browser-dynamic.umd.min.js'}
  }
});
```

Instruuje kompilator TypeScriptu ładowarki SystemJS, aby zachować metadane dekoratorów w transkompilowanym kodzie, ponieważ Angular bazuje na adnotacjach przy wykrywaniu i rejestrowaniu komponentów.

Transkompiluje kod na składnię ES5.

Używa formatu modułów CommonJS.

Kod aplikacji przechowujemy w katalogu app, a kod do uruchamiania aplikacji będzie znajdował się w pliku main.ts.

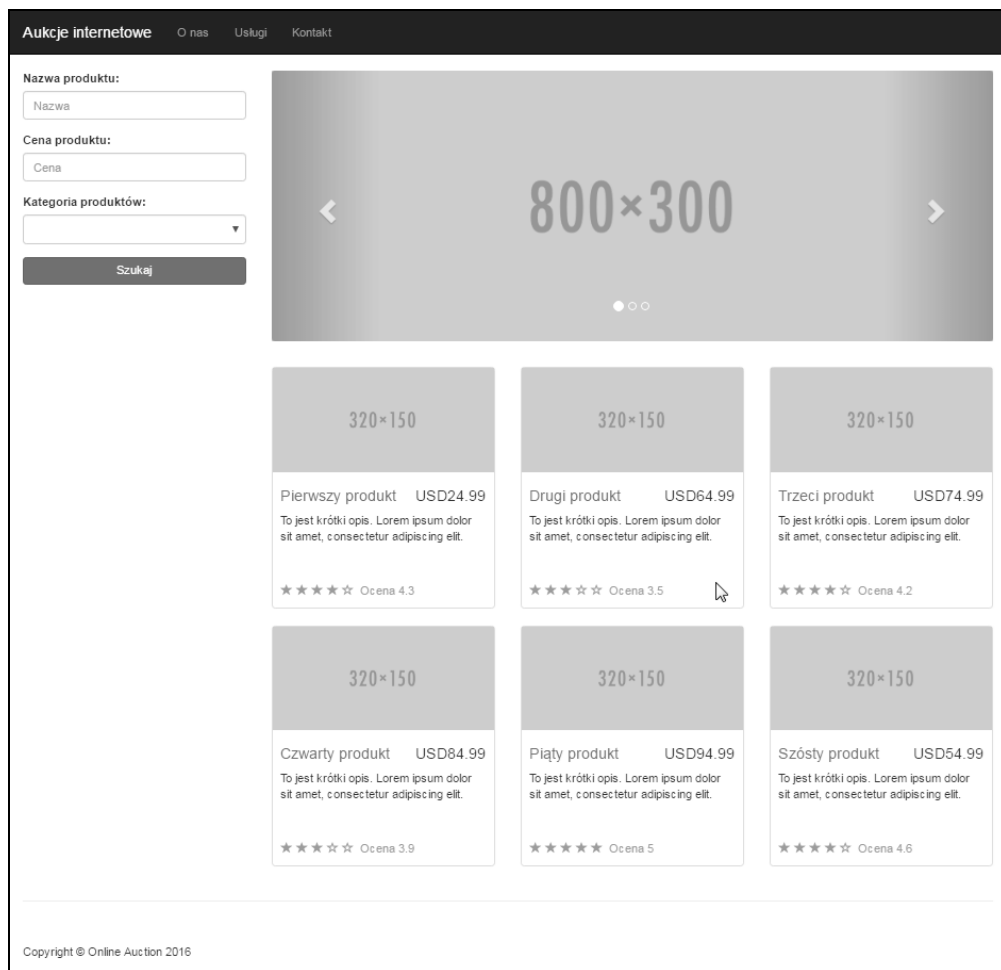
Plik *systemjs.config.js* musi zostać załączony w pliku *index.html*, tak jak pokazano w następnym punkcie w listingu 2.16. Ta konfiguracja pakietu app pozwala na użycie w pliku *index.html* linii `<script>System.import('app')</script>`, która załaduje zawartość *app/main.ts*.

To kończy konfigurację środowiska programistycznego dla projektu aukcji. Teraz jesteś gotowy, aby rozpocząć pisanie kodu aplikacji.

UWAGA Kiedy powstawał ten tekst, zespół Angular pracował nad komponentami Angular Material dla frameworku Angular (zobacz <https://material.angular.io>). Kiedy będą gotowe, być może zechcesz ich użyć zamiast biblioteki Bootstrap Twittera.

2.5.2. Tworzenie strony głównej

W tym ćwiczeniu utworzymy stronę główną, która zostanie podzielona na kilka komponentów Angular. Ułatwi to utrzymywanie kodu i umożliwi ponowne użycie komponentów w innych widokach. Na rysunku 2.4 można zobaczyć stronę główną z zaznaczonymi wszystkimi komponentami.



Rysunek 2.4. Strona główna aplikacji aukcji internetowych z zaznaczonymi komponentami

Musimy utworzyć katalogi do przechowywania wszystkich komponentów i usług w następujący sposób:

```

app
├── components
│   ├── application
│   ├── carousel
│   └── footer

```

```

├── navbar
├── product-item
├── search
├── stars
└── services

```

Każdy katalog w folderze *components* zawiera kod dla odpowiedniego komponentu. Pozwala to przechowywać razem wszystkie pliki związane z pojedynczym komponentem. Większość komponentów składa się z dwóch plików — pliku HTML i pliku TypeScript. Czasami jednak warto dodać plik CSS ze stylizacją charakterystyczną dla danego komponentu. Katalog *services* będzie zawierał plik z klasami, które serwują dane do aplikacji.

Pierwsza wersja strony głównej składa się z siedmiu komponentów. W tym ćwiczeniu omówimy i utworzymy trzy najciekawsze komponenty znajdujące się w katalogach *application*, *product-item* i *stars*. Da Ci to okazję, żeby napisać nieco kodu, a na końcu tego ćwiczenia możesz skopiować pozostałe komponenty do katalogu projektu.

UWAGA W części praktycznej w rozdziale 3. zrefaktoryzujemy kod, aby zintegrować karuzelę i produkty w komponencie HomeComponent.

Punktem wejścia aplikacji jest *index.html*. Skopiowałeś ten plik z folderu *angular-seed*, a teraz musisz go zmodyfikować (zobacz listing 2.16). Plik *index.html* jest dość mały i zbytnio się nie zwiększy, ponieważ większość zależności będzie ładowana przez SystemJS, a cały interfejs użytkownika jest reprezentowany przez pojedynczy komponent główny (najwyższego poziomu) Angular, który wewnętrznie używa elementów potomnych.

Listing 2.16. Plik index.html

```

<!DOCTYPE html>
<html>
<head>
  <title>R02: Aukcje internetowe </title>
  <link rel="stylesheet" href="node_modules/bootstrap/dist/css/bootstrap.css">

  <script src="node_modules/jquery/dist/jquery.min.js"></script>
  <script src="node_modules/bootstrap/dist/js/bootstrap.min.js"></script>

  <script src="node_modules/core-js/client/shim.min.js"></script>
  <script src="node_modules/zone.js/dist/zone.js"></script>

  <script src="node_modules/typescript/lib/typescript.js"></script>
  <script src="node_modules/systemjs/dist/system.src.js"></script>
  <script src="systemjs.config.js"></script>

  <script>
    System.import('app').catch(function (err) {console.error(err)});
  </script>
</head>
<body>
<auction-application></auction-application>
</body>
</html>

```

Dodaje CSS biblioteki Bootstrap.

Dodaje Bootstrap i jQuery do obsługi komponentu karuzeli.

Ładuje main.ts zgodnie z konfiguracją w pliku systemjs.config.js.

Zawartość pliku *main.ts* w katalogu *app* pozostaje taka sama jak w projekcie *angular-seed*:

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';
platformBrowserDynamic().bootstrapModule(AppModule);
```

Zaktualizujemy plik *app.module.ts*, aby zadeklarować wszystkie komponenty i usługi, których będziemy używać w aplikacji aukcji internetowych. Plik został pokazany w listingu 2.17.

Listing 2.17. Zaktualizowany plik *app.module.ts*

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './components/application/application';
import { CarouselComponent } from './components/carousel/carousel';
import { FooterComponent } from './components/footer/footer';
import { NavbarComponent } from './components/navbar/navbar';
import { ProductItemComponent } from './components/product-item/product-item';
import { SearchComponent } from './components/search/search';
import { StarsComponent } from './components/stars/stars';
import { ProductService } from './services/product-service';

@NgModule({
  imports: [ BrowserModule ],
  declarations: [ AppComponent, ← Deklaruje wszystkie komponenty,
    CarouselComponent,          których będzie używał moduł.
    FooterComponent,
    NavbarComponent,
    ProductItemComponent,
    SearchComponent,
    StarsComponent ],
  providers: [ ProductService ], ← Deklaruje dostawcę dla usługi ProductService,
  bootstrap: [ AppComponent ]   którą nieco później wstrzykniemy do komponentu
                                ApplicationComponent.
})
export class AppModule { }
```

W tym module deklarujemy wszystkie komponenty i dostawcę dla jednej usługi, którą za chwilę utworzymy. Deklaracja dostawcy dla usługi jest wymagana przez mechanizm wstrzykiwania zależności. O dostawcach i wstrzykiwaniu porozmawiamy w rozdziale 4.

KOMPONENT APLIKACJI

Komponent aplikacji jest głównym komponentem aukcji i jako taki jest deklarowany w module *AppModule*. Służy jako host dla wszystkich innych komponentów. Kod źródłowy komponentu składa się z trzech plików: *application.ts*, *application.html* i *application.css*. Zakładamy, że znasz podstawy CSS, więc nie będziemy tutaj analizować tego pliku. Przejrzemy pierwsze dwa pliki.

Utwórzmy komponent *AppComponent* i zapiszmy go w pliku *application.ts* znajdującym się w katalogu *app/components/application*. Zawartość tego pliku została pokazana w listingu 2.18.

Listing 2.18. Plik application.ts

```
import {Component, ViewEncapsulation} from '@angular/core';
import {Product, ProductService} from '../services/product-service';
```

Importuje klasy, które implementują ProductService. Te klasy będą serwować dane.

```
@Component({
  selector: 'auction-application',
  templateUrl: 'app/components/application/application.html',
  styleUrls: ['app/components/application/application.css'],
  encapsulation: ViewEncapsulation.None
})
```

Zmienia klasę AppComponent w komponent Angular poprzez adnotowanie jej dekoratorem @Component.

Selektor definiuje nazwę niestandardowego znacznika HTML użytego w pliku index.html.

Szablon HTML będzie znajdował się w pliku application.html.

CSS znajduje się w pliku application.css.

```
export default class AppComponent {
```

Eksportuje AppComponent, ponieważ jest on wykorzystywany w innej klasie, czyli AppModule.

```
  products: Array<Product> = [];
```

Używa typów sparametryzowanych (zobacz dodatek B), aby zapewnić, że tablica products będzie zawierać tylko obiekty typu Product.

```
  constructor(private productService: ProductService) {
    this.products = this.productService.getProducts();
  }
```

Pobiera listę produktów i przypisuje je do właściwości products. Wszystkie właściwości komponentu stają się dostępne w szablonie widoku poprzez wiązanie danych.

W TypeScriptie można poinstruować Angular za pomocą argumentów konstruktora, żeby wstrzyknął żądane obiekty (takie jak ProductService).

```
}
```

Samo zadeklarowanie argumentów konstruktora z typem poinstruuje Angular, żeby utworzyć instancję i wstrzyknąć ten obiekt (ProductService). Wstrzykiwalne obiekty muszą być skonfigurowane z dostawcami i zadeklarowaliśmy jednego z nich wcześniej w module AppModule. Kwalifikator `private` zmieni `productService` w zmienną składową klasy, więc będziemy uzyskiwać do niej dostęp jako `this.productService`.

UWAGA Listing 2.18 wykorzystuje strategię hermetyzacji widoku `ViewEncapsulation.None`, aby style z pliku `.css` aplikacji zastosować nie tylko do komponentu `AppComponent`, ale do całej aplikacji. Różne strategie hermetyzacji widoku omówimy w rozdziale 6.

Utwórzmy plik `application.html` o zawartości takiej, jaką widać w listingu 2.19.

Listing 2.19. Plik application.html

```
<auction-navbar></auction-navbar>

<div class="container">
  <div class="row">

    <div class="col-md-3">
      <auction-search></auction-search>
    </div>

    <div class="col-md-9">
      <div class="row carousel-holder">
        <div class="col-md-12">
          <auction-carousel></auction-carousel>
        </div>
      </div>
    </div>
  </div>
</div>
```

```

    </div>
  </div>
  <div class="row">
    <div *ngFor="let prod of products" class="col-sm-4 col-lg-4 col-md-4">
      <auction-product-item [product]="prod"></auction-product-item>
    </div>
  </div>
</div>
</div>
<auction-footer></auction-footer>

```

Będziemy używać wielu niestandardowych elementów HTML, które reprezentują komponenty: `<auction-navbar>`, `<auction-search>`, `<auction-carousel>`, `<auction-product-item>` i `<auction-footer>`. Dodamy je w taki sam sposób jak `<auction-application>` w pliku *index.html*.

Najciekawszą częścią tego pliku jest sposób wyświetlania listy produktów. Każdy produkt będzie reprezentowany na stronie internetowej przez ten sam fragment HTML. Ponieważ jest wiele produktów, musimy renderować ten sam kod HTML wielokrotnie. Dyrektywa `NgFor` jest używana wewnątrz właściwości `template` komponentu, aby utworzyć pętlę przez listę pozycji w kolekcji danych i renderować znaczniki HTML dla każdej pozycji. Do reprezentowania dyrektywy `NgFor` można użyć składni skrótowej `*ngFor`.

```

<div *ngFor="let prod of products" class="col-sm-4 col-lg-4 col-md-4">
  <auction-product-item [product]="prod"></auction-product-item>
</div>

```

Ponieważ `*ngFor` znajduje się w elemencie `<div>`, każda iteracja pętli będzie renderować `<div>` z zawartością znajdującą się w nim odpowiedniej pozycji `<auction-product-item>`. Aby przekazać instancję produktu do `ProductComponent`, używamy nawiasów kwadratowych do powiązania właściwości `[product]="prod"`, gdzie `[product]` odwołuje się do nazwanego poprzez wiązanie właściwości produktu wewnątrz komponentu reprezentowanego przez `<auction-product-item>`, a `prod` to lokalna zmienna szablonu zadeklarowana w locie w dyrektywie `*ngFor` jako `let prod`. Wiązania właściwości omówimy szczegółowo w rozdziale 5.

Style `col-sm-4 col-lg-4 col-md-4` pochodzą z biblioteki Bootstrap Twittera, gdzie szerokość okna jest podzielona na 12 niewidocznych kolumn. W tym przykładzie chcemy przydzielić 4 kolumny (jedna trzecia szerokości `<div>`), jeśli urządzenie ma małe (sm oznacza 768 pikseli lub więcej), duże (lg oznacza 1200 pikseli lub więcej) i średnie (md oznacza 992 piksele lub więcej) rozmiary ekranu.

Ponieważ nie określamy żadnych kolumn dla bardzo małych urządzeń (xs oznacza ekrany poniżej 768 pikseli), cała szerokość `<div>` zostanie przydzielona dla pojedynczego `<auction-product>`. Aby zobaczyć, jak zmienia się układ strony w przypadku różnych rozmiarów ekranu, zawęż okno przeglądarki, aby jego szerokość wynosiła mniej niż 768 pikseli. Więcej informacji o systemie siatki (ang. *grid system*) biblioteki Bootstrap można znaleźć w dokumentacji Bootstrap na stronie <http://getbootstrap.com/css/#grid>.

UWAGA `AppComponent` opiera się na istnieniu innych komponentów (takich jak `ProductItemComponent`), które utworzymy w kolejnych krokach. Jeśli teraz spróbujesz uruchomić aplikację aukcji, zobaczysz błędy w konsoli programisty w Twojej przeglądarce.

KOMPONENT POZYCJI PRODUKTU

W katalogu *product-item* utwórz plik *product-item.ts*, który deklaruje komponent `ProductItemComponent` reprezentujący pojedynczą pozycję produktu z aukcji. Kod źródłowy *product-item.ts* ma strukturę podobną do kodu *application.ts*: instrukcje `import` pojawiają się na górze, a następnie mamy deklarację klasy komponentu z adnotacją `@Component` (zobacz listing 2.10).

Listing 2.20. Plik *product-item.ts*

```
import {Component, Input} from '@angular/core';
import StarsComponent from 'app/components/stars/stars';
import {Product} from 'app/services/product-service';

@Component({
  selector: 'auction-product-item',
  templateUrl: 'app/components/product-item/product-item.html'
})
export default class ProductItemComponent {
  @Input() product: Product;
}
```

Właściwość `product` komponentu jest adnotowana dekoratorem `@Input()`. Oznacza to, że będzie ona udostępniona komponentowi nadrzędnemu, który może powiązać z nią wartość. Właściwości wejściowe omówimy szczegółowo w rozdziale 6.

Utwórz plik *product-item.html*, który będzie zawierał szablon komponentu produktu pokazany w listingu 2.21 (będzie on reprezentowany przez cenę, tytuł i opis).

Listing 2.21. Plik *product-item.html*

```
<div class="thumbnail">
  
  <div class="caption">
    <h4 class="pull-right">{{ product.price }}</h4>
    <h4><a>{{ product.title }}</a></h4>
    <p>{{ product.description }}</p>
  </div>
  <div>
    <auction-stars [rating]="product.rating"></auction-stars>
  </div>
</div>
```

Używamy tutaj kolejnego typu wiązania danych: wyrażenia umieszczonego w podwójnych nawiasach klamrowych. Angular ewaluuje wartość wyrażenia w nawiasach klamrowych, zamienia wynik na łańcuch znaków i zastępuje wyrażenie w szablonie z wynikowym

łańcuchem znaków. Wewnętrznie ten proces jest implementowany za pomocą interpolacji łańcuchów znaków.

Zwróć uwagę na znacznik `<auction-stars>`, który reprezentuje `StarsComponent` i został zadeklarowany w module `AppModule`. Wiążemy wartość `product.rating` z właściwością `rating` komponentu `StarsComponent`. Aby to działało, właściwość `rating` musi być zadeklarowana jako właściwość wejściowa w komponencie `StarsComponent`, który utworzymy w następnej kolejności.

KOMPONENT GWIAZDEK

Komponent gwiazdek będzie wyświetlał ocenę produktu. Na rysunku 2.5 widać, że wyświetla on średnią ocenę 4.3 oraz ikony gwiazdek reprezentujących tę ocenę.

Angular zapewnia zaczepy cyklu życia komponentów (zobacz rozdział 6.) umożliwiające definiowanie metod wywołania zwrótnego, które zostaną wywołane w określonych momentach cyklu życia komponentu. W tym komponencie użyjemy wywołania zwrótnego `ngOnInit()`, które będzie wywołane, gdy utworzona zostanie instancja komponentu i zainicjowane jego właściwości. Utwórzmy w katalogu `stars` plik `stars.ts` z następującą zawartością.



Rysunek 2.5.
Komponent gwiazdek

Listing 2.22. Plik `stars.ts`

```
import {Component, Input, OnInit} from '@angular/core';

@Component({
  templateUrl: 'app/components/stars/stars.html',
  styles: ['.starrating { color: #d17581; }'],
  selector: 'auction-stars'
})
export default class StarsComponent implements OnInit {
  @Input() count: number = 5;
  @Input() rating: number = 0;
  stars: boolean[] = [];

  ngOnInit() {
    for (let i = 1; i <= this.count; i++) {
      this.stars.push(i > this.rating);
    }
  }
}
```

Importuje interfejs `OnInit`, w którym zadeklarowana jest metoda `ngOnInit()`.

Oznacza `rating` i `count` jako dane wejściowe, żeby inne komponenty mogły przypisywać do nich wartości poprzez wiązanie danych.

Każdy element tej tablicy reprezentuje pojedynczą gwiazdkę, która ma być renderowana.

Inicjuje gwiazdki na podstawie wartości dostarczonej przez komponent nadrzędny.

Właściwość `count` określa całkowitą liczbę gwiazdek, które mają być renderowane. Jeśli ta właściwość nie jest inicjowana przez element nadrzędny, komponent domyślnie renderuje pięć gwiazdek.

Właściwość `rating` przechowuje średnią ocenę określającą liczbę gwiazdek, które powinny zostać wypełnione kolorem, oraz to, ile gwiazdek powinno pozostać pustych. W tablicy `stars` elementy z wartością `false` reprezentują puste gwiazdki, a elementy z wartością `true` reprezentują gwiazdki wypełnione kolorem.

Tablicę stars inicjujemy w wywołaniu zwrotnym cyklu życia `ngOnInit()`, które będzie używane w szablonie do renderowania gwiazdek. Metoda `ngOnInit()` jest wywoływana tylko raz, zaraz po tym, jak właściwości komponentu, które mają wiązanie danych, zostaną sprawdzone po raz pierwszy i przed sprawdzeniem jakichkolwiek elementów potomnych tego komponentu. Gdy wywoływana jest metoda `ngOnInit()`, wszystkie właściwości przekazywane z widoku nadrzędnego są już zainicjowane, więc można użyć wartości `rating` do obliczenia wartości w tablicy stars.

Alternatywnie można zmienić tablicę stars w metodę pobierającą, aby obliczyć ją w locie, ale taka metoda pobierająca byłaby wywoływana za każdym razem, gdy Angular synchronizuje model z widokiem. Dokładnie taka sama tablica byłaby obliczana wielokrotnie.

Utwórzmy szablon komponentu StarsComponent w pliku *stars.html*, tak jak pokazano w listingu 2.23.

Listing 2.23. Plik stars.html

```
<p>
  <span *ngFor="let star of stars"
    class="starrating glyphicon glyphicon-star"
    [class.glyphicon-star-empty]="star">
  </span>
  <span>Ocena {{ rating }}</span>
</p>
```

Używaliśmy już dyrektywy `NgFor` i wyrażenia wiązania danych umieszczonego w nawiasach klamrowych w komponencie `AppComponent`. Tutaj wiążemy nazwę klasy CSS z wyrażeniem `[class.glyphicon-star-empty]="star"`. Jeśli wyrażenie w podwójnym cudzysłowie znajdujące się po prawej stronie ewaluuje do wartości `true`, klasa CSS `glyphicon-star-empty` jest dodawana do atrybutu `class` elementu `<span>`.

SKOPIOWANIE RESZTY KODU

Aby dokończyć ten projekt, skopiuj brakujące komponenty z katalogu *r02/auction* do odpowiednich katalogów projektu.

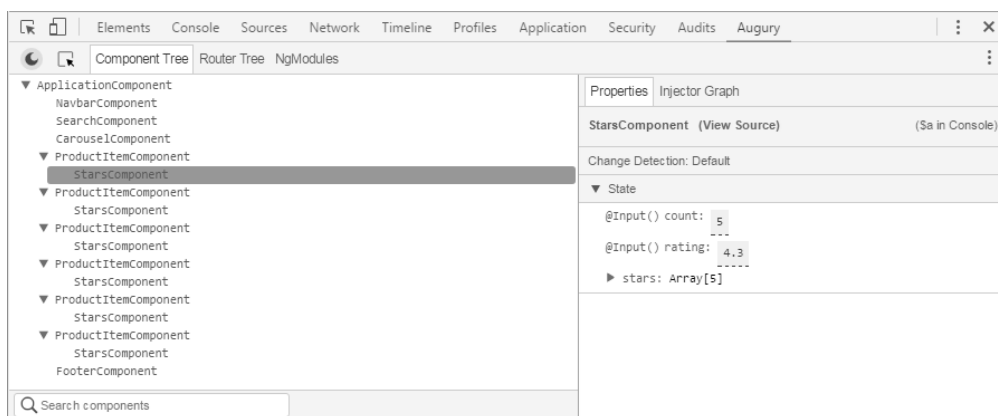
- Katalog *services* zawiera plik *product-service.ts*, który deklaruje dwie klasy, `Product` i `ProductService`. Stąd pochodzą dane dla aukcji. Więcej szczegółów na temat zawartości tego pliku podamy w części praktycznej w rozdziale 3.
- Katalog *navbar* zawiera kod dla górnego paska nawigacyjnego.
- Katalog *footer* zawiera kod stopki strony.
- Katalog *search* zawiera początkowy kod dla komponentu `SearchComponent` będącego formularzem, który opracujemy w rozdziale 7.
- Katalog *carousel* zawiera kod implementujący suwak biblioteki Bootstrap w górnej części strony głównej.

2.5.3. Uruchomienie aplikacji aukcji internetowych

Aby uruchomić aplikację aukcji internetowych, otwórz okno wiersza poleceń w katalogu projektu i uruchom `live-server`. Możesz to zrobić za pomocą polecenia `npm start`, które zostało skonfigurowane w pliku `package.json` w celu uruchamiania serwera `live-server`. Otwarta zostanie przeglądarka i powinieneś zobaczyć stronę główną, taką jaką pokazano na rysunku 2.4. Strona szczegółów produktu nie została jeszcze zaimplementowana, więc linki w nazwach produktów nie działają.

Podczas rozwijania aplikacji zalecamy używanie przeglądarki Chrome, ponieważ ma najlepsze narzędzia do debugowania kodu. Miej włączony panel narzędzi dla programistów podczas uruchamiania wszystkich przykładów kodu. Jeśli zobaczysz nieoczekiwane rezultaty, sprawdź komunikaty o błędach w zakładce *Console* (konsola).

Dostępne jest też świetne rozszerzenie przeglądarki Chrome o nazwie *Augury*, które jest wygodnym narzędziem debugowania dla aplikacji Angular. Po zainstalowaniu tego rozszerzenia pojawi się dodatkowa zakładka *Augury* w panelu narzędzi dla programistów Chrome (zobacz rysunek 2.6), która umożliwia wyświetlanie i modyfikowanie wartości komponentów aplikacji podczas jej działania.



Rysunek 2.6. Panel rozszerzenia Augury

2.6. Podsumowanie

W tym rozdziale po raz pierwszy pisałeś aplikację Angular. Omówiliśmy pokrótce główne zasady i najważniejsze elementy konstrukcyjne aplikacji Angular. W kolejnych rozdziałach opiszemy je szczegółowo. Utworzyłeś również wstępną wersję aplikacji aukcji internetowych. Zobaczyłeś, jak skonfigurować środowisko programistyczne oraz strukturę projektu frameworku Angular.

- Aplikacja Angular jest reprezentowana przez hierarchię komponentów, które są spakowane w moduły.
- Każdy komponent Angular zawiera szablon dla renderowania interfejsu użytkownika oraz adnotowaną klasę, która implementuje funkcjonalność tego komponentu.

- Szablony i style mogą być umieszczone lokalnie w kodzie lub przechowywane w osobnych plikach.
- Ładowarka modułów SystemJS umożliwia dzielenie aplikacji na moduły ES6 i dynamiczne składanie wszystkiego w trakcie działania aplikacji.
- Parametry konfiguracji ładowarki SystemJS można określić w osobnym pliku konfiguracyjnym.
- Używanie menedżera npm do zarządzania zależnościami jest najprostszym sposobem konfigurowania nowego projektu Angular.

Skorowidz

A

- ActivatedRoute
 - wyodrębnianie parametrów, 97
- adnotacje, 438
- adres URL, 88
- aktualizowanie interfejsu użytkownika, 221
- analyzer kodu, 35, 36
 - ESLint, 36
 - JSLint, 36
- Angular, 23
 - serwer Node, 273
- Angular CLI, 36, 358
 - nowy projekt, 359
- AngularJS, 25
- anulowanie strumieni obserwowalnych, 181
- API, 219
- aplikacja aukcji internetowych, 70
 - bundler Webpack, 362
 - dodanie funkcjonalności oceniania, 221
 - dodanie nawigacji, 119
 - filtrowanie produktów, 187
 - komponent aplikacji, 76
 - komponent gwiazdek, 80
 - komponent pozycji produktu, 79
 - mechanizm DI, 151
 - powiadomienia o ofertach, 294
 - strona główna, 74, 152
 - testy jednostkowe, 328
 - uruchamianie serwera Node, 362
 - uruchomienie, 82, 126
 - uruchomienie klienta, 364
 - uruchomienie testów, 367
 - wdrożenie, 362
 - wstępna konfiguracja, 71
 - wyszukiwanie produktów, 294
- aplikacja pogodowa, 184
 - tester Karma, 327
 - testowanie, 314

- testowanie komponentu, 321
- testowanie usługi, 319
- aplikacja ze wstrzykiwaniem zależności, 137
- aplikacje
 - jednostronicowe, 85
 - SPA, 112
- architektura aplikacji, 26, 30
- asercja, 307
- asynchroniczny walidator SSN, 252
- AsyncPipe, 278
- atrybuty, 166
 - walidacji HTML, 233
- aukcje internetowe, 70

B

- bąbelkowanie zdarzeń, 199
- BDD, behavior-driven development, 306
- bezpieczeństwo, 363
- biblioteka, 22
 - Bootstrap, 23
 - jQuery, 23
 - Node.js, 24
 - Polymer, 24
 - React, 24
 - RxJS, 24
- biblioteka testowa
 - funkcje, 310
 - testowanie komponentów, 313
 - testowanie nawigacji routera, 312
 - testowanie usług, 312
- bloki konstrukcyjne nawigacji, 89
- błąd 404, 97
- Bootstrap, 23
- Bower, 36
- bundler Webpack, 38, 362

C

CD, continuous delivery, 359
 CI, continuous integration, 359
 ciągła integracja, CI, 359
 ciągle
 dostarczanie oprogramowania, CD, 359
 wdrażanie, 359
 cykl życia komponentów, 210

D

definiowanie modelu formularza, 245
 deklarowanie
 dostawcy, 135, 144
 interfejsu, 433
 zmiennych, 378
 dekoratory, 417
 destrukuryzacja, 168, 387
 obiekту, 388
 tablic, 389
 zagnieżdżonego obiektu, 389
 DI, dependency injection, 129
 dodawanie
 dyrektyw walidacji, 256
 funkcjonalności oceniania, 221
 interfejsu RESTful API, 272
 listy kategorii, 257
 nawigacji, 119
 routingu, 125
 dokument HTML, 164
 DOM, 164
 dostawca, provider, 132
 dwukierunkowe wiązanie danych, 26, 162, 171
 dyrektywa, 54, 169
 formArrayName, 244
 formControl, 243
 formControlName, 242
 formGroup, 241
 formGroupName, 242
 ngContent, 205
 NgForm, 236
 NgModel, 237
 NgModelGroup, 237
 RouterLink, 123
 dyrektywy formularzy, 241
 działanie Angular, 39
 dziedziczenie, 393, 428
 dzielenie aplikacji na moduły, 115

E

ECMAScript 6, 371
 elementy konstrukcyjne aplikacji, 51
 Ember.js, 23
 ESLint, 36
 Ext JS, 22

F

fabryka, 145
 Fetch API, 266
 filtrowanie produktów, 187
 format JSON, 269
 Forms API, 234–257
 formularz wyszukiwania, 256
 dodanie walidacji, 256
 formularze
 HTML, 232
 oparte na szablonach, 232, 235
 reaktywne, 178, 232, 240
 sterowane szablonami, 178
 walidacja, 247
 framework, 22
 Angular, 23
 AngularJS, 25
 Ember.js, 23
 Ext JS, 22
 Jasmine, 23, 307
 Node.js, 24, 266
 frameworki testowe, 307
 funkcja, 419
 async(), 311
 beforeEach(), 311
 describe(), 307
 fakeAsync(), 311
 getStockPrice(), 386
 inject(), 311
 super(), 397
 funkcje
 asynchroniczne, 402
 generatora, 385
 parametry domyślne, 420
 parametry opcjonalne, 420
 przeglądarki, 232
 strzałkowe, 380
 testowe, 309

G

generatory, 385
 getter, 196, 396
 głębokie linkowanie, 105
 gniazdo, 290
 Grunt, 36
 Gulp, 36

H

hierarchia
 tras, 102
 wstrzykiwaczy, 148

I

identyfikator produktu, 154
 implementacja
 interfejsu, 435
 komunikacji komponentów, 193
 powiadomień o ofertach, 294
 wyszukiwania produktów, 294
 wzorca Mediator, 205
 inferencja typów, 418
 instalowanie, 433
 Jasmine, 309
 plików definicji typów, 223, 440
 instancja komponentu, 32
 interfejs
 API, 219, 264
 Forms API, 234, 235, 257
 RESTful API, 272
 Testing API, 312
 użytkownika, 221
 interfejsy wywoływalne, 436
 iterowanie, 391

J

Jasmine, 23, 306
 instalacja, 309
 uruchamianie testów, 307
 JavaScript, 35
 język
 Dart, 412
 JavaScript, 35
 TypeScript, 35, 266, 411
 jQuery, 23
 JSLint, 36
 jspm, 36, 64

K

Karma
 uruchamianie testów, 325
 klasa, 393, 423
 FormArray, 240
 FormBuilder, 246, 252
 FormControl, 240
 FormGroup, 240
 LoginGuard, 108
 OpaqueToken, 147
 Product, 157
 ProductService, 157, 295
 Review, 157
 klasy
 jako interfejsy, 438
 testowe, 309
 kompilator, 414
 tsc, 357
 komponent, 32, 52, 201
 ApplicationComponent, 122
 HomeComponent, 91, 121, 212, 297
 pogodowy, 321
 ProductDescriptionComponent, 104
 ProductDetailComponent, 92, 103, 120, 154
 ProductDetailComponentParam, 98
 SellerInfoComponent, 104
 komponenty
 cykl życia, 210
 potomne, 208
 testowe, 309
 komunikacja
 klient-serwer
 protokół WebSocket, 283
 między komponentami, 194
 usługi z serwerem, 290
 w pełnym duplexie, 284
 w półduplexie, 284
 z serwerami, 263
 konfiguracja
 ładowarki SystemJS, 316
 produkcyjna, 354
 programistyczna, 353
 SystemJS, 60
 walidatorów, 252
 wstępna projektu, 71
 konstruktory, 394
 kontenery, 194
 kontrakt kodu, 433

L

leniwe ładowanie, 117
 linkowanie głębokie, 105
 literały szablonów, 372
 lokalizacja, 87

Ł

ładowanie
 kodu, 37, 271
 modułów dynamicznie, 406
 ładownia modułów, 343
 ES6, 406
 SystemJS, 55
 ładownia wstępne, 347
 łańcuchy znaków, 148, 372
 szablonów, 374
 wieloliniowe, 373
 łączenie obietnic w łańcuch, 401

M

matcher, 307
 toBeAnInstanceOf(), 311
 toBePromise(), 311
 toContainError(), 311
 toHaveCssClass(), 311
 toHaveCssStyle(), 311
 toHaveText(), 311
 toImplement(), 311
 toThrowErrorWith(), 311
 mechanizm wykrywania zmian, 217
 menedżer pakietów, 36, 62
 Bower, 36
 jspm, 36, 64
 npm, 36, 64
 metadane, 438
 metoda, 427
 Array.fill(), 223
 forEach(), 391
 getProducts(), 212
 navigate(), 94
 ngAfterContentChecked(), 211
 ngAfterContentInit(), 211
 ngAfterViewChecked(), 212
 ngAfterViewInit(), 211
 ngOnChanges(), 212, 215–217
 ngOnInit(), 212
 onSearch(), 260

metody
 pobierające, 196, 396
 ustawiające, 196, 396
 minifikatory, 37
 model, 235, 240
 pull, 174
 push, 174
 modularyzacja, 27
 moduły, 51, 115, 403
 leniwie ładowane, 117
 modyfikacja modułu głównego, 125
 modyfikatory dostępu, 425
 monitorowanie wersji programistycznej, 336

N

narzędzia programisty, 35
 narzędzie
 Karma, 325, 367
 TSLint, 37, 441
 Webpack, 335
 nawigacja, 85
 bloki konstrukcyjne, 89
 metoda navigate(), 94
 oparta na interfejsie History API, 88
 oparta na znaku kratki, 88
 w kliencie, 89
 nazwy plików, 343
 Node, 24, 36, 266
 framework Angular, 273
 npm, 36, 64
 rozpoczynanie projektu, 65
 uruchomienie aplikacji, 71

O

obiekt
 błędu, 248
 Http, 276
 Request, 265
 WebSocket, 287
 obiekty
 DOM, 164
 iterowalne, 176
 mutowalne, 213
 niemutowalne, 213
 obserwowalne, 176
 obietnice, 264, 398
 ES6, 399
 obserwatory, 174

obserwowalne strumienie zdarzeń, 176
obsługa

- błędów 404, 97
- interfejsu Forms API, 235, 257
- Shadow DOM, 105
- WebSocket, 275

odpakowywanie obiektów obserwowalnych, 278

Odwroćenie Sterowania, 131

ograniczenia wyszukiwania, 298

opakowywanie usługi w strumień obserwowalny, 288

operator

- Elvis, 322
- rozwijania, 126, 383
- reszty, 383

outlet, 86, 112

P

pakiet @angular/http, 275

panel rozszerzenia Augury, 82

parametry

- domyślne, 420
- opcjonalne, 375, 420

pętle

- for-in, 391
- for-of, 392

pierwsza aplikacja, 44

plik

- angular2-webpack-starter/package.json, 356
- angular2-webpack-starter/typings.d.ts, 357
- app.component.ts, 69, 93
- app.module.ts, 69, 125, 155, 317
- app.routing.ts, 92, 316
- app.spec.ts, 317
- application.html, 77, 123
- application.spec.ts, 329
- application.ts, 77, 122
- attribute-vs-property.ts, 166
- auction/client/karma.conf.js, 367
- auction/client/karma-test-runner.js, 367
- auction/client/package.json, 369
- auction/client/webpack.test.config.js, 368
- auction/package.json, 363
- auction-rest-server.ts, 272
- auction-unit-tests.html, 328
- basic-ng-content.ts, 206
- basic-webpack-starter/index.html, 351
- basic-webpack-starter/package.json, 350
- basic-webpack-starter/vendor.ts, 348

basic-webpack-starter/webpack.config.js, 349

billing.js, 407

client/app/main.ts, 276

custom-observable-service.ts, 288

custom-observable-service-subscriber.ts, 289

exposing-child-api.ts, 220

fatArrow.html, 382

filter-pipe.ts, 188

home.css, 122

home.ts, 121, 189

index.html, 44, 339

input_property_binding.ts, 195

karma.conf.js, 325

karma-test-runner.js, 326

luxury.lazy.module.ts, 119

luxury.module.ts, 115

main.js, 48, 339

main.ts, 46, 93, 142

main_aux.ts, 113

main-asyncpipe.ts, 279

main-basic.ts, 138

main-child.ts, 102

main-form.ts, 280

main-luxury.ts, 116

main-luxury-lazy.ts, 118

main-navigate.ts, 95

main-param.ts, 98

main-with-guard.ts, 109

main-with-service.ts, 282

mediator.ts, 203

moduleLoader.html, 407

my-express-server.ts, 270

ng-content-selector.ts, 209

ng-onchanges-with-param.ts, 213

observable-events.ts, 179

observable-events-http.ts, 181

order.ts, 204

output-property-binding.ts, 197

package.json, 66, 271, 275

pipe-tester.ts, 186

price-quoter.ts, 202

product.ts, 139

product-detail.html, 156, 226

product-detail.ts, 121, 155, 227

product-item.css, 124

product-item.html, 79, 124

product-item.ts, 79, 124

product-service.spec.ts, 330

product-service.ts, 139, 282

rest.html, 383

plik

- shipping.js, 407
- simple-websocket-client.html, 286
- simple-websocket-server.ts, 285
- spread.html, 385
- stars.html, 81
- stars.spec.ts, 331
- stars.ts, 80, 224
- stock.ts, 202
- systemjs.config.js, 68, 73
- temperature-pipe.ts, 186
- template-binding.ts, 169
- test.html, 314
- thisAndThat.html, 381
- tsconfig.json, 68, 268, 416
- two-way-binding.ts, 172
- two-way-websocket-client.html, 292
- weather.service.spec.ts, 320
- weather.service.ts, 319
- weather.spec.ts, 323
- weather.ts, 321
- webpack.config.js, 340
- websocket-observable-service.ts, 290
- websocket-observable-service-subscriber.ts, 291

pliki

- definicji typów, 357, 439
- konfiguracyjne
 - oddzielne, 276
 - wspólne, 274
- testowe, 308

- pobieranie notowań akcji, 180

podejście

- oparte na szablonach, 234
- reaktywne, 234

pola

- brudne, 253
- czyste, 253
- dotknięte, 252
- niedotknięte, 252
- oczekujące, 253

- polecenia CLI, 361

polecenie

- npm run build, 351
- npm start, 352

- Polymer, 24

- postładowarki, 347

- pośredniczenie w komunikacji, 201

- potok AsyncPipe, 278

- potoki niestandardowe, 184, 185

- powiadomienia o ofertach, 294

- ProductDetailComponent

- modyfikacja komponentu, 154

- ProductItemComponent

- dyrektywa RouterLink, 123

- programowanie reaktywne, 174

- protokoły HTTP, 263

- protokół WebSocket, 283

- przeciążanie funkcji, 424

- przekazywanie danych do tras, 97, 100

- przełączanie wstrzykiwaczy, 141

- przepływ pracy klient-serwer, 280

- przetwarzanie asynchroniczne, 398

R

- React, 24

- refaktoryzacja, 246

- formularza, 245

- kodu, 121

- szablonu, 259

- rekompilacja, 271

- renderowanie produktów, 142

- RESTful API, 272

- router

- testowanie, 316

- routing, 86

- rozglaszanie ofert, 299

- strona klienta, 300

- strona serwera, 301

- rozwiązywanie modułów, 441

- rozwój oparty na zachowaniach, 306

- RxJS, 24

- rzutowanie na wiele obszarów, 207

S

- semantyczne typy wejściowe, 234

- serwer

- Node

- Angular, 273

- komunikacja z usługą WebSocket, 290

- obsługa wyszukiwania produktu, 298

- uruchamianie, 362

- wysyłanie danych, 284

- zasoby statyczne, 274

- webpack-dev-server, 341

- WWW, 266

- setter, 196, 396

- Shadow DOM, 105

sieć

- CDN, 45
- unpkg, 45
- składnia wiązania zdarzeń, 163
- składowe
 - instancji, 427
 - klasy, 424
 - statyczne, 427
- skrypty npm, 276
- słowo kluczowe
 - const, 378
 - export, 404
 - function, 396
 - implements, 434
 - import, 404
 - let, 378
 - super, 397
- SPA, single-page application, 85
- specyfikacja ECMAScript 6, 371
- sprawdzanie
 - poprawności formularza, 255
 - statusu pola, 252
- status pola, 252
- strategie
 - lokalizacji, 87
 - wykrywania zmian, 218
- strona główna, 74
- strumienie
 - obserwowalne, 174, 264, 287
 - anulowanie, 181
 - zdarzeń
 - obserwowalne, 176
- strzeżenie tras, 107
- SystemJS, 36, 55, 57, 316
 - konfiguracja, 60
 - ładowanie skryptów, 58
 - opcja baseUrl, 60
 - opcja defaultJSExtensions, 60
 - opcja map, 61
 - opcja packages, 61
 - opcja paths, 61
 - opcja transpiler, 62
 - opcja typescriptOptions, 62
- szablon walidacji formularza rejestracji, 254
- szablony HTML, 169

Ś

- ścieżki względne, 345
- środowisko uruchomieniowe Node.js, 24

T

- tester, 324
- Testing API, 312
- testowalność komponentów, 132
- testowanie
 - aplikacji pogodowej, 314
 - komponentów, 313, 329, 331
 - komponentu pogodowego, 321
 - nawigacji routera, 312
 - routera, 316
 - usług, 312, 330
 - usługi pogodowej, 319
- testy
 - end-to-end, 306
 - jednostkowe, 306, 328
 - obciążeniowe, 306
- token, 132
- Traceur REPL, 379
- transkompilacja
 - klasy, 424
 - kodu, 36
 - TypeScriptu, 415
 - w locie, 95
- transkompilatory, 413
- trasa, 91
 - opisu produktu, 101
 - Szczegóły produktu, 99
- trasy podrzędne, 101
- tryb produkcyjny, 216
- TSLint, 37
 - kontrolowanie stylu kodu, 441
- tworzenie
 - aplikacji aukcji internetowych, 70
 - aplikacji SPA, 112
 - aplikacji TypeScript-Angular, 441
 - komponentu HomeComponent, 121
 - komponentu ProductDetailComponent, 120
 - konfiguracji produkcyjnych, 353
 - konfiguracji programistycznych, 353
 - metody, 427
 - modelu formularza, 258
 - paczek, 335
 - podstawowej konfiguracji Webpack, 348
 - serwera WWW, 266
 - strony głównej, 74
 - TypeScript, 35, 266, 411
 - dodawanie metadanych, 438
 - dziedziczenie klas, 428
 - funkcje, 419

TypeScript
 instalacja kompilatora, 414
 interfejsy, 433
 klasy, 423
 metody, 427
 modyfikatory dostępu, 425
 parametry domyślne, 420
 parametry opcjonalne, 420
 pliki definicji typów, 439
 typy niestandardowe z interfejsami, 433
 typy opcjonalne, 417
 typy sparametryzowane, 430

typy
 generyczne, 32
 niestandardowe z interfejsami, 433
 opcjonalne, 417
 sparametryzowane, 430

U

udostępnianie interfejsu API, 219

uruchamianie
 aplikacji, 50, 71
 aplikacji aukcji internetowych, 126
 serwera Node, 362
 testów, 324, 333
 zadań
 Grunt, 36
 Gulp, 36

usługa
 pogodowa, 319
 ProductService, 212
 WebSocket, 290

używanie
 dziedziczenia, 429
 ładówek, 344
 obietnicy, 400
 typu sparametryzowanego, 430

W

walidacja, 233
 formularza, 247
 formularza rejestracyjnego, 250, 254
 formularzy opartych na szablonach, 254
 formularzy reaktywnych, 247
 walidator Validators.minLength(), 248

walidatory
 asynchroniczne, 251
 grup, 249

niestandardowe, 249
 standardowe, 248

wartości domyślne, 375

wdrażanie aplikacji, 335

wdrożenia dev i prod, 338

Webpack, 38, 335, 338, 362
 podstawowa konfiguracja, 348

WebSocket, 263
 komunikacja klient-serwer, 283
 rozgłaszanie ofert, 299
 zamiana obiektu w strumień, 287

wiązanie
 atrybutu, 163, 167
 danych, 55, 162
 dwukierunkowe, 27, 162, 171
 HTML z modelem, 246
 szablonu, 163, 168, 170
 właściwości, 163, 165
 z innerHTML, 210
 z właściwościami i atrybutami, 164
 ze zdarzeniami, 163

właściwości, 164
 wejściowe, 195
 wyjściowe, 197

właściwość
 useFactory, 144
 useValue, 144
 viewProviders, 150

włączanie trybu produkcyjnego, 216

wstrzykiwacze, 133
 hierarchia, 148
 przełączanie, 141

wstrzykiwanie
 obiektu HTTP do usługi, 140, 280, 282
 usługi produktowej, 137
 zależności, DI, 29, 129, 294

wtyczki, 347

wybór menedżera pakietów, 62

wydajność, 34

wykrywanie zmian, 217

wyodrębnianie parametrów, 97

wyrażenia
 funkcji strzałkowych, 421
 lambda, 380, 421

wyszukiwanie produktów, 294
 dostarczanie wyników, 297
 obsługa na serwerze, 298
 protokół HTTP, 295
 testowanie funkcjonalności, 299

wywołania zwrotne, 399

wzbogacanie formularza HTML, 238

wzorzec

Mediator, 201

Odwrócenie Sterowania, 131

Wstrzykiwanie Zależności, 130

Z

zagnieżdżone funkcje wywołań zwrotnych, 399

zakres

bloku dla funkcji, 380

zmiennych, 376

zamiana obiektu WebSocket, 287

zdarzenia niestandardowe, 197

zestaw testowy, 307

zmiana szablonów, 205

zmiennie

statyczne, 395

z zakresem bloku, 378

znacznik

<input>, 164

<ng-content>, 207

<router-outlet>, 86

<script>, 56

reference, 441

znak

kratki, 88

strzałki, 380

Ż

żądanie GET, 276

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

Tworzenie aplikacji w języku TypeScript z wykorzystaniem frameworka Angular 2 jest dziś jednym z najwydajniejszych sposobów rozwijania średnich i dużych aplikacji internetowych. Takie aplikacje można bez problemu uruchamiać w każdej nowoczesnej przeglądarce, również na platformach mobilnych. Separacja kodu interfejsu od logiki aplikacji, prosta modularyzacja aplikacji, bezproblemowe przetwarzanie danych asynchronicznych, świetne narzędzia i nowoczesne komponenty interfejsu użytkownika — to tylko kilka z wielu zalet tandemu Angular 2 – TypeScript.

Ta książka jest przeznaczona dla programistów, którzy korzystają z Angular JS lub pracują z innym frameworkiem i potrafią kodować w JavaScriptcie. Przedstawiono tu zagadnienia związane z danymi i widokami, interakcjami użytkowników z formularzami i komunikacją z serwerami, a także sposoby testowania i wdrażania aplikacji napisanej w Angular 2. Wyjaśniono działanie routera Angular, techniki wstrzykiwania zależności, wiązania i potoki. Nie zabrakło szczegółowego opisu korzystania z formularzy oraz komunikacji z serwerami. Co istotne, autorzy położyli duży nacisk na stosowanie dobrych praktyk w programowaniu.

Najważniejsze zagadnienia:

- Przegląd architektury Angular 2 i metodyka budowy aplikacji
- Składnia języka TypeScript i kompilacja kodu TypeScript na JavaScript (ECMAScript 5)
- Programowanie reaktywne z obserwowalnymi strumieniami
- Wzorzec projektowy Mediator i cykl życia komponentu
- Automatyzacja procesów kompilacji i wdrażania
- Narzędzia i biblioteki przydatne w pracy z Angular 2

Angular 2: nowoczesne narzędzie dla najlepszych projektantów!

Yakov Fain — mistrz Javy i autor wielu książek o rozwijaniu oprogramowania. Jest współzałożycielem dwóch firm: Farata Systems i SuranceBay. Chętnie prowadzi warsztaty, podczas których zdradza sekrety frameworka Angular i platformy Java.

Anton Moiseev — jest głównym programistą w firmie SuranceBay. Od 10 lat tworzy aplikacje w technologiach Java i .NET. Koncentruje się na najlepszych praktykach technologii internetowych. Prowadził wiele sesji szkoleniowych dotyczących frameworków Angular JS i Angular 2.



księgarnia internetowa



<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>

Sprawdź najnowsze promocje:
● <http://helion.pl/promocje>
Książki najchętniej czytane:
● <http://helion.pl/bestsellery>
Zamów informacje o nowościach:
● <http://helion.pl/nowości>

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ



KOD KORZYŚCI

ISBN 978-83-283-3638-4



cena: 79,00 zł