

AngularJS

Profesjonalne techniki

Adam Freeman

Tytuł oryginału: Pro AngularJS

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-0197-9

Original edition copyright © 2014 by Adam Freeman.
All rights reserved.

Polish edition copyright © 2015 by HELION SA.
All rights reserved.

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

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

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

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

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

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

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

Printed in Poland.

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

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

Spis treści

	O autorze	15
	O recenzencie technicznym	16
Część I	Zaczynamy	17
Rozdział 1.	Zaczynamy	19
	Wymagania	19
	Jaka jest struktura niniejszej książki?	20
	Część I. Zaczynamy	20
	Część II. Praca z AngularJS	20
	Część III. Praca z modułami i usługami	20
	Czy w książce znajdę wiele przykładów?	20
	Gdzie znajdę przykładowe fragmenty kodu?	23
	Jak skonfigurować środowisko programistyczne?	23
	Wybór przeglądarki internetowej	23
	Wybór edytora tekstów	24
	Instalacja Node.js	24
	Instalacja serwera WWW	25
	Instalacja systemu przeznaczonego do testowania	25
	Utworzenie katalogu AngularJS	26
	Przeprowadzenie prostego testu	28
	Podsumowanie	29
Rozdział 2.	Pierwsza aplikacja w AngularJS	31
	Przygotowanie projektu	31
	Użycie AngularJS	33
	Dodanie biblioteki AngularJS do pliku HTML	33
	Utworzenie modelu danych	35
	Utworzenie kontrolera	37
	Utworzenie widoku	39
	Wyjście poza podstawy	42
	Użycie dwukierunkowego dołączania modelu	42
	Zdefiniowanie i stosowanie funkcji w kontrolerach	44

Użycie funkcji w zależności od innych zdefiniowanych funkcji	46
Reakcja na działania użytkownika	48
Filtrowanie i zmiana kolejności danych modelu	51
Pobieranie danych z wykorzystaniem technologii Ajax	55
Podsumowanie	57
Rozdział 3. Umieszczenie AngularJS w kontekście	59
Sytuacje, w których AngularJS sprawdza się doskonale	59
Poznanie aplikacji dwukierunkowych oraz w postaci pojedynczej strony	60
Wzorzec MVC	62
Model	63
Kontroler	65
Widok	65
Usługi typu RESTful	66
Najczęstsze pułapki podczas stosowania wzorca MVC	68
Umieszczenie logiki w nieodpowiednim miejscu	68
Przyjęcie formatu danych stosowanego w magazynie danych	68
Kurczowe trzymanie się starych sposobów	69
Podsumowanie	69
Rozdział 4. Wprowadzenie do HTML i Bootstrap CSS	71
Poznajemy HTML	72
Anatomia elementu HTML	73
Atrybuty	74
Zawartość elementu	74
Elementy typu void	75
Struktura dokumentu	75
Poznajemy framework Bootstrap	77
Stosowanie podstawowych klas Bootstrap	79
Użycie frameworka Bootstrap do nadawania stylu tabeli	80
Użycie frameworka Bootstrap do tworzenia formularzy sieciowych	83
Użycie frameworka Bootstrap do utworzenia układu opartego na siatce	85
Podsumowanie	89
Rozdział 5. Wprowadzenie do JavaScript	91
Przygotowanie przykładowego projektu	92
Element <script>	93
Polecenia w JavaScript	94
Definiowanie i użycie funkcji	94
Definiowanie funkcji z parametrami	95
Definiowanie funkcji zwracającej wartość	96
Wykrywanie funkcji	97
Użycie zmiennych i typów	98
Użycie typów podstawowych	99
Tworzenie obiektów	101
Praca z obiektami	104
Użycie operatorów JavaScript	110
Użycie konstrukcji warunkowych	110
Operatory równości i identyczności	111
Jawna konwersja typów	114

Tablice	116
Użycie stylu dosłownej tablicy	117
Wykrywanie tablicy	117
Odczyt i modyfikacja zawartości tablicy	118
Wyświetlenie zawartości tablicy	119
Użycie wbudowanych metod przeznaczonych do pracy z tablicami	119
Porównywanie wartości undefined i null	120
Sprawdzenie pod kątem wartości null lub undefined	122
Obietnice	124
Format JSON	128
Podsumowanie	129
Rozdział 6. SportsStore — przykładowa aplikacja	131
Rozpoczęcie pracy	132
Przygotowanie danych	132
Przygotowanie aplikacji	135
Wyświetlenie przykładowych danych produktu	138
Utworzenie kontrolera	138
Wyświetlanie informacji o produktach	140
Wyświetlenie listy kategorii	143
Utworzenie listy kategorii	143
Generowanie łączy nawigacji po kategoriach	144
Wybór kategorii	148
Podświetlenie wybranej kategorii	150
Dodanie stronicowania	152
Podsumowanie	156
Rozdział 7. SportsStore — nawigacja i zakupy	157
Przygotowanie przykładowego projektu	157
Użycie rzeczywistych danych produkcyjnych	157
Obsługa błędów Ajax	159
Utworzenie widoków częściowych	161
Utworzenie koszyka na zakupy	163
Zdefiniowanie modułu koszyka i usługi	163
Utworzenie widżetu koszyka na zakupy	165
Dodanie przycisku wyboru produktu	168
Dodanie nawigacji na podstawie adresu URL	170
Definiowanie tras URL	170
Użycie routingu adresów URL do zapewnienia nawigacji	173
Rozpoczęcie prac nad finalizacją zamówienia	174
Finalizacja zamówienia	176
Podsumowanie	177
Rozdział 8. SportsStore — zamówienia i administracja	179
Przygotowanie przykładowego projektu	179
Pobieranie danych adresowych	179
Dodanie formularza weryfikacji danych	181
Dodanie pozostałych elementów formularza sieciowego	185

Składanie zamówień	187
Rozbudowa serwera Deployd	187
Zdefiniowanie funkcji kontrolera	189
Wywołanie funkcji kontrolera	190
Zdefiniowanie widoku	190
Usprawnienia w aplikacji	191
Administrowanie katalogiem produktów	192
Przygotowanie serwera Deployd	192
Utworzenie aplikacji administracyjnej	194
Implementacja uwierzytelniania	195
Definiowanie widoku głównego i kontrolera	198
Implementacja funkcji przeglądania zamówień	200
Implementacja funkcji zmiany zawartości katalogu produktów	203
Podsumowanie	207

Część II Praca z AngularJS 209

Rozdział 9. Anatomia aplikacji AngularJS 211

Przygotowanie przykładowego projektu	212
Praca z modułami	213
Określenie granic aplikacji AngularJS	213
Użycie modułów do zdefiniowania komponentów AngularJS	214
Definiowanie kontrolera	215
Definiowanie dyrektywy	221
Definiowanie filtra	223
Definiowanie usługi	226
Użycie modułów do organizacji kodu	229
Cykl życiowy modułu	232
Podsumowanie	233

Rozdział 10. Użycie dyrektyw dołączania i szablonów 235

Kiedy i dlaczego należy używać dyrektyw?	236
Przygotowanie przykładowego projektu	236
Użycie dyrektyw dołączania danych	237
Przeprowadzenie (lub uniknięcie) jednokierunkowego dołączenia danych	239
Przeprowadzenie dwukierunkowego dołączenia danych	241
Użycie szablonów dyrektyw	243
Generowanie powtarzających się elementów	244
Generowanie wielu elementów najwyższego poziomu	250
Praca z widokami częściowymi	251
Użycie dyrektywy ng-include jako atrybutu	255
Warunkowe zastępowanie elementów	256
Ukrycie nieprzetworzonych osadzonych szablonów wyrażeń dołączania danych	259
Podsumowanie	261

Rozdział 11. Użycie dyrektyw elementów i zdarzeń 263

Przygotowanie przykładowego projektu	264
Użycie dyrektyw elementu	264
Wyświetlanie, ukrywanie i usuwanie elementów	265
Zarządzanie klasami i stylami CSS	269

Obsługa zdarzeń	274
Utworzenie własnej dyrektywy zdarzeń	277
Zarządzanie atrybutami specjalnymi	279
Zarządzanie atrybutami boolowskimi	279
Zarządzanie innymi atrybutami	281
Podsumowanie	282
Rozdział 12. Praca z formularzami sieciowymi	283
Przygotowanie przykładowego projektu	284
Użycie dwukierunkowego dołączania danych w elementach formularza sieciowego	285
Niejawne tworzenie właściwości modelu	286
Sprawdzenie, czy obiekt modelu danych został utworzony	289
Weryfikacja formularzy sieciowych	290
Przeprowadzenie podstawowej weryfikacji formularza sieciowego	292
Wyświetlanie komunikatów w trakcie weryfikacji formularza sieciowego	296
Użycie CSS do nadania stylu komunikatom	297
Użycie zmiennych specjalnych w celu wyświetlania komunikatów	301
Wstrzymanie wyświetlania komunikatów	304
Użycie atrybutów dyrektywy formularza sieciowego	306
Użycie elementów <input>	306
Użycie elementu <textarea>	309
Użycie elementów <select>	310
Podsumowanie	314
Rozdział 13. Użycie kontrolerów i zakresów	315
Kiedy i dlaczego używać kontrolerów i zakresów?	316
Przygotowanie przykładowego projektu	316
Poznajemy podstawy	317
Utworzenie i zastosowanie kontrolera	317
Konfiguracja zakresu	318
Modyfikacja zakresu	320
Organizowanie kontrolerów	321
Użycie kontrolera monolitycznego	322
Ponowne użycie kontrolera	324
Dziedziczenie kontrolerów	328
Użycie wielu kontrolerów	335
Kontroler bez zakresu	337
Wyraźne uaktualnienie zakresu	338
Konfiguracja jQuery UI	338
Kontrola stanu przycisku	340
Zliczanie kliknięć przycisku	341
Podsumowanie	342
Rozdział 14. Użycie filtrów	343
Kiedy i dlaczego używać filtrów?	344
Przygotowanie przykładowego projektu	344
Pobieranie pliku lokalizacji	345
Filtrowanie pojedynczych wartości danych	346
Formatowanie wartości pieniężnych	347
Formatowanie innych wartości pieniężnych	348

Formatowanie dat	349
Zmiana wielkości liter ciągu tekstowego	352
Generowanie danych w formacie JSON	353
Lokalizacja danych wyjściowych filtru	354
Filtrowanie kolekcji	357
Ograniczenie liczby elementów	357
Wybór elementów	360
Sortowanie elementów	362
Łączenie filtrów	366
Utworzenie własnego filtru	367
Utworzenie filtru przeznaczonego do formatowania wartości daty	367
Utworzenie filtru kolekcji	369
Budowa filtru na bazie istniejącego filtru	371
Podsumowanie	372
Rozdział 15. Tworzenie własnych dyrektyw	373
Kiedy i dlaczego tworzyć własne dyrektywy?	374
Przygotowanie przykładowego projektu	374
Utworzenie własnej dyrektywy	375
Zdefiniowanie dyrektywy	375
Implementacja funkcji link	376
Zniesienie zależności właściwości danych	379
Obsługa zmiany danych	382
Praca z jqLite	387
Nawigacja po obiektowym modelu dokumentu	387
Modyfikacja elementów	391
Tworzenie i usuwanie elementów	393
Obsługa zdarzeń	396
Inne metody jqLite	397
Uzyskanie dostępu do funkcji AngularJS z poziomu jqLite	398
Zastąpienie jqLite przez jQuery	399
Podsumowanie	400
Rozdział 16. Tworzenie skomplikowanych dyrektyw	401
Przygotowanie przykładowego projektu	402
Definiowanie skomplikowanych dyrektyw	402
Definiowanie sposobu zastosowania dyrektywy	403
Użycie szablonu dyrektywy	407
Użycie funkcji jako szablonu	409
Użycie zewnętrznego szablonu	410
Wybór szablonu zewnętrznego za pomocą funkcji	411
Zastępowanie elementu	413
Zarządzanie zakresami dyrektywy	415
Utworzenie wielu kontrolerów	417
Zdefiniowanie oddzielnego zakresu dla każdego egzemplarza dyrektywy	418
Utworzenie odizolowanych zakresów	421
Podsumowanie	430

Rozdział 17. Zaawansowane funkcje dyrektyw	431
Przygotowanie przykładowego projektu	432
Użycie transkluzji	432
Użycie funkcji compile	435
Użycie kontrolerów w dyrektywach	438
Dodanie innej dyrektywy	441
Utworzenie własnych elementów formularza sieciowego	443
Obsługa zmian zewnętrznych	445
Obsługa zmian wewnętrznych	447
Formatowanie wartości danych	448
Weryfikacja własnych elementów formularza sieciowego	449
Podsumowanie	452
 Część III Usługi AngularJS	453
Rozdział 18. Praca z modułami i usługami	455
Kiedy i dlaczego należy tworzyć usługi i moduły?	455
Przygotowanie przykładowego projektu	456
Użycie modułów do nadania struktury aplikacji	458
Obsługa pojedynczego modułu	458
Utworzenie nowego modułu	460
Utworzenie i użycie usługi	461
Użycie metody factory()	461
Użycie metody service()	464
Użycie metody provider()	466
Użycie wbudowanych modułów i usług	468
Podsumowanie	469
 Rozdział 19. Usługi dla obiektów globalnych, błędów i wyrażeń	471
Przygotowanie przykładowego projektu	471
Uzyskanie dostępu do obiektów globalnych API DOM	471
Kiedy i dlaczego używać usług obiektu globalnego?	472
Uzyskanie dostępu do obiektu window	472
Uzyskanie dostępu do obiektu document	473
Użycie usług \$interval i \$timeout	474
Uzyskanie dostępu do adresu URL	475
Rejestracja danych	481
Wyjątki	482
Kiedy i dlaczego używać usługi \$exceptionHandler?	482
Praca z wyjątkami	483
Bezpośrednia praca z usługą \$exceptionHandler	483
Implementacja własnej procedury obsługi wyjątków	484
Praca z niebezpiecznymi danymi	485
Kiedy i dlaczego używać usług przeznaczonych do pracy z niebezpiecznymi danymi?	486
Wyświetlanie niebezpiecznych danych	486
Użycie niebezpiecznego mechanizmu dołączania danych	487
Wyraźne zaufanie danym	490

Praca z wyrażeniami i dyrektywami AngularJS	491
Kiedy i dlaczego używać usług wyrażen i dyrektyw?	492
Konwersja wyrażenia na funkcję	492
Interpolacja ciągów tekstowych	495
Kompilacja zawartości	498
Podsumowanie	499
Rozdział 20. Usługi dla technologii Ajax i obietnic	501
Kiedy i dlaczego używać usług Ajax?	502
Przygotowanie przykładowego projektu	502
Żądania Ajax	502
Wykonywanie żądania Ajax	505
Otrzymywanie odpowiedzi na żądanie Ajax	506
Konfiguracja żądań Ajax	509
Ustawienie wartości domyślnych żądania Ajax	513
Użycie interceptorów Ajax	515
Obietnice	516
Pobieranie i użycie obiektu deferred	518
Użycie obietnicy	519
Dlaczego obietnice nie są zwykłymi zdarzeniami?	521
Łączenie obietnic ze sobą	522
Grupowanie obietnic	524
Podsumowanie	526
Rozdział 21. Usługi dla REST	527
Kiedy i dlaczego używać usług typu REST?	528
Przygotowanie przykładowego projektu	528
Utworzenie usługi typu RESTful	528
Utworzenie aplikacji AngularJS	531
Użycie usługi \$http	536
Wyświetlenie danych produktu	536
Usunięcie produktu	538
Utworzenie produktu	538
Uaktualnienie produktu	539
Przetestowanie implementacji Ajax	539
Ukrycie żądań Ajax	539
Instalacja modułu ngResource	542
Użycie usługi \$resource	543
Konfiguracja akcji usługi \$resource	548
Utworzenie komponentu gotowego do użycia z usługą \$resource	549
Podsumowanie	551
Rozdział 22. Usługi dla widoków	553
Kiedy i dlaczego używać usług widoku?	553
Przygotowanie przykładowego projektu	554
Istota problemu	554
Użycie routingu URL	556
Instalacja modułu ngRoute	556
Definiowanie adresów URL tras	557

Wyświetlanie wybranego widoku	558
Połączenie kodu i znaczników HTML	559
Użycie parametrów trasy	562
Uzyskanie dostępu do tras i parametrów tras	564
Konfiguracja tras	567
Użycie kontrolerów z trasami	568
Dodanie zależności do tras	570
Podsumowanie	573
Rozdział 23. Usługi dla animacji i dotknięć	575
Przygotowanie przykładowego projektu	575
Animacja elementów	575
Kiedy i dlaczego używać usług animacji?	576
Instalacja modułu ngAnimation	576
Definiowanie i stosowanie animacji	577
Uniknięcie niebezpieczeństwa w postaci jednoczesnych animacji	579
Obsługa zdarzeń dotknięć	580
Kiedy i dlaczego używać zdarzeń dotknięć?	581
Instalacja modułu ngTouch	581
Obsługa gestu machnięcia	581
Użycie zamiennika dla dyrektywy ng-click	582
Podsumowanie	582
Rozdział 24. Usługi rejestracji komponentów i ich wstrzykiwania	583
Kiedy i dlaczego używać usług rejestracji komponentów i ich wstrzykiwania?	583
Przygotowanie przykładowego projektu	583
Rejestracja komponentów AngularJS	584
Zarządzanie wstrzykiwaniem zależności	586
Ustalenie zależności funkcji	586
Pobieranie egzemplarzy usługi	588
Uproszczenie procesu wywołania	589
Pobranie usługi \$injector z elementu głównego	590
Podsumowanie	591
Rozdział 25. Testy jednostkowe	593
Kiedy i dlaczego przeprowadzać testy jednostkowe?	594
Przygotowanie przykładowego projektu	594
Instalacja modułu ngMock	594
Utworzenie konfiguracji testowej	594
Utworzenie przykładowej aplikacji	596
Praca z Karma i Jasmine	597
Przeprowadzanie testów	599
Poznajemy atrapę obiektu	601
API i obiekty testowe	601
Testowanie kontrolera	602
Przygotowanie testu	603
Użycie atrapy obiektów	604
Symulacja odpowiedzi HTTP	604
Symulacja czasu	608
Testowanie rejestracji danych	610

Testowanie innych komponentów 612

 Testowanie filtru 612

 Testowanie dyrektywy 614

 Testowanie usługi 615

Podsumowanie 617

Skorowidz 619

ROZDZIAŁ 2



Pierwsza aplikacja w AngularJS

Najlepszym sposobem rozpoczęcia pracy z AngularJS jest po prostu utworzenie aplikacji sieciowej. W tym rozdziale zostanie zaprezentowany proces przygotowania prostej aplikacji. Na początku przygotujemy statyczną atrapę aplikacji docelowej, a następnie dodamy do niej funkcje AngularJS, tworząc w ten sposób prostą, dynamiczną aplikację sieciową. W rozdziałach od 6. do 8. zobaczysz, jak utworzyć znacznie bardziej skomplikowaną i realistyczną aplikację AngularJS. Prosty przykład z tego rozdziału jest w zupełności wystarczający do zademonstrowania najważniejszych komponentów aplikacji AngularJS i jednocześnie przygotowuje grunt dla materiału znajdującego się w pozostałych rozdziałach.

Przygotowanie projektu

W rozdziale 1. zobaczyłeś, jak przygotować i przetestować środowisko programistyczne, z którego korzystałem podczas pisania niniejszej książki. Jeżeli chcesz wypróbować zaprezentowane w niej przykłady, najwyższa pora, aby wspomniane środowisko było w pełni skonfigurowane i działało.

Na początek utworzymy atrapę aplikacji w postaci statycznego dokumentu HTML pokazującego docelową aplikację, nad którą będziemy pracować w tym rozdziale. Aplikacja to lista rzeczy do zrobienia. W katalogu *angularjs* utwórz nowy plik o nazwie *todo.html* i umieść w nim kod przedstawiony na listingu 2.1.

Listing 2.1. Początkowa zawartość pliku *todo.html*

```
<!DOCTYPE html>
<html data-ng-app>
<head>
  <title>Lista rzeczy do zrobienia</title>
  <link href="bootstrap.css" rel="stylesheet" />
  <link href="bootstrap-theme.css" rel="stylesheet" />
</head>
<body>
  <div class="page-header">
    <h1>Lista rzeczy do zrobienia użytkownika Adam</h1>
  </div>
  <div class="panel">
    <div class="input-group">
      <input class="form-control" />
      <span class="input-group-btn">
        <button class="btn btn-default">Dodaj</button>
      </span>
    </div>
  </div>
```

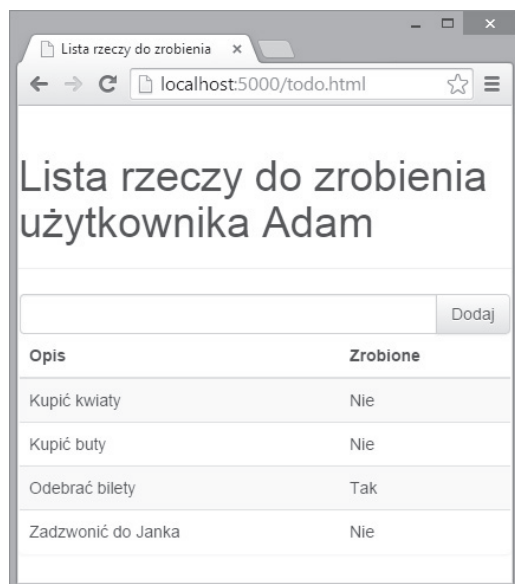
```

<table class="table table-striped">
  <thead>
    <tr>
      <th>Opis</th>
      <th>Zrobione</th>
    </tr>
  </thead>
  <tbody>
    <tr><td>Kupić kwiaty</td><td>Nie</td></tr>
    <tr><td>Kupić buty</td><td>Nie</td></tr>
    <tr><td>Odebrać bilety</td><td>Tak</td></tr>
    <tr><td>Zadzwoń do Janka</td><td>Nie</td></tr>
  </tbody>
</table>
</div>
</body>
</html>

```

■ **Wskazówka** Od tej chwili, o ile nie zostanie wskazane inaczej, wszystkie pliki umieszczaj w katalogu *angularjs*, który utworzyłeś w poprzednim rozdziale. Przedstawionych w książce przykładowych fragmentów kodu nie musisz wpisywać samodzielnie. Wszystkie przykłady można pobrać bezpłatnie ze strony poświęconej książce (<http://helion.pl/ksiazki/angupt.htm>). Przygotowane do pobrania przykłady są kompletne, pogrupowane rozdziałami i zawierają wszystkie pliki niezbędne do utworzenia oraz przetestowania omawianych aplikacji.

Tak przygotowany plik nie używa AngularJS. W rzeczywistości nie zawiera nawet elementu `<script>` odpowiedzialnego za import pliku *angular.js*. Wkrótce dodamy plik JavaScript i zaczniemy stosować funkcje oferowane przez AngularJS. Jednak na obecnym etapie plik *todo.html* zawiera statyczne elementy HTML tworzące szkielet atrapy aplikacji — na górze znajduje się nagłówek, poniżej tabela zawierająca listę rzeczy do zrobienia. Aby zobaczyć efekt dotychczasowej pracy, wyświetl plik *todo.html* w przeglądarce internetowej (patrz rysunek 2.1).



Rysunek 2.1. Wygenerowana w przeglądarce internetowej zawartość początkowego pliku *todo.html*

-
- **Uwaga** Aby zachować prostotę przykładu omawianego w tym rozdziale, cały kod aplikacji będzie umieszczony w pliku *todo.html*. W przypadku standardowych aplikacji AngularJS zwykle stosowana jest starannie wybrana struktura dla plików. Ponieważ tutaj nie tworzymy skomplikowanego przykładu, umieszczenie wszystkiego w pojedynczym pliku nie spowoduje żadnych utrudnień. W rozdziale 6. rozpoczniemy proces tworzenia znacznie bardziej zaawansowanej aplikacji AngularJS i wtedy poruszę temat struktury plików w kontekście budowanej aplikacji.
-

Użycie AngularJS

Statyczny kod HTML w pliku *todo.html* służy jako miejsce zarezerwowane dla podstawowej funkcjonalności, którą chcemy utworzyć. Użytkownik powinien mieć możliwość wyświetlenia listy rzeczy do zrobienia, zaznaczenia już wykonanych zadań oraz dodawania nowych do listy. W kolejnych punktach rozdziału do aplikacji dodamy bibliotekę AngularJS i pewne podstawowe funkcje, a tym samym ożywimy aplikację listy rzeczy do zrobienia. W celu zachowania prostoty przyjąłem założenie, że istnieje tylko jeden użytkownik i nie ma konieczności zachowywania stanu danych w aplikacji.

Dodanie biblioteki AngularJS do pliku HTML

Dodanie biblioteki AngularJS do pliku HTML jest bardzo łatwe. W dokumencie HTML wystarczy jedynie umieścić element `<script>` odpowiedzialny za import pliku *angular.js*, utworzyć moduł AngularJS oraz zastosować odpowiedni atrybut dla elementu `<html>`, jak przedstawiono na listingu 2.2.

Listing 2.2. *Utworzenie i zastosowanie modułu AngularJS w pliku todo.html*

```
<!DOCTYPE html>
<html ng-app="todoApp">
<head>
  <title>Lista zadań do zrobienia</title>
  <link href="bootstrap.css" rel="stylesheet" />
  <link href="bootstrap-theme.css" rel="stylesheet" />
  <script src="angular.js"></script>
  <script>
    var todoApp = angular.module("todoApp", []);
  </script>
</head>
<body>
  <div class="page-header">
    <h1>Lista rzeczy do zrobienia użytkownika Adam</h1>
  </div>
  <div class="panel">
    <div class="input-group">
      <input class="form-control" />
      <span class="input-group-btn">
        <button class="btn btn-default">Dodaj</button>
      </span>
    </div>
    <table class="table table-striped">
      <thead>
        <tr>
          <th>Opis</th>
          <th>Zrobione</th>
        </tr>
      </thead>
      <tbody>
        <tr><td>Kupić kwiaty</td><td>No</td></tr>
```

```

        <tr><td>Kupić buty</td><td>No</td></tr>
        <tr><td>Odebrać bilety</td><td>Yes</td></tr>
        <tr><td>Zadzwonić do Janka</td><td>No</td></tr>
    </tbody>
</table>
</div>
</body>
</html>

```

Aplikacje AngularJS są tworzone na podstawie co najmniej jednego modułu. Z kolei moduł jest tworzony za pośrednictwem wywołania `angular.module()`, jak pokazano w poniższym fragmencie kodu:

```

...
var todoApp = angular.module("todoApp", []);
...

```

Omówienie modułów znajdziesz w rozdziałach 9. i 18. Powyżej zobaczyłeś, jak utworzony został moduł dla aplikacji przedstawionej na listingu 2.2. Argumentami metody `angular.module()` są nazwa modułu przeznaczonego do utworzenia oraz tablica innych modułów, które będą niezbędne. W omawianym przykładzie utworzyłem moduł o nazwie `todoApp`, stosując tym samym nieco dezorientującą konwencję dodawania przyrostka `App` do nazwy modułu aplikacji. Drugi argument wywołania `angular.module()` to pusta tablica, która wskazuje, że do działania tworzonego modułu nie są wymagane żadne inne moduły. (Pewne funkcje AngularJS są dostępne w różnych modułach. Sposób tworzenia własnych modułów będzie przedstawiony w rozdziale 18.).

■ **Ostrzeżenie** Najczęściej popełniany błąd polega na pominięciu argumentu wskazującego zależności, co prowadzi do wygenerowania błędu. Koniecznie *musisz* podać argument określający zależności. Jeśli zależności nie są wymagane, to wskazujący je argument powinien mieć postać pustej tablicy. Temat używania zależności zostanie poruszony w rozdziale 18.

Nakazanie bibliotece AngularJS zastosowania modułu odbywa się za pomocą atrybutu `ng-app`. Sposób działania AngularJS polega na rozszerzeniu kodu znaczników HTML, co odbywa się przez dodawanie nowych elementów, atrybutów, klas i specjalnych komentarzy (te ostatnie są jednak rzadko stosowane). Biblioteka AngularJS dynamicznie kompiluje kod HTML w dokumencie w celu wyszukania i przetworzenia wymienionych dodatków, a następnie tworzy aplikację. Wbudowane funkcje AngularJS można uzupełniać kodem JavaScript w celu dostosowania do własnych potrzeb sposobu działania aplikacji oraz zdefiniowania własnych rozszerzeń dla kodu znaczników HTML.

■ **Uwaga** Stosowana przez AngularJS kompilacja w niczym nie przypomina kompilacji znanej z projektów tworzonych w językach C# lub Java, w których kompilator przetwarza kod źródłowy w celu wygenerowania danych wyjściowych możliwych do wykonania przez środowisko uruchomieniowe. Lepszym określeniem będzie stwierdzenie, że biblioteka AngularJS analizuje elementy HTML po wczytaniu zawartości dokumentu przez przeglądarkę internetową, a następnie używa standardowego API DOM i funkcji JavaScript w celu dodania i usunięcia elementów, konfiguracji procedur obsługi zdarzeń itd. W trakcie tworzenia aplikacji AngularJS nie występuje wyraźny etap kompilacji. Wystarczy zmodyfikować pliki HTML i JavaScript, a następnie wczytać je w przeglądarce internetowej.

Najważniejszym dodatkiem AngularJS w kodzie znaczników HTML jest atrybut `ng-app`. Wymieniony atrybut wskazuje, że element `<html>` na listingu 2.2 zawiera moduł, który powinien być skompilowany i przetworzony przez AngularJS. Kiedy AngularJS to jedyny używany framework JavaScript w aplikacji, przyjęło się dodawanie atrybutu `ng-app` do elementu `<html>`, jak to zrobiłem na listingu 2.2. W przypadku łączenia AngularJS z innymi technologiami, na przykład jQuery, można zawęzić zasięg działania aplikacji AngularJS przez zastosowanie atrybutu `ng-app` dla wybranego elementu w dokumencie.

Zastosowanie AngularJS w kodzie znaczników HTML

Dodawanie niestandardowych atrybutów i elementów do dokumentu HTML może wydawać się dziwne, zwłaszcza dla osób od dłuższego czasu zajmujących się tworzeniem aplikacji sieciowych i przyzwyczajonych do trzymania się standardu HTML. Jeżeli nie jesteś przekonany do idei atrybutów takich jak `ng-app`, to możesz skorzystać z podejścia alternatywnego. Polega ono na użyciu atrybutów `data` i poprzedzaniu dyrektyw AngularJS przedrostkiem `data-`. Dokładne omówienie dyrektyw znajdziesz w części II. W tym miejscu wystarczy wiedzieć, że `ng-app` jest dyrektywą, którą można zastosować w następujący sposób:

```
...
<html data-ng-app="todoApp">
...
```

W książce stosuję konwencję AngularJS polegającą na użyciu atrybutu `ng-app` oraz wszystkich pozostałych usprawnień dostępnych dla HTML. Zalecam Ci takie samo podejście. Oczywiście możesz zastosować inne, jeśli chcesz lub gdy używane narzędzia programistyczne nie pozwalają na przetwarzanie niestandardowych elementów i atrybutów HTML.

Utworzenie modelu danych

AngularJS obsługuje wzorzec MVC (ang. *Model-View-Controller*), którego omówienie znajdziesz w rozdziale 3. Ujmując rzecz najprościej, MVC wymaga podzielenia aplikacji na trzy odmienne obszary: dane aplikacji (model), logikę działającą na wspomnianych danych (kontrolery) oraz logikę odpowiedzialną za wyświetlanie danych (widoki).

Dane w omawianej tutaj prostej aplikacji listy rzeczy do zrobienia są umieszczone w różnych elementach HTML. Nazwa użytkownika została podana w nagłówku:

```
...
<h1>Lista rzeczy do zrobienia użytkownika Adam</h1>
...
```

Natomiast lista poszczególnych rzeczy do zrobienia znajduje się w elementach `<td>` tabeli:

```
...
<tr><td>Kupić kwiaty</td><td>No</td></tr>
...
```

Nasze pierwsze zadanie polega więc na wyodrębnieniu wszystkich danych i oddzieleniu ich od elementów HTML, aby w ten sposób utworzyć model. Separacja danych od sposobu ich wyświetlania użytkownikowi to kluczowa koncepcja we wzorcu MVC, co zostanie wyjaśnione w rozdziale 3. Ponieważ aplikacje AngularJS istnieją w przeglądarce internetowej, w omawianym przykładzie model trzeba zdefiniować w języku JavaScript w elemencie `<script>`, jak przedstawiono na listingu 2.3.

Listing 2.3. Utworzenie modelu danych w pliku *todo.html*

```
<!DOCTYPE html>
<html ng-app="todoApp">
<head>
  <title>Lista rzeczy do zrobienia</title>
  <link href="bootstrap.css" rel="stylesheet" />
  <link href="bootstrap-theme.css" rel="stylesheet" />
  <script src="angular.js"></script>
  <script>
    var model = {
```

```

        user: "Adam",
        items: [{ action: "Kupić kwiaty", done: false },
                 { action: "Kupić buty", done: false },
                 { action: "Odebrać bilety", done: true },
                 { action: "Zadzwonić do Janka", done: false }
        ]
    };

    var todoApp = angular.module("todoApp", []);
</script>
</head>
<body>
    <div class="page-header">
        <h1>Lista rzeczy do zrobienia</h1>
    </div>
    <div class="panel">
        <div class="input-group">
            <input class="form-control" />
            <span class="input-group-btn">
                <button class="btn btn-default">Dodaj</button>
            </span>
        </div>
        <table class="table table-striped">
            <thead>
                <tr>
                    <th>Opis</th>
                    <th>Zrobione</th>
                </tr>
            </thead>
            <tbody>
            </tbody>
        </table>
    </div>
</body>
</html>

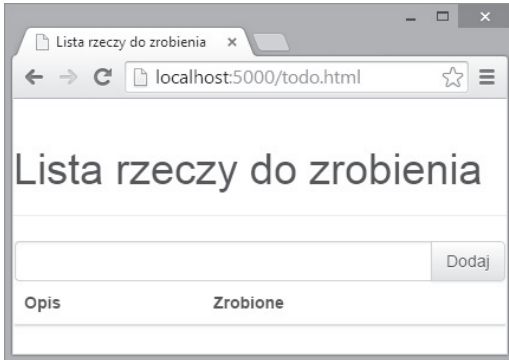
```

-
- **Wskazówka** Tutaj zastosowałem pewne uproszczenie. Model może zawierać także logikę niezbędną do utworzenia, wczytania, przechowywania i modyfikowania obiektów danych. W aplikacji AngularJS logika często znajduje się po stronie serwera i jest wykonywana przez serwer WWW. Więcej informacji na ten temat znajdziesz w rozdziale 3.
-

W kodzie został zdefiniowany obiekt JavaScript o nazwie `model` wraz z właściwościami odpowiadającymi danym, które wcześniej znajdowały się w różnych elementach HTML. Właściwość `user` definiuje nazwę użytkownika, natomiast właściwość `items` określa tablicę obiektów tworzących listę rzeczy do zrobienia.

Najczęściej nie będziesz definiować modelu bez jednoczesnego określenia pozostałych komponentów wzorca MVC. W tym miejscu chcę jednak zaprezentować sposób tworzenia prostej aplikacji AngularJS. Efekt wprowadzonych dotąd zmian pokazano na rysunku 2.2.

-
- **Wskazówka** W praktycznie każdym projekcie aplikacji AngularJS zachodzi konieczność zdefiniowania podstawowych komponentów wzorca MVC, a następnie połączenia ich ze sobą. Wówczas można odnieść wrażenie, że wykonywany jest krok wstecz, zwłaszcza gdy punktem wyjścia jest statyczna atrapa aplikacji, czyli podejście zastosowane w tym rozdziale. Czas poświęcony na przygotowanie wspomnianych komponentów *na pewno* się zwróci. Przykład większego projektu realizowanego w taki sposób poznasz w rozdziale 6., w którym zaczniemy budować znacznie bardziej skomplikowaną i realistyczną aplikację AngularJS. Na początku wymagane będzie włożenie znacznej ilości pracy w przygotowanie wstępnej konfiguracji aplikacji, ale później dodawanie kolejnych funkcji okaże się niezwykle łatwe i szybkie.
-



Rysunek 2.2. Nasza aplikacja po utworzeniu modelu danych

Utworzenie kontrolera

Kontroler definiuje logikę biznesową niezbędną do obsługi widoku, choć użyte tutaj wyrażenie *logika biznesowa* nie jest zbyt trafne. Najlepszym sposobem opisanego kontrolera jest wyjaśnienie, jakiego rodzaju logiki nie zawiera — pozostałe rodzaje znajdują więc zastosowanie w kontrolerze.

Logika odpowiedzialna za przechowywanie lub pobieranie danych stanowi część *modelu*. Z kolei logika pomagająca w formatowaniu danych i wyświetlaniu ich użytkownikowi jest częścią *widoku*. Kontroler jest umieszczony między modelem i widokiem i łączy je ze sobą. Do zadań kontrolera należy reakcja na działania użytkownika, uaktualnianie danych w modelu oraz dostarczanie widokowi wymaganych danych.

W tym momencie naprawdę nie ma znaczenia, czy to jest jasne dla Ciebie. Zanim zakończysz lekturę książki, będziesz swobodnie posługiwał się wzorcem MVC i potrafisz stosować go w aplikacjach AngularJS. Informacje szczegółowe dotyczące wzorca MVC znajdziesz w rozdziale 3., natomiast wyraźną separację między poszczególnymi komponentami MVC zaczniesz dostrzegać w rozdziale 6., w którym będziemy budować znacznie bardziej realistyczną aplikację sieciową w technologii AngularJS.

-
- **Wskazówka** Nie przejmuj się, jeśli nie masz głowy do wzorców. Wielokrotnie wspomniany tutaj wzorec MVC w dużej mierze opiera się na zdrowym rozsądku i jak się przekonasz, dość luźno stosuję go w tej książce. Wzorce to po prostu narzędzia, które mają pomagać programistom. Masz więc wolną rękę i możesz dostosować je do własnych potrzeb. Po przyswojeniu terminologii związanej z MVC możesz wybrać elementy najprzydatniejsze do Twoich potrzeb, a tym samym zaadaptować MVC i AngularJS do opracowywanych projektów i preferowanego stylu programowania.
-

Utworzenie kontrolera następuje przez wywołanie metody `controller()` obiektu `Module` zwróconego przez metodę `angular.module()`, jak pokazano w poprzednim punkcie. Argumentami metody `controller()` są nazwa nowego kontrolera oraz funkcja, która będzie wywołana w celu zdefiniowania funkcjonalności kontrolera (patrz listing 2.4).

Listing 2.4. Przykład utworzenia kontrolera w pliku `todo.html`

```
<!DOCTYPE html>
<html ng-app="todoApp">
<head>
  <title>Lista rzeczy do zrobienia</title>
  <link href="bootstrap.css" rel="stylesheet" />
  <link href="bootstrap-theme.css" rel="stylesheet" />
  <script src="angular.js"></script>
  <script>
    var model = {
      user: "Adam",
```

```

        items: [{ action: "Kupić kwiaty", done: false },
                  { action: "Kupić buty", done: false },
                  { action: "Odebrać bilety", done: true },
                  { action: "Zadzwoń do Janka", done: false }]
    };

    var todoApp = angular.module("todoApp", []);

    todoApp.controller("ToDoCtrl", function ($scope) {
        $scope.todo = model;
    });
</script>
</head>
<body ng-controller="ToDoCtrl">
    <div class="page-header">
        <h1>Lista rzeczy do zrobienia</h1>
    </div>
    <div class="panel">
        <div class="input-group">
            <input class="form-control" />
            <span class="input-group-btn">
                <button class="btn btn-default">Dodaj</button>
            </span>
        </div>
        <table class="table table-striped">
            <thead>
                <tr>
                    <th>Opis</th>
                    <th>Zrobione</th>
                </tr>
            </thead>
            <tbody></tbody>
        </table>
    </div>
</body>
</html>

```

Wedle konwencji nazwa kontrolera powinna mieć postać `<Nazwa>Ctrl`, gdzie człon `<Nazwa>` pomoże w określeniu roli kontrolera w aplikacji. W rzeczywistych aplikacjach najczęściej znajduje się wiele kontrolerów. W prezentowanej tutaj wystarczy tylko jeden, któremu nadałem nazwę `ToDoCtrl`.

■ **Wskazówka** Nazwy kontrolerów to tylko konwencja i możesz stosować dowolne. Idea powszechnie stosowanych konwencji polega na tym, że dzięki nim programiści znający AngularJS będą mogli bardzo szybko ustalić strukturę projektu.

Przyznaję, że przedstawiony kontroler może być rozczarowujący, ponieważ jest możliwie najprostszy. Jednym z podstawowych zadań kontrolera jest dostarczenie widokom niezbędnych danych. Nie zawsze zachodzi potrzeba, aby widoki miały dostęp do pełnego modelu. Dlatego też kontroler można wykorzystać do wyraźnego określenia zbioru danych dostępnych dla widoku. Wspomniany zbiór nosi nazwę *zakresu* (ang. *scope*).

Argumentem dla funkcji kontrolera utworzonego w omawianym przykładzie jest `$scope`, czyli znak `$` i słowo `scope`. W aplikacji AngularJS rozpoczynające się od znaku `$` nazwy zmiennych oznaczają wbudowane funkcjonalności AngularJS. Kiedy widzisz znak `$`, to zwykle odwołuje się on do wbudowanej *usługi*, która zazwyczaj jest samodzielnym komponentem dostarczającym funkcjonalności wielu komponentom. Jednak `$scope` ma znaczenie specjalne — służy do udostępnienia danych i funkcji widokom. Więcej informacji o zakresie znajdziesz w rozdziale 13., natomiast wbudowane usługi będą omówione w rozdziałach od 18. do 25.

Chcemy, aby w tworzonej tutaj aplikacji widoki miały dostęp do całego modelu. Dlatego też zdefiniowałem właściwość `todo` w obiekcie usługi `$scope` i przypisałem jej cały model w następujący sposób:

```
...
$scope.todo = model;
...
```

Takie podejście stanowi rodzaj wstępu do możliwości wykorzystania w widokach danych pochodzących z modelu, co zostanie wkrótce zademonstrowane. Za pomocą atrybutu `ng-controller` wskazałem także w dokumencie HTML obszar, za który będzie odpowiedzialny dany kontroler. Ponieważ tworzymy prostą aplikację zawierającą tylko jeden kontroler, atrybut `ng-controller` został umieszczony w elemencie `<body>`:

```
...
<body ng-controller="ToDoCtrl">
...
```

Wartością atrybutu `ng-controller` jest nazwa kontrolera, czyli w omawianym przykładzie `ToDoCtrl`. Do tematu kontrolerów jeszcze powróćmy w rozdziale 13.

Utworzenie widoku

Wygenerowanie widoku następuje przez połączenie danych dostarczanych przez kontroler z elementami HTML tworzącymi zawartość wyświetlaną przez przeglądarkę internetową. Na listingu 2.5 przedstawiono użycie jednego z rodzajów adnotacji nazywanego *dołączaniem danych* (ang. *data binding*), co powoduje umieszczenie w dokumencie HTML danych pochodzących z modelu.

Listing 2.5. Wyświetlenie w pliku `todo.html` danych modelu za pomocą widoku

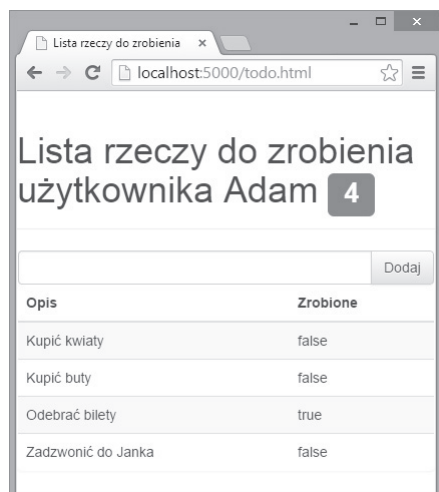
```
...
<body ng-controller="ToDoCtrl">
  <div class="page-header">
    <h1>
      Lista rzeczy do zrobienia użytkownika {{todo.user}}
    <span class="label label-default">{{todo.items.length}}</span>
    </h1>
  </div>
  <div class="panel">
    <div class="input-group">
      <input class="form-control" />
      <span class="input-group-btn">
        <button class="btn btn-default">Dodaj</button>
      </span>
    </div>
    <table class="table table-striped">
      <thead>
        <tr>
          <th>0pis</th>
          <th>Zrobione</th>
        </tr>
      </thead>
      <tbody>
        <tr ng-repeat="item in todo.items">
          <td>{{item.action}}</td>
          <td>{{item.done}}</td>
        </tr>
      </tbody>
    </table>
  </div>
</body>
```

```

    </table>
  </div>
</body>
...

```

Efekt połączenia modelu, kontrolera i widoku możesz zobaczyć w przeglądarce internetowej po wyświetleniu pliku *todo.html* (patrz rysunek 2.3). Wygenerowany kod HTML zostanie omówiony w kolejnych punktach.



Rysunek 2.3. Efekt zdefiniowania widoku w pliku *todo.html*

Wstawianie wartości modelu

AngularJS używa notacji podwójnych nawiasów klamrowych (`{{ i }}`) w celu wskazania wyrażenia dołączania danych. Zawartość wyrażenia jest obliczana przez JavaScript, dane i funkcje są ograniczone do zakresu definiowanego przez kontroler. W omawianym przykładzie można uzyskać dostęp do jedynie tych części modelu, które zostały przypisane obiektowi `$scope` podczas definiowania kontrolera. Używane są nazwy właściwości utworzone w obiekcie `$scope`.

Oznacza to, że jeśli chcesz uzyskać dostęp do właściwości `model.user`, to należy zdefiniować wyrażenie dołączania odwołujące się do `todo.user`. Wynika to z faktu przypisania obiektu modelu do właściwości `$scope.todo`.

AngularJS kompiluje kod znaczników HTML w dokumencie, wyszukuje atrybut `ng-controller`, a następnie wywołuje funkcję kontrolera `ToDoCtrl` w celu zdefiniowania zakresu używanego do utworzenia widoku. Po napotkaniu wyrażenia dołączania danych AngularJS wyszukuje w obiekcie `$scope` wskazaną wartość i umieszcza ją w dokumencie HTML. Na przykład wyrażenie

```

...
Lista rzeczy do zrobienia użytkownika {{todo.user}}
...

```

jest przetwarzane i przekształcane na postać poniższego ciągu tekstowego:

```
Lista rzeczy do zrobienia użytkownika Adam
```

Powyższa technika nosi nazwę *dołączania danych* lub *dołączania modelu* — wartość pochodząca z modelu jest dołączana do zawartości elementu HTML. Istnieje kilka różnych sposobów dołączania danych; zostaną one przedstawione w rozdziale 10.

Obliczanie wartości wyrażenia

Zawartością wyrażenia dołączania danych może być dowolne, prawidłowe polecenie języka JavaScript. Oznacza to możliwość przeprowadzenia operacji w celu utworzenia nowych danych na podstawie modelu. Na listingu 2.5 wykorzystano tę możliwość do wyświetlenia liczby elementów znajdujących się na liście rzeczy do zrobienia:

```
...
<div class="page-header">
  Lista rzeczy do zrobienia użytkownika {{todo.user}}<span class="label label-
default">{{todo.items.length}}</span>
</div>
...
```

AngularJS oblicza wartość wyrażenia i wyświetla liczbę elementów znajdujących się w tablicy. Dzięki temu użytkownik wie, ile elementów znajduje się na liście rzeczy do zrobienia. Wspomniana liczba jest wyświetlana w nagłówku dokumentu HTML (i sformatowana za pomocą klasy `label` zdefiniowanej przez Bootstrap CSS).

■ **Wskazówka** Wyrażenia należy wykorzystywać jedynie do przeprowadzania prostych operacji niezbędnych do przygotowania danych do wyświetlenia. Nie używaj poleceń dołączania danych w celu wykonywania skomplikowanej logiki lub operacji na modelu, ponieważ są to zadania przeznaczone dla kontrolera. Bardzo często można spotkać się z logiką, którą trudno zaklasyfikować jako odpowiednią dla widoku lub kontrolera, a podjęcie decyzji, co zrobić z tego rodzaju kodem, również może być trudne. Moja rada brzmi: nie przejmuj się. Dokonaj wyboru i nie spowalniaj prac, logikę zawsze można później przenieść w inne miejsce, jeśli zajdzie potrzeba. Jeżeli naprawdę nie wiesz, co zrobić, najlepiej umieścić logikę w kontrolerze. W 60% przypadków takie rozwiązanie okazuje się właściwą decyzją.

Użycie dyrektyw

Wyrażenia są używane także z dyrektywami, które wskazują AngularJS sposób przetwarzania wyświetlanej treści. Na listingu 2.5 użyłem atrybutu `ng-repeat` stosującego dyrektywę nakazującą AngularJS wygenerowanie dla każdego obiektu w zbiorze elementu zawierającego wspomnianą dyrektywę i odpowiedniej zawartości:

```
...
<tr ng-repeat="item in todo.items">
  <td>{{item.action}}</td><td>{{item.done}}</td>
</tr>
...
```

Wartość atrybutu `ng-repeat` jest podawana w formacie `<nazwa> in <zbiór>`. W omawianym listingu wyrażenie ma postać `item in todo.items` i oznacza wygenerowanie elementów `<tr>` i `<td>` dla każdego obiektu w tablicy `todo.items` oraz przypisanie każdego obiektu tablicy do zmiennej o nazwie `item`.

Za pomocą zmiennej `item` można zdefiniować wyrażenie dołączania danych dla właściwości każdego obiektu w tablicy i tym samym otrzymać przedstawiony poniżej kod HTML:

```
...
<tr ng-repeat="item in todo.items" class="ng-scope">
  <td class="ng-binding">Kupić kwiaty</td>
  <td class="ng-binding">false</td>
</tr>
<tr ng-repeat="item in todo.items" class="ng-scope">
  <td class="ng-binding">Kupić buty</td>
  <td class="ng-binding">false</td>
</tr>
<tr ng-repeat="item in todo.items" class="ng-scope">
  <td class="ng-binding">Odebrać bilety</td>
  <td class="ng-binding">true</td>
</tr>
```

```
<tr ng-repeat="item in todo.items" class="ng-scope">
  <td class="ng-binding">Zadzwoń do Janka</td>
  <td class="ng-binding">false</td>
</tr>
...
```

Jak zobaczysz w dalszych rozdziałach książki, dyrektywy stanowią podstawę sposobu działania AngularJS, a `ng-repeat` to jedna z najczęściej używanych dyrektyw.

Wyjście poza podstawy

Na tym etapie zdefiniowaliśmy podstawowe komponenty wzorca MVC. W ten sposób powstała dynamiczna wersja aplikacji, której statyczną atrapę przygotowaliśmy na początku rozdziału. Mając już opanowane solidne podstawy, możemy przystąpić do użycia pewnych znacznie bardziej zaawansowanych technik w celu dodania nowych funkcji i zbudowania nieco bardziej rozbudowanej aplikacji. W kolejnych punktach w tworzonej aplikacji listy rzeczy do zrobienia wykorzystamy różne funkcje AngularJS. Wskażę też rozdziały książki, w których dowiesz się nieco więcej o wspomnianych funkcjach.

Użycie dwukierunkowego dołączania modelu

Zastosowane w omawianym przykładzie dołączanie nosi nazwę *dołączania jednokierunkowego* — wartości są pobierane z modelu, a następnie wstawiane w elementach szablonu. Ta technika jest całkiem standardowa i dość powszechnie stosowana w programowaniu sieciowym. Na przykład podczas pracy z biblioteką jQuery bardzo często używam pakietu szablonu Handlebars, który zapewnia wspomniany rodzaj dołączania danych i jest użyteczny do generowania kodu znaczników HTML na podstawie obiektów danych.

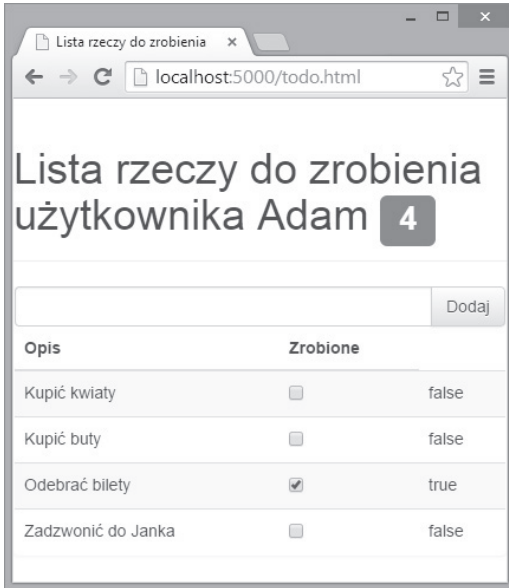
AngularJS idzie o krok dalej i zapewnia *dołączanie dwukierunkowe* — model jest używany w celu wygenerowania elementów, a zmiany elementu powodują wprowadzenie odpowiednich modyfikacji w modelu. Aby zademonstrować implementację dołączania dwukierunkowego, zmodyfikujemy plik *todo.html* (patrz listing 2.6). Po wprowadzonej zmianie stan każdego zadania do wykonania będzie wskazywany przez pole wyboru.

Listing 2.6. Dodanie pól wyboru do pliku *todo.html*

```
...
<tr ng-repeat="item in todo.items">
  <td>{{item.action}}</td>
  <td><input type="checkbox" ng-model="item.done" /></td>
  <td>{{item.done}}</td>
</tr>
...
```

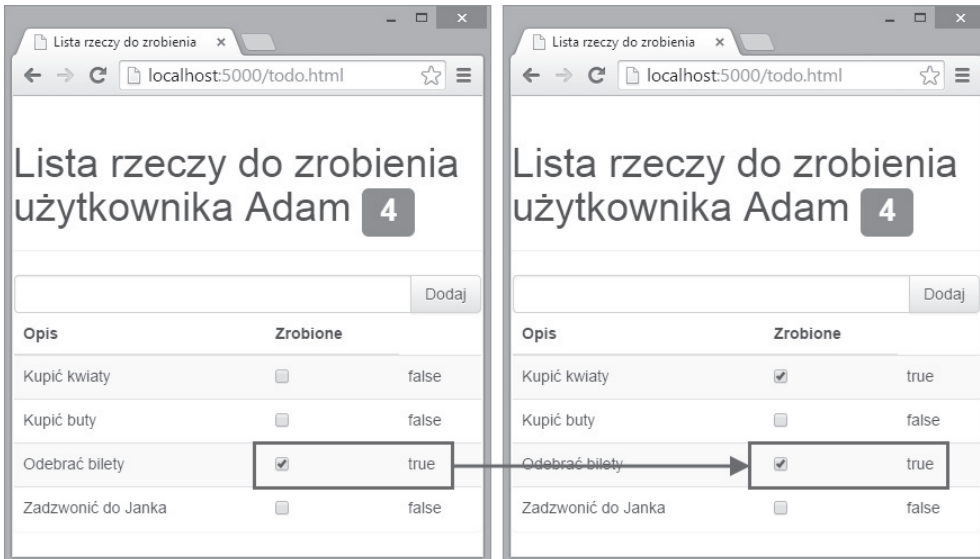
W tabeli umieściliśmy nowy element `<td>` przeznaczony do przechowywania elementu `<input>` w postaci pola wyboru. Najważniejszym dodatkiem jest tutaj atrybut `ng-model` nakazujący AngularJS utworzenie dwukierunkowego wiązania między wartością elementu `<input>` i właściwością `done` odpowiadającego mu obiektu danych (to będzie obiekt przypisany przez dyrektywę `ng-repeat` zmiennej `item` podczas generowania elementów).

Po pierwszej kompilacji kodu HTML biblioteka AngularJS wykorzysta wartość właściwości `done` w celu ustawienia wartości elementu `<input>`. Ponieważ używane jest pole wyboru, więc wartość `true` powoduje zaznaczenie pola, natomiast `false` usunięcie zaznaczenia pola. Efekt wprowadzonej zmiany możesz zobaczyć w przeglądarce internetowej po wyświetleniu w niej dokumentu *todo.html* (patrz rysunek 2.4). Jak możesz dostrzec, stan pól wyboru odpowiada wartościom `true` i `false`, które pozostawiono w tabeli, aby tym samym ułatwić demonstrację sposobu działania funkcji dołączania danych.



Rysunek 2.4. Dodanie pól wyboru do tworzonej aplikacji

Działanie dwukierunkowego dołączania danych stanie się oczywiste, jeśli zaznaczysz pierwszy element listy i później usuniesz jego zaznaczenie — powinieneś zauważyć, że zaznaczenie lub usunięcie zaznaczenia pola wyboru powoduje również zmianę wartości tekstowej w kolumnie znajdującej się po prawej stronie pola. AngularJS stosuje dynamiczne dołączanie wartości. Dwukierunkowe dołączanie danych (takie jak zastosowane tutaj dla elementu `<input>`) powoduje uaktualnienie modelu, co z kolei prowadzi do uaktualnienia innych elementów opartych na dołączaniu danych. W omawianym przykładzie wartości elementu `<input>` i prawej kolumny są ze sobą zsynchronizowane, jak pokazano na rysunku 2.5.



Rysunek 2.5. Użycie dwukierunkowego dołączania danych

Dwukierunkowe dołączanie danych może być stosowane dla elementów pobierających dane wejściowe od użytkownika, co w praktyce oznacza elementy formularzy sieciowych HTML. Temat formularzy sieciowych będzie dokładnie omówiony w rozdziale 12. Dzięki dynamicznemu i „żywemu” modelowi tworzenie skomplikowanych aplikacji z użyciem AngularJS stało się bardzo łatwe. W książce zobaczysz wiele przykładów dynamicznego zachowania AngularJS.

- **Wskazówka** Wartości `true` i `false` w prawej kolumnie są wyświetlane, aby ułatwić Ci dostrzeżenie efektu zastosowania dwukierunkowego dołączania danych. W rzeczywistych projektach zwykle nie należy wyświetlać tych wartości. Na szczęście rozszerzenie Batarang dla przeglądarki internetowej Google Chrome znacznie ułatwia przeglądanie i monitorowanie modelu (a także innych funkcji AngularJS). Więcej informacji o rozszerzeniu Batarang znajdziesz w rozdziale 1.

Zdefiniowanie i stosowanie funkcji w kontrolerach

Kontrolery definiują *funkcje* w zdefiniowanym zakresie. Wspomniane funkcje operują na danych w modelu i są odpowiedzialne za implementację logiki biznesowej w aplikacji. Funkcje zdefiniowane przez kontroler pozwalają na wyświetlenie danych użytkownikowi oraz na uaktualnienie modelu na podstawie działań podejmowanych przez użytkownika.

W celu zademonstrowania prostej funkcji zmienimy etykietę wyświetlaną po prawej stronie nagłówka naszej aplikacji. Po wprowadzeniu zmiany etykieta będzie wyświetlała jedynie liczbę niewykonanych jeszcze zadań. Zmiany konieczne do wprowadzenia przedstawiono na listingu 2.7. Przy okazji usuwamy prawą kolumnę wyświetlającą wartości `true` i `false`, ponieważ była nam potrzebna jedynie do pokazania efektu zmian w modelu danych zachodzących na skutek działania dwukierunkowego dołączania danych.

Listing 2.7. Zdefiniowanie i zastosowanie funkcji kontrolera w pliku `todo.html`

```
<!DOCTYPE html>
<html ng-app="todoApp">
<head>
    <title>Lista rzeczy do zrobienia</title>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script src="angular.js"></script>
    <script>
        var model = {
            user: "Adam",
            items: [{ action: "Kupić kwiaty", done: false },
                    { action: "Kupić buty", done: false },
                    { action: "Odebrać bilety", done: true },
                    { action: "Zadzwonić do Janka", done: false } ]
        };

        var todoApp = angular.module("todoApp", []);

        todoApp.controller("ToDoCtrl", function ($scope) {
            $scope.todo = model;

            $scope.incompleteCount = function () {
                var count = 0;
                angular.forEach($scope.todo.items, function (item) {
                    if (!item.done) { count++ }
                });
                return count;
            }
        })
    </script>

```

```

    });
  </script>
</head>
<body ng-controller="ToDoCtrl">
  <div class="page-header">
    <h1>
      Lista rzeczy do zrobienia użytkownika {{todo.user}}
      <span class="label label-default" ng-hide="incompleteCount() == 0">
        {{incompleteCount()}}
      </span>
    </h1>
  </div>
  <div class="panel">
    <div class="input-group">
      <input class="form-control" />
      <span class="input-group-btn">
        <button class="btn btn-default">Dodaj</button>
      </span>
    </div>
    <table class="table table-striped">
      <thead>
        <tr>
          <th>Opis</th>
          <th>Zrobione</th>
        </tr>
      </thead>
      <tbody>
        <tr ng-repeat="item in todo.items">
          <td>{{item.action}}</td>
          <td><input type="checkbox" ng-model="item.done" /></td>
        </tr>
      </tbody>
    </table>
  </div>
</body>
</html>

```

Funkcja jest definiowana przez dodanie jej do obiektu `$scope` przekazywanego funkcji kontrolera. Na listingu 2.7 zdefiniowano funkcję zwracającą liczbę jeszcze niewykonanych zadań. Wspomniana liczba jest określana przez zliczenie obiektów tablicy `$scope.todo.items`, których wartością właściwości `done` jest `false`.

■ **Wskazówka** Do zliczenia liczby elementów tablicy wykorzystaliśmy metodę `angular.forEach()`. Biblioteka AngularJS zawiera wiele użytecznych metod pomocniczych uzupełniających możliwości języka JavaScript. Wspomniane metody pomocnicze zostaną omówione w rozdziale 5.

Nazwa właściwości użytej w celu dołączenia funkcji do obiektu `$scope` jest nazwą danej funkcji. W omawianym przykładzie funkcja nosi nazwę `incompleteCount()` i może być wywołana w zakresie atrybutu `ng-controller`, który powoduje dołączenie kontrolera do elementów HTML tworzących widok.

Na listingu 2.7 funkcja `incompleteCount()` została użyta dwukrotnie. Po raz pierwszy w celu prostego dołączenia wartości wskazującej na liczbę elementów:

```

...
<span class="label label-default" ng-hide="incompleteCount() == 0">
  {{incompleteCount()}}
</span>
...

```

Zwróć uwagę na wywołanie funkcji z wykorzystaniem nawiasów klamrowych. Ponieważ argumentami funkcji mogą być obiekty, to możliwe jest zdefiniowanie ogólnego przeznaczenia funkcji gotowej do użycia z różnymi obiektami danych. Budowana przez nas aplikacja jest na tyle prosta, że nie przekazujemy żadnych argumentów funkcji. Zamiast tego pobieramy wymagane dane bezpośrednio z obiektu `$scope` w kontrolerze.

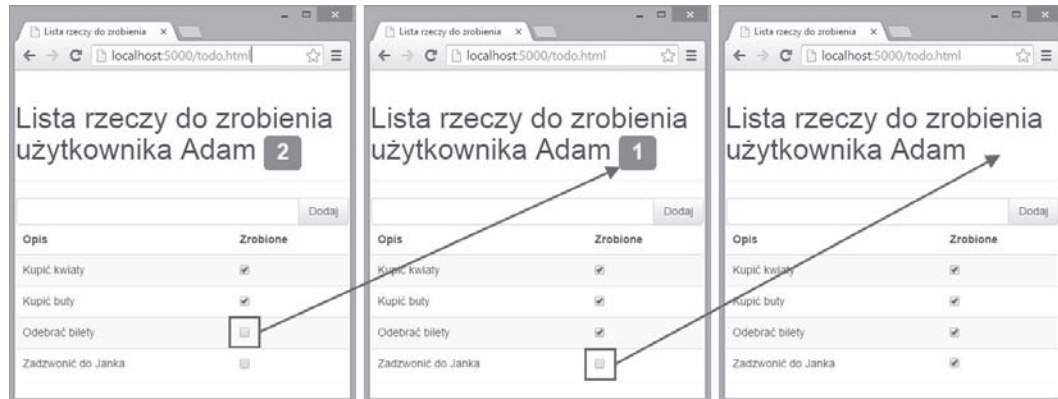
Po raz drugi funkcja została użyta w połączeniu z dyrektywą:

```
...
<span class="label default" ng-hide="incompleteCount() == 0">
  {{incompleteCount()}}
</span>
...
```

Dyrektywa `ng-hide` powoduje ukrycie elementu i jego zawartości, jeżeli wartością wyrażenia przypisanego jako atrybut dyrektywy będzie `true`. W omawianym przykładzie następuje wywołanie funkcji `incompleteCount()` i sprawdzenie, czy liczba zadań do wykonania wynosi 0. Jeżeli nie ma żadnych rzeczy do wykonania, to etykieta wyświetlająca liczbę zadań pozostałych do wykonania zostanie ukryta.

■ **Wskazówka** Dyrektywa `ng-hide` to tylko jedna z wielu przeznaczonych do przeprowadzania operacji na obiekowym modelu dokumentu (ang. *Document Object Model*) automatycznie na podstawie stanu modelu AngularJS. Szczegółowe omówienie dyrektyw znajdziesz w rozdziale 11., natomiast temat tworzenia własnych dyrektyw będzie poruszony w rozdziałach od 15. do 17.

Efekt działania zdefiniowanej funkcji możesz zobaczyć w przeglądarce internetowej po wyświetleniu w niej dokumentu *todo.html* (patrz rysunek 2.6). Jak możesz dostrzec, zaznaczenie lub usunięcie zaznaczenia pola wyboru powoduje zmianę liczby wyświetlanej przez etykietę w nagłówku. Zaznaczenie wszystkich elementów powoduje ukrycie etykiety.



Rysunek 2.6. Efekt użycia funkcji zdefiniowanej w kontrolerze

Użycie funkcji w zależności od innych zdefiniowanych funkcji

Jednym z tematów nieustannie przewijających się przez społeczność AngularJS jest możliwość wykorzystania naturalnych cech charakterystycznych HTML, CSS i JavaScript do tworzenia aplikacji sieciowych. Na przykład funkcje kontrolera są definiowane za pomocą funkcji JavaScript. Istnieje więc możliwość przygotowania funkcji opartych na możliwościach oferowanych przez inne funkcje zdefiniowane w tym samym kontrolerze. Na listingu 2.8 przedstawiono przykład funkcji pobierającej klasę CSS na podstawie liczby niewykonanych zadań na liście rzeczy do zrobienia.

Listing 2.8. *Utworzenie funkcji, której działanie jest oparte na innej funkcji zdefiniowanej w kontrolerze*

```

<!DOCTYPE html>
<html ng-app="todoApp">
<head>
  <title>Lista rzeczy do zrobienia</title>
  <link href="bootstrap.css" rel="stylesheet" />
  <link href="bootstrap-theme.css" rel="stylesheet" />
  <script src="angular.js"></script>
  <script>
    var model = {
      user: "Adam",
      items: [{ action: "Kupić kwiaty", done: false },
               { action: "Kupić buty", done: false },
               { action: "Odebrać bilety", done: true },
               { action: "Zadzwoń do Janka", done: false }]
    };

    var todoApp = angular.module("todoApp", []);

    todoApp.controller("ToDoCtrl", function ($scope) {
      $scope.todo = model;

      $scope.incompleteCount = function () {
        var count = 0;
        angular.forEach($scope.todo.items, function (item) {
          if (!item.done) { count++ }
        });
        return count;
      }

      $scope.warningLevel = function () {
        return $scope.incompleteCount() < 3 ? "label-success" : "label-warning";
      }
    });
  </script>
</head>
<body ng-controller="ToDoCtrl">
  <div class="page-header">
    <h1>
      Lista rzeczy do zrobienia użytkownika {{todo.user}}
      <span class="label label-default" ng-class="warningLevel()"
            ng-hide="incompleteCount() == 0">
        {{incompleteCount()}}
      </span>
    </h1>
  </div>
  <!-- ... pominięto w celu zachowania zwięzłości ... -->
</body>
</html>

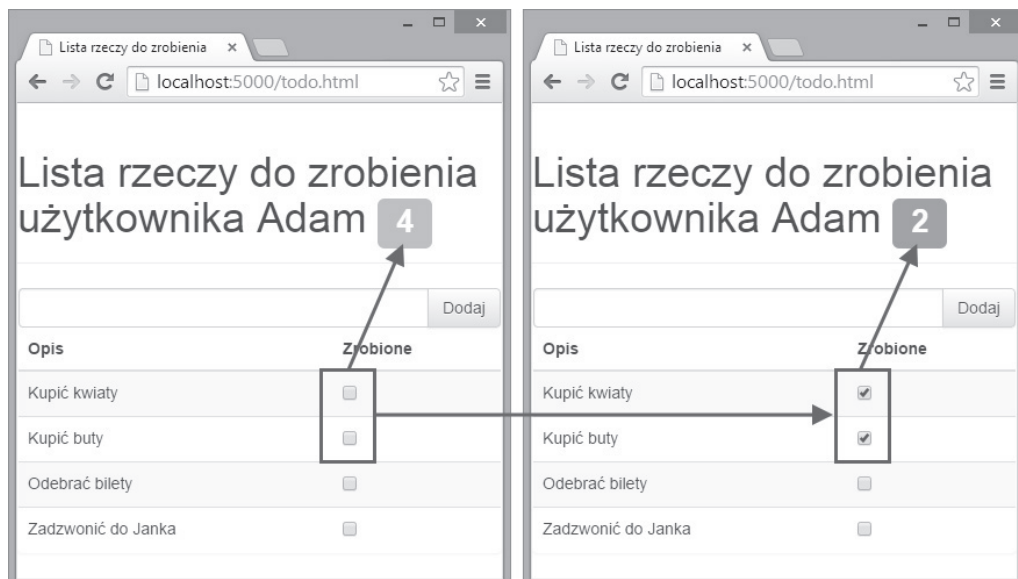
```

Na listingu zdefiniowaliśmy nową funkcję o nazwie `warningLevel`, która zwraca nazwę klasy Bootstrap CSS na podstawie liczby pozostałych do wykonania zadań na liście rzeczy do zrobienia określonej przez wywołanie funkcji `incompleteCount()`. Tego rodzaju podejście zmniejsza ilość powielonego kodu w kontrolerze i jak się przekonasz w rozdziale 25., może znacznie ułatwić proces przeprowadzania testów jednostkowych.

Możliwość użycia funkcji `warningLevel()` została wskazana dyrektywą `ng-class` w następujący sposób:

```
...
<span class="label" ng-class="warningLevel()" ng-hide="incompleteCount() == 0">
...
```

Dyrektywa powoduje użycie klasy CSS wskazanej przez funkcję. Efektem jest zmiana koloru etykiety w dokumencie HTML, jak pokazano na rysunku 2.7. Pełne omówienie dyrektyw AngularJS znajdziesz w części II książki, natomiast temat tworzenia własnych dyrektyw będzie poruszony w rozdziałach od 15. do 17.



Rysunek 2.7. Użycie dyrektywy w celu zastosowania klasy dla elementów

- **Wskazówka** Zwróć uwagę na istnienie w elemencie `<span>` dwóch dyrektyw, z których każda opiera się na innej funkcji. Możesz dowolnie łączyć funkcje i dyrektywy, aby w ten sposób uzyskać efekt wymagany w aplikacji. Być może trudno to dostrzec w drukowanej wersji książki, ale etykieta jest wyświetlana w kolorze zielonym, gdy do wykonania zostały trzy zadania lub mniej. W przeciwnym razie kolor etykiety jest pomarańczowy.

Reakcja na działania użytkownika

Zobaczyłeś, jak funkcje i dyrektywy mogą być ze sobą łączone w celu przygotowania funkcjonalności oferowanych przez aplikację. Wspomniane połączenie zapewnia bardzo dużą funkcjonalność w aplikacji AngularJS. Jeden z najlepszych efektów połączeń uzyskujemy, gdy dyrektywy i funkcje są używane do reakcji na działania podejmowane przez użytkownika. Na listingu 2.9 przedstawiono wprowadzone w pliku `todo.html` modyfikacje, które pozwalają użytkownikowi na tworzenie nowych zadań do wykonania.

Listing 2.9. Dodanie kodu pozwalającego aplikacji reagować na działania użytkownika

```
<!DOCTYPE html>
<html ng-app="todoApp">
<head>
  <title>Lista rzeczy do zrobienia</title>
  <link href="bootstrap.css" rel="stylesheet" />
  <link href="bootstrap-theme.css" rel="stylesheet" />
  <script src="angular.js"></script>
```

```

<script>
  var model = {
    user: "Adam",
    items: [{ action: "Kupić kwiaty", done: false },
             { action: "Kupić buty", done: false },
             { action: "Odebrać bilety", done: true },
             { action: "Zadzwoń do Janka", done: false }]
  };

  var todoApp = angular.module("todoApp", []);

  todoApp.controller("ToDoCtrl", function ($scope) {
    $scope.todo = model;
    $scope.incompleteCount = function () {
      var count = 0;
      angular.forEach($scope.todo.items, function (item) {
        if (!item.done) { count++; }
      });
      return count;
    }

    $scope.warningLevel = function () {
      return $scope.incompleteCount() < 3 ? "label-success" : "label-warning";
    }

    $scope.addNewItem = function (actionText) {
      $scope.todo.items.push({ action: actionText, done: false });
    }
  });
</script>
</head>
<body ng-controller="ToDoCtrl">
  <div class="page-header">
    <h1>
      Lista rzeczy do zrobienia użytkownika {{todo.user}}
      <span class="label label-default" ng-class="warningLevel()"
            ng-hide="incompleteCount() == 0">
        {{incompleteCount()}}
      </span>
    </h1>
  </div>
  <div class="panel">
    <div class="input-group">
      <input class="form-control" ng-model="actionText" />
      <span class="input-group-btn">
        <button class="btn btn-default"
              ng-click="addNewItem(actionText)">Dodaj</button>
      </span>
    </div>
    <table class="table table-striped">
      <thead>
        <tr>
          <th>Opis</th>
          <th>Zrobione</th>
        </tr>
      </thead>
      <tbody>
        <tr ng-repeat="item in todo.items">

```

```

        <td>{{item.action}}</td>
        <td><input type="checkbox" ng-model="item.done" /></td>
    </tr>
</tbody>
</table>
</div>
</body>
</html>

```

W aplikacji zdefiniowaliśmy funkcję o nazwie `addNewItem()`, która pobiera tekst nowego zadania do wykonania, a następnie dodaje obiekt do modelu danych. Wspomniany tekst jest używany jako wartość właściwości `action`, natomiast wartością właściwości `done` jest `false`:

```

...
$scope.addNewItem = function(actionText) {
    $scope.todo.items.push({ action: actionText, done: false});
}
...

```

To jest pierwsza przedstawiona funkcja modyfikująca model. W projektach rzeczywistych aplikacji zwykle istnieje mniej więcej równy podział między funkcjami pobierającymi i przygotowującymi dane dla widoku a reagującymi na działania użytkownika i odpowiednio uaktualniającymi model. Zwróć uwagę, że przedstawiona funkcja została zdefiniowana jako standardowa funkcja JavaScript, a uaktualnienie modelu jest możliwe z wykorzystaniem metody `push()` używanej przez JavaScript do obsługi tablic.

Zaleta omawianego przykładu kryje się w zastosowaniu dwóch dyrektyw. Oto pierwsza z nich:

```

...
<input class="form-control" ng-model="actionText" />
...

```

To jest dokładnie ta sama dyrektywa `ng-model`, którą wykorzystaliśmy wcześniej do konfiguracji pól wyboru. Wielokrotnie spotkasz tę dyrektywę podczas pracy z elementami formularza. Warto zwrócić uwagę na podanie nazwy właściwości uaktualnianej przez dyrektywę — nie jest ona częścią modelu. Dyrektywa `ng-model` dynamicznie utworzy właściwość w zakresie kontrolera. W ten sposób umożliwia dynamiczne tworzenie właściwości modelu przeznaczonych do obsługi danych wejściowych dostarczanych przez użytkownika. Właściwość dynamiczna została użyta w drugiej dyrektywie omawianego przykładu:

```

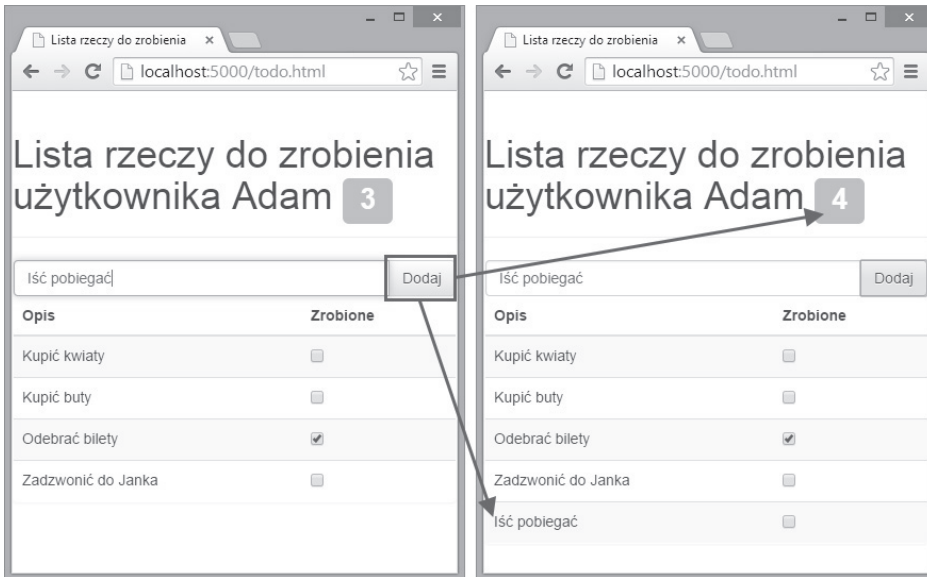
...
<button class="btn btn-default" ng-click="addNewItem(actionText)">Dodaj</button>
...

```

Dyrektywa `ng-click` konfiguruje procedurę obsługi odpowiedzialną za obliczenie wyrażenia po wywołaniu zdarzenia `click`. W omawianym przykładzie wyrażenie wywołuje funkcję `addNewItem()` i przekazuje argument w postaci właściwości dynamicznej `actionText`. Efektem jest dodanie do listy nowego zadania do wykonania wraz z tekstem podanym przez użytkownika w polu danych wejściowych (patrz rysunek 2.8).

■ **Wskazówka** Prawdopodobnie przyjąłeś zasadę, aby nie dodawać żadnego kodu obsługi zdarzeń do poszczególnych elementów. Dlatego też zastosowanie dyrektywy `ng-click` w elemencie `<button>` może wydawać się dziwne. Nie przejmuj się tym. Kiedy biblioteka AngularJS kompiluje plik HTML i napotyka wymienioną dyrektywę, to konfiguruje procedurę obsługi, stosując podejście dyskretnego kodu JavaScript — kod procedury obsługi zdarzeń będzie oddzielony od elementu. Bardzo ważne jest odróżnianie dyrektyw AngularJS od generowanego w trakcie kompilacji na ich podstawie kodu HTML i JavaScript.

Zwróć uwagę, że etykieta wyświetlająca liczbę zadań do wykonania jest automatycznie uaktualniana po dodaniu nowego elementu do listy. Jedną z zalet stosowanego przez AngularJS modelu aplikacji jest to, że współdziałanie poleceń dołączania danych i zdefiniowanych funkcji tworzy podstawę dla funkcjonalności aplikacji.



Rysunek 2.8. Użycie funkcji i dyrektyw w celu utworzenia nowych elementów listy rzeczy do zrobienia

Filtrowanie i zmiana kolejności danych modelu

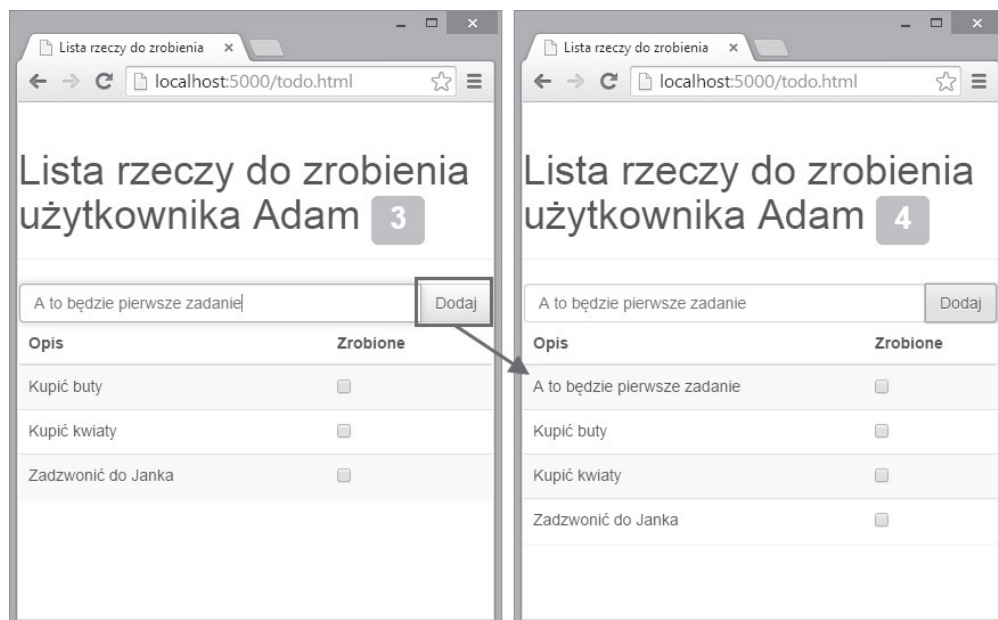
W rozdziale 14. zostanie omówiona funkcja *filtrowania* w AngularJS, która zapewnia elegancki sposób przygotowania danych w modelu do wyświetlania w widoku bez konieczności definiowania własnych funkcji. Wprawdzie nie ma nic złego w definiowaniu funkcji, ale filtry są z reguły rozwiązaniami ogólnego przeznaczenia i dlatego można wielokrotnie korzystać z nich w aplikacji. Na listingu 2.10 przedstawiono zmiany wprowadzone w pliku *todo.html* mające na celu zademonstrowanie operacji filtrowania.

Listing 2.10. Dodanie funkcji filtrowania do pliku *todo.html*

```
...
<tbody>
  <tr ng-repeat="item in todo.items | filter:{done: false} | orderBy:'action'">
    <td>{{item.action}}</td>
    <td><input type="checkbox" ng-model="item.done" /></td>
  </tr>
</tbody>
...
```

Filtrowanie można przeprowadzić na dowolnej części danych modelu. Jak możesz zobaczyć, w omawianej aplikacji filtry wykorzystaliśmy do wskazania danych używanych przez dyrektywę `ng-repeat`, a następnie umieszczanych na liście rzeczy do zrobienia. Tutaj zastosowaliśmy dwa filtry: `filter` (irytująca nazwa dla tak użytecznego komponentu) i `orderBy`.

Filtr `filter` wybiera obiekty na podstawie zdefiniowanych w nim kryteriów. Zdecydowaliśmy się na wybór elementów, dla których wartością właściwości `done` jest `false`. Oznacza to, że wykonane zadania nie będą wyświetlane na liście rzeczy do zrobienia. Z kolei filtr `orderBy` sortuje elementy danych i został użyty do posortowania listy względem wartości właściwości `action`. Szczegółowe omówienie filtrów znajdziesz w rozdziale 14. Efekt zastosowania wymienionych filtrów możesz zobaczyć w przeglądarce internetowej po wyświetleniu w niej dokumentu *todo.html* (patrz rysunek 2.9). Dodaj nowe zadanie do wykonania, a następnie kliknij pole wyboru w kolumnie *Zrobione*.



Rysunek 2.9. Użycie filtrowania i zmiany kolejności elementów

- **Wskazówka** Zwróć uwagę, że podczas użycia filtru `orderBy` właściwość, według której następuje sortowanie, jest podawana w postaci dosłownego ciągu tekstowego ujętego w apostrofy. Domyślnie w bibliotece AngularJS przyjęto założenie, że wszystko jest właściwością zdefiniowaną w danym zakresie. W przypadku braku znaków cytowania będzie wyszukiwana właściwość o nazwie `action`. Tego rodzaju rozwiązanie jest użyteczne w trakcie programowego definiowania wartości, ale jednocześnie wymaga pamiętania o użyciu dosłownych ciągów tekstowych, jeśli zachodzi potrzeba wskazania stałej.

Gdy do listy zostanie dodany nowy element, będzie wstawiony z zachowaniem kolejności alfabetycznej. Natomiast po zaznaczeniu pola wyboru dany element zostanie ukryty. (Dane w modelu nie są sortowane. Operacja sortowania jest przeprowadzana podczas przetwarzania dyrektywy `ng-repeat` w celu przygotowania wierszy tabeli).

Usprawnienie filtru

W poprzednim przykładzie zademonstrowano działanie operacji filtrowania. Otrzymany wynik jest jednak praktycznie bezużyteczny, ponieważ zaznaczone zadania na zawsze pozostają ukryte przed użytkownikiem. Na szczęście bardzo łatwo można utworzyć własny filtr, jak przedstawiono na listingu 2.11.

Listing 2.11. Utworzenie własnego filtru w pliku `todo.html`

```
...
<script>
  var model = {
    user: "Adam",
    items: [{ action: "Kupić kwiaty", done: false },
             { action: "Kupić buty", done: false },
             { action: "Odebrać bilety", done: true },
             { action: "Zadzwoń do Janka", done: false }],
  };
</script>
```

```

var todoApp = angular.module("todoApp", []);

todoApp.filter("checkedItems", function () {
  return function (items, showComplete) {
    var resultArr = [];
    angular.forEach(items, function (item) {
      if (item.done == false || showComplete == true) {
        resultArr.push(item);
      }
    });
    return resultArr;
  };
});

todoApp.controller("ToDoCtrl", function ($scope) {
  $scope.todo = model;
  // ... pominięto w celu zachowania zwięzłości ...
});
</script>
...

```

Metoda `filter()` definiowana przez moduł AngularJS jest używana w celu utworzenia *fabryki* filtrów, która zwraca funkcję stosowaną do filtrowania zbioru obiektów danych. W tym momencie nie przejmuj się użyciem wyrażenia *fabryka*. Wystarczy wiedzieć, że użycie metody `filter()` wymaga przekazania funkcji, której wartością zwrótną jest inna funkcja odpowiedzialna za dostarczenie przefiltrowanych danych. W omawianym przykładzie filtr otrzymał nazwę `checkedItems`, natomiast funkcja rzeczywiście przeprowadzająca filtrowanie ma dwa argumenty:

```

...
return function (items, showComplete) {
...

```

Argument `items` będzie dostarczony przez AngularJS; to zbiór obiektów przeznaczonych do filtrowania. Zapewnia wartość dla argumentu `showComplete` po zastosowaniu filtru i jest używany do określenia, czy zadania oznaczone jako wykonane mają być uwzględnione w filtrowanych danych. Sposób zastosowania własnego filtru przedstawiono na listingu 2.12.

Listing 2.12. Przykład zastosowania własnego filtru w pliku `todo.html`

```

...
<div class="panel">
  <div class="input-group">
    <input class="form-control" ng-model="actionText" />
    <span class="input-group-btn">
      <button class="btn btn-default"
        ng-click="addNewItem(actionText)">Dodaj</button>
    </span>
  </div>

  <table class="table table-striped">
    <thead>
      <tr>
        <th>Opis</th>
        <th>Zrobione</th>
      </tr>
    </thead>
    <tbody>

```

```

<tr ng-repeat=
  "item in todo.items | checkedItems:showComplete | orderBy:'action'">
  <td>{{item.action}}</td>
  <td><input type="checkbox" ng-model="item.done" /></td>
</tr>
</tbody>
</table>

<div class="checkbox-inline">
  <label><input type="checkbox" ng_model="showComplete"> Pokaż zakończone</label>
</div>
</div>
...

```

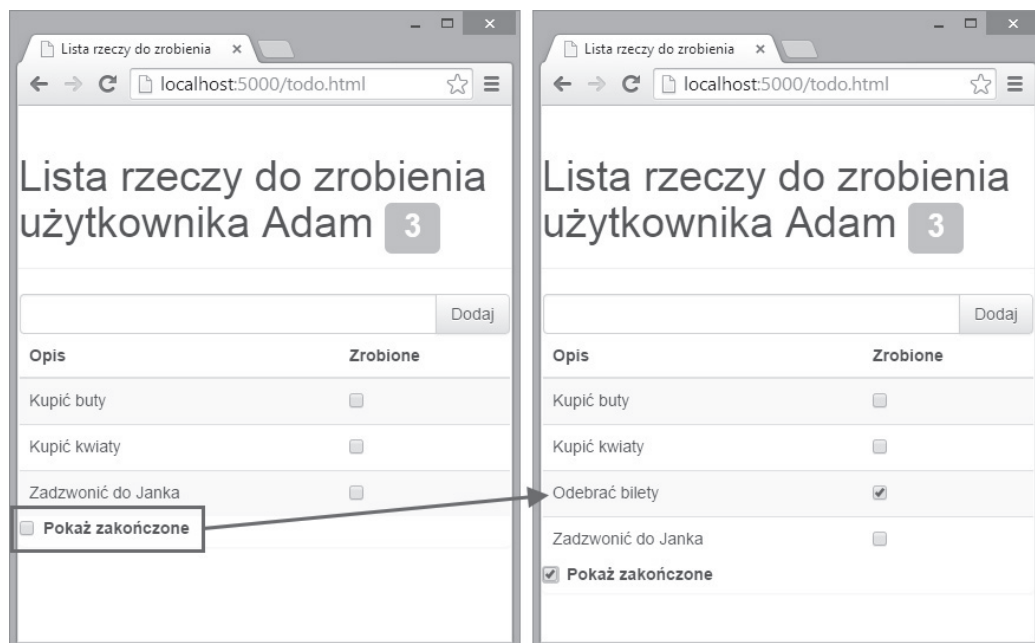
Do aplikacji dodaliśmy pole wyboru używające dyrektywy `ng-model` w celu ustawienia wartości modelu o nazwie `showComplete`; wartość ta jest przekazywana zdefiniowanemu wcześniej własnemu filtrowi w dyrektywie `ng-repeat` w tabeli.

```

...
<tr ng-repeat="item in todo.items | checkedItems:showComplete | orderBy:'action'">
...

```

Składnia własnych filtrów jest dokładnie taka sama jak dla filtrów wbudowanych. W omawianym przykładzie podaliśmy nazwę filtra tworzonego z wykorzystaniem metody `filter()`, następnie dwukropek, a dalej nazwę właściwości modelu, która ma być przekazana funkcji filtra. Użyliśmy właściwości modelu `showComplete`, co oznacza, że stan pola wyboru będzie wykorzystany do kontrolowania wyświetlania na liście zakończonych zadań. Efekt wprowadzonych zmian pokazano na rysunku 2.10.



Rysunek 2.10. Skutek użycia własnego filtra

Pobieranie danych z wykorzystaniem technologii Ajax

Ostatnią zmianą wprowadzoną w aplikacji będzie pobieranie danych jako danych w formacie JSON z wykorzystaniem żądań Ajax. (Dokładne omówienie formatu JSON znajdziesz w rozdziale 5.). W katalogu *angularjs* tworzymy plik o nazwie *todo.json* i umieszczamy w nim dane przedstawione na listingu 2.13.

Listing 2.13. Zawartość pliku *todo.json*

```
[{ "action": "Kupić kwiaty", "done": false },
  { "action": "Kupić buty", "done": false },
  { "action": "Odebrać bilety", "done": true },
  { "action": "Zadzwonić do Janka", "done": false }]
```

Jak możesz zobaczyć, dane w formacie JSON są podobne do dosłownych obiektów JavaScript. To jest główny powód, dla którego JSON jest formatem dominującym w aplikacjach sieciowych. Na listingu 2.14 przedstawiono zmiany, jakie trzeba wprowadzić w dokumencie *todo.html*, aby móc wczytywać dane z pliku *todo.json*, zamiast używać lokalnie zdefiniowanej tablicy.

Listing 2.14. Wykonywanie żądań Ajax w celu pobierania danych w formacie JSON

```
...
<script>
  var model = {
    user: "Adam"
  };

  var todoApp = angular.module("todoApp", []);

  todoApp.run(function ($http) {
    $http.get("todo.json").success(function (data) {
      model.items = data;
    });
  });

  todoApp.filter("checkedItems", function () {
    return function (items, showComplete) {
      var resultArr = [];
      angular.forEach(items, function (item) {
        if (item.done == false || showComplete == true) {
          resultArr.push(item);
        }
      });
      return resultArr;
    }
  });

  todoApp.controller("ToDoCtrl", function ($scope) {
    $scope.todo = model;

    $scope.incompleteCount = function () {
      var count = 0;
      angular.forEach($scope.todo.items, function (item) {
        if (!item.done) { count++ }
      });
      return count;
    }
  });
});
```

```

$scope.warningLevel = function () {
    return $scope.incompleteCount() < 3 ? "label-success" : "label-warning";
}

$scope.addNewItem = function(actionText) {
    $scope.todo.items.push({ action: actionText, done: false});
}
});
</script>
...

```

Usunęliśmy tablicę `items` ze statycznie zdefiniowanego modelu dodanych oraz dodaliśmy wywołanie metody `run()` zdefiniowanej przez moduł AngularJS. Wymieniona metoda `run()` pobiera funkcję wywoływaną po przeprowadzeniu przez AngularJS początkowej konfiguracji i używaną do jednorazowych zadań.

Argumentem funkcji przekazywanej metodzie `run()` jest `$http`, który nakazuje AngularJS użycie obiektu usługi zapewniającego obsługę żądań Ajax. Zastosowanie argumentów wskazujących biblioteczce AngularJS wymagane funkcje jest częścią podejścia nazywanego wstrzykiwaniem zależności, które będzie tematem rozdziału 9.

Usługa `$http` zapewnia dostęp do niskiego poziomu żądań Ajax. W omawianym przykładzie wyrażenie „niski poziom” wcale nie oznacza zbyt niskiego poziomu, przynajmniej w porównaniu z usługą `$resources` używaną do pracy z usługami sieciowymi RESTful. (Usługi sieciowe RESTful i obiekt usługi `$resources` zostaną omówione w rozdziale 21.). W przedstawionym przykładzie zastosowaliśmy metodę `$http.get()` w celu wykonania żądania HTTP GET do serwera i pobrania pliku *todo.json*:

```

...
$http.get("todo.json").success(function (data) {
    model.items = data;
});
...

```

Wynikiem zwróconym przez metodę `get()` jest tak zwana *obietnica*, czyli obiekt używany do przedstawienia zadania, którego wykonywanie zakończy się w przyszłości. Sposób działania obietnic będzie przedstawiony w rozdziale 5., natomiast ich szczegółowe omówienie znajdziesz w rozdziale 20. Teraz wystarczy wiedzieć, że wywołanie metody `success()` obiektu obietnicy pozwala na wskazanie funkcji, która będzie wywołana po zakończeniu żądania Ajax do serwera. Dane pobrane z serwera zostaną przetworzone w celu utworzenia obiektu JavaScript i przekazane funkcji `success()` jako argument `data`. W omawianym przykładzie wspomniane dane wykorzystujemy do uaktualnienia modelu:

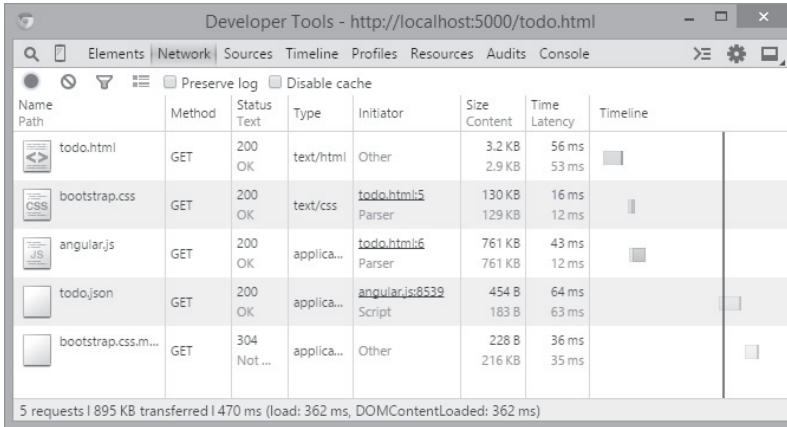
```

...
$http.get("todo.json").success(function (data) {
    model.items = data;
});
...

```

Jeżeli w przeglądarce internetowej wyświetlisz plik *todo.html*, to nie zobaczysz żadnej różnicy w sposobie działania aplikacji. Jednak dane zostały pobrane z serwera za pomocą drugiego żądania HTTP. Możesz się o tym przekonać, używając narzędzi F12 i przeglądając informacje o połączeniach sieciowych (patrz rysunek 2.11).

Konieczność potwierdzenia z wykorzystaniem przeglądarki internetowej faktu użycia żądań Ajax pokazuje, jak biblioteka AngularJS ułatwia pracę z zewnętrznymi plikami i danymi. Do tego będziemy jeszcze powracać w książce, ponieważ stanowi to fundament dla wielu funkcji, które AngularJS udostępnia w celu utworzenia znacznie bardziej złożonych aplikacji sieciowych.



Name Path	Method	Status Text	Type	Initiator	Size Content	Time Latency	Timeline
todo.html	GET	200 OK	text/html	Other	3.2 KB 2.9 KB	56 ms 53 ms	
bootstrap.css	GET	200 OK	text/css	todo.html:5 Parser	130 KB 129 KB	16 ms 12 ms	
angular.js	GET	200 OK	applica...	todo.html:6 Parser	761 KB 761 KB	43 ms 12 ms	
todo.json	GET	200 OK	applica...	angular.js:8539 Script	454 B 183 B	64 ms 63 ms	
bootstrap.css.m...	GET	304 Not ...	applica...	Other	228 B 216 KB	36 ms 35 ms	

5 requests | 895 KB transferred | 470 ms (load: 362 ms, DOMContentLoaded: 362 ms)

Rysunek 2.11. Potwierdzenie otrzymania danych za pomocą żądania Ajax

Podsumowanie

W tym rozdziale zobaczyłeś, jak utworzyć pierwszą, prostą aplikację AngularJS. Pracę rozpoczęliśmy od statycznej atrapy HTML, a następnie opracowaliśmy aplikację dynamiczną opartą na wzorcu MVC i pobierającą z serwera WWW dane w formacie JSON. W trakcie pracy zetknęliśmy się z wszystkimi najważniejszymi komponentami i funkcjami udostępnianymi programistom przez AngularJS. Ponadto dowiedziałeś się, w których rozdziałach książki szukać dokładniejszych informacji o poszczególnych komponentach.

Skoro zobaczyłeś, jak używać biblioteki AngularJS, teraz możemy wykonać krok wstecz i poznać pewne szczegóły dotyczące kontekstu, w którym istnieje AngularJS. Rozpoczniemy od wzorca MVC, który dokładnie zostanie omówiony w kolejnym rozdziale.

Skorowidz

A

- adres URL, 475, 478
- Ajax, 55, 502
- akcje, 548
- akcje obiektu dostępu, 545
- AngularJS w kontekście, 59
- animacje, 215
 - CSS3, 575
 - jednoczesne, 579
- animowanie
 - elementów, 575
 - przejsć, 577
- API, 601
 - DOM, 471
 - Fluent, 220
 - History, 478
 - RESTful, 203, 531
- aplikacja
 - administracyjna, 194
 - Deployd, 132
 - SportsStore, 131, 157
- aplikacje
 - AngularJS, 211
 - dwukierunkowe, 60
- atrapa usługi
 - \$httpBackend, 607
 - \$interval, 608
 - \$log, 611
 - \$timeout, 608
- atrapy obiektów, 601, 604
- atrybut, 74
 - highlight, 223
 - ng-app, 214

- ng-controller, 40, 219
- ng-repeat, 41
- required, 295
- atrybuty
 - boolowskie, 279, 281
 - dla pola wyboru, 308
 - elementów <input>, 307
 - niestandardowe, 35
 - weryfikacji, 294

B

- biblioteka
 - AngularJS, 26, 33
 - jQuery, 387
- jQuery, 387
- błędy Ajax, 159
- Bootstrap CSS, 91

C

- CRUD, 66
- CSS, Cascading Style Sheets, 77
- cykl życiowy modułu, 232

D

- dane
 - adresowe, 179
 - asynchroniczne, 549
 - JSON, 508
 - produkcyjne, 157
 - REST, 545
 - widoku, 65

definiowanie

- adresów URL tras, 557
- animacji, 577
- dyrektywy, 221, 375
- filtru, 223
- funkcji JavaScript, 95
- funkcji kontrolera, 189
- funkcji z parametrami, 95
- funkcji zwracającej wartość, 96
- komponentów AngularJS, 214
- kontrolera, 148, 215
- kontrolera RESTful, 203
- restrykcyjnych opcji, 403
- skomplikowanych dyrektyw, 402
- tras, 170, 563
- usługi, 226
- wartości, 228
- widoku, 190, 205
- widoku głównego, 198
- widoku uwierzytelnienia, 197

dekrementacja wartości, 600

dodawanie

- atrybutów konfiguracji, 550
- atrybutów niestandardowych, 35
- biblioteki AngularJS, 33
- danych, 134, 529
- dyrektywy, 165, 375, 441, 614
- dziedziczonych danych, 331
- dziedziczonych funkcji, 331
- elementów, 389
- elementu <form>, 293
- elementu <script>, 460, 540
- filtru, 224, 612
- formularza, 181
- funkcji filtrowania, 51
- funkcji monitorującej, 383
- kontrolera, 199, 217, 219
- nawigacji, 170
- obsługi tras, 561
- obsługiwanego atrybutu, 379
- odniesień, 206
- pól wyboru, 42, 43
- produktu do koszyka, 170
- przycisku, 168, 169
- rejestracji danych, 610
- stronicowania, 152
- testów, 609
- usługi, 616
- widoku, 218

zależności do tras, 570

zależności modułu, 577

żądania Ajax, 605

dodatki AngularJS, 26

dokument HTML, 72

dołączanie danych, data binding, 39, 43, 235, 239, 286, 487

dwukierunkowe, 42, 241, 285, 426

jednokierunkowe, 239, 423

osadzone, 241

DOM, Document Object Model, 75, 263

domknięcie, 385

dosłowna tablica, 117

dosłowny obiekt, 102

dostarczanie danych lokalnych, 494, 495

dostawca \$httpProvider, 514

dostęp do

adresu URL, 475

API RESTful, 203, 204

funkcji AngularJS, 398

kolekcji, 192

obiektów globalnych, 471

obiektu document, 473

obiektu window, 472

parametrów tras, 564

tras, 564

dwukierunkowe dołączanie

danych, 42, 241, 285, 426

modelu, 42

dwukropek, 103

dyrektywa, 41, 165, 236, 374

cartSummary, 166

disabled, 281

increment, 541

ng-app, 138

ng-bind-html, 487, 488

ng-class, 249, 270, 276

ng-class-even, 273

ng-class-odd, 273

ng-click, 50, 288, 581

ng-cloak, 260

ng-controller, 318

ng-disabled, 185, 281

ng-hide, 46, 161, 266, 268

ng-href, 281

ng-if, 267, 268

ng-include, 162, 200, 251–256

ng-model, 54, 286–290, 445

ng-repeat, 141, 244–248

- ng-repeat-end, 250
- ng-repeat-start, 250
- ng-show, 266
- ng-src, 281
- ng-srcset, 281
- ng-style, 270, 272
- ng-switch, 256–258
- ng-transclude, 433
- ng-view, 172, 579
- promiseObserver, 525
- dyrektywy
 - atributu boolowskiego, 280
 - dołączania danych, 238
 - elementów, 265
 - jako atrybut, 405
 - jako element, 405
 - jako komentarz, 406
 - jako wartości atrybutu klasy, 406
 - obsługujące animacje, 578
 - skomplikowane, 401
 - szablonów, 243
 - zdarzeń, 274, 277
- działania użytkownika, 48
- działanie koszyka na zakupy, 163
- dziedziczenie
 - funkcjonalności, 102
 - kontrolerów, 319, 328, 330, 332

E

- edycja danych, 207
- edytor tekstów, 24
- elastyczny układ, 88
- element, 73, 76
 - <a>, 146, 562
 - <button>, 185
 - <div>, 160
 - <form>, 182, 293, 294
 - <html>, 33, 76
 - <input>, 184, 286, 302, 306–308
 - , 378, 499
 - <meta>, 88
 - , 390, 395
 - <optgroup>, 313, 314
 - <option>, 311, 312
 - <panel>, 434
 - <script>, 33, 93, 460, 540
 - <select>, 310, 312
 - , 284

- <style>, 183, 578
- <table>, 80, 86
- <tbody>, 436
- <textarea>, 309
- <thead>, 82
- <tr>, 271, 436
- , 498

elementy

- formularza sieciowego, 84, 185
- najwyższego poziomu, 250
- nawigacyjne, 146
- obiektu Module, 215, 459
- potomne, 389
- typu void, 75
- własne formularza, 443
- wyświetlające treść, 141

F

- fabryka, 53, 222
- filtr, 223, 343–372
 - currency, 142, 346
 - date, 346
 - filter, 360
 - json, 346, 353
 - limitTo, 357, 366, 370
 - lowercase, 346, 352
 - number, 346
 - orderBy, 51, 146, 362, 366
 - pageCount, 154, 156
 - range, 154
 - skip, 370
 - take, 371, 372
 - unique, 144, 147
 - uppercase, 346, 352
- filtrowanie, 51
 - kolekcji, 357
 - produktów, 149
- finalizacja zamówienia, 174, 176
- format
 - dosłownego obiektu, 102
 - JSON, 55, 128
- formatowanie
 - daty, 349, 351, 356
 - wartości danych, 448
 - wartości pieniężnych, 347, 348
 - zlokalizowanych danych, 354
- formaty danych, 68
- formularz, 83, 181–185, 283

framework

- AngularJS, 19
- Bootstrap CSS, 27, 71, 77

funkcja

- addNetItem(), 288
- addUser(), 305
- beforeEach(), 600
- changeData(), 437
- cityFn(), 429
- compile, 435, 437
- createProduct(), 547
- editOrCreateProduct(), 535, 560
- editProduct(), 560
- fabryki, 317
- getCategoryClass(), 151
- getCity(), 429
- getCountry(), 319
- handleClick(), 341, 585
- incompleteCount(), 46
- incrementCounter(), 597
- inject(), 617
- it(), 617
- link, 376
- listProduct(), 537
- logClick(), 587
- printMessage(), 98
- reverseText(), 334
- setAddress(), 323
- shiftFour(), 331
- worker, 222

funkcje

- API DOM, 472
- domknięć, 229
- dyrektyw, 431
- fabryki, 222
- filtrowania, 51
- Jasmine, 598
- JavaScript, 91
- konstruktora, 465
- kontrolera, 44, 189, 190
- monitorujące, 383
- operujące na danych, 533

G

generowanie

- danych, 353
- elementów, 141, 245, 378
- elementów nawigacyjnych, 146

listy, 394

łączy, 144

wielu elementów, 250

XML, 513

gest machnięcia, 581

gesty, 581

grupowanie obietnic, 524, 525

H

hierarchia zakresów, 336

HTML, 71, 91

HTML5, 477

I

IIFE, 386

implementacja

filtrów, 153

uwierzytelniania, 195

wzorca MVC, 62

informacje o produkcie, 140, 572

instalacja

AngularJS, 136

Bootstrap CSS, 136

modułu ngAnimation, 576

modułu ngMock, 594

modułu ngResource, 542

modułu ngRoute, 556

modułu ngTouch, 581

Node.js, 24

serwera WWW, 25

systemu przeznaczonego do testowania, 25

interceptor żądania, 515

interpolacja ciągów tekstowych, 495

J

Jasmine, 597

JavaScript, 91

definiowanie funkcji, 94

funkcje z parametrami, 95

funkcje zwracającej wartość, 96

obiekty, 101

obietnice, 124

operatory, 110

polecenia, 94

tablice, 116

typy, 99

- wartości specjalne, 120
- wykrywanie funkcji, 97
- wykrywanie obiektów, 104
- zmiennie, 98
- jawna konwersja typów, 114
- jednokierunkowe dołączanie danych, 239, 423
- jQuery, 387, 399
- jQuery, 61, 399
- jQuery UI, 338
- JSON, JavaScript Object Notation, 128

K

- Karma, 597
- katalog
 - AngularJS, 26
 - produktów, 192, 203
- klasa
 - grid-row, 87
 - ngFade.ng-enter, 578
- klasy
 - Bootstrap, 79, 80, 87
 - weryfikacji, 297
- klucze obiektów danych, 246
- kolejność sortowania, 363
- kolekcja, 133, 529
- kolekcja użytkowników, 193
- kompilacja zawartości, 498
- komunikacja
 - dwukierunkowa, round-trip, 60
 - między zakresami, 325
- komunikat o błędzie, 161, 229
- konfiguracja
 - akcji usługi \$resource, 548
 - interpolacji, 496
 - jQuery UI, 338
 - oprogramowania Karma, 595
 - routingu, 171, 174
 - serwera Deployd, 187
 - środowiska programistycznego, 23
 - tras, 567
 - usługi \$resource, 544
 - zakresu, 318
 - zadań Ajax, 509
- konstrukcje warunkowe, 110
- konstruktor, 317
- kontrola stanu przycisku, 340
- kontroler, 37, 62, 65, 315–342
- ngModel, 446, 451

- productListCtrl, 150
- tableCtrl, 572
- tomorrowCtrl, 220
- kontrolery
 - bez zakresu, 337
 - monolityczne, 322
 - najwyższego poziomu, 138
 - w dyrektywach, 438
- konwersja
 - ciągów tekstowych, 115
 - liczb, 114
 - wyrażenia, 492
- koszyk na zakupy, 163

L

- liczba kliknięć, 457
- lista kategorii, 143
- logika
 - domeny, 68
 - domeny biznesowej, 65, 68
 - magazynu danych, 68
 - modelu, 63
- lokalizacja, 356
- lokalizacja danych, 354

Ł

- łącza, 144
- łączenie
 - filtrów, 366
 - kodu i znaczników, 559
 - metod, 378
 - obietnic, 126, 522

M

- magazyn danych, 68
- metoda
 - \$apply(), 341
 - \$broadcast(), 327
 - \$get(), 547
 - \$http.get(), 158
 - \$injector.invoke(), 589
 - \$location.path(), 556, 560
 - \$log.log(), 612
 - \$q.defer(), 519
 - \$render, 446
 - \$route.reload(), 572

metoda

\$routeProvider.when(), 558
 \$sce.trustAsHtml(), 491
 \$setViewValue(), 447
 \$watch(), 341
 addProduct(), 165, 169
 angular.forEach(), 45
 angular.fromJson(), 129
 angular.isArray(), 377
 angular.isDefined(), 123
 angular.isFunction(), 98
 angular.isObject(), 104
 angular.isString(), 100
 angular.isUndefined(), 123
 angular.module(), 34, 137, 213, 230
 annotate(), 586
 appendTo(), 400
 assertEmpty(), 612
 children(), 388, 389
 concat(), 120
 config(), 232, 233
 constant(), 233
 controller(), 37, 148, 317
 css(), 223, 392
 decorator(), 585
 delete(), 67, 544
 directive(), 166, 222
 error(), 126, 482, 506
 expect(), 606, 607
 factory(), 461, 482, 572, 617
 filter(), 53, 143
 find(), 390
 flush(), 607, 609
 get(), 56, 127
 getProducts(), 165
 has(), 588
 html5Mode(), 478
 invoke(), 589
 join(), 120
 listProduct(), 537
 log(), 585
 Module.config(), 232
 Module.directive(), 278, 375
 Module.factory(), 164
 Module.run(), 232
 Module.service(), 464
 Module.value(), 228, 229
 on(), 278
 otherwise(), 195, 558

pop(), 120
 POST, 67
 provider(), 461, 466
 push(), 50, 120
 PUT, 67
 query(), 545
 removeProduct(), 165
 reset(), 612
 respond(), 606
 reverse(), 120
 run(), 232, 233
 scope.\$apply(), 447
 service(), 228, 461, 464
 shift(), 120
 slice(), 120
 sort(), 120
 splice(), 120, 154
 success(), 125, 506
 then(), 126, 127, 507
 trustAsHtml(), 490
 unshift(), 120

metody

do obsługi tablic, 120
 HTTP, 67
 integracji zakresu, 338
 jqLite, 387, 391, 398
 konwersji, 116
 obiektu angular.mock, 602
 obiektu obietnicy, 521
 obiektu Resource, 546
 usługi \$http, 505, 506
 usługi \$httpBackend, 606
 usługi \$injector, 586
 usługi \$location, 475
 usługi \$log, 481
 usługi \$provide, 584
 usługi \$q, 518
 usługi \$route, 566
 weryfikacji, 451
 zakresu, 326
 model, 37, 62, 63
 danych, 35
 domen, 63
 widoku, 63
 moduł, 213, 455
 customFilters, 145
 koszyka, 163
 ngAnimation, 576
 ngMock, 594, 601

moduł
 ngResource, 542
 ngRoute, 556, 557
 ngTouch, 580, 581
 modyfikacja
 dyrektywy, 226
 elementów, 391
 kontekstu stylu, 79
 obiektów danych, 546
 wielkości elementu, 79
 właściwości obiektu, 105
 zakresu, 320
 zawartości tablicy, 118
 monitorowanie poprawności formularza, 295
 MVC, Model-View-Controller, 19, 35, 62

N

nadawanie stylu
 komunikatom, 297
 tabeli, 267
 nadpisywanie
 danych, 332
 funkcji, 332
 narzędzie
 LiveReload, 27
 Yeoman, 23
 nawiasy
 klamrowe, 40
 kwadratowe, 106
 nawigacja, 170, 562
 nazwa, name, 74
 attributu, 74
 kontrolera, 38
 niebezpieczne dane, 485
 niejawne tworzenie właściwości, 286
 niejawnie zdefiniowane właściwości modelu, 289

O

obiekt
 \$scope, 125
 angular.mock, 602
 deferred, 518
 definicji, 403
 document, 473
 dostępu, 545
 modelu danych, 289
 Module, 214, 226

 newTodo, 289
 Resource, 546
 typu singleton, 226
 usługi, 462
 window, 472
 obiektowy model dokumentu, 75
 obiekty
 bez wartości, 229
 dodawanie metod, 108
 dodawanie właściwości, 107
 modyfikacja właściwości, 105
 rozbudowa, 103
 sprawdzanie identyczności, 112
 sprawdzanie równości, 112
 sprawdzenie właściwości, 109
 testowe, 601
 usuwanie właściwości, 107
 wykrywanie, 104
 wyświetlenie właściwości, 106
 obietnice, 124, 516, 521
 obliczanie wartości wyrażeń, 41, 380–382, 428
 obsługa
 akcji, 548
 błędów Ajax, 159
 finalizacji zamówienia, 170
 funkcji animacji, 215
 gestu machnięcia, 581
 kontrolera dyrektywy, 439
 koszyka, 168
 modułu, 458
 routingu, 561
 stronicowania, 155
 tablic, 120
 wyjątków, 484
 wyświetlania danych produktu, 138
 zdarzeń, 274, 396, 582
 zdarzeń click, 147
 zdarzeń dotknięć, 580
 zmian wewnętrznych, 447
 zmian zewnętrznych, 445
 zmiany danych, 382
 oczekiwane odpowiedzi, 606
 odblokowanie przycisku, 184
 odczyt
 tablicy, 118
 wartości właściwości, 105
 właściwości obiektu, 105
 odizolowany zakres, 424
 ograniczenie liczby elementów, 357

opcje konfiguracji tras, 568
 operator
 identyczności, 111
 konkatenacji, 114
 równości, 111
 operatory JavaScript, 110
 optymalizacja, 191
 organizacja kodu, 229
 organizowanie kontrolerów, 321

P

parametr zachłanny, 563
 parametry trasy, 562
 pierwsza aplikacja, 31
 plik
 adminControllers.js, 199
 adminMain.html, 195, 199
 adminOrders.html, 201
 adminProduct.html, 205
 adminProducts.html, 198
 ajax.html, 514
 angular.js, 136
 angular-animate.js, 576
 angular-resource.js, 136
 angular-route.js, 136
 app.html, 137
 app.js, 616
 bootstrap.css, 136
 bootstrap.html, 77, 82
 bootstrap-theme.css, 136
 cart.js, 164
 cartSummary.html, 166
 checkoutSummary.html, 170, 175
 compileFunction.html, 435
 components.html, 584, 586
 controllers.html, 316, 335
 controllers.js, 230, 330, 332
 controllers/productListControllers.js, 148
 controllerTest.js, 602, 609
 customFilters.js, 143, 367, 369
 directiveControllers.html, 438
 directives.html, 239, 260, 275, 376, 402, 425
 directives.js, 458
 directiveScopes.html, 428
 directiveTest.js, 615
 domApi.html, 472, 476, 478
 editorView.html, 534
 exceptions.html, 483

expressions.html, 492
 filters.html, 344, 350, 360
 filters.js, 230
 filterTest.js, 613
 firstTest.js, 597, 599, 600
 forms.html, 284, 292
 htmlData.html, 486
 increment.js, 540, 549
 jqLite.html, 392, 395
 list.html, 253
 placeOrder.html, 183, 185
 products.html, 532
 products.js, 536, 568
 promises.html, 517, 522, 524
 serviceTest.js, 617
 sportsStore.js, 139, 189
 swipe.html, 581
 table.html, 251, 253
 tableView.html, 533, 550
 thankYou.html, 190
 todo.html, 31
 pliki testowe, 597
 pobieranie
 danych, 55, 159
 danych adresowych, 179
 danych z zakresu, 377
 egzemplarzy usługi, 588
 klucza, 246
 parametrów trasy, 567
 pliku lokalizacji, 345
 usługi \$injector, 590
 podświetlenie
 kategorii, 150
 przycisku, 152
 podział odpowiedzialności, 62
 pole wyboru, 308
 polecenia w JavaScript, 94
 ponowne użycie kontrolera, 324
 poprawiona zmienna, 385
 potwierdzenie otrzymania danych, 57
 problem leksykalny, 385
 programowanie asynchroniczne, 125
 przecinek, 103
 przeglądanie
 produktów, 203
 zamówień, 200, 202
 przeglądarka, 23
 przekazywanie argumentów, 319
 przetwarzanie danych, 508

przewijanie
 elementów, 480
 okna przeglądarki, 479

przycisk
 wyboru produktu, 168
 nawigacyjny, 146

przygotowanie
 aplikacji, 135
 testu, 603

przypisywanie właściwości, 106

R

rejestracja
 danych, 481, 610
 komponentów, 584
 REST, Representational State Transfer, 527
 RIA, Rich Internet Applications, 19
 routing, 171, 172, 556
 rozbudowa obiektów, 103
 rozszerzanie modułu, 145
 rozszerzenie Batarang, 44
 rozwiązywanie zależności, 145, 603

S

sanityzacja, 489, 490
 serwer Deployd, 28, 187, 192
 siatka, 85–87
 składanie zamówień, 187
 składnia własnych filtrów, 54
 sortowanie, 51
 elementów, 362
 obiektów, 362
 według funkcji, 363
 z użyciem predykatów, 364
 SportsStore, 131
 administracja, 192
 dane produkcyjne, 157
 koszyk, 163
 nawigacja, 170
 usprawnienia, 191
 widok częściowy, 161
 wyświetlenie danych produktu, 138
 wyświetlenie listy kategorii, 143
 zamówienia, 174, 187
 sprawdzanie
 obiektu modelu danych, 289
 równości i identyczności obiektów, 112

równości i identyczności typów, 113
 testów, 604
 wyników, 608

stan
 elementu `<input>`, 299
 przycisku, 340

stronicowanie, 152, 155, 156

struktura
 aplikacji, 458
 danych, 132, 528
 dokumentu, 75
 katalogu, 135
 projektu, 212
 tabeli, 82

styl
 dosłownej tablicy, 117
 tabeli, 80

style
 Bootstrap, 79
 CSS, 77, 182, 269, 298
 sygnalizacja wyniku, 522
 symbol waluty, 348
 symulacja
 czasu, 608
 odpowiedzi HTTP, 604
 szablony
 dyrektyw, 243, 407
 wyrażen, 259

T

tabele, 81, 86, 271
 tablice, 116
 technologia Ajax, 55
 test, 28
 test jednostkowy, 131, 593, 594
 testowanie, 599
 API, 530
 aplikacji, 534
 dyrektywy, 614
 filtru, 612
 implementacji Ajax, 539
 kontrolera, 602
 rejestracji danych, 610
 usługi, 615
 usługi danych, 134
 transformacja
 ciągu tekstowego, 487
 odpowiedzi, 510
 ządania, 511

transkluzja, 269, 432, 434

trasa, 557

trasy URL, 170, 172

tworzenie

aplikacji administracyjnej, 194

aplikacji AngularJS, 31, 132, 213, 531

aplikacji sieciowej, 31

dwukierunkowego dołączania danych, 426

dyrektywy, 166, 221

dyrektywy zdarzeń, 277

elementów, 393

elementów <optgroup>, 313

elementów listy, 51

filtru, 52, 371

filtru kolekcji, 369

formularzy sieciowych, 83

jednokierunkowego dołączania danych, 423

kolekcji użytkowników, 193

konfiguracji testowej, 594

kontrolera, 37, 138, 215, 317

kontrolera monolitycznego, 322

koszyka, 163

listy kategorii, 143

modelu danych, 35

modułu, 213, 456, 460

obiektów, 101, 547

odizolowanych zakresów, 421

produktu, 535, 538

struktury danych, 132, 528

struktury katalogu, 135

testów jednostkowych, 598

układu opartego na siatce, 85

układu strony, 136

usługi, 227, 456, 461

usługi typu RESTful, 528

widoków częściowych, 161

widoku, 39

widżetu koszyka, 165

wielu kontrolerów, 219, 417

wielu widoków, 218

własnego filtru, 367

własnych dyrektyw, 373, 375

własnych elementów, 443

właściwości modelu, 286

typ, 99

boolean, 99

number, 101

string, 99

typy wartości atrybutów, 295

U

uaktualnianie

zakresu, 338

kontrolera, 152

produktu, 539

widoku, 155

układ

oparty na siatce, 85, 87

strony, 136

ukrywanie

elementów, 260, 265

zadań Ajax, 539

umieszczanie logiki

biznesowej, 68

domeny, 68

magazynu danych, 68

uproszczenie procesu wywołania, 589

usługa, 226

\$anchorScroll, 479

\$animation, 576

\$compile, 498

\$errorHandler, 484

\$exceptionHandler, 482, 483

\$http, 125, 203, 504, 536

\$httpBackend, 606, 607

\$injector, 586, 590

\$interpolate, 496

\$interval, 474, 609

\$location, 475, 559

\$log, 481, 585, 611

\$parse, 493

\$provide, 584

\$q, 518

\$resource, 542–544, 549

\$rootElement, 590

\$route, 171, 557, 566

\$sce, 490

\$scope, 218, 228

\$swipe, 580

\$timeout, 474, 609

\$window, 473

cart, 165

days, 228

usługi

dla animacji, 575

dla błędów, 471

dla dotknięć, 575

dla obiektów globalnych, 471

- dla REST, 527
- dla technologii Ajax, 501
- dla widoków, 553
- dla wyrażeń, 471
- oferujące wyrażenia, 491
- rejestracji komponentów, 583
- sieciowe RESTful, 56, 64–67, 528
- wbudowane, 468, 469
- usprawnienie filtru, 52
- usuwanie
 - elementów, 265, 267, 393
 - obiektu danych, 547
 - produktu, 538
- uwierzytelnianie, 195, 197, 198
- używanie
 - adresów URL, 477
 - AngularJS, 33
 - API Fluent, 220
 - atrap, 601, 604
 - atrapy usługi \$log, 611
 - atrybutów weryfikacji, 294
 - CSS, 297
 - danych odizolowanego zakresu, 430
 - dosłownych obiektów, 102
 - dwukierunkowego dołączania modelu, 42
 - dyrektyw, 41, 48, 236, 238
 - dyrektyw dołączania danych, 237
 - dyrektyw elementu, 264
 - dyrektywy ng-bind-html, 488
 - dyrektywy ng-class-odd, 273
 - dyrektywy ng-disabled, 281
 - dyrektywy ng-if, 268
 - dyrektywy ng-include, 251, 255, 256
 - dyrektywy ng-repeat, 244
 - dyrektywy ng-switch, 256, 258
 - elementu <input>, 286, 306, 308
 - elementu <select>, 310
 - elementu <textarea>, 309
 - filtrów, 52, 225, 269, 343
 - filtru limitTo, 357, 359
 - filtru orderBy, 362
 - formatowania danych, 448
 - frameworka Bootstrap, 80, 83, 85
 - funkcji, 46
 - funkcji compile, 435
 - funkcji do sortowania, 365
 - funkcji jako metod, 103
 - funkcji jako szablonu, 409
 - funkcji JavaScript, 94
 - interceptorów Ajax, 515
 - jednokierunkowego dołączania danych, 425
 - jQuery, 513
 - konstrukcji warunkowych, 110
 - kontrolera monolitycznego, 322, 323
 - kontrolerów, 316
 - kontrolerów w dyrektywach, 438
 - kontrolerów z trasami, 568
 - metody factory(), 461
 - metody provider(), 466
 - metody service(), 464
 - metody then(), 507
 - modułów, 214, 229, 458
 - obiektu deferred, 518, 521
 - obietnicy, 519
 - operatorów JavaScript, 110
 - parametrów trasy, 562
 - pól wyboru, 308
 - routingu, 173, 556, 564
 - stylu dosłownej tablicy, 117
 - symbolu waluty, 348
 - szablonu dyrektywy, 243, 407
 - ścieżki adresu URL, 173
 - transkluzji, 432, 434
 - typów podstawowych, 99
 - usług rejestracji komponentów, 583
 - usług widoku, 553
 - usługi, 327, 461, 463
 - usługi \$anchorScroll, 479
 - usługi \$http, 504, 536
 - usługi \$interval, 474
 - usługi \$location, 476, 559
 - usługi \$log, 481
 - usługi \$parse, 493
 - usługi \$resource, 543, 549
 - usługi animacji, 576
 - wartości, 229
 - wbudowanych zmiennych, 247, 248
 - widżetu koszyka, 167
 - wielu kontrolerów, 335
 - własnego filtru, 54, 368
 - własnej akcji, 548
 - zewnętrznego szablonu, 410
 - zmiennych, 98, 99
 - zmiennych specjalnych, 301

W

wartości domyślne żądania, 513
 wartość, value, 74
 null, 120, 122
 undefined, 120, 122
 warunkowe zastępowanie elementów, 256
 wbudowane zmienne ng-repeat, 248
 wczytywanie
 danych JSON, 507
 kontrolera, 196
 weryfikacja
 adresów, 295
 danych, 181
 formularza, 290–293, 296, 301, 449
 własnych elementów, 449
 zmiennych, 296
 widoczność elementu, 265
 widok, 37, 39, 62, 65
 widok uwierzytelnienia, 197
 widoki częściowe, 161, 251
 widżet koszyka, 165, 167
 własne dyrektywy, 374
 własny filtr, 52
 właściwości
 interceptora, 516
 modelu, 286, 289
 weryfikacji, 451
 właściwość
 \$httpProvider.interceptor, 516
 city, 319
 controller, 440, 571
 require, 441
 resolve, 570
 worker, 222, 317
 wstawianie wartości modelu, 40
 wstrzykiwanie zależności, 216, 586
 wstrzymanie wyświetlania komunikatów, 304, 306
 wybór
 elementów, 360
 kategorii, 150
 szablonu zewnętrznego, 411
 widoków częściowych, 252
 wyjątki, 482
 wykonywanie żądania Ajax, 505
 wykrywanie
 funkcji, 97
 gestów, 582
 obiektów, 104

wypełniony element <input>, 299, 300
 wyrażenia, 491
 wyrażenie typu IIFE, 386
 wyszukiwanie
 elementów potomnych, 389
 modułu, 214
 wyświetlanie
 błędów, 159
 danych, 536
 danych produktu, 138
 danych REST, 545
 elementów, 265, 267
 informacji o produktach, 140
 komunikatów, 296–303
 listy kategorii, 143
 niebezpiecznych danych, 486
 treści, 141
 widoku, 172
 właściwości obiektu, 106
 wybranego widoku, 558
 zaufanych danych, 487
 zawartości tablicy, 119
 wywołanie funkcji kontrolera, 190
 wzorzec, 63
 wzorzec MVC, 35, 62

Z

zabezpieczenie kolekcji, 193
 zakres, scope, 38, 159, 315–342
 zakresy odizolowane, 421
 zakupy, 163
 zależności
 funkcji, 586
 właściwości danych, 379
 zarządzanie
 atributami boolowskimi, 280
 zakresami dyrektywy, 415
 zastępowanie elementów, 256, 413
 zastosowanie
 dyrektyw, 223
 kontrolera, 44, 149, 317
 zlokalizowanego formatowania, 356
 zawartość elementu, 74
 zdarzenia, 274, 277, 396, 521
 usługi \$location, 476
 zakresu, 327
 zdarzenie
 click, 50, 147
 ngTouch, 581

zdefiniowane trasy, 558

zliczanie kliknięć, 341

zmiana

 danych, 382

 elementu <option>, 311

 trasy, 566

 wartości danych, 445

 wartości w zakresie, 321

 wielkości liter, 352

 właściwości dataValue, 446

 wybranej wartości, 312

zmiany

 wewnętrzne, 447

 zewnętrzne, 445

zmienna

 \$dirty, 296

 \$error, 296

 \$index, 247

 \$invalid, 296

 \$oodd, 249

 \$pristine, 296

 \$valid, 296

 displayMode, 554

 ng-dirty, 297

 ng-invalid, 297

 ng-pristine, 297

 ng-valid, 297

zmienne

 globalne, 98

 lokalne, 98

 specjalne, 301

 wbudowane, 247

znacznik, 73

znak @, 423

Ż

żądania

 Ajax, 55, 125, 158, 502, 506

 oczekiwane, 606

żądanie

 GET, 56, 505

 POST, 67, 505

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



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

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

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

GRUPA WYDAWNICZA



Helion SA

AngularJS

Profesjonalne techniki

AngularJS to szkielet do tworzenia zaawansowanych aplikacji JavaScript, wykorzystujący w pełni możliwości współczesnych przeglądarek internetowych. Pozwala on na stosowanie wzorca MVC (ang. Model View Controller) po stronie klienta, a co za tym idzie, na zachowanie wysokiej jakości oraz przejrzystej struktury kodu nawet w przypadku aplikacji intensywnie używających komunikacji sieciowej. Za tym popularnym szkieletem stoi firma Google, która dba o jego ciągły rozwój.

Ta książka została w całości poświęcona szkieletowi AngularJS. Sięgnij po nią i przekonaj się, w jaki sposób przygotować środowisko programistyczne, zbudować pierwszą aplikację i uzyskać dostęp do kontekstu. W kolejnych rozdziałach zaznajomisz się z możliwościami biblioteki Bootstrap oraz przejdziesz krótki kurs programowania w języku JavaScript. W części drugiej szczególny nacisk został położony na detale związane z pracą z AngularJS. Dowiesz się stąd, jak korzystać z dyrektyw, tworzyć własne dyrektywy oraz używać kontrolerów i zakresów. Ostatnia część książki została poświęcona pracy z modułami i usługami. Znajdziesz tu wskazówki, jak pobrać dane z usług sieciowych, przetworzyć je i zaprezentować użytkownikowi. Książka ta jest obowiązkową lekturą każdego programisty pracującego z AngularJS.

Dzięki tej książce:

- poznasz tajniki programowania w JavaScriptcie
- dowiesz się, do czego służą dyrektywy i jak tworzyć własne
- pobierzesz dane z udostępnionych usług sieciowych
- przetestujesz stworzony kod
- poznasz tajemnice AngularJS

Apress

Helion

27937 numer katalogowy

księgarnia internetowa

<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

Sprawdź najnowszą promocję:

• <http://helion.pl/promocje>

Książki najchętniej czytane:

• <http://helion.pl/bestsellery>

Zamów informacje o nowościach:

• <http://helion.pl/nowosci>

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

sięgnij po **WIĘCEJ**



KOD KORZYŚCI

ISBN 978-83-283-0197-9



9 788328 301979

Informatyka w najlepszym wydaniu

cena: 99,00 zł