TWÓJ PODRĘCZNIK PROGRAMISTY!

Apress

AngularJS

Profesjonalne techniki

Adam Freeman

Helion

Tytuł oryginału: Pro AngularJS

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-0197-9

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.

Spis treści

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.

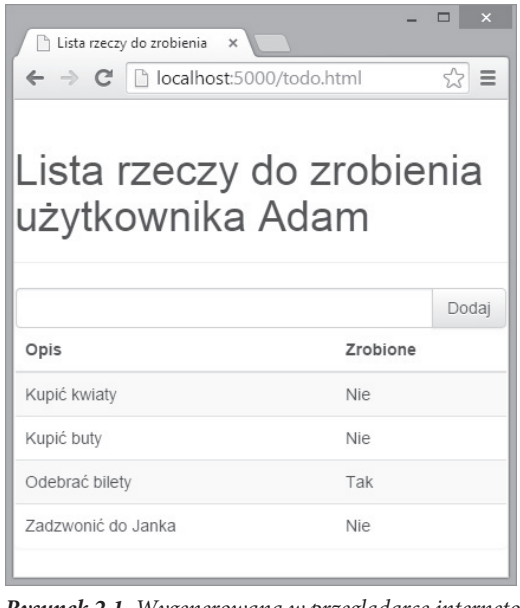
Listing 2.1. *Początkowa zawartość pliku todo.html*

```
<!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>
```

```
        <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>Zadzwonić 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 = {
```

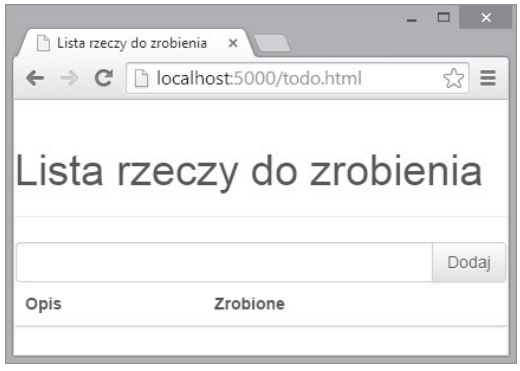
```
            user: "Adam",
            items: [{ action: "Kupić kwiaty", done: false },
                { action: "Kupić buty", done: false },
                { action: "Odebrać bilety", done: true },
                { action: "Zadzwonić do Janka", done: false }]
        };

        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 opisania 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 wzorzec 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: "Zadzwonić 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 powrócimy w rozdziale 13.

Utworzenie widoku

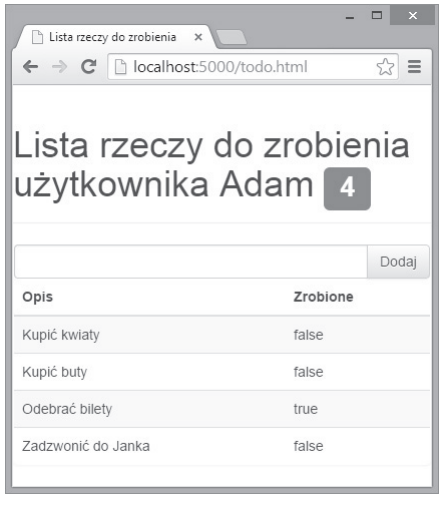
Wygenerowanie widoku następuje przez połączenie danych dostarczanych przez kontroler z elementami HTML tworzącymi zawartość wyświetlaną przez przeglądarkę internetową. Na listingu 2.5 przedstawiono użycie jednego z rodzajów adnotacji nazywanego *dołączaniem danych* (ang. *data binding*), co powoduje 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>Opis</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>
...
```

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 umieść 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">Zadzwonić 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.

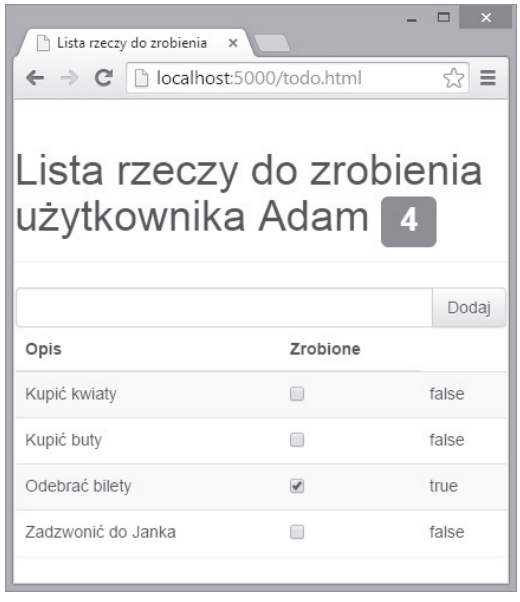
AngularJS idzie o krok dalej i zapewnia *dołączanie dwukierunkowe* — model jest używany w celu wygenerowania elementów, a zmiany elementu powodują wprowadzenie odpowiednich modyfikacji w modelu. Aby zademonstrować implementację dołączania dwukierunkowego, zmodyfikujemy plik *todo.html* (patrz listing 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>
...
```

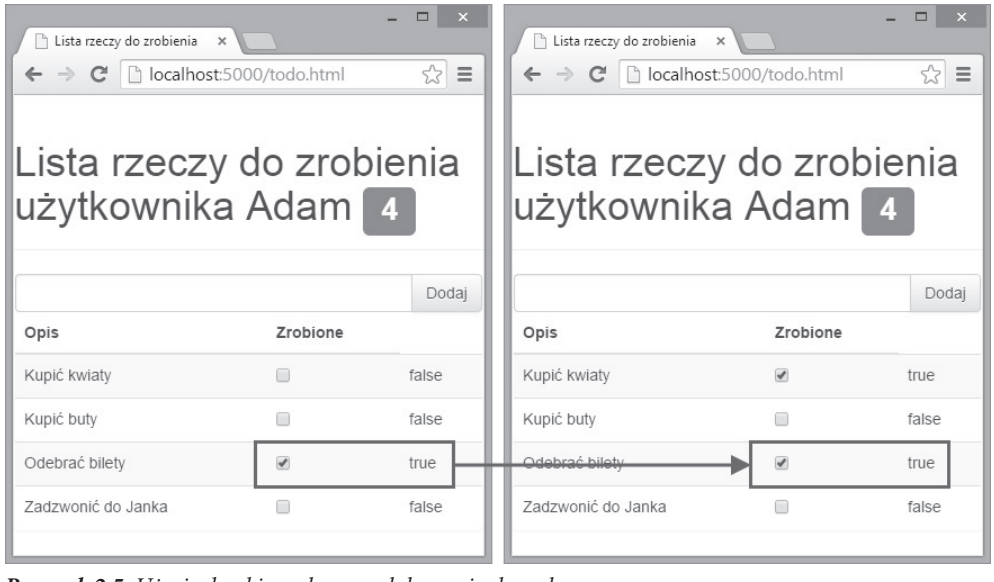
W tabeli umieściliśmy nowy element `<td>` przeznaczony do przechowywania elementu `<input>` w postaci pola wyboru. Najważniejszym dodatkiem jest tutaj atrybut `ng-model` nakazujący AngularJS utworzenie dwukierunkowego wiązania między wartością elementu `<input>` i właściwością `done` odpowiadającego 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>
</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.

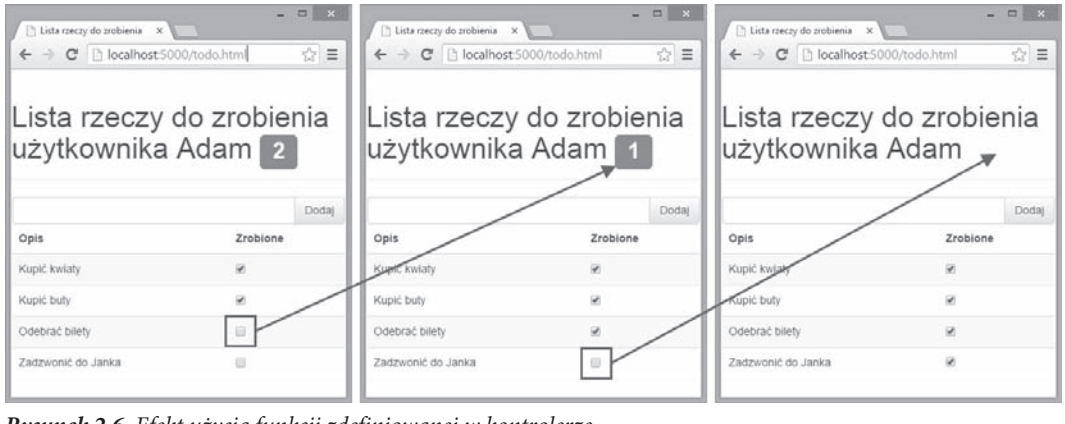
Po raz drugi funkcja została użyta w połączeniu z dyrektywą:

```
...
<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 obiektowym 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*

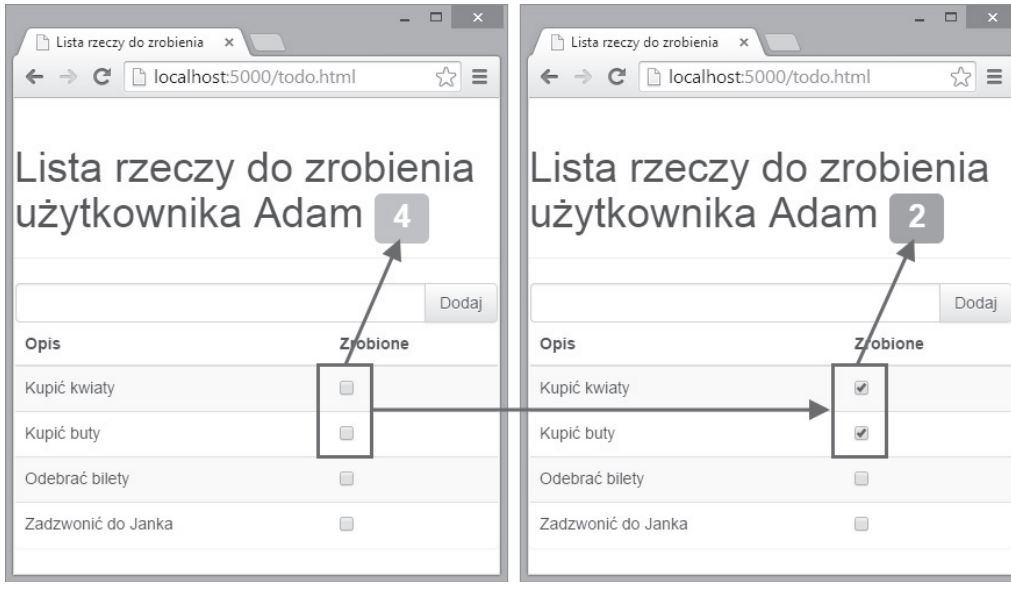
```
<!DOCTYPE html>
<html ng-app="todoApp">
<head>
    <title>Lista rzeczy do zrobienia</title>
    <link href="bootstrap.css" rel="stylesheet" />
    <link href="bootstrap-theme.css" rel="stylesheet" />
    <script src="angular.js"></script>
    <script>
        var model = {
            user: "Adam",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

            $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: "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;
            }

            $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" />
...
```

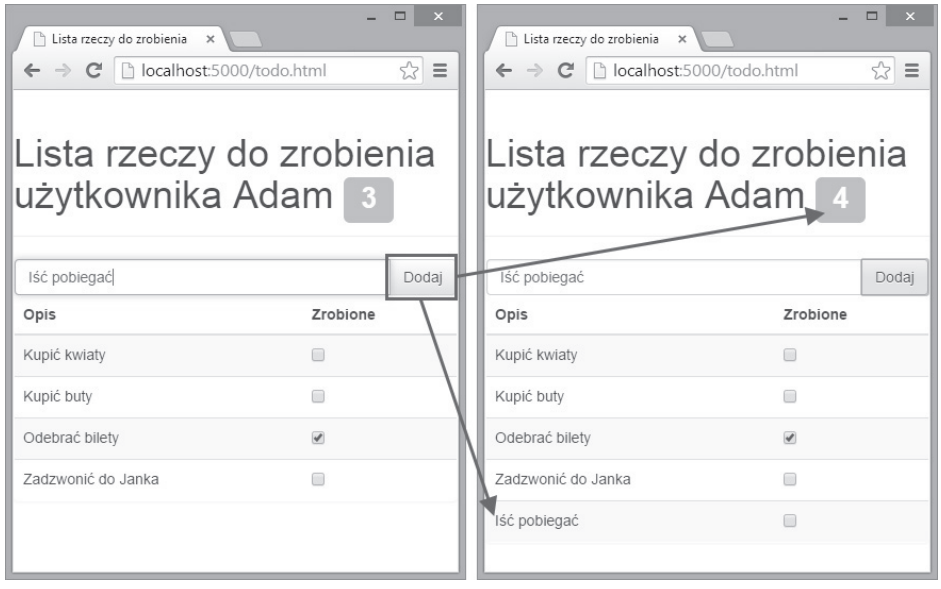
To jest dokładnie ta sama dyrektywa `ng-model`, którą wykorzystaliśmy wcześniej do konfiguracji pól wyboru. Wielokrotnie spotkasz tę dyrektywę podczas pracy z elementami formularza. Warto zwrócić uwagę na podanie nazwy właściwości uaktualnianej przez dyrektywę — nie jest ona częścią modelu. Dyrektywa `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

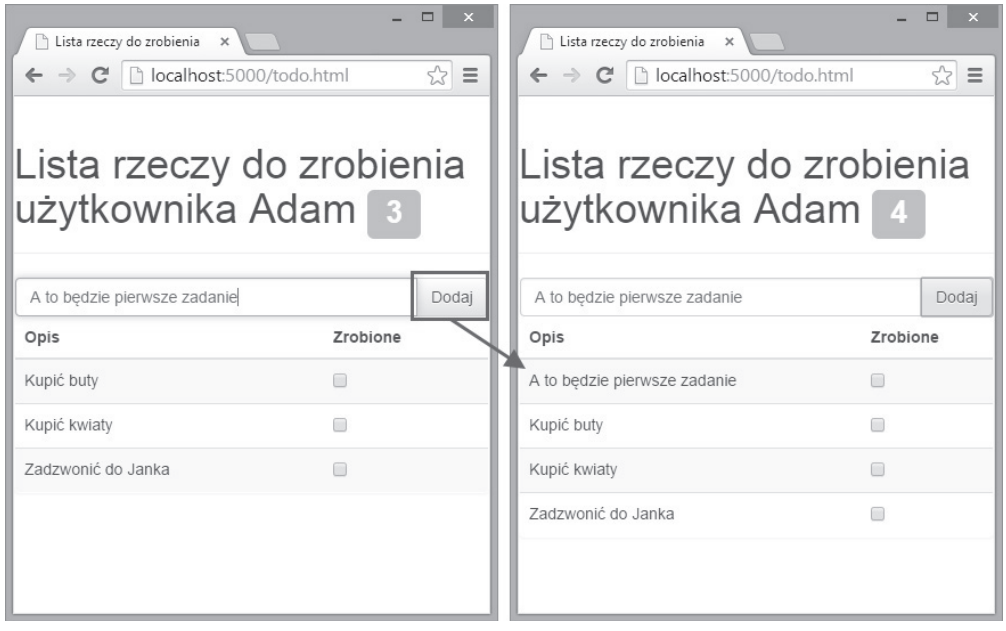
W rozdziale 14. zostanie omówiona funkcja *filtrowania* w AngularJS, która zapewnia elegancki sposób przygotowania danych w modelu do wyświetlania w widoku bez konieczności definiowania własnych funkcji. Wprawdzie nie ma nic złego w definiowaniu funkcji, ale filtry są z reguły rozwiązaniami ogólnego 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: "Zadzwonić do Janka", done: false }],
    };
```

```
    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 zwrotną 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*

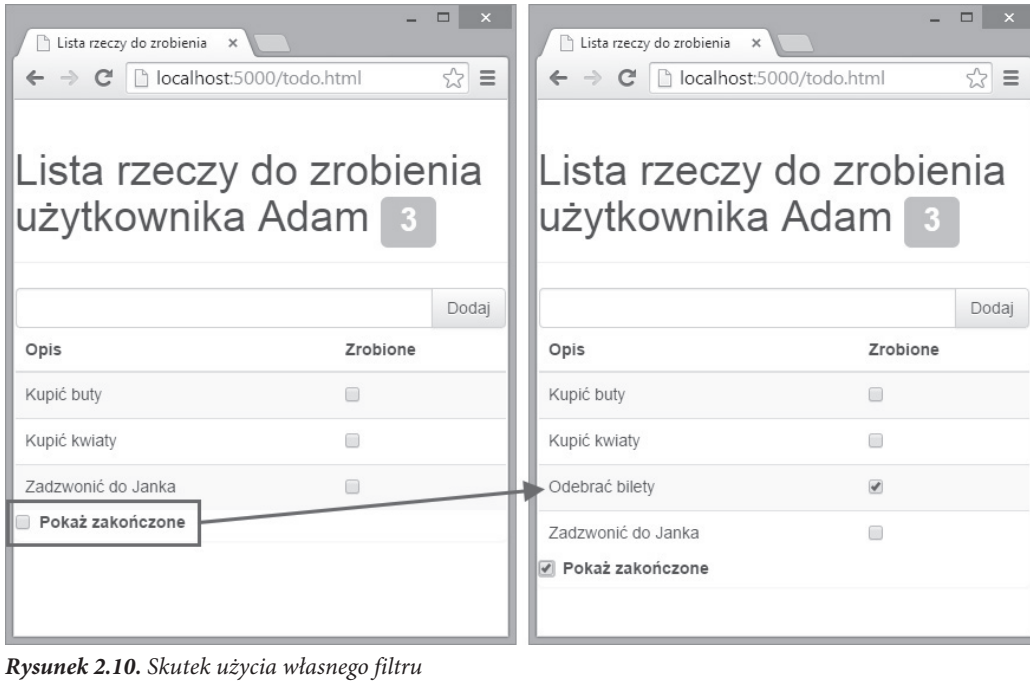
```
...
<div class="panel">
    <div class="input-group">
        <input class="form-control" ng-model="actionText" />
        <span class="input-group-btn">
            <button class="btn btn-default"
                    ng-click="addNewItem(actionText)">Dodaj</button>
        </span>
    </div>

    <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 filtru tworzonego z wykorzystaniem metody `filter()`, następnie dwukropek, a dalej nazwę właściwości modelu, która ma być przekazana funkcji filtru. 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 filtru*

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 bibliotece 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*:

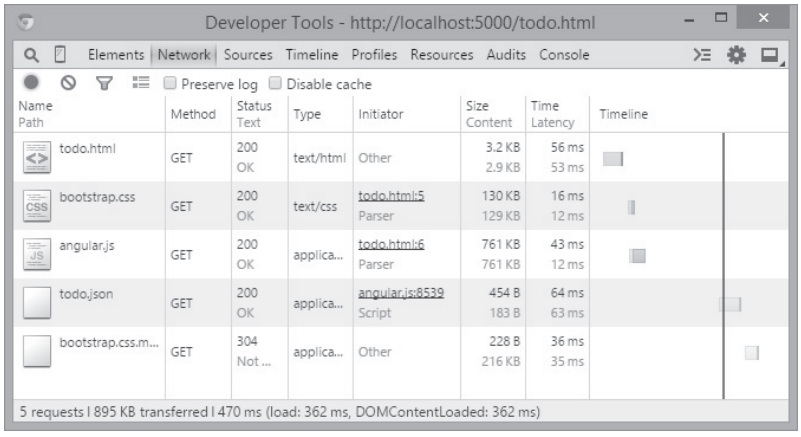
```
...
$http.get("todo.json").success(function (data) {
    model.items = data;
});
...
```

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.



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

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

T

U

W

Z

Ż

OBOWIĄZKOWA LEKTURA PROGRAMISTY ANGULARJS!

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 JavaScripcie
- 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 najnowsze promocje:
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

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ

KOD KORZYŚCI

ISBN 978-83-283-0197-9

cena: 99,00 zł