Dariusz Kalbarczyk
Arkadiusz Kalbarczyk

ANGULARJS

PIERWSZE KROKI

Helion

Opieka redakcyjna: Ewelina Burska
Projekt okładki: Studio Gravite/Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki
Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock.

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/angupk
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-0586-1

Printed in Poland.

Spis treści

Rozdział 2.

$scope — niepozorny obiekt

Wprowadzenie

W tym rozdziale zajmiemy się wspomnianym przez nas wcześniej obiektem `$scope`.

Jego podstawowe zadania to:

- transportowanie modelu pomiędzy widokiem a kontrolerem;
- nasłuchiwanie zdarzeń bądź zmian zachodzących w modelu;
- propagacja zmian modelu.

Mimo że odgrywa wyjątkową rolę, `$scope` to wciąż zwykły obiekt POJO. Oznacza to, że możemy dowolnie przypisywać mu oraz modyfikować atrybuty według własnego uznania. Wyróżnia go fakt, iż w większości przypadków jest on za nas automatycznie tworzony i wstrzykiwany.

$scope i $rootScope

W fazie ładowania początkowego aplikacji (tzw. *bootstrap*) AngularJS tworzy wiązanie (binduje) pomiędzy znacznikiem zawierającym dyrektywę `ng-app` a wszystkim, co jest zawarte w elementach poniżej.

`$rootScope` jest rodzicem wszystkich obiektów `$scope` i znajduje się najwyżej w hierarchii. Instancja `$rootScope` jest tworzona w momencie bootstrapowania aplikacji. Każdy program posiada dokładnie jeden taki obiekt, po którym dziedziczą wszystkie inne obiekty `scope`. Nie zalecamy przypisywania mu zbyt wielu atrybutów, gdyż jest on czymś na wzór obiektu globalnego, którego nie powinno się zaśmiecać. Przy wykorzystywaniu więcej niż jednej biblioteki lub frameworka istnieje ryzyko wystąpienia

zbieżności nazw atrybutów bądź metod przypisanych do globalnych obiektów. Tego typu problemy są niezwykle uciążliwe w usuwaniu.

Wspominaliśmy już, że każdy element przypisany do `$scope` jest od razu dostępny w widoku. Przypisywanie atrybutów i funkcji do modelu po stronie kontrolera odbywa się w sposób ukazany w listingu 2.1.

Listing 2.1. *Kontroler — przypisanie atrybutów*

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" ng-app="app">
<head>
    <title>Kontroler – przypisanie atrybutów</title>
</head>
<body>
    <div ng-controller="dateCtrl">
        Data: {{orginal() | date}}
    </div>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/
    ↪1.4.0-beta.5/angular.min.js">
    </script>
    <script>
        var app = angular.module('app', []);
        app.run(function ($rootScope) {
            $rootScope.dateOrginal = new Date();
        });
        app.controller('dateCtrl', function ($rootScope, $scope) {
            $scope.orginal = function () {
                return $rootScope.dateOrginal;
            };
        });
    </script>
</body>
</html>
```

W powyższym przykładzie zdefiniowaliśmy w `$rootScope` właściwość `dateOrginal`. Następnie w kontrolerze `dateCtrl` stworzyliśmy funkcję `orginal`, która zwraca nam datę z `$rootScope`.

Alternatywa dla $scope

Istnieje również możliwość przypisywania atrybutów do modelu po stronie widoku bez odwoływania się do `scope`. W tym celu korzystamy z dyrektywy `ng-model`. Jest ona dokładnie opisana w rozdziale 5., poświęconym dyrektywom wbudowanym. Na tym etapie warto zapamiętać, że `ng-model` inicjuje nam `$scope`, którego możemy użyć w kontrolerze.

```
<html ng-app>
...
      <div ng-model='wiadomosc'>
            <p> {{ wiadomosc }} </p>
      </div>
</html>
```

Stosując się do dobrych praktyk, powinniśmy w miarę możliwości wybierać wariant pierwszy, bliższy ideologii MVC.

Dziedziczenie

W przykładzie z listingu 2.1 wykorzystaliśmy wcześniej wspomniany `$rootScope`. Jak już mówiliśmy, staramy się nie przypisywać atrybutów do obiektu głównego, lecz do nowo utworzonego `scope` znajdującego się w hierarchii poniżej.

```
app.controller('dateCtrl', function ($scope) {
    $scope.wiadomosc = "Przypisujemy wiadomosc do widoku!";
    $scope.funkcjaA = function() {
       return wiadomosc + "Dodajemy dodatkowe zdanie";
    }
}
```

`$scope` odwzorowuje strukturę DOM. Oznacza to, że możemy swobodnie zagnieżdżać jego obiekty.

Korzystanie z obiektów `$scope` nie wymaga ich wcześniejszej deklaracji. Większość obiektów `$scope` tworzona jest dzięki metodzie `$new()`, wywoływanej za każdym razem, gdy napotykana jest dyrektywa `ng-controller`.

Nowy obiekt zostaje automatycznie zagnieżdżony poniżej obiektu `$rootScope`. Poza jednym wyjątkiem (izolowanym `scope`) wszystkie obiekty `$scope` mają dostęp do obiektów znajdujących się w hierarchii nad nimi. Jeżeli AngularJS nie znajdzie pożądanej informacji w `scope` na swoim poziomie, to rozpocznie przeszukiwanie obiektu znajdującego się wyżej, aż dojdzie do `$rootScope`.

Zobaczmy na listingu 2.2, jak możemy korzystać z dziedziczenia kontrolerów:

Listing 2.2. *Kontrolery — dziedziczenie*

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" ng-app="app">
<head>
    <title>Kontrolery – dziedziczenie</title>
</head>
<body>
    <div ng-controller="defaultCtrl">
        <div ng-controller="inheritanceCtrl">
            <input type="text" ng-model="uczen.imie" placeholder="Imie
            ↪Ucznia"></input>
            <button ng-click="poprawaTestu()">Poprawa testu </button>
        </div>
        {{ uczen }}
    </div>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/
    ↪1.4.0-beta.5/angular.min.js">
    </script>
    <script>
```

```
        var app = angular.module('app', []);
        app.controller('defaultCtrl', function ($scope) {
            $scope.uczen = { zdanyTest: false };
        });
        app.controller('inheritanceCtrl', function ($scope) {
            $scope.poprawaTestu = function () {
                $scope.uczen.zdanyTest = true;
            }
        });
    </script>
</body>
</html>
```

Ponieważ `inheritanceCtrl` związaliśmy w hierarchii niżej niż `defaultCtrl`, otrzymuje on dostęp do metod kontrolera bazowego. Tutaj warto odnotować, iż dziedziczenie w Angularze odbywa się w jednym kierunku. W tym wypadku kontroler potomny jest silnie powiązany z rodzicem, czyli może odwoływać się do jego metod. Jednakże kontroler bazowy nie może bezpośrednio odwoływać się do potomka. By uzyskać dostęp do owych metod, należy wykorzystać przesyłanie zdarzeń (ang. *event dispatching*). W większości przypadków, kiedy musimy odwoływać się do metod potomnych, oznacza to, że powinniśmy się przyjrzeć naszemu kodowi, gdyż najprawdopodobniej robimy coś źle.

Aby później mieć możliwość odwołania się do naszego `scope`, musimy umieścić dyrektywę `ng-controller` w dowolnym elemencie DOM znajdującym się na tym samym bądź wyższym poziomie hierarchii co model (a konkretnie nasze odwołanie do niego poprzez interpolację).

```
<html ng-app>
...
<div ng-controller="Kontroler">
    <p> {{ wiadomosc }} </p>
</div>
...
</html>
```

Dyrektywa `ng-controller` należy do grupy tzw. tworzących dyrektyw. Za każdym razem, gdy Angular napotyka jedną z takich dyrektyw, zostaje utworzona nowa instancja `scope`, dlatego wcześniejsza deklaracja w kontrolerze nie jest wymagana.

Wielu czytelników na pewno zadaje sobie pytanie, jaki jest sens wprowadzenia koncepcji dziedziczenia do `scope`.

By na nie odpowiedzieć, posłużymy się opisaną w rozdziale 5. dyrektywą `ng-repeat`. Przytoczymy krótki opis tej dyrektywy: *`Ng-repeat` pozwala nam iterować po dowolnej kolekcji obiektów, dodatkowo tworzy osobne elementy szablonu DOM dla każdego z elementów kolekcji*.

Listing 2.3 najlepiej nam to zobrazuje.

Listing 2.3. *Przykład zastosowania dyrektywy ng-repeat*

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" ng-app="app">
<head>
    <title>Przykład zastosowania dyrektywy ng-repeat</title>
</head>
<body>
    <div ng-controller="defaultCtrl">
        <ul>
            <li ng-repeat="oferta in oferty">
                <p> Nazwa: {{ oferta.nazwa }}  ||   cena: {{oferta.cena }} </p>
            </li>
        </ul>
    </div>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/
    ↪1.4.0-beta.5/angular.min.js">
    </script>
    <script>
        var app = angular.module('app', []);
        app.controller('defaultCtrl', function ($scope) {
            $scope.oferty = [
            { nazwa: 'Krzesło', cena: 149.99 },
            { nazwa: 'Stolik', cena: 189.99 },
            { nazwa: 'Szafka', cena: 205.99 },
            ];
        });
    </script>
</body>
</html>
```

Zmienne z każdego obiektu w kolekcji zostaną przypisane do `scope`, by później zostać *zrenderowanymi* przez widok.

Właśnie w tym momencie pojawia się problem. Aby każdą nową zmienną przypisać do `$scope`, musielibyśmy nadpisywać poprzednią ze względu na zbieżność nazw atrybutów. Dlatego też każdemu elementowi kolekcji przypisujemy nowy `scope`. Dana zmienna będzie „żyć" jedynie w obrębie swojego `scope`. Wszystkie nowo utworzone obiekty układają się w hierarchię przypominającą tę ze struktury DOM. Mamy możliwość wykorzystania tej samej nazwy dla zmiennej w różnych obiektach `scope`.

Podobnie jak w przypadku programowania zorientowanego obiektowo dziedziczenie pozwala na izolację atrybutów i funkcjonalności poszczególnych elementów modelu.

Dziedziczenie obiektów `scope` w Angularze odbywa się z użyciem wcześniej wspomnianej metody `$new()`.

```
var obiektBazowy = $rootScope;
var obiektPochodny = obiektBazowy.$new();
obiektBazowy.imie = 'Marian';
obiektPochodny.nazwisko = 'Kowalski';
```

W celu zniszczenia danego obiektu `scope` należy zastosować metodę `$destroy()`, która usuwa wszystkie obiekty pochodne (i ich pochodne) z obiektu bazowego. Od tej chwili dany `scope` jest gotowy na „odśmiecanie", czyli tzw. *garbage collection*.

Izolowany scope

Możliwe jest również utworzenie tzw. *izolowanego scope*, który nie dziedziczy po swoich rodzicach — jest to wcześniej przez nas wspomniany wyjątek. Używamy go podczas tworzenia komponentów, które chcielibyśmy później kilkakrotnie wykorzystać.

Tworząc izolowany `scope`, tak naprawdę bawimy się z pewnymi własnościami obiektu `scope`.

Wyobraźmy sobie sytuację, iż stworzyliśmy dyrektywę służącą np. do wyświetlania menu na stronie naszej restauracji. Nasza dyrektywa zawiera szablon dla wyświetlanych informacji. Podpinamy kontroler do modułu, przypisujemy potrawy do `$scope` i przypinamy dyrektywę. Gdybyśmy teraz umieścili kilka tagów z dyrektywą wewnątrz kodu HTML, to wyświetlana byłaby jedna i ta sama informacja. By temu zapobiec, musielibyśmy stworzyć osobny kontroler z nową instancją `scope` dla każdej potrawy. Pomysł czasochłonny i zmuszający do pisania masy nowego kodu, nie jest to więc najlepsze rozwiązanie. Tutaj właśnie wkraczają izolowane obiekty `scope`.

Aby odizolować `scope`, musimy wewnątrz naszej dyrektywy umieścić element `scope`.

```
...
return {
            scope: {}
}
...
```

Od tej chwili poszczególne instancje dyrektywy będą izolować swój lokalny `scope`. Możemy wiązać różne elementy przypisane do `scope`.

$digest(), $apply() i $watch()

Jak wcześniej wspominaliśmy, `scope` nie służy jedynie jako most dla danych. Do jego obowiązków należy między innymi nasłuchiwanie zmian zachodzących w modelu. W tym celu wykorzystujemy opisany w dalszej części tego rozdziału `$swatch`. `Scope` posiada również umiejętność wprowadzania (propagacji) zmian w modelu, znajdujących się wewnątrz aplikacji bądź pochodzących spoza niej.

Nasłuchiwanie oraz $watch()

Po przypisaniu `$watch` do wybranego elementu AngularJS zaczyna oczekiwać na ewentualne zmiany. W momencie ich zajścia wywoływana jest tzw. funkcja nasłuchująca (ang. *listener function*), która może reagować na te zmiany. Przyjrzyjmy się bliżej temu, jak wygląda nasłuchiwanie zmian przez kanciastego, ukazane na listingu 2.4.

Listing 2.4. *Nasłuchiwanie*

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" ng-app="app">
<head>
    <title>AngularJS - $watch</title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.1/
    ↪css/bootstrap.min.css">
</head>
<body>
    <div ng-controller="defaultCtrl">
        <div class="well">Liczba: {{number}}</div>
        <div>
            <a class="btn btn-success" href="#" ng-click="add()"> + </a>
            <a class="btn btn-danger" href="#" ng-click="dec()"> - </a>
        </div>
    </div>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/
    ↪1.4.0-beta.5/angular.min.js">
    </script>
    <script>
        var app = angular.module('app', []);
        app.controller('defaultCtrl', function ($scope) {
            $scope.number = 1;
            $scope.$watch('number', function () {
                console.log('Liczba: ' + $scope.number);
            });
            $scope.add = function () {
                $scope.number++;
            };
            $scope.dec = function () {
                $scope.number--;
            };
        });
    </script>
</body>
</html>
```

Wcześniej powiedzieliśmy, że dzięki *live binding* każda zmiana zachodząca w kontekście Angulara jest przez niego wyłapywana. Naturalnie rodzi się więc pytanie, czy każdy element przypisany do `$scope` otrzymuje od razu własny obiekt nasłuchujący. Odpowiedź brzmi: nie, gdyż nasłuchiwanie zmian na wszystkich elementach zajęłoby zbyt dużo czasu. Mamy do wyboru dwa sposoby zadeklarowania nasłuchiwania wybranych elementów:

- Pierwszy z nich to… interpolacja. Kiedy Angular napotyka interpolację w widoku, to wie, że automatycznie musi stworzyć obiekt nasłuchujący (w tym wypadku *implicit watcher*) na dany element.

```
<div>
        {{ watchedElement }}
</div>
```

- Istnieje również możliwość tworzenia nasłuchiwaczy własnoręcznie. Struktura typowego obiektu nasłuchującego (ang. *explicit watcher*) prezentuje się mniej więcej tak:

```
$watch('watchedElement', function(newValue, oldValue) {
            //functions body…
});
```

Pierwszy parametr to nazwa elementu modelu będącego pod obserwacją. Drugi parametr to funkcja nasłuchująca reagująca na zachodzące zmiany — jej wywołanie następuje za każdym razem, gdy wartość obserwowanego elementu ulega zmianie. Porównanie odbywa się poprzez metodę `angular.equals()`; wykonywana jest również metoda `angular.copy()` w celu zapisania obecnej wartości elementu. Obydwa przypadki zawarte są w naszym poprzednim przykładzie.

Zapewne niejedna osoba zastanawiała się, w jaki sposób Angular dowiaduje się o tych zmianach zachodzących w modelu. Za ich nasłuchiwanie odpowiada cykl `$digest()`.

$digest()

`$digest` rozpoczyna się jako efekt wywołania `$scope.digest()`. Jest to cykl ewaluacji kolejno wszystkich obiektów nasłuchujących występujących w danym `scope` oraz jego potomkach. Ponieważ zachodzące zmiany wywołują tzw. funkcje nasłuchujące, które mogą modyfikować dowolne elementy modelu (w tym te sprawdzone już wcześniej), `$digest()` powtarzany jest dopóty, dopóki owe wezwania nie ustaną. Nawet jeżeli podczas wykonywania cyklu nie zostanie wezwana żadna funkcja nasłuchująca, zostanie on powtórzony co najmniej raz w celu upewnienia się, iż nie zaszła żadna zmiana. Jeśli zdarzy się tak, że cykl wpadnie w pętlę nieskończoną, wówczas po 10 iteracjach zostanie zwrócony błąd.

Wywołanie cyklu następuje automatycznie, np. dzięki dyrektywom `ng-model` czy `ng-click`. Bezpośrednio jednak wywoływany jest najpierw `$apply()`, który to później wywołuje `$digest()`.

Może zaistnieć sytuacja, w której trzeba będzie wywołać `$apply()` manualnie. Angular zbudowany jest tak, by wychwytywać zmiany zachodzące między widokiem a modelem automatycznie, ale dzieje się to wyłącznie w obrębie jego kontekstu. W sytuacji, gdy zmiana modelu odbywa się poza kontekstem Angulara, należy go o niej poinformować, wywołując `$apply()` manualnie — to stąd Angular wie, że musi rozpocząć nasłuchiwanie. Nie powinniśmy nigdy bezpośrednio wywoływać `$digest()`. Prawidłowo powinniśmy wywołać `$apply()`, który później wykona cykl `$digest()`.

$apply()

Usługa `$apply` zachowuje się jak goniec wysyłany spoza kontekstu Angulara w celu poinformowania o zaistniałych zmianach. Innymi słowy, `$apply()` służy do integracji Angulara z innymi frameworkami bądź bibliotekami. `$apply()` zawiera funkcję pobieraną jako parametr, za której wykonanie odpowiada `$eval`. Do jego zadań należy również sprawdzenie, czy owa funkcja jest wykonywalna, oraz ewentualne poinformowanie Angulara o wykrytych nieścisłościach poprzez zwrócenie wyjątku. Wykorzystywana jest tu tzw. obsługa wyjątków z poziomu aplikacji (ang. *Application level error handling*). Jej wartość najczęściej doceniana jest wraz ze wzrostem poziomu skomplikowania aplikacji.

Następnie wywoływany jest cykl `$digest()`. Gdy mówimy o integracji z Angularem, mamy na myśli właśnie tę usługę: wystarczy otoczyć kod wewnątrz `$apply()` — prawda, że proste?

Wiesz już, jak działa `$apply()`, przejdźmy teraz do przykładu, który pokaże Ci jego zastosowanie praktyczne. Na pytanie, co stanie się w momencie uruchomienia poniższej strony, najlepiej odpowie listing 2.5:

Listing 2.5. *$watch*

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" ng-app="app">
<head>
    <title>AngularJS - $watch</title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.1/
    ↪css/bootstrap.min.css">
</head>
<body>
    <div ng-controller="defaultCtrl">
        <div class="well">Wiadomość: {{msg}}</div>
    </div>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/
    ↪1.4.0-beta.5/angular.min.js">
    </script>
    <script>
        var app = angular.module('app', []);
        app.controller('defaultCtrl', function ($scope) {
            $scope.go = function () {
                setTimeout(function () {
                    $scope.msg = 'Wow, jestem opóźnioną informacją!';
                    console.log('message:' + $scope.msg);
                }, 2000);
            }
            $scope.go();
        });
    </script>
</body>
</html>
```

Wynik wywołania powyższej strony będzie nie do końca zgodny z naszymi oczekiwaniami. Naszym celem było uaktualnienie w widoku `{{msg}}` po dwóch sekundach. Tak się jednak nie stało, mimo że teoretycznie program zadziałał i po dwóch sekundach w logu otrzymaliśmy oczekiwany tekst. Dlaczego widok nie został uaktualniony? Jak rozwiązać ten problem? Do tego posłuży nam `$apply()`. Przeanalizujmy teraz listing 2.6.

Listing 2.6. *$apply()*

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" ng-app="app">
<head>
    <title>AngularJS - $apply()</title>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/
    ↪3.3.1/css/bootstrap.min.css">
</head>
<body>
```

```
    <div ng-controller="defaultCtrl">
        <div class="well">Wiadomość: {{msg}}</div>
    </div>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/
    ↪1.4.0-beta.5 /angular.min.js">
    </script>
    <script>
        var app = angular.module('app', []);
        app.controller('defaultCtrl', function ($scope) {
            $scope.go = function () {
                setTimeout(function () {
                    $scope.msg = 'Wow, jestem opóźnioną informacją!';
                    console.log('message:' + $scope.msg);
                    $scope.$apply();
                }, 2000);
            }
            $scope.go();
        });
    </script>
</body>
</html>
```

Jak widać, dodaliśmy tylko `$scope.$apply()`, zmieniło to jednak zasadniczo działanie całej aplikacji. Tym razem otrzymaliśmy odpowiedni log oraz zmianę z dwusekundowym opóźnieniem po stronie widoku.

Najważniejsze przesłanie płynące z tej części rozdziału jest takie: wszędzie tam, gdzie AngularJS nie może wykryć zmian samodzielnie, musimy to zrobić ręcznie.

Quiz

1. Co to jest `$scope`?
2. Czym się różni `$scope` od `$rootScope`?
3. Co to jest drzewo DOM?
4. Jak stworzyć izolowany `scope`?
5. Co to są obiekty nasłuchujące?
6. Jak działa cykl `$digest`?
7. W jakich sytuacjach należy korzystać z usługi `$apply`?

Skorowidz

P

R

S

T

U

W

X

Z

ANGULARJS jest frameworkiem MVC napisanym w JavaScripcie. Ta stworzona przez Google'a technologia daje programistom ogromne możliwości, a w ciągu ostatnich dwóch lat dla wielu firm stała się standardem w tworzeniu aplikacji sieciowych. O jej popularności najlepiej świadczą coraz liczniejsze oferty pracy dla programistów AngularJS. Każdy, kto poważnie myśli o tworzeniu aplikacji dla sieci, powinien poznać cechy tego frameworka, jego zalety oraz ograniczenia.

Jeśli i Ty chciałbyś rozpocząć przygodę z AngularJS i w pełni korzystać z opcji oferowanych przez JavaScript, sięgnij po tę książkę! Dowiesz się z niej, jak szybko i sprawnie tworzyć dynamiczne, łatwe w utrzymaniu aplikacje internetowe działające po stronie klienta. Poznasz praktyczne przykłady, które pomogą Ci zrozumieć prezentowany materiał, a także nauczysz się samodzielnie pisać wydajny kod z wykorzystaniem AngularJS, zaś dzięki podsumowującym każdy rozdział pytaniom kontrolnym skutecznie utrwalisz zdobyte wiadomości.

- **Podstawy AngularJS i środowisko pracy dewelopera**
- **Wzorzec MVC w aplikacjach internetowych SPA**
- **Modularyzacja i wstrzykiwanie zależności**
- **Dyrektywy, filtry i funkcje**
- **Sposoby komunikacji z serwerem**
- **Dobre praktyki w stosowaniu AngularJS**
- **Testowanie aplikacji internetowych**

Naucz się obsługi nowego frameworka!

Dariusz Kalbarczyk — programista od ponad 15 lat związany z rynkiem IT. Współtwórca grupy AngularJS Warsaw, przez lata współpracujący z największymi firmami na rynku finansowym i telekomunikacyjnym (brał udział m.in. w tworzeniu opartego na najnowszych technologiach bankowego mobilnego systemu transakcyjnego).

Arkadiusz Kalbarczyk — student informatyki w Polsko-Japońskiej Akademii Technik Komputerowych. Swoje umiejętności miał okazję rozwijać w międzynarodowym środowisku: Kuwejcie oraz Irlandii. Współtwórca grupy AngularJS Warsaw.

Helion

30213 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

ISBN 978-83-283-0586-1

9 788328 305861

Informatyka w najlepszym wydaniu

cena: 39,90 zł