



Technologia i rozwiązania

AngularJS

Tworzenie aplikacji webowych Receptury

Wykorzystaj potencjał AngularJS!



Matt Frisbie

[PACKT] open source*
PUBLISHING community experience distilled

Tytuł oryginału: AngularJS Web Application Development Cookbook

Tłumaczenie: Radosław Meryk

ISBN: 978-83-283-2148-9

Copyright © Packt Publishing 2014. First published in the English language under the title „AngularJS Web Application Development Cookbook – (9781783283354)”.

Polish edition copyright © 2016 by Helion SA.
All rights reserved.

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

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

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

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

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

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

Printed in Poland.

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

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

Spis treści

O autorze	9
O recenzentach	10
Przedmowa	11
Rozdział 1. Maksymalne wykorzystanie możliwości AngularJS	17
Wprowadzenie	18
Budowanie prostych dyrektyw na poziomie elementu	18
Omówienie spektrum dyrektyw	20
Manipulowanie modelem DOM	25
Dyrektywy łączące	27
Interfejs z dyrektywą z wykorzystaniem odizolowanego zakresu	30
Interakcje pomiędzy zagnieżdżonymi dyrektywami	34
Opcjonalne kontrolery zagnieżdżonych dyrektyw	36
Dziedziczenie zakresu dyrektywy	37
Szablony dyrektyw	39
Odizolowany zakres	42
Transkluzje dyrektyw	43
Dyrektywy rekurencyjne	46
Rozdział 2. Rozszerzenie zestawu narzędzi o filtry i typy usług	53
Wprowadzenie	54
Wykorzystywanie filtrów uppercase i lowercase	54
Używanie filtrów number i currency	56
Używanie filtra date	59
Debugowanie z wykorzystaniem filtra json	61
Używanie filtrów danych na zewnątrz szablonu	63
Wykorzystanie wbudowanych filtrów wyszukiwania	65
Tworzenie łańcuchów filtrów	67
Tworzenie niestandardowych filtrów danych	69
Tworzenie niestandardowych filtrów wyszukiwania	72
Filtrowanie z wykorzystaniem niestandardowych komparatorów	74
Budowanie filtra wyszukiwania od podstaw	76
Budowanie niestandardowego filtra wyszukiwania od podstaw	79
Używanie wartości i stałych usług	81

Używanie fabryk usług	83
Korzystanie z usług	85
Korzystanie z dostawców usług	86
Korzystanie z dekoratorów usług	88
Rozdział 3. Animacje w AngularJS	91
Wprowadzenie	91
Tworzenie prostych animacji typu fade in i fade out	92
Replikowanie metod slideUp() i slideDown() biblioteki jQuery	96
Tworzenie animacji wejściowych z wykorzystaniem ngIf	99
Tworzenie animacji leave i concurrent z wykorzystaniem ngView	105
Tworzenie animacji move za pomocą ngRepeat	112
Tworzenie animacji addClass za pomocą ngShow	121
Tworzenie animacji removeClass za pomocą ngClass	126
Tworzenie wsadowych animacji stagger	131
Rozdział 4. Kreowanie i organizowanie aplikacji	135
Wprowadzenie	135
Ręczne ładowanie aplikacji	135
Bezpieczne używanie metody \$apply	138
Organizacja pliku aplikacji i modułów	143
Ukrywanie frameworka AngularJS przed użytkownikiem	146
Zarządzanie szablonami aplikacji	148
Składnia Controller as	152
Rozdział 5. Praca z zakresami i modelem	155
Wprowadzenie	155
Konfigurowanie zdarzeń frameworka AngularJS i korzystanie z nich	156
Zarządzanie dziedziczeniem obiektu \$scope	159
Praca z formatkami frameworka AngularJS	170
Korzystanie z elementów <select> i dyrektywy ngOptions	177
Budowanie magistrali zdarzeń	184
Rozdział 6. Testowanie w AngularJS	191
Wprowadzenie	191
Konfigurowanie i uruchamianie środowiska testowego w programach Yeoman i Grunt	192
Jak działa Protractor?	195
Włączanie testów E2E i Protractor w systemie Grunt	196
Pisanie prostych testów jednostkowych	200
Pisanie prostych testów E2E	206
Konfigurowanie prostej makiety serwera backend	211
Pisanie testów DAMP	214
Używanie wzorca testów Obiekt strony	216
Rozdział 7. Szybki AngularJS	223
Wprowadzenie	223
Miny-pułapki frameworka AngularJS	224
Tworzenie uniwersalnego wywołania zwrotnego dla obserwatorów	226
Inspekcja obserwatorów aplikacji	227

Skuteczne stosowanie typów \$watch i zarządzanie nimi	229
Optymalizowanie aplikacji z wykorzystaniem obserwatorów referencji	231
Optymalizowanie aplikacji z wykorzystaniem obserwatorów równości	234
Optymalizowanie aplikacji z wykorzystaniem obiektu \$watchCollection	236
Optymalizowanie aplikacji poprzez wyrejestrowywanie obserwatorów	238
Optymalizowanie wyrażań obserwatorów z wiązaniem szablonów	239
Optymalizowanie aplikacji z wykorzystaniem fazy kompilacji w ng-repeat	241
Optymalizowanie aplikacji z wykorzystaniem konstrukcji track by w ng-repeat	243
Przycinanie obserwowanych modeli	245
Rozdział 8. Obietnice	247
Wprowadzenie	247
Implementacja prostej obietnicy	248
Łańcuchy obietnic i handlerów obietnic	254
Implementacja powiadomień dla obietnic	259
Implementacja barier obietnic z wykorzystaniem wywołania \$q.all()	262
Tworzenie wrapperów obietnic za pomocą wywołania \$q.when()	264
Korzystanie z obietnic za pośrednictwem obiektu \$http	266
Używanie obietnic z usługą \$resource	269
Korzystanie z obietnic wraz z biblioteką Restangular	270
Włączanie obietnic do natywnych resolverów ścieżek	272
Implementacja zagnieżdżonych resolverów ui-router	274
Rozdział 9. Co nowego w AngularJS 1.3?	279
Wprowadzenie	279
Korzystanie z mechanizmów HTML5 do wprowadzania danych typu datetime	280
Łączenie obserwatorów z wykorzystaniem kolekcji \$watchGroup	281
Sprawdzanie poprawności z wykorzystaniem dyrektywy ng-strict-di	283
Zarządzanie wejściem modelu z wykorzystaniem dyrektywy ngModelOptions	284
Wykorzystywanie stanów \$touched i \$submitted	289
Porządkowanie komunikatów o błędach formularzy z wykorzystaniem ngMessages	291
Przycinanie listy obserwatorów z wykorzystaniem leniwego wiązania	294
Tworzenie niestandardowych walidatorów formularzy i korzystanie z nich	298
Rozdział 10. Sztuczki frameworka AngularJS	303
Wprowadzenie	303
Manipulowanie aplikacją z poziomu konsoli	304
Stosowanie zasady DRY podczas pisania kontrolerów	307
Korzystanie z dyrektywy ng-bind zamiast ng-cloak	309
Komentarze w plikach JSON	310
Tworzenie niestandardowych komentarzy AngularJS	312
Bezpieczne odwotywanie się do głębokich właściwości za pomocą obiektu \$parse	315
Zapobieganie nadmiarowemu parsowaniu	318
Skorowidz	323

Praca z zakresami i modelem

W tym rozdziale omówimy następujące receptury:

- Konfigurowanie zdarzeń frameworka AngularJS i korzystanie z nich.
- Zarządzanie dziedziczeniem obiektu `$scope`.
- Praca z formatkami frameworka AngularJS.
- Korzystanie z elementów `<select>` i dyrektywy `ngOptions`.
- Budowanie magistrali zdarzeń.

Wprowadzenie

Framework AngularJS dostarcza konstrukcji pozwalających na zarządzanie modyfikowaniem danych w całej aplikacji, głównie wykorzystując architekturę modyfikacji modelu. Mechanizmy wiązania danych frameworka AngularJS dają użytkownikom możliwość tworzenia rozbudowanych narzędzi na bazie tej architektury, jak również tworzenia kanałów komunikacji, które pozwalają skutecznie dotrzeć do wnętrza aplikacji.

Konfigurowanie zdarzeń frameworka AngularJS i korzystanie z nich

AngularJS oferuje potężną infrastrukturę zdarzeń, która daje możliwość zarządzania aplikacją w scenariuszach, w których wiązanie danych może być nieodpowiednie albo niepraktyczne. Nawet przy rygorystycznie zorganizowanej topologii aplikacji we frameworku AngularJS istnieje wiele zastosowań dla zdarzeń.

Jak to zrobić?

Zdarzenia AngularJS są identyfikowane przez ciągi znaków i dostarczają „ładunku”, który może przyjąć postać obiektu, funkcji albo prymitywu. Zdarzenie może być dostarczone za pośrednictwem zakresu nadrzędnego, który wywołuje `$scope.$broadcast()`, zakresu potomnego (albo tego samego zakresu), który wywołuje `$scope.$emit()`.

Metody `$scope.$on()` można użyć w dowolnym miejscu, gdzie można użyć obiektu zakresu, tak jak pokazano poniżej:

(app.js)

```
angular.module('myApp', [])
.controller('Ctrl', function($scope, $log) {
  $scope.$on('myEvent', function(event, data) {
    $log.log(event.name + ' obserwowane z ładunkiem ', data);
  });
});
```

Rozgłaszanie zdarzeń

Metoda `$scope.$broadcast()` wywołuje zdarzenie wewnątrz siebie oraz we wszystkich zakresach potomnych. W wersji 1.2.7 frameworka AngularJS wprowadzono optymalizację dla wywołania `$scope.$broadcast()`, ale ponieważ to działanie powoduje schodzenie w dół przez hierarchię zakresu w celu dotarcia do nasłuchujących zakresów potomnych, to w przypadku nadużywania tego mechanizmu istnieje możliwość wprowadzenia problemów wydajności. Rozgłaszanie można zaimplementować w następujący sposób:

(app.js)

```
angular.module('myApp', [])
.directive('myListener', function($log) {
  return {
    restrict: 'E',
    // każda dyrektywa powinna mieć swój własny zakres
    scope: true,
    link: function(scope, el, attrs) {
```



```

    // metoda do generowania zdarzeń
    scope.sendDown = function() {
        scope.$broadcast('mojeZdarzenie', {origin: attrs.local});
    };
    // metoda nasłuchiwania zdarzeń
    $scope.$on('mojeZdarzenie', function(event, data) {
        $log.log(
            event.name +
            ' obserwowane w ' +
            attrs.local +
            ', pochodzące z ' +
            data.origin
        );
    });
}
});

```

(*index.html*)

```

<div ng-app="myApp">
  <my-listener local="zewnątrzny">
    <button ng-click="sendDown()">Prześlij w dół</button>
  <my-listener local="środkowy">
    <my-listener local="pierwszy wewnętrzny"></my-listener>
    <my-listener local="drugi wewnętrzny"></my-listener>
  </my-listener>
</my-listener>
</div>

```

W tej konfiguracji kliknięcie przycisku *Prześlij w dół* spowoduje, że w konsoli przeglądarki wyświetlą się następujące komunikaty:

```

mojeZdarzenie obserwowane w zewnątrzny, pochodzące z zewnątrzny
mojeZdarzenie obserwowane w środkowy, pochodzące z zewnątrzny
mojeZdarzenie obserwowane w pierwszy wewnętrzny, pochodzące z zewnątrzny
mojeZdarzenie obserwowane w drugi wewnętrzny, pochodzące z zewnątrzny

```

JSFiddle: <http://jsfiddle.net/msfrisbie/dn0zjep9/>

Emitowanie zdarzenia

Jak można oczekiwać, metoda `$scope.$emit()` wykonuje operację odwrotną do `$scope.$broadcast()`. Wywołuje wszystkich słuchaczy zdarzenia, którzy istnieją w obrębie tego samego zakresu albo dowolnego zakresu nadrzędnego wzdłuż łańcucha prototypu w górę, aż do zakresu `$rootScope`. Można to zaimplementować w następujący sposób:

```

(app.js)

angular.module('myApp', [])
.directive('myListener', function($log) {
  return {
    restrict: 'E',
    // każda dyrektywa powinna mieć swój własny zakres
    scope: true,
    link: function(scope, el, attrs) {
      // metoda do generowania zdarzeń
      scope.sendUp = function() {
        scope.$emit('mojeZdarzenie', {origin: attrs.local});
      };
      // metoda nasłuchiwania zdarzeń
      scope.$on('mojeZdarzenie', function(event, data) {
        $log.log(
          event.name +
            ' obserwowane w ' +
            attrs.local +
            ', pochodzące z ' +
            data.origin
        );
      });
    }
  };
});

```

(index.html)

```

<div ng-app="myApp">
  <my-listener local="zewnątrzny">
    <my-listener local="środkowy">
      <my-listener local="pierwszy wewnętrzny">
        <button ng-click="sendUp()">
          Prześlij pierwszy w górę
        </button>
      </my-listener>
    <my-listener local="drugi wewnętrzny">
      <button ng-click="sendUp()">
        Prześlij drugi w górę
      </button>
    </my-listener>
  </my-listener>
</div>

```

W tym przykładzie kliknięcie przycisku *Prześlij pierwszy w górę* spowoduje wyświetlenie w konsoli przeglądarki następujących komunikatów:

mojeZdarzenie obserwowane w pierwszy wewnętrzny, pochodzące z pierwszy wewnętrzny
 mojeZdarzenie obserwowane w środkowy, pochodzące z pierwszy wewnętrzny
 mojeZdarzenie obserwowane w zewnętrzny, pochodzące z pierwszy wewnętrzny

Kliknięcie przycisku *Prześlij drugi w górę* spowoduje wyświetlenie w konsoli przeglądarki następujących komunikatów:

mojeZdarzenie obserwowane w drugi wewnętrzny, pochodzące z drugi wewnętrzny
 mojeZdarzenie obserwowane w środkowy, pochodzące z drugi wewnętrzny
 mojeZdarzenie obserwowane w zewnętrzny, pochodzące z drugi wewnętrzny

JSFiddle: <http://jsfiddle.net/msfrisbie/a344o7vo/>

Anulowanie rejestracji słuchacza zdarzeń

Po utworzeniu procesu słuchacza zdarzeń, na zasadzie podobnej do działania metody `$scope.$watch()`, proces słuchacza będzie istniał przez cały czas życia obiektu zakresu, do którego został dołączony. Metoda `$scope.$on()` zwraca funkcję anulowania rejestracji, która musi być przechwycona w momencie deklaracji. Wywołanie tej funkcji anulowania rejestracji zapobiega ocenianiu funkcji wywołania zwrotnego dla tego zdarzenia. Można to przełączyć za pomocą wzorca `setup/teardown` w następujący sposób:

(*app.js*)

```
angular.module('myApp', [])
  .controller('Ctrl', function($scope, $log) {
    $scope.setup = function() {
      $scope.teardown = $scope.$on('mojeZdarzenie', function(event, data) {
        $log.log(event.name + ' obserwowane z ładunkiem ', data);
      });
    };
  });
```

Wywołanie `$scope.setup()` zainicjuje wiązanie zdarzenia, natomiast wywołanie `$scope.teardown()` spowoduje zniszczenie tego wiązania.

Zarządzanie dziedziczeniem obiektu \$scope

Zakresy w frameworku AngularJS podlegają tym samym regułom dziedziczenia prototypowego, jak zwykle obiekty JavaScript. Przy odpowiednim zarządzaniu mogą one być bardzo skutecznie używane w aplikacji. Istnieją jednak pewne zagrożenia, z których należy zdawać sobie sprawę. Niebezpieczeństw można uniknąć dzięki przestrzeganiu najlepszych praktyk.

Przygotuj się

Przypuśćmy, że aplikacja zawiera następujący kod:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function() {})

(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl" ng-init="data=123">
    <input ng-model="data" />
    <div ng-controller="Ctrl">
      <input ng-model="data" />
    </div>
    <div ng-controller="Ctrl">
      <input ng-model="data" />
    </div>
  </div>
</div>
```

Jak to zrobić?

W zaprezentowanej konfiguracji egzemplarze obiektu `$scope` wewnątrz zagnieżdżonych egzemplarzy `Ctrl` będą dziedziczyć na poziomie prototypów po nadrzędnym obiekcie `$scope` egzemplarza `Ctrl`. W momencie ładowania strony wszystkie trzy pola tekstowe będą wypełnione ciągiem `123`, a kiedy zmienimy wartość `<input>` nadrzędnego kontrolera `Ctrl`, oba pola tekstowe związane z potomnymi egzemplarzami `$scope` zaktualizują się po kolei, ponieważ wszystkie trzy są powiązane z tym samym obiektem. Jednak kiedy zmienimy wartości dowolnego pola tekstowego powiązanego z potomnym obiektem `$scope`, pozostałe pola tekstowe nie odzwierciedlą tej zmiany, a wiązanie danych z tego pola tekstowego będzie zakłócone do chwili ponownego załadowania aplikacji.

Aby to naprawić, należy po prostu dodać obiekt, który jest zagnieżdżony w dowolnym pierwotnym typie w zakresie. Można tego dokonać w następujący sposób:

```
(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl" ng-init="data.value=123">
    <input ng-model="data.value" />
    <div ng-controller="Ctrl">
      <input ng-model="data.value" />
    </div>
  </div>
```

```

    <div ng-controller="Ctrl">
      <input ng-model="data.value" />
    </div>
  </div>
</div>

```

JSFiddle: <http://jsfiddle.net/msfrisbie/penaakxy/>

Teraz można zmodyfikować dowolne spośród trzech pól tekstowych, a zmiana będzie odzwierciedlona na pozostałych dwóch. Wszystkie trzy są nadal powiązane z tym samym obiektem \$scope w obiekcie \$scope nadrzędnego kontrolera Ctrl.

Warto stosować zasadę, aby w przypadku polegania na dziedziczeniu po obiekcie \$scope w dowolnej formie zawsze utrzymywać jeden poziom zagnieżdżenia obiektu dla wszystkich typów (zwłaszcza typów prymitywnych). Kolokwialnie określa się tę własność jako „zawsze używaj kropki”.

Jak to działa?

Kiedy wartość właściwości \$scope zostanie zaktualizowana na podstawie pola wejściowego, powoduje to przypisanie wartości do właściwości \$scope, z którą to pole wejściowe jest powiązane. Tak jak w przypadku dziedziczenia prototypowego, przypisanie do właściwości obiektu spowoduje podążanie za łańcuchem prototypu aż do oryginalnego egzemplarza, ale przypisanie do prymitywu spowoduje utworzenie nowego egzemplarza prymitywu w lokalnej właściwości \$scope. W poprzednim przykładzie przed dodaniem poprawki .value nowy, lokalny egzemplarz był odłączony od odziedziczonej wartości, co skutkowało podwójnymi wartościami właściwości \$scope.

Co dalej?

Poniższe dwa przykłady są uważane za złe praktyki (z oczywistych powodów). Znacznie łatwiej jest utrzymywanie przynajmniej jednego poziomu zagnieżdżenia obiektów dla dowolnych danych, dla których jest potrzebne dziedziczenie w dół drzewa \$scope aplikacji.

Istnieje możliwość ponownego ustanowienia tego dziedziczenia poprzez usunięcie prymitywnej właściwości z lokalnego obiektu \$scope:

```

(app.js)

angular.module('myApp', [])
.controller('outerCtrl', function($scope) {
  $scope.data = 123;
})

```

```
.controller('innerCtrl', function($scope) {
    $scope.reattach = function() {
        delete($scope.data);
    };
});
```

(*index.html*)

```
<div ng-app="myApp">
  <div ng-controller="outerCtrl">
    <input ng-model="data" />
    <div ng-controller="innerCtrl">
      <input ng-model="data" />
    </div>
    <div ng-controller="innerCtrl">
      <input ng-model="data" />
      <button ng-click="reattach()">Ponownie dołącz</button>
    </div>
  </div>
</div>
```

JSFiddle: <http://jsfiddle.net/msfrisbie/ghsa3nym/>

Możliwy jest również bezpośredni dostęp do nadrzędnego obiektu \$scope za pomocą notacji \$scope.\$parent. W tym przypadku dziedziczenie jest całkowicie ignorowane. Można to zrobić w następujący sposób:

(*app.js*)

```
angular.module('myApp', [])
.controller('Ctrl', function() {});
```

(*index.html*)

```
<div ng-app="myApp">
  <div ng-controller="Ctrl" ng-init="data=123">
    <input ng-model="data" />
    <div ng-controller="Ctrl">
      <input ng-model="$parent.data" />
    </div>
    <div ng-controller="Ctrl">
      <input ng-model="$parent.data" />
    </div>
  </div>
</div>
```

Kłopotliwe dyrektywy wbudowane

W poprzednich przykładach jawnie zademonstrowano zagnieżdżone zakresy, które dziedziczą na poziomie prototypów po nadrzędnym obiekcie `$scope`. W rzeczywistej aplikacji prawdopodobnie byłoby to bardzo łatwe do wykrycia i zdebugowania. Jednak framework AngularJS dostarcza szereg dyrektyw wbudowanych, które niejawnie tworzą swoje własne zakresy i jeżeli prototypowe dziedziczenie zakresów nie jest stosowane rozważnie, może to spowodować problemy. Istnieje sześć wbudowanych dyrektyw, które tworzą swoje własne zakresy: `ngController`, `ngInclude`, `ngView`, `ngRepeat`, `ngIf` oraz `ngSwitch`.

W poniższych przykładach pokazano wykorzystanie identyfikatora `$scope $id` w szablonie w celu zademonstrowania tworzenia nowego zakresu.

ngController

Użycie dyrektywy `ngController` powinno być oczywiste, ponieważ logika kontrolera polega na dołączaniu funkcji i danych do nowego zakresu potomnego utworzonego za pomocą dyrektywy `ngController`.

ngInclude

Niezależnie od włączanej zawartości HTML, dyrektywa `ng-include` opakuje ją wewnątrz nowego zakresu. Ponieważ dyrektywa `ng-include` jest standardowo używana w celu wstawiania komponentów monolitycznej aplikacji, które nie zależą od ich otoczenia, jest mniejsze prawdopodobieństwo, że stosowanie tej dyrektywy spowoduje problemy związane z dziedziczeniem obiektu `$scope`.

Poniżej zamieszczono nieprawidłowe rozwiązanie:

(app.js)

```
angular.module('myApp', [])
.controller('Ctrl', function($scope) {
    $scope.data = 123;
});
```

(index.html)

```
<div ng-app="myApp">
  <div ng-controller="Ctrl">
    Scope id: {{ $id }}
    <input ng-model="data " />
    <ng-include src="'innerTemplate.html'"></ng-include>
  </div>

  <script type="text/ng-template" id="innerTemplate.html">
    <div>
```

```

        Scope id: {{ $id }}
        <input ng-model="data " />
    </div>
</script>
</div>

```

Nowy zakres wewnątrz skompilowanej dyrektywy `ng-include` dziedziczy po obiekcie `$scope` kontrolera, ale wiązanie do jego prymitywnej wartości powoduje taki sam problem.

Oto prawidłowe rozwiązanie:

(*app.js*)

```

angular.module('myApp', [])
.controller('Ctrl', function($scope) {
    $scope.data = {
        val: 123
    };
});

```

(*index.html*)

```

<div ng-app="myApp">
    <div ng-controller="Ctrl">
        Scope id: {{ $id }}
        <input ng-model="data.val" />
        <ng-include src="'innerTemplate.html'"></ng-include>
    </div>

    <script type="text/ng-template" id="innerTemplate.html">
        <div>
            Scope id: {{ $id }}
            <input ng-model="data.val" />
        </div>
    </script>
</div>

```

JSFiddle: <http://jsfiddle.net/msfrisbie/c8nLk676/>

ngView

Z perspektywy dziedziczenia prototypowanego dyrektywa `ng-view` działa identycznie, jak dyrektywa `ng-include`. Do wstawionego skompilowanego szablonu dostarczany jest nowy potomny obiekt `$scope`, a prawidłowe dziedziczenie po nadrzędnym zakresie `$scope` można osiągnąć dokładnie w taki sam sposób.

ngRepeat

Dyrektywa ngRepeat sprawia najwięcej problemów, jeśli chodzi o nieprawidłowe zarządzanie dziedziczeniem obiektu `$scope`. Każdy element stworzony przez repeater uzyskuje własny zakres, a modyfikacje zakresów potomnych (takie jak edytowanie inline danych na liście) nie mają wpływu na oryginalny obiekt, jeżeli jest on powiązany z typami prymitywnymi.

Poniżej zamieszczono nieprawidłowe rozwiązanie:

(app.js)

```
angular.module('myApp', [])
.controller('Ctrl', function($scope) {
  $scope.names = [
    'Alshon Jeffrey',
    'Brandon Marshall',
    'Matt Forte',
    'Martellus Bennett',
    'Jay Cutler'
  ];
});
```

(index.html)

```
<div ng-app="myApp">
  <div ng-controller="Ctrl">
    Scope id: {{ $id }}
    <pre>{{ names | json }}</pre>
    <div ng-repeat="name in names">
      Scope id: {{ $id }}
      <input ng-model="name" />
    </div>
  </div>
</div>
```

Zgodnie z tym, co napisano wcześniej, zmiana wartości w polach wejściowych służy tylko zmodyfikowaniu egzemplarzy prymitywu w zakresie potomnym, a nie w oryginalnym obiekcie. Jednym ze sposobów, aby to naprawić, jest restrukturyzacja obiektu danych w taki sposób, aby zamiast iterowania po typach pierwotnych iterował po obiektach opakowujących te typy pierwotne.

Oto prawidłowe rozwiązanie:

(app.js)

```
angular.module('myApp', [])
.controller('Ctrl', function($scope) {
  $scope.players = [
    {name: 'Alshon Jeffrey' },
```

```

    { name: 'Brandon Marshall' },
    { name: 'Matt Forte' },
    { name: 'Martellus Bennett' },
    { name: 'Jay Cutler' }
  ];
});

```

(*index.html*)

```

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    Scope id: {{ $id }}
    <pre>{{ players | json }}</pre>
    <div ng-repeat="player in players">
      Scope id: {{ $id }}
      <input ng-model="player.name" />
    </div>
  </div>
</div>

```

JSFiddle: <http://jsfiddle.net/msfrisbie/zesj1gb6/>

W tym rozwiązaniu oryginalna tablica jest modyfikowana właściwie i wszystko jest w najlepszym porządku. Czasami jednak restrukturyzowanie obiektu w aplikacji nie jest wykonalne. W takim przypadku modyfikowanie tablicy łańcuchów znaków na tablicę obiektów wydaje się dziwnym obejściem. W idealnym przypadku byłoby lepiej, gdyby było można iterować po tablicy łańcuchów znaków bez jej uprzedniego modyfikowania. Jest to możliwe dzięki skorzystaniu z konstrukcji `track by` w wyrażeniu dyrektywy `ng-repeat`.

Oto prawidłowe rozwiązanie:

(*app.js*)

```

angular.module('myApp', [])
.controller('Ctrl', function($scope) {
  $scope.players = [
    'Alshon Jeffrey',
    'Brandon Marshall',
    'Matt Forte',
    'Martellus Bennett',
    'Jay Cutler'
  ];
});

```

(*index.html*)

```

<div ng-app="myApp">

```

```

<div ng-controller="Ctrl">
  Scope id: {{ $id }}
  <pre>{{ players | json }}</pre>
  <div ng-repeat="player in players track by $index">
    Scope id: {{ $id }}
    <input ng-model="players[$index]" />
  </div>
</div>
</div>

```

JSFiddle: <http://jsfiddle.net/msfrisbie/ovas398h/>

Teraz pomimo tego, że repeater iteruje po elementach tablicy `players`, to ze względu na to, że potomne obiekty `$scope` stworzone dla każdego elementu w dalszym ciągu będą dziedziczyć prototyp tablicy `players`, repeater powiąże właściwy element w tablicy za pomocą obiektu `$index`.

Ponieważ typy pierwotne w JavaScript są niemutowalne, modyfikacja elementu typu pierwotnego w tablicy spowoduje jego całkowite zastąpienie. Gdy następuje ta zamiana, to ze względu na to, że przy zwykłym wykorzystaniu dyrektywy `ng-repeat` elementy tablicy są identyfikowane według ich wartości tekstowej, z perspektywy dyrektywy `ng-repeat` sytuacja wygląda tak, jakby dodano nowy element. W związku z tym cała zawartość tablicy zostanie wyrenderowana od nowa — ze względów związanych z wygodą użytkowania i wydajnością ta funkcjonalność jest oczywiście niepożądana. Problem rozwiązuje zastosowanie klauzuli `track by $index` w wyrażeniu dyrektywy `ng-repeat`. W tym przypadku elementy tablicy są identyfikowane według indeksu, a nie wartości tekstowej. To zapobiega ciągłemu ponownemu renderowaniu.

ngIf

Ponieważ dyrektywa `ng-if` niszczy zawartość modelu DOM zagnieżdżoną wewnątrz niej za każdym razem, gdy wyrażenie dyrektywy przyjmie wartość `false`, to przy każdej kompilacji wewnętrznej zawartości odziedziczy ona nadrzędny obiekt `$scope`. Jeżeli cokolwiek wewnątrz dyrektywy `ng-if` nieprawidłowo odziedziczy po nadrzędnym obiekcie `$scope`, wtedy przy każdej ponownej kompilacji dane `$scope` potomka będą usunięte.

Poniżej zamieszczono nieprawidłowe rozwiązanie:

(*app.js*)

```

angular.module('myApp', [])
.controller('Ctrl', function($scope) {
  $scope.data = 123;
  $scope.show = false;
});

```

(*index.html*)

```
<div ng-app="myApp">
  <div ng-controller="Ctrl">
    Scope id: {{ $id }}
    <input ng-model="data " />
    <input type="checkbox" ng-model="show" />
    <div ng-if="show">
      Scope id: {{ $id }}
      <input ng-model="data " />
    </div>
  </div>
</div>
```

Za każdym razem, gdy użytkownik zaznaczy pole wyboru, nowo utworzony obiekt `$scope` odziedziczy wartości po nadrzędnym obiekcie `$scope` i wyczyści istniejące dane. To jest oczywiście niepożądane w wielu scenariuszach. Zamiast tego problem rozwiązuje proste wykorzystanie jednego pośredniego poziomu obiektu.

Oto prawidłowe rozwiązanie:

(*app.js*)

```
angular.module('myApp', [])
.controller('Ctrl', function($scope) {
  $scope.data = {
    val: 123
  };
  $scope.show = false;
});
```

(*index.html*)

```
<div ng-app="myApp">
  <div ng-controller="Ctrl">
    Scope id: {{ $id }}
    <input ng-model="data.val" />
    <input type="checkbox" ng-model="show" />
    <div ng-if="show">
      Scope id: {{ $id }}
      <input ng-model="data.val" />
    </div>
  </div>
</div>
```

JSFiddle: <http://jsfiddle.net/msfrisbie/hq7r5frm/>

ngSwitch

Działanie dyrektywy `ngSwitch` pod wieloma względami można porównać do połączonego działania kilku dyrektyw `ngIf`. Jeżeli cokolwiek wewnątrz aktywnego obiektu `$scope` dyrektywy `ng-if` niepoprawnie odziedziczy od obiektu `$scope` rodzica, to przy każdej ponownej kompilacji oraz aktualizacji obserwowanej wartości przełącznika dane potomnego obiektu `$scope` będą wyczyszczone.

Poniżej zamieszczono nieprawidłowe rozwiązanie:

(app.js)

```
angular.module('myApp', [])
.controller('Ctrl', function($scope) {
    $scope.data = 123;
});
```

(index.html)

```
<div ng-app="myApp">
  <div ng-controller="Ctrl">
    Scope id: {{ $id }}
    <input ng-model="data " />
    <div ng-switch on="data ">
      <div ng-switch-when="123">
        Scope id: {{ $id }}
        <input ng-model="data " />
      </div>
      <div ng-switch-default>
        Scope id: {{ $id }}
        Default
      </div>
    </div>
  </div>
</div>
```

W tym przykładzie, kiedy zewnętrzny znacznik `<input>` zostanie ustawiony na pasującą wartość 123, wtedy wewnętrzny znacznik `<input>` zagnieżdżony w dyrektywie `ng-switch` zgodnie z oczekiwaniami odziedziczy tę wartość. Jednakże w przypadku aktualizacji wewnętrznego pola wejściowego odziedziczona wartość nie zostanie zmodyfikowana, ponieważ łańcuch dziedziczenia prototypowego jest naruszony.

Oto prawidłowe rozwiązanie:

(app.js)

```
angular.module('myApp', [])
.controller('Ctrl', function($scope) {
    $scope.data = {
```

```

        val: 123
    };
});

(index.html)

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    Scope id: {{ $id }}
    <input ng-model="data.val" />
    <div ng-switch on="data.val">
      <div ng-switch-when="123">
        Scope id: {{ $id }}
        <input ng-model="data.val" />
      </div>
      <div ng-switch-default>
        Scope id: {{ $id }}
        Default
      </div>
    </div>
  </div>
</div>

```

JSFiddle: <http://jsfiddle.net/msfrisbie/8kh41wdm/>

Praca z formatkami frameworka AngularJS

AngularJS zapewnia ścisłą integrację z elementami formularzy HTML. Zawiera dyrektywy pozwalające na szybkie i łatwe budowanie animowanych i wyposażonych w style strony formularzy wraz z mechanizmami walidacji.

Jak to zrobić?

Formularze frameworka AngularJS definiuje się za pomocą znacznika `<form>`, który odpowiada natywnej dyrektywie frameworka AngularJS. Przykład jej użycia zaprezentowano w poniższym kodzie. Atrybut `novalidate` jest instrukcją dla przeglądarki do zignorowania wbudowanych mechanizmów walidacji formularzy:

```

<form novalidate>
  <!-- kontrolki wejściowe formularza -->
</form>

```

Kontrolki wejściowe HTML należy umieścić wewnątrz znaczników `<form>`. Każdy egzemplarz znacznika `<form>` tworzy obiekt `FormController`, który zarządza wszystkimi kontrolkami i zagnieżdżonymi formularzami. Na bazie tego mechanizmu jest zbudowana cała infrastruktura obsługi formularzy frameworka AngularJS.

Ponieważ przeglądarki nie pozwalają zagnieżdżać znaczników `form`, do zagnieżdżania formularzy należy stosować dyrektywę `ng-form`.

Co oferują formularze?

Przypuśćmy, że w aplikacji mamy kontroler zawierający formularz w następującej postaci:

```
<div ng-controller="Ctrl">
  <form novalidate name="myform">
    <input name="myinput" ng-model="formdata.myinput" />
  </form>
</div>
```

W przypadku aplikacji zawierającej kontroler w tej postaci do obiektu `$scope` kontrolera jest dostarczany konstruktor obiektu `FormController` za pomocą `$scope.myform`. Konstruktor ten zawiera wiele przydatnych atrybutów i funkcji. Dostęp do pojedynczych pozycji formularza dla każdego pola wejściowego można uzyskać za pośrednictwem potomnych obiektów `FormController` nadrzędnego obiektu `FormController`. Na przykład `$scope.myform.myinput` to obiekt `FormController` do wprowadzania tekstu.

Aby powiązania stanu i walidacji mogły działać, kontrolki wejściowe muszą być powiązane z dyrektywą `ng-model`.

Śledzenie stanu formularzy

Kontrolki wejściowe i formularze są dostarczane z własnymi sterownikami. Framework AngularJS śledzi stan zarówno indywidualnych kontrolki wejściowych, jak i całego formularza, używając dychotomii czysty-brudny (ang. *pristine-dirty*). Stan „czysty” (ang. *pristine*) odpowiada sytuacji, w której kontrolki wejściowe są ustawione na wartości domyślne, natomiast stan „brudny” (ang. *dirty*) odnosi się do dowolnej operacji modyfikacji danych modelu powiązanych z kontrolkami wejściowymi. Stan „czysty” całego formularza to wynik logicznej operacji AND na wszystkich stanach czystych kontrolki wejściowych albo wynik operacji NOR dla wszystkich stanów brudnych. Zgodnie z odwróconą definicją stan „brudny” całego formularza reprezentuje wynik operacji OR wszystkich stanów brudnych albo wynik operacji NAND wszystkich stanów czystych.

JSFiddle: <http://jsfiddle.net/msfrisbie/trjfdwvc/>

Wymienione stany mogą być użyte na kilka różnych sposobów.

Zarówno z elementami `<form>`, jak i `<input>` są związane klasy CSS `ng-pristine` i `ng-dirty`, które są do nich automatycznie stosowane na podstawie stanu, w jakim znajduje się formularz. Te klasy CSS mogą zostać użyte do nadania stylu kontrolkom wejściowym na podstawie ich stanu, tak jak pokazano poniżej:

```
form.ng-pristine {
}
input.ng-pristine {
}
form.ng-dirty {
}
input.ng-dirty {
}
```

Wszystkie egzemplarze klasy `FormController` oraz egzemplarze klasy `ngModelController` znajdujące się wewnątrz nich mają dostępne właściwości typu `Boolean` `$pristine` i `$dirty`. Można z nich skorzystać w logice biznesowej kontrolera albo do sterowania działaniami użytkownika wewnątrz formularza.

W przykładzie zamieszczonym poniżej wyświetla się komunikat **Wprowadź wartość** tak długo, aż wartość w polu wejściowym zostanie zmodyfikowana.

(*app.js*)

```
angular.module('myApp', [])
.controller('Ctrl', function($scope) {
    $scope.$watch('myform.myinput.$pristine', function(newval) {
        $scope.isPristine = newval;
    });
});
```

(*index.html*)

```
<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <form novalidate name="myform">
      <input name="myinput" ng-model="formdata.myinput" />
    </form>
    <div ng-show="isPristine">
      Wprowadź wartość
    </div>
  </div>
</div>
```

JSFiddle: <http://jsfiddle.net/msfrisbie/unxbyun2/>

Można także wykryć, czy kontrolka wejściowa jest w stanie czystym bezpośrednio w widoku, w czasie gdy obiekt formularza jest dołączany do zakresu.

(*index.html*)

```
<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <form novalidate name="myform">
      <input name="myinput" ng-model="formdata.myinput" />
      <div ng-show="myform.myinput.$pristine">
        Wprowadź wartość
      </div>
    </form>
  </div>
</div>
```

JSFiddle: <http://jsfiddle.net/msfrisbie/pr3L1e2b/>

Można również wymusić stan czysty lub brudny formularza lub kontrolki wejściowej, używając metod `$setDirty()` albo `$setPristine()`. Wykorzystanie tego sposobu nie uwzględnia aktualnych wartości kontrolki wejściowych w tym momencie. Jest to jedynie przesłonięcie wartości Boolean `$pristine` i `$dirty` oraz ustawienie odpowiadających im klas CSS `ng-pristine` lub `ng-dirty`. Wywołanie tych metod będzie propagowane do wszystkich formularzy nadrzędnych.

Walidowanie formularzy

Oprócz dychotomii czysty-brudny, dla formularzy frameworka AngularJS obowiązuje również dychotomia poprawny-niepoprawny. Do pól wejściowych w formularzach można przypisywać reguły walidacji, które muszą być spełnione, aby formularze były poprawne. Framework AngularJS korzystając z dychotomii poprawny-niepoprawny śledzi poprawność zarówno indywidualnych pól wejściowych, jak i całych formularzy. „Poprawny” odpowiada stanowi, w którym pola wejściowe spełniają wszystkie przypisane do nich wymagania walidacji, natomiast „niepoprawny” odnosi się do pola wejściowego, dla którego nie jest spełniona przynajmniej jedna reguła walidacji. Stan „poprawny” całego formularza to wynik wykonania logicznej operacji AND stanów „poprawny” wszystkich kontrolki wejściowych albo wynik operacji NOR stanów „niepoprawny” wszystkich kontrolki wejściowych. Zgodnie z odwróconą definicją stan „niepoprawny” całego formularza reprezentuje wynik logicznej operacji OR wszystkich stanów „niepoprawny” albo wynik operacji NAND wszystkich stanów „poprawny”.

JSFiddle: <http://jsfiddle.net/msfrisbie/ejpsrfgz/>

Elementy `<form>` i `<input>` mają klasy CSS `ng-valid` i `ng-invalid` podobne do klas `ng-pristine` i `ng-dirty`, które są ustawiane automatycznie na podstawie stanu, w jakim znajduje się formularz. Te klasy CSS można wykorzystać do nadawania kontrolkom wejściowym stylu na podstawie stanu. Oto przykład:

```
form.ng-valid {
}
input.ng-valid {
}
form.ng-invalid {
}
input.ng-invalid {
}
```

Dla wszystkich wewnętrznych egzemplarzy klas `FormController` i `ngModelController` dostępne są atrybuty typu Boolean `$valid` i `$invalid`. Można z nich skorzystać w logice biznesowej kontrolera albo do sterowania działaniami użytkownika wewnątrz formularza.

W poniższym przykładzie pokazano, jak wyświetlić komunikat **Pole wejściowe nie może być puste** w czasie, kiedy pole do wprowadzania danych jest puste:

(*app.js*)

```
angular.module('myApp', [])
.controller('Ctrl', function($scope) {
  $scope.$watch('myform.myinput.$invalid', function(newval) {
    $scope.isInvalid = newval;
  });
});
```

(*index.html*)

```
<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <form novalidate name="myform">
      <input name="myinput"
        ng-model="formdata.myinput"
        required />
    </form>
    <div ng-show="isInvalid">
      Pole wejściowe nie może być puste
    </div>
  </div>
</div>
```

JSFiddle: <http://jsfiddle.net/msfrisbie/40bdaey4/>

Alternatywnie, ze względu na to, że obiekt formularza jest dołączony do zakresu, możliwe jest wykrycie bezpośrednio w widoku, czy pole wejściowe jest prawidłowe.

(*app.js*)

```
angular.module('myApp', [])
.controller('Ctrl', function() {});
```

(*index.html*)

```
<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <form novalidate name="myform">
      <input name="myinput"
        ng-model="formdata.myinput"
        required />
      <div ng-show="myform.myinput.$pristine">
        Pole wejściowe nie może być puste
      </div>
    </form>
  </div>
</div>
```

JSFiddle: <http://jsfiddle.net/msfrisbie/bc2hn05p/>

Walidatory wbudowane i niestandardowe

W frameworku AngularJS istnieją następujące wbudowane podstawowe walidatory:

- e-mail,
- max,
- maxlength,
- min,
- minlength,
- number,
- pattern,
- required,
- url.

Chociaż są one przydatne i w większości nie wymagają wyjaśnień, to może się zdarzyć, że zajdzie potrzeba stworzenia niestandardowego walidatora. Aby to zrobić, należy zbudować dyrektywę, która będzie obserwować wartość modelu odpowiadającą temu polu wejściowemu, przeprowadzić dla niej analizę, a następnie ustawić poprawność tego pola, używając metody `$setValidity()`.

W ramach wydania 1.3 wprowadzono alternatywną metodę tworzenia niestandardowych walidatorów formularzy. Więcej informacji można znaleźć w recepturze *Tworzenie niestandardowych walidatorów formularzy i korzystanie z nich* w rozdziale 9., „Co nowego w AngularJS 1.3?”.

Poniżej pokazano sposób stworzenia niestandardowego walidatora, który sprawdza, czy w polu wejściowym wprowadzono liczbę pierwszą:

(*app.js*)

```
angular.module('myApp', [])
.directive('ensurePrime', function() {
  return {
    require: 'ngModel',
    link: function(scope, element, attrs, ctrl) {
      function isPrime(n) {
        if (n<2) {
          return false;
        }

        var m = Math.sqrt(n);

        for (var i=2; i<=m; i++) {
          if (n%i === 0) {
            return false;
          }
        }
        return true;
      }

      $scope.$watch(giganticObject, function() { ...
        if (isPrime(newval)) {
          ctrl.$setValidity('prime', true);
        }
        else {
          ctrl.$setValidity('prime', true);
        }
      });
    }
  };
});
```

(*index.html*)

```
<div ng-app="myApp">
  <form novalidate name="myform">
    <input type="number"
      ensure-prime name="myinput"
      ng-model="formdata.myinput"
      required />
```

```

</form>
<div ng-show="myform.myinput.$pristine">
  NAleży wprowadzić liczbę pierwszą
</div>
</div>

```

JSFiddle: <http://jsfiddle.net/msfrisbie/7mhqvgcp/>

Jak to działa?

Formularze AngularJS w celu sprawdzenia stanu formularzy i stanu walidacji sięgają do istniejącej architektury wiązania danych. Egzemplarze klasy `FormController` powiązane z formularzem oraz pola wejściowe znajdujące się wewnątrz tworzą bardzo przyjemny, modułowy sposób zarządzania przepływem danych wewnątrz formularza.

Korzystanie z elementów `<select>` i dyrektywy `ngOptions`

AngularJS dostarcza dyrektywę `ngOptions`, pozwalającą na wypełnianie w aplikacji elementów `<select>`. Chociaż na pierwszy rzut oka wydaje się to trywialne, dyrektywa `ngOptions` wykorzystuje skomplikowany obiekt `comprehension_expression`, który pozwala wypełnić rozwijaną listę na podstawie obiektów danych na kilka sposobów.

Przygotuj się

Załóżmy, że aplikacja zawiera następujący kod:

```

(app.js)
angular.module('myApp', [])
.controller('Ctrl', function($scope) {
  $scope.players = [
    {
      number: 17,
      name: 'Alshon',
      position: 'WR'
    },
    {
      number: 15,
      name: 'Brandon',
      position: 'WR'
    }
  ]
});

```

```

    },
    {
      number: 22,
      name: 'Matt',
      position: 'RB'
    },
    {
      number: 83,
      name: 'Martellus',
      position: 'TE'
    },
    {
      number: 6,
      name: 'Jay',
      position: 'QB'
    }
  ];

  $scope.team = {
    '3B': {
      number: 9,
      name: 'Brandon'
    },
    '2B': {
      number: 19,
      name: 'Marco'
    },
    '3B': {
      number: 48,
      name: 'Pablo'
    },
    'C': {
      number: 28,
      name: 'Buster'
    },
    'SS': {
      number: 35,
      name: 'Brandon'
    }
  };
});

```

Jak to zrobić?

Dyrektywa `ngOptions` pozwala na wypełnienie elementu `<select>` zarówno na podstawie tablicy, jak i atrybutów obiektu.

Wypełnianie na podstawie tablicy

Obiekt `comprehension_expression` pozwala zdefiniować sposób mapowania tablicy danych do zbioru tagów `<option>` wraz z tekstowym opisem i odpowiednimi wartościami. Łatwiejsza implementacja polega na zdefiniowaniu tylko ciągu etykiety. W tym przypadku aplikacja domyślnie ustawi wartość `<option>` na cały element tablicy, tak jak pokazano poniżej:

(*index.html*)

```
<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <!-- etykieta na wartość w tablicy -->
    <select ng-model="player"
      ng-options="p.name for p in players">
      </select>
    </div>
  </div>
```

Powyższy kod skompiluje się do postaci zamieszczonej poniżej (klasy CSS związane z formularzami zostały pominięte):

```
<select ng-model="player"
  ng-options="player.name for player in players">
  <option value="?" selected="selected"></option>
  <option value="0">Alshon</option>
  <option value="1">Brandon</option>
  <option value="2">Matt</option>
  <option value="3">Martellus</option>
  <option value="4">Jay</option>
</select>
```

JSFiddle: <http://jsfiddle.net/msfrisbie/vy62c575/>

W tym przypadku wartości poszczególnych opcji są indeksami tablicy odpowiadającymi poszczególnym elementom. Ponieważ model, do którego jest dołączona aplikacja, nie jest zainicjowany do żadnego z istniejących elementów, framework AngularJS tymczasowo wstawi wartość `null` na listę. W tym momencie pusta wartość będzie obciążona. W chwili dokonania wyboru model `player` zostanie przypisany do całego obiektu pod tym indeksem tablicy.

Jawne definiowanie wartości opcji

Jeżeli nie chcemy, aby wartość HTML `<option>` została przypisana do indeksu tablicy, możemy to przesłonić za pomocą klauzuli `track by`, tak jak pokazano poniżej:

(*index.html*)

```
<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <!-- etykieta na wartość w tablicy -->
    <select ng-model="player"
      ng-options="p.name for p in players track by p.number">
    </select>
  </div>
</div>
```

Po skompilowaniu uzyskamy następujący kod:

```
<select ng-model="player"
  ng-options="p.name for p in players track by p.number">
  <option value="" selected="selected"></option>
  <option value="17">Alshon</option>
  <option value="15">Brandon</option>
  <option value="22">Matt</option>
  <option value="83">Martellus</option>
  <option value="6">Jay</option>
</select>
```

JSFiddle: <http://jsfiddle.net/msfrisbie/umehb407/>

Dokonanie wyboru w dalszym ciągu spowoduje przypisanie odpowiedniego obiektu w tablicy do modelu player.

Jawne definiowanie przypisania modelu opcji

Jeżeli zamiast sztywnego ustawiania elementów `<option>` na atrybut `number` każdego elementu tablicy chcemy mieć jawną kontrolę nad wartością każdego elementu `<option>`, możemy skorzystać z następującego sposobu:

(*index.html*)

```
<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <!-- etykieta na wartość w tablicy -->
    <select ng-model="player"
      ng-options="p.number as p.name for p in players">
    </select>
  </div>
</div>
```

Powyższy kod skompiluje się do postaci zamieszczonej poniżej (klasy CSS związane z formularzami zostały pominięte):


```

<select ng-model="player"
  ng-options="p.number as p.name for p in players">
  <option value="?" selected="selected"></option>
  <option value="17">Alshon</option>
  <option value="15">Brandon</option>
  <option value="22">Matt</option>
  <option value="83">Martellus</option>
  <option value="6">Jay</option>
</select>

```

JSFiddle: <http://jsfiddle.net/msfrisbie/jtsz46cp/>

Jednak teraz, gdy zostanie wybrany element `<option>`, do modelu `player` zostanie przypisany tylko atrybut `number` odpowiedniego obiektu.

Implementacja grup opcji

Aby skorzystać z możliwości grupowania dla elementów `<select>`, możemy dodać klauzulę `group by`, tak jak pokazano poniżej:

(*index.html*)

```

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <!-- etykieta na wartość w tablicy -->
    <select ng-model="player"
      ng-options="p.name group by p.position for p in
        players">
      </select>
    </div>
  </div>
</div>

```

Powyższy kod skompiluje się do następującej postaci:

```

<select ng-model="player"
  ng-options="p.name group by p.position for p in players">
  <option value="?" selected="selected"></option>
  <optgroup label="WR">
    <option value="0">Alshon</option>
    <option value="1">Brandon</option>
  </optgroup>
  <optgroup label="RB">
    <option value="2">Matt</option>
  </optgroup>
  <optgroup label="TE">
    <option value="3">Martellus</option>
  </optgroup>
  <optgroup label="QB">

```

```

    <option value="4">Jay</option>
  </optgroup>
</select>

```

JSFiddle: <http://jsfiddle.net/msfrisbie/2d6mdt9m/>

Opcje null

Aby zezwolić na opcje null, możemy jawnie je zdefiniować wewnątrz znacznika `<select>`, tak jak pokazano poniżej:

(*index.html*)

```

<select ng-model="player" ng-options="comprehension_expression">
  <option value="">Wybierz gracza</option>
</select>

```

Wypełnianie na podstawie obiektu

Elementy `<select>` korzystające z dyrektywy `ngOptions` mogą również zostać wypełnione atrybutami obiektu. Mechanizm ten działa podobnie do sposobu polegającego na przetwarzaniu tablicy danych. Jedyną różnicą polega na konieczności zdefiniowania par klucz-wartość w obiekcie, które będą wykorzystane do wygenerowania listy elementów `<option>`. W prostym przypadku w celu zmapowania właściwości `number` obiektu `value` na cały obiekt `value` można dodać następujący kod:

(*index.html*)

```

<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <!-- etykieta na wartość w tablicy -->
    <select ng-model="player"
      ng-options="p.number for (pos, p) in team">
    </select>
  </div>
</div>

```

Po skompilowaniu uzyskamy następujący kod:

```

<select ng-model="player"
  ng-options="p.number for (pos, p) in team">
  <option value="?" selected="selected"></option>
  <option value="1B">9</option>
  <option value="2B">19</option>
  <option value="3B">48</option>
  <option value="C">28</option>
  <option value="SS">35</option>
</select>

```

JSFiddle: <http://jsfiddle.net/msfrisbie/zofojs7n/>

Domyślnie wartościami `<option>` są łańcuchy klucza, ale model `player` nadal będzie przypisany do całego obiektu, do którego odnosi się klucz.

Jawne definiowanie wartości opcji

Jeżeli nie chcemy, aby do wartości elementu HTML `<option>` była przypisana właściwość klucza, możemy to przesłonić za pomocą klauzuli `select as`:

(*index.html*)

```
<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <!-- etykieta na wartość w tablicy -->
    <select ng-model="player"
      ng-options="p.number as p.name for (pos, p) in team">
    </select>
  </div>
</div>
```

Po skompilowaniu uzyskamy następujący kod:

```
<select ng-model="player"
  ng-options="p.number as p.name for (pos, p) in team">
  <option value="" selected="selected"></option>
  <option value="1B">Brandon</option>
  <option value="2B">Marco</option>
  <option value="3B">Pablo</option>
  <option value="C">Buster</option>
  <option value="SS">Brandon</option>
</select>
```

JSFiddle: <http://jsfiddle.net/msfrisbie/ssLzvta/>

Teraz, kiedy zostanie wybrany element `<option>`, do modelu `player` zostanie przypisana tylko właściwość `number` odpowiedniego obiektu.

Jak to działa?

Dyrektywa `ngOptions` po prostu analizuje przekazany do niej element typu wyliczeniowego na fragmenty, które mogą zostać przekształcone na tagi `<option>`.

Co dalej?

Wewnątrz znacznika `<select>`, ze względów wydajnościowych, preferowane jest użycie dyrektywy `ngOptions` zamiast `ngRepeat`. W przypadku wartości w rozwijanych kontrolkach wiązanie danych nie jest jak konieczne, dlatego implementacja bazująca na `ngRepeat`, która musi przeglądać wiele wartości w kolekcji, wprowadza w aplikacji niepotrzebny narzut wiązania danych.

Budowanie magistrali zdarzeń

W zależności od przeznaczenia aplikacji czasami może pojawić się potrzeba skorzystania z architektury *publikacja-subskrypcja* (od ang. *publish-subscribe* — *pub-sub*), która pozwala zrealizować kilka funkcji. Framework AngularJS dostarcza właściwego zestawu narzędzi, który pozwala zaimplementować tę architekturę. Jednak by zapobiec obniżeniu wydajności i utrzymać dobrą organizację aplikacji, trzeba wziąć pod uwagę pewne sprawy.

Dawniej używanie usługi `$broadcast` z poziomu zakresu zawierającego bardzo dużą liczbę zakresów potomnych, ze względu na dużą liczbę potencjalnych słuchaczy, którzy potrzebowali obsłużenia, wносиło znaczące obniżenie wydajności. W wydaniu AngularJS 1.2.7 wprowadzono optymalizację usługi `$broadcast`. Polega ona na ograniczeniu zasięgu zdarzenia tylko do tych zakresów, które nasłuchują tego zdarzenia. Dzięki wprowadzonemu usprawnieniu można swobodniej korzystać z usługi `$broadcast` w aplikacji. Nadal jednak istnieje luka do wypełnienia w celu obsługi aplikacji usługowych wymagających architektury *pub-sub*. Mówiąc prosto, aplikacja powinna być w stanie nadawać zdarzenie do subskrybentów bez konieczności korzystania z metody `$rootScope.$broadcast()`.

Przygotuj się

Przypuśćmy, że mamy aplikację zawierającą wiele różniących się od siebie zakresów, które muszą zareagować na pojedyncze zdarzenie, tak jak pokazano poniżej:

```
(app.js)

angular.module('pubSubApp', [])
  .controller('Ctrl',function($scope) {})
  .directive('myDir',function() {
    return {
      scope: {},
      link: function(scope, el, attrs) {}
    };
  });
```

W tym przykładzie pokazano tylko pojedynczy kontroler i dyrektywę, ale z magistrali zdarzeń może korzystać nieograniczona liczba komponentów, które mają dostęp do obiektu zakresu.

Jak to zrobić?

Aby uniknąć korzystania z metody `$rootScope.$broadcast()`, obiekt `$rootScope` będzie użyty jako centralny punkt mechanizmu rozsyłania komunikatów obejmującego całą aplikację. Skorzystanie z metody `$rootScope.$on()` i `$rootScope.$emit()` pozwala podzielić właściwe nadawanie komunikatów do pojedynczego zakresu i umożliwia zakresom potomnym wstrzykiwanie obiektu `$rootScope` i korzystanie za jego pośrednictwem z magistrali zdarzeń.

Prosta implementacja

Najprostsza i najbardziej naiwna implementacja polega na wstrzyknięciu obiektu `$rootScope` do każdej lokalizacji, w której jest potrzebny dostęp do magistrali zdarzeń, i skonfigurowaniu zdarzeń lokalnie, tak jak przedstawiono poniżej:

(*index.html*)

```
<div ng-app="myApp">
  <div ng-controller="Ctrl">
    <button ng-click="generateEvent()">Generuj zdarzenie</button>
  </div>
  <div my-dir<</div>
</div>
```

(*app.js*)

```
angular.module('myApp', [])
.controller('Ctrl', function($scope, $rootScope, $log) {
  $scope.generateEvent = function() {
    $rootScope.$emit('busEvent');
  };
  $rootScope.$on('busEvent', function() {
    $log.log('Wywołano handler!');
  });
})
.directive('myDir', function($rootScope, $log) {
  return {
    scope: {},
    link: function(scope, el, attrs) {
      $rootScope.$on('busEvent', function() {
        $log.log('Wywołano handler!');
      });
    }
  };
});
```

JSFiddle: <http://jsfiddle.net/msfrisbie/5ot5scja/>

Przy takiej konfiguracji z magistrali zdarzeń może korzystać nawet dyrektywa z odizolowanym zakresem. Może ona komunikować się z kontrolerem, z którym w innej sytuacji nie mogłaby się komunikować.

Porządkowanie

Uważny czytelnik być może zauważył, że używanie tego wzorca wprowadza pewien problem. Kontrolery w AngularJS nie są singletonami i dlatego korzystanie z tego rodzaju architektury obejmującej wiele aplikacji wymaga ostrożnego zarządzania pamięcią.

W szczególności, gdy kontroler w aplikacji ulegnie zniszczeniu, słuchacz zdarzenia dowiązany do obcego zakresu, który został zadeklarowany wewnątrz niego, nie będzie posprzątany przez mechanizm odśmiecania, a to może doprowadzić do wycieków pamięci. Aby temu zapobiec, można zarejestrować słuchacza zdarzeń z metodą `$on()`. Spowoduje to zwrócenie funkcji deregistracji, którą trzeba wywołać w odpowiedzi na zdarzenie `$destroy`. Można to zrobić w następujący sposób:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl', function($scope, $rootScope, $log) {
    $scope.generateEvent = function() {
        $rootScope.$emit('busEvent');
    };

    var unbind = $rootScope.$on('busEvent', function() {
        $log.log('Wywołano handler!');
    });

    $scope.$on('$destroy', unbind);
})
.directive('myDir', function($rootScope, $log) {
    return {
        scope: {},
        link: function(scope, el, attrs) {
            var unbind = $rootScope.$on('busEvent', function() {
                $log.log('Wywołano handler!');
            });

            scope.$on('$destroy', unbind);
        }
    };
});
```

JSFiddle: <http://jsfiddle.net/msfrisbie/xq05p9dt/>

Magistrala zdarzeń jako usługa

Logikę magistrali zdarzeń można oddelegować do fabryki usług. Stworzoną usługę można następnie wstrzyknąć jako zależność w dowolnym miejscu, co pozwala na zgłaszanie zdarzeń obejmujących zakres całej aplikacji do wszystkich słuchaczy, niezależnie od miejsca, w którym istnieją. Można to zrobić w następujący sposób:

```
(app.js)

angular.module('myApp', [])
.controller('Ctrl',function($scope, EventBus, $log) {
    $scope.generateEvent = function() {
        EventBus.emitMsg('busEvent');
    };

    EventBus.onMsg(
        'busEvent',
        function() {
            $log.log('Wywołano handler!');
        },
        $scope
    );
})
.directive('myDir',function($log, EventBus) {
    return {
        scope: {},
        link: function(scope, el, attrs) {
            EventBus.onMsg(
                'busEvent',
                function() {
                    $log.log('Wywołano handler!');
                },
                scope
            );
        }
    };
})
.factory('EventBus', function($rootScope) {
    var eventBus = {};
    eventBus.emitMsg = function(msg, data) {
        data = data || {};
        $rootScope.$emit('busEvent');
    };
    eventBus.onMsg = function(msg, func, scope) {
        var unbind = $rootScope.$on(msg, func);
        if (scope) {
            scope.$on('$destroy', unbind);
        }
        return unbind;
    };
});
```

```

    };
    return eventBus;
  });

```

JSFiddle: <http://jsfiddle.net/msfrisbie/m88ruycx/>

Magistrala zdarzeń jako dekorator

Najlepszą i najczystsza implementacją magistrali zdarzeń jest domyślne dodanie metod narzędziowych `publish` i `subscribe` do wszystkich zakresów. Jest to możliwe dzięki udekorowaniu obiektu `$rootScope` podczas inicjalizacji aplikacji, w szczególności podczas fazy `config`.

(*app.js*)

```

angular.module('myApp', [])
.config(function($provide) {
  $provide.decorator('$rootScope', function($delegate){
    // dodanie do prototypu konstruktora, aby umożliwić
    // wykorzystanie w odizolowanych zakresach
    var proto = $delegate.constructor.prototype;

    proto.subscribe = function(event, listener) {
      var unsubscribe = $delegate.$on(event, listener);
      this.$on('$destroy', unsubscribe);
    };

    proto.publish = function(event, data) {
      $delegate.$emit(event, data);
    };

    return $delegate;
  });
})
.controller('Ctrl', function($scope, $log) {
  $scope.generateEvent = function() {
    $scope.publish('busEvent');
  };
});

$rootScope.$on('busEvent', function() {
  $log.log('Wywołano handler!');
});
})
.directive('myDir', function($log) {
  return {
    scope: {},
    link: function(scope, el, attrs) {
      $rootScope.$on('busEvent', function() {

```



```

    });
  }
});
    $log.log('Wywołano handler!');
  });
}

```

JSFiddle: <http://jsfiddle.net/msfrisbie/5madmyzt/>

Jak to działa?

Magistrala zdarzeń spełnia rolę pojedynczego celu pośredniego pomiędzy odrębnymi podmiotami w aplikacji. Dzięki temu, że zdarzenia nie uciekają od obiektu `$rootScope`, który może być wstrzyknięty jako zależność, tworzymy sieć przesyłania komunikatów o zasięgu całej aplikacji.

Co dalej?

Mówiąc o zdarzeniach, należy zawsze wziąć pod uwagę względy wydajności. Czystsze i bardziej skuteczne jest oddelegowanie jak największej części aplikacji do warstwy wiązania danych (modelu), ale w przypadku zdarzeń globalnych, które wymagają propagacji zdarzeń (na przykład logowanie i wylogowanie), zdarzenia mogą być niezmiernie przydatnym narzędziem.

Skorowidz

\$templateCache, 18, 19

A

AngularJS

- błędy, 194
- brudne sprawdzanie, 226, 229
- dyrektywa, *Patrz:* dyrektywa
- grunt-angular-templates, 152
- obietnica, 247
- obserwator, 224, 225, 229, 230
 - drzewo, 227
 - plytki, 236
 - pojedynczej głębokości, 238
 - pośredniego poziomu, 236
 - referencyjny, 231, 232, 233, 236, 238, 282
 - równości, 234, 235, 239, 240
 - uniwersalny, 227
 - wyrejestrowanie, 238, 239
- rozszerzenie Grunt, *Patrz:* Grunt
- Scenario Runner, *Patrz:* Scenario Runner
- składnia controler as, 152, 153, 154
- szybki, 223
- ukrywanie przed użytkownikiem, 146, 147
- wersja 1.3, 279
- wydajność, 226

animacja, 91

- addClass, 121
- CSS, 95, 100, 101, 102, 105, 107, 108, 113, 115, 122, 123, 127, 128
- CSS3, 91
- dwukierunkowa, 99
- enter, 102, 103, 107, 112, 120
- fade in, 92, 93, 100, 122
- fade out, 92, 93, 122

- JavaScript, 91, 100, 102, 105, 107, 109, 116, 122, 123, 127, 128
- leave, 100, 105, 107, 110
- move, 112, 117, 120
- ng-hide, 94, 96
- ng-show, 94, 96, 98
- slide-in, 112
- slide-out, 112, 127
- stagger, 131, 133
- wsadowa, 131, 133

aplikacja

- badanie zakresów, 304
- inicjowanie, 135, 136
- ładowanie ręczne, 136
- manipulowanie z poziomu konsoli, 304, 306, 307
- szablon, 148, 151
 - lokalizacja, 148, 149, 151

atribut

- dyrektywa, *Patrz:* dyrektywa atrybutu
- elementu, *Patrz:* element atrybut
- ouinterval, 32
- require, 34

B

Batarang, 304

biblioteka

- AngularJS, 47
- jQuery, 47
- Restangular, *Patrz:* Restangular

D

dane

- data, 59
- wiązanie, 31, 294
 - biała lista, 31
 - dwustronne, 63
 - pojedyncze, 294, 295, 296

dekorator usług, *Patrz:* usługa dekorator
DOM, 25, 92

dostawca usług, *Patrz:* usługa dostawca
dyrektywa, 20

- atrybutu, 22
- klasy, 23
- komentarza, 24, 314
- kontroler, *Patrz:* kontroler dyrektywy
- łącząca, 27
- ng-app, 135
- ng-bind, 310
- ng-class, 126
- ngClass, 121, 126
- ng-cloak, 146, 147, 309
- ng-controller, 152
- ngController, 163
- ngForm, 121, 126, 171
- ngHide, 121, 126
- ngIf, 163, 167
- ngInclude, 163
- ng-message, 291, 292
- ng-messages, 291
- ngMessages, 121, 126, 291
- ng-model, 284
- ngModel, 121, 126, 298
- ngModelOptions, 284
 - \$rollbackViewValue, 288
 - allowInvalid, 286
 - debounce, 285, 288
 - getterSetter, 286
 - timezone, 287
 - updateOn, 285, 288
- ng-move, 119
- ngOptions, 177, 178, 182, 183, 184
- ng-repeat, 117, 131, 241, 242, 243, 296
 - track by, 243, 244
- ngRepeat, 112, 163, 165, 166, 167, 184
- ng-show, 122
- ngShow, 121, 126
- ng-strict-di, 283, 284
- ngSwitch, 163, 169

- ng-template, 150
- ng-view, 274
- ngView, 163, 164
- rekurencyjna, 46, 47, 49
- szablon, 19, 21, 39, 40
- transkluzja, 43, 45
- zagnieżdżona, 34, 50
 - kontroler opcjonalny, 36, 37
- zakres, 20, 29, 30, 37, 38
 - odizolowany, 30, 31, 33, 42, 43
- dziedziczenie, 29, 37, 38, 159, 160

E

element

- atrybut, 25
- option, 180
- select, 177
 - grupowanie, 181
- tworzenie, 18

F

fabryka usług, *Patrz:* usługa fabryka
filtr, 72

- currency, 56, 58, 59, 64
- danych, 54, 69
- date, 59, 60, 61
- json, 61, 62
- limitTo, 69
- lowercase, 54, 55, 56
- łańcuch, 67, 68, 69
- niestandardowy, 64, 69, 71, 72
- number, 56, 57, 58, 59, 64
- optymalizacja, 81
- orderBy, 69
- uppercase, 54, 55, 56
- wolno działający, 224
- wstrzykiwanie zależności, 64
- wyszukiwania, 65, 68, 72, 74, 76
 - ng-repeat, 113, 224
 - tworzenie, 76, 77, 78, 79, 80
 - z niestandardowym komparatorem, 74, 76
- Finite State Machine, *Patrz:* maszyna stanów skończonych
- formularz, 170, 171, 284
 - komunikat o błędzie, 291, 292
 - szablon, 292, 293
- kontrolka wejściowa, 171, 173, 284, 285

stan, 289
 \$pristine, 289
 \$submitted, 290
 \$touched, 289
 brudny, 171, 173
 czysty, 171, 173
 walidacja, 173, 175, 298, 299, 300, 301
 niestandardowa, 175, 176
 FSM, *Patrz:* maszyna stanów skończonych
 funkcja
 compile, 27
 definition, 18
 deregistracji, 186
 expect<HTTPverb>, 205
 filtrów, 63
 link, 20, 35, 51
 when<HTTPverb>, 205

G

Grunt, 152, 192, 193, 197, 198, 199

H

handler, 252, 253, 256, 257
 definiowanie, 254
 łańcuch, 255, 257
 odrzucenia, 252, 255
 powiadomień, 260, 261
 rozwiązania, 252, 253
 hermetyzacja, 18, 83, 205
 HTML element, *Patrz:* element

I

instrukcja
 \$log.log, 25
 ng-transclude, 45
 npm install, 197
 protractor:run, 198
 require, 34
 restrict, 19
 update, 197

J

Jasmine, 193, 196
 JavaScript, 113
 JSON, 61

K

Karma, 192, 193
 klasa
 animacji, 93
 CSS, 94
 dyrektywa, *Patrz:* dyrektywa klasy
 FormController, 172, 174
 ng-dirty, 172
 ng-enter, 100, 104
 ng-leave, 110
 ngModelController, 172, 174
 ng-pristine, 172
 ng-scope, 105
 singletonów wstrzykiwalna, 54
 -stagger, 132
 komentarza dyrektywa, *Patrz:* dyrektywa
 komentarza
 HTML, 312
 niestandardowy, 312, 313
 kompilator HTML, 19
 kontroler, 36, 37
 aplikacji, 27
 dyrektywy, 27
 refaktoryzacja, 308
 testowanie, 204

L

liczba zmiennoprzecinkowa, 58

M

maszyna stanów, 102
 animacji, 95
 klas AngularJS, 94, 95
 ng-show, 94, 98
 skończonych, 92
 metoda
 \$animate.addClass, 125
 \$animate.leave, 110
 \$animate.removeClass, 130
 \$apply, 138, 140, 143
 \$parse, 320, 321
 \$q.all, 263
 \$q.reject, 265
 \$q.when, 264, 265
 \$rootScope.\$broadcast, 185
 \$rootScope.\$emit, 185

metoda

- \$rootScope.\$on, 185
- \$scope.\$apply, 139, 141, 142
- \$scope.\$broadcast, 156
- \$scope.\$digest, 141
- \$scope.\$emit, 157
- \$scope.\$on, 156, 159
- \$scope.\$watch, 159, 226
- \$scope.\$watchCollection, 226
- \$scope.\$watchGroup, 226
- \$setPristine, 173
- \$watch, 227
- \$watchGroup, 281, 282
- angular.bootstrap, 136, 137
- angular.copy, 235
- angular.element, 47, 48
- angular.equals, 235
- angular.extend, 309
- catch, 258
- data, 229
- enter, 100
- error, 266
- expect, 205
- expectGET, 205
- finally, 258
- func, 33
- jQuery ready, 137
- map, 245
- ngAnimate, 122
- promise.then, 259
- publish, 188
- setInterval, 139
- slideDown, 96
- slideUp, 96
- subscribe, 188
- success, 266
- then, 254
- when, 205

model, 71, 155

- DOM, *Patrz:* DOM

moduł

- ngAnimate, 92, 107
- ngMessages, 291
- ngMockE2E, 210
- ngResource, 269

O

obiekt

- \$filter, 63, 64
- \$http, 266
- \$injector, 304
- \$location, 306
- \$parse, 315, 316, 317, 318
- \$q, 249
- \$resource, 269, 270
- \$rootScope, 185, 188, 305
- \$scope, 159, 160, 161, 163, 307, 308
- \$watch, 239
- \$watchCollection, 224, 236, 238
- angular, 304
- attrs, 27
- comprehension_expression, 177, 179
- Date, 280, 281
- definicji dyrektywy, 18
- FormController, 171
- jQuery, 47
- odroczone, *Patrz:* odroczenie
- serializowanie, 61, 62
- strony, 216, 221
- obietnica, 247, 248, 250, 251, 273, 277
 - handler, *Patrz:* handler
 - implementacja, 248
 - JavaScript, 249
 - łańcuch, 254, 256, 257
 - modyfikowanie, 250
 - oceniona, 252
 - powiadomienie, 259
 - zbiorcza ocena grupy, 262
- odroczenie, 251, 253

P

pakiet npm, 197, 213

- para klucz-wartość, 20, 82

plik

- *.conf.js, 193, 196
- angular.js, 136
- angularanimate.js, 92
- Gruntfile, 192
- Gruntfile.js, 198
- inventory-controller.js, 144
- JSON, 310
- package.json, 197
- protractor.conf.js, 199

procedura nasłuchiwania zdarzeń, 29
 Protractor, 193, 195, 196, 197, 210
 browser, 195
 element, 196
 konfiguracja, 199
 zmienna
 globalna, 195
 przeglądarka, 195, 197
 przejście
 CSS, 94, 99, 100, 101, 105, 107, 113, 114,
 122, 127
 CSS3, 91
 punkt kotwicy, 29

R

rekurencja, 46, 47, 49
 repeater, 66
 optymalizacja, 81
 resolver
 deklarowanie, 272
 routingu, 272
 stanu, 274
 ścieżki, 273, 277
 ui-router, 274
 Restangular, 270, 271

S

Scenario Runner, 193, 195
 Selenium, 197
 Selenium WebDriver, 193, 195, 197, 210
 serwer
 backend, 210, 214
 express.js, 214
 makieta, 210, 211, 214
 Selenium, 210
 WebDriver, 210
 stała usługi, 81, 82, 83

T

tag, 20
 test, 191
 DAMP, 214, 216
 E2E, *Patrz:* test od końca do końca
 end-to-end, *Patrz:* test od końca do końca
 jednostkowy, 144, 192, 194, 206, 215
 jakość, 215
 Karma, *Patrz:* Karma

 tworzenie, 200, 201, 203, 205
 uruchamianie, 205
 metadane, 194
 od końca do końca, 192, 193, 194, 200, 210, 211
 Protractor, *Patrz:* Protractor
 tworzenie, 206, 207, 210
 składnia Jasmine, *Patrz:* Jasmine
 wzorzec Obiekt strony, 216, 220, 221
 transkluzja, 43, 45
 typ
 date, 280
 month, 280
 time, 280
 week, 280
 wejściowy, 280

U

usługa, 54, 85
 \$animate, 103, 104, 110, 111, 119, 125, 130
 \$broadcast, 184
 optymalizacja, 184
 \$compile, 47, 51
 \$http, 266
 \$httpBackend, 205
 \$locale, 59
 \$parse, 316, 317, 318, 320
 \$provide, 88
 \$resource, 269
 \$rootScope, 141
 dekorator, 88, 89
 dostawca, 86, 88
 fabryka, 83, 84, 85, 187
 manipulowanie typami, 306
 stała, *Patrz:* stała usługi
 value, 81
 wartość, *Patrz:* wartość usługi

W

warstwa prezentacji, 71
 wartość usługi, 81, 83
 WebDriver, *Patrz:* Selenium WebDriver
 wiązanie danych, *Patrz:* dane wiązanie
 właściwość
 \$validators, 298
 promise, 250
 template, 18

wrapper
 \$timeout, 143
 obietnic, 264
 scope.\$apply, 29
 wyciek pamięci, 186
 wywołanie
 jqLite, 26
 jQuery, 26
 wywołanie zwrotne, 226, 227, 230, 249, 253
 \$timeout, 306
 \$watch, 281

Y

Yeoman, 192, 193, 196

Z

zagnieżdżanie, 49
 zasada
 DAMP, 214, 215, 216
 DRY, 214, 216, 307
 zdarzenie, 156
 \$destroy, 186
 \$document mousemove, 29
 addClass, 121
 AngularJS, 34
 emitowanie, 157
 enter, 99, 102, 103, 112
 leave, 105, 107
 magistrala, 184, 187, 188
 move, 112, 113, 117
 nasłuchiwanie, 29
 ngClass, 121
 ngForm, 121

ngHide, 121
 ngIf, 99, 100, 105
 ngInclude, 99, 105
 ngMessage, 99, 105
 ngMessages, 121
 ngModel, 121
 ngRepeat, 99, 105
 ngShow, 121
 ngSwitch, 99, 105
 ngView, 99, 105, 107
 removeClass, 126
 rozgłaszanie, 156
 slide-in, 105
 slide-out, 105
 wydajność, 189
 zmiany, 281
 znacznik form, 171
 znak
 \$, 59, 64, 75
 \$\$, 254
 &, 33
 ::, 294
 ?, 37
 @, 31, 33
 {{}}, 239, 240
 =, 33
 €, 64

Ż

żądanie
 \$http.get, 268
 obsługa warunkowa, 214

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>

AngularJS

Tworzenie aplikacji webowych. Receptury

AngularJS jest popularnym frameworkiem służącym do pisania aplikacji internetowych o różnej skali. Jest to świetny projekt open source, dzięki któremu praca staje się wydajna i przyjemna. Tworzenie oprogramowania mającego naprawdę wysoką jakość wymaga od programisty stałego rozwijania umiejętności. Konieczne jest doskonałe opanowanie narzędzi programistycznych i uczenie się nowych funkcji, które one oferują. Proces ten w istocie nigdy się nie kończy.

Jeśli już znasz podstawowe cechy AngularJS i postanowiłeś nabyć biegłości w pracy z tym frameworkiem, to masz w rękach książkę przeznaczoną dla Ciebie. Przedstawione w niej metodologie i strategię pozwolą Ci na sprawne budowanie wydajnych i skalowalnych aplikacji internetowych. W tym praktycznym poradniku znajdziesz ponad 90 łatwych do wykonania receptur. Od razu zaczniesz korzystać z praktycznych rozwiązań. Z pewnością docenisz jasne podejście do problemów, klarowne wyjaśnienia i czytelne wskazówki dotyczące tworzenia aplikacji produkcyjnych. Oczywiście nie zabrakło licznych fragmentów kodu opatrzonego komentarzem.

AngularJS to świetne narzędzie — poznaj jego tajniki!



Dzięki tej książce poznasz:

- wzorce, strategię i praktyki, dzięki którym stworzysz prawdziwie skalowalne, wydajne i efektywne aplikacje internetowe
- najnowsze narzędzia ostatniej wersji frameworka — AngularJS 1.3
- rozwiązania najczęstszych problemów z testowaniem i optymalizacją kodu

Matt Frisbie — absolwent Uniwersytetu Illinois w Urbana-Champaign. Jest deweloperem w firmie DoorDash (YC S13). Odpowiada za wdrożenie frameworka AngularJS. W swojej pracy koncentruje się na projektach dotyczących infrastruktury, prognoz i obsługi danych. Jest również autorem serii klipów wideo *Learning AngularJS* dostępnych za pośrednictwem witryny O'Reilly Media.

[PACKT] open source
PUBLISHING community experience distilled

Helion

41746 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/nawosci>

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 978-83-283-2148-9



9 788328 321489

Informatyka w najlepszym wydaniu

cena: 59,00 zł