Technologia i rozwiązania

AngularJS
Praktyczne przykłady

Praktyczne wprowadzenie do AngularJS!

Chandermani

Tytuł oryginału: AngularJS by Example

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-1615-7

Copyright © Packt Publishing 2015. First published in the English language under the title
'AngularJS by Example – (9781783553815)'.

Polish edition copyright © 2016 by Helion S.A.
All rights reserved.

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

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

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

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

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

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

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

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

O autorze 9

O recenzentach 11

Przedmowa 13

Zawartość książki 14
Co jest potrzebne? 15
Adresaci książki 15
Konwencje 15
Kod do pobrania 16
Errata 17
Piractwo 17

Rozdział 1. Pierwsze kroki 19

Podstawy wzorca model – widok – kontroler 20
Zwyczajowa aplikacja Witaj Angular (Odgadnij liczbę!) 21
Przygotowywanie roboczego serwera WWW 21
Tworzenie aplikacji Odgadnij liczbę! 22
Model aplikacji 23
Kontroler 24
Widok aplikacji 25
Wstawki 28
Dyrektywy 29
Wyrażenia 30
Dowiązywanie danych 32
Kontroler po raz wtóry 35
Zasięg 36
Inicjalizacja aplikacji 39
Narzędzia 41
Zasoby 41
Podsumowanie 42

Kup ksi k Pole ksi k

Rozdział 2. Piszemy pierwszą aplikację: 7-minutowy trening 45

Czym jest 7-minutowy trening? 46
Pobieranie kodu aplikacji 47
Organizacja kodu 48
Organizacja kodu JavaScriptu 48
Model aplikacji 7-minutowy trening 51
Dodawanie modułów aplikacji 53
Kontroler aplikacji 54
Wstrzykiwanie zależności 55
Implementacja kontrolera 59
Weryfikacja implementacji 63
Implementacja zmiany ćwiczenia 65
Stosowanie $watch do obserwowania zmian w modelu 66
Zmiana ćwiczeń przy użyciu API obietnic 69
Widok aplikacji 7-minutowy trening 72
Stosowanie ng-show do określania ścieżki do obrazka 73
Stosowanie dyrektywy ng-style i paska postępu frameworku Bootstrap 74
Dodawanie strony początkowej i końcowej 75
Wprowadzenie od aplikacji jednostronicowych 75
Dostarczanie dodatkowych informacji o ćwiczeniach 82
Dodawanie paneli opisu i wideo 82
Wyświetlanie pozostałego czasu treningu przy użyciu filtrów 90
Tworzenie filtra konwertującego sekundy na format godzinowy 90
Dodawanie informacji o następnym ćwiczeniu przy użyciu ng-if 94
Podsumowanie 97

Rozdział 3. Stosowanie kolejnych dobrodziejstw AngularJS 99

Formatowanie instrukcji wykonywania ćwiczenia 101
Przedstawienie dyrektywy ng-bind-html 102
Użycie ng-bind-html do wyświetlania instrukcji wykonywania ćwiczeń 103
Śledzenie postępu wykonywania ćwiczeń za pomocą klipów audio 105
Implementacja obsługi audio 106
Wstrzymywanie treningu 117
Implementacja wstrzymywania treningu w kontrolerze WorkoutController 118
Dodawanie fragmentu widoku do wstrzymywania i wznawiania treningu 120
Zatrzymywanie klipów po wstrzymaniu treningu 123
Wstrzymywanie i wznawianie treningu przy użyciu klawiatury 124
Rozbudowa panelu wideo 125
Refaktoryzacja panelu wideo i kontrolera 126
Odtwarzanie wideo w oknie dialogowym 127
Animacje w aplikacjach AngularJS 133
Animacje CSS w AngularJS 134
Animacje JavaScriptu w AngularJS 136
Dodawanie animacji do aplikacji 7-minutowy trening 138

Kup ksi k Pole ksi k

Zapis historii treningu przy użyciu usług AngularJS **141**
Podstawowe informacje o usługach AngularJS 141
Implementacja zapisu historii treningów 146
Dodawanie widoku historii treningów 149
Zdarzenia w AngularJS 152
Wzbogacanie śledzenia historii przy użyciu zdarzeń 155
Trwałe przechowywanie danych przy użyciu magazynu przeglądarki 157
Filtrowanie historii treningów 158
Filtrowanie i sortowanie z użyciem dyrektywy ng-repeat 159
Podsumowanie **163**

Rozdział 4. Tworzenie aplikacji Mój trening 165

Aplikacja Mój trening — zakres problemu **166**
Wymagania aplikacji Mój trening 167
Model aplikacji Mój trening **168**
Współużytkowanie modelu treningu 168
Model jako usługa 169
Układ aplikacji Mój trening **170**
System nawigacyjny aplikacji Mój trening i jej trasy 171
Implementacja list treningów i ćwiczeń **174**
WorkoutService jako repozytorium danych treningów i ćwiczeń 175
Kontrolery list ćwiczeń i treningów 176
Widoki list ćwiczeń i treningów **176**
Tworzenie treningu **177**
Tworzenie lewego paska nawigacyjnego 178
Dodawanie usługi WorkoutBuilderService 179
Dodawanie ćwiczeń z poziomu paska nawigacyjnego 181
Implementacja kontrolera WorkoutDetailController 181
Implementacja widoku do tworzenia treningów 185
Obsługa formularzy w AngularJS 186
Dyrektywa form i sprawdzanie poprawności formularzy 211
Obiekty zasięgu po raz wtóry 221
Podsumowanie **227**

Rozdział 5. Dodawanie warstwy trwałości danych 229

AngularJS i interakcje z serwerem **230**
Przygotowanie trwałego magazynu danych 231
Podstawowe informacje o usłudze $http **233**
Aplikacja Mój trening i jej integracja z serwerem **236**
Wczytywanie danych ćwiczeń i treningów 236
Wykonywanie podstawowych operacji na ćwiczeniach i treningach 252
Usługa $resource **259**
Usługi typu RESTful i ich API 259
Podstawowe informacje o usłudze $resource 261
Wyjaśnienie akcji usługi $resource 262
Wywoływanie akcji zasobów 263
Dostęp do danych ćwiczeń przy użyciu usługi $resource **265**

Kup ksi k Pole ksi k

Podstawowe operacje z użyciem usługi $resource **268**
Stosowanie usługi $resource do obsługi innych rodzajów punktów końcowych 270
Funkcje przechwytujące żądania i odpowiedzi **271**
Zastosowanie funkcji przechwytującej do przekazania klucza API 273
Funkcje przekształcające żądania i odpowiedzi **275**
Obsługa błędów wyznaczania tras w przypadku odrzucenia obietnicy **277**
Obsługa nieodnalezionych treningów 278
Poprawianie aplikacji 7-minutowy trening **280**
Podsumowanie **280**

Rozdział 6. Tworzenie i stosowanie dyrektyw 283

Dyrektywy — wprowadzenie **284**
Anatomia dyrektyw **285**
Tworzenie dyrektywy workout-tile 285
Prezentacja dyrektywy ng-click 289
Tworzenie dyrektywy do zdalnej walidacji nazwy treningu **292**
Dyrektywa remote-validator (dla AngularJS 1.3 i starszych) 293
Dyrektywa remote-validator dla Angular 1.3 299
Aktualizacje modelu z użyciem zdarzenia blur **301**
Zastosowanie właściwości priority do zmiany kolejności wykonywania funkcji compile i link 302
Cykl życia dyrektywy 304
Priorytet a wiele dyrektyw użytych w tym samym elemencie 306
Implementacja dyrektywy wizualnego wskaźnika zdalnej walidacji **307**
Usługa $compile 308
Funkcja kontrolera dyrektywy 310
Komunikacja między dyrektywami — integracja wskaźnika zdalnych operacji i dyrektywy walidacyjnej 310
Wstrzykiwanie kodu HTML w funkcji kompilującej dyrektywy 314
Prezentacja szablonów dyrektyw i transkluzji 315
Wyjaśnienie izolowanego zasięgu dyrektyw **317**
Tworzenie dyrektywy przycisku wykonującego zdalną operację 320
Integracja AngularJS i jQuery **325**
Integracja wtyczki jQuery Owl Carousel z aplikacją Treningomat 328
Tunelowanie zdarzeń jQuery przy użyciu dyrektyw 332
Podsumowanie **335**

Rozdział 7. Testowanie aplikacji AngularJS 337

Potrzeba automatyzacji **338**
Testowanie w AngularJS **338**
Rodzaje testów 339
Kto i kiedy ma testować? 339
Ekosystem testowy AngularJS 340
Wprowadzenie do tworzenia testów jednostkowych **341**
Konfiguracja środowiska Karma do wykonywania testów jednostkowych 342
Zarządzanie zależnościami przy użyciu menedżera zależności Bower 344
Testy jednostkowe komponentów AngularJS 345

Kup ksi k   Pole ksi k

Wprowadzenie do tworzenia testów typu E2E **373**
Prezentacja narzędzia Protractor 374
Przygotowywanie środowiska Protractor do testów E2E 375
Pisanie testów E2E dla naszej aplikacji 376
Przygotowywanie danych na serwerze pod kątem testów E2E 379
Więcej testów E2E 381
Podsumowanie **385**

Rozdział 8. Obsługa często występujących scenariuszy 387

Tworzenie nowych aplikacji **388**
Projekty startowe 388
Yeoman 390
Tworzenie aplikacji wielojęzycznych (umiędzynarodawianie) **392**
Wsparcie dla aplikacji wielojęzycznych w AngularJS 393
Obsługa uwierzytelniania i autoryzacji **398**
Uwierzytelnianie w oparciu o cookies 399
Uwierzytelnianie w oparciu o żetony 401
Obsługa autoryzacji 405
Wzorce komunikacji i współdzielenia danych **408**
Stosowanie adresów URL do przekazywania danych pomiędzy stronami 409
Stosowanie obiektów zasięgu 409
Stosowanie usług 411
Komunikacja między dyrektywami 411
Stosowanie zdarzeń 412
Wydajność **413**
Wytyczne dotyczące wydajności 414
Przydatne biblioteki dodatkowe **426**
Podsumowanie **427**

Skorowidz 429

Kup ksi k Pole ksi k

Kup ksi k
Pole ksi k

Dodawanie warstwy trwałości danych

Nadszedł czas, by porozmawiać z serwerem! Nie ma nic zabawnego w tworzeniu treningu, dodawaniu do niego ćwiczeń i zapisywaniu go tylko po to, by później się okazało, że nasz wysiłek został bezpowrotnie stracony, gdyż dane nie zostały nigdzie trwale zapisane. Musimy coś z tym zrobić.

Aplikacje rzadko kiedy stanowią zamkniętą całość. Każda aplikacja konsumencka, niezależnie od swej wielkości, zawiera fragmenty, które prowadzą interakcję z elementami spoza tej aplikacji. W przypadku aplikacji internetowych te połączenia będą zazwyczaj odwoływać się do serwera. Aplikacje prowadzą interakcję z serwerami, by przeprowadzać uwierzytelnianie i autoryzację użytkowników, zapisywać i pobierać dane, sprawdzać ich poprawność oraz wykonywać inne tym podobne operacje.

Ten rozdział przedstawia konstrukcje i możliwości AngularJS związane z komunikacją klient – serwer. W ramach poznawania tych zagadnień zaimplementujemy warstwę trwałości danych aplikacji *Mój trening*. Warstwa ta będzie wczytywać i zapisywać dane ćwiczeń i treningów na serwerze.

W tym rozdziale omówię następujące zagadnienia:

- **Przygotowanie rozwiązania serwerowego do trwałego przechowania danych treningu —** utworzymy konto w serwisie MongoLab i skorzystamy z jego API typu REST, by nawiązać z nim połączenie i zapisać w nim dane treningu.
- **Przedstawienie usługi `$http` —** jest to jedna z kluczowych usług AngularJS i służy ona do prowadzenia interakcji z serwerami za pomocą protokołu HTTP. Dowiesz się, jak jej używać do wykonywania wszelkiego rodzaju żądań HTTP, takich jak `GET`, `POST`, `PUT` oraz `DELETE`.

Kup ksi k Pole ksi k

- **Implementacja, wczytanie i zapisane danych treningu** — zastosujemy usługę `$http`, by wczytywać i zapisywać dane treningu w bazie danych MongoLab.
- **Tworzenie i używanie obietnic** — w poprzednich rozdziałach spotkałeś się już z obietnicami. W tym rozdziale nie tylko będziesz ich używać (w ramach obsługi wywołań HTTP), lecz także dowiesz się, jak tworzyć własne obietnice i ich używać.
- **Stosowanie odwołań do innych domen** — ponieważ skorzystamy z serwera MongoLab znajdującego się w innej domenie, poznasz ograniczenia, jakie przeglądarki narzucają na takie połączenia. Dowiesz się także, w jaki sposób techniki JSONP i CORS ułatwiają komunikację z innymi domenami oraz poznasz wsparcie dla techniki JSONP dostępne w AngularJS.
- **Stosowanie usługi `$resource` do tworzenia punktów końcowych typu RESTful** — usługa `$resource` jest abstrakcją utworzoną w oparciu o usługę `$http` i służącą do obsługi udostępnianych przez serwery punktów końcowych typu RESTful. Już wcześniej poznałeś tę usługę i sposoby jej stosowania.
- **Wczytywanie i zapisywanie ćwiczeń przy użyciu usługi `$resource`** — zmodyfikujemy nieco fragmenty naszego systemu, by zastosować usługę `$resource` do wczytywania i zapisywania danych ćwiczeń.
- **Funkcje przechwytujące żądania i odpowiedzi** — dowiesz się, czym są funkcje przechwytujące i jak można ich używać do przechwytywania wywołań w potokach żądań i odpowiedzi oraz do modyfikowania sposobu obsługi zdalnych wywołań.
- **Funkcje przekształcające żądania i odpowiedzi** — podobnie jak funkcje przechwytujące operują one na poziomie zawartości komunikatów. Poznasz dokładnie ich działanie na kilku przykładach.

A zatem zaczynamy!

AngularJS i interakcje z serwerem

Niemal wszystkie interakcje pomiędzy klientem i serwerem sprowadzają się zazwyczaj do wysyłania na serwer żądań HTTP i odbierania przesyłanych z niego odpowiedzi. W przypadku aplikacji JavaScriptu komunikacja z serwerem jest zazwyczaj realizowana przy użyciu technologii AJAX. AngularJS udostępnia dwie usługi służące do obsługi takiej komunikacji:

- `$http` — to podstawowy komponent frameworku, który służy do prowadzania interakcji ze zdalnymi serwerami przy użyciu komunikacji bazującej na technologii AJAX. Można ją porównać z funkcją `ajax` biblioteki jQuery, która działa w podobny sposób.
- `$resource` — ta usługa stanowi abstrakcję utworzoną w oparciu o usługę `$http`, której zadaniem jest ułatwianie komunikacji z usługami typu RESTful (*https://pl.wikipedia.org/wiki/Representational_State_Transfer*).

Zanim jednak zaczniesz szczegółowo poznawać obie powyższe usługi, musimy przygotować platformę serwerową, której będziemy używać do przechowywania danych i zarządzania nimi.

Kup ksi k Pole ksi k

Przygotowanie trwałego magazynu danych

Do zapewnienia trwałości danych wykorzystamy bazę danych MongoDB (*https://www.mongodb.org/*), która służy do przechowywania dokumentów, przy czym do utworzenia bazy i zarządzania nią skorzystamy z usługi MongoLab (*https://mongolab.com/*). Wybraliśmy usługę MongoLab, gdyż udostępnia ona interfejs pozwalający na prowadzenie bezpośredniej interakcji z bazą danych. Dzięki temu będziemy mogli uniknąć instalowania na serwerze dodatkowego oprogramowania do obsługi i udostępniania baz danych MongoDB.

> Udostępnianie magazynu danych bądź bazy danych klientom nigdy nie jest dobrym pomysłem. Ponieważ jednak naszym celem w tej książce jest nauka AngularJS i sposobów prowadzenia komunikacji kilient – serwer, skorzystamy z takiej możliwości i będziemy korzystać bezpośrednio z bazy MongoDB utworzonej w serwisie MongoLab.
>
> Istnieje także nowy rodzaj aplikacji, stworzonych w oparciu o rozwiązania typu *noBackend*. W ich przypadku programiści aplikacji klienckich nie wiedzą, jakie konkretne rozwiązanie serwerowe jest używane. Wszelkie interakcje z serwerem ograniczają się do wywoływania funkcji API udostępnianych przez serwer. Więcej informacji na temat rozwiązań typu *noBackend* można znaleźć na stronie *http://nobackend.org/*.

Naszym pierwszym zadaniem będzie założenie konta w serwisie MongoLab i utworzenie bazy danych. Poniżej opisałem czynności, które należy w tym celu wykonać:

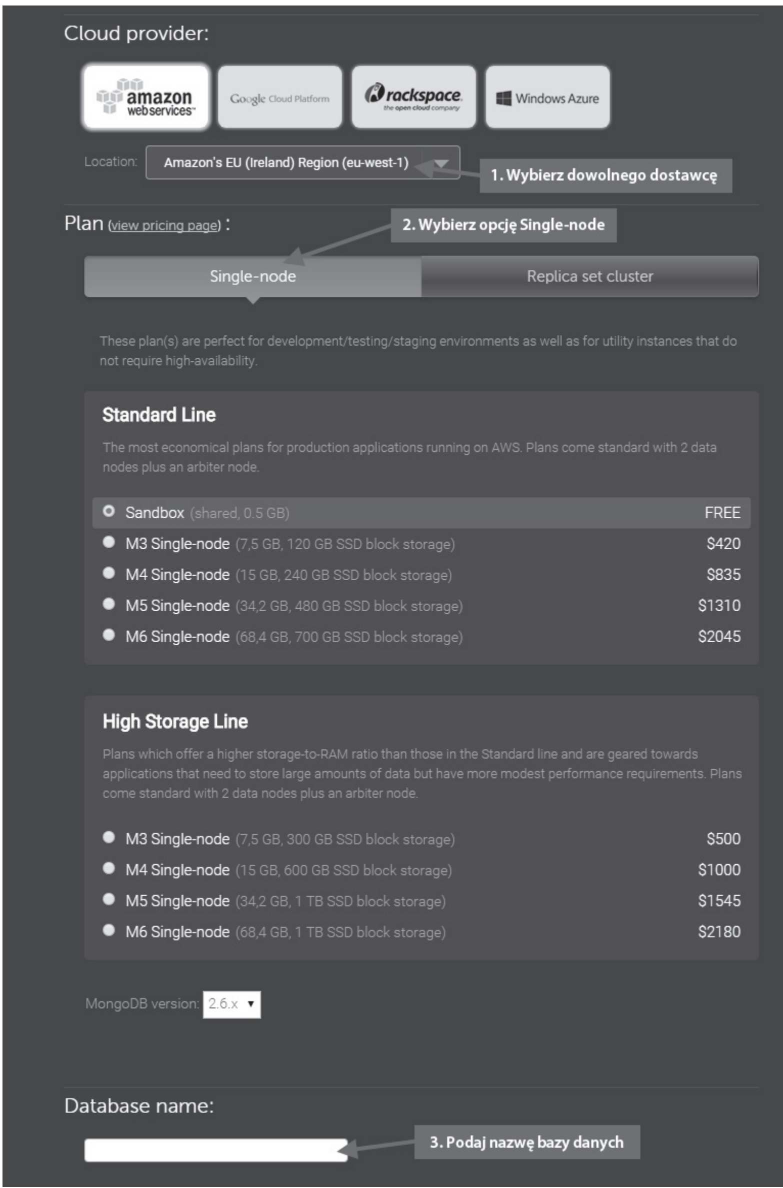
1. Wejdź na stronę *https://mongolab.com/* i utwórz nowe konto, postępując zgodnie z wyświetlanymi instrukcjami.
2. Po utworzeniu konta zaloguj się na nie i utwórz nową bazę danych, klikając przycisk *Create New* umieszczony na stronie głównej konta.

   Na kolejnej stronie podaj kilka informacji i ustawień dotyczących tworzonej bazy danych. Rysunek 5.1 ilustruje, w jaki sposób można utworzyć bezpłatną bazę danych i jakie informacje należy podać.
3. Utwórz bazę danych i zapisz jej nazwę.
4. Po utworzeniu bazy danych otwórz ją, przejdź na kartę *Collection* i utwórz dwie kolekcje:
   - `exercises` — do przechowywania dane ćwiczeń aplikacji *Moj trening*;
   - `workouts` — do przechowywania danych treningów.

   W świecie MongoDB kolekcje odpowiadają tabelom baz danych.

> MongoDB należy do rodzaju baz danych nazywanych bazami dokumentowymi. W bazach tego typu podstawowymi pojęciami są dokumenty, atrybuty oraz ich wzajemne powiązania. Ponadto, w odróżnieniu od tradycyjnych baz danych, schemat baz MongoDB nie jest sztywny.
>
> W tym rozdziale nie będziemy się zajmować wyjaśnianiem, czym są dokumentowe bazy danych ani w jaki sposób należy opracować strukturę magazynu używanego przez naszą aplikację. Aplikacja *Mój trening* ma niewielkie wymagania, jeśli chodzi o bazę danych, i na jej potrzeby w zupełności wystarczą dwie kolekcje dokumentów. Właściwie nawet nie będziemy używali dokumentowej bazy danych w prawdziwym sensie tego terminu.

Kup ksi k   Pole ksi k



Rysunek 5.1. Tworzenie bezpłatnej bazy danych w serwisie MongoLab

5. Po dodaniu kolekcji dodaj konto użytkownika bazy danych; można to zrobić na karcie *User*.
6. Kolejnym krokiem jest określenie klucza API konta MongoLab. Ten klucz będzie następnie dodawany do wszystkich żądań przesyłanych do serwisu MongoLab. Oto, co należy zrobić, by pobrać klucz API:

Kup ksi k

Pole ksi k

1. Kliknij nazwę użytkownika (lecz nie nazwę konta) wyświetloną w prawym górnym rogu, aby otworzyć profil użytkownika.
2. Skopiuj bieżący klucz API wyświetlony w sekcji *API Key*.

7. Ostatnią czynnością jest włączenie możliwości korzystania z API typu RESTful zapewniającego dostęp do danych. W tym celu należy kliknąć przycisk *Enable Data API access* i potwierdzić żądanie w wyświetlonym oknie dialogowym, klikając przycisk *Enable*.

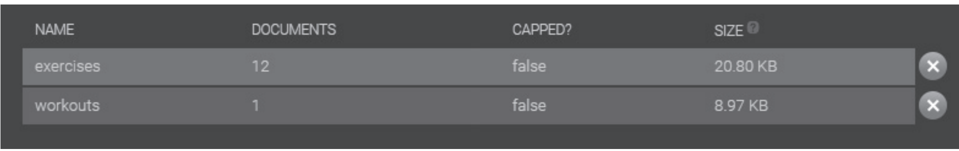
W ten sposób utworzyliśmy bazę danych i jej schemat; teraz musimy zająć się utworzeniem początkowej zawartości naszej bazy.

Utworzenie początkowej zawartości bazy danych

Aplikacja *Mój trening* zawiera już domyślny trening składający się z 12 ćwiczeń. Musimy teraz zapisać te dane w bazie danych.

W pierwszej kolejności otwórzmy plik *seed.js* dostępny w przykładach dołączonych do książki, w katalogu *rozdzial05\punkt_kontrolny_1\app\js*. Zawiera on dane treningu i ćwiczeń zapisane w formacie JSON oraz szczegółowe instrukcje dotyczące tego, w jaki sposób należy je zapisać w bazie danych w serwisie MongoLab.

Po wypełnieniu danymi początkowymi nasza baza danych będzie zawierać 1 trening w kolekcji `workouts` i 12 ćwiczeń w kolekcji `exercises`. Można to sprawdzić na stronie MongoLab, kolekcje powinny wyglądać jak na rysunku 5.2.



NAME	DOCUMENTS	CAPPED?	SIZE
exercises	12	false	20.80 KB
workouts	1	false	8.97 KB

Rysunek 5.2. Kolekcje bazy danych po wypełnieniu danymi początkowymi

Teraz wszystko już jest przygotowane, możemy zatem zająć się poznawaniem usługi `$http` i implementacją zapewniania trwałości danych aplikacji *Mój trening*.

Podstawowe informacje o usłudze $http

`$http` jest podstawową usługą AngularJS, która służy do wykonywania żądań AJAX. Usługa ta udostępnia interfejs API umożliwiający wykonywanie wszystkich operacji (żądań) HTTP, takich jak `GET`, `POST`, `PUT`, `DELETE` i kilka innych.

Komunikacja HTTP ma charakter asynchroniczny. Wysyłając żądanie HTTP, przeglądarka nie czeka na otrzymanie odpowiedzi, tylko zaraz po jego wysłaniu kontynuuje przetwarzanie. Oznacza to, że musimy zarejestrować jakąś funkcję zwrotną, która zostanie wywołana po otrzy-

Kup ksi k Pole ksi k

maniu odpowiedzi z serwera. Obietnice (ang. *promise*) AngularJS i ich API ułatwiają obsługę asynchronicznej komunikacji i są powszechnie używane wraz z usługą `$http`, o czym się niebawem przekonasz.

Poniższy przykład przedstawia podstawową składnię usługi `$http`:

```
$http(config)
```

Wywołanie usługi wymaga przekazania obiektu konfiguracyjnego i zwraca obietnicę. Obiekt `config` zawiera zestaw właściwości określających postać i sposób wykonania żądania. Właściwości te określają między innymi typ operacji HTTP (`GET`, `POST`, `PUT` itd.), adres URL zdalnego serwera, parametry łańcucha zapytania i wysyłane nagłówki.

Wszelkie szczegółowe informacje dotyczące opcji konfiguracyjnych usługi `$http` są dostępne w jej dokumentacji, na stronie *https://code.angularjs.org/1.3.3/docs/api/ng/service/$http*. W dalszej części rozdziału, podczas prac nad implementacją aplikacji, skorzystamy z kilku tych opcji.

Wywołanie usługi `$http` zwraca obiekt obietnicy. Oprócz standardowych funkcji tworzących API obietnic (takich jak `then`) ten obiekt zawiera także dwie dodatkowe funkcje zwrotne, `success` i `error`, które są wywoływane zależnie od tego, czy żądanie HTTP zakończyło się pomyślnie, czy też w trakcie jego wykonywania wystąpiły błędy.

Oto podstawowy sposób wykonywania żądań HTTP przy użyciu usługi `$http`:

```
$http({method: 'GET', url: '/endpoint'}).
    success(function(data, status, headers, config) {
        // wywoływana, gdy żądanie zostanie wykonane prawidłowo
    }).
    error(function(error, status, headers, config) {
        // wywoływana, gdy podczas wykonywania żądania wystąpią błędy
        // parametr error zawiera przyczynę problemu
    });
```

Powyższy kod przesyła żądanie HTTP pod adres `/endpoint`, a później, kiedy odpowiedź jest już dostępna, wywołuje albo funkcję zwrotną `success`, albo `error`.

> Odpowiedzi HTTP o kodach z zakresu od 200 do 299 są uznawane za prawidłowe. Odpowiedzi o kodach 40x i 50x są traktowane jako błędy i powodują wywołanie funkcji zwrotnej `error`.

Funkcje zwrotne `success` i `error` mają cztery parametry:

- `data` lub `error` — zawiera odpowiedź zwróconą przez serwer, mogą to być dane lub, w przypadku żądania zakończonego niepowodzeniem, informacje o błędzie.
- `status` — zawiera kod statusu odpowiedzi HTTP.
- `headers` — zawiera nagłówki odpowiedzi HTTP.
- `config` — zawiera obiekt konfiguracyjny przekazany w wywołaniu usługi `$http`.

Kup ksi k

Pole ksi k

W przypadku żądań AJAX przedstawiona powyżej składnia usługi `$http` jest stosowana bardzo rzadko. Usługa ta udostępnia grupę metod pomocniczych, upraszczających generowanie konkretnych typów żądań HTTP. Można do niej zaliczyć następujące metody:

- `$http.get(url, [config])`,
- `$http.post(url, data, [config])`,
- `$http.put(url, data, [config])`,
- `$http.delete(url, [config])`,
- `$http.head(url, [config])`,
- `$http.jsonp(url, [config])`.

W wywołaniach każdej z tych funkcji można przekazać (jako ostatni argument) ten sam (opcjonalny) obiekt `config`.

Interesującym aspektem standardowej konfiguracji usługi `$http` jest to, że zapewnia ona bardzo łatwą obsługę danych w formacie JSON. Oto jej najważniejsze efekty:

- W przypadku operacji `GET`, jeśli odpowiedź zawiera dane w formacie JSON, to framework automatycznie przetworzy przekazane dane w formie łańcuchowej na obiekt JavaScriptu. W efekcie pierwszy parametr funkcji zwrotnej `success` (`data`) będzie zawierał nie łańcuch znaków, lecz obiekt JavaScriptu.
- W przypadku wykonywania żądań `POST` lub `PUT` obiekty są automatycznie serializowane, a przed wysłaniem żądania określana jest odpowiednia wartość nagłówka typu zawartości (`Content-Type: application/json`).

Czy to oznacza, że usługa `$http` nie potrafi obsługiwać danych w innych formatach? Absolutnie nie. `$http` jest usługą o charakterze ogólnym, która generuje żądania AJAX i może obsługiwać dowolne formaty żądań i odpowiedzi. Każde żądanie AJAX w AngularJS jest wykonywane przy użyciu usługi `$http` — bezpośrednio lub pośrednio. Na przykład zdalne widoki wczytywane za pomocą dyrektyw `ng-view` lub `ng-include` w niewidoczny dla nas sposób korzystają właśnie z usługi `$http`.

> Warto zajrzeć do przykładu opublikowanego w serwisie jsFiddle na stronie *http://jsfiddle.net/cmyworld/doLhmgL6/*, który przedstawia zastosowanie usługi `$http` do przesyłania na serwer danych w formacie tradycyjnie używamy przez formularze HTML korzystające z metody `POST` (`'Content-Type': 'application/x-www-form-urlencoded'`).

To wyłącznie AngularJS sprawia, że korzystanie z danych w formacie JSON jest tak łatwe, eliminując konieczność pisania powtarzającej się logiki serializacji i deserializacji oraz ustawiania odpowiednich nagłówków HTTP, których zazwyczaj wymaga obsługa danych w formacie JSON.

Dysponując już podstawową znajomością usługi `$http`, możemy jej użyć do zaimplementowania czegoś użytecznego. Spróbujmy zatem zapewnić trwałość danych w naszej aplikacji.

Kup ksi k Pole ksi k

Aplikacja Mój trening i jej integracja z serwerem

Zgodnie z informacjami podanymi w poprzednim podrozdziale wszelkie interakcje klient – serwer mają charakter asynchroniczny. Rozwiązanie to stanie się bardziej zrozumiałe, kiedy zmodyfikujemy aplikację *Mój trening* tak, by wczytywała dane z serwera.

W poprzednim rozdziale zbiór domyślnych ćwiczeń i jeden trening zostały na stałe zakodowane w implementacji usługi `WorkoutService`. Przekonajmy się, w jaki sposób można je wczytać z serwera.

Wczytywanie danych ćwiczeń i treningów

Na początku tego rozdziału zapisaliśmy w bazie danych informacje z pliku *seed.js*. Teraz zależy nam na tym, by w jakiś sposób wyświetlić te dane w widokach aplikacji. Zastosujemy do tego REST API usługi MongoLab.

> REST API serwisu MongoLab korzysta z klucza API do autoryzacji wszystkich żądań. Każde żądanie kierowane do punktów końcowych MongoLab musi zawierać w łańcuchu zapytania parametr o postaci `apikey=`*<klucz>*, gdzie *<klucz>* to klucz API użytkownika serwisu odczytany podczas przygotowywania bazy zgodnie z informacjami podanymi we wcześniejszej części rozdziału. Trzeba pamiętać, że ten klucz jest zawsze skojarzony z kontem użytkownika. dlatego nie należy udostępniać go innym osobom.

API MongoLab zapewnia możliwość przeglądania i aktualizacji danych przy użyciu adresów o przewidywalnej postaci. Poniżej przedstawiam typowe wzorce dostępu do punktów końcowych dowolnych kolekcji MongoDB (zakładając, że adres URL bazy danych ma postać `https://api.mongolab.com/api/1/databases`):

- *<bazadanych>*`/collections/`*<nazwa>*`?apiKey=`*<klucz>* — udostępnia dwa typy żądań:
  - `GET` — pobiera wszystkie obiekty dostępne w kolekcji o podanej nazwie.
  - `POST` — dodaje nowy obiekt do kolekcji o podanej nazwie. MongoLab używa właściwości `_id` w celu unikalnej identyfikacji dokumentów (obiektów). Jeśli przesłane dane nie będą zawierać tej właściwości, to zostanie ona wygenerowana automatycznie.
- *<bazadanych>*`/collections/`*<nazwa>*`/`*<id>*`?apiKey=`*<klucz>* — udostępnia trzy typy żądań:
  - `GET` — pobiera z kolekcji o podanej nazwie konkretny dokument (element kolekcji) określony za pomocą identyfikatora (*id*, który będzie dopasowywany do właściwości `_id`).
  - `PUT` — aktualizuje określony element (*id*) kolekcji o podanej nazwie.
  - `DELETE` — usuwa z kolekcji o podanej nazwie element o określonym identyfikatorze (*id*).

Szczegółowe informacje na temat interfejsu REST API można znaleźć w dokumentacji serwisu MongoLab, na stronie *http://docs.mongolab.com/data-api/*.

Teraz nic już nie stoi na przeszkodzie, byśmy zajęli się implementacją stron z listami ćwiczeń i treningów.

Zanim zaczniemy, należy skopiować z przykładów dołączonych do książki działającą wersję aplikacji *Mój trening*, dostępną w katalogu *rozdzial04\punkt_kontrolny_7*. Jest to pełna wersja aplikacji zawierająca także działający formularz do tworzenia i edycji ćwiczeń, który mieliśmy zaimplementować samodzielnie, w ramach ćwiczenia.

Wczytywanie list ćwiczeń i treningów z serwera

Aby pobrać listy ćwiczeń i treningów z bazy danych w serwisie MongoLab, musimy zmodyfikować dwie metody usługi `WorkoutService` — `getExercises` i `getWorkouts`.

Poniżej przedstawiam zmiany, które należy wprowadzić w kodzie metody `getExercises`, w pliku *services.js* umieszczonym w katalogu *app\js\shared*:

```
service.getExercises = function () {
  var collectionsUrl =
            "https://api.mongolab.com/api/1/databases/<nazwa>/collections";
  return $http.get(collectionsUrl + "/exercises", {
          params: { apiKey: '<klucz>'}
  });
};
```

W powyższym kodzie fragmenty `<nazwa>` i `<klucz>` należy zastąpić odpowiednio nazwą bazy danych w serwisie MongoLab i kluczem API.

Ponadto koniecznie należy pamiętać o dodaniu do deklaracji usługi `WorkoutService` zależności od usługi `$http`.

Powyższa, zmodyfikowana wersja funkcji tworzy adres URL odwołujący się do serwisu MongoLab, a następnie wywołuje funkcję `$http.get`, by pobrać listę ćwiczeń. Pierwszym argumentem jej wywołania jest adres URL, na jaki należy wysłać żądanie, a drugim obiekt konfiguracyjny (`config`).

Właściwość `params` obiektu konfiguracyjnego zapewnia możliwość dodawania do adresu URL parametrów łańcucha zapytania. W naszym przypadku parametrem tym będzie klucz API (`?apiKey=98dkdd`) zapewniający dostęp do API MongoLab.

Ta nowa wersja funkcji `getExercises` zwraca obietnicę, a zatem musimy zmodyfikować kod, który jej używa.

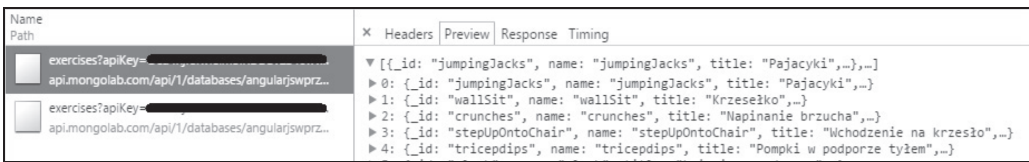
Kup ksi k Pole ksi k

W tym celu musimy otworzyć plik *exercise.js* znajdujący się w katalogu *WorkoutBuilder* i zmodyfikować kontroler `ExerciseListController`, a konkretnie zastąpić istniejącą implementację funkcji `init` (kod umieszczony wewnątrz niej) poniższą instrukcją:

```
WorkoutService.getExercises().success(function (data) {
    $scope.exercises = data;
});
```

Zastosowaliśmy w niej funkcję zwrotną `success` obietnicy, by dowiązać listę ćwiczeń do kontrolera. Powyższy kod wyraźnie pokazuje asynchroniczne działanie usługi `$http` i obietnic, gdyż dane ćwiczeń są zapisywane we właściwości `exercises` w funkcji zwrotnej wywoływanej dopiero po odebraniu odpowiedzi z serwera.

Spróbujmy teraz wyświetlić stronę z listą ćwiczeń w przeglądarce (`#/builder/exercises`) i upewnić się, że zostanie ona wczytana z serwera. Lista żądań sieciowych wykonanych przez przeglądarkę powinna zawierać pozycje takie jak na rysunku 5.3.



Rysunek 5.3. Żądania pobierające dane z serwera MongoLab

Jak widać, ćwiczenia są wczytywane prawidłowo. A co z treningami? Wczytywanie listy treningów można poprawić w podobny sposób.

Zmodyfikujmy zatem funkcję `getWorkouts` w usłudze `WorkoutService` tak, by wczytywała dane z serwera. Implementacja tej funkcji będzie bardzo podobna do implementacji funkcji `get`↳`Exercises`, z tą różnicą, że adres URL będzie się odwoływał do kolekcji o nazwie `workouts`. Następnie musimy także zmodyfikować funkcję `init` kontrolera `WorkoutListController`, analogicznie jak wcześniej zmodyfikowaliśmy funkcję `init` usługi `ExerciseListController`.

To było łatwe! Tak samo możemy poprawić wszystkie pozostałe fragmenty aplikacji pobierające dane treningów i ćwiczeń. Niemniej zanim to zrobimy, warto zwrócić uwagę, że wciąż można poprawić implementację aplikacji.

Pierwszym problemem związanym z aktualną implementacją funkcji `getExercises` i `getWorkouts` jest to, że nazwa bazy danych i klucz API są podane na stałe w kodzie, co w przyszłości może utrudnić jego utrzymanie. Optymalnym rozwiązaniem będzie wstrzyknięcie tych wartości do usługi `WorkoutService`.

Bazując na zdobytej wcześniej wiedzy i znajomości AngularJS, wiemy, że gdyby `WorkoutService` była usługą typu `provider`, to moglibyśmy przekazać do niej dane konfiguracyjne niezbędne do odpowiedniego przygotowania usługi na etapie ładowania aplikacji. To rozwiązanie pozwoliłoby nam skonfigurować usługę przed jej użyciem. Nadszedł zatem czas, by zastosować tę teorię w praktyce!

Kup ksi k

Pole ksi k

Implementacja dostawcy usługi WorkoutService

Implementacja usługi `WorkoutService` w formie dostawcy pozwoli nam określić używane przez nią informacje — nazwę bazy danych i klucz API — na etapie konfiguracji.

Skopiujmy zatem definicję usługi `WorkoutService` z pliku *services.js* dostępnego w przykładach dołączonych do książki, w katalogu *rozdzial05\punkt_kontrolny_1\app\js\shared*, i wklejmy go do analogicznego pliku w tworzonej aplikacji. W ten sposób zamienimy wcześniejszą usługę na usługę typu `provider`.

Nowa wersja usługi definiuje metodę `configure`, odpowiadającą za określenie nazwy bazy danych i klucza API oraz przygotowanie adresu URL połączenia. Oto jej kod:

```
this.configure = function (dbName, key) {
    database = database;
    apiKey = key;
    collectionsUrl = apiUrl + dbName + "/collections";
}
```

Usunęliśmy także funkcje `setupInitialExercises`, `setupInitialWorkouts` oraz `init`, gdyż dane będą teraz pobierane z serwera MongoLab.

Zmieniliśmy również implementacje funkcji `getExercises` i `getWorkouts` tak, by korzystały z przygotowanych wcześniej parametrów:

```
service.getExercises = function () {
    return $http.get(collectionsUrl + "/exercises", { params: { apiKey: apiKey }
});
};
```

I w końcu utworzenie obiektu `service` zostało przeniesione do funkcji `$get` dostawcy. `$get` jest funkcją wytwórczą odpowiedzialną za utworzenie właściwej usługi.

Kolejną czynnością będzie zmodyfikowanie funkcji `config` modułu `app` i wstrzyknięcie konfiguracji serwera MongoLab do nowej implementacji usługi `WorkoutService` (za pomocą `Workout➥ServiceProvider`).

Otwórzmy zatem plik *app.js* i w definicji funkcji `config` dodajmy do już istniejącej listy dostawców zależność od dostawcy `WorkoutServiceProvider`. Ponadto w kodzie tej funkcji musimy dodać wywołanie metody `configure` tego dostawcy, określające nazwę używanej bazy danych i klucz API:

```
WorkoutServiceProvider.configure("<mojaBaza>", "<mójKluczAPI>");
```

Ta nowa implementacja usługi `WorkoutService` jest znaczenie lepsza od poprzedniej, ponieważ zapewnia możliwość konfiguracji usługi przez jej użyciem.

Kup ksi k Pole ksi k

Zastosowanie usługi typu `provider` może się wydawać nadmiernie skomplikowanym rozwiązaniem, gdyż podobny efekt można by uzyskać, tworząc usługę typu `constant`, taką jak poniżej:

```
angular.module('app').constant('dbConfig', {
    database: "<mojaBaza>",
    apiKey: "<mójKluczAPI>"
});
```

A następnie wstrzykując ją do istniejącej implementacji usługi `WorkoutService`.

Zastosowanie usługi typu `provider` ma jednak tę zaletę, że dane konfiguracyjne nie są dostępne globalnie. W przypadku użycia usługi typu `constant`, takiej jak `dbConfig`, każda inna usługa lub kontroler mogłyby uzyskać dostęp do nazwy bazy danych lub klucza API — wystarczyłoby wstrzyknąć usługę `dbConfig` — a to byłoby raczej niepożądane.

Przedstawione do tej pory modyfikacje wciąż nie są jeszcze kompletne, o czym można się przekonać, odświeżając stronę z listą treningów. W konsoli przeglądarki zostanie wyświetlony komunikat `unknown provider WorkoutServiceProvider`[1].

Mamy tu do czynienia z błędem AngularJS związanym z tym, że funkcja `config` modułu została wywołana przed zarejestrowaniem dostawcy. Przyczyną tego błędu jest kolejność odwołań do skryptów w pliku *index.html*, a konkretnie to, że odwołanie do skryptu *app.js* jest umieszczone przed odwołaniem do skryptu *service.js*.

Powyższy błąd został zgłoszony (*https://github.com/angular/angular.js/issues/7139*) i opracowano metodę jego ominięcia — polega ona na wywołaniu funkcji `config` na samym końcu, już po zarejestrowaniu wszystkich dostawców i usług. Wymaga to jednak przeniesienia kodu funkcji `config` do nowego pliku.

> Powyższy błąd faktycznie występował w czasie, kiedy pisałem ten rozdział, ale w nowszych wersjach AngularJS 1.3 problem ten został rozwiązany. Pomimo to wciąż będziemy korzystać z przedstawionego poniżej rozwiązania, gdyż zapewnia ono prawidłowe działanie kodu niezależnie od używanej wersji AngularJS.

Aby rozwiązać opisany wcześniej problem, musimy skopiować do roboczej wersji aplikacji pliki *app.js* i *config.js* dostępne w przykładach dołączonych do książki, w katalogu *rozdzial05\punkt_kontrolny_1*. Następnie trzeba zaktualizować wywołanie funkcji `configure` usługi `Workout`
`↪ServiceProvider` w pliku *config.js*, podając w nim nazwę bazy danych i klucz API. Nie możemy także zapomnieć o dodaniu odwołania do pliku *config.js* na samym końcu listy odwołań w pliku *index.html*.

Jeśli teraz odświeżymy stronę z listą ćwiczeń lub treningów, przekonamy się, że dane są prawidłowo pobierane z serwera bazy danych.

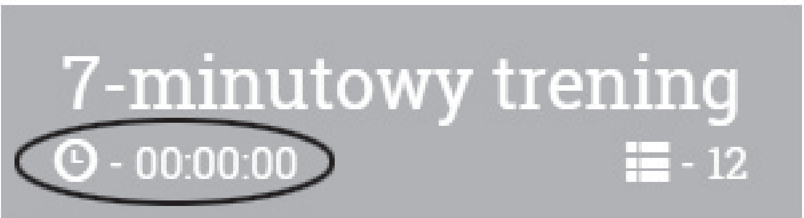
---

[1] Nieznany dostawca `WorkoutServiceProvider` — *przyp. tłum.*

Kup ksi k Pole ksi k

> Jeśli masz problemy z tworzoną implementacją aplikacji, to jej działającą wersję znajdziesz w przykładach dołączonych do książki, w katalogu *rozdzial05\punkt_kontrolny_1*.
>
> Przed próbą uruchomienia tego kodu należy także pamiętać, by zmienić nazwę bazy danych i klucz API.

Wszystko wygląda dobrze i lista jest wyświetlana prawidłowo. No, prawie! Jest pewien drobny problem z listą treningów. Jeśli dokładnie przyjrzymy się elementom tej listy (a właściwie jedynemu jej elementowi, przedstawionemu na rysunku 5.4), to zauważymy go bez trudu:



Rysunek 5.4. Drobny błąd na liście treningów

Okazuje się, że po wprowadzeniu zmian przestało działać wyliczanie czasu trwania treningu! Ale dlaczego tak się stało? Aby odpowiedzieć na to pytanie, musimy przeanalizować, w jaki sposób zostały zaimplementowane te obliczenia. Usługa `WorkoutService` (w pliku *model.js*) definiuje funkcję `totalWorkoutDuration`, która służy do wyliczania czasu trwania treningu.

Różnica polega na zawartości tablicy treningów dowiązanej do widoku. W poprzednich rozdziałach tworzyliśmy tablicę z obiektami modelu, tworzonymi przy użyciu usługi `WorkoutPlan`. Teraz natomiast dane są pobierane z serwera, do widoku dowiązywana jest zwyczajna tablica obiektów JavaScriptu, które z oczywistych powodów nie dysponują żadną logiką do wykonywania obliczeń.

Problem ten możemy jednak rozwiązać — wystarczy odwzorować odpowiedź przesłaną z serwera na obiekty klasy modelu i zwrócić tablicę tych obiektów.

Odwzorowywanie danych z serwera na modele aplikacji

Odwzorowywanie danych z serwera na model aplikacji i na odwrót nie jest konieczne, jeśli definicje modelu i postać danych przechowywanych na serwerze odpowiadają sobie. Jeśli spojrzymy na klasę modelu `Exercise` i na dane ćwiczeń, które zapisaliśmy w bazie MongoLab, okaże się, że są identyczne, więc żadne odwzorowywanie nie jest konieczne.

Odwzorowywanie odpowiedzi nadesłanej z serwera na dane modelu nabiera znaczenia gdy:

- model definiuje jakiekolwiek funkcje;
- dane modelu zapisane w bazie mają inną strukturę niż ich reprezentacja używana w kodzie;

Kup ksi k

Pole ksi k

- ta sama klasa modelu jest używana do reprezentacji danych pochodzących z różnych źródeł (co może się zdarzyć w aplikacjach typu mashup, łączących dane pochodzące z różnych źródeł).

Usługa `WorkoutPlan` jest doskonałym przykładem znaczenia różnic między reprezentacją modelu i jego trwale przechowywanymi danymi. Aby zauważyć te różnice, wystarczy przyjrzeć się danym przedstawionym na rysunku 5.5.

Model

```
WorkoutPlan {
    name: "7minWorkout",
    title: "7-minutowy trening",
    description: "Intensywny trening składający się z 12 ćwiczeń.",
    restBetweenExercises: 30,
    exercises : [
      { details: Exercise, duration: 30 },
      { details: Exercise, duration: 30 },
      ...
      { details: Exercise, duration: 30 }
    ],
    totalWorkoutDuration: function() {...}
};
```

```
Exercise(args) {
    name = "jumpingJacks";
    title = "Pajacyki";
}
```

Dane przechowywane na serwerze

```
WorkoutPlan {
    "_id": "7minWorkout",
    "name": "7minWorkout",
    "title": "7-minutowy trening",
    "description": "Intensywny trening składający się z 12 ćwiczeń.",
    "restBetweenExercises": 30,
    "exercises" : [
      { "name": "jumpingJacks", "duration": 30 },
      { "name": "wallSit", "duration": 30},
      ...
      { "name": "sidePlank", "duration": 30},
    ]
};
```

Rysunek 5.5. Postać danych modelu i danych przechowywanych na serwerze

Oto dwie podstawowe różnice między danymi modelu i danymi przechowywanymi na serwerze:

- Model definiuje funkcję `totalWorkoutDuration`.
- Inna jest reprezentacja tablicy `exercises`. W przypadku modelu tablica `exercises` zawiera obiekty `Exercise` (zapisane we właściwości `details`), natomiast w danych przechowywanych na serwerze znajduje się wyłącznie identyfikator lub nazwa ćwiczenia.

To wyraźnie pokazuje, że wczytywanie i zapisywanie treningów wymaga odwzorowywania modelu. Dla zachowania spójności mamy zamiar zaimplementować odwzorowywanie danych, i to zarówno danych ćwiczeń, jak i treningów.

W pierwszej kolejności musimy zmienić implementację funkcji `getExercises` usługi `Workout`
`↪Service`:

```
service.getExercises = function () {
    return $http.get(collectionsUrl + "/exercises", { params: { apiKey: apiKey }
})
        .then(function (response) {
            return response.data.map(function (exercise) {
                return new Exercise(exercise);
            })
        });
};
```

A ponieważ obecnie wynikiem zwracanym przez funkcję `getExercises` nie jest obiekt obietnicy zwracany przez usługę `$http` (więcej o niej w dalszej części tego podpunktu), lecz standardowy

Kup ksi k Pole ksi k

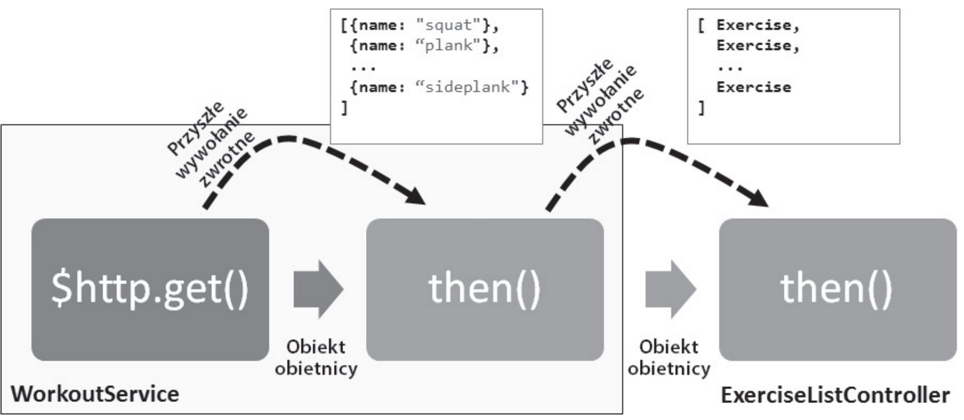
obiekt obietnicy, to wszędzie tam, gdzie wywołujemy funkcję `getExercises`, zamiast funkcji `success` musimy użyć funkcji `then`.

Zmieńmy więc w poniższy sposób implementację funkcji `init` w kontrolerach `ExercisesNav↪Controller` i `ExerciseListController`:

```
var init = function () {
    WorkoutService.getWorkouts().then(function (data) {
        $scope.workouts = data;
    });
};
```

Spójrzmy teraz na fragment kodu funkcji `getExercises` wyróżniony pogrubieniem. Dzieje się w nim kilka interesujących rzeczy, które należy zrozumieć:

- Po pierwsze, w funkcji zwrotnej przekazywanej w wywołaniu funkcji `then` (jako jej pierwszy parametr) wywołujemy funkcję `Array.map`, która odwzorowuje tablicę ćwiczeń zwróconą z serwera na tablicę obiektów `Exercise`. Ogólnie rzecz biorąc, funkcja `Array.map` jest zazwyczaj używana do odwzorowywania jednej tablicy na inną. Więcej informacji dotyczących działania tej funkcji można znaleźć w dokumentacji Mozilla Developer Network (MDN), na stronie *https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map*.
- Po drugie, ten kod stanowi doskonały przykład zastosowania łańcuchów obietnic. Pierwsza obietnica jest zwracana przez wywołanie `$http.get`. Do niej dołączamy funkcję zwrotną `then`, która także zwraca obietnicę, i to właśnie ten obiekt zostanie zwrócony do kodu wywołującego jako wynik. Cały ten proces można zilustrować za pomocą diagramu przedstawionego na rysunku 5.6.



Rysunek 5.6. Schematyczna postać łańcucha obietnic

W przyszłości, po wyznaczeniu pierwszej obietnicy (`$http.get`), zostanie wywołana funkcja zwrotna `then`, do której zostanie przekazana lista ćwiczeń pobrana z serwera. Funkcja zwrotna `then` przetworzy odpowiedź i zwróci nową tablicę obiektów `Exercise`. Ta tablica zostanie następnie użyta podczas wyznaczania kolejnej obietnicy i przekazana do funkcji zwrotnej zdefiniowanej w kontrolerze `ExerciseListController`.

> Obietnica zwrócona przez funkcję `then` jest wyznaczana przez wartość wynikową zwróconą przez jej funkcję zwrotną `success` lub `error`.

Dopiero funkcja zwrotna `then` zdefiniowana w kontrolerze `ExerciseListController` zapisuje przekazaną tablicę obiektów `Exercise` w zmiennej `exercises`. Na diagramie przedstawionym na rysunku 5.6 dane przekazywane w procesie wyznaczania poszczególnych obietnic zostały przedstawione nad przerywanymi strzałkami.

Łańcuch obietnic działa jak potok do przetwarzania odpowiedzi; stanowi on wzorzec zapewniający ogromne możliwości i, jak pokazał powyższy przykład, można go używać do wykonywania naprawdę interesujących rzeczy.

- Po trzecie, ze względu na subtelną różnicę między wynikiem zwracanym przez funkcje `success` i `then`, zamiast funkcji zwrotnej `success` obietnicy zwracanej przez wywołanie `$http.get` używamy funkcji `then`. Funkcja `success` zwraca początkową obietnicę (w naszym przypadku obietnica ta jest tworzona przez wywołanie `$http.get`), natomiast funkcja `then` zwraca nową obietnicę, wyznaczaną przez wartość wynikową zwróconą przez funkcję `success` lub `error`, do których została dołączona funkcja `then`.

> Ponieważ zamiast funkcji `success` używamy funkcji `then`, do funkcji zwrotnej jest przekazywany jeden obiekt zawierający wszystkie cztery właściwości: `config`, `data`, `header` oraz `status`.

Zanim zajmiemy się kolejnymi zagadnieniami, omówię dokładniej **łańcuchy obietnic**, analizując kilka prostych przykładów. To bardzo użyteczne rozwiązanie, zapewniające ogromne możliwości.

Wyjaśnienie łańcuchów obietnic

Tworzenie łańcuchów obietnic polega na przekazaniu wyniku wyznaczenia jednej obietnicy do innej obietnicy. Ponieważ obietnica stanowi opakowanie dla operacji asynchronicznych, tworzenie łańcuchów obietnic pozwala nam zarządzać kodem asynchronicznym poprzez zdefiniowanie sekwencji operacji, a nie w formie zagnieżdżonych wywołań zwrotnych. We wcześniejszej części rozdziału spotkałeś się już z przykładem takiego łańcucha obietnic — posłużył nam on do pobrania ćwiczeń z serwera (pierwsza operacja asynchroniczna), przekształcenia ich (druga operacja asynchroniczna) oraz dowiązania do widoku (trzecia operacja asynchroniczna). Proces ten został także przedstawiony na diagramie z rysunku 5.6.

Technika ta umożliwia tworzenie łańcuchów o dowolnej długości, o ile tylko wszystkie kolejne metody w łańcuchu będą zwracać obietnice. W naszym przypadku obie metody, `$http.get` i `then`, zwracają obietnice.

Przeanalizujmy teraz znacznie prostszy przykład łańcuchów obietnic. Kod tego przykładu został przedstawiony poniżej:

```
var promise = $q.when(1);
var result = promise
```


Kup ksi k Pole ksi k

```
.then(function (i) { return i + 1;})
   .then(function (i) { return i + 1;})
   .then(function (i) { return i + 1;});
   .then(function (i) { console.log("Wartość i wynosi:" + i);});
```

> Przykład demonstrujący tworzenie łańcucha obietnic jest dostępny w serwisie jsFiddle, na stronie *http://jsfiddle.net/cmyworld/9ak1gahe/*.

Powyższy kod tworzy łańcuch obietnic, przy czym każda z funkcji w łańcuchu inkrementuje przekazaną do niej wartość i przekazuje ją dalej, do kolejnej obietnicy. Ostateczna wartość `i`, przekazana do ostatniego wywołania funkcji `then`, będzie wynosiła `4`.

> Funkcja `$q.when(1)` zwraca obietnicę, której wyznaczenie zwróci wartość `1`.

Zgodnie z podanymi wcześniej informacjami tworzenie łańcuchów obietnic jest możliwe dlatego, że funkcja `then` zwraca obietnicę. W efekcie wyznaczania obietnicy zwracana jest wartość funkcji zwrotnej `success` lub `error`. Jeśli przyjrzymy się poprzedniemu przykładowi, to zauważymy, że w każdej funkcji zwrotnej (z wyjątkiem ostatniej), wykonywanej w przypadku prawidłowego wyznaczenia obietnicy, została użyta instrukcja `return`.

Nieco zaskakujące może być natomiast działanie łańcucha obietnic w przypadku, gdy ich wyznaczenie zakończy się niepowodzeniem. Najlepiej będzie zilustrować go na przykładzie:

```
var errorPromise = $q.reject("niepowodzenie");
var resultError = errorPromise.then(function (data) {
    return "powodzenie";
}, function (e) {
    return "niepowodzenie";
});
resultError.then(function (data) {
    console.log("W funkcji zwrotnej success, dane:" + data);
}, function (e) {
    console.log("W funkcji zwrotnej error, dane:" + e);
});
```

Wywołanie `$q.reject` tworzy obietnicę, która zostanie odrzucona i zwróci przy tym wartość `error`. A zatem także wyznaczenie obietnicy `resultError` zwróci wartość `error`.

Teraz jednak powstaje pytanie: jaki komunikat wyświetli funkcja zwrotna `resultError.then`? Okazuje się, że komunikatem tym będzie: `W funkcji zwrotnej success, dane: niepowodzenie`, gdyż zostanie wywołana funkcja zwrotna `success`, a nie `error`. A to z kolei wynika z faktu, że w obu funkcjach zwrotnych — `success` i `error` — przekazanych w wywołaniu `errorPromise.then` (lub `resultError`) użyliśmy zwyczajnej instrukcji `return`.

Kup ksi k Pole ksi k

Gdybyśmy chcieli, by wyznaczanie wszystkich obietnic w łańcuchu zakończyło się niepowodzeniem, to w każdej funkcji zwrotnej `error` musielibyśmy odrzucić obietnicę. Zmieńmy zatem obietnicę `resultError` w następujący sposób:

```
var resultError = errorPromise.then(function (data) {
    return "powodzenie";
}, function (e) {
    return $q.reject(e);
});
```

W tym przypadku w następnym wywołaniu `then` w łańcuchu zostanie wywołana prawidłowa funkcja zwrotna, a w oknie konsoli pojawi się komunikat: `W funkcji zwrotnej error, dane: niepowodzenie`.

Zwrócenie w funkcji zwrotnej `error` wyniku wywołania `$q.reject(e)` sprawi, że wyznaczoną wartością obietnicy `resultError` będzie odrzucona obietnica (wywołanie `$q.reject` zwraca obietnicę, która zawsze będzie odrzucona).

Tworzenie łańcuchów obietnic jest rozwiązaniem zapewniającym ogromne możliwości, a jego dobre opanowanie pozwoli Ci tworzyć bardziej zwięzły i lepiej zorganizowany kod. W tym rozdziale będziemy go używali bardzo intensywnie, do obsługi odpowiedzi przesyłanych z serwera oraz przetwarzania i wczytywania danych.

A teraz wróćmy do prac nad rozwojem aplikacji, do miejsca, w którym je wcześniej przerwaliśmy, czyli do wczytywania z serwera danych ćwiczeń i treningów.

Wczytywanie danych ćwiczeń i treningów z serwera

Skoro poprawiliśmy już implementację funkcji `getExercise` usługi `WorkoutService`, możemy zaimplementować także inne operacje związane z pobieraniem danych ćwiczeń i treningów. W tym celu należy skopiować implementacje funkcji `getExercise`, `getWorkouts` oraz `getWorkout` usługi `WorkoutService` z pliku *services.js* umieszczonego w katalogu *rozdzial05\punkt_kontrolny_2\app\js\shared*.

> Funkcje `getWorkout` i `getExercise` pobierają odpowiednie dane na podstawie nazwy treningu lub ćwiczenia. Każdy element baz danych MongoLab ma właściwość `_id`, która w unikalny sposób go identyfikuje. W przypadku obiektów `Exercise` i `WorkoutPlan` tym unikalnym identyfikatorem jest nazwa ćwiczenia bądź treningu, dlatego nazwa i właściwość `_id` zawsze będą identyczne.

Należy zwrócić szczególną uwagę na implementacje funkcji `getWorkouts` i `getWorkout`, gdyż w obu jest wykonywane przekształcenie danych związane z różnicami między modelem aplikacji i formatem danych przechowywanych na serwerze.

Funkcja `getWorkouts` przypomina funkcję `getExercises`, z tym że tworzy obiekt `WorkoutPlan`, a tablica `exercises` nie jest odwzorowywana na tablicę obiektów klasy `Exercises` — funkcja pozostawia strukturę danych zwróconych z serwera, o postaci `{name:'`*`nazwa`*`', duration:`*`wartosc`*`}`.

Kup ksi k Pole ksi k

Implementacja funkcji getWorkout wymaga wykonania całkiem rozbudowanych operacji odwzorowywania danych. Poniżej przedstawiam jej kod w aktualnej, docelowej postaci:

```
service.getWorkout = function (name) {
    return $q.all([service.getExercises(), $http.get(collectionsUrl
        + "/workouts/" + name, { params: { apiKey: apiKey } })])
        .then(function (response) {
            var allExercises = response[0];
            var workout = new WorkoutPlan(response[1].data);

            angular.forEach(response[1].data.exercises, function (exercise) {
                exercise.details = allExercises.filter(function (e) {
                      return e.name === exercise.name; })[0];
            });
            return workout;
        });
};
```

W kodzie tej funkcji dzieje się całkiem sporo, a Ty musisz dobrze zrozumieć sposób jej działania.

Operacje wykonywane przez funkcję getWorkout rozpoczynają się od wywołania funkcji $q.all. Służy ona do oczekiwania na zakończenie wywołań większej liczby obietnic. Jej argumentem jest tablica obietnic, a sama funkcja także zwraca obietnicę. Ta zagregowana obietnica jest wyznaczana lub odrzucana (sygnalizuje błąd), gdy wszystkie obietnice w tablicy zostaną wyznaczone lub odrzucone. W powyższej funkcji, w wywołaniu $q.all, przekazaliśmy tablicę zawierającą dwie obietnice: pierwszą z nich jest obietnica zwrócona przez funkcję service.get↪Exercises, a drugą wynik zwrócony przez wywołanie $http.get (które pobiera trening o określonym identyfikatorze).

Parametr response przekazywany do funkcji zwrotnej obietnicy $q.all także jest tablicą. Zawiera ona wartości zwrócone przez funkcje zwrotne każdej z obietnic zapisanych w tablicy przekazanej w wywołaniu $q.all. W naszym przypadku response[0] zawiera listę ćwiczeń, a response[1] — odebrane z serwera dane na temat treningu (response[1].data zawiera sekcję danych odpowiedzi HTTP).

Kiedy już uzyskamy informacje dotyczące treningu i pełną listę ćwiczeń, kolejna operacja aktualizuje tablicę exercises treningu, zapisując w niej tablicę prawidłowych obiektów klasy Exercise. W tym celu w tablicy allExercises wyszukiwane są nazwy ćwiczeń zapisane w elementach tablicy workout.exercises zwróconej z serwera. W efekcie zyskujemy kompletny obiekt WorkoutPlan zawierający prawidłowo przygotowaną tablicę ćwiczeń — exercises.

Te zmiany funkcji usługi WorkoutService zmuszają nas także do odpowiedniego zmodyfikowania kodu, który ich używa. Już wcześniej poprawiliśmy implementacje kontrolerów Exercises↪NavController i ExerciseListController. Teraz w podobny sposób musimy zmodyfikować kontroler WorkoutListController. Funkcje getWorkout i getExercise nie są używane bezpośrednio w kodzie kontrolera, ale korzystają z nich usługi używane do tworzenia treningów.

Kup ksi k Pole ksi k

Dlatego teraz zajmiemy się poprawieniem implementacji tej usługi i stron prezentujących szczegółowe informacje o treningach i ćwiczeniach.

Poprawianie stron ze szczegółowymi informacjami o treningach i ćwiczeniach

W tym podpunkcie poprawimy stronę ze szczegółowymi informacjami o treningu, natomiast wprowadzenie niezbędnych zmian w kodzie strony z informacjami o ćwiczeniu pozostanie jako zadanie do samodzielnego wykonania, gdyż jest niemal identyczne.

Kontroler `ExercisesNavController`, używany do wyświetlania systemu nawigacyjnego strony treningu, został już poprawiony, możemy się zatem zająć poprawianiem kontrolera `Workout` `↳DetailController`.

Kontroler `WorkoutDetailController` nie wczytuje danych treningu bezpośrednio, lecz używa do tego celu funkcji określonej w parametrze `resolve` definicji trasy (patrz: kod konfiguracyjny tras umieszczony w pliku *config.js*); gdy zmienia się trasa, funkcja ta wstrzykuje do kontrolera aktualnie wybrany trening (zapisując go we właściwości `selectedWorkout`). Z kolei sama funkcja określająca właściwość `selectedWorkout` jest zależna od usługi `WorkoutBuilderService`, której używa do wczytywania nowego bądź już istniejącego treningu. Dlatego w pierwszej kolejności musimy zmodyfikować usługę `WorkoutBuilderService`.

Za pobranie szczegółowych informacji o treningu odpowiada funkcja `startBuilding`. Poniższy przykład pokazuje, jak powinien wyglądać jej zmodyfikowany kod:

```
service.startBuilding = function (name) {
    var defer = $q.defer();
    if (name) {
        WorkoutService.getWorkout(name).then(function (workout) {
            buildingWorkout = workout;
            newWorkout = false;
            defer.resolve(buildingWorkout);
        });
    } else {
        buildingWorkout = new WorkoutPlan({});
        defer.resolve(buildingWorkout);
        newWorkout = true;
    }
    return defer.promise;
};
```

W powyższej implementacji zastosowaliśmy usługę `$q`, wchodzącą w skład API obietnic, aby za jej pomocą utworzyć i wyznaczyć obietnicę. Zastosowane rozwiązanie wymaga od nas utworzenia własnej obietnicy, gdyż tworzenie i zwracanie nowego treningu jest procesem synchronicznym, natomiast wczytywanie istniejącego — nie jest. A zatem dla zachowania spójności działania funkcji w każdym z przypadków będzie ona zwracać obietnicę — czy to w razie tworzenia nowego treningu, czy też edycji treningu, który już istnieje.

Kup ksi k Pole ksi k

Aby przetestować implementację, wystarczy wyświetlić stronę ze szczegółowymi informacjami o dowolnym istniejącym treningu, na przykład `7minWorkout`, podając jego nazwę na końcu adresu `#/builder/workouts/`. Dane treningu powinny zostać wyświetlone na stronie z niewielkim opóźnieniem.

W powyższej funkcji po raz pierwszy utworzyliśmy własną obietnicę, dlatego to doskonały moment, by nieco dokładniej poznać to zagadnienie.

Tworzenie i wyznaczanie własnych obietnic

Tworzenie i wyznaczanie standardowych obietnic wymaga wykonania kilku kroków:

1. Utworzenia nowego obiektu obietnicy, `defer`, poprzez wywołanie funkcji `$q.defer()`. Pod względem pojęciowym obiekt `defer` można by porównać do czynności, która kiedyś, w przyszłości, zostanie zakończona.
2. Zwrócenia obiektu obietnicy na końcu kodu funkcji poprzez wywołanie funkcji `defer.promise`.
3. W dowolnym momencie w przyszłości należy wywołać funkcję `defer.resolve(data)`, aby wyznaczyć obietnicę z użyciem przekazanych danych (`data`), bądź też funkcję `defer.reject(error)`, by ją odrzucić z konkretną informacją o błędzie (`error`). Wywołanie funkcji `resolve` oznacza, że zadanie zostało zakończone pomyślnie, natomiast wywołanie funkcji `reject` — że podczas jego realizacji wystąpił błąd.

Implementacja funkcji `startBuilding` przedstawiona w poprzednim podrozdziale działa dokładnie według tego schematu.

Interesującym aspektem przedstawionej implementacji funkcji `startBuilding` jest to, że w przypadku wykonywania kodu z klauzuli `else` obietnica jest wyznaczana natychmiast (wywołujemy metodę `defer.resolve`, przekazując do niej nową instancję obiektu treningu), i to jeszcze zanim zostanie ona zwrócona do kodu wywołującego. W efekcie w przypadku tworzenia nowego treningu obietnica jest wyznaczana od razu, w momencie zakończenia wykonywania funkcji `startBuilding`.

W możliwości tworzenia i wyznaczania własnych obietnic tkwi ogromny potencjał. Jest ona bardzo użyteczna w sytuacjach wymagających wywołania i koordynacji większej liczby wywołań asynchronicznych, nim będzie można zwrócić ostateczny wynik. Przeanalizujmy hipotetyczny przykład funkcji pewnej usługi, której zadanie polega na pobraniu i zwróceniu opinii o produkcie z kilku platform handlu elektronicznego:

```
getProductPriceQuotes(productCode) {
    var defer = $q.defer()
    var promiseA = getQuotesAmazon(productCode);
    var promiseB = getQuotesBestBuy(productCode);
    var promiseE = getQuotesEbay(productCode);
    $q.all([promiseA, promiseB, promiseE])
      .then(function (response) {
```


Kup ksi k Pole ksi k

```
            defer.resolve([buildModel(response[0]),
                buildModel(response[1]), buildModel(response[2])]);
        });
      defer.promise;
}
```

Funkcja `getProductPriceQuotes` musi wykonywać asynchroniczne odwołania do kilku usług zajmujących się handlem elektronicznym, scalić uzyskane dane i zwrócić uzyskane wyniki. Takimi skoordynowanymi operacjami asynchronicznymi można zarządzać przy użyciu API obietnic (obiektów `defer`). W powyższym przykładzie zastosowaliśmy wywołanie `$q.all`, które umożliwia odłożenie wykonania operacji do momentu wyznaczenia większej liczby obietnic. Po zakończeniu wszystkich zdalnych wywołań zostaje wywołana funkcja zwrotna, przekazana w wywołaniu funkcji `then`. Hipotetyczna funkcja `buildModel` zostaje użyta do utworzenia wspólnego modelu `Quote`, ponieważ może się zdarzyć, że wyniki zwracane przez poszczególne zdalne usługi będą się od siebie różnić. I w końcu ostatnie wywołanie, `defer.promise`, scala dane nowego modelu i zwraca je w postaci tablicy. To doskonale skoordynowane poczynania!

Istnieje kilka reguł, czy też wskazówek, które warto stosować podczas korzystania z API obietnic (obiektów `defer`). Przedstawiam je na poniższej liście:

- Raz wyznaczonej obietnicy nie można już zmienić. Obietnicę można porównać z instrukcją `return`, która zostanie wykonana kiedyś w przyszłości. Kiedy jednak obietnica zostanie wyznaczona, jej wartości nie można już zmienić.
- Funkcję `then` istniejącego obiektu obietnicy można wywoływać dowolnie wiele razy, niezależnie od tego, czy obietnica została wyznaczona, czy nie.
- Wywołanie funkcji `then` obiektu obietnicy, która już została wyznaczona lub odrzucona, sprawi, że odpowiednia funkcja zwrotna zostanie wywołana natychmiast.

Zachowanie odpowiadające utworzeniu własnego obiektu obietnicy i wyznaczeniu jej można jednak uzyskać także w inny sposób — korzystając z funkcji `$q.when` API obietnic.

Funkcja when usługi $q

Spróbujemy teraz pójść na całość i pozbyć się kilku wierszy kodu z implementacji funkcji `startBuilding`. W tym celu skorzystamy z funkcji `$q.when`. Stosowanie własnych obiektów obietnic tylko po to, by zapewnić spójność wyników zwracanych przez funkcję `startBuilding`, można uznać za pewną przesadę. Funkcję `$q.when` opracowano z myślą o takich właśnie sytuacjach.

Funkcja `when` pobiera przekazany argument i zwraca obietnicę:

```
when(value);
```

Argument `value` może być zwyczajnym obiektem języka JavaScript bądź obiektem obietnicy. Jeśli wartością przekazaną w wywołaniu funkcji `when` jest zwyczajny obiekt, to zwrócona przez nią obietnica zostanie wyznaczona z użyciem tej wartości. Jeśli jednak przekazaną wartością była obietnica, to jej wyznaczona wartość zostanie użyta podczas wyznaczania obietnicy zwróconej przez funkcję `$q.when`. Zobaczmy teraz, jak możemy zastosować tę funkcję w naszej funkcji `startBuilding`.

Kup ksi k Pole ksi k

Istniejącą implementację funkcji `startBuilding` należy zastąpić następującym kodem:

```
service.startBuilding = function (name) {
    if (name) {
        return WorkoutService.getWorkout(name).then(function (workout) {
            buildingWorkout = workout;
            newWorkout = false;
            return buildingWorkout;
        });
    }
    else {
        buildingWorkout = new WorkoutPlan({});
        newWorkout = true;
        return $q.when(buildingWorkout);
    }
};
```

W powyższym fragmencie zmodyfikowane wiersze kodu zostały wyróżnione pogrubieniem. Wywołanie funkcji `$q.when` zostało umieszczone w kodzie klauzuli `else` i służy do zwrócenia nowego obiektu `WorkoutPlan` przy użyciu obietnicy.

W ten sposób udało się nam pozbyć kilku wierszy z kodu funkcji `startBuilding`, a jednocześnie zachować jej wcześniejszy sposób działania. Co więcej, teraz rozumiesz także, jak działa funkcja `$q.when` i kiedy można jej używać. Możemy zatem zabrać się za dokończenie strony prezentującej szczegółowe informacje o treningu.

Ciąg dalszy poprawiania stron ze szczegółowymi informacjami o treningach i ćwiczeniach

Poprawienie funkcji `startBuilding` wystarcza, by zapewnić, że strona z informacjami o treningu będzie wczytywała dane. Łatwo możemy się o tym przekonać i upewnić, że zarówno nowe treningi, jak i już istniejące będą wczytywane prawidłowo.

W kontrolerze `WorkoutDetailController` nie musimy implementować żadnych funkcji zwrotnych. A dlaczego? Otóż dlatego, że zadba o to właściwość konfiguracyjna `resolve` podana w definicji trasy. Przedstawiłem ją już w poprzednim rozdziale, gdzie została zastosowana do wstrzyknięcia do kontrolera `WorkoutDetailController` obiektu `selectedWorkout`. Zobaczmy teraz, w jaki sposób modyfikacje związane z wywołaniami asynchronicznymi i obietnicami wpływają na funkcję `resolve`.

Wyznaczanie tras a obietnice

Kiedy spojrzymy na konfigurację trasy dla strony *Konstruktora treningów* podaną w wywołaniu funkcji `$routeProvider.when`, przekonamy się, że funkcja `selectedWorkout` zdefiniowana we właściwości `resolve` składa się obecnie tylko z jednego wiersza kodu:

```
return WorkoutBuilderService.startBuilding($route.current.params.id);
```


Kup książkę Poleć książkę

Zgodnie z tym, czego dowiedziałeś się z poprzedniego rozdziału, właściwość konfiguracyjna `resolve` służy do wstrzykiwania zależności do kontrolera przed utworzeniem jego obiektu. W powyższym przypadku wartością zwracaną przez funkcję `selectedWorkout` jest obiekt obietnicy, a nie gotowy obiekt `WorkoutPlan`.

Kiedy wartością zwróconą przez funkcję podaną we właściwości `resolve` jest obietnica, to mechanizm wyznaczania tras AngularJS poczeka na jej wyznaczenie i dopiero, gdy to nastąpi, wczyta odpowiednią trasę. Po wyznaczeniu obietnicy uzyskane dane zostają wstrzyknięte do kontrolera, dokładnie tak samo, jak to ma miejsce w przypadku zwrócenia normalnej wartości. Także w naszej implementacji, po wyznaczeniu obietnicy, dane wybranego treningu zostają automatycznie wstrzyknięte do kontrolera `WorkoutDetailController`. Możemy się o tym przekonać, dwukrotnie klikając kafelek z tytułem treningu wyświetlony na liście treningów — strona *Konstruktora treningów* zostanie wyświetlona z zauważalnym opóźnieniem.

Oczywistą zaletą korzystania z właściwości konfiguracyjnej `resolve` w definicjach tras jest to, że pozwala nam ona uniknąć umieszczania w kodzie kontrolera asynchronicznych (`then`) funkcji zwrotnych, takich jak te, których użyliśmy do wczytania listy treningów w kontrolerze `WokroutListController`.

Modyfikacje musimy wprowadzić także na stronie prezentującej szczegółowe informacje o ćwiczeniu. W tym przypadku jednak w definicji tras nie używamy właściwości `resolve`, co oznacza, że do wczytania danych ćwiczenia będziemy musieli skorzystać z obietnic i funkcji zwrotnych, które umieścimy w funkcji `init` kontrolera. Zmodyfikowany kod usługi `Exercise` ↪`BuilderService` i kontrolera `ExerciseDetailController` można znaleźć w katalogu *rozdzial05\punkt_kontrolny_2* w przykładach dołączonych do książki. Aby zapewnić prawidłowe działanie wczytywania danych ćwiczeń, wystarczy go skopiować do tworzonej wersji aplikacji. Możemy także spróbować zaimplementować modyfikacje samodzielnie, a następnie je porównać.

> Kod aplikacji na obecnym etapie jej rozwoju można znaleźć w przykładach dołączonych do książki, w katalogu *rozdzial05\punkt_kontrolny_2*.

Teraz zajmiemy się poprawieniem możliwości tworzenia i aktualizacji ćwiczeń i treningów.

Wykonywanie podstawowych operacji na ćwiczeniach i treningach

Jeśli chodzi o podstawowe operacje na danych, czyli ich **tworzenie**, **odczyt**, **aktualizację** oraz **usuwanie**[2], to wszystkie związane z nimi funkcje musimy zmodyfikować w taki sposób, by korzystały z obietnic.

---

[2] Te cztery podstawowe operacje są także często nazywane „operacjami CRUD” od angielskich słów *create*, *read*, *update* i *delete*, oznaczających odpowiednio: tworzenie, odczyt, aktualizację i usuwanie — *przyp. tłum.*

Kup ksi k Pole ksi k

We wcześniejszej części rozdziału przedstawiony został wzorzec adresów URL serwisu MongoLab, umożliwiający wykonywanie wszystkich tych operacji. Warto teraz ponownie zajrzeć do tego podpunktu rozdziału i przypomnieć sobie postać tych adresów i sposób wykonywania poszczególnych operacji, będziemy ich bowiem potrzebować podczas tworzenia i aktualizowania treningów.

Zanim przystąpimy do implementacji, koniecznie musisz zrozumieć, w jaki sposób MongoLab identyfikuje elementy kolekcji i z jakiej strategii generowania identyfikatorów będziemy korzystać. Każdy element kolekcji przechowywanych w serwisie MongoLab jest w unikalny sposób identyfikowany (w ramach tej kolekcji) przez właściwość `_id`. Podczas tworzenia nowego elementu kolekcji bądź to samodzielnie podajemy identyfikator, bądź też zostaje on wygenerowany przez serwer. Po określeniu wartości właściwości `_id` nie można jej już zmieniać. W przypadku naszego modelu rolę unikalnych identyfikatorów ćwiczeń i treningów będą pełnić ich właściwości `name` — wartości tej właściwości będziemy także zapisywać we właściwości `_id` (dzięki czemu nie będą one automatycznie generowane przez serwer). Trzeba także pamiętać, że używane w aplikacji klasy modelu nie zawierają właściwości `_id`, co oznacza, że trzeba ją utworzyć przed zapisaniem w bazie nowego rekordu.

W pierwszej kolejności zajmiemy się poprawieniem możliwości tworzenia nowych treningów.

Poprawianie i tworzenie nowych treningów

Skorzystamy teraz z metody wstępującej tworzenia kodu i zaczniemy od poprawienia kodu usługi `WorkoutService`. Poniżej przedstawiam nową postać kodu metody `addWorkout`:

```
service.addWorkout = function (workout) {
  if (workout.name) {
    var workoutToSave = angular.copy(workout);
    workoutToSave.exercises =
      workoutToSave.exercises.map(function (exercise) {
          return {
                  name: exercise.details.name,
                  duration: exercise.duration
          }
      });
    workoutToSave._id = workoutToSave.name;
    return $http.post(collectionsUrl + "/workouts", workoutToSave,
             { params: { apiKey: apiKey }})
        .then(function (response) { return workout });
}}
```

W funkcji `getWorkout` musimy odwzorować dane zapisane w modelu używanym na serwerze na model używany w aplikacji; w powyższej funkcji musimy zrobić coś odwrotnego. Ponieważ nie chcemy modyfikować modelu dowiązanego do widoku, zatem pierwszą operacją wykonywaną przez funkcję jest skopiowanie danych treningu.

Następnie odwzorowujemy tablicę ćwiczeń (`workoutToSave.exercises`) do bardziej zwartego formatu, nadającego się zapisania w bazie danych. W tablicy `exercises`, która zostanie zapisana na serwerze, chcemy umieścić wyłącznie nazwy ćwiczeń i czasy ich wykonywania.

Kup książkę Poleć książkę

Następnie we właściwości `_id` zapisujemy nazwę treningu, co pozwoli nam w unikalny sposób zidentyfikować go w kolekcji `Workouts` w bazie danych.

> **Słowo ostrzeżenia**
>
> Bardzo uproszczone rozwiązanie, jakim jest używanie *nazwy* treningu bądź ćwiczenia jako identyfikatora rekordu (`_id`) w bazie MongoDB, nie sprawdzi się w żadnej większej aplikacji. Trzeba pamiętać, że tworzymy tu aplikację internetową, która może być jednocześnie używana przez wielu użytkowników. Ponieważ zawsze istnieje prawdopodobieństwo podania identycznych nazw treningu lub ćwiczenia przez dwóch różnych użytkowników, potrzebujemy jakiegoś mocniejszego mechanizmu, który pozwoliłby zagwarantować unikalność tych nazw.
>
> API typu REST używane przez serwis MongoLab przysparza jeszcze jednego problemu. Otóż jeśli zostanie przesłane powtórzone żądanie `POST` zawierające pole `id` o tej samej wartości, to pierwsze żądanie utworzy nowy dokument, a drugie go zaktualizuje zamiast zakończyć się niepowodzeniem. Oznacza to, że żadne testy sprawdzające powtórzenia wartości pola `id` wykonywane po stronie klienta nie są nas w stanie zabezpieczyć przed utratą danych. W takich sytuacjach preferowanym rozwiązaniem jest skorzystanie z automatycznego generowania wartości identyfikatorów.

Na samym końcu funkcji `addWorkout` wywołujemy funkcję `$http.post`, do której przekazujemy adres URL, na jaki należy wysłać żądanie, dane, które należy wysłać, oraz dodatkowy parametr łańcucha zapytania (`apiKey`). Ta ostatnia instrukcja `return` może wyglądać znajomo, gdyż ponownie korzystamy tu z *łańcucha obietnic*, aby proces wyznaczania obietnicy zwrócił obiekt treningu.

> W standardowych rozwiązaniach związanych z tworzeniem nowych danych generowanie unikalnych identyfikatorów jest wykonywane na serwerze (w przeważającej większości przypadków odpowiada za nie baza danych). A zatem odpowiedź na takie żądanie będzie zawierała wygenerowany identyfikator. W takich przypadkach przed zwróceniem obiektu do kodu wywołującego trzeba go będzie zmodyfikować.

No to może spróbujemy zaimplementować operację aktualizacji? Funkcję `updateWorkout` można zaktualizować w podobny sposób, a jedyną różnicą będzie konieczność zastosowania funkcji `$http.put`:

```
return $http.put(collectionsUrl + "/workouts/" + workout.name,
  workoutToSave, { params: { apiKey: apiKey } });
```

Adres URL zastosowany w powyższym kodzie zawiera dodatkowy fragment (`workout.name`); to identyfikator elementu kolekcji, który należy zmodyfikować.

> W serwisie MongoLab żądanie `PUT` tworzy dokument przekazany w treści żądania, o ile nie uda się go odnaleźć w kolekcji. A więc wykonując żądania `PUT`, należy się upewnić, że początkowy element kolekcji istnieje. W tym celu wystarczy wcześniej wykonać żądanie `GET` dotyczące tego samego dokumentu i upewnić się, że został on pobrany.

Kup ksi k Pole ksi k

Ostatnią operacją, którą musimy poprawić, jest usuwanie treningów. Poniżej przedstawiam bardzo prostą implementację tej operacji, sprowadzającą się do wywołania funkcji `$http.delete` i przekazania do niej odpowiedniego adresu URL:

```
service.deleteWorkout = function (workoutName) {
    return $http.delete(collectionsUrl + "/workouts/" +
        workoutName, { params: { apiKey: apiKey } });
};
```

Po wprowadzeniu tych zmian możemy już się zająć poprawieniem usługi `WorkoutBuilderService` i kontrolera `WorkoutDetailController`. Poniżej przedstawiam nową wersję kodu funkcji `save` usługi `WorkoutBuilderService`:

```
service.save = function () {
    var promise = newWorkout ?
      WorkoutService.addWorkout(buildingWorkout) :
      WorkoutService.updateWorkout(buildingWorkout);
    promise.then(function (workout) {
        newWorkout = false;
    });
    return promise;
};
```

Większość jej kodu wygląda tak samo jak wcześniej, a jedynymi różnicami są przeniesienie instrukcji określającej wartość właściwości `newWorkout` do funkcji zwrotnej przekazywanej w wywołaniu funkcji `then` i zwrócenie obietnicy jako wyniku wykonania funkcji.

Ostatnia modyfikacja dotyczy kodu kontrolera `WorkoutDetailController`. W jego metodach `save` i `delete` musimy zastosować znane już rozwiązanie wykorzystujące funkcję zwrotną:

```
$scope.save = function () {
    $scope.submitted = true; // wymuszenie sprawdzenia poprawności
    if ($scope.formWorkout.$invalid) return;
    WorkoutBuilderService.save().then(function (workout) {
        $scope.workout = workout;
        $scope.formWorkout.$setPristine();
        $scope.submitted = false;
    });
}
service.delete = function () {
    if (newWorkout) return; // nowego treningu nie można usuwać
    return WorkoutService.deleteWorkout(buildingWorkout.name);
}
```

I to już wszystko. Teraz możemy już tworzyć nowe treningi, aktualizować treningi istniejące oraz je usuwać. I wcale nie było to aż tak trudne!

Wypróbujmy zatem efekty naszych zmian. W tym celu wystarczy wyświetlić stronę *Konstruktora treningów*, utworzyć nowy trening i go zapisać. Warto także spróbować wprowadzić

Kup ksi k Pole ksi k

zmiany w którymś z już istniejących treningów. W obu przypadkach wszystko powinno działać bez żadnych problemów.

> W razie jakichkolwiek problemów z aplikacją jej działająca wersja obejmująca zmiany wprowadzone na tym etapie prac jest dostępna w przykładach dołączonych do książki, w katalogu *rozdzial05\punkt_kontrolny_3*.

Coś ciekawego dzieje się z żądaniami sieciowymi, kiedy wykonujemy żądania `POST` i `PUT` w celu zapisania danych. Aby się o tym przekonać, należy otworzyć listę operacji sieciowych wykonywanych przez przeglądarkę (*F12*) i przyjrzeć się wykonanym żądaniom. Ich rejestr powinien wyglądać jak na rysunku 5.5.

Path		Text	
workouts?apiKey=Ocd4hjf8RvieIM3IRDDS... api.mongolab.com/api/1/databases/angul...	OPTIONS	200 OK	xhr
workouts?apiKey=Ocd4hjf8RvieIM3IRDDS... api.mongolab.com/api/1/databases/angul...	POST	200 OK	xhr

Rysunek 5.7. Rejestr operacji sieciowych wykonywanych podczas zapisu nowego treningu

Okazuje się, że przed wykonaniem właściwego żądania `POST` na ten sam adres zostało wysłane żądanie `OPTIONS`. To, z czym mamy tu do czynienia, jest nazywane **żądaniem sprawdzającym** (ang. *preflight request*). A jest ono wykonywane dlatego, że odwołujemy się do innej domeny — *api.mongolab.com*.

Zagadnienia związane z wykonywaniem żądań HTTP kierowanych do innych domen są bardzo ważne i koniecznie należy je dobrze zrozumieć i dowiedzieć się, jakie konstrukcje pozwalające na wykonywanie takich żądań udostępnia AngularJS.

Odwołania do innych domen a AngularJS

Odwołania do innych domen (ang. *cross-domain requests*) są żądaniami kierowanymi do zasobów udostępnianych na innych domenach. W przypadku, gdy takie żądania są generowane przez aplikacje JavaScriptu, podlegają pewnym ograniczeniom narzucanym przez przeglądarki, wynikającym ze stosowania zasad tego samego pochodzenia (ang. *same-origin policy*). Ograniczenia te uniemożliwiają wykonywanie żądań AJAX do domen innych niż ta, z której skrypt został pobrany. Kontrola źródła jest przeprowadzana w sposób ścisły, na podstawie protokołu, nazwy hosta oraz numeru portu.

W przypadku naszej aplikacji wszystkie odwołania kierowane na adresy rozpoczynające się od `https://api.mongolab.com` są właśnie odwołaniami do innej domeny, gdyż kod źródłowy aplikacji jest pobierany z innej domeny (najprawdopodobniej z `https://localhost/...`).

Istnieją pewne sposoby omijania tego ograniczenia, jak również pewne standardy, które pozwalają je nieco złagodzić. W dalszej części rozdziału przedstawiam dwie najczęściej stosowane spośród tych technik, a mianowicie:

Kup ksi k

Pole ksi k

- **JSONP** — *JSON with Padding*;
- **CORS** — współdzielenie zasobów o innym pochodzeniu (ang. *crosss-origin resource sharing*).

Jedną z najczęściej stosowanych technik omijania ograniczeń narzucanych przez zasadę tego samego pochodzenia jest JSONP.

Stosowanie JSONP do wykonywania żądań do innych domen

Mechanizm JSONP służący do wykonywania zdalnych wywołań bazuje na fakcie, że przeglądarki mogą wykonywać skrypty pochodzące z dowolnych źródeł, jeśli zostaną one podane w znaczniku `<script>`. W rzeczywistości kilka plików frameworku AngularJS używanych w naszej aplikacji jest pobieranych z serwerów CDN (*ajax.googleapis.com*), a nasza aplikacja odwołuje się do nich przy użyciu znaczników `script`.

W przypadku techniki JSONP, zamiast generować zwyczajne żądanie na serwer, dynamicznie generowany jest znacznik `script`, którego atrybut `src` zawiera adres dokładnie tego samego punktu końcowego na serwerze, do którego chcemy się odwołać. Dodanie tego elementu `script` do drzewa DOM dokumentu spowoduje wykonanie żądania skierowanego do docelowego serwera.

Serwer musi odpowiedzieć na to żądanie, przesyłając odpowiedź zapisaną w odpowiednim formacie, której zawartość jest umieszczona wewnątrz wywołania funkcji (nazwa tej techniki, JSONP, pochodzi właśnie od tej dodatkowej warstwy „opakowującej” właściwą zawartość odpowiedzi).

Funkcja AngularJS `$http.jsonp` ukrywa wszelkie złożoności stosowania tej techniki i udostępnia bardzo proste API służące do generowania żądań JSONP. Sposób generowania żądań JSONP można przeanalizować na przykładzie opublikowanym w serwisie jsFiddle, na stronie *http://jsfiddle.net/cmyworld/v9y4uby2/*. Przykład ten używa *Yahoo Stock API*, by pobierać notowania dowolnych spółek.

Zastosowana w tym przykładzie funkcja `getQuote` wygląda następująco:

```
$scope.getQuote = function () {
  var url =
    "https://query.yahooapis.com/v1/public/yql?q=
    select%20*%20from%20yahoo.finance.
    quote%20where%20symbol%20in%20(%22" + $scope.symbol +
    "%22)&format=json&env=store%3A%2F%2Fdatatables.
    org%2Falltableswithkeys&callback=JSON_CALLBACK";
  $http.jsonp(url).success(function (data) {
    $scope.quote = data;
  });
};
```


Kup ksi k
Pole ksi k

Aby wykonać żądanie JSONP w aplikacji AngularJS, funkcja `jsonp` wymaga dodania do początkowego adresu URL dodatkowego parametru łańcucha zapytania: `callback=JSON_CALLBACK`. W niewidoczny dla nas sposób funkcja ta generuje znacznik `script` oraz funkcję. Następnie zastępuje łańcuch `JSON_CALLBACK` nazwą wygenerowanej funkcji i wykonuje żądanie.

Spróbujmy teraz otworzyć podany wcześniej przykład w serwisie jsFiddle i wpisać w nim takie symbole jak `GOOG`, `MSFT` lub `YHOO`, by się przekonać, że faktycznie pobiera on notowania giełdowe wybranej spółki. W rejestrze operacji sieciowych przeglądarki pojawią się takie wpisy:

```
https://query.yahooapis.com/... &callback=angular.callbacks._1
```

W powyższym przykładzie automatycznie wygenerowana funkcja ma nazwę `angular.callback._1`. Natomiast odpowiedź ma następującą postać:

```
angular.callback._1({"query": ...});
```

Jak widać, odpowiedź jest zapisana wewnątrz funkcji zwrotnej. AngularJS analizuje i wykonuje tę odpowiedź, co w efekcie prowadzi do wywołania funkcji zwrotnej `angular.callbacks._1`. Z kolei ta funkcja przekazuje dane do zdefiniowanej w naszym kodzie funkcji zwrotnej `success`.

Mam nadzieję, że informacje te pozwolą Ci zrozumieć działanie techniki JSONP i mechanizmu służącego do wykonywania żądań JSONP. Technika ta ma jednak pewne ograniczenia:

- Przede wszystkim pozwala ona wyłącznie na wykonywanie żądań `GET` (co jest oczywiste, ponieważ żądania te powstają przez wygenerowanie znaczników `script`).
- Dodatkowo także serwer musi obsługiwać pewne elementy tego rozwiązania, a mianowicie musi zapisywać zwracane wyniki wewnątrz funkcji zwrotnej.
- Oprócz tego z techniką tą zawsze jest związane pewne zagrożenie, gdyż bazuje ona na dynamicznym generowaniu i wstrzykiwaniu znaczników `script`.
- I w końcu także obsługa potencjalnych błędów nie jest zbytnio godna zaufania, ponieważ nie jest łatwo określić, z jakiego powodu nie udało się wczytać skryptu.

W pierwszej kolejności musimy jednak zdać sobie sprawę z faktu, że JSONP jest bardziej sposobem ominięcia problemu niż jego rozwiązaniem. W czasie, gdy rozwój internetu kierował nas w stronę Sieci 2.0, w której aplikacje typu mushup stawały się coraz bardziej popularne, a coraz więcej usług udostępniało swoje internetowe API, opracowano znacznie lepsze rozwiązanie: CORS.

CORS — Cross-Origin Resource Sharing

CORS to mechanizm, dzięki któremu serwery mogą zarządzać dostępem z innych domen, zapewniając tym samym przeglądarkom możliwość wykonywania z poziomu skryptów żądań skierowanych do innych domen. Dzięki temu standardowi aplikacje konsumenckie (takie jak nasza aplikacja *Mój trening*) są w stanie wykonywać określone — podstawowe — typy żądań, bez żadnych specjalnych wymagań i przygotowań. Do tych podstawowych żądań zaliczają się żądania `GET`, `POST` (w razie korzystania z określonych typów MIME) oraz `HEAD`. Wszystkie inny typy żądań HTTP są uznawane za żądania złożone.

Kup ksi k Pole ksi k

W przypadku żądań złożonych mechanizm CORS wymaga, by były one poprzedzone żądaniem HTTP `OPTIONS` (nazywanym także żądaniem sprawdzającym), które sprawdza, jakie żądania mogą być przesyłane na dany serwer z innych domen. Dopiero po pomyślnym wykonaniu takiego sprawdzenia można przesłać właściwe żądanie.

> Więcej informacji na temat CORS można znaleźć w dokumentacji MDN, dostępnej na stronie *https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS*.

Najlepszą cechą mechanizmu CORS jest to, że klient nie musi wprowadzać żadnych zmian, jak było w przypadku techniki JSONP. Cały ten mechanizm wymiany komunikatów jest całkowicie niewidoczny dla kodu, który z niego korzysta, a wywołania AJAX wykonywane przy użyciu standardowych usług AngularJS są realizowane bez żadnych problemów.

Stosowanie mechanizmu CORS wymaga zastosowania odpowiedniej konfiguracji po stronie serwera, a serwery MongoLab zostały tak przygotowane i zapewniają możliwość obsługi żądań nadsyłanych z innych domen. Przedstawione wcześniej żądanie `POST` skierowane do serwisu MongoLab było poprzedzone sprawdzającym żądaniem `OPTIONS`.

W ten sposób poznaliśmy podstawowe zagadnienia związane z usługą `$http` i wykonywaniem żądań do innych domen. Kolejnym zagadnieniem, któremu musimy się przyjrzeć, jest usługa `$resource`.

Usługa $resource

Prezentację usługi `$resource` powinniśmy zacząć od wyjaśnienia, dlaczego jest nam ona potrzebna. Wydaje się bowiem, że usługa `$http` jest w stanie wykonywać wszelkie potrzebne interakcje ze zdalnymi serwerami. Dlaczego jest nam zatem potrzebna kolejna abstrakcja i z jakiego rodzaju systemów pozwala ona korzystać?

Aby odpowiedzieć na to pytanie, musimy omówić nowy rodzaj usług (przy czym chodzi tu o usługi działające na serwerze, a nie usługi AngularJS): usługi typu *RESTful*.

Usługi typu RESTful i ich API

„Istnieje już do tego odpowiednie API!”.

To nie jest hasło reklamowe Apple'a, choć doskonale odpowiada ono obecnym realiom. Istnieją już API umożliwiające wykonywanie chyba wszystkich możliwych operacji. Niemal wszystkie usługi, zarówno te dostępne publicznie, jak i prywatne (takie jak Google, Facebook, Twitter i inne), udostępniają swoje API. A jeśli z danego API można korzystać za pomocą protokołu

Kup ksi k Pole ksi k

HTTP, to istnieje całkiem spore prawdopodobieństwo, że będzie to API typu RESTful. Nie musimy zresztą szukać zbyt daleko, wszak także serwis MonogLab udostępnia API typu RESTful, a co więcej, nawet już z niego korzystaliśmy!

REST (ang. *Representational State Transfer*) jest stylem architektury, który definiuje komponenty systemu w formie zasobów. Akcje są definiowane na poziomie zasobów, a serwer kontroluje przepływ procesów dynamicznie, korzystając przy tym z pojęcia hipermediów.

> W tym rozdziale nie będziemy zajmować się szczegółowo zagadnieniem samych usług typu RESTful, lecz skoncentrujemy się na tym, w jaki sposób AngularJS ułatwia korzystanie z usług tego typu. Jeśli Cię interesują zagadnienia związane z działaniem usług typu RESTful, to zajrzyj do doskonałego artykułu w witrynie InfoQ — *http://www.infoq.com/articles/webber-rest-workflow*. To naprawdę fascynująca lektura!

Większość interfejsów API, przygotowanych jako API typu RESTful, może nie być prawdziwymi usługami tego typu, a jedynie spełniać kilka spośród wszystkich narzucanych im wymagań. Usługi typu RESTful używane za pośrednictwem protokołu HTTP mają przynajmniej następujące cechy wspólne:

- Zasoby są zdefiniowane przy użyciu adresów URL. Oto dwa przykłady takich zasobów:
  - Zasoby kolekcji — są one pobierane przy użyciu adresów URL o postaci `http://myserver.com/resources`.
  - Zasoby elementów kolekcji — są one pobierane przy użyciu adresów URL o postaci `http://myserver.com/resources/id`, gdzie `id` identyfikuje konkretny zasób w kolekcji.
- Czasownik HTTP `GET` służy do pobierania danych z kolekcji, czy też do pobierania zasobów *elementu kolekcji*.
- Żądania HTTP `POST` służą do tworzenia nowych zasobów.
- Żądania HTTP `PUT` służą do aktualizacji istniejących zasobów.
- Żądania HTTP `DELETE` służą do usuwania zasobów.

A teraz wróćmy na początek tego rozdziału, do podpunktu pt. „Wczytywanie danych ćwiczeń i treningów”, i przyjrzyjmy się wzorcom adresów URL punktów kontrolnych serwisu MongoLab; są one zgodne z podanymi wcześniej założeniami.

AngularJS ma usługę `$resource` przeznaczoną właśnie do obsługi serwerów udostępniających usługi typu RESTful, do których można odwoływać się za pomocą protokołu HTTP. W kilku kolejnych częściach rozdziału wyjaśnię, jak działa usługa `$resource`, i wykorzystamy ją w wybranych fragmentach aplikacji *Mój trening*.

Kup ksi k Pole ksi k

Podstawowe informacje o usłudze $resource

Usługa `$resource` jest abstrakcją utworzoną w oparciu o usługę `$http`, która ma nam ułatwiać korzystanie z usług typu RESTful (działających na serwerach). W AngularJS zasób jest definiowany w następujący sposób:

```
$resource(url, [paramDefaults], [actions]);
```

A oto parametry wywołania tej usługi:

- *`url`* — określa adres URL punktu końcowego. W tym adresie można umieszczać sparametryzowane argumenty, poprzedzając je znakiem dwukropka (`:`). Poniżej przedstawiam dwa przykłady takich adresów URL:
  - `/collection/:identifier` — w tym adresie URL został sparametryzowany fragment identyfikatora;
  - `/:collection/:identifier` — natomiast w tym adresie zostały sparametryzowane oba fragmenty, zarówno kolekcji, jak i identyfikatora.

  Jeśli wartość parametru nie jest dostępna w momencie wywołania, to dany parametr zostanie usunięty z adresu URL. Sposób działania tych parametrów wyjaśniam na przykładach zamieszczonych poniżej.
- *`paramDefaults`* — ma podwójne znaczenie. W przypadku sparametryzowanych adresów URL parametr *`paramDefaults`* określa domyślne wartości zamienników, natomiast wszystkie dodatkowe wartości zostaną dołączone do łańcucha zapytania umieszczonego na końcu adresu URL.

  Przeanalizujmy adres URL o postaci `/users/:name`. Tabela 5.1 przedstawia finalną postać adresu URL w zależności od przekazanego parametru *`paramDefaults`*.

Tabela 5.1. Wpływ parametrów na postać adresu URL

Wartość parametru paramDefaults	Wynikowy adres URL
`{}`	`/users`
`{name:'dawid'}`	`/users/dawid`
`{search:'dawid'}`	`/users?search=szukane`
`{name:'dawid', search:'szukane'}`	`/users/dawid?search=szukane`

  Jak się niebawem dowiesz, te parametry można także przesłaniać podczas wywoływania konkretnej akcji.
- *`actions`* — jest jedynie zwyczajną funkcją JavaScriptu dołączoną do obiektu `$resource` w celu wykonania określonej czynności. Obiekt `$resource` udostępnia standardowy zestaw operacji, które przeważnie można wykonywać na wszystkich zasobach, takich jak `get`, `query`, `save` oraz `delete`. Parametr `actions` pozwala rozszerzyć domyślną listę akcji o własne operacje bądź zmodyfikować działanie którejś z predefiniowanych akcji.

Kup ksi k Pole ksi k

Parametr `actions` jest obiektem, którego kluczami są nazwy akcji, a wartościami obiekty `config`, przy czym są to te same obiekty `config`, które są używane w usłudze `$http` (przekazywane jako drugi argument jej wywołania).

Utworzenie zasobu za pomocą przedstawionej wcześniej instrukcji deklaracji zasobu zwraca klasę `Resource`. Klasa ta hermetyzuje konfigurację, którą podaliśmy podczas jej tworzenia. Aby wygenerować żądanie przy użyciu tej klasy, należy wywołać jedną z udostępnianych przez nią metod akcji (w tym także akcji, które sami zdefiniowaliśmy).

Przyjrzyjmy się teraz kilku praktycznym przykładom wywoływania akcji zasobów i spróbujmy nieco lepiej zrozumieć przeznaczenie trzeciego parametru usługi `$resource` — `actions`.

Wyjaśnienie akcji usługi $resource

Aby zrozumieć, w jaki sposób należy wywoływać akcje zasobów i rolę parametru `actions` w definiowaniu zasobów, przeanalizujmy poniższy przykład. Załóżmy, że zasób jest używany w następujący sposób:

```
var Exercises = $resource('https://api.mongolab.com/
  api/1/databases/angularjsbyexample/collections/
  exercises/:param,{},{update:{action:PUT'}});
```

Powyższa instrukcja tworzy klasę `Exercises`, udostępniającą w sumie sześć akcji: `get`, `save`, `update`, `query`, `remove` oraz `delete`. Pięć z nich to standardowe akcje definiowane dla każdego zasobu, natomiast szósta, `update`, została dodana do tej klasy zasobów poprzez przekazanie funkcji w parametrze `actions` (trzecim argumencie wywołania usługi `$resource`). Deklaracja parametru `actions` ma następującą postać:

```
actions:{akcja1 : config, akcja2 : config, akcja3 : config}
```

Ten kod definiuje trzy akcje wraz z ich konfiguracją. Używany tu obiekt `config` jest tym samym obiektem, który jest stosowany w wywołaniu usługi `$http`.

W powyższym przykładzie obiekt `config` przekazany w celu skonfigurowania akcji `update` zawiera tylko jedną właściwość, `action` (nie należy go mylić z parametrem `action` usługi `$resource`), określającą czasownik protokołu HTTP, który należy zastosować w momencie wywołania metody akcji `update`.

W przypadku pięciu domyślnych akcji usługi `$resource` stosowane obiekty `config` mają następującą postać:

```
{  'get':    {method:'GET'},
   'save':   {method:'POST'},
   'query':  {method:'GET', isArray:true},
   'remove': {method:'DELETE'},
   'delete': {method:'DELETE'}
};
```


Kup ksi k   Pole ksi k

Czasowniki protokołu HTTP stosowane w tych akcjach mają sens i są zgodne z wzorcem adresów URL usług typu RESTful. Nieco zaskakujące jest pominięcie akcji `update` lub akcji wykonującej żądania HTTP `PUT`. Z tego względu w przypadku definiowania punktu końcowego usługi typu RESTful może się pojawić konieczność uzupełnienia standardowej listy akcji o akcję `update` wykonującą żądania `PUT`. Właśnie to zrobiliśmy w pierwszym przykładzie przedstawionym w tym punkcie rozdziału.

W powyższej konfiguracji interesujący wydaje się atrybut `isArray` zastosowany w akcji `query`. Aby zrozumieć jego działanie, musisz zobaczyć, w jaki sposób są wywoływane akcje zasobów.

Wywoływanie akcji zasobów

Wywołanie usługi `$resource` zamieszczone na początku poprzedniego punktu rozdziału tworzy jedynie klasę zasobu o nazwie `Exercises`. Aby wykonać operację odwołującą się do serwera, musimy wywołać jedną z sześciu metod akcji zdefiniowanych w klasie `Exercises`. Poniżej przedstawiam trzy proste przykłady takich wywołań:

```
Exercises.query();   // pobiera wszystkie treningi
Exercises.get({id:'test'});   // pobiera ćwiczenie o identyfikatorze 'test'
Exercises.save({},workout);   // zapisuje trening
```

W przypadku metod akcji korzystających z żądań `GET` ogólna składnia ich wywołań ma następującą postać:

```
Exercises.actionName([parameters],[successcallback],[errorcallback]);
```

Z kolei w przypadku metod akcji korzystających z żądań `POST` ogólna składania ich wywołań wygląda następująco:

```
Exercises.actionName([parameters],[postData],[successcallback],[errorcallback]);
```

Jak widać, akcje korzystające z żądań `POST` wymagają przekazania dodatkowego parametru, `postData`, zawierającego faktyczne dane, które zostaną wysłane na serwer.

Ostatnie dwa parametry, `successcallback` i `errorcallback`, to funkcje zwrotne, które zostaną wywołane po odebraniu odpowiedzi, przy czym to, która z nich zostanie wykonana, będzie zależeć od statusu odpowiedzi.

Wywołanie akcji zasobu powoduje zwrócenie jednego z dwóch możliwych wyników:

- obiektu klas `Resource` (czyli obiektu zasobu) — jest on zwracany, jeśli parametr konfiguracyjny `isArray` danej akcji przyjmie wartość `false`, jak dzieje się, na przykład, w przypadku akcji `get`;
- pustej tablicy — jest ona zwracana, jeśli parametr konfiguracyjny `isArray` przyjmie wartość `true`, jak się dzieje, na przykład, w przypadku akcji `query`.

Kup ksi k Pole ksi k

Jak widać, działanie usługi `$resource` całkowicie różni się pod tym względem od działania usługi `$http`, która zawsze zwraca obietnice.

Jeśli zachowamy zwróconą wartość, to AngularJS zapisze w zwróconym obiekcie lub tablicy odpowiedź, która w przyszłości zostanie odebrana z serwera. Ten sposób działanie usługi `$resource` pozwala na tworzenie kodu, w którym nie trzeba implementować funkcji zwrotnych. Na przykład w kontrolerze `ExerciseListController` moglibyśmy wczytać wszystkie ćwiczenia, używając poniższego wywołania:

```
$scope.exercises = Exercises.query();
```

Powyższe wywołanie akcji `query` powoduje natychmiastowe zwrócenie pustej tablicy. W przyszłości, kiedy serwer prześle odpowiedź, jej zawartość zostanie zapisana w tablicy. A co ciekawsze, dzięki fantastycznej infrastrukturze dowiązywania danych, jaką ma AngularJS, wszelkie dowiązania tablicy `exercises` do widoku zostaną automatycznie zaktualizowane.

Kolejnym interesującym aspektem parametru konfiguracyjnego `isArray` akcji jest to, że w razie nieprawidłowego określenia jego wartości może on doprowadzić do nieprawidłowego przetworzenia odpowiedzi. Parametr ten pomaga AngularJS określić, czy odpowiedź należy zdeserializować jako tablicę czy jako obiekt. W razie zastosowania nieprawidłowej wartości tego parametru AngularJS zgłasza wyjątek o następującej treści:

```
"Error in resource configuration. Expected response to contain an object but got
↪an array"[3]
```

W przeciwnym razie zgłaszany jest wyjątek o treści:

```
"Error in resource configuration. Expected response to contain an array but got
↪an object"[4]
```

Bardzo łatwo można doprowadzić do zgłoszenia tych wyjątków. Spróbujmy wykonać następujące wywołania:

```
Exercises.get(); // zwraca tablicę
Exercises.query({params:'plank'}); // zwraca obiekt ćwiczenia
```

Wykonanie pierwszego wywołanie z powyższego przykładu spowoduje zgłoszenie pierwszego z przedstawionych wcześniej błędów, natomiast kolejne wywołanie spowoduje zgłoszenie drugiego błędu. Wystarczy spojrzeć na konfigurację obu metod akcji — `get` i `query` — by się przekonać, dlaczego te błędy zostały zgłoszone.

Zanim przejdziemy do kolejnego zagadnienia, koniecznie należy jeszcze raz o czymś przypomnieć. Chodzi o różnice w wartościach wynikowych zwracanych przez usługi `$resource` i `$http`.

[3] Błąd w konfiguracji zasobu. Oczekiwano odpowiedzi zawierającej obiekt, a otrzymana odpowiedź zawierała tablicę — *przyp. tłum.*

[4] Błąd w konfiguracji zasobu. Oczekiwano odpowiedzi zawierającej tablicę, a otrzymana odpowiedź zawierała obiekt — *przyp. tłum.*

Kup ksi k Pole ksi k

Usługa `$http` zawsze zwraca *obietnicę*, natomiast w przypadku usługi `$resource` może to być obiekt klasy `Resource` bądź tablica. Z tego względu istnieje możliwość bezpośredniego dowiązania wyników zwracanych przez usługę `$resource` do widoku, bez konieczności stosowania funkcji zwrotnych.

Obiekt i kolekcja zasobów zwracane przez wywołanie akcji zawierają kilka przydatnych właściwości:

- `$promise` — zawiera obietnicę używaną do obsługi żądania. W razie potrzeby można jej użyć do oczekiwania na odebranie odpowiedzi, podobnie jak w przypadku korzystania z usługi `$http`. Ewentualnie można także skorzystać z funkcji zwrotnych `successcallback` i `errorcallback` rejestrowanych podczas wywoływania usługi `$resource`.
- `$resolved` — przyjmuje wartość `true`, jeśli powyższa obietnica została wyznaczona, bądź wartość `false` w przeciwnym razie.

Spróbujmy teraz wykorzystać zdobytą wiedzę w praktyce i zmienić fragment aplikacji *Mój trening* w taki sposób, by odwoływała się do serwera, używając usługi `$resource`.

Dostęp do danych ćwiczeń przy użyciu usługi $resource

Do tej pory używaliśmy usługi `$http` do zarządzania danymi treningów i ćwiczeń. Aby jednak dokładniej poznać działanie usługi `$resource`, zmienimy teraz implementację naszej aplikacji w taki sposób, by korzystała z tej usługi do wczytywania i zapisywania danych ćwiczeń.

Zaczniemy od zmodyfikowania pliku *services.js*, w którym wewnątrz usługi `WorkoutService`, bezpośrednio przed funkcją `service.getExercises`, dodamy poniższy fragment kodu:

```
service.Exercises = $resource(collectionsUrl + "/exercises/:id",
  { apiKey: apiKey}, { update: { method: 'PUT' } });
```

Ta instrukcja tworzy klasę `Resource` skonfigurowaną w taki sposób, by używała podanego adresu URL i klucza API. Klucz API jest przekazywany za pomocą kolekcji domyślnych parametrów.

Następnie musimy usunąć z usługi `WorkoutService` wszystkie funkcje związane z obsługą ćwiczeń. Dotyczy to funkcji `service.getExercises`, `service.getExercise`, `service.updateExercise`, `service.addExercise` oraz `service.deleteExercise`. Obecnie wszystkie operacje związane z ćwiczeniami będą wykonywane z użyciem zasobów.

Funkcja `$resource` należy do modułu `ngResource`, dlatego do pliku *index.html* musimy dodać odpowiednie odwołanie; należy je umieścić poniżej odwołań do innych modułów AngularJS:

```
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/
  1.3.3/angular-resource.js"></script>
```


Kup ksi k Pole ksi k

Następnie do pliku *app.js* musimy dodać zależność od modułu `ngResource`:

```
angular.module('app', [...,'ngResource']);
```

Na koniec musimy także dodać usługę `$resource` jako zależność do naszej usługi `WorkoutService`. Trzeba przy tym pamiętać, że zależność tę należy dodać do funkcji `this.$get`.

Powyższe zmiany mają również wpływ na funkcję `service.getWorkout`, gdyż jej implementacja jest zależna od funkcji `getExercises`. Z tego względu wywołanie `service.getExercises()` umieszczone wewnątrz wywołania `$q.all` należy zastąpić fragmentem:

```
service.Exercises.query().$promise
```

Akcja `query` zwraca pustą tablicę, która dysponuje predefiniowaną właściwością `$promise`, na którą może oczekiwać wywołanie `$q.all`.

A teraz, ponieważ usunęliśmy z usługi kilka funkcji, musimy poprawić pozostałe fragmenty aplikacji, które jej używały.

Zacznijmy od zmodyfikowania implementacji listy ćwiczeń, gdyż to będzie najprościej zrobić. W pierwszej kolejności poprawimy metodę `init` kontrolera `ExercisesNavController` zdefiniowaną w pliku *exercise.js* w katalogu *WorkoutBuilder*. Jej dotychczasową zawartość należy zastąpić poniższą instrukcją:

```
$scope.exercises = WorkoutService.Exercises.query();
```

Dokładnie w taki sam sposób trzeba zmodyfikować implementację funkcji `init` kontrolera `ExerciseListController`.

Pusta tablica zwracana przez wywołanie funkcji `query` zostanie wypełniona danymi w przyszłości, po odebraniu odpowiedzi. Po zaktualizowaniu modelu `exercises` także widok, do którego jest on dowiązany, zostanie automatycznie zaktualizowany. I nie wymaga to stosowania żadnych funkcji zwrotnych!

A teraz zajmiemy się poprawieniem strony do dodawania ćwiczeń (`#/builder/exercises/new`), kontrolera `ExerciseDetailController` oraz usług, które są przez nie używane. W ich przypadku wszystkie odwołania do usługi `$http` należy zastąpić odwołaniami do usługi `$resource`. Zaczniemy od zmodyfikowania funkcji `startBuilding` usługi `ExerciseBuilderService` zdefiniowanej w pliku *services.js* w katalogu *WorkoutBuilder*. Poniżej przedstawiam jej nową implementację:

```
service.startBuilding = function (name) {
    if (name) {
        buildingExercise = WorkoutService.Exercises.get({ id: name }, function (data) {
            newExercise = false;
        });
    }
    else {
        buildingExercise = new Exercise({});
        newExercise = true;
```


Kup ksi k   Pole ksi k

```
        }
        return buildingExercise;
    };
```

Zastosowaliśmy tu metodę akcji `get` zasobu `Exercise`, aby na podstawie podanej nazwy (`{id:name}`) pobrać dane konkretnego ćwiczenia. Trzeba przy tym pamiętać, że nazwa ćwiczenia jest używana jako jego identyfikator.

Zanim przypiszemy fladze `newExercise` wartość `false`, musimy zaczekać na otrzymanie odpowiedzi. Używamy w tym celu funkcji zwrotnej wywoływanej w przypadku pomyślnego wykonania operacji. Warto zauważyć, że parametr `data` tej funkcji zwrotnej i zmienna `buildingExercise` odwołują się do tego samego obiektu zasobu.

Ponieważ nie korzystamy już z obietnic, kod klauzuli `else` został przywrócony do wcześniejszej postaci, używanej przed zastosowaniem usługi `$http`.

Aby poprawić implementację kontrolera `ExerciseDetailController`, musimy przywrócić wcześniejszą wersję jego metody `init`, w której nie były używane funkcje zwrotne:

```
$scope.exercise =
    ExerciseBuilderService.startBuilding($routeParams.id);
```

W ten sposób udało się nam poprawić wszystkie fragmenty aplikacji związane z pobieraniem danych ćwiczeń. Jak można się przekonać, zastosowanie zasobów faktycznie umożliwiło uproszczenie kodu — wyeliminowaliśmy przeważającą większość funkcji zwrotnych używanych przez usługę `$http`. Asynchroniczny charakter odwołań do zasobu został w znacznej mierze ukryty, co ma zarówno swoje dobre, jak i złe strony. Z jednej strony to dobrze, gdyż pozwala uprościć kod, z drugiej źle, bo ukrywa jego asynchroniczny charakter, a to z kolei często prowadzi do problemów ze zrozumieniem działania kodu i przyczyn występujących w nim błędów.

Ukryty koszt ukrywania asynchronicznego charakteru kodu

Głównym celem usługi `$resource` jest uproszczenie korzystania z usług typu RESTful. Przy okazji można wyeliminować liczbę funkcji zwrotnych, które trzeba by zaimplementować, gdyby usługa `$resource` nie była używana. Stosowanie tej abstrakcji ma jednak swoją cenę. W ramach przykładu przeanalizujmy następujący fragment kodu:

```
$scope.exercises = WorkoutService.Exercises.query();
console.log($scope.exercises.length);
```

Można by sądzić, że wywołanie `console.log` spowoduje wyświetlenie długości tablicy `exercises`, ale takie przypuszczenie byłoby całkowicie błędne. W rzeczywistości właściwość `$scope.exercises` zawiera pustą tablicę, a zatem wywołanie funkcji `log` wyświetli wartość `0`. Tablica ta zostanie wypełniona danymi w przyszłości, gdy zostaną one przesłane przez serwer. Silnik języka JavaScript nie będzie czekał na otrzymanie tych danych w pierwszej instrukcji powyższego fragmentu. Sprawia on wrażenie, że wszystkie operacje są wykonywane sekwencyjnie, choć w rzeczywistości jest inaczej.

Kup ksi k Pole ksi k

> Ponieważ cykle przeglądu są wykonywane po odebraniu przez usługę `$resource` odpowiedzi z serwera, dowiązywanie danych do interfejsu użytkownika wciąż działa prawidłowo.
>
> W ramach tych cykli przeglądu wykonywane jest sprawdzanie zmian pozwalające określić, czy w modelu aplikacji wprowadzono jakieś modyfikacje. Wszystkie te zmiany modelu wywołują aktywację czujek, które z kolei wymuszają aktualizację dowiązań modelu od interfejsu użytkownika i wstawek. Zagadnienia związane z cyklami przeglądu zostały opisane w rozdziale 3., pt. „Stosowanie kolejnych dobrodziejstw AngularJS".

Jeśli którekolwiek z wykonywanych operacji zależą od czasu, w jakim dane zostaną odebrane, to konieczne będzie zastosowanie obietnic i funkcji zwrotnych. Właśnie w taki sposób działa funkcja `startBuilding`, która czeka na odebranie danych ćwiczenia, a dopiero potem ustawia flagę `newExercise`.

> Nie sugeruję bynajmniej, aby nie używać usługi `$resource`. W rzeczywistości jest to fantastyczna usługa, która umożliwia wyeliminowanie sporej ilości kodu, który byłby niezbędny w przypadku stosowania usługi `$http`. Niemniej jednak, stosując usługę `$resource`, należy pamiętać o jej osobliwościach i ich konsekwencjach.

A teraz zajmiemy się poprawieniem podstawowych operacji (CRUD) na ćwiczeniach.

Podstawowe operacje z użyciem usługi $resource

Zasób `Exercise` zdefiniowany w usłudze `WorkoutService` dysponuje już akcjami `save` i `update` (to akcje niestandardowe, które zostały dodane przez nas). Reszta sprowadza się do wywoływania odpowiednich akcji wewnątrz funkcji usługi `WorkoutBuilderService`.

Funkcją usługi `ExerciseBuilderService`, którą poprawimy w pierwszej kolejności, będzie `save`. Poniżej przedstawiam jej nową implementację:

```
service.save = function () {
    if (!buildingExercise._id) buildingExercise._id = buildingExercise.name;
    var promise = newExercise ?
            WorkoutService.Exercises.save({}, buildingExercise).$promise
            : buildingExercise.$update({ id: buildingExercise.name });
    return promise.then(function (data) {
        newExercise = false;
        return buildingExercise;
    });
};
```


Kup ksi k   Pole ksi k

W powyższej implementacji wywołujemy odpowiednią akcję zasobu, zależnie od stanu właściwości newExercise. Następnie pobieramy używaną obietnicę i korzystamy z niej do utworzenia (przy użyciu funkcji then) łańcucha obietnic, który w przyszłości zwróci to samo ćwiczenie.

Operacja save nie tylko używa klasy Resource (Exercise), lecz także obiektu Resource (building↪Exercise). Powyższy kod prezentuje istotną różnicę między klasą Resource i obiektem zasobu. Trzeba pamiętać, że buildingExercise jest obiektem zasobu ustawianym przez funkcję start↪Building wywoływaną w kontrolerze ExerciseDetailController.

Obiekt zasobu jest zazwyczaj tworzony w ramach wywołania operacji get odpowiadającej mu klasy Resource, na przykład:

```
buildingExercise = WorkoutService.Exercises.get({ id: name });
```

Powyższe wywołanie tworzy obiekt zasobu ćwiczenia. Z kolei następne wywołanie tworzy tablicę:

```
$scope.exercises = WorkoutService.Exercises.query();
```

Po odebraniu odpowiedzi z serwera tablica ta zostanie wypełniona obiektami zasobów ćwiczeń.

Akcje dostępne w obiekcie zasobu odpowiadają akcjom klasy Resource, a jedyna różnica polega na tym, że ich nazwy zaczynają się od znaku $. Co więcej, akcje obiektu zasobu mogą operować na danych przechowywanych w tym obiekcie. Na przykład zastosowane we wcześniejszym przykładzie wywołanie buildingObject.$update nie wymaga przekazywania danych w formie argumentu, natomiast jest to konieczne w przypadku wywołania Exercise.save (gdzie dane ćwiczenia są przekazywane jako drugi argument wywołania).

Zamieszczona poniżej tabela 5.2 zawiera zestawienie różnic w sposobie korzystania z klasy Resource i obiektów zasobów.

Tabela 5.2. Różnice między stosowaniem klasy Resource i obiektu zasobu

	Klasa Resource	**Obiekt zasobu**
Tworzenie	Jest tworzona przy użyciu wywołania `$resource(url, param, actions)`	Jest tworzony w ramach wykonania akcji. Oto przykład: `exercise = WorkoutService. Exercises.get({id:name});`
Akcje (przeglądanie danych)	`Exercises.get({id:name});` `Exercise.query();`	`exercise.$get({id:name});` `exercise.$query();`

Akcje (operacje CRUD)	`Exercise.save({}, data);`
	`Exercise.update({id:name}, data);`
	`Exercise.delete({id:name});`
		`exercise.$save({});`
		`exercise.$update({id:name});`
		`exercise.$delete({id:name});`

Wartość wynikowa	Zwraca obiekt zasobu lub tablicę obiektów zasobu zawierającą dodatkowe właściwości `$promise` i `$resolved`.	Zwraca obiekt obietnicy.

Kup książkę Poleć książkę

> Obiektu zasobu należy używać w przypadkach, gdy operacja jest wykonywana w kontekście jednego elementu. Przykładami takich operacji są `update` i `delete`. W pozostałych sytuacjach preferowane jest stosowanie usługi `$resource`.

Operacja usuwania jest bardzo prosta — sprowadza się do wywołania akcji `$delete` obiektu zasobu, która zwraca obiekt obietnicy:

```
service.delete = function () {
    return buildingExercise.$delete({ id: buildingExercise.name });
};
```

W przypadku tej operacji nie musimy modyfikować kodu kontrolera `WorkoutDetailController`, gdyż podobnie jak wcześniej także teraz funkcje `save` i `delete` usługi `WorkoutBuilderService` zwracają obietnice.

W ten sposób poprawki w funkcji `$resource` są już gotowe i możemy przetestować implementację. Spróbujmy zatem wczytać różne ćwiczenia i je edytować, by się upewnić, czy wszystko działa prawidłowo.

> W razie problemów działający kod aplikacji po tym etapie prac można znaleźć w przykładach dołączonych do książki, w katalogu *rodzial05\punkt_kontrolny_4*.

Funkcja `$resource` jest bardzo użyteczną usługą AngularJS, ułatwiającą korzystanie z punktów końcowych usług typu RESTful obsługiwanych przy użyciu protokołu HTTP. A co z innymi punktami końcowymi, które *nie są zgodne z architekturą REST*? No cóż, w ich przypadku zawsze można korzystać z usługi `$http`. Jeśli jednak także podczas korzystania z takich punktów końcowych chcielibyśmy posługiwać się usługą `$resource`, to musimy pamiętać o kilku różnicach w sposobie dostępu do zasobów.

Stosowanie usługi $resource do obsługi innych rodzajów punktów końcowych

O ile punkt końcowy używa protokołu HTTP oraz zwraca i pobiera dane w formacie JSON (bądź w innym formacie, który można skonwertować do formatu JSON), to można z niego korzystać przy użyciu usługi `$resource`. W takich sytuacjach może być konieczne utworzenie odrębnych klas `Resource` służących do przeglądania danych i do wykonywania pozostałych operacji CRUD. Na przykład przeanalizujmy poniższe deklaracje:

```
$resource('/users/active');      // do przeglądania i pobierania
$resource('/users/createnew');   // do tworzenia
$resource('/users/update/:id');  // do aktualizacji
```


Kup ksi k

Pole ksi k

W takich przypadkach większość wywołań akcji jest wykonywana przy użyciu klasy `Resource`, a akcje obiektów zasobów mogą nie funkcjonować prawidłowo.

Może się nawet zdarzyć, że takie punkty końcowe nie są zgodne ze standardowymi sposobami korzystania z akcji protokołu HTTP. Na przykład żądania `POST` mogą być używane zarówno do zapisu, jak i do aktualizacji danych, a żądania `DELETE` mogą w ogóle nie być obsługiwane. Mogą występować także inne, podobne rozbieżności.

W ten sposób dotarliśmy do końca prezentacji usługi `$resource`. Zakończmy ją podsumowaniem wszystkiego, czego się o niej dowiedziałeś:

- Usługa `$resource` jest bardzo użyteczna do obsługi interakcji z usługami typu RESTful. Można jej także używać do obsługi innych rodzajów punktów końcowych.
- O ile punkt końcowy jest zgodny ze wzorcem działania charakterystycznego dla usług typu RESTful, to usługa `$resource` pozwala nam uniknąć tworzenia większości powtarzalnego kodu wymaganego do prowadzenia interakcji.
- Wywołania akcji usługi `$resource` zwracają obiekt zasobu (lub ich tablicę), który w przyszłości zostanie wypełniony danymi. Odróżnia to ją od usługi `$http`, która zawsze zwraca obiekt obietnicy.
- Dzięki temu, że akcje usługi `$resource` zwracają obiekty zasobów, niektóre scenariusze korzystania z tych zasobów możemy implementować bez stosowania funkcji zwrotnych. Nie oznacza to jednak wcale, że operacje wykonywane przy użyciu usługi `$resource` i obiektów zasobów są operacjami synchronicznymi.

W ten sposób poznałeś możliwości i sposoby korzystania z usług `$http` i `$resource`. Usługi te zapewniają ogromne możliwości i są w stanie zaspokoić wszystkie nasze potrzeby związane z interakcją z serwerami. W dalszej części tego rozdziału opiszę kilka ogólnych scenariuszy obsługi i pewne bardziej zaawansowane zagadnienia związane ze stosowaniem usług `$http` i `$resource`. W pierwszej kolejności zajmiemy się funkcjami przechwytującymi żądania i odpowiedzi.

Funkcje przechwytujące żądania i odpowiedzi

Jak można się domyślać, funkcje przechwytujące żądania i odpowiedzi (ang. *interceptors*, nazywane także czasami **interceptorami żądań i odpowiedzi**) umożliwiają przechwytywanie żądań i odpowiedzi oraz ich wzbogacanie lub modyfikowanie. Typowe przypadki zastosowania tych funkcji obejmują uwierzytelnianie, globalną obsługę błędów, operacje na nagłówkach HTTP, modyfikowanie adresów URL punktów końcowych, globalną logikę powtarzania operacji oraz kilka innych rozwiązań.

Kup ksi k Pole ksi k

Funkcje przechwytujące są implementowane w formie potoku i wywoływane jedna za drugą, dokładnie tak samo jak w przypadku potoków *analizy* i *formatowania* kontrolera `NgModelController` (opisanych w poprzednim rozdziale).

Funkcje przechwytujące mogą działać w czterech miejscach, dlatego istnieją aż cztery potoki tych funkcji. Są one wykonywane:

- przed wysłaniem żądania;
- po wystąpieniu błędu żądania (błąd żądania może się wydawać czymś dziwnym, niemniej jednak kiedy żądanie jest przekazywane w potoku od jednej funkcji do drugiej i jeśli jedna z nich dorzuci żądanie, na przykład ze względu na błąd walidacji danych, to takie żądanie trafia do potoku błędów wraz z informacją o przyczynie problemów);
- po odebraniu odpowiedzi z serwera;
- po odebraniu błędu z serwera bądź w przypadku, gdy z powodu jakichś uchybień komponent potoku odpowiedzi odrzuci prawidłową odpowiedź.

Funkcje przechwytujące w AngularJS są zazwyczaj implementowane jako *usługi typu* `factory`. Następnie, podczas etapu konfiguracji modułu, są one dodawane do kolekcji funkcji przechwytujących usługi `$httpProvider`.

Typowy schemat tworzenia usługi tego typu przedstawia poniższy przykład:

```
myModule.factory('myHttpInterceptor', function ($q, dependency1, dependency2) {
    return {
        'request': function (config) {},
        'requestError': function (rejection) {},
        'response': function (response) {},
        'responseError': function (rejection) {}
    };});
```

Poniższa instrukcja pokazuje, w jaki sposób usługa ta jest rejestrowana na etapie konfiguracji modułu:

```
$httpProvider.interceptors.push('myHttpInterceptor');
```

Funkcje przechwytujące `request` i `requestError` są wywoływane przed wysłaniem żądania, natomiast funkcje `response` i `responseError` — po odebraniu odpowiedzi. Implementowanie wszystkich czterech funkcji przechwytujących nie jest wymagane. Wystarczy zaimplementować tylko jedną z nich — tę, która spełnia nasze potrzeby.

Szkielet zastosowania funkcji przechwytujących można znaleźć w dokumentacji AngularJS, w sekcji *Interceptors* (na stronie *https://code.angularjs.org/1.3.3/docs/api/ng/service/$http*).

> Funkcja `$httpProvider` jest elementem AngularJS, którego użyliśmy po raz pierwszy. Podobnie jak inni *dostawcy* pozwala nam ona skonfigurować działanie usługi `$http` na etapie konfigurowania modułu.

Kup ksi k

Pole ksi k

Aby się przekonać, jak działają funkcje przechwytujące, spróbujmy taką funkcję zaimplementować!

Zastosowanie funkcji przechwytującej do przekazania klucza API

Nasza obecna implementacja usługi `WorkoutService` jest zaśmiecona odwołaniami do klucza API umieszczonymi we wszystkich wywołaniach lub deklaracjach związanych z usługami `$http` i `$resource`. Chodzi o fragmenty kodu takie jak ten poniżej:

```
$http.get(collectionsUrl + "/workouts", { params: { apiKey: apiKey} });
```

Każde odwołanie do API MongoLab wymaga dodania do łańcucha zapytania klucza API. Jest całkowicie jasne, że jeśli zaimplementujemy funkcję przechwytującą dodającą klucz API do każdego żądania kierowanego do MongoLab, to będziemy mogli pozbyć się z wywołań API MongoLab określania wartości właściwości `params`.

A zatem czas skorzystać z funkcji przechwytującej! Otwórzmy plik *services.js* umieszczony w katalogu *shared* i na jego końcu dodajmy poniższy fragment kodu:

```
angular.module('app')
    .provider('ApiKeyAppenderInterceptor', function () {
        var apiKey = null;
        this.setApiKey = function (key) {
            apiKey = key;
        }
        this.$get = ['$q', function ($q) {
            return {
                'request': function (config) {
                    if (apiKey && config && config.url.toLowerCase()
                            .indexOf("https://api.mongolab.com") >= 0) {
                        config.params = config.params || {};
                        config.params.apiKey = apiKey;
                    }
                    return config || $q.when(config);
                }
            }
        }];
    });
```

Powyższy kod tworzy dostawcę (usługę typu `provider`, a nie usługę typu `factory`). Funkcja `setApiKey` tego dostawcy pozwala na ustawienie klucza API, jeszcze zanim będzie mogła zostać użyta funkcja przechwytująca.

Jeśli chodzi o funkcję wytwórczą zwracaną przez `$get`, to implementujemy w niej wyłącznie *funkcję przechwytującą* `request`. Funkcja ta ma tylko jeden parametr, `config`, i zwraca obiekt `config` bądź obietnicę, której wyznaczenie zwróci ten obiekt. Jest to dokładnie ten sam obiekt `config`, który jest używany przez usługę `$http`.

Kup ksi k Pole ksi k

W naszej implementacji funkcji przechwytującej upewniamy się, że wartość właściwości `apiKey` została podana, a żądanie jest przesyłane na adres `api.mongolab.com`. Jeśli oba te warunki zostaną spełnione, to aktualizujemy obiekt `params`, dodając do niego właściwość `apiKey`, dzięki czemu klucz API będzie dodawany do łańcucha zapytania.

Nasza funkcja przechwytująca jest już gotowa, ale sposób jej implementacji wymaga wprowadzenia pewnych zmian.

Obecnie usługa `WorkoutService` nie potrzebuje już klucza API, więc możemy poprawić jej funkcję `configure`. Zmodyfikujmy zatem kod pliku *config.js*, dodając do funkcji modułu `config` zależność od dostawcy `ApiKeyAppenderInterceptorProvider`.

Wewnątrz funkcji `config`, na samym jej początku, musimy dodać dwa wiersze kodu:

```
ApiKeyAppenderInterceptorProvider.setApiKey("<mój_klucz>");
$httpProvider.interceptors.push('ApiKeyAppenderInterceptor');
```

Metodę `configure` dostawcy `WorkoutServiceProvider` musimy zmodyfikować w następujący sposób:

```
WorkoutServiceProvider.configure("<nazwa_bazy_danych>");
```

Sama deklaracja funkcji `configure` dostawcy `WorkoutServiceProvider` także wymaga modyfikacji. Poniżej został przedstawiony jej kod, którym należy zastąpić dotychczasową implementację tej funkcji umieszczoną w pliku *services.js*, w katalogu *shared*:

```
this.configure = function (dbName) {
    database = database;
    collectionsUrl = apiUrl + dbName + "/collections";
}
```

Ostatnim zadaniem jest usunięcie odwołań do klucza API ze wszystkich wywołań funkcji usług `$http` i `$resource`. Obecnie deklaracja zasobu powinna wyglądać następująco:

```
$resource(collectionsUrl + "/exercises/:id", {}, { update: { method: 'PUT' } });
```

Natomiast w przypadku wywołań funkcji usługi `$http` należy z nich usunąć obiekt `params`.

Teraz już możemy przetestować naszą implementację. W tym celu w przeglądarce wyświetlmy dowolną listę lub stronę ze szczegółowymi informacjami o ćwiczeniach lub treningach i sprawdźmy, czy są one wyświetlane prawidłowo. Warto także spróbować dodać punkt wstrzymania w kodzie funkcji przechwytującej i przekonać się, kiedy jest ona wywoływana.

> Zaktualizowany kod aplikacji jest dostępny w przykładach dołączonych do książki, w katalogu *rozdzial05/punkt_kontrolny_5*.

Kup ksi k   Pole ksi k

Przechwytywanie żądań i odpowiedzi zapewnia ogromne możliwości, pozwalające na zaimplementowanie dowolnych, nowoczesnych rozwiązań związanych ze zdalnymi wywołaniami wykonywanymi przy użyciu protokołu HTTP. Ich właściwe wykorzystanie umożliwia uproszczenie implementacji i zredukowanie ilości powtarzającego się kodu.

Funkcje przechwytujące operują na poziomie, na którym mogą manipulować pełnymi żądaniami i odpowiedziami. Dotyczy to nagłówków, punktów końcowych oraz samych komunikatów! Istnieje jednak jeszcze jeden mechanizm AngularJS, podobny do funkcji przechwytujących, ale związany wyłącznie z przekształceniami zawartości żądań i odpowiedzi. Są to tak zwane **funkcje przekształcające** (ang. *transformers*).

Funkcje przekształcające żądania i odpowiedzi

Zadaniem funkcji przekształcających jest zmiana postaci danych wejściowych z jednego formatu na inny. Funkcje te są dołączane do potoku przetwarzania żądań i odpowiedzi stosowanego w AngularJS i pozwalają na przekształcanie zawartości wysyłanych żądań i odbieranych odpowiedzi. Doskonałym przykładem takiej funkcji przekształcającej jest globalna funkcja stosowana w AngularJS, która przekształca odpowiedzi tekstowe zapisane w formacie JSON na odpowiedni obiekt JavaScriptu (i na odwrót).

Takie przekształcenie danych można wykonywać zarówno podczas wywołania żądania, jak i podczas przetwarzania odpowiedzi, dlatego dostępne są dwa potoki funkcji przekształcających — jeden do przekształcania żądań, a drugi do przekształcania odpowiedzi.

Funkcje przekształcające mogą być rejestrowane na dwa sposoby:

- Globalnie, dla wszystkich żądań lub odpowiedzi. Na tym globalnym poziomie jest rejestrowana standardowa funkcja przekształcająca łańcuchy JSON na obiekty i na odwrót. Aby zarejestrować globalną funkcję przekształcającą, trzeba ją dodać na początku lub końcu jednej z dwóch tablic: `$httpProvider.defaults.transformRequest` lub `$httpProvider.defaults.transformResponse`.

> Usługa `$http` także udostępnia właściwość `defaults`, stanowiącą odpowiednik właściwości `$http` ↪`Provider.defaults`. Dzięki temu można zmieniać te ustawienia konfiguracyjne w dowolnym momencie działania aplikacji.

- Lokalnie, w konkretnym wywołaniu akcji usługi `$http` lub `$resource`. Obiekt `config` udostępnia dwie właściwości: `transformRequest` i `transformResponse`. Można ich używać do rejestrowania dowolnych funkcji przekształcających. Tak określone funkcje przekształcające przesłaniają funkcje zarejestrowane dla danej akcji na poziomie globalnym.

Kup ksi k Pole ksi k

Obiekty $httpProvider.defaults i $http.defults zawierają także ustawienia związane z domyślnymi nagłówkami HTTP, wysyłanymi we wszystkich żądaniach.

W niektórych przypadkach mogą się one okazać bardzo przydatne. Na przykład, jeśli aplikacja serwerowa wymaga przesyłania jakiegoś nagłówka w każdym żądaniu, to można go dodać do kolekcji $http.defaults.headers.common, a w efekcie będzie on automatycznie dodawany do wszystkich żądań:

```
$http.defaults.headers.common.Authorization = 'Basic YmVlcDpib29w';
```

Wróćmy jednak do funkcji przekształcających. Z punktu widzenia implementacji funkcja przekształcająca ma tylko jeden argument, data, i musi zwracać wynik, którym są przekształcone dane.

Poniżej przedstawiam implementację jednej z funkcji przekształcających używanych przez AngularJS do zmiany obiektów JavaScriptu na łańcuchy znaków w formacie JSON. Funkcja ta należy do kodu frameworku AngularJS:

```
function(d) {
    return isObject(d) && !isFile(d) ? toJson(d) : d;
}
```

Funkcja ta pobiera parametr d i przekształca go na łańcuch znaków, wywołując wewnętrzną funkcję frameworku o nazwie toJson. Funkcja ta jest rejestrowana przez framework w globalnym potoku funkcji przekształcającej żądania.

Lokalne funkcje przekształcające są przydatne w przypadkach, kiedy wolimy uniknąć stosowania globalnego potoku funkcji przekształcających, a jednocześnie chcemy zrobić coś niestandardowego. Poniższy przykład pokazuje, w jaki sposób można zarejestrować funkcję przekształcającą na poziomie żądania HTTP:

```
service.Exercises = $resource(collectionsUrl + "/exercises/:id", {}, {
    update: { method: 'PUT' },
        get: {
            transformResponse: function (data) {
                return JSON.parse(data); }
             }
});
```

W powyższej deklaracji klasy Resource rejestrujemy funkcję przekształcającą odpowiedzi, która będzie używana przez akcję get. Funkcja ta konwertuje wejściowy łańcuch znaków (data) na obiekt, czyli działa analogicznie do globalnej funkcji przekształcającej odpowiedzi.

Słowo ostrzeżenia

Stosowanie lokalnych funkcji przekształcających w konkretnych akcjach usług $http lub $resource przesłania funkcje przekształcające określone globalnie.

Kup ksi k Pole ksi k

W powyższym przykładzie zmienna `data` będzie zawierała odpowiedź otrzymaną z serwera zapisaną w formie łańcucha znaków, a nie zdeserializowanego obiektu. Zapisując naszą niestandardową funkcję przekształcającą podpowiedzi we właściwości `transformResponse`, przesłoniliśmy domyślną funkcję przekształcającą, która odpowiada za deserializację odpowiedzi zapisanej w formacie JSON.

Jeśli się okaże, że konieczne jest wykonanie także domyślnej funkcji przekształcającej, niezbędne będzie utworzenie tablicy i zapisanie w niej zarówno standardowej, jak i niestandardowej funkcji przekształcającej, a następnie zapisanie tej tablicy we właściwości `transform↪Request` lub `transformResponse`. Poniżej przedstawiam przykład takiego rozwiązania:

```
service.Exercises = $resource(collectionsUrl + "/exercises/:id", {}, {
  update: { method: 'PUT' },
  get: {
      transformResponse: $http.defaults.transformResponse.concat(function (value) {
          return doTransform(value); })
  }
});
```

Następnym zagadnieniem, którym się zajmiemy, będzie wyznaczanie tras w przypadku odrzucenia obietnicy.

Obsługa błędów wyznaczania tras w przypadku odrzucenia obietnicy

Strona *Konstruktor treningów* wchodząca w skład aplikacji *Mój trening* zależy od funkcji w parametrze konfiguracyjnym `resolve`, której zadaniem jest wstrzyknięcie do kontrolera `WorkoutDetailController` wybranego treningu.

Właściwość konfiguracyjna `resolve` ma jeszcze jedną zaletę, która uwidacznia się w przypadkach, gdy którakolwiek ze zdefiniowanych w niej funkcji zwraca obietnicę, tak jak przedstawiona poniżej funkcja `selectedWorkout`:

```
return WorkoutBuilderService.startBuilding(
  $route.current.params.id);
```

Jeśli obietnicę uda się wyznaczyć prawidłowo, to dana zostanie wstrzyknięta do kontrolera. A co się stanie w razie odrzucenia obietnicy albo jeśli podczas jej wyznaczania pojawią się błędy? W przypadku powyższej obietnicy błędy mogą się pojawić, jeżeli w adresie URL podamy nazwę nieistniejącego treningu, na przykład `/builder/workouts/dummy`, a następnie spróbujemy wyświetlić stronę z informacjami o tym treningu. Podobne efekty pojawią się w przypadku wystąpienia błędów serwera. Jeśli obietnicy nie uda się wyznaczyć, mogą się zdarzyć dwie rzeczy:

- Po pierwsze, trasa w aplikacji nie zostanie zmieniona. Jeśli odświeżymy stronę w przeglądarce, jej zawartość zostanie wyczyszczona.

Kup ksi k Pole ksi k

- Po drugie, w głównym zasięgu aplikacji, `$rootScope`, zostanie rozgłoszone zdarzenie `$routeChangeError` (pamiętasz zapewne, że zdarzenia AngularJS mogą być emitowane za pomocą funkcji `$emit` i rozgłaszane przy użyciu funkcji `$broadcast`).

Tego zdarzenia możemy użyć, aby w jakiś wizualny sposób poinformować użytkownika o problemach z odnalezieniem ścieżki. Spróbujmy usprawnić stronę *Konstruktora treningów*, stosując w niej takie rozwiązanie.

Obsługa nieodnalezionych treningów

Błąd na stronie treningu można wywołać, próbując wywołać nieistniejący trening. Taki błąd musi zostać pokazany na poziomie pojemnika, poza dyrektywą `ng-view`.

Zaktualizujmy zatem plik *index.html* i dodajmy do niego następujący wiersz, umieszczając go przed deklaracją `ng-view`:

```
<label ng-if="routeHasError" class="alert alert-danger">{{routeError}}</label>
```

Kolejną zmianę musimy wprowadzić w pliku *root.js* — w jego kodzie trzeba zmodyfikować procedurę obsługi zdarzeń `$routeChangeSuccess`, dodając do niej wyróżniony wiersz kodu:

```
$scope.$on('$routeChangeSuccess', function (event, current,previous) {
    $scope.currentRoute = current;
    $scope.routeHasError = false;
});
```

Ponadto do tego samego pliku należy także dodać poniższą procedurę obsługi zdarzeń `$route➥ChangeError`:

```
$scope.$on('$routeChangeError', function (event, current, previous, error) {
    if (error.status === 404 && current.originalPath === "/builder/workouts/:id")
{
        $scope.routeHasError = true;
        $scope.routeError = current.routeErrorMessage;
    }
});
```

I w końcu ostatnią zmianę musimy wprowadzić w pliku *config.js* — tutaj do konfiguracji trasy dla strony z formularzem edycji treningów należy dodać właściwość `routeErrorMessage`:

```
$routeProvider.when('/builder/workouts/:id', {
    // ... istniejący kod konfiguracyjny ...
    topNav: 'partials/workoutbuilder/top-nav.html',
    routeErrorMessage:"Nie udało się wczytać wybranego treningu!",
    // ... istniejący kod konfiguracyjny ...
});
```


Kup ksi k Pole ksi k

A teraz przetestujmy te zmiany i spróbujmy wczytać w przeglądarce stronę `/builder/workouts/`
`↪nie_ma_takiego_treningu`. Na stronie powinien zostać wyświetlony komunikat, taki jak ten przedstawiony na rysunku 5.8.

Rysunek 5.8. Komunikat informujący o problemach z wczytaniem danych treningu

Jak widać, implementacja obsługi błędów tego typu była całkiem prosta. Zadeklarowaliśmy jedynie dwie właściwości modelu, `routeError` i `routeHasError`, z których pierwsza jest używana do przechowywania komunikatu o błędzie, a druga informuje, czy w aktualnie wybranej trasie wystąpił błąd, czy nie.

W procedurach obsługi zdarzeń `$routeChangeSuccess` i `$routeChangeError` określamy wartości tych właściwości, aby uzyskać zamierzone efekty. W implementacji procedury obsługi zdarzeń `$routeChangeError` wykonujemy dodatkowy test, aby się upewnić, że komunikat o błędzie zostanie wyświetlony tylko w przypadku, gdy nie uda się odnaleźć treningu. Warto także zwrócić uwagę na właściwość `routeErrorMessage`, którą dodaliśmy do danych konfiguracyjnych trasy. Podobna modyfikacja danych konfiguracyjnych tras została wprowadzona w poprzednim rozdziale, w celu określania elementów nawigacyjnych aktualnie wyświetlonego widoku.

W ten sposób zadbaliśmy o obsługę błędów związanych z wyświetlaniem nieistniejących treningów na stronie *Konstruktora treningów*, ale analogiczne zmiany należy też wprowadzić na stronie do edycji ćwiczeń. Zadanie to potraktujemy jako ćwiczenie do samodzielnego wykonania, a kod aplikacji po jego zaimplementowaniu można znaleźć w przykładach dołączonych do książki.

> Kod aplikacji na obecnym etapie jej rozwoju można znaleźć w przykładach dołączonych do książki, w katalogu *rozdzial05\punkt_kontrolny_6*.

Kolejną ważną zmianą, którą musimy zaimplementować, jest poprawienie aplikacji *7-minutowy trening*, gdyż aktualnie pozwala ona na przeprowadzanie tylko jednego, konkretnego treningu.

Kup ksi k Pole ksi k

Poprawianie aplikacji 7-minutowy trening

W obecnej postaci aplikacja *7-minutowy trening* (czy też *Treningomat*) jest w stanie prezentować tylko jeden, ściśle określony trening. Musimy ją zatem poprawić w taki sposób, by zapewniała możliwość wykonywania dowolnego planu treningowego zbudowanego przy użyciu aplikacji *Mój trening*. Oczywistym jest, że te dwa programy trzeba ze sobą zintegrować. Podstawy do zapewnienia tej integracji zostały już zrobione. Dysponujemy już wspólnymi usługami zapewniającymi dostęp do modeli aplikacji oraz usługą `WorkoutService` przeznaczoną do wczytywania danych — to nam w zupełności wystarczy do rozpoczęcia prac.

Zmodyfikowanie aplikacji *7-minutowy trening* i przekształcenie jej w aplikację *Treningomat* wymaga czterech, opisanych poniżej kroków:

1. Usunięcie z kontrolera aplikacji *7-minutowy trening* zakodowanych w nim na stałe ćwiczeń i treningu.
2. Poprawienie strony początkowej w taki sposób, by pokazywała listę dostępnych treningów, i zapewnienie użytkownikowi możliwości wybrania jednego z nich, który chce wykonać.
3. Poprawienie konfiguracji trasy dla strony treningu w taki sposób, aby w formie parametru była w niej przekazywana nazwa wybranego treningu.
4. Wczytanie wybranego treningu przy użyciu usługi `WorkoutService` i rozpoczęcie jego wykonywania.

Dodatkowo należy także zmienić nazwę aplikacji, gdyż dotychczasowa — *7-minutowy trening* — jest już nieaktualna. Chyba najbardziej odpowiednia będzie teraz nazwa *Mój trening*. A zatem, przy okazji, powinniśmy usunąć ze wszystkich widoków wcześniejszą nazwę — *7-minutowy trening*.

Wprowadzenie tych zmian będzie doskonałym ćwiczeniem do samodzielnego wykonania! I właśnie dlatego nie opiszę tu sposobu jego implementacji. Spróbuj sam wprowadzić wszystkie zmiany, a następnie porównaj je z naszą implementacją, którą znajdziesz w przykładach dołączonych do książki, w katalogu *rozdzial05\punkt_kontrolny_7*.

Tak oto dotarliśmy do końca rozdziału, warto by zatem podsumować wszystko, czego się z niego dowiedziałeś.

Podsumowanie

Obecnie dysponujemy już aplikacją o całkiem sporych możliwościach. Możemy wykonywać treningi, wczytywać je i aktualizować, możemy także śledzić historię wykonywanych treningów. A jeśli się nieco cofniemy i przyjrzymy wprowadzonym modyfikacjom, to dojdziemy do wniosku, że udało się nam to osiągnąć, wprowadzając minimalne zmiany w kodzie. Mogę się założyć, że

Kup ksi k		Pole ksi k

próba wprowadzania analogicznych zmian w aplikacji pisanej z użyciem biblioteki jQuery lub innego frameworka JavaScriptu wymagałoby wprowadzenia znacznie większych i poważniejszych zmian.

Niniejszy rozdział zaczęliśmy od utworzenia bazy danych MongoDB na serwerach serwisu MongoLab. Ponieważ MongoLab zapewnia dostęp do baz danych przy użyciu API typu RESTful, zaoszczędziliśmy trochę czasu, rezygnując z konfiguracji własnej infrastruktury serwera MongoDB.

Pierwszą konstrukcją AngularJS przedstawioną w tym rozdziale była usługa `$http`. Stanowi ona podstawowy mechanizm zapewniający komunikację z serwerami przy użyciu protokołu HTTP.

Poznałeś także obiekt konfiguracyjny (`config`) usługi `$http` i zobaczyłeś, jak się go używa do konfigurowania żądań HTTP. Oprócz tego dowiedziałeś się, w jaki sposób standardowa konfiguracja usługi `$http` pozwala na bardzo proste korzystanie z udostępnianych na serwerach punktów końcowych, które pobierają i zwracają dane w formacie JSON.

Oprócz tego przekonałeś się, że cała infrastruktura usługi `$http` bazuje na *obietnicach* i wywołaniach *funkcji zwrotnych* i ma całkowicie asynchroniczny charakter.

W tym rozdziale po raz pierwszy utworzyłeś własną obietnicę, dowiedziałeś się, jak można je tworzyć i wyznaczać.

W ramach prezentowania tych zagadnień poprawiliśmy także aplikację *Mój trening* w taki sposób, by korzystała z usługi `$http` do wczytywania i zapisywania danych ćwiczeń i treningów. Przy tej okazji poznałeś problemy związane z korzystaniem ze zdalnych zasobów dostępnych na innych domenach. Dowiedziałeś się również, czym jest JSONP (sposób umożliwiający obejście stosowanej przez przeglądarki zasady *tego samego pochodzenia*) i jak można wykonywać żądania JSONP w aplikacjach AngularJS. Wspomniałem też o CORS, nowym standardzie komunikacji pomiędzy różnymi domenami.

Kolejnym zagadnieniem opisanym w tym rozdziale była usługa `$resource`. Jej podstawowym przeznaczeniem jest ułatwienie korzystania z usług typu *RESTful*. W ramach jej poznawania zmodyfikowaliśmy kod wczytujący i zapisujący dane ćwiczeń na serwerze, zastępując używaną w nim wcześniej usługę `$http` usługą `$resource`. Wprowadzenie tych zmian pozwoliło skrócić kod i ograniczyć liczbę używanych w nim funkcji zwrotnych.

Następnie skoncentrowaliśmy się na kilku często stosowanych rozwiązaniach związanych z odwołaniami do zdalnych serwerów. Dowiedziałeś się, czym są *funkcje przechwytujące*, używane przez AngularJS do manipulowania żądaniami i odpowiedziami HTTP na poziomie globalnym. Stworzyliśmy także własną funkcję przechwytującą, której zadaniem było dodawanie klucza API do każdego żądania kierowanego do serwisu MongoLab.

Kup ksi k

Pole ksi k

Kolejnym opisanym zagadnieniem były funkcje przekształcające, które pozwalają przechwytywać zawartość żądań i odpowiedzi i wykonywać na nich operacje. Przekonaliśmy się także, w jaki sposób AngularJS używa tych funkcji przekształcających do obsługi danych w formacie JSON, które są przez framework automatycznie serializowane i deserializowane.

Ostatnim z zagadnień przedstawionych w tym rozdziale była obsługa błędów związanych z wyznaczaniem tras, które mogą się pojawiać, gdy funkcja używana w parametrze konfiguracyjnym `resolve` trasy zwraca obietnicę.

W ten sposób poznałeś większość elementów konstrukcyjnych używanych do tworzenia aplikacji AngularJS, oprócz jednego, i to niezwykle ważnego: dyrektyw. Używaliśmy ich wszędzie, lecz sami jeszcze żadnej nie napisaliśmy. Dlatego kolejny rozdział zostanie w całości poświęcony zagadnieniu dyrektyw. Zaimplementujemy w nim kilka niewielkich dyrektyw, takich jak zdalny walidator, przycisk AJAX oraz dyrektywa wyświetlająca podpowiedzi dotyczące walidacji danych, z której skorzystamy w aplikacji *Konstruktor treningów*. Ponadto dowiesz się, jak można zintegrować wtyczkę jQuery jako dyrektywę AngularJS.

Kup ksi k Pole ksi k

Skorowidz

A

adnotacja
  bezpośrednia, 58
  zależności, 58
adres URL, 81, 409
AJAX, 311
akcje zasobów, 263
aktualizacja modelu, 191, 217, 301
analiza danych, 166
animacje, 100, 133, 136
  CSS, 134
API, 70
aplikacja
  7-minutowy trening, 45
  Mój trening, 165
  Treningomat, 328, 382
aplikacje jednostronicowe, 75, 97, 420, 421
aplikacje wielojęzyczne, 388, 392, 393
argument
  path, 78
  routeConfig, 78
asynchroniczna funkcja, 300
atrapa serwera, 379
atrybut, 288
  remove-validator, 294
  submitting, 324
automatyzacja, 338
autoryzacja, 398, 405

B

baza danych, 232
  MongoDB, 231
biblioteka, 19
  angular-translate, 394
  jQuery, 105, 290, 325
biblioteki dodatkowe, 426
bieżący obiekt zasięgu, 30
błąd 404, 73

C

cookies, 399
CORS, 258
CSS, 121
cykl
  przeglądu, 113, 115
  życia dyrektywy, 304
czujka, watch, 66, 208, 331, 415

D

dane
  modelu, 64
  o treningach, 151
  tylko do odczytu, 415
debugowanie testów jednostkowych, 353
dedykowany język, 288
definicja
  modelu aplikacji, 165
  widoku konstruktora, 166
DI, dependency injection, 39, 337
dodatek Batarang, 425
dodawanie
  animacji, 138
  ćwiczeń, 181
  modułów aplikacji, 53
  paneli, 82
  stron pomocniczych, 166
usługa WorkoutBuilderService, 179
dodawanie widoku historii, 149
dokumentacja, 42

Kup ksi k

Pole ksi k

DOM, 40
dostawca $routeProvider, 78
dostęp do
  danych, 265
  tras, 406
dowiązania, 331
  danych, 32, 34
  danych dwukierunkowe, 34
  danych jednokierunkowe, 33
  właściwości obiektów zasięgu, 193
drzewo DOM, 308
DSL, domain-specific language, 288
dyrektywa, 28, 283, 284
  ajax-button, 323, 324
  busy-indicator, 313, 314
  form, 211
  media-player, 111
  ng-app, 29
  ng-bind-html, 102
  ng-class, 122
  ng-click, 29, 289, 291, 320
  ng-controller, 30, 79
  ng-enter, 136
  ng-enter-active, 136
  ng-form, 220
  ng-if, 46, 95
  ng-include, 85, 86, 87
  ng-leave, 136
  ng-leave-active, 136
  ng-messages, 205, 206, 207
  ng-model, 29, 166, 188, 192
  ng-model-options, 191
  ng-move, 136
  ng-move-active, 136
  ng-options, 189
  ng-repeat, 87, 88, 159, 162, 417
  ng-show, 29, 73
  ng-src, 73
  ng-style, 74
  ng-view, 76
  owl-carousel, 331, 333
  popover, 322
  przycisku, 320
  remote-validator, 293, 299, 310, 312, 364
  translate, 397
  walidacyjna, 310
  wizualnego wskaźnika, 307
  workout-tile, 285
  workout-title, 368
dyrektywy
  cykl życia, 304
  funkcja kontrolera, 310
  izolowany zasięg, 317
  komunikacja, 411
  szablony, 315
  tworzenie, 285
  właściwość require, 296
  zdalna walidacja nazwy, 292
dziedziczenie
  przez prototyp, 225
  zasięgów, 166, 221
dzielenie aplikacji, 421

E

E2E, end-to-end, 337, 339, 374
edycja treningów, 177
efekt
  opóźnienia, 192
  zastosowania dowiązań, 33
ekosystem testowy, 340
element span, 109
elementy, 288

F

faza konfiguracji/uruchamiania, 77
fazy konsolidacji, 306
filtr, 31, 49, 90
  date, 91
  filter, 91, 159
  lowercase, 91
  number, 91
  orderBy, 161
  secondsToTime, 92
  translate, 396
  upperacase, 91
filtrowanie historii treningów, 158
format godzinowy, 90
formatowanie, 166
  instrukcji, 101
formaty wyrażeń, 88
formularz edycji treningów, 201
formularze, 166, 186
  generowane dynamicznie, 219
framework
  Bootstrap, 74
  filtry, 393

Kup ksi k
Pole ksi k

funkcja
    $addControl, 212
    $digest(), 113
    $modal.open, 131
    $removeControl, 213
    $scope.$apply, 419
    $scope.$on, 152
    $setDirty(), 213
    $setPristine(), 213
    $setUntouched(), 213
    $setValidity, 209
    $watch, 68
    asynchroniczne, 300
    catch, 70
    compile, 290, 302, 314
    directive, 285
    errorCallback, 70
    factory, 144
    fillImages, 329
    finally, 70
    function, 161
    kontrolera dyrektywy, 310
    link, 297, 302
    ngModelCtrl.$setViewValue, 302
    notifyCallback, 70
    module, 347
    pauseResumeToggle(), 125
    reset, 218
    resourceUrlWhiteList, 85
    scope.$apply, 292
    successCallback, 70
    startWorkout, 93, 148
    then, 295, 298
    trackWorkoutUpdate, 155
    when, 78, 250
funkcje
    analizujące, 196
    formatujące, 196
    konsolidujące, 298, 306
    nasłuchujące, 68
    przechwytujące, 403
    przekształcające żądania, 275
    zwrotne, 138
        error, 234
        success, 234

G

generatory, 390
generowanie danych wejściowych, 166
gra Odgadnij liczbę!, 21

H

hierarchia
    elementów HTML, 409
    obiektów zasięgu, 317
    zasięgów, 89, 409
historia treningów, 100, 151, 156

I

IDE, 41
ignorowanie problemów, 31
implementacja, 63
    $interval, 357
    $timeout, 357
    filtru, 46
        secondsToTime, 92
    kontrolera, 46, 59
        WorkoutAudioController, 108
        WorkoutDetailController, 181, 185
    list treningów, 174
    niestandardowych walidatorów, 209
    obsługi audio, 106
    usługi, 147
    warstw, 121
        z przyciskami, 120
    wstrzymywania treningu, 118
    zapisu historii, 146
    zmiany ćwiczenia, 65, 68
informacje
    o ćwiczeniach, 82
    o dyrektywach, 283
    o obietnicach, 69
    o sesji, 399
    o treningach, 248, 251
    o usłudze $http, 233
    o usłudze $resource, 261
    o usługach, 141
    o wstrzykiwaniu zależności, 56
inicjalizacja
    aplikacji, 39
    modułu, 77
instrukcje wykonywania ćwiczenia, 101

Kup ksi k   Pole ksi k

integracja
  AngularJS i jQuery, 325
  z serwerem, 236
interakcje z serwerem, 230
interceptor żądań i odpowiedzi, 271
interfejs
  API, 70, 212
  użytkownika, 21
izolowany zasięg dyrektyw, 317

J

jQuery, 325
JSONP, 257

K

Karma, 342
katalogi, 50
klasa Resource, 269
klasy, 288
  CSS, 200
  końcowe, 136
  początkowe, 136
klip, 111
  audio, 105
  wideo, 133
klucz API, 273
kod
  HTML, 315, 317
  JavaScriptu, 370
  kontrolera, 222, 223
komentarze, 288
kompilacja, 314
komunikacja między dyrektywami, 283, 310, 411
komunikat
  o błędach walidacji, 214
  o błędzie, 205, 216, 296
  o zapisie formularza, 214
konfiguracja, config, 77
  narzędzia Protractor, 376
  środowiska Karma, 342
  zależności kontrolera, 348
konsolidacja, 306
konstrukcje
  formularzy, 187
  podstawowe aplikacji, 49
kontekst, 31
  uwierzytelniania użytkownika, 406
kontrola
  aktualizacji modelu, 191
  jakości, 339
kontroler, controller, 20, 24, 35
  aplikacji, 54
  dyrektywy, 311
  FormController, 212
  NgModelController, 194
  WorkoutAudioController, 108
  WorkoutController, 118, 329, 352, 355
  WorkoutDetailController, 181, 185, 208
kontrolery list ćwiczeń, 176
konwertowanie
  liczb rzeczywistych, 196
  sekund, 90

L

leniwe wczytywanie, 420
  modułów, 422
licznik, 94
lista treningów, 174, 381
literały łańcuchowe
  dynamiczne, 395
  stałe, 395
lokalizacja zdalna, 86
lokalizowanie, 392

Ł

łańcuch
  obietnic, 71, 243, 244
  znaków, 122
łączenie widoku z obiektem zasięgu, 40

M

magazyn
  danych, 231
  przeglądarki, 157
menedżer zależności Bower, 344
metoda
  $broadcast, 154
  $emit, 153
  $modal.open, 128
  broadcast, 153
  otherwise, 78
  uniqueUserName, 294
metody nasłuchujące, 115

Kup ksi k

Pole ksi k

minimalizacja
  czujek, 417
  kodu, 58
modalne okno dialogowe, 127
model, model, 20, 326
  aplikacji, 23, 51
  aplikacji Mój trening, 168
  jako usługa, 169
moduł, module, 26
  angular-dynamic-locale, 394
  aplikacji, 39
  http-server, 22
  ngMock, 347
  ui.bootstrap, 128
moduły aplikacji, 53, 423
modyfikacje kodu HTML, 316
MongoDB, 231
MVC, Model View Controller, 19

N

narzędzia
  diagnostyczne, 28
  do testowania, 340
  Yeoman, 390
narzędzie
  Batarang, 41
  Bower, 344
  jsFiddle, 41
  Karma, 342
  Konsola przeglądarki, 41
  Plunker, 41
  Protractor, 374
nawias klamrowy, 28
nazwa klucza błędu, 298
nazwy usług, 143
niepożądane aktualizacje, 217
Node Package Manager, 342
Node.js, 22
normalizacja, 287
notacja obiektowa, 168

O

obiekt, 122
  $error, 214
  $rootScope, 40
  $scope, 28, 35, 61
  filter, 160
  konfiguracyjny dyrektywy, 285
  resolve, 183
obiekty
  stron, 382
  zasięgu, 221, 409
  zasobu, 269
obietnica, promise, 69, 295
obsługa
  audio, 99, 104, 106
  autoryzacji, 398, 405
  błędów, 277
  dużych zbiorów danych, 417
  formularzy, 186
  nieodnalezionych treningów, 278
  scenariuszy, 387
  tłumaczenia tekstów, 394
  uwierzytelniania, 398
  zdarzeń, 152, 154
odtwarzanie
  dźwięku, 107
  wideo, 127
odwołania do domen, 256
odwzorowywanie danych, 241
ograniczenia wyrażeń, 31
okno dialogowe, 127
określanie definicji serwera, 360
operacje na
  ćwiczeniach, 252
  stylach CSS, 122
optymalizacja czujek, 415
organizacja kodu, 48–51

P

panel
  opisu, 82
  wideo, 82, 100, 125
parametr
  $modalInstance, 132
  attr, 297
  comparator, 160
  controller, 129
  ctrls, 297
  element, 297
  expression, 161
  resolve, 130
  scope, 129, 297
  size, 130
  templateUrl, 129

parametry funkcji $watch, 66
pasek
  nawigacyjny, 171, 178
  postępu, 74
pętla
  $evalAsync, 113, 114
  listy $watch, 114
pierwsza aplikacja, 45
plik
  config.spec.js, 371, 372
  directives.js, 295, 299, 308
  exercises.js, 176
  karma.config.js, 369
  services.spec.js, 362
  workout.js, 176, 215
  workout-tile.html, 286
pobieranie modułu, 55
poprawianie
  aplikacji, 280
  komunikatów, 214
poprawność
  danych, 198
  formularzy, 211
potok
  analizy, 194
  formatowania, 194
predykat, 160
preprocesor, 369
prezentacja $watch, 66
priorytet, 306
proces sprawdzania poprawności danych, 205
programowanie w oparciu o testy, 339
projektowanie
  modelu aplikacji, 23
  widoku, 46
projekty startowe, 388
protokół OAuth 2.0, 401
przebieg uwierzytelniania, 399
przechowywanie danych, 157, 229
  w pamięci, 424
  zdalnych, 423
przechwytywanie żądań, 271
przeglądanie historii, 100
przejście, 135
przekazywanie
  danych, 409
  zdarzeń, 153
przycisk, 320
  do wstrzymywania, 121
pseudoselektor, 134
punkt końcowy, 270

R

refaktoryzacja
  kontrolera, 126
  panelu wideo, 126
repozytorium danych treningów, 175
rodzaje testów, 339
role, 407
rozbudowa panelu wideo, 125
rozmiar strony, 415
rozszerzenia IDE, 41
rozszerzenie Batarang, 37, 425

S

scenariusze, 387
  obsługi formularzy, 166
selektor, 326
Selenium WebDriver, 374
serwer
  Selenium, 374
  WWW, 21
serwis
  GitHub, 27
  jsFiddle, 67, 393
singleton, 141
skrypty osadzone, 86
sortowanie, 159
sprawdzanie
  poprawności danych, 166, 198, 202–205
  poprawności formularzy, 211
  zależności, 355
  zawartości obiektów zasięgu, 37
  zawartości tablicy, 112
  zmian, 113, 114
stan
  formularza, 166, 218
  modelu, 199
  pola, 166
sterowanie przepływem, 31
stosowanie
  adresów URL, 409
  animacji, 135
  dyrektywy, 283
    ng-include, 85
    ng-model, 188

Kup ksi k
Pole ksi k

ng-repeat, 87
ng-show, 73
ng-style, 74
JSONP, 257
obiektów stron, 382
obiektów zasięgu, 409
szpiegów Jasmine, 355
transkluzji, 316
usług, 411
$location, 81
$resource, 270
zasięgu $rootScope, 411
zdarzeń, 412
strona
końcowa, 75
początkowa, 75
strony pomocnicze, 166
struktura
katalogów, 48
kodu HTML, 317
modelu, 168
style CSS, 121
symbole wstawki, 28
synchronizacja
elementu widoku, 302
klipów audio, 111, 116
system nawigacyjny aplikacji, 171
szablon
widoku, 109
dyrektywy, 315
szpieg Jasmine, 355

Ś

ścieżka do obrazka, 73
ścisłe zezwalanie kontekstowe, 101
śledzenie historii, 155
środowisko
Karma, 342
Protractor, 375

T

tablica, 122
TDD, test-driven development, 339
testowanie, 338
aplikacji, 337
aplikacji Treningomat, 382
implementacji $interval, 357
jednoczesne dyrektyw, 366
usługi WorkoutService, 361
współdziałania dyrektyw, 368
wznawiania treningu, 358
zatrzymywania treningu, 358
testy
dyrektywy remote-validator, 364
E2E, 338, 373–376, 381
jednostkowe, 339, 338, 341
dyrektyw, 363
komponentów, 345
kontrolera WorkoutController, 352, 355
kontrolerów, 348
tras, 371
usług, 359
z użyciem modułu ngMock, 347
tłumaczenie tekstów, 394
transkluzja, 315
trasowanie, 426
trasy, 76, 79
aplikacji Mój trening, 171
trening, 53, 117, 177
trwałość danych, 229
tunelowanie zdarzeń, 332
tworzenie
aplikacji, 22, 165, 388
jednostronicowej, 46
wielojęzycznych, 392
atrapy serwera, 379
bezpłatnej bazy danych, 232
dyrektyw, 283, 292, 334
filtra, 90
formularzy, 166
łańcucha obietnic, 71, 245
modułu, 55
obietnic, 230, 249
paska nawigacyjnego, 178
testów
jednostkowych, 341
typu E2E, 373
treningów, 177, 185, 253
usług, 142–144
widoków, 76

U

układ aplikacji Mój trening, 170
ukrywanie treści, 416
uruchamianie, run, 78

Kup ksi k
Pole ksi k

usługa, 49, 141, 146, 411
$compile, 308
$http, 229, 233, 424
$interval, 62, 420
$location, 80, 81
$modal, 141
$parse, 291, 298
$q, 250
$resource, 230, 259–270
$route, 76
$routeParams, 219
$templateCache, 86, 368
$timeout, 420
$translate, 396
factory, 144
jsFiddle, 88
MongoLab, 231
scope.$evalAsync, 333
sessionContext, 406
WorkoutBuilderService, 179
WorkoutHistoryTracker, 146, 148
WorkoutService, 239, 361
usługi
typu constant, 142, 143
typu provider, 144
typu RESTful, 259
typu service, 143
typu value, 142
usprawnienie panelu wideo, 100
ustawienia regionalne, 394
uwierzytelnianie, 388, 398
w oparciu o cookies, 399
w oparciu o żetony, 401, 405
użycie
API obietnic, 69
dyrektywy
ng-bind-html, 103
ng-form, 220
ng-if, 94
ng-model-options, 191
ng-repeat, 159
filtrów, 90
klawiatury, 124
magazynu przeglądarki, 157
obietnic, 230
usługi $resource, 265
wielu dyrektyw, 306
zdarzeń, 155

W

walidacja, 205, 214, 313
nazwy treningu, 292
walidator niestandardowy, 207
warstwy z przyciskami, 120
warunkowe wyświetlanie treści, 407, 416
wczytywanie danych, 236
weryfikacja
formularza, 201
implementacji, 63
widok, view, 20, 326
aplikacji, 25, 72
do tworzenia treningów, 185
widoki
cząstkowe, 46, 79
list ćwiczeń, 176
właściwości dyrektywy ng-repeat, 162
właściwość
$asyncValidators, 299
$error, 213
$parent, 226
animation, 135
busy, 308
close, 131
dismiss, 131
onClick, 320
opened, 132
priority, 302, 307
replace, 322
require, 296, 311
resolve, 371
result, 131
scope, 308
submitting, 320, 324
templateUrl, 368
wskaźnik zdalnej walidacji, 307
współdzielenie
danych, 408
zasobów pomiędzy domenami, 401
współużytkowanie modelu treningu, 168
wstawki, 28
wstrzykiwanie
kodu HTML, 314
przez właściwości, 56
zależności, DI, 39, 45, 55, 57, 337
usługi, 148
wstrzymywanie treningu, 117, 120, 124

Kup ksi k Pole ksi k

wtyczka
  Batarang, 89
  Owl Carousel, 328
wtyczki biblioteki jQuery, 327
wydajność, 388, 413, 414
wykrywanie zmian, 112
wymagania aplikacji, 165, 167
wyrażenia, 28, 30
wyświetlanie instrukcji, 103
wytyczne dotyczące wydajności, 414
wywoływanie akcji, 263
wyznaczanie tras, 182, 251, 277
  nieodnalezionych, 184
wznawianie treningu, 120, 124, 358
wzorce
  architektoniczne, 19
  komunikacji, 388, 408
  współdzielenia danych, 408
wzorzec MVC, 20

X

XSS, cross-site scripting, 102

Z

zapis historii treningów, 141, 146, 210
zarządzanie
  pakietami, 342
  testami E2E, 382
  zależnościami, 344
zasięg, 30, 36, 89, 146, 313
  $rootScope, 411
  główny, 38
  izolowany, 130, 317, 319, 322
  nadrzędny, 37, 319
  pokrewny, 317
  potomny, 37
zastosowanie funkcji
  analizującej, 197
  formatującej, 197
zatrzymywanie
  klipów, 123
  treningu, 358
zdalna walidacja danych, 302
zdarzenia, 152, 412
  myszy, 120, 419
zdarzenie
  afterAction, 333
  blur, 300, 301
zintegrowane środowisko programistyczne, IDE, 41
zmiany w modelu, 66
znacznik
  <body>, 25

, 101
  <div>, 29
  <img>, 73
  input, 301
znaczniki dyrektywy, 287
znak kropki, 104

Ż

żądania, 360
żądanie AJAX, 311
żeton, 401

Kup książkę Poleć książkę

Kup ksi k

Pole ksi k

PROGRAM
PARTNERSKI
GRUPY WYDAWNICZEJ HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW
w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA WYDAWNICZA
Helion SA

AngularJS
Praktyczne przykłady

AngularJS to szkielet, który pozwalał tworzyć zaawansowane aplikacje działające w środowisku przeglądarki internetowej. Daje nam do dyspozycji wiele narzędzi, które wcześniej można było zastosować tylko w „tradycyjnych" językach. Wśród tych narzędzi są zarówno wzorzec Model-View-Controller (MVC), jak i rozbudowane możliwości testowania. To właśnie dzięki AngularJS możesz w pełni wykorzystać potencjał współczesnych aplikacji internetowych!

Wszyscy programiści od dawna wiedzą, że najlepszym sposobem na nauczenie się nowych narzędzi, języków programowania czy bibliotek jest korzystanie z gotowych przykładów. Właśnie takie podejście do AngularJS dominuje w tej książce. Dzięki niej błyskawicznie opanujesz wszystkie aspekty używania tego szkieletu w codziennej pracy. W kolejnych rozdziałach znajdziesz praktyczne przykłady zastosowania wzorca MVC, wiązania danych, używania gotowych dyrektyw oraz usług. Dowiesz się, jak podłączyć się do zasobów serwera dzięki $http, oraz utrwalisz dane użytkownika poprzez wprowadzenie warstwy trwałości danych. Na sam koniec zapoznasz się z metodami tworzenia własnych dyrektyw i testowania poszczególnych komponentów aplikacji oraz ze wskazówkami na temat wydajności Twojej aplikacji. Sięgnij po tę książkę, jeżeli chcesz błyskawicznie poznać i wykorzystać w praktyce szkielet AngularJS!

Najszybszy sposób na naukę AngularJS!

Dzięki tej książce:

- zaznajomisz się ze wzorcem MVC
- zastosujesz gotowe dyrektywy, a następnie przygotujesz własne
- wykorzystasz dostępne usługi
- przetestujesz swoją aplikację
- poznasz dobre rady na temat wydajności
- wykorzystasz szkielet AngularJS w praktyce

Chandermani — programista specjalizujący się w tworzeniu aplikacji internetowych. Ma ponad 10-letnie doświadczenie w pisaniu aplikacji o różnym stopniu skomplikowania na podstawie rozwiązań firmy Microsoft. Zakochany w AngularJS od pierwszego spotkania. Lider zespołu programistów w Technovert.