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, Pazdrijowski, Zaprucki

ISBN: 978-83-283-3638-4

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.

Spis treści

2 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 TypeScripcie, 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 ładuje 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: {                                    <-- Mapuje nazwy modułów Angular na ich lokalizacje CDN.
        '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: {   <-- Określa skrypt główny dla każdego modułu Angular.
        '@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');   <-- Instruuje SystemJS, aby ładować moduł główny z pliku main.ts.
  </script>
</head>
<body>
  <hello-world></hello-world>   <-- Niestandardowy element HTML <hello-world></hello-world> reprezentuje komponent, który jest zaimplementowany w pliku main.ts.
</body>
</html>
```

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

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

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

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

Właściwość template definiuje znaczniki HTML dla renderowania tego komponentu.

Adnotowana klasa HelloWorldComponent reprezentuje komponent.

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

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 cudzysłowów wewnątrz szablonu i łamać 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 komponencie 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 = {}));

// Moduł
(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 aplikacji
(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 transkompilator 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 skrytpu 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';
  }
}

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

// Funkcja inicjująca aplikacji
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
```

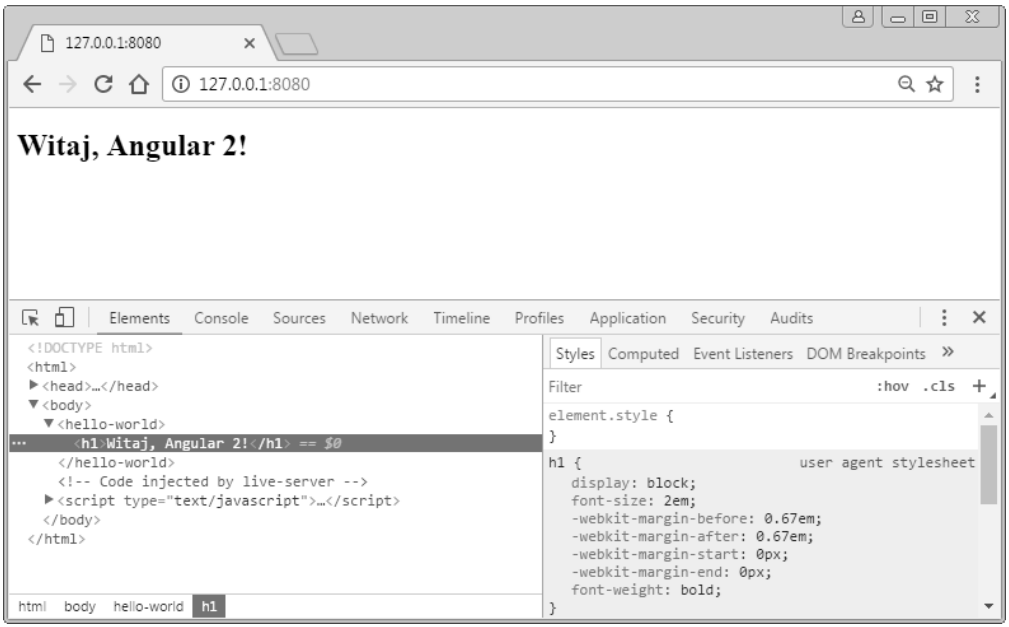
Wolimy odświeżanie okna przeglądarki na bieżąco, więc zainstaluj i uruchom `live-server` przy użyciu podobnej procedury:

```
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.

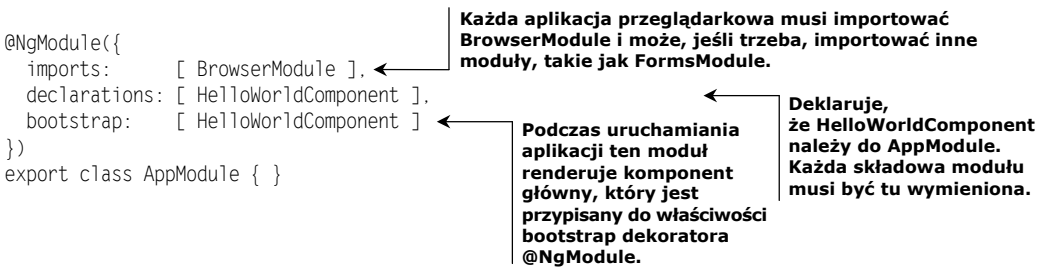
2.2.1. Moduły

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:



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 ładować 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 HelloComponent {}
```

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 przyjrzysz 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 cudzysłowów. 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 powiązanym 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.setElementStyle(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 dziewięciu 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 *modułem* 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 ładuje 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 źródła 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 modułem:

```
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 nastepujące 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 oryginalne 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*. Zróbmy 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 modułu.

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.

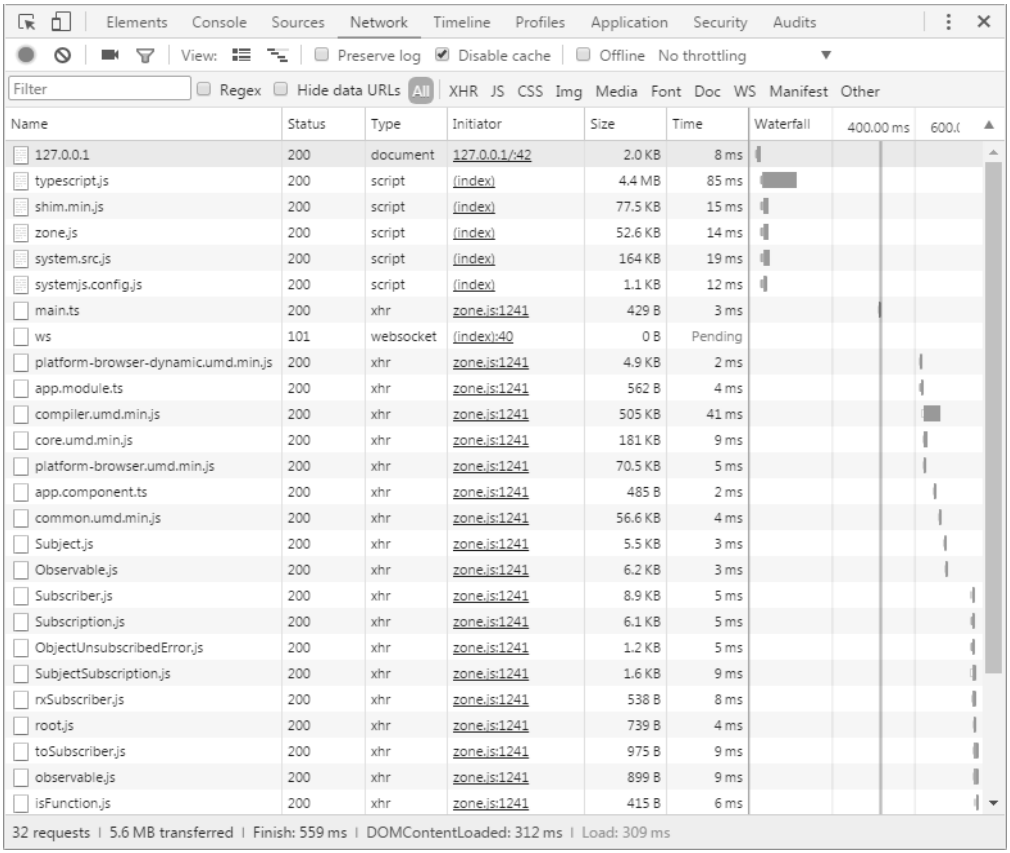
Nie przejmuj się rozmiarem pobierania. Zoptymalizujemy to w rozdziale 10. Ponieważ używamy serwera `live-server`, zaraz po zmodyfikowaniu i zapisaniu kodu tej aplikacji nastąpi odświeżenie strony w przeglądarce i załadowana zostanie najnowsza wersja kodu. Zastosujmy teraz to, czego się nauczyłeś, w aplikacji, 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 Placehold.it (*http://placehold.it*), która generuje elementy zastępcze o określonych rozmiarach. Aby



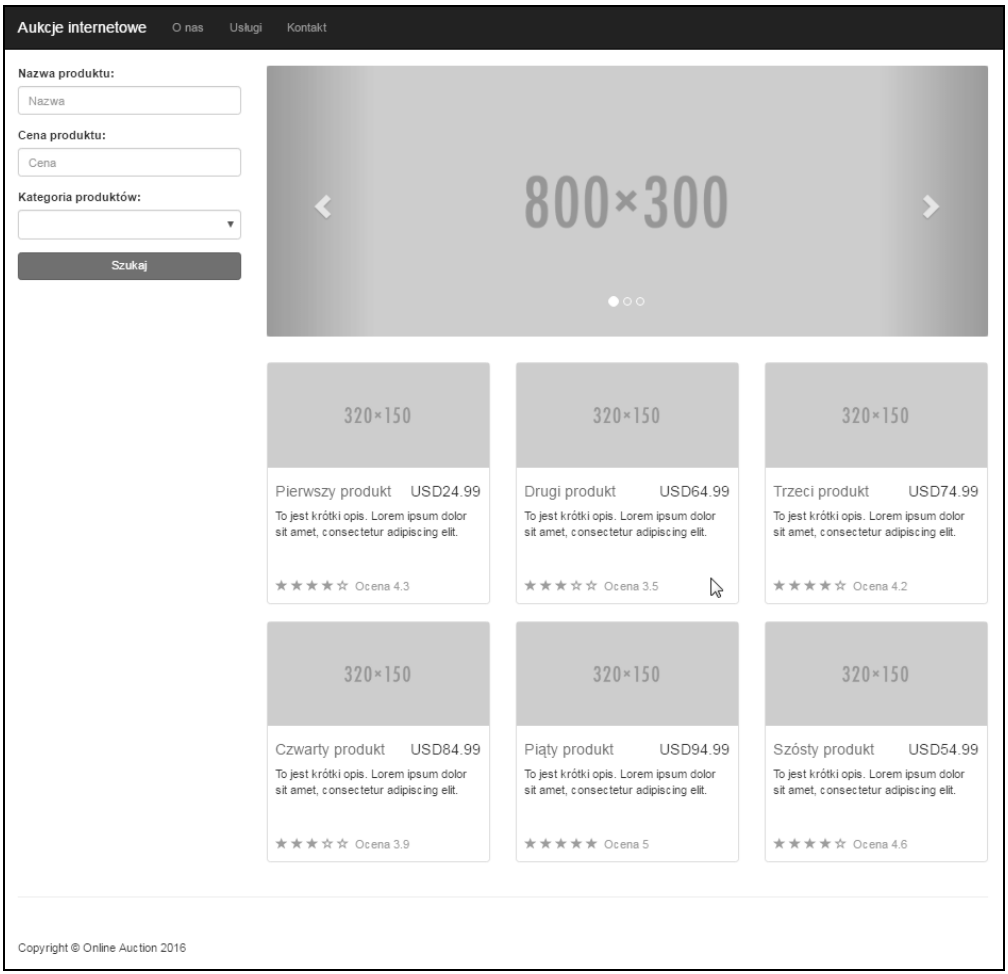
Rysunek 2.2. Uruchomienie aplikacji z projektu zarządzanego przez npm

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.

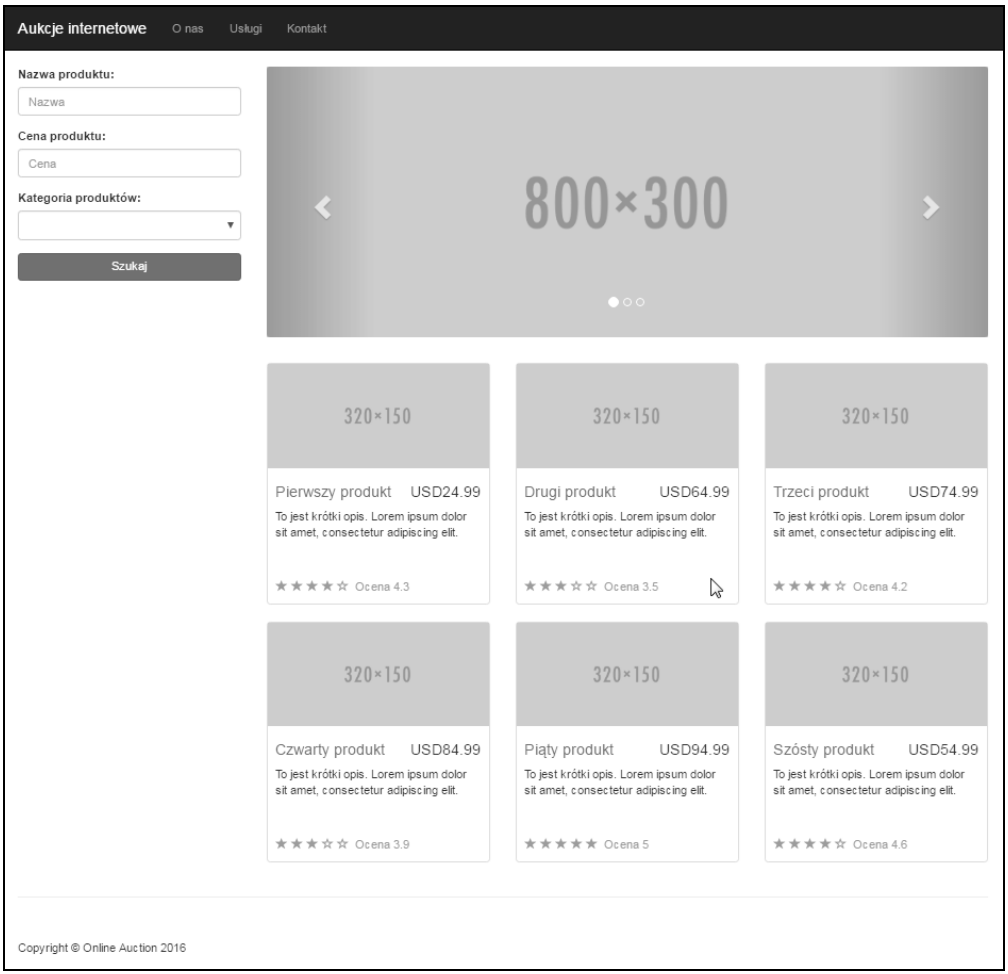
Plik *systemjs.config.js* musi zostać załączony w pliku *index.html*, tak jak pokazano w następnym punkcie w listingu 2.16. Ta konfiguracja pakietu `app` pozwala na użycie w pliku *index.html* linii `<script>System.import('app')</script>`, która załaduje zawartość *app/main.ts*.

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>                    Dodaje CSS biblioteki Bootstrap.
  <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>
                                                             Dodaje Bootstrap
                                                             i jQuery do obsługi
  <script src="node_modules/core-js/client/shim.min.js"></script>  komponentu karuzeli.
  <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>                                                  Ładuje main.ts zgodnie
</head>                                                      z konfiguracją w pliku
<body>                                                       systemjs.config.js.
<auction-application></auction-application>
</body>
</html>
```

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);
```

Zaktualizujmy 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 ApplicationComponent 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: [ ApplicationComponent,
                    CarouselComponent,
                    FooterComponent,
                    NavbarComponent,
                    ProductItemComponent,
                    SearchComponent,
                    StarsComponent],
    providers:    [ProductService],
    bootstrap:    [ ApplicationComponent ]
})
export class AppModule { }
```

Deklaruje wszystkie komponenty, których będzie używał moduł.

Deklaruje dostawcę dla usługi ProductService, którą nieco później wstrzykniemy do komponentu ApplicationComponent.

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. Przejrzymy pierwsze dwa pliki.

Utwórzmy komponent `ApplicationComponent` 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';

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

export default class ApplicationComponent {

  products: Array<Product> = [];

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

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

Zmienia klasę ApplicationComponent 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.

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

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

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

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

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 `ViewEncapsula ↪tion.None`, aby style z pliku *.css* aplikacji zastosować nie tylko do komponentu `ApplicationComponent`, 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 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>
</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 znajdującej 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, zawęź 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 `ApplicationComponent` 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">
  <img src="http://placehold.it/320x150">
  <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.

Rysunek 2.5. Komponent gwiazdek

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

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 `ApplicationComponent`. 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.

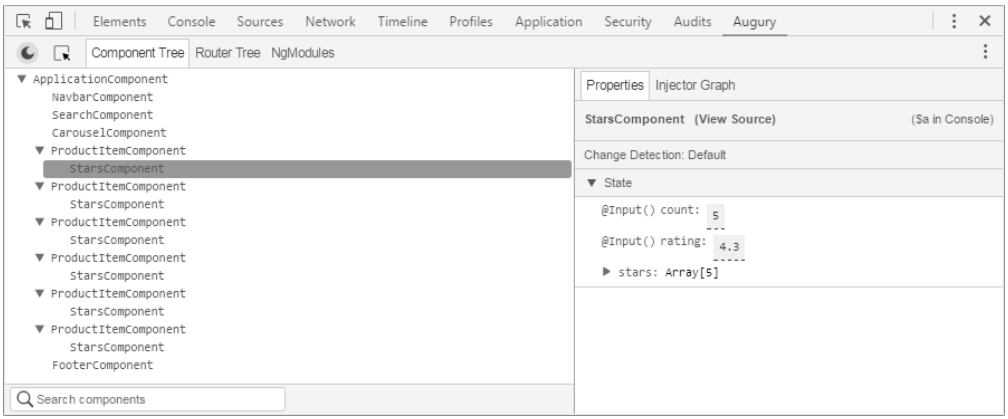
- Katalog *services* zawiera plik *product-service.ts*, który deklaruje dwie klasy, `Product` i `ProductService`. Stąd pochodzą dane dla aukcji. Więcej szczegółów na temat zawartości tego pliku podamy w części praktycznej w rozdziale 3.
- Katalog *navbar* zawiera kod dla górnego paska nawigacyjnego.
- 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

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

Z

Ż

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 JavaScripcie. 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.

ISBN 978-83-283-3638-4

9 788328 336384

cena: 79,00 zł