Zasięg i bindowanie

Oto bardzo prosta aplikacja angularowa:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
</head>
<body ng-app>
     <input type='text' ng-model='imie'>
     <div>Witaj, <span ng-bind='imie'></span>!</div>
</body>
</html>
```

Kiedy przeglądarka wyświetla tę stronę i kiedy załaduje angulara, angular znajduje w stronie element oznaczony atrybutem *ng-app* i opiekuje się tym elementem. Opieka polega na tym, że angular tworzy obiekt zwany zasięgiem. Następnie angular w treści strony znajduje wszystkie kontrolki formularza oznaczone atrybutem *ng-model* (u nas jest jedna – pole tekstowe) i wiąże je dwustronnie z odpowiednimi atrybutami zasięgu. W naszym przypadku: pole tekstowe zostaje powiązane z atrybutem *imie* zasięgu. Dwustronne powiązanie polega na tym, że angular obserwuje zmiany w modelu (czyli w atrybucie *imie* zasięgu) i jak model się zmieni, wpisuje aktualną wartość tego atrybutu w odpowiednie miejsce widoku (u nas: w pole tekstowe). I w drugą stronę: jeśli zmieni się wartość wpisana w pole tekstowe, angular przepisze ją w atrybut *imie* zasięgu. Następnie angular wyszukuje wszystkie elementy na stronie oznaczone atrybutem *ng-bind* i wiąże je jednostronnie z modelem. Takie powiązanie jednostronne polega na tym, że kiedy zmieni się wartość w modelu, zostanie ona wpisana w treść tego elementu.

Możemy mieć wiele elementów powiązanych z tym samym atrybutem zasięgu. Możemy w zasięgu mieć wiele atrybutów:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
</head>
<body ng-app>
     <h1>na tej stronie witany jest <span ng-bind='imie'></span>
co ma <span ng-bind='wiek'></span> lat</h1>
     <input type='text' ng-model='imie'>
     <input type='text' ng-model='imie'>
     <div>Witaj, <span ng-bind='imie'></span>!</div>
     ile masz lat? <input type='text' ng-model='wiek'><br>
     <select ng-model="wiek">
          <option value="12">12</option>
          <option value="21">21</option>
          <option value="65">65</option>
     </select><br>
     12: <input type="radio" name="wiek" value="12" ng-
model="wiek"><br>
     21: <input type="radio" name="wiek" value="21" ng-
model="wiek"><br>
     65: <input type="radio" name="wiek" value="65" ng-
model="wiek"><br>
</body>
</html>
```

Przy bindowaniu jednostronnym możemy podawać bardziej wyrafinowane wyrażenia. Nie mogą to być dowolne wyrażenia javaskryptowe – angular ma własny język wyrażeń. Te wyrażenia są ewaluowane w kontekście zasięgu:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
</head>
<body ng-app>
     <label>waga: <input type='text' ng-model='waga'></label><br>
     <label>wzrost: <input type='text' ng-
model='wzrost'></label><br>
     twoje bmi wynosi <span ng-bind='waga / ((wzrost / 100) *
(wzrost / 100))' />
</body>
</html>
```

To, co zrobiliśmy, nie jest eleganckie, bo umieściliśmy logikę aplikacji w szablonie. Niedługo zobaczymy, jak robić to bardziej elegancko.

W wyrażeniach angularowych można używać operatora |. Służy on do przepuszczania wartości przez filtry. Używa się go tak:

```
wyrażenie | nazwafiltra[:argument]
```

Przykładowo:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
</head>
<body ng-app>
     <label>waga: <input type='text' ng-model='waga'></label><br>
     <label>wzrost: <input type='text' ng-
model='wzrost'></label><br>
     twoje bmi wynosi <span ng-bind='(waga / ((wzrost / 100) *
(wzrost / 100))) | number:2' ></span>
</body>
</html>
```

Listę filtrów, które domyślnie mamy w angularze, znajdziemy w dokumentacji: *https://docs.angularjs.org/api/ng/filter*

Wyrażenia korzystające z filtrów mogą być – jak każde wyrażenia – częścią wyrażeń:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
</head>
<body ng-app>
     <input type='text' ng-model='napis'><br>
     <span ng-bind='(napis | lowercase) + (napis | uppercase)'
></span>
</body>
</html>
```

Wiązanie jednostronne można też robić przy użyciu podwójnych nawiasów klamrowych:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
</head>
<body ng-app>
     wiek: <input type='text' ng-model='wiek'><br>
     Do emerytury zostało ci <span>{{ 67 - wiek }}</span> lat.
</body>
</html>
```

Takie podwójne klamry mogą być też wrzucone w środek innego tekstu:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
</head>
<body ng-app>
     wiek: <input type='text' ng-model='wiek'><br>
     Do emerytury zostało ci {{ 67 - wiek }} lat.
</body>
</html>
```

Kontrolery

Jeśli chcemy, żeby nowotworzony zasięg był jakoś przygotowywany (na przykład żeby były w nim umieszczane jakieś funkcje), tworzymy kontrolera. W tym celu najpierw rejestrujemy moduł, a potem w tym module tworzymy kontrolera. Moduł jest pojemnikiem do przechowywania kontrolerów i innych komponentów.

Kontroler jest funkcją. Kiedy angular będzie tworzył nowy zasięg, uruchomi kontrolera. Jeśli poprosimy, angular naszemu kontrolerowi przekaże zasięg. Proszenie polega na zadeklarowaniu, że nasza funkcja przyjmuje parametr o nazwie *$scope*. Kiedy angular uruchamia kontrolera, sprawdza, jak nazywają się jego parametry i w zależności od ich nazw przekazuje w nich odpowiednie rzeczy.

Oto prosty przykład kontrolera:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope) {
          $scope.wiek = 0;
          $scope.doEmerytury = function () {
               return 67 - $scope.wiek;
          }
     });
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
          wiek: <input type='text' ng-model='wiek'><br>
          Do emerytury zostało ci {{ doEmerytury() }} lat.
     </div>
</body>
</html>
```

Jak widać, dzięki kontrolerowi możemy wynieść logikę aplikacji z widoku.

Atrybut *ng-click* pozwala określić, co ma się dziać po kliknięciu na element:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope) {
          $scope.punkty = 0;
          $scope.zwieksz = function () {
               ++$scope.punkty;
          };
     });
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
          score: {{ punkty }}<br>
          <button ng-click='zwieksz()'>kliknij mnie</button>
     </div>
</body>
</html>
```

Jeśli zamiast *ng-click* użyjemy *ng-dblclick*, funkcja będzie wykonywana w reakcji na podwójne kliknięcie:

```
(...)
<button ng-dblclick='zwieksz()'>kliknij mnie</button>
(...)
```

Listę takich atrybutów pozwalających obsługiwać różne zdarzenia możemy znaleźć w liście wszystkich dyrektyw, pod adresem *https://docs.angularjs.org/api/ng*

wstrzykiwanie zależności

Jak widzieliśmy, kontroler może poprosić o wstrzyknięcie w niego zależności – na przykład zależności *$scope*, zawierającej zasięg.

Jednym ze sposobów na poproszenie o zależność jest – jak widzieliśmy – zadeklarowanie w kontrolerze parametru o odpowiedniej nazwie. Dla przypomnienia, oto przykład:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script
src='//ajax.googleapis.com/ajax/libs/angularjs/1.2.22/angular.min.
js'></script>
    <script>
angular.module('MojaAplikacja', [])
    .controller('MojKontroler', function ($scope) {
        $scope.komunikat = 'ala ma asa';
    });
    </script>
</head>
<body data-ng-app='MojaAplikacja'>
     <div data-ng-controller='MojKontroler'>
        {{ komunikat }}
     </div>
</body>
</html>
```

Ten sposób ma tę wadę, że jeśli z jakiegoś powodu zmienimy nazwę parametru (na przykład bo zminifikujemy nasz kod javaskryptowy), zależność nie będzie wstrzyknięta. Dlatego są jeszcze dwa inne sposoby.

Możemy w atrybucie *$inject* funkcji (bo pamiętajmy, że funkcja jest obiektem) umieścić tablicę z nazwami zależności, które mają być wstrzykiwane w kolejne parametry. Oto przykład:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script
src='//ajax.googleapis.com/ajax/libs/angularjs/1.2.22/angular.min.
js'></script>
    <script>
function kontroler(s) {
    s.komunikat = 'ola ma asa';
};
kontroler['$inject'] = ['$scope'];
angular.module('MojaAplikacja', [])
    .controller('MojKontroler', kontroler);
    </script>
</head>
<body data-ng-app='MojaAplikacja'>
     <div data-ng-controller='MojKontroler'>
        {{ komunikat }}
     </div>
</body>
</html>
```

Niestety, przy stosowaniu tego sposobu piszemy rozwlekły kod – musimy umieścić konstruktor w tymczasowej zmiennej. Tej wady nie ma trzeci sposób. Metodzie *controller* możemy w drugim parametrze przekazać nie funkcję, a tablicę zawierającą listę nazw potrzebnych zależności oraz – na końcu – funkcję. Oto przykład:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script
src='//ajax.googleapis.com/ajax/libs/angularjs/1.2.22/angular.min.
js'></script>
    <script>
angular.module('MojaAplikacja', [])
    .controller('MojKontroler', ['$scope', function (s) {
        s.komunikat = 'ula ma asa';
    }]);
    </script>
</head>
<body data-ng-app='MojaAplikacja'>
     <div data-ng-controller='MojKontroler'>
        {{ komunikat }}
     </div>
</body>
</html>
```

Jeżeli chcemy dostać więcej zależności, po prostu podajemy w tablicy więcej nazw. Oto przykład (korzystający z serwisu *$http*, który poznamy później):

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script
src='//ajax.googleapis.com/ajax/libs/angularjs/1.2.22/angular.min.
js'></script>
    <script>
angular.module('MojaAplikacja', [])
    .controller('MojKontroler', ['$scope', '$http', function (s,
h) {
        s.komunikat = 'ula ma asa';
        console.log(h);
    }]);
    </script>
</head>
<body data-ng-app='MojaAplikacja'>
     <div data-ng-controller='MojKontroler'>
        {{ komunikat }}
     </div>
</body>
</html>
```

zagnieżdżanie zasięgów

Jeśli elementy oznaczone *ng-controller* poumieszczamy jedne w drugich, ich zasięgi będą połączone ze sobą – jedne zasięgi będą prototypami innych. To znaczy, że z wewnętrznych zasięgów możemy zaglądać do zewnętrznych, ale z zewnętrznych do wewnętrznych nie zajrzymy:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('KontrolerOjciec', function () {})
     .controller('KontrolerSyn', function () {});
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='KontrolerOjciec'>
          <h1>miasto: {{ miasto }}</h1>
          kraj: <input type='text' ng-model='kraj'>
          <div ng-controller='KontrolerSyn'>
               <h2>kraj: {{ kraj }}</h2>
               miasto: <input type='text' ng-model='miasto'>
          </div>
     </div>
</body>
</html>
```

Takie zaglądanie działa też w javaskrypcie – i też tylko w jedną stronę:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('KontrolerOjciec', function ($scope) {
          $scope.f = function () {
               console.log($scope.kraj, $scope.miasto);
          };
     })
     .controller('KontrolerSyn', function ($scope) {
          $scope.g = function () {
               console.log($scope.kraj, $scope.miasto);
          };
     });
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='KontrolerOjciec'>
          kraj: <input type='text' ng-model='kraj'><br>
          <button ng-click='f()'>kliknij mnie</button>
          <div ng-controller='KontrolerOjciec'>
               miasto: <input type='text' ng-model='miasto'><br>
               <button ng-click='f()'>kliknij mnie</button>
          </div>
     </div>
</body>
</html>
```

Nie zajrzymy też do zasięgu, który jest rodzeństwem:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('KontrolerOjciec', function () {})
     .controller('KontrolerCorka', function () {})
     .controller('KontrolerSyn', function () {});
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='KontrolerOjciec'>
          <h1>kraj: {{ kraj }}</h1>
          <h1>miasto: {{ miasto }}</h1>
          <h1>rzeka: {{ rzeka }}</h1>
          kraj: <input type='text' ng-model='kraj'>
          <div ng-controller='KontrolerSyn'>
               <h2>kraj: {{ kraj }}</h2>
               <h2>miasto: {{ miasto }}</h2>
               <h2>rzeka: {{ rzeka }}</h2>
               miasto: <input type='text' ng-model='miasto'>
          </div>
          <div ng-controller='KontrolerCorka'>
               <h2>kraj: {{ kraj }}</h2>
               <h2>miasto: {{ miasto }}</h2>
               <h2>rzeka: {{ rzeka }}</h2>
               rzeka: <input type='text' ng-model='rzeka'>
          </div>
     </div>
</body>
</html>
```

wysyłanie komunikatów w górę i w dół

Z każdego zasięgu możemy metodą *$broadcast* wysłać komunikat. Taki komunikat zostanie dostarczony do tego zasięgu, który go wysłał, i do jego potomków. Każdy komunikat ma nazwę. W każdym zasięg możemy zarejestrować, jaka ma być reakcja na komunikat o danej nazwie:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope) {
          $scope.komunikat = '...';
          $scope.klik = function () {
               $scope.$broadcast('lufcik');
          };
          $scope.$on('lufcik', function () {
               $scope.komunikat = 'dostałem komunikat';
          });
     })
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
          <p>{{ komunikat }}</p>
          <button ng-click='klik()'>broadcast</button>
          <div ng-controller='MojKontroler'>
               <p>{{ komunikat }}</p>
               <button ng-click='klik()'>broadcast</button>
               <div ng-controller='MojKontroler'>
                    <p>{{ komunikat }}</p>
                    <button ng-click='klik()'>broadcast</button>
               </div>
          </div>
     </div>
</body>
</html>
```

Komunikaty można też wysyłać metodą *$emit*. Ona wysyła komunikat w drugą stronę: do zasięgu, który wysłał komunikat, i do wszystkich jego przodków:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope) {
          $scope.komunikat = '...';
          $scope.klik = function () {
               $scope.$emit('lufcik');
          };
          $scope.$on('lufcik', function () {
               $scope.komunikat = 'dostałem komunikat';
          });
     })
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
          <p>{{ komunikat }}</p>
          <button ng-click='klik()'>broadcast</button>
          <div ng-controller='MojKontroler'>
               <p>{{ komunikat }}</p>
               <button ng-click='klik()'>broadcast</button>
               <div ng-controller='MojKontroler'>
                    <p>{{ komunikat }}</p>
                    <button ng-click='klik()'>broadcast</button>
               </div>
          </div>
     </div>
</body>
</html>
```

Kiedy zasięg dostanie komunikat, angular uruchamia funkcje zarejestrowane (metodą *$on*) jako słuchacze tego komunikatu. Przekazuje im przy tym obiekt z różnymi informacjami o tym komunikacie. Ten obiekt ma między innymi metodę *stopPropagation*. Kiedy któryś z zasięgów wywoła tę metodę, angular nie będzie już przekazywał tego komunikatu kolejnym zasięgom:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope) {
          $scope.komunikat = '...';
          $scope.klik = function () {
               $scope.$emit('lufcik');
          };
          $scope.$on('lufcik', function (message) {
               $scope.komunikat = 'dostałem komunikat';
               message.stopPropagation();
          });
     })
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
          <p>{{ komunikat }}</p>
          <button ng-click='klik()'>broadcast</button>
          <div ng-controller='MojKontroler'>
               <p>{{ komunikat }}</p>
               <button ng-click='klik()'>broadcast</button>
               <div ng-controller='MojKontroler'>
                    <p>{{ komunikat }}</p>
                    <button ng-click='klik()'>broadcast</button>
               </div>
          </div>
     </div>
</body>
</html>
```

Do wysyłanego komunikatu można dołączyć dodatkowe dowolne dane. W tym celu przy wywoływaniu metody *broadcast* albo *emit* przekazujemy dodatkowy parametr z dowolnym obiektem. Te dane dostanie funkcja zarejestrowana metodą *$scope.$on* w drugim parametrze:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope) {
          $scope.komunikat = '...';
          $scope.klik = function () {
               $scope.$emit('lufcik', 'rachu-ciachu!');
          };
          $scope.$on('lufcik', function (message, dane) {
               $scope.komunikat = 'dostałem komunikat z danymi: ' +
dane;
          });
     })
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
          <p>{{ komunikat }}</p>
          <button ng-click='klik()'>broadcast</button>
          <div ng-controller='MojKontroler'>
               <p>{{ komunikat }}</p>
               <button ng-click='klik()'>broadcast</button>
               <div ng-controller='MojKontroler'>
                    <p>{{ komunikat }}</p>
                    <button ng-click='klik()'>broadcast</button>
               </div>
          </div>
     </div>
</body>
</html>
```

wysyłanie komunikatów do wszystkich

Czasem chcemy wysłać komunikat do wszystkich zasięgów. Są na to dwa sposoby. Oba korzystają z faktu, że każdy zasięg ma atrybut *$rootScope*, w którym jest zasięg-korzeń (zasięg będący przodkiem wszystkich innych):

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope, $rootScope) {
          $scope.klik = function () {
               $rootScope.komunikat = 'pająk';
          };
     })
     </script>
</head>
<body ng-app='MojModul'>
     <p>{{ komunikat }}</p>
     <div ng-controller='MojKontroler'>
          <button ng-click='klik()'>broadcast</button>
     </div>
</body>
</html>
```

Pierwszy sposób polega na tym, że używamy zasięgu-korzenia do wysłania komunikatu do niego samego i wszystkich jego dzieci:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($rootScope, $scope) {
          $scope.komunikat = '...';
          $scope.klik = function () {
               $rootScope.$broadcast('lufcik');
          };
          $scope.$on('lufcik', function (message) {
               $scope.komunikat = 'dostałem komunikat';
          });
     })
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
          <p>{{ komunikat }}</p>
          <button ng-click='klik()'>broadcast</button>
          <div ng-controller='MojKontroler'>
               <p>{{ komunikat }}</p>
               <button ng-click='klik()'>broadcast</button>
          </div>
          <div ng-controller='MojKontroler'>
               <p>{{ komunikat }}</p>
               <button ng-click='klik()'>broadcast</button>
          </div>
     </div>
</body>
</html>
```

Kiedyś, w dawnych wersjach angulara ten sposób był mało wydajny. Wtedy czasem stosowano inny sposób: używano metody *emit* zasięgu-korzenia do wysłania komunikatu, który zostanie dostarczony tylko do tego zasięgu-korzenia, i używano metody *$on* zasięgu-korzenia do zarejestrowania słuchacza, który będzie powiadomiony o tym komunikacie. Przy stosowaniu tego sposobu trzeba było zadbać o to, żeby odrejestrowywać słuchacza, kiedy zasięg, który go zarejestrował, jest niszczony. Przez to ten sposób jest mniej wygodny, a ponieważ współcześnie nie ma powodu, żeby go stosować, nie podaję tu jego przykładu.

Używanie wyrażeń w różnych miejscach HTML-a

źródło obrazka

Możemy użyć wyrażenia angularowego w atrybucie *src* obrazka:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
</head>
<body ng-app>
     <input type='number' min='1' max='6' ng-model='liczba'><br>
     <img src='http://students.alx.pl/zasoby/kosci/
{{ liczba }}.png'>
</body>
</html>
```

To działa, ale ma jedną wadę. Kiedy przeglądarka załaduje stronę, zanim wystartuje angular, przeglądarka spróbuje załadować z serwera zasób *http://students.alx.pl/zasoby/kosci/{{ liczba }}.png*. W ten sposób wysyłamy niepotrzebnie jedno żądanie.

Rozwiązaniem tego problemu jest użycie angularowej dyrektywy *ng-src*. Jest to dyrektywa, która obserwuje dane wyrażenie i zawsze, kiedy jego wartość się zmieni, wpisuje jego wartość w atrybut *src* danego elementu:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
</head>
<body ng-app>
     <input type='number' min='1' max='6' ng-model='liczba'><br>
     <img ng-src='http://students.alx.pl/zasoby/kosci/
{{ liczba }}.png'>
</body>
</html>
```

klasy elementów

Możemy używać wyrażeń angularowych w hateemelowym atrybucie *class*:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <style>
.duza {
     color: red;
}
.mala {
     color: green;
}
     </style>
</head>
<body ng-app>
     <input type='number' ng-model='liczba'><br>
     <p class='{{ liczba > 10 ? "duza" : "mala" }}'>Czarna krowa w
kropki bordo gryzła trawę kręcąc mordą.</p>
</body>
</html>
```

Ale takie i podobne efekty można uzyskiwać prościej, używając dyrektywy *ng-class*:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <style>
.liczba {
     color: red;
}
.mala {
     color: green;
}
.parzysta {
     background: yellow;
}
     </style>
</head>
<body ng-app>
     <input type='number' ng-model='liczba'><br>
     <p class='liczba' ng-class='{mala: liczba < 10, parzysta:
liczba % 2 == 0}'>Czarna krowa w kropki bordo gryzła trawę kręcąc
mordą.</p>
</body>
</html>
```

ukrywanie i pokazywanie elementów

Dyrektywy *ng-show* i *ng-hide* pozwalają ukrywać i odkrywać elementy, kiedy wartość danego wyrażenia stanie się prawdą.

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
</head>
<body ng-app>
     <input type='number' ng-model='liczba'><br>
     <p ng-show='liczba % 2 == 0'>Czarna krowa w kropki bordo</p>
     <p ng-hide='liczba % 2 == 0'>gryzła trawę kręcąc mordą.</p>
</body>
</html>
```

Powtarzanie fragmentów szablonu

W angularze mamy dyrektywę *ng-repeat*. Pozwala ona powtórzyć dany element wiele razy. Używa się jej tak:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
</head>
<body ng-app>
     <ul>
          <li ng-repeat='liczba in [3, 5, 7]'>{{ liczba }}</li>
     </ul>
</body>
</html>
```

Ta dyrektywa powtarza zadany element wiele razy. Dla każdego z nich tworzy osobny zasięg będący dzieckiem tego zasięgu, w którym wydaliśmy polecenie *ng-repeat*. W tym zasięgu tworzona jest zmienna, której nazwę wymyśliliśmy sami i podaliśmy w dyrektywie *ng-repeat* (w naszym przykładzie jest to zmienna liczba). W tej zmiennej umieszczane są kolejne elementy tablicy. Oprócz tej zmiennej w tym zasięgu tworzone są też inne zmienne (opisane na *https://docs.angularjs.org/api/ng/directive/ngRepeat*), między innymi:

- *$index* – numer iteracji,
- *$first* – czy to jest pierwsza iteracja,
- *$last* – czy to jest ostatnia iteracja.

Oto przykład:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
</head>
<body ng-app>
     <ul>
          <li ng-repeat='liczba in [3, 5, 7]'>{{ $index }} &ndash;
{{ liczba }}</li>
     </ul>
</body>
</html>
```

powtarzanie kilku elementów

Żeby powtórzyć kilka elementów, używamy dyrektyw *ng-repeat-start* i *ng-repeat-end*:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
</head>
<body ng-app>
     <h1 ng-repeat-start='liczba in [3, 5, 7]'>{{ liczba }}</h1>
     <p>jej kwadrat to {{ liczba * liczba }}</p>
     <p>jej sześcian to {{ liczba * liczba * liczba }}</p>
     <p ng-repeat-end>jej kwadrat ma {{ (liczba * liczba +
'').length }} cyfr</p>
</body>
</html>
```

zagnieżdżanie pętli

Pętle tworzone dyrektywą *ng-repeat* można zagnieżdżać:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
</head>
<body ng-app>
     <div ng-repeat='dzielna in [3, 5, 7]'>
          <div ng-repeat='dzielnik in [2, 22, 222]'>
               <p>{{ dzielna }} / {{ dzielnik }} = {{ dzielna /
dzielnik | number:2 }}</p>
          </div>
     </div>
</body>
</html>
```

Jeśli w takiej zagnieżdżonej pętli zechcemy korzystać ze zmiennej *$index* (lub którejś innej zmiennej tworzonej przez *ng-repeat*), może się przydać dyrektywa *ng-init*. Pozwala ona stworzyć w bieżącym zasięgu nową zmienną i wpisać do niej wynik wyrażenia:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
</head>
<body ng-app>
     <div ng-repeat='dzielna in [3, 5, 7]' ng-init='i1 = $index +
1'>
          <div ng-repeat='dzielnik in [2, 22, 222]' ng-init='i2 =
$index + 1'>
               <p>Punkt {{ i1 }}.{{ i2 }}: {{ dzielna }} /
{{ dzielnik }} = {{ dzielna / dzielnik | number:2 }}</p>
          </div>
     </div>
</body>
</html>
```

Jak działa bindowanie

Angularowe bindowanie korzysta z niżejpoziomowych mechanizmów.

niżejpoziomowe mechanizmy, z których korzysta bindowanie

Każdy zasięg przechowuje (w atrybucie *$$watchers*) listę obserwatorów, którzy mają być powiadomieni, kiedy wartość jakiegoś wyrażenia angularowego albo jakiejś funkcji zmieni swoją wartość. Wygodnie jest oglądać, jak to działa, w konsoli Firebuga. W tym celu najpierw piszemy kontroler, który umieści referencję do zasięgu w zmiennej globalnej:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope) {
          window.$scope = $scope;
     });
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
     </div>
</body>
</html>
```

Teraz możemy obejrzeć listę watcherów wydając w firebugowej konsoli polecenie:

```
$scope.$$watchers
```

Prawdopodobnie teraz ta lista jest pusta. Dodamy do niej watchera pilnującego, kiedy zmieni się zasięgowy atrybut *imie*:

```
$scope.$watch('imie', function () { alert('imię zmieniło się na '
+ $scope.imie); })
```

Spróbujmy zmienić zawartość zasięgowego atrybutu *imie* i zobaczmy, czy coś uruchomi naszą funkcję:

```
$scope.imie = 'Mirek'
```

Jak widać, nic nie uruchomiło naszej funkcji. Co nie jest takie dziwne – bo jak twórcy angulara mieliby zrobić, żeby zawsze, kiedy dane wyrażenie zmieni swoją wartość, uruchamiany był zadany kod? Uruchamianie kilka razy na sekundę funkcji, która sprawdzałaby wszystkie obserwowane wyrażenia, byłoby bardzo niewydajne. W prostych przypadkach można by użyć *Object.observe*, ale nadal jeszcze mało przeglądarek je obsługuje[1].

1 To zdanie napisałem 25 sierpnia 2012. Źródło: *http://kangax.github.io/compat-table/es7/#Object.observe*

Twórcy angulara zrobili inaczej. Zasięg ma metodę *$digest*. Ta metoda przegląda wszystkie watchery, dla każdego sprawdza, czy jego wartość się zmieniła (każdy watcher pamięta ostatnio widzianą wartość swojego wyrażenia) i jeśli się zmieniła, uruchamia jego funkcję. Spróbujmy więc wywołać *$digest*:

```
$scope.$digest()
```

Po wydaniu tego polecenia powinniśmy zobaczyć alerta z komunikatem *imię zmieniło się na Mirek*.

Każdy zasięg ma też metodę *$apply*. Przekazujemy jej pewną funkcję, a ona uruchamia tę funkcję, po czym wywołuje *$digest*. Możemy więc też zmienić imię w taki sposób:

```
$scope.$apply(function () { $scope.imie = 'Juliusz'; })
```

Od razu po wydaniu tego polecenia powinniśmy zobaczyć alerta z komunikatem *imię zmieniło się na Juliusz*.

jak działa przenoszenie zmian w modelu do widoku

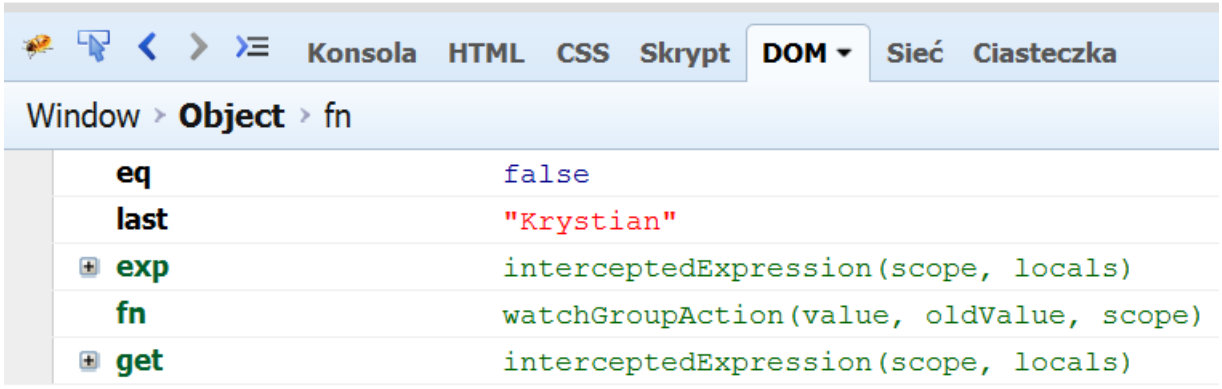
Przyjrzyjmy się takiej angularowej aplikacji:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope) {
          window.$scope = $scope;
          $scope.imie = 'Krystian';
     });
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
          Witaj <span>{{ imie }}</span>!
     </div>
</body>
</html>
```

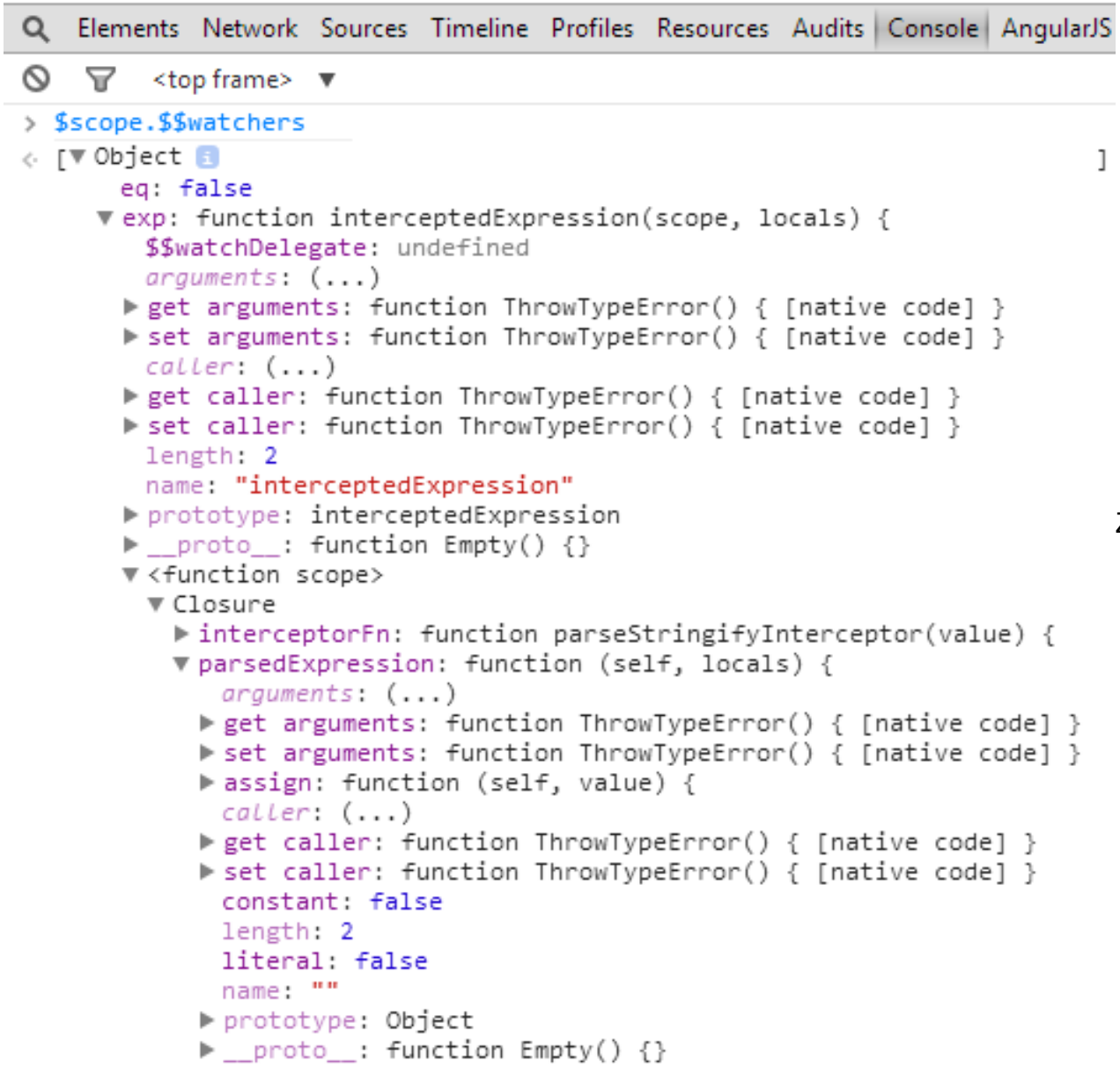
W firebugowej konsoli obejrzyjmy, jakie watchery są ustawione w tym zasięgu:

```
$scope.$$watchers
```

Jak widać, angular stworzył watchera. Ten watcher obserwuje wyrażenie *imie*. Niestety, kiedy oglądamy tego watchera w firebugu, nigdzie nie widać, jakie wyrażenie on obserwuje, widać tylko jego ostatnią zaobserwowaną wartość (i stąd możemy wywnioskować, że to jest właśnie interesujący nas watcher od wyrażenia *imie*):



To dlatego, że obserwowane wyrażenie jest przechowywane w domknięciu pewnej funkcji, a Firebug nie pokazuje domknięć funkcji. Pokazuje je panel narzędzi deweloperskich w Chromie. Tam możemy obejrzeć watchera, zajrzeć w pierwsze domknięcie funkcji przechowywanej w jego atrybucie *exp*, tam znaleźć atrybut *parsedException*, w którym jest funkcja, obejrzeć jej pierwsze domknięcie i w nim znaleźć zmienną *ident*, w której jest napis *imie*:



Z

To ten właśnie watcher pilnuje zmian w modelu i kiedy trzeba zmienia treść wyświetlaną w widoku. Możemy zresztą spróbować. Najpierw wydajmy w konsoli firebuga polecenie:

```
$scope.$apply(function () { $scope.imie = 'Juliusz'; })
```

Zauważymy, że imię wyświetlane w widoku zmieni się. A teraz skasujmy watchera i spróbujmy ponownie zmienić imię:

```
$scope.$$watchers[0] = undefined
$scope.$apply(function () { $scope.imie = 'Damian'; })
```

Tym razem zmiana nie będzie widoczna w widoku.

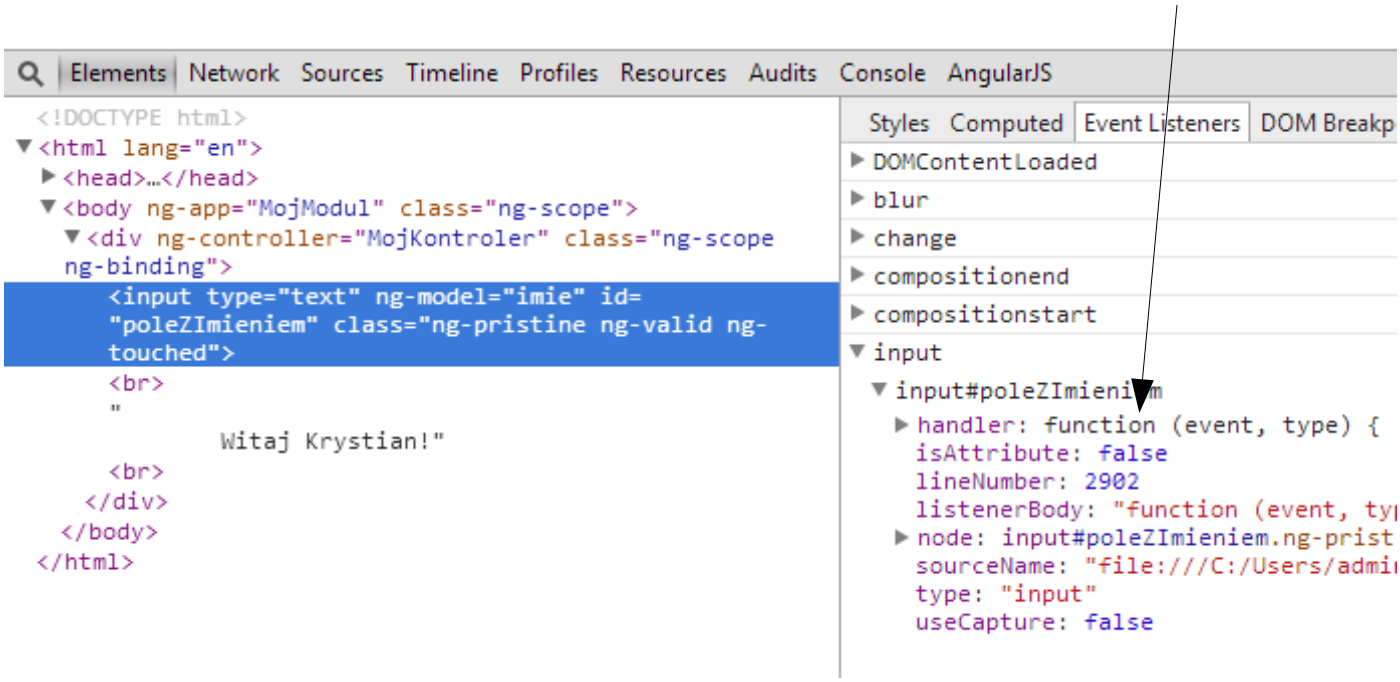
jak działa przenoszenie zmian w widoku do modelu

A teraz przyjrzyjmy się prostej aplikacji, w której będzie bindowanie w dwie strony:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope) {
          $scope.imie = 'Krystian';
     });
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
          <input type='text' ng-model='imie'
id='poleZImieniem'><br>
          Witaj {{ imie }}!<br>
     </div>
</body>
</html>
```

Warto rozumieć, jak się dzieje, że po wpisaniu imienia w pole tekstowe jest ono wpisywane do modelu i do spana w widoku.

Obejrzyjmy tę aplikację w Chromie. W panelu narzędzi deweloperskich możemy znaleźć funkcję przyczepioną do zdarzenia *input* pola tekstowego:



Tę funkcję przyczepił do pola tekstowego angular. Ta funkcja po każdej zmianie treści wpisanej w pole tekstowe w metodzie *$apply* wykonuje funkcję, która aktualizuje model. Żeby przekonać się, że rzeczywiście to ta funkcja zajmuje się tym, możemy obejrzeć jej źródła, ale nie są one łatwe do czytania. Możemy też spróbować odczepić tę funkcję od tego zdarzenia. W tym celu klikamy prawym przyciskiem na funkcji i wybieramy *store as global variable*:

W konsoli pojawia się globalna zmienna *temp1*. Teraz możemy odczepić tę funkcję od zdarzenia poleceniem:

```
document.getElementById('poleZImieniem').removeEventListener('input', temp1)
```

Teraz można się przekonać, że kiedy będziemy wpisywali tekst w pole tekstowe, nie będzie on przepisywany do widoku.

kiedy to ma znaczenie

Wyobraźmy sobie aplikację, w której możemy wystawiać i kasować faktury. Po wykonaniu każdego działania wyświetlany jest komunikat:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope) {
          $scope.komunikat = '';
          $scope.wystawFakture = function() {
               // ...
               $scope.komunikat = 'faktura została wystawiona';
          };
          $scope.skasujFakture = function() {
               // ...
               $scope.komunikat = 'faktura została skasowana';
          };
     });
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
          <p ng-hide='komunikat == ""' style='border: 1px solid
black; background: #ffa'>{{ komunikat }}</p>
          <button ng-click='wystawFakture()'>wystaw
fakturę</button><br>
          <button ng-click='skasujFakture()'>skasuj
fakturę</button><br>
     </div>
</body>
</html>
```

Wyobraźmy sobie, że chcemy, żeby te komunikaty znikały po dwu sekundach. Możemy spróbować zrobić to tak:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope) {
          $scope.komunikat = '';
          $scope.ustawKomunikat = function (komunikat) {
               $scope.komunikat = komunikat;
               setTimeout(function () {
                    $scope.komunikat = '';
               }, 2000);
          };
          $scope.wystawFakture = function() {
               $scope.ustawKomunikat('faktura została wystawiona');
          };
          $scope.skasujFakture = function() {
               // ...
               $scope.ustawKomunikat('faktura została skasowana');
          };
     });
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
          <p ng-hide='komunikat == ""' style='border: 1px solid
black; background: #ffa'>{{ komunikat }}</p>
          <button ng-click='wystawFakture()'>wystaw
fakturę</button><br>
          <button ng-click='skasujFakture()'>skasuj
fakturę</button><br>
     </div>
</body>
</html>
```

Ale to nie zadziała. Bo angular zauważa tylko te zmiany w modelach, po których na zasięgu została wywołana metoda *$digest*. Możemy przerobić nasz przykład tak, żeby czyszczenie komunikatu było robione w metodzie *$apply* – to spowoduje, że przykład zacznie działać:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope) {
          $scope.komunikat = '';
          $scope.ustawKomunikat = function (komunikat) {
               $scope.komunikat = komunikat;
               setTimeout(function () {
                    $scope.$apply(function () {
                         $scope.komunikat = '';
                    });
               }, 2000);
          };
          $scope.wystawFakture = function() {
               $scope.ustawKomunikat('faktura została wystawiona');
          };
          $scope.skasujFakture = function() {
               // ...
               $scope.ustawKomunikat('faktura została skasowana');
          };
     });
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
          <p ng-hide='komunikat == ""' style='border: 1px solid
black; background: #ffa'>{{ komunikat }}</p>
          <button ng-click='wystawFakture()'>wystaw
fakturę</button><br>
          <button ng-click='skasujFakture()'>skasuj
fakturę</button><br>
     </div>
</body>
</html>
```

Do odkładania na przyszłość zmian w modelach można też użyć serwisu *$timeout* (*https://docs.angularjs.org/api/ng/service/$timeout*). Jest to funkcja, która opakowuje *setTimeout* i sama dba o wywołanie *$apply*:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope, $timeout) {
          $scope.komunikat = '';
          $scope.ustawKomunikat = function (komunikat) {
               $scope.komunikat = komunikat;
               $timeout(function () {
                    $scope.komunikat = '';
               }, 2000);
          };
          $scope.wystawFakture = function() {
               $scope.ustawKomunikat('faktura została wystawiona');
          };
          $scope.skasujFakture = function() {
               // ...
               $scope.ustawKomunikat('faktura została skasowana');
          };
     });
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
          <p ng-hide='komunikat == ""' style='border: 1px solid
black; background: #ffa'>{{ komunikat }}</p>
          <button ng-click='wystawFakture()'>wystaw
fakturę</button><br>
          <button ng-click='skasujFakture()'>skasuj
fakturę</button><br>
     </div>
</body>
</html>
```

Deferredy i promisy

Deferred i promisa to para obiektów, które służą do przesłania wartości (liczby, napisu, dowolnego obiektu itp.) między dwoma miejscami programu. Deferred to dziura, do której można wrzucić wartość. Promisa to dziura, z której ta wartość wypadnie. Wiele bibliotek javaskryptowych ma mechanizmy do tworzenia deferredów i promis – ma go i angular. Do tworzenia deferredów i promis służy w angularze serwis *$q*. Są dwa sposoby na korzystanie z deferredów – różnią się one sposobem, w jaki wrzuca się wartość w deferreda.

Proste eksperymenty z deferredami wygodnie jest robić w interaktywnej konsoli Firebuga. Żeby mieć w niej dostęp do serwisu *$q*, zrobimy kontrolera kopiującego referencję do $q do zmiennej globalnej:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($q) {
          window.$q = $q;
     });
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
     </div>
</body>
</html>
```

sposób pierwszy, korzystający z metody *resolve*

Do stworzenia pary obiektów (deferred, promisa) służy metoda *$q.defer()*:

```
$q.defer()
```

Zwraca ona deferreda. Sparowaną z nią promisę możemy pobrać z jego atrybutu *promise*:

```
p = d.promise
```

Żeby dowiedzieć się, kiedy z promisy wypadnie wartość, rejestrujemy słuchacza:

```
p.then(function (wynik) { alert('mamy wynik: ' + wynik); })
```

Żeby wrzucić wartość do deferreda, wołamy jego metodę *resolve*:

```
d.resolve(1337)
```

Po wykonaniu tego polecenia powinniśmy zobaczyć, że wykonał się słuchacz zarejestrowany na promisie – powinien pojawić się alert z komunikatem *mamy wynik: 1337*.

Promisa, która raz dostała wartość, pamięta ją. Jeśli spróbujemy zarejestrować słuchacza na promisie połączonej ze zresolwerowanym deferredem, ten słuchacz zostanie wywołany od razu. Możemy się o tym przekonać robiąc:

```
p.then(function (wynik) { alert('drugi słuchacz dostał wynik: ' +
wynik); })
```

Od razu po wykonaniu tego polecenia pojawi się alert z komunikatem *drugi słuchacz dostał wynik: 1337*.

Więc para metod *resolve, then* nadaje się do przekazania w inne miejsce programu wyniku jakiegoś działania, które zostanie wykonane raz. Jeśli chcemy powiadomić, że działania nie udało się wykonać, służy do tego metoda *reject*. Żeby dostać ewentualnie powiadomienie o zredżektowaniu deferreda, rejestrujemy na promisie słuchacza metodą *catch*. Oto przykład:

```
d = $q.defer()
p = d.promise
p.catch(function (wynik) { alert('mamy redżekta: ' + wynik); })
d.reject(new Error('nie udało się wykonać obliczeń'))
```

Jeśli chcemy, żeby coś wykonało się wtedy, kiedy wykonywanie działania zakończyło się – niezależnie od tego, czy zakończyło się sukcesem czy porażką (czy na deferredzie zawołano *resolve* czy *reject*), możemy zarejestrować słuchacza metodą *finally* promisy:

```
d = $q.defer()
p = d.promise
p.finally(function () { console.log('koniec!'); })
d.reject(1337);
```

W Javaskrypcie słowa *catch* i *final* są zastrzeżone. Jeśli chcemy, żeby nasz kod nie wywalał się na niektórych przeglądarkach (na przykład IE8 i Androidzie 2.x), piszemy:

```
d = $q.defer()
p = d.promise
p['catch'](function (wynik) { alert('mamy redżekta: ' + wynik); })
d.reject(new Error('nie udało się wykonać obliczeń'))
```

Jest też drugi sposób na odrzucenie deferreda. Zamiast wołać na nim metodę *reject*, możemy zawołać metodę *resolve* i przekazać jej specjalną wartość stworzoną przez *$q.reject*:

```
d = $q.defer()
p = d.promise
p.catch(function (err) { console.log('błąd: ' + err); })
d.resolve($q.reject(new Error('przepełnienie stosu')))
```

Jeśli chcemy zarejestrować dwa słuchacze: jednego, który będzie uruchomiony, kiedy deferred zostanie zresolvowany, i drugiego, który zostanie uruchomiony, kiedy deferred zostanie zredżektowany, możemy to zrobić jednym wywołaniem metody *then*:

```
d = $q.defer()
p = d.promise
p.then(function (wynik) { alert('ok: ' + wynik); }, function (err)
{alert('błąd: ' + err); })
d.reject(new Error('przepełnienie stosu'))
```

Jeśli chcemy przekazywać co pewien czas różne wartości – na przykład bo chcemy powiadamiać o postępach w obliczeniach – służy do tego metoda *notify*. Tak przekazywane komunikaty możemy odbierać listenerem przekazanym jako trzeci parametr metodzie *then* promisy. Oto przykład:

```
d = $q.defer()
p = d.promise
p.then(function (wynik) { alert('ok: ' + wynik); }, function (err)
{alert('błąd: ' + err); }, function (postep) { alert('postęp: ' +
postep); })
d.notify(10)
d.notify(75)
d.notify(100)
```

sposób drugi, w który konstruktorowi promisa przekazujemy funkcję rozwiązującą deferreda

Jest też drugi sposób na korzystanie z deferredów i promis. Przy tym sposobie w ogóle nie widzimy deferreda – jest on przed nami, programistami, całkowicie ukryty. Zamiast tego od razu tworzymy promisę, a konstruktorowi przekazujemy funkcję, którą angular natychmiast uruchamia przekazując jej metody *resolve* i *reject* deferreda. Oto przykład:

```
$q(function (resolve, reject) {
  resolve(1337);
}).then(
  function (wynik) {
    console.log('wynik: ' + wynik);
  }, function (err) {
    console.log('błąd: ' + err);
  }
)
```

Albo:

```
$q(function (resolve, reject) {
  reject(new Error('serwer nie odpowiada'));
}).then(
  function (wynik) {
    console.log('wynik: ' + wynik);
  }, function (err) {
    console.log('błąd: ' + err);
  }
)
```

Przy korzystaniu z tego sposobu nie mamy dostępu do metody *notify*.

then zwraca promisę

Metoda *then* promisy tworzy nową promisę. Wypadnie z niej to, co zwróci słuchacz tej pierwszej promisy. Oto przykład:

```
d = $q.defer();
p1 = d.promise;
p2 = p1.then(function (wynik) {
  return wynik * 2;
});
p2.then(function (wynik) {
  console.log('wynik: ' + wynik);
});
d.resolve(357);
```

Ten pierwszy słuchacz może też zwrócić redżekta stworzonego przez *$q.reject*:

```
d = $q.defer();
p1 = d.promise;
p2 = p1.then(function (wynik) {
  if (wynik == 0) {
    return $q.reject(new Error('dzielenie przez 0'));
  }
  return 1 / wynik;
});
p2.then(function (wynik) {
  console.log('wynik: ' + wynik);
}, function (err) {
  console.log('błąd: ' + err);
});
d.resolve(0);
// albo: d.resolve(10);
```

Wysyłanie żądań

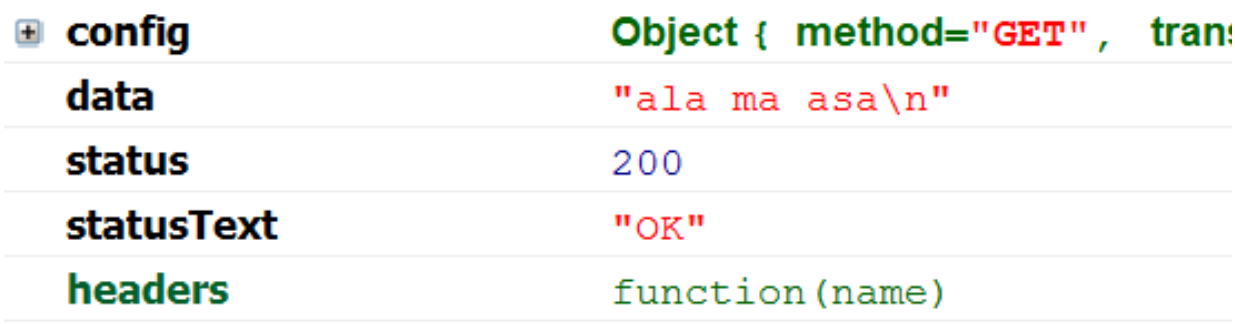
Kontroler może poprosić o serwis *$http*. Ten serwis jest funkcją, która pozwala wysyłać żądania HTTP. Tej funkcji przekazujemy słownik z informacjami o żądaniu, które ma być wysłane – możemy w nim podać na przykład metodę HTTP oraz URL. Ta funkcja zaczyna wysyłanie hatetepowego żądania typu get. Ta funkcja kończy działanie, zanim przeglądarka wyśle żądanie, ale tworzy parę *deferred, promisa*. I zwraca promisę, i tworzy callbacka, który jak przyjdzie odpowiedź, wrzuci tę odpowiedź do deferreda. Dzięki temu z tej funkcji dostajemy promisę, w której możemy się zarejestrować, żeby dostać odpowiedź, kiedy ta przyjdzie. Przykładowo, jeśli w mamy plik *dane.php* o treści:

```
ala ma asa
```

to w taki sposób możemy pobrać te dane:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope, $http) {
          $http({method: 'GET', url: 'dane.php'}).then(function
(x) {
               console.log(x);
          });
     });
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
     </div>
</body>
</html>
```

Wynik, który dostajemy z promisy (w powyższym przyladku jest on w zmiennej x), to obiekt zawierający różne informacje o odpowiedzi HTTP, która przyszła z serwera. Wygląda on tak:



Jak widać, atrybut *data* tego obiektu zawiera treść odpowiedzi.

Z kolei metoda *headers* tego obiektu pozwala dostać treść dowolnego nagłówka HTTP. Jest ona niewrażliwa na wielkość liter w nazwie nagłówka. Oto przykład jej użycia:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope, $http) {
          $http({method: 'GET', url: 'dane.php'}).then(function
(x) {
               console.log(x.headers('CONTENT-tYpE'));
          });
     });
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
     </div>
</body>
</html>
```

Obiekt zwrócony przez funkcję *$http* oprócz metody *then* (którą to metodę ma każda promisa), ma też metody *success* i *error*. Te metody pozwalają przekazać callbacka, który zostanie uruchomiony kiedy serwer odeśle odpowiedź z kodem błędu 200 (to robi metoda *success*) albo kodem błędu innym niż 200 (to robi metoda *error*). Oba callbacki w argumencie otrzymują treść odpowiedzi, którą dostaliśmy od serwera, kod błędu oraz funkcję pozwalającą sprawdzić treść dowolnego nagłówka HTTP:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope, $http) {
          var h = $http({method: 'GET', url: 'dane.php'});
          h.success(function (tresc, kod, naglowek) {
               console.log('sukces: ' + tresc);
               console.log('kod błędu: ' + kod);
               console.log('nagłówek: ' + naglowek('CONTENT-
type'));
          });
          h.error(function (tresc, kod, naglowek) {
               console.log('error: ' + tresc);
               console.log('kod błędu: ' + kod);
               console.log('nagłówek: ' + naglowek('CONTENT-
type'));
          });
     });
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
     </div>
</body>
</html>
```

Obie te metody (i wiele innych metod w angularze) zwracają obiekt, na którym je wywołaliśmy – dzięki temu powyższy przykład można zapisać i tak:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope, $http) {
          $http({method: 'GET', url: 'dane.php'})
               .success(function (x) {
                    console.log('sukces: ' + x);
               })
               .error(function (x) {
                    console.log('error: ' + x);
               });
     });
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
     </div>
</body>
</html>
```

Żądania można też wysyłać inaczej. Obiekt *$http* ma metody *get* i *post*, które pozwalają wysyłać żądania typu get i post. Oto przykład:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope, $http) {
          $http.get('dane.txt').then(function (odpowiedz) {
               console.log('kod błędu: ' + odpowiedz.status);
               console.log('treść odpowiedzi: ' + odpowiedz.tresc);
          });
     });
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
          {{ komunikat }}
```

```
     </div>
</body>
</html>
```

Promisa zwrócona przez metodę *get* ma – tak samo jak ta zwrócona przez *$http* – dwie specjalne metody: *success* i *error*:

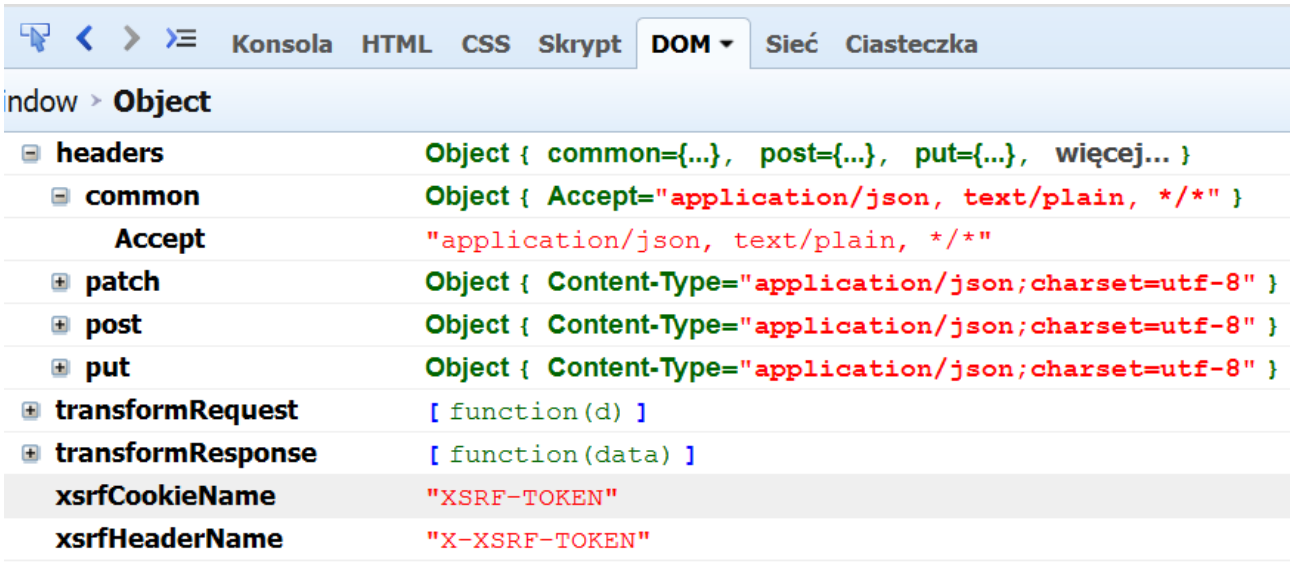
```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope, $http) {
          var p = $http.get('danexxx.php')
          p.success(function (dane, kodBledu, dajNaglowek) {
               $scope.komunikat = 'ok; treść odpowiedzi: ' + dane +
', kod błędu: ' + kodBledu + ', content-type:' +
dajNaglowek('content-type');
          });
          p.error(function (dane, kodBledu, dajNaglowek) {
               $scope.komunikat = 'błąd; treść odpowiedzi: ' + dane
+ ', kod błędu: ' + kodBledu + ', content-type:' +
dajNaglowek('content-type');
          });
     });
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
          <p ng-bind='komunikat'></p>
     </div>
</body>
</html>
```

Metody *success* i *error* są napisane w stylu *fluent interface* – zwracają obiekt, na którym były wywołane. Dzięki temu powyższy przykład można też zapisać tak:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope, $http) {
          $http.get('dane.php')
               .success(function (dane, kodBledu, dajNaglowek) {
                    $scope.komunikat = 'ok; treść odpowiedzi: ' +
dane + ', kod błędu: ' + kodBledu + ', content-type:' +
dajNaglowek('content-type');
               })
               .error(function (dane, kodBledu, dajNaglowek) {
                    $scope.komunikat = 'błąd; treść odpowiedzi: ' +
dane + ', kod błędu: ' + kodBledu + ', content-type:' +
dajNaglowek('content-type');
               });
     });
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
          <p ng-bind='komunikat'></p>
     </div>
</body>
</html>
```

transformacje żądań i odpowiedzi

W *$http.defaults* są różne domyślne ustawienia mówiące, jak mają być wysyłane żądania HTTP.



Jest tam na przykład – w *$http.defaults.transformRequest* – lista funkcji, którymi zostaną przetworzone dane wszystkich wysyłanych żądań. I – w *$http.defaults.transformResponse* – lista funkcji, którymi zostaną przetworzone dane wszystkich przyszłych odpowiedzi. Domyślnie na obu tych listach jest po jednej funkcji.

Funkcja domyślnie przetwarzająca dane żądania wykrywa, jeśli jest w nich jakiś obiekt. Jeśli tak, serializuje go jsonem.

Funkcja domyślnie przetwarzająca dane odpowiedzi wykrywa, jeśli jest w nich jakiś json. Jeśli tak, deserializuje go.

Dla eksperymentu można spróbować, jak działają te funkcje w konsoli Firebuga pisząc:

```
$http.defaults.transformRequest[0]({kwota: 17})
$http.defaults.transformResponse[0]('{"kwota": "17"}')
```

Dzięki temu mechanizmowi jeśli serwer w odpowiedzi na nasze żądanie przyśle jsona, nasz callback dostanie go już rozpakowanego. Możemy się o tym przekonać, jeśli napiszemy plik *dane.txt* o takiej treści:

```
{"imie": "Mirek"}
```

i spróbujemy go pobrać w taki sposób:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope, $http) {
          $http.get('dane.txt').success(function (tresc) {
               console.log(tresc);
          });
     });
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
     </div>
</body>
</html>
```

Po uruchomieniu takiego programu zobaczymy, że polecenie *console.log* wyświetla obiekt, a nie napis.

metoda POST

Metoda *post* obiektu *$http* pozwala wysyłać żądania hatetepową metodą *post*. Tej metodzie przekazujemy dwa parametry: URL i dane do przesłania. Dane są przesyłane po prostu jako napis – nie są automatycznie przerabiane na parametry postaci *?parametr1=wartość1&parametr2=wartość2*. Więc jeśli będziemy chcieli odebrać tak przesłane dane pehapem, nie znajdziemy ich w *$_GET*, a tylko w *$HTTP_RAW_POST_DATA*. Możemy to wypróbować pisząc na przykład w pliku *test.php* taki program:

```
<?php
print "dostalismy te dane: ";
print_r($HTTP_RAW_POST_DATA);
```

i pisząc taki program javaskryptowy:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope, $http) {
          $http.post('test.php', 'czarna krowa w kropki
bordo').success(function (tresc) {
               console.log(tresc);
          });
     });
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
     </div>
</body>
</html>
```

Jeśli ten drugi parametr (dane do przesłania) jest obiektem, zostanie (jak już wiemy) zserializowany do jsona. Możemy się o tym przekonać pisząc taki program:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope, $http) {
          $http.post('test.php', {wiek: 17, kwota:
75}).success(function (tresc) {
               console.log(tresc);
          });
     });
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
     </div>
</body>
</html>
```

Jeśli chcemy przesłać serwerowi dane w postaci parametrów HTTP (czyli jako napis *?parametr1=wartość1&parametr2=wartość2*), możemy użyć funkcji *$.param* z biblioteki jQuery. Warto tylko jeszcze zmienić nagłówek *content-type* w wysyłanym żądaniem: domyślnie jest w nim *application/json*, a kiedy przesyłamy dane w postaci parametrów, powinniśmy w tym nagłówku mieć *application/x-www-form-urlencoded.* Możemy to wypróbować pisząc taki program w *test.php*:

```
<?php
print "dostalismy te dane: ";
print_r($_POST);
```

i taki program javaskryptowy:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='//code.jquery.com/jquery-2.1.1.min.js'></script>
     <script src='angular.js'></script>
     <script>
angular.module('MojModul', [])
     .controller('MojKontroler', function ($scope, $http) {
          $http.post(
               'test.php',
               $.param({wiek: 17, kwota: 75}),
               {headers: {'Content-Type': 'application/x-www-form-
urlencoded; charset=UTF-8'}}
          ).success(function (tresc) {
               console.log(tresc);
          });
     });
     </script>
</head>
<body ng-app='MojModul'>
     <div ng-controller='MojKontroler'>
     </div>
</body>
</html>
```

Walidowanie aplikacji pisanych w angularze

Kiedy spróbujemy zwalidować aplikację napisaną w angularze walidatorem HTML-a (na przykład walidatorem ze strony *http://validator.w3.org*), zauważymy, że nasza aplikacja się nie zwaliduje. To dlatego, że angular używa niestandardowych atrybutów (na przykład *ng-controller*), a standard HTML5 przewiduje, że niestandarowe atrybuty powinny mieć nazwy zaczynające się od *data-*. Angular rozumie też, jeśli jego atrybuty zaczniemy od *data-* - na przykład *data-ng-controller*. Możemy więc tak pisać, jeśli chcemy walidować naszą aplikację.

Oto przykład HTML-a, który się nie waliduje:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
</head>
<body ng-app>
     <input type='text' ng-model='imie'>
     <div>Witaj, <span ng-bind='imie'></span>!</div>
</body>
</html>
```

A oto ten sam HTML przerobiony tak, żeby się walidował:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script src='angular.js'></script>
</head>
<body data-ng-app>
     <input type='text' data-ng-model='imie'>
     <div>Witaj, <span data-ng-bind='imie'></span>!</div>
</body>
</html>
```

Tworzenie własnych serwisów

Bywa, że kontroler korzysta z czegoś (napisu, funkcji, obiektu itp.), co chcemy żeby było tworzone poza tym kontrolerem. Taką rzecz tworzoną poza kontrolerem i dostarczaną do kontrolera nazywamy serwisem.

Z kilku powodów możemy chcieć tworzyć coś poza serwisem:

- żeby podczas testów można było zamiast tej rzeczy dać jakąś inną, na przykład wydmuszkę,
- żeby jedna, raz utworzona rzecz mogła być użyta w kilku kontrolerach,
- dla porządku – żeby logika biznesowa była poza kontrolerem.

Angular ma mechanizm pozwalający dostarczać serwisy do kontrolerów. Widzieliśmy go już przy okazji serwisu *$scope* czy *$http*. Oprócz gotowych serwisów, dostarczanych przez angulara – takich jak *$scope* – możemy też tworzyć własne serwisy. Serwisy tworzymy tak, że dostarczamy angularowi nazwę serwisu oraz przepis na jego stworzenie. Od tej chwili każdy kontroler (i nie tylko kontroler, ale o tym zaraz) może (sposobem, który już widzieliśmy) poprosić o wstrzyknięcie mu serwisu, a angular go stworzy – korzystając według naszego przepisu – i wstrzyknie.

Są cztery rodzaje przepisów na tworzenie serwisu: przepisy typu *value*, *factory*, *service* i *provider*. Do rejestrowania tych przepisów służą odpowiednio metody *value*, *factory*, *service* i *provider*.

tworzenie serwisów metodą *value*

Najprostsze są przepisy typu *value*. W zasadzie nie są to nawet przepisy: metoda *value* pozwala po prostu zarejestrować pod wybraną przez nas nazwą dowolną wartość. Kiedy kontroler o nią poprosi, zostanie ona wstrzyknięta w kontrolera. Oto przykład:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script
src='//ajax.googleapis.com/ajax/libs/angularjs/1.2.22/angular.min.
js'></script>
    <script>
angular.module('MojaAplikacja', [])
    .value('powitanie', 'Dzień dobry! ')
    .controller('MojKontroler', function ($scope, powitanie) {
        $scope.komunikat = powitanie + 'Oto przykład użycia metody
value.';
    });
    </script>
</head>
<body data-ng-app='MojaAplikacja'>
     <div data-ng-controller='MojKontroler'>
        {{ komunikat }}
     </div>
</body>
</html>
```

W taki sposób możemy udostępniać dowolne wartości: nie tylko napisy czy liczby, ale też funkcje, dowolne obiekty itp. Oto przykład – serwis pozwalający losować liczbę całkowitą z zadanego zakresu:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script
src='//ajax.googleapis.com/ajax/libs/angularjs/1.2.22/angular.min.
js'></script>
    <script>
angular.module('MojaAplikacja', [])
    .value('losuj', function (a, b) {
        return a + Math.floor(Math.random() * (b - a + 1));
    })
    .controller('MojKontroler', function ($scope, losuj) {
        $scope.komunikat = 'wylosowalem liczbe: ' + losuj(1, 6);
    });
    </script>
</head>
<body data-ng-app='MojaAplikacja'>
     <div data-ng-controller='MojKontroler'>
        {{ komunikat }}
     </div>
</body>
</html>
```

tworzenie serwisów metodą *factory*

Czasem trzeba, żeby funkcja będąca serwisem korzystała z jakiegoś innego serwisu – stworzonego przez naz (jak nasz serwis *losuj*) lub nie przez naz (jak stworzony przez twórców angulara serwis *$http*). Ale skąd ma go wziąć? Ktoś musi go w nią wstrzyknąć. My tego nie zrobimy – bo choć samo wstrzyknięcie nie byłoby bardzo trudne, to nie mielibyśmy skąd wziąć serwisu *$http*. Wstrzykiwać serwisy umie tylko angular. Więc jeśli serwis ma mieć coś wstrzyknięte, musi być stworzony nie przez nas, ale przez angulara. Więc robi się tak, że my tworzymy fabrykę umiejącą stworzyć serwis, dajemy tę fabrykę angularowi i mówimy, co ma być w nią wstrzyknięte, a kiedy ktoś poprosi o serwis, angular stworzy go używając naszej fabryki. Do tego służy metoda *factory*. Oto przykład jej użycia – serwis, który korzystając z serwisów *$http* i *losuj* pobiera z *openweathermap.org* informację o pogodzie w losowym miejscu na Ziemi:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script
src='//ajax.googleapis.com/ajax/libs/angularjs/1.2.22/angular.min.
js'></script>
    <script>
```

```
angular.module('MojaAplikacja', [])
    .value('losuj', function (a, b) {
        return a + Math.floor(Math.random() * (b - a + 1));
    })
     .factory('pogodaLosowegoMiejsca', function (losuj, $http) {
          return function() {
               var φ = losuj(-90, 90);
               var λ = losuj(-180, 180);
               var url =
'http://api.openweathermap.org/data/2.5/weather?lat=' + φ +
'&lon=' + λ;
               return $http.get(url);
          }
     })
    .controller('MojKontroler', function ($scope,
pogodaLosowegoMiejsca) {
          $scope.pogoda = {
               lat: 0,
               lon: 0,
               miasto: '',
               temperatura: 0
          };
        pogodaLosowegoMiejsca().success(function (pogoda) {
               $scope.pogoda = {
                    lat: pogoda.coord.lat,
                    lon: pogoda.coord.lon,
                    miasto: pogoda.name,
                    temperatura: pogoda.main.temp - 273.15
               };
          });
    });
    </script>
</head>
<body data-ng-app='MojaAplikacja'>
     <div data-ng-controller='MojKontroler'>
          <h2>pogoda w punkcie {{ pogoda.lat }},
{{ pogoda.lon }}</h2>
          najbliższe miasto: {{ pogoda.miasto }}<br>
          temperatura: {{ pogoda.temperatura | number:0 }} °C
     </div>
</body>
</html>
```

tworzenie serwisów metodą *service*

Czasem bywa, że chcemy udostępniać naszym kontrolerom serwis tworzony przy użyciu napisanego przez nas konstruktora. Przykładowo, oto konstruktor tworzący generator liczb losowych, który nie powtarza dwa razy pod rząd tej samej liczby:

```
function Generator(losuj) {
  this.a = 1;
  this.b = 6;
  this.poprzednia = NaN;
  this.losuj = function() {
    var nowa;
    while ((nowa = losuj(this.a, this.b)) == this.poprzednia) {}
    this.poprzednia = nowa;
    return nowa;
  }
}
```

Jak widać, ten generator do działania potrzebuje naszego serwisu *losuj* (tego z poprzednich przykładów). Tego generatora używa się tak:

```
function losuj(a, b) {
    return a + Math.floor(Math.random() * (b - a + 1));
}
var g = new Generator(losuj);
console.log(g.losuj(5, 8));
console.log(g.losuj(5, 8));
console.log(g.losuj(5, 8));
```

Jeśli zechcemy taki generator udostępnić naszym kontrolerom jako serwis, możemy zrobić to przy użyciu znanej już nam metody *factory*, w taki sposób:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script
src='//ajax.googleapis.com/ajax/libs/angularjs/1.2.22/angular.min.
js'></script>
    <script>
function Generator(losuj) {
  this.poprzednia = NaN;
  this.losuj = function(a, b) {
    var nowa;
    while ((nowa = losuj(a, b)) == this.poprzednia) {}
    this.poprzednia = nowa;
    return nowa;
  }
}
angular.module('MojaAplikacja', [])
    .value('losuj', function (a, b) {
        return a + Math.floor(Math.random() * (b - a + 1));
    })
     .factory('generator', function (losuj) {
          return new Generator(losuj);
     })
    .controller('MojKontroler', function ($scope, generator) {
        $scope.liczby = [];
          for (var i=0; i < 10; i++) {
               $scope.liczby.push(generator.losuj(3, 7));
          }
    });
    </script>
</head>
<body data-ng-app='MojaAplikacja'>
     <div data-ng-controller='MojKontroler'>
          {{ liczby }}
     </div>
</body>
</html>
```

To samo można też zrobić trochę bardziej skrótowo. Jeśli fabrykę serwisów zarejestrujemy nie metodą *factory* a metodą *service*, angular będzie uruchamiał ją nie jak funkcję, a jak konstruktora – przez *new*. Oto powyższy przykład przerobiony tak, żeby używał metody *service*:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script
src='//ajax.googleapis.com/ajax/libs/angularjs/1.2.22/angular.min.
js'></script>
    <script>
angular.module('MojaAplikacja', [])
    .value('losuj', function (a, b) {
        return a + Math.floor(Math.random() * (b - a + 1));
    })
     .service('generator', function Generator(losuj) {
          this.poprzednia = NaN;
          this.losuj = function(a, b) {
               var nowa;
               while ((nowa = losuj(a, b)) == this.poprzednia) {}
               this.poprzednia = nowa;
               return nowa;
          }
     })
    .controller('MojKontroler', function ($scope, generator) {
        $scope.liczby = [];
          for (var i=0; i < 10; i++) {
               $scope.liczby.push(generator.losuj(3, 7));
          }
    });
    </script>
</head>
<body data-ng-app='MojaAplikacja'>
     <div data-ng-controller='MojKontroler'>
          {{ liczby }}
     </div>
</body>
</html>
```

Zauważmy, że tutaj twórcy Angulara niezręcznie wybrali nazwy. Słowo *service* oznacza rzecz, którą można wstrzykiwać, oraz jeden (choć niejedyny) z typów przepisów na tworzenie takich wstrzykiwalnych rzeczy.

tworzenie serwisów metodą *provider*

Czasem potrzebujemy skonfigurować jakąś fabrykę serwisów, zanim nasze kontrolery zaczną pobierać z niej serwis. Tak bywa zwłaszcza wtedy, kiedy chcemy raz napisać serwis, a potem używać go w różnych aplikacjach. Przydaje się wtedy metoda *provider*. Podajemy jej pewien konstruktor, a angular stworzy tym konstruktorem obiekt. Kiedy potrzebny będzie serwis, angular wywoła na tym obiekcie metodę *$get*, a ona ma stworzyć serwis. Więc metodzie *provider* przekazujemy fabrykę fabryki serwisu. Kiedy korzystamy z metody *provider*:

- fabryka fabryki serwisu to napisany przez nas konstruktor, który dostarczamy angularowej metodzie *provider*,
- fabryka serwisu to metoda *$get* obiektu stworzonego tym konstruktorem (fabryką fabryki),
- serwis to dowolna wartość zwrócona przez tę metodę *$get* (przez serwis).

Jeśli chcemy, żeby fabryka serwisu stworzona przez fabrykę fabryki serwisu została jakoś dostosowana przy starcie aplikacji, możemy przy użyciu metody *config* zarejestrować funkcję, która zostanie uruchomiona przy starcie aplikacji. W tę funkcję można wstrzyknąć fabrykę, byleśmy znali jej nazwę – a nazwa fabryki to nazwa serwisu z dodanym słowem *Provider*. Można tak wstrzykiwać zarówno fabryki napisane przez nas jak i fabryki napisane przez twórców angulara (na przykład *$httpProvider*).

Oto przykład:

```
<!doctype html>
<html lang='en'>
<head>
     <meta charset='UTF-8'>
     <title>angular</title>
     <script
src='//ajax.googleapis.com/ajax/libs/angularjs/1.2.22/angular.min.
js'></script>
    <script>
angular.module('MojaAplikacja', [])
    .value('losuj', function (a, b) {
        return a + Math.floor(Math.random() * (b - a + 1));
    })
     .provider('losujBezPowtorzen', function () {
          var zakres = [1, 6];
          this.ustawZakres = function(a, b) {
               zakres[0] = a;
               zakres[1] = b;
          }
          this.$get = function (losuj) {
               var poprzednia = NaN;
```

```
                return function () {
                     var a = zakres[0];
                     var b = zakres[1];
                     var nowa;
                     while ((nowa = losuj(a, b)) == poprzednia) {}
                     poprzednia = nowa;
                     return nowa;
                }
           }
     })
     .config(function (losujBezPowtorzenProvider) {
           losujBezPowtorzenProvider.ustawZakres(2014, 2073);
     })
    .controller('MojKontroler', function (losujBezPowtorzen) {
           console.log(losujBezPowtorzen());
           console.log(losujBezPowtorzen());
           console.log(losujBezPowtorzen());
           console.log(losujBezPowtorzen());
           console.log(losujBezPowtorzen());
    });
    </script>
</head>
<body data-ng-app='MojaAplikacja'>
     <div data-ng-controller='MojKontroler'>
     </div>
</body>
</html>
```

metoda *constant*

Serwisy można też tworzyć metodą *constant*. Działa ona podobnie do metody *value*, ale serwisy stworzone nią można wstrzykiwać w funkcję zarejestrowaną w metodzie *config.*

kiedy użyć której metody

Kiedy potrzebujemy mieć w naszej aplikacji serwis, najprościej jest użyć metody *value*. Chyba że potrzebujemy coś wstrzyknąć w ten serwis – wtedy musimy użyć metody *factory*. Chyba że chcemy ten serwis tworzyć przez *new Konstruktor()* – wtedy wygodniej będzie użyć metody *service*. Albo chyba że chcemy przy starcie aplikacji konfigurować fabrykę serwisów – wtedy musimy użyć metody *provider*.

Crud w angularze

W katalogu *materialy/crud1* jest prosty crud zrobiony w angularze. Pozwala on zarządzać listą zadań do zrobienia: przeglądać je, tworzyć, kasować i edytować.

restowe api

W katalogu *api* jest napisany w PHP program pozwalający:

- pobrać listę wszystkich zadań,
- stworzyć nowe zadanie,
- skasować istniejące zadanie,
- zmienić istniejące zadanie.

Dzięki plikowi *.htaccess*, który jest w tym katalogu, te pehapowe programy można wywoływać przez ładne restowe API:

- żeby pobrać wszystkie zadania, wysyłamy żądanie GET do zasobu *api/zadania*,
- żeby pobrać zadanie o numerze (dla przykładu) 1337, wysyłamy żądanie GET do zasobu *api/zadanie/1337*,
- żeby skasować zadanie o numerze (dla przykładu) 1337, wysyłamy żądanie DELETE do zasobu *api/zadanie/1337*,
- żeby zaktualizować zadanie o numerze (dla przykładu) 1337, wysyłamy żądanie PUT do zasobu *api/zadanie/1337*, w treści żądania przesyłając opis zadania zakodowany w JSON-ie,
- żeby stworzyć nowe zadanie, wysyłamy żądanie POST do zasobu *api/zadania*, w treści żądania przesyłając opis zadania zakodowany w JSON-ie.

Przy uruchamianiu tej aplikacji trzeba zajrzeć do pliku *.htaccess* i odpowiednio poprawić w nim linijkę z dyrektywą *RewriteBase*.

Pisanie tego cruda warto zacząć od poeksperymentowania z tym API, na przykład wysyłając ręcznie żądania przy użyciu *putty*.

Oto przykład pobrania zadań:

```
GET /nic/crud1/api/zadania HTTP/1.1
Host: psobolewski.students.alx.pl

HTTP/1.1 200 OK
Date: Sat, 27 Sep 2014 12:46:55 GMT
Server: Apache/2.2.16 (Debian)
X-Powered-By: PHP/5.3.29-1~dotdeb.0
Cache-Control: no-cache
Content-Length: 435
Content-Type: text/html

[{"id":"1","opis":"naprawi\u0107 formularz
klienta","uwagi":"Formularz edycji klienta jest popsuty - lista
rozwijana nie dzia\u0142a.","data":"2015-03-27"},
{"id":"3","opis":"przetestowa\u0107
ankiet\u0119","uwagi":"Przetestowa\u0107 mechanizm
ankiet.","data":"2015-04-12"},
{"id":"4","opis":"zez\u0142omowa\u0107
monitory","uwagi":"","data":"2015-04-01"},
{"id":"5","opis":"zez\u0142omowa\u0107
monitory","uwagi":"","data":"2015-04-01"}]
```

Oto przykład pobrania zadania o id=3:

```
GET /nic/crud1/api/zadanie/3 HTTP/1.1
Host: psobolewski.students.alx.pl


HTTP/1.1 200 OK
Date: Sat, 27 Sep 2014 12:48:33 GMT
Server: Apache/2.2.16 (Debian)
X-Powered-By: PHP/5.3.29-1~dotdeb.0
Cache-Control: no-cache
Content-Length: 119
Content-Type: text/html

{"id":"3","opis":"przetestowa\u0107
ankiet\u0119","uwagi":"Przetestowa\u0107 mechanizm
ankiet.","data":"2015-04-12"}
```

Oto przykład skasowania zadania o id=3:

```
DELETE /nic/crud1/api/zadanie/3 HTTP/1.1
Host: psobolewski.students.alx.pl

HTTP/1.1 200 OK
Date: Sat, 27 Sep 2014 12:50:15 GMT
Server: Apache/2.2.16 (Debian)
X-Powered-By: PHP/5.3.29-1~dotdeb.0
Cache-Control: no-cache
Content-Length: 3
Content-Type: text/html
```

Oto przykład dopisania uwagi do zadania o id=4 (do policzenia, ile znaków ma nasz json, można użyć narzędzia *http://string-functions.com/length.aspx*):

```
PUT /nic/crud1/api/zadanie/4 HTTP/1.1
Host: psobolewski.students.alx.pl
Content-Length: 93

{"id":"4","opis":"zez\u0142omowa\u0107 monitory","uwagi":"testowa
uwaga","data":"2015-04-01"}

HTTP/1.1 200 OK
Date: Sat, 27 Sep 2014 13:02:01 GMT
Server: Apache/2.2.16 (Debian)
X-Powered-By: PHP/5.3.29-1~dotdeb.0
Cache-Control: no-cache
Content-Length: 19
Content-Type: text/html

{"result": "ok"}
```

Oto przykład stworzenia nowego zadania:

```
POST /nic/crud1/api/zadania HTTP/1.1
Host: psobolewski.students.alx.pl
Content-Length: 70

{"opis":"testowe zadanie","uwagi":"testowa uwaga","data":"1410-07-
24"}

HTTP/1.1 200 OK
Date: Sat, 27 Sep 2014 13:27:43 GMT
Server: Apache/2.2.16 (Debian)
X-Powered-By: PHP/5.3.29-1~dotdeb.0
Cache-Control: no-cache
Content-Length: 22
Content-Type: text/html

{"result":"ok","id":5}
```

serwis *zadanieService*

Jak widać w *todo.js*, tworzymy serwis *zadanieService*:

```
aplikacja.factory('zadanieService', ['$http', function
zadaniaFactory($http) {
    return {
        pobierzZadania: function pobierzZadania() {
            return $http.get('api/zadania');
        },
        klonujZadanie: function klonujZadanie(zadanie) {
            return {
                id: zadanie.id,
                opis: zadanie.opis,
                uwagi: zadanie.uwagi,
                data: zadanie.data
            };
        },
        skasujZadanie: function skasujZadanie(zadanie) {
            return $http.delete('api/zadanie/' + zadanie.id);
        },
        zapiszZadanie: function zapiszZadanie(zadanie) {
            if (zadanie.id == undefined) {
                return $http.post('api/zadania/', zadanie);
            } else {
                return $http.put('api/zadanie/' + zadanie.id,
zadanie);
            }
        }
    };
}]);
```

Ten serwis abstrahuje komunikację z serwerem i typowe działania na zadaniach. Jest on obiektem, który nie ma żadnego stanu – jest tylko pojemnikiem na funkcje pozwalające pobrać z serwera list zadań, skasować zadanie, zapisać zadanie (czyli stworzyć nowe albo zapisać zmiany w istniejącym) i sklonować zadanie. Ten serwis nie zawiera żadnych metod związanych z interfejsem użytkownika naszej aplikacji. Zauważmy, że metody komunikujące się z serwerem zwracają promisy pozwalające rejestrować callbacki, które zostaną wykonane, kiedy z serwera przyjdzie odpowiedź – jeszcze nam się to przyda.

Serwis robimy metodą *factory* – nie możemy tu użyć metody *value*, bo potrzebujemy w nasz serwis wstrzyknąć serwis *$http*.

Zwróćmy uwagę, że fabrykę serwisu rejestrujemy tak, żeby nasz kod nie popsuł się przy minifikacji (minifikację możemy zrobić na przykład przy użyciu *http://closure-compiler.appspot.com/home*) – metodzie *factory* przekazujemy listę serwisów do wstrzyknięcia i naszą fabrykę serwisu.

pokazywanie listy zadań

W aplikacji jest tylko jeden kontroler – *KontrolerZadan*.

Tworzy on w zasięgu dwa atrybuty:

```
$scope.zadania = [];
$scope.jeszczeNieMaDanych = true;
```

Po czym pobiera zadania z serwera:

```
function pobierzZadaniaDoTabeli() {
    return zadanieService.pobierzZadania().success(function
(zadania) {
        $scope.zadania = zadania;
        $scope.jeszczeNieMaDanych = false;
    });
}
(...)
pobierzZadaniaDoTabeli();
```

Zauważmy, że funkcja *pobierzZadaniaDoTabeli* korzysta z faktu, że *zadanieService.pobierzZadania* zwraca promisę. Zauważmy, że funkcja *pobierzZadaniaDoTabeli* zwraca promisę pozwalającą zarejestrować callbacka, który zostanie uruchomiony po pobraniu listy zadań z serwera – jeszcze nam się to przyda.

Jak widać, atrybut *jeszczeNieMaDanych* pamięta, czy dane są pobrane. Wyświetleniem tabeli z zadaniami zajmuje się ta część pliku *index.html*:

```
<div data-ng-show='jeszczeNieMaDanych'>jeszcze nie ma danych</div>
<table>
    <tr>
        <th></th>
        <th>opis</th>
        <th>uwagi</th>
        <th>data</th>
        <th></th>
    </tr>
    <tr data-ng-repeat='zadanie in zadania'>
    (...)
```

dodawanie nowych zadań

Żeby użytkownik mógł dodawać nowe zadania, w kontrolerze mamy to:

```
$scope.noweZadanie = {};
(...)
$scope.zapiszNoweZadanie = function zapiszNoweZadanie(zadanie) {
    zadanieService.zapiszZadanie(zadanie).success(function () {
        pobierzZadaniaDoTabeli().success(function (zadania) {
            $scope.noweZadanie = {};
        });
    });
};
```

A w *index.html* to:

```
<tr>
    <td><button type='submit' data-ng-
click='zapiszNoweZadanie(noweZadanie)'>s</button></td>
    <td><input type='text' data-ng-model='noweZadanie.opis'
form='noweZadanie' /></td>
    <td><input type='text' data-ng-model='noweZadanie.uwagi'
form='noweZadanie' /></td>
    <td><input type='date' data-ng-model='noweZadanie.data'
form='noweZadanie' /></td>
</tr>
```

Pewnie lepiej by było, gdyby zapisywanie zadania nie było uruchamiane kliknięciem na guzik (przez *data-ng-click*), ale wysłaniem formularza (przez *data-ng-submit*) – wtedy można by też zapisać zadanie przez chlapnięcie entera w którymś z pól zadania. Ale ten formularz musiałby być wyniesiony poza tabelę (bo w HTML-u nie można mieć formularza w *tr*), a pola i guzik by się do niego odwoływały przez atrybut *form*. A w IE9, niestety, nie działa atrybut *form* w polach formularza – więc jeśli chcemy, żeby nasza aplikacja działała w IE9, musimy to zrobić w ten mniej zręczny sposób, widoczny na powyższym przykładzie.

kasowanie zadań

Żeby użytkownik mógł kasować zadania, przed każdym zadaniem dajemy przycisk, a po każdym zadaniu warunkowo wyświetlany komunikat *trwa zapisywanie zmian*:

```
<td>
    <button data-ng-click='skasujZadanie(zadanie)'>x</button>
    (...)
</td>
(...)
<td><span data-ng-show='czyTrwaZapisywanieZmian(zadanie)'>trwa
zapisywanie zmian</span></td>
```

W kontrolerze mamy taki kod:

```
var trwaZapisywanieZmian = new Set();
(...)
$scope.skasujZadanie = function skasujZadanie(zadanie) {
    trwaZapisywanieZmian.add(zadanie.id);
    zadanieService.skasujZadanie(zadanie).success(function () {
        pobierzZadaniaDoTabeli().success(function (zadania) {
            trwaZapisywanieZmian.delete(zadanie.id);
        });
    });
};
(...)
$scope.czyTrwaZapisywanieZmian = function(zadanie) {
    return trwaZapisywanieZmian.has(zadanie.id);
}
```

Jak widać, informację o tym, w których zadaniach trwa zapisywanie zmian, przechowujemy w zbiorze. Ponieważ javaskrypt w niektórych przeglądarkach nie zna klasy *Set*, w pliku *index.html* dołączyliśmy dający go shim:

```
<script src='//cdnjs.cloudflare.com/ajax/libs/es6-shim/0.18.0/es6-
shim.min.js'></script>
```

Jest to shim, który daje funkcjonalności ecmascriptu 6 w przeglądarkach, które nie rozumieją ecmascriptu 6.

edycja zadań

Zadania można edytować.

W tabeli z zadaniami w każdym wierszu (czyli przy każdym zadaniu) jest przycisk z literą *e*. Wciśnięcie tego przycisku powoduje, że pola w tym wierszu stają się edytowalnymi polami tekstowymi. W każdym wierszu są jeszcze dwa przyciski: *s* i *a*. Kiedy już wprowadzimy jakieś zmiany w zadaniu, wciśnięcie przycisku *s* zapisuje te zmiany, a wciśnięcie przycisku *a* je wycofuje.

Zmiana pól w komórce w pola edytowalne jest zrobiona tak, że w HTML-u wiersz w tabeli ma taki fragment:

```
<tr data-ng-repeat='zadanie in zadania'>
     <td>
          (...)
          <button data-ng-click='edytujZadanie(zadanie)' data-ng-
disabled='czyEdytowane(zadanie)'>e</button>
          (...)
     </td>
     <td data-ng-show='!
czyEdytowane(zadanie)'>{{ zadanie.opis }}</td>
     <td data-ng-show='czyEdytowane(zadanie)'><input type='text'
data-ng-model='edytowane(zadanie).opis' /></td>
     <td data-ng-show='!
czyEdytowane(zadanie)'>{{ zadanie.uwagi }}</td>
     <td data-ng-show='czyEdytowane(zadanie)'><input type='text'
data-ng-model='edytowane(zadanie).uwagi' /></td>
     <td data-ng-show='!
czyEdytowane(zadanie)'>{{ zadanie.data }}</td>
     <td data-ng-show='czyEdytowane(zadanie)'><input type='date'
data-ng-model='edytowane(zadanie).data' /></td>
     (...)
</tr>
```

A w *todo.js* w kontrolerze jest taki fragment:

```
aplikacja.controller('KontrolerZadan', ['$scope',
'zadanieService', function ($scope, zadanieService) {
     (...)
    var edytowane = {};
     (...)
    $scope.edytujZadanie = function(zadanie) {
        edytowane[zadanie.id] =
zadanieService.klonujZadanie(zadanie);
    };
    $scope.czyEdytowane = function(zadanie) {
        return edytowane[zadanie.id] != undefined;
    }
    $scope.edytowane = function(zadanie) {
        return edytowane[zadanie.id];
    }
     (...)
}]);
```

Jak widać, w domknięciu zasięgu mamy zmienną *edytowane*, a w niej słownik z treściami edytowanych zadań. W tym słowniku kluczami są id zadań, a wartościami słowniki z polami tych zadań. Kliknięcie na przycisk *e* uruchamia metodę *edytujZadanie*, które robi kopię zadania i umieszcza tę kopię w słowniku *edytowane*. Dzięki temu dla każdego edytowanego zadania pamiętamy zarówno jego stan sprzed edycji (z tablicy *zadania*) jak i stan wyedytowany (w słowniku *edytowane*). Dzięki temu każdą edycję można zapisać albo wycofać.

Jak widać, mamy jeszcze dwie proste funkcje związane z edytowaniem zadań. Funkcja *czyEdytowane* mówi, czy dane zadanie jest edytowane. Funkcja *edytowane* zwraca edytowaną wersję danego zadania. W HTML-u dla każdego zadania sprawdzamy (funkcją *czyEdytowane*), czy dane zadanie jest edytowane. Jeśli nie jest edytowane, jego pola (opis, uwagi, datę) pobieramy z tablicy *zadania* i po prostu wyświetlamy w komórce tabeli (elemencie *td*). Jeśli zadanie jest edytowane, jego pola pobieramy (funkcją *edytowane*) ze słownika *edytowane* i wyświetlamy jako pola tekstowe. Ponieważ angular robi dwustronne powiązanie między polem tekstowym a słownikiem z polami zadania będącym wartością w słowniku *edytowane*, wszystko, co zmienimy w polach tekstowych, jest zapamiętane.

Kiedy klikniemy przycisk *a*, uruchamiamy funkcję *anulujZmianyWZadaniu*. Widać to w tym fragmencie HTML-a:

```
<tr data-ng-repeat='zadanie in zadania'>
     <td>
          <button data-ng-
click='skasujZadanie(zadanie)'>x</button>
          <button data-ng-click='edytujZadanie(zadanie)' data-ng-
disabled='czyEdytowane(zadanie)'>e</button>
          <button data-ng-
click='zapiszZmianyWZadaniu(edytowane(zadanie))' data-ng-
disabled='! czyEdytowane(zadanie)'>s</button>
          <button data-ng-click='anulujZmianyWZadaniu(zadanie)'
data-ng-disabled='! czyEdytowane(zadanie)'>a</button>
     </td>
```

Ta funkcja – jak widać w *todo.js* – kasuje zadanie ze słownika *edytowane*:

```
$scope.anulujZmianyWZadaniu = function(zadanie) {
    delete edytowane[zadanie.id];
}
```

Jak widać w powyższym fragmencie HTML-a, kliknięcie przycisku *s* uruchamia funkcję *zapiszZmianyWZadaniu*. Ta funkcja – jak widać w *todo.js* – korzystając z *zadanieService* zapisuje zadanie na serwerze, po czym (i tu przydaje nam się, że *zadanieService.zapiszZadanie* zwraca promisę) pobiera z serwera wszystkie zadania, po czym usuwa zadanie ze słownika *edytowane*.

```
$scope.zapiszZmianyWZadaniu = function
zapiszZmianyWZadaniu(zadanie) {
     trwaZapisywanieZmian.add(zadanie.id);
     zadanieService.zapiszZadanie(zadanie).success(function () {
          pobierzZadaniaDoTabeli().success(function (zadania) {
               delete edytowane[zadanie.id];
               trwaZapisywanieZmian.delete(zadanie.id);
          });
     });
};
```

Zauważmy przy tym, że w czasie zapisywania zmian w zadaniu, przy tym zadaniu wyświetlany jest komunikat *trwa zapisywanie zmian*. Jest to zrobione tak samo jak przy kasowaniu zadań.

mogliśmy zwracać to, czego jeszcze nie dostaliśmy

W serwisie *zadanieService* jest metoda *pobierzZadania*. Zwraca ona nie listę zadań, tylko obietnicę (ang. *promise*) tej listy zadań:

```
pobierzZadania: function pobierzZadania() {
     return $http.get('api/zadania');
},
```

Kto wywołuje tę metodę, żeby zrobić coś z otrzymaną listą zadań, musi zarejestrować słuchacza:

```
zadanieService.pobierzZadania().success(function (zadania) {
     $scope.zadania = zadania;
     $scope.jeszczeNieMaDanych = false;
});
```

Można by też rozważyć inne podejście. Metoda *pobierzZadania* mogłaby zwracać pustą tablicę, zapamiętywać (na przykład w domknięciu) referencję do tej tablicy, a kiedy już przyjdą dane z serwera, umieszczać je w tej tablicy. Dzięki temu ten, kto korzysta z tej metody, w prostych zastosowaniach mógłby nie trudzić się rejestrowaniem listenerów. Umieszczałby otrzymaną tablicę w zasięgu, w HTML-u pisał kod wyświetlający ją, a sam angular dbałby o to, żeby przerysiwać ten fragment strony, kiedy zawartość tabeli się zmieni.

Przykład takiego podejścia jest w katalogu *zwracamy_zanim_dostajemy*. Jest to korzystająca z tego samego api co *crud1* aplikacja wyświetlająca listę zadań (bez możliwości kasowania, edytowania i tworzenia nowych zadań).

Jest tam (w *todo.js*) taki serwis:

```
aplikacja.factory('zadanieService', function zadaniaFactory($http) {
  return {
    pobierzZadania: function pobierzZadania() {
      var zadaniaDoZwrotu = [];
      $http.get('api/zadania').success(function (zadaniaOtrzymane) {
        for (var i=0; i<zadaniaOtrzymane.length; i++) {
          zadaniaDoZwrotu.push(zadaniaOtrzymane[i]);
        }
      });
      return zadaniaDoZwrotu;
    }
  }
});
```

W kontrolerze wywołujemy funkcję *pobierzZadania* a jej wynik wpisujemy w zasięg:

```
aplikacja.controller('KontrolerZadan', function ($scope,
zadanieService) {
    $scope.zadania = zadanieService.pobierzZadania();
});
```

W HTML-u wyświetlamy tę listę:

```
<table>
  <tr>
    <th>opis</th>
    <th>uwagi</th>
    <th>data</th>
  </tr>
  <tr data-ng-repeat='zadanie in zadania'>
    <td>{{ zadanie.opis }}</td>
    <td>{{ zadanie.uwagi }}</td>
    <td>{{ zadanie.data }}</td>
  </tr>
</table>
```

Takie podejście warto rozumieć, bo stosuje je angularowy serwis *$resource*.

Przy tym przykładzie warto zastanowić się nad kilkoma sprawami.

Problem 1. W serwisie po otrzymaniu danych z serwera przerzucamy je do tablicy *zadania*, tak:

```
for (var i=0; i<zadaniaOtrzymane.length; i++) {
  zadaniaDoZwrotu.push(zadaniaOtrzymane[i]);
}
```

Czy dałoby się zrobić to prościej, w taki sposób?:

```
zadaniaDoZwrotu = zadaniaOtrzymane;
```

Problem 2. W serwisie po otrzymaniu danych z serwera przerzucamy je do tablicy *zadania*, tak:

```
for (var i=0; i<zadaniaOtrzymane.length; i++) {
  zadaniaDoZwrotu.push(zadaniaOtrzymane[i]);
}
```

Skoro angular zauważa zmiany w modelu i przy każdej zmianie odrysowuje odpowiedni fragment strony, to czy oznacza to, że tabla z zadaniami zostanie w trakcie wykonywania tej pętli przerysowana tyle razy, ile razy ta pętla wykona *push*?

Tego nie wiem

Czy mogę używać dowolnej wersji angulara z dowolną wersją jquery? Czy któreś wersje mogą się gryźć?