

Valerio De Sanctis

ASP.NET Core 2 i Angular 5

Przewodnik
dla Full-Stack
Web Developera

Helion 

Packt 

Tytuł oryginału: ASP.NET Core 2 and Angular 5:
Full-Stack Web Development with .NET Core and Angular

Tłumaczenie: Rafał Jońca

ISBN: 978-83-283-4643-7

Copyright © Packt Publishing 2017. First published in the English language under the title 'ASP.NET Core 2 and Angular 5 – (9781788293600)'

Polish edition copyright © 2018 by Helion SA
All rights reserved.

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

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

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

Autor oraz HELION SA 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.

HELION SA
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/asp2an.zip>

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/asp2an>
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	13
O redaktorach merytorycznych	15
Wstęp	19
Rozdział 1. Przygotowanie środowiska	25
Dwóch graczy, jeden cel	26
Rewolucja ASP.NET	26
Co nowego w Angularze?	27
Podejście od początku do końca	28
Aplikacja typu SPA	29
Typowe funkcjonalności nowoczesnych aplikacji SPA	29
Oczekiwania właściciela produktu	30
Projekt przykładowej aplikacji SPA	32
Wizja	33
To nie będzie typowa aplikacja „Witaj, świecie”	33
Aplikacja TestMakerFree	33
Podstawowe funkcjonalności i wymagania	34
Przygotowanie środowiska pracy	35
Uwaga — pomyśl, zanim to zrobisz	36
Mit niedziałającego kodu	36
Pozostań otwarty na nowości, ale wprowadzaj je odpowiedzialnie	37
Wersje narzędzi i bibliotek	38
Konfiguracja projektu	38
Alternatywna konfiguracja z wykorzystaniem wiersza poleceń	42
Test konfiguracji środowiska	42

Poznanwanie projektu	44
Pliki konfiguracyjne	45
Plik Program.cs	45
Plik Startup.cs	47
Plik appsettings.json	50
Plik package.json	51
Plik tsconfig.json	54
Pliki konfiguracyjne narzędzia Webpack	56
Kod po stronie serwerowej	61
Plik Controllers/HomeController.cs	61
Plik Controllers/SampleDataController.cs	61
Folder /Views/	62
Kod po stronie klienckiej	63
Folder /ClientApp/app/	64
Przygotowanie projektu	66
Pamięć podręczna i pliki statyczne	66
Mechanizm stosowany w przeszłości	67
Powrót do przyszłości	67
Czas na test	70
Czyszczenie aplikacji klienckiej	71
Ograniczenie liczby komponentów	72
Klasy AppModule	74
Aktualizacja NavMenu	75
Odnosiniki	76
Poruszana tematyka	77
Podsumowanie	77
Rozdział 2. Część serwerowa wykorzystująca .NET Core	79
Przepływ danych	79
Zadania modelu widoku	82
Pierwszy model widoku	82
Klasa QuizViewModel	83
Klasa QuizController	84
Dodatkowe metody akcji	86
Sprawdzenie, czy wszystko działa	88
Dodawanie pozostałych kontrolerów	89
Klasa QuestionViewModel	89
Klasa QuestionController	90
Klasa AnswerViewModel	91
Klasa AnswerController	92
Klasa ResultViewModel	93
Klasa ResultController	94
Działanie routingu	96
Definiowanie routingu	96
Routing dawniej i dziś	96
Obsługa routingu w .NET Core	98

Dodawanie nowych ścieżek	99
Atrapa dostawcy danych	103
Obsługa pojedynczych elementów	103
Poruszana tematyka	107
Podsumowanie	107
Rozdział 3. Część kliencka korzystająca z frameworka Angular	109
Wzorzec nawigacji	110
Powiązanie ogół-szczegóły	111
Kliencka część interfejsu dotyczącego quizu	111
Komponent QuizListComponent	112
Nowa klasa HttpClient	114
Metoda onSelect()	117
Plik szablonu	117
Plik arkusza stylów	118
Dodanie komponentu	118
Test	120
Klasa QuizComponent	121
Dodanie plików komponentu	121
Dodanie komponentu	122
Test	123
Dołączanie dodatkowych list	124
Wiele instancji jednego komponentu	125
Testowanie i debugowanie	128
Interfejs OnInit i zdarzenia cyklu życia	129
Implementacja metody ngOnInit	131
Testowanie poprawionej wersji	133
Dwukierunkowe dowiązanie danych	133
Wyłączenie dwukierunkowego dowiązania danych	134
Routing po stronie klienckiej	135
Strategie PathLocationStrategy i HashLocationStrategy	136
Refaktoryzacja aplikacji	136
Rejestracja nowej ścieżki	138
Aktualizacja komponentu QuizComponent	139
Aktualizacja komponentu QuizListComponent	142
Test routingu	143
Dodanie nowych komponentów	143
Komponent AboutComponent	144
Komponent LoginComponent	145
Komponent PageNotFoundComponent	145
Aktualizacja klasy AppModule	146
Test całej aplikacji	148
Poruszana tematyka	151
Podsumowanie	151

Rozdział 4. Model danych wykorzystujący Entity Framework Core	153
Przygotowania	154
Instalacja Entity Framework Core	154
Podejścia do modelowania danych	156
Najpierw model	156
Najpierw baza danych	157
Najpierw kod	158
Podjęcie decyzji	159
Tworzenie encji	160
Klasa ApplicationUser	160
Klasa Quiz	161
Klasa Question	164
Klasa Answer	165
Klasa Result	166
Definiowanie relacji	168
Wzorzec leniwego wczytywania danych w relacjach jeden-do-wielu	170
Konfiguracja obiektu DbContext	170
Strategie inicjalizacji bazy danych	172
Wybór bazy danych	172
Aktualizacja pliku appsettings.json	173
Tworzenie bazy danych	174
Aktualizacja pliku Startup.cs	174
Dodanie migracji początkowej	174
Błąd braku pliku	176
Działanie mechanizmu migracji	177
Implementacja wypełniania bazy danymi	177
Utworzenie klasy DbSeeder	178
Użycie DbSeeder w Startup.cs	184
Wypełnienie bazy danymi początkowymi	186
Aktualizacja klasy QuizController	186
Narzędzie Mapster	187
Instalacja	187
Podstawy użycia	187
Aktualizacja klasy	188
Testowanie dostawcy danych	191
Poruszana tematyka	192
Podsumowanie	192
Rozdział 5. Interakcje po stronie klienckiej	195
Dodawanie, aktualizacja i usuwanie quizów	195
Aktualizacja klasy QuizController	196
Dostosowanie części klienckiej	200
Dodanie komponentu QuizEditController	200
Aktywacja trybu edycji	204
Implementacja funkcjonalności usuwania	206
Pierwszy test poważnej interakcji klienta z serwerem	207
Przepływ komunikacji między klientem i serwerem	211

Pytania, odpowiedzi i wyniki	212
Zadania po stronie serwerowej	213
Klasa QuestionController	213
Klasa AnswerController	217
Klasa ResultController	221
Klasa BaseApiController	225
Zadania po stronie klienckiej	228
Dodanie interfejsów	228
Komponent QuestionListComponent	228
Komponent QuestionEditComponent	234
Komponent AnswerListComponent	237
Komponent AnswerEditComponent	240
Komponent ResultListComponent	241
Komponent ResultEditComponent	243
Pełnowymiarowy test aplikacji	245
Poruszana tematyka	249
Podsumowanie	250
Rozdział 6. Arkusze stylów i układ interfejsu graficznego	251
<hr/>	
Czy jest aż tak źle?	251
Wprowadzenie do LESS	252
Języki arkuszy stylów	252
CSS	253
Przykładowy kod CSS	253
Czym jest LESS i dlaczego warto go używać?	254
Zmienne	255
Dyrektywy importu	256
Zagnieżdżanie selektorów	256
Domieszki (mixin)	257
Pseudoklasa :extend	258
Dokumentacja LESS	259
Sass, Stylus i inne możliwości	260
Implementacja LESS	260
Instalacja kompilatora LESS	261
Kompilacja plików LESS za pomocą narzędzia Webpack	263
Samodzielne definiowanie stylów kontra użycie frameworka CSS	264
Podejście „zróbmy wszystko sami”	265
Zalety	265
Wady	265
Podejście wykorzystujące framework CSS	265
Zalety	266
Wady	266
Wnioski	266
Praca z Bootstrapem	267
Zmiana motywu	267
Przebudowanie plików dystrybucyjnych zewnętrznych dostawców	268
Sprawdzenie nowego motywu	270

Zmiana struktury interfejsu użytkownika	271
Komponent AppComponent	271
Komponent NavMenuComponent	273
Komponent QuizSearchComponent	275
Plik SVG z logo	277
Szybki test	277
Stylowanie komponentów	278
Enkapsulacja CSS	278
Komponent HomeComponent	282
Komponent QuizListComponent	283
Komponent QuizComponent	289
Komponent QuizEditComponent	294
Komponenty pytania, odpowiedzi i wyniku	295
Całościowy test zmian interfejsu	298
Poruszana tematyka	299
Podsumowanie	299
Rozdział 7. Formularze i weryfikacja danych	301
Walidacja danych	302
Formularze we frameworku Angular	302
Formularze sterowane szablonami	302
Formularze sterowane modelem	304
Pierwszy reaktywny formularz	307
Dodanie referencji do ReactiveFormsModule	307
Uaktualnienie komponentu QuizEditComponent	308
Dodanie walidatorów	312
Uaktualnienie komponentów	315
Komponent QuestionEditComponent	315
Komponent AnswerEditComponent	316
Komponent ResultEditComponent	319
Debugowanie i testowanie	321
Jak wygląda model formularza?	321
Operator potoku	322
Reagowanie na zmiany	323
Obserwowanie obiektu Observable	323
Rozbudowa dziennika aktywności	326
Debugowanie po stronie klienta	326
Testy jednostkowe formularzy	327
Poruszana tematyka	328
Podsumowanie	328
Rozdział 8. Uwierzytelnianie i autoryzacja	329
Uwierzytelniać czy tego nie robić?	330
Uwierzytelnianie	330
Uwierzytelnianie przy udziale strony trzeciej	331

Autoryzacja	332
Autoryzacja przy udziale strony trzeciej	333
Rozwiązania własne czy firm trzecich?	334
Mechanizmy uwierzytelniania wbudowane w .NET Core	335
Konfiguracja .NET Core Identity	335
Konfiguracja usługi Identity	335
Klasa ApplicationUser jako klasa potomna	336
Uaktualnienie klasy DbContext	337
Modyfikacja klasy DbSeeder	338
Aktualizacja bazy danych	343
Dodanie migracji dotyczącej usługi Identity	343
Zastosowanie migracji	344
Opcja 1. — aktualizacja	344
Opcja 2. — usunięcie i ponowne utworzenie	345
Wypełnienie bazy danymi	345
Sposoby uwierzytelniania	346
Sesje	346
Tokeny	348
Sygnatury	349
Uwierzytelnianie dwuetapowe	349
Wnioski	349
Implementacja uwierzytelniania JWT	350
Dodanie usługi uwierzytelniania do klasy startowej	350
Aktualizacja plików AppSettings	352
Klasa TokenController	353
Aktualizacja klasy BaseApiController	354
Dodanie klasy TokenController	355
Klasa TokenRequestViewModel	358
Klasa TokenResponseViewModel	359
Test narzędziem Postman	359
Formularz logowania w Angularze	361
Interfejs TokenResponse	361
Klasa AuthService	361
Nowa wersja komponentu LoginComponent	365
Dodanie tokena do nagłówka żądania HTTP	373
Wymuszenie autoryzacji	376
Dostosowanie klienta	376
Komponent NavMenuComponent	377
Komponent QuizComponent	379
Ochrona serwera	380
Pobranie identyfikatora aktualnego użytkownika	380
Sprawdzenie uwierzytelniania na styku klient-serwer	381
Poruszana tematyka	382
Podsumowanie	383

Rozdział 9. Tematy zaawansowane	385
Wygasanie tokena i tokeny odświeżania	385
Czym jest token odświeżania?	386
Zadania po stronie serwerowej	387
Dodanie encji dla tokena	387
Implementacja tokena odświeżania	389
Zadania po stronie klienckiej	394
Aktualizacja interfejsu TokenResponse	395
Aktualizacja klasy AuthService	395
Dodanie klasy AuthResponseInterceptor	396
Test działania aplikacji	399
Rejestracja nowego użytkownika	400
Zadania po stronie serwerowej	400
Klasa UserController	400
Klasa UserModel	402
Zadania po stronie klienckiej	402
Interfejs User	403
Klasa RegisterComponent	403
Plik AppModule	406
Komponent LoginComponent	407
Komponent NavMenu	407
Test działania aplikacji	407
Uwierzytelnianie dzięki firmom trzecim	409
Działanie uwierzytelniania OAuth2	409
Mechanizm jawny czy niejawny?	410
Wnioski	412
Logowanie do Facebooka	412
Tworzenie aplikacji Facebooka	412
Mechanizm niejawny	415
Aktualizacja klasy TokenController	416
Dodanie komponentu LoginFacebookComponent	421
Test działania aplikacji	427
Mechanizm jawny	427
Instalacja pakietu Authentication.Facebook	428
Konfiguracja usługi uwierzytelniania poprzez Facebooka	428
Aktualizacja pliku appsettings.json	429
Aktualizacja klasy TokenController	430
Komponent LoginExternalProvider	435
Test działania aplikacji	438
Poruszana tematyka	439
Podsumowanie	439
Rozdział 10. Prace wykończeniowe i wdrożenie	441
Przejdźcie na SQL Server	441
Instalacja SQL Server 2017 Express Edition	442
Instalacja SQL Server Management Studio	442
Konfiguracja bazy danych	443

Dodanie konfiguracji połączenia z bazą SQL Server	446
Modyfikacja konfiguracji połączenia z bazą danych	447
Dodanie produkcyjnego adresu URL u zewnętrznych dostawców	447
Aktualizacja pliku launchSettings.json	448
Publikacja aplikacji internetowej	448
Tworzenie profilu publikacji	449
Publikacja poprzez protokół FTP	450
Profil publikacji do folderu	451
Publikacja aplikacji internetowej	451
Konfiguracja serwera i IIS	452
Instalacja modułu ASP.NET Core dla IIS	452
Dodanie nowej witryny	453
Konfiguracja puli aplikacji	455
Uruchamianie silnika	456
Analiza typowych błędów po wdrożeniu	457
Przeanalizowanie komunikatu w przeglądarce	457
Narzędzie Event Viewer	461
Moduł logowania w ASP.NET Core	461
Sprawdzenie serwera Kestrel	462
Wyłączenie renderowania po stronie serwera	464
Poruszana tematyka	464
Podsumowanie	465
Skorowidz	467

Część serwerowa wykorzystująca .NET Core

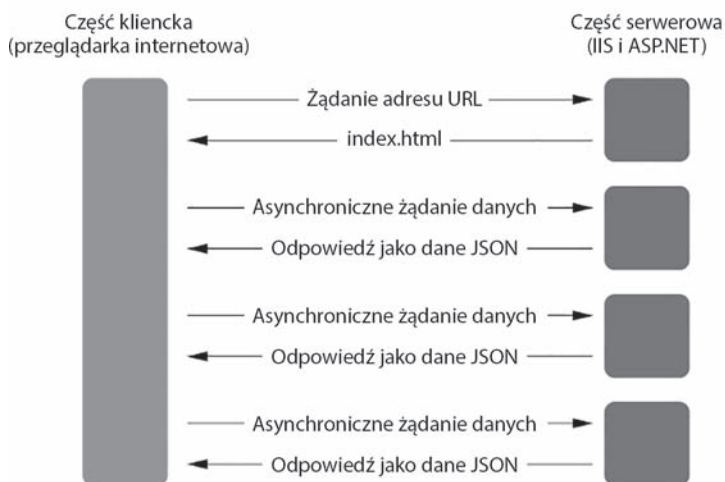
Skoro udało nam się uruchomić szkielet aplikacji, zacznijmy analizować mechanizm interakcji klient-serwer wykorzystywany przez oba frameworki. Innymi słowy, omówmy, w jaki sposób Angular będzie pobierał dane z .NET Core za pomocą całkowicie nowej struktury obsługującej jednocześnie wersje MVC i API.

W tej chwili nie będziemy zajmować się tym, w jaki sposób .NET Core otrzyma dane z obiektów sesyjnych, lokalnych plików, relacyjnych baz danych itp. Tym tematem zajmiemy się później. Na razie skupimy się na przykładowych statycznych danych, byś mógł lepiej zrozumieć, jak działa wzajemna komunikacja między Angularem i .NET Core. Wykorzystamy dobrze ustrukturyzowany, wysoce konfigurowalny i wygodny interfejs, który stosuje rozwiązania podobne do tych, jakie stosował przykładowy kontroler `SampleDataController` dołączony do szablonu SPA omówionego w rozdziale 1.

Przepływ danych

Jak zapewne wiesz, natywna aplikacja webowa działająca zgodnie z modelem SPA obsługuje komunikację między klientem i serwerem w następujący sposób:

Komunikacja w natywnej aplikacji webowej



W tworzonej aplikacji rolę pliku *index.html* przedstawionego na schemacie odgrywa widok */Views/Index.cshtml* zwracany przez akcję *Index* znajdującą się w kontrolerze *HomeController*. Ogólna zasada działania pozostaje jednak taka sama.

Jeśli zastanawiasz się, czym są asynchroniczne żądania danych, odpowiedź jest bardzo prosta — to dowolne treści, których pobranie wymaga komunikacji z serwerem. Często taka komunikacja jest wynikiem akcji użytkownika, np. kliknięcia przycisku, edycji danych w formularzu, kliknięcia łącza przejścia do innej strony, wysłania formularza itp. Jeżeli realizowane zadanie jest bardzo proste (lub wymaga minimalnej ilości danych), klient najczęściej potrafi obsłużyć je samodzielnie bez dodatkowej komunikacji. Takie proste zadania to na przykład ukrycie lub wyświetlenie części danych, przejście do innej części tej samej strony, walidacja danych wpisanych w formularzu przed ich wysłaniem itp.

Przedstawiony schemat pokazuje, co musimy zrobić — zdefiniować i zaimplementować wzorzec obsługi żądań bazujących na formacie JSON i przesyłać do klienta wymagane dane. Ponieważ zdecydowaliśmy się na aplikację wymagającą wielu różnych rodzajów informacji, z pewnością wykonamy kilka standardowych zestawów żądań typu **CRUD** związanych z obsługiwanyimi przez aplikację typami danych.

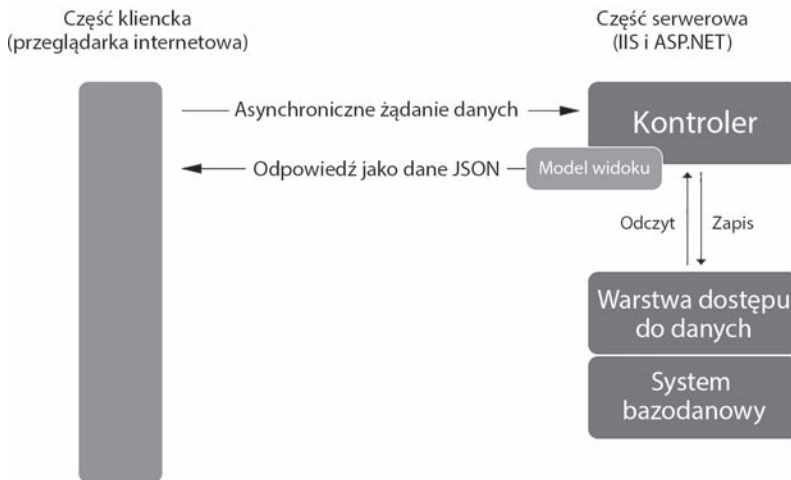
Dla osób, które jeszcze nie słyszały o **CRUD** — to skrót od *Create-Read-Update-Delete*, czyli czterech podstawowych funkcji związanych z trwałym zapisem danych. Skrót ten stał się popularny po tym, jak James Martin wspomniał o nim w książce *Managing the Database Environment* wydanej w 1983 roku. Obecnie pojawia się bardzo często w kontekście programowania API.

Gdy weźmiemy pod uwagę plan działania aplikacji opisany w rozdziale 1., możemy szybko zdefiniować podstawowe rodzaje treści, jakich będziemy wymagać. Mamy więc **quizey**, które będą stanowiły główny rodzaj treści prezentowanej w aplikacji. Quiz składa się z **pytań** (jednego lub większej liczby), a każde pytanie ma listę **odpowiedzi**. Z quizem powiązane są też **wyniki**. W pewnym momencie do aplikacji będziemy musieli też wprowadzić użytkowników, co pozwoli dodać mechanizm uwierzytelniania i autoryzacji, a tym samym definiować, kto ma prawo do przeglądania, edycji lub usunięcia określonych danych.

Dla każdego z wymienionych rodzajów danych przygotujemy odpowiedni zestaw żądań umożliwiających przeglądanie listy wpisów, edycję i pobieranie pojedynczego wpisu, a także usuwanie wybranego wpisu.

Zanim przejdziemy dalej, przyjrzyjmy się dokładniej pojedynczemu wysłanemu przez klienta cyklowi **żądania danych**, który powoduje przygotowanie przez serwer **odpowiedzi w formacie JSON**. Najczęściej taki cykl nazywamy skrótowo **cyklem żądanie-odpowieź**.

Cykl żądanie-odpowieź



Aby zareagować na dowolne **żądanie danych** wysłane przez klienta, musimy zdefiniować po stronie serwera **kontroler** o następujących możliwościach:

- **odczytu i (lub) zapisu danych** przy użyciu warstwy dostępu do danych;
- **organizacji tych danych** na podstawie wygodnego modelu widoku z opcją serializacji do formatu JSON;
- **serializacji modelu widoku i wysłania go do klienta** jako odpowiedzi.

Na podstawie wymienionych punktów łatwo dojść do wniosku, że model widoku (`ViewModel`) to kluczowy element. Nie zawsze jednak jest on najważniejszy — wszystko zależy od rodzaju tworzonego projektu. Zanim przejdziemy dalej, przyjrzyjmy się bliżej obiektom `ViewModel`.

Zadania modelu widoku

Wiemy, że klasa `ViewModel` to klasa typu kontenerowego odpowiedzialna jedynie za przechowywanie danych kierowanych do wyświetlenia na stronie WWW. W standardowych aplikacjach ASP.NET działających na zasadach MVC **model widoku** jest tworzony przez **kontroler** w odpowiedzi na żądanie GET na podstawie danych otrzymanych z **modelu**. Utworzony model widoku trafia do **widoku**, gdzie najczęściej służy do wypełnienia zawartości strony lub pól formularzy.

Głównym powodem tworzenia modelu widoku zamiast bezpośredniego użycia encji modelu jest chęć przekazania tylko tych właściwości, z których rzeczywiście chcemy skorzystać. Pozostałe właściwości obiektów dziedzicznych zostaną pominięte, więc ilość przesyłanych danych zmniejszy się. Dodatkową zaletą jest zwiększone bezpieczeństwo, bo wybrane pola z danymi wrażliwymi możemy łatwo wykluczyć z przesyłania protokołem HTTP.

W standardowym kontekście API webowego, gdzie dane są przekazywane w łatwych do serializacji formatach takich jak JSON lub XML, model widoku można w wielu sytuacjach z powodzeniem zastąpić dynamicznym obiektem tworzonym jednorazowo na potrzeby konkretnej odpowiedzi, na przykład:

```
var response = new {  
    Id = "1",  
    Title = "Tytuł",  
    Description = "Opis"  
};
```

Takie podejście sprawdza się w małych projektach lub prototypach, gdzie tworzenie jednej klasy lub wielu klas modeli widoków byłoby stratą czasu. W naszym przypadku jednak projekt znacząco skorzysta na dobrze zdefiniowanych, silnie typowanych strukturach modeli widoków, nawet jeśli za każdym razem będą one konwertowane na format JSON.

Pierwszy model widoku

Skoro znasz już zasady rządzące cyklem żądanie-odpowiedź, możemy przystąpić do budowania pierwszych elementów. Choć część klienta jeszcze nie istnieje, łatwo odgadnąć, czego będziemy potrzebowali — metod CRUD dla każdego z rodzajów treści wymienionych kilka akapitów wcześniej.

Osoby zaznajomione z systemem ASP.NET MVC zapewne od razu chciałyby przystąpić do utworzenia osobnego kontrolera dla każdego rodzaju treści. Zanim jednak przejdziemy do kontrolerów, warto utworzyć modele widoków, aby obsługa danych odbywała się od razu przy użyciu **silnego typowania**.

Klasa QuizViewModel

Zacznijmy od najważniejszego rodzaju danych w tworzonej aplikacji. Będzie to również najbardziej rozbudowany element.

Chwileczkę, ale dlaczego zaczynamy od modelu widoku, skoro nie mamy jeszcze modelu danych? Skąd uzyskamy dane?

To nietrywialne pytanie wymaga odpowiedzi, zanim przejdziemy dalej. Jedną z ogromnych zalet budowania aplikacji internetowych za pomocą ASP.NET i Angulara jest to, że możemy pisać kod bez przejmowania się źródłami danych. Możemy zająć się nimi później, gdy będziemy wiedzieć, czego tak naprawdę potrzebujemy. Oczywiście nie jest to wymóg i równie dobrze moglibyśmy zacząć od źródeł danych, jeśli:

- dokładnie wiemy, co chcemy wykonać;
- posiadamy już odpowiednie zestawy encji i (lub) zdefiniowane oraz wypełnione struktury danych;
- jesteśmy przyzwyczajeni do zaczynania tworzenia aplikacji od danych, a nie od interfejsu użytkownika.

Każdy z wymienionych powodów jest słuszny — nie spowoduje, że zostaniemy zwolnieni. Z drugiej strony rozpoczęcie prac najpierw nad częścią kliencką pozwoli rozwiązać od razu wiele wątpliwości związanych ze sposobem działania aplikacji, a tym samym ułatwi podjęcie decyzji co do wymaganych rodzajów danych. Budując tę aplikację, skorzystamy właśnie z tego podejścia, zaczniemy więc prace nad klasą `QuizViewModel`, choć nie mamy jeszcze klas **źródeł danych** i **encji**.

Z poziomu panelu *Eksplorator rozwiązań* (*Solution Explorer*) kliknij prawym klawiszem myszy węzeł *TestMakerFreeWebApp* (główna aplikacja), a następnie utwórz nowy folder */ViewModels/*. Następnie kliknij prawym klawiszem myszy nowo utworzony folder i wykonaj standardowe polecenie *Dodaj/Nowy element* (*Add/New Item*).

W widoku drzewa wybierz *ASP.NET Core/Kod* (*ASP.NET Core/Code*), zaznacz element *Klasa* (*Class*), nadaj mu nazwę `QuizViewModel.cs`. Kliknij przycisk *Dodaj* (*Add*), aby utworzyć nowy plik w folderze */ViewModels/*.

Otwórz nowy plik i zamień jego oryginalną treść na poniższą:

```
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
```

```

using System.ComponentModel;
using System.Linq;
using System.Threading.Tasks;

namespace TestMakerFreeWebApp.ViewModels
{
    [JsonObject(MemberSerialization.OptOut)]
    public class QuizViewModel
    {
        #region Konstruktor
        public QuizViewModel()
        {

        }
        #endregion

        #region Właściwości
        public int Id { get; set; }
        public string Title { get; set; }
        public string Description { get; set; }
        public string Text { get; set; }
        public string Notes { get; set; }
        [DefaultValue(0)]
        public int Type { get; set; }
        [DefaultValue(0)]
        public int Flags { get; set; }
        public string UserId { get; set; }
        [JsonIgnore]
        public int ViewCount { get; set; }
        public DateTime CreatedDate { get; set; }
        public DateTime LastModifiedDate { get; set; }
        #endregion
    }
}

```

Jak możesz się przekonać, klasa przypomina typowy obiekt POCO z raczej standardowym zestawem właściwości. Obiekt quizu będzie miał tytuł (Title), opis (Description) itp. W tworzonym obiekcie nadal brakuje kilku elementów, przede wszystkim odniesień do pytań, odpowiedzi i wyników — dodamy je na dalszym etapie prac.

Klasa QuizController

Przejdźmy teraz do utworzenia klasy QuizController.

1. Z poziomu panelu *Eksplorez rozwiązań* (*Solution Explorer*) otwórz folder */Controllers/*.

- Następnie kliknij prawym klawiszem myszy folder i wykonaj standardowe polecenie *Dodaj/Nowy element (Add/New Item)*.

Nie korzystaj z polecenia *Dodaj/Kontroler (Add/Controller)*, bo spowoduje to uruchomienie kreatora, który doda do projektu kilka zależności, a tego na razie nie potrzebujemy.

- W widoku drzewa wybierz *ASP.NET Core/Sieć Web (ASP.NET Core/Web)*, zaznacz *Klasa kontrolera interfejsu API (Web API Controller Class)*, nadaj nowemu plikowi nazwę *QuizController.cs* i kliknij *Dodaj (Add)*, aby dodać nowy plik w folderze */Controllers/*. Plik pojawi się obok istniejących już plików *HomeController.cs* i *SampleDataController.cs*, o których wspomniałem w rozdziale 1.

Kontroler został utworzony z kilkoma przykładowymi metodami, których nie potrzebujemy. Usuń całą istniejącą zawartość pliku i wstaw na jej miejsce poniższy kod:

```
using System;
using Microsoft.AspNetCore.Mvc;
using Newtonsoft.Json;
using TestMakerFreeWebApp.ViewModels;
using System.Collections.Generic;
using System.Linq;

namespace TestMakerFreeWebApp.Controllers
{
    [Route("api/[controller]")]
    public class QuizController : Controller
    {
        // GET api/quiz/latest
        [HttpGet("Latest/{num?}")]
        public IActionResult Latest(int num = 10)
        {
            var sampleQuizzes = new List<QuizViewModel>();

            // Dodaj pierwszy przykładowy quiz
            sampleQuizzes.Add(new QuizViewModel()
            {
                Id = 1,
                Title = "Którą postacią z Shingeki No Kyojin (Atak tytanów) jesteś?",
                Description = "Test osobowości bazujący na anime",
                CreatedDate = DateTime.Now,
                LastModifiedDate = DateTime.Now
            });

            // Dodaj kilka następnych przykładowych quizów
            for (int i = 2; i <= num; i++)
            {
                sampleQuizzes.Add(new QuizViewModel()
                {
```

```

        Id = i,
        Title = String.Format("Przykładowy quiz {0}", i),
        Description = "To jest przykładowy quiz",
        CreatedDate = DateTime.Now,
        LastModifiedDate = DateTime.Now
    });
}

// Przekaż wyniki w formacie JSON
return new JsonResult(
    sampleQuizzes,
    new JsonSerializerSettings()
    {
        Formatting = Formatting.Indented
    });
}
}
}

```

Przyjrzyjmy się dokładnie dodanemu kodowi.

Zdefiniowaliśmy metodę `Latest` przyjmującą pojedynczy, opcjonalny parametr typu `int` o nazwie `num` i wartości domyślnej 10. Metoda przyjmuje dowolne żądanie GET zgodne z zasadami zdefiniowanymi w atrybucie `HttpGet`. Takie rozwiązanie nazywa się **routingiem atrybutowym**, o którym więcej informacji znajdziesz w dalszej części rozdziału. Na razie skupmy się na zawartości omawianej metody.

Działanie kodu jest bardzo proste, bo nie mamy (jeszcze) źródła danych — po prostu zwracamy kilka przykładowych obiektów `QuizViewModel`. Choć tylko symulujemy prawdziwą odpowiedź, czynimy to w sposób ustrukturyzowany i wiarygodny, czyli respektujemy limit zwracanych elementów, a także przekazujemy różniące się między sobą elementy. W zasadzie stosujemy to samo podstawowe podejście, które zostało zaproponowane w pliku `SampleDataController.cs` wygenerowanym przy użyciu szablonu Angular SPA w rozdziale 1.

Zauważ, że zwracamy wartości typu `JsonResult`, co jest najlepszym rozwiązaniem, gdy korzystamy z klas modeli widoków z atrybutem `JsonObject` zapewnianym przez framework `Newtonsoft.Json`. To zdecydowanie lepsze rozwiązanie niż zwracanie zwykłego typu `string` lub `IEnumerable <string>`, bo poza automatyczną serializacją danych ustawione zostaną również niezbędne nagłówki odpowiedzi (`Content-Type`, `charset` itp.).

Dodatkowe metody akcji

Zanim przejdziemy do innych tematów, skorzystajmy z okazji i dodajmy jeszcze dwie metody akcji do klasy `QuizController`, co pozwoli zasymulować różne strategie pobierania danych: otrzymywanie quizów ułożonych alfabetycznie lub całkowicie losowych.

Metoda ByTitle

Umieść poniższy kod tuż po metodzie Latest():

```

/// <summary>
/// GET: api/quiz/ByTitle
/// Pobiera {num} quizów posortowanych po tytule (od A do Z)
/// </summary>
/// <param name="num">liczba quizów do pobrania</param>
/// <returns>{num} quizów posortowanych po tytule</returns>
[HttpGet("ByTitle/{num:int?}")]
public IActionResult ByTitle(int num = 10)
{
    var sampleQuizzes = ((JsonResult)Latest(num)).Value
        as List<QuizViewModel>;

    return new JsonResult(
        sampleQuizzes.OrderBy(t => t.Title),
        new JsonSerializerSettings()
        {
            Formatting = Formatting.Indented
        });
}

```

Nowa metoda wewnętrznie używa metody Latest, która zwraca listę przykładowych quizów, a następnie sortuje je alfabetycznie i zwraca nową listę.

Metoda Random()

W bardzo podobny sposób zaimplementujemy metodę Random():

```

/// <summary>
/// GET: api/quiz/mostViewed
/// Pobiera {num} losowych quizów
/// </summary>
/// <param name="num">liczba quizów do pobrania</param>
/// <returns>{num} losowych quizów</returns>
[HttpGet("Random/{num:int?}")]
public IActionResult Random(int num = 10)
{
    var sampleQuizzes = ((JsonResult)Latest(num)).Value
        as List<QuizViewModel>;

    return new JsonResult(
        sampleQuizzes.OrderBy(t => Guid.NewGuid()),
        new JsonSerializerSettings()
        {
            Formatting = Formatting.Indented
        });
}

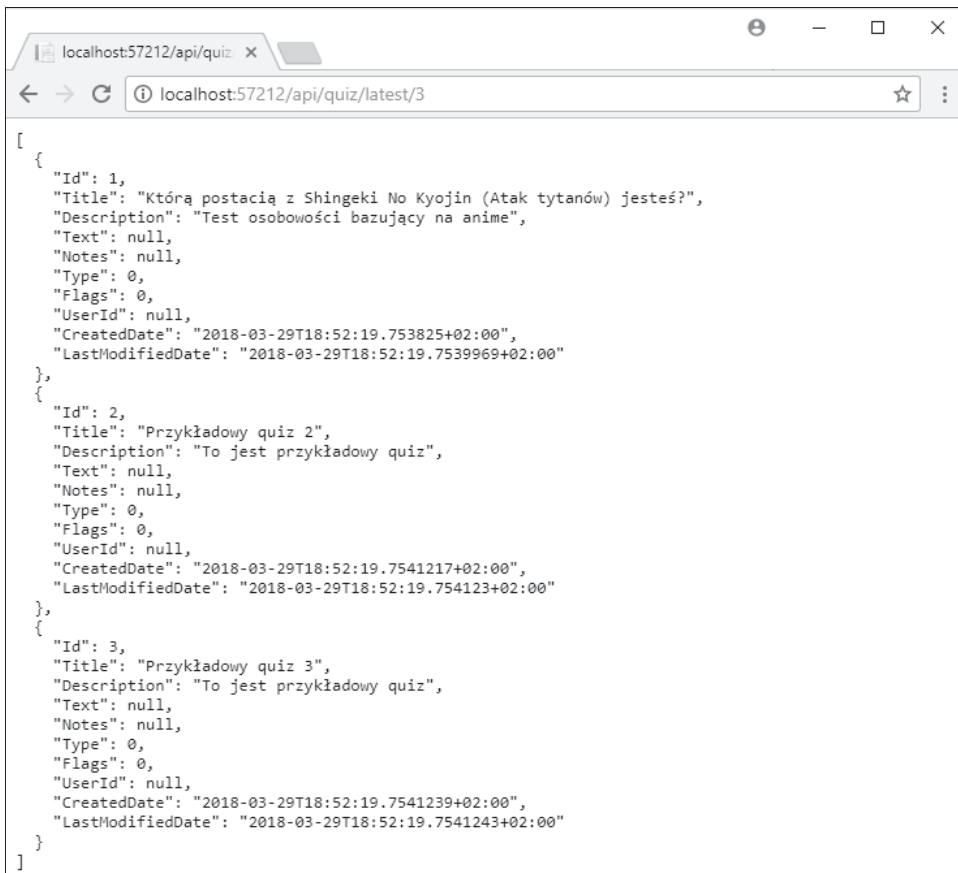
```

Sprawdzenie, czy wszystko działa

Wypróbujmy nowy kontroler, uruchamiając aplikację w trybie debugowania i wpisując w pasku adresu przeglądarki internetowej poniższy adres:

```
http://localhost:<port>/api/quiz/latest/3
```

Jeśli wszystko zostało wykonane poprawnie, powinien ukazać się poniższy wynik.



```
[
  {
    "Id": 1,
    "Title": "Którą postacią z Shingeki No Kyojin (Atak tytanów) jesteś?",
    "Description": "Test osobowości bazujący na anime",
    "Text": null,
    "Notes": null,
    "Type": 0,
    "Flags": 0,
    "UserId": null,
    "CreateDate": "2018-03-29T18:52:19.753825+02:00",
    "LastModifiedDate": "2018-03-29T18:52:19.7539969+02:00"
  },
  {
    "Id": 2,
    "Title": "Przykładowy quiz 2",
    "Description": "To jest przykładowy quiz",
    "Text": null,
    "Notes": null,
    "Type": 0,
    "Flags": 0,
    "UserId": null,
    "CreateDate": "2018-03-29T18:52:19.7541217+02:00",
    "LastModifiedDate": "2018-03-29T18:52:19.754123+02:00"
  },
  {
    "Id": 3,
    "Title": "Przykładowy quiz 3",
    "Description": "To jest przykładowy quiz",
    "Text": null,
    "Notes": null,
    "Type": 0,
    "Flags": 0,
    "UserId": null,
    "CreateDate": "2018-03-29T18:52:19.7541239+02:00",
    "LastModifiedDate": "2018-03-29T18:52:19.7541243+02:00"
  }
]
```

Zwróć uwagę, że właściwość `ViewCount` nie pojawia się w wynikach w formacie JSON. To celowe działanie, bo atrybut został oznaczony jako `JsonIgnore`, czyli jest pomijany na etapie serializacji.

Pierwszy kontroler działa bardzo dobrze. Docień ten fakt! Już niebawem będzie odpowiadał w przykładowej aplikacji za wszystkie operacje związane z quizami.

Dodawanie pozostałych kontrolerów

Skoro wiesz już, jak dodawać kontrolery i modele widoków, dodajmy po parze kontroler-model dla każdego z pozostałych typów danych. Aby się nie powtarzać, pominię opisy tworzenia plików i po prostu zaprezentuję pokrótce opisany kod źródłowy, który należy umieścić w każdym z plików.

Klasa QuestionViewModel

Czym byłby quiz bez pytań? Dodaj plik *QuestionViewModel.cs* w folderze */ViewModels/* i umieść w nim poniższy kod:

```
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
using System.Threading.Tasks;

namespace TestMakerFreeWebApp.ViewModels
{
    [JsonObject(MemberSerialization.OptOut)]
    public class QuestionViewModel
    {
        #region Konstruktor
        public QuestionViewModel()
        {
        }
        #endregion

        #region Właściwości
        public int Id { get; set; }
        public int QuizId { get; set; }
        public string Text { get; set; }
        public string Notes { get; set; }
        [DefaultValue(0)]
        public int Type { get; set; }
        [DefaultValue(0)]
        public int Flags { get; set; }
        [JsonIgnore]
        public DateTime CreatedDate { get; set; }
        public DateTime LastModifiedDate { get; set; }
        #endregion
    }
}
```

Kod bardzo przypomina skonstruowaną wcześniej klasę `QuizViewModel`, ale używa innych właściwości. Jedną z nich jest `QuizId`, co stanowi raczej oczywisty dodatek, bo każde pytanie musi być powiązane z quizem w relacji **jeden do wielu**. Każdy quiz będzie składał się z wielu pytań.

Klasa `QuestionController`

Klasa `QuestionController`, podobnie jak wcześniej klasa `QuizController`, powinna się znaleźć w folderze `/Controllers/`. Umieść ją w pliku o nazwie `QuestionController.cs`:

```
using System;
using Microsoft.AspNetCore.Mvc;
using Newtonsoft.Json;
using TestMakerFreeWebApp.ViewModels;
using System.Collections.Generic;

namespace TestMakerFreeWebApp.Controllers
{
    [Route("api/[controller]")]
    public class QuestionController : Controller
    {
        // GET api/question/all
        [HttpGet("All/{quizId}")]
        public IActionResult All(int quizId)
        {
            var sampleQuestions = new List<QuestionViewModel>();

            // Dodaj pierwsze przykładowe pytanie
            sampleQuestions.Add(new QuestionViewModel()
            {
                Id = 1,
                QuizId = quizId,
                Text = "Co cenisz w swoim życiu najbardziej?",
                CreatedDate = DateTime.Now,
                LastModifiedDate = DateTime.Now
            });

            // Dodaj kilka innych przykładowych pytań
            for (int i = 2; i <= 5; i++)
            {
                sampleQuestions.Add(new QuestionViewModel()
                {
                    Id = i,
                    QuizId = quizId,
                    Text = String.Format("Przykładowe pytanie {0}", i),
                    CreatedDate = DateTime.Now,
                    LastModifiedDate = DateTime.Now
                });
            }
        }
    }
}
```



```

// Przekaż wyniki w formacie JSON
return new JsonResult(
    sampleQuestions,
    new JsonSerializerSettings()
    {
        Formatting = Formatting.Indented
    });
}
}
}

```

Zwróć uwagę, że zamiast metody `Latest` zdefiniowanej w `QuizController` użyliśmy tym razem metody `All`, która zwraca wszystkie pytania związane z określonym quizem na podstawie przekazanego identyfikatora (`quizId`).

Implementacja metody `Latest` nie miałaby dużego sensu, bo pytania same w sobie nie mają żadnej wartości. Powinny być pobierane i prezentowane użytkownikowi tylko w powiązaniu z quizem, którego dotyczą. W takiej sytuacji użycie metody `All` ma znacznie większy sens.

Klasa `AnswerViewModel`

Odpowiedzi są w relacji **jeden do wielu** z pytaniami, podobnie jak pytania z quizami, kod klasy `AnswerViewModel` bardzo przypomina więc kod klasy `QuestionViewModel`:

```

using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;

namespace TestMakerFreeWebApp.ViewModels
{
    [JsonObject(MemberSerialization.OptOut)]
    public class AnswerViewModel
    {
        #region Konstruktor
        public AnswerViewModel()
        {
        }
        #endregion

        #region Właściwości
        public int Id { get; set; }
        public int QuizId { get; set; }
        public int QuestionId { get; set; }
        public string Text { get; set; }
        public string Notes { get; set; }
        }
    }
}

```

```

        [DefaultValue(0)]
        public int Type { get; set; }
        [DefaultValue(0)]
        public int Flags { get; set; }
        [DefaultValue(0)]
        public int Value { get; set; }
        [JsonIgnore]
        public DateTime CreatedDate { get; set; }
        public DateTime LastModifiedDate { get; set; }
    #endregion
    }
}

```

Są tylko dwie istotne różnice:

- pojawiła się właściwość odnosząca się do `QuestionId`, co pozwala na poprawne określenie relacji między odpowiedzią i pytaniem;
- pojawiła się właściwość `Value`, która posłuży nam później do określenia **wartości punktowej** odpowiedzi.

Klasa AnswerController

Oto kod dotyczący klasy `AnswerController`:

```

using System;
using Microsoft.AspNetCore.Mvc;
using Newtonsoft.Json;
using TestMakerFreeWebApp.ViewModels;
using System.Collections.Generic;

namespace TestMakerFreeWebApp.Controllers
{
    [Route("api/[controller]")]
    public class AnswerController : Controller
    {
        // GET api/answer/all
        [HttpGet("All/{questionId}")]
        public IActionResult All(int questionId)
        {
            var sampleAnswers = new List<AnswerViewModel>();

            // Dodaj pierwszą przykładową odpowiedź
            sampleAnswers.Add(new AnswerViewModel()
            {
                Id = 1,
                QuestionId = questionId,
                Text = "Przyjaciół i rodzinę",
                CreatedDate = DateTime.Now,
            });
        }
    }
}

```

```

        LastModifiedDate = DateTime.Now
    });

    // Dodaj kilka następnych przykładowych odpowiedzi
    for (int i = 2; i <= 5; i++)
    {
        sampleAnswers.Add(new AnswerViewModel()
        {
            Id = i,
            QuestionId = questionId,
            Text = String.Format("Przykładowa odpowiedź {0}", i),
            CreatedDate = DateTime.Now,
            LastModifiedDate = DateTime.Now
        });
    }

    // Przekaż wyniki w formacie JSON
    return new JsonResult(
        sampleAnswers,
        new JsonSerializerSettings()
        {
            Formatting = Formatting.Indented
        });
    }
}

```

Obecnie ten kod jest bardzo podobny do kodu klasy `QuestionController`, choć w przyszłości z pewnością ulegnie zmianie.

Klasa `ResultViewModel`

Kontynuujmy pracę i przygotujmy klasę `ResultViewModel`:

```

using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;

namespace TestMakerFreeWebApp.ViewModels
{
    [JsonObject(MemberSerialization.OptOut)]
    public class ResultViewModel
    {
        #region Konstruktor
        public ResultViewModel()
        {

```

```

    }
    #endregion

    #region Właściwości
    public int Id { get; set; }
    public int QuizId { get; set; }
    public string Text { get; set; }
    public string Notes { get; set; }
    [DefaultValue(0)]
    public int Type { get; set; }
    [DefaultValue(0)]
    public int Flags { get; set; }
    [JsonIgnore]
    public DateTime CreatedDate { get; set; }
    public DateTime LastModifiedDate { get; set; }
    #endregion
}
}

```

I tym razem kod jest bardzo podobny do wcześniej przedstawionego. Pewne zmiany pojawiają się w przyszłości.

Klasa ResultController

Umieść w odpowiednim pliku poniższy kod klasy ResultController:

```

using System;
using Microsoft.AspNetCore.Mvc;
using Newtonsoft.Json;
using TestMakerFreeWebApp.ViewModels;
using System.Collections.Generic;

namespace TestMakerFreeWebApp.Controllers
{
    [Route("api/[controller]")]
    public class ResultController : Controller
    {
        // GET api/question/all
        [HttpGet("All/{quizId}")]
        public IActionResult All(int quizId)
        {
            var sampleResults = new List<ResultViewModel>();

            // Dodaj pierwszy przykładowy wynik
            sampleResults.Add(new ResultViewModel()
            {
                Id = 1,
                QuizId = quizId,
            });
        }
    }
}

```

```

        Text = "Co cenisz w swoim życiu najbardziej?",
        CreatedDate = DateTime.Now,
        LastModifiedDate = DateTime.Now
    });

    // Dodaj kilka innych przykładowych wyników
    for (int i = 2; i <= 5; i++)
    {
        sampleResults.Add(new ResultViewModel()
        {
            Id = i,
            QuizId = quizId,
            Text = String.Format("Przykładowe pytanie {0}", i),
            CreatedDate = DateTime.Now,
            LastModifiedDate = DateTime.Now
        });
    }

    // Przekaż wyniki w formacie JSON
    return new JsonResult(
        sampleResults,
        new JsonSerializerSettings()
        {
            Formatting = Formatting.Indented
        });
    }
}
}
}

```

Kod źródłowy klas `ResultViewModel` i `ResultController` jest bardzo podobny do kodu źródłowego klas `QuestionViewModel` i `QuestionController`. Powód jest raczej oczywisty — pytania i wyniki są w sposób bezpośredni powiązane z quizem; nie pojawia się pośrednik, jak to miało miejsce przy odpowiedziach.

Skoro utworzyliśmy już wszystkie niezbędne kontrolery i modele widoków, możemy bezpiecznie pozbyć się pliku `SampleDataController.cs`, bo nie będzie nam już do niczego potrzebny. W panelu *Eksplorator rozwiązań* (*Solution Explorer*) przejdź do folderu `/Controllers/`, kliknij go prawym klawiszem myszy i usuń. Nie spowoduje to żadnych błędów, bo część kliencką, która używała tego kontrolera, usunęliśmy już w rozdziale 1.

Jeśli chcesz zachować plik `SampleDataController.cs`, aby się na nim wzorować w przyszłości, utwórz podkatalog `/Controllers/_usuniete` i przenieś do niego ten plik, czyli postąp podobnie jak w rozdziale 1. z komponentami `counter` i `fetchdata`.

Po wyczyszczeniu projektu przyjrzyjmy się dokładniej zagadnieniu routingu. To jeden z ważniejszych tematów, poświęćmy mu więc nieco miejsca.

Działanie routingu

W rozdziale 1. wspomiałem o tym, że potoki ASP.NET Core zostały napisane całkowicie od nowa, aby połączyć moduły MVC i WebAPI w jeden lekki framework. Choć to bez wątpienia dobry kierunek, w konsekwencji tej zmiany dosyć szybko musimy poznać wiele nowych elementów. Obsługa **routingu** jest tego doskonałym przykładem, bo nowe mechanizmy wprowadzają w tym zakresie istotne zmiany względem rozwiązań stosowanych w przeszłości.

Definiowanie routingu

Zanim przejdziemy do opisów, zdefiniujmy, czym tak naprawdę jest **routing**.

W dużym skrócie: routing adresów URL to funkcjonalność części serwerowej systemu, która pozwala programiście obsługiwać żądania HTTP o adresach niepowiązanych z plikami fizycznymi. Technikę tę wykorzystuje się z wielu różnych powodów, ale głównymi są:

- możliwość zapewnienia dynamicznym stronom WWW znaczących i **zrozumiałych dla ludzi** nazw, aby zwiększyć czytelność lub **zoptymalizować strony dla wyszukiwarek** (SEO — ang. *Search Engine Optimization*);
- możliwość zmiany nazw plików fizycznych w projekcie bez konieczności zmiany ich ogólnodostępnych adresów URL;
- możliwość wykorzystania przekierowań i aliasów.

Routing dawniej i dziś

W czasach, gdy ASP.NET było tylko mechanizmem **Web Forms**, routing był ściśle powiązany z plikami fizycznymi. Aby zaimplementować sensowne konwencje nazewnictwa, programiści byli zmuszeni do instalacji i konfiguracji dedykowanych narzędzi w postaci zewnętrznych filtrów ISAPI, takich jak **ISAPI Rewrite firmy HeliconTech**. Od IIS7 dostępny jest wbudowany moduł **IIS URL Rewrite**.

W momencie wydania **ASP.NET MVC** całkowicie przepisano wzorzec **routingu**, a programiści zyskali możliwość zdefiniowania własnego routingu bazującego na konwencjach w dedykowanym pliku (w zależności od szablonu w pliku *RouteConfig.cs* lub *Global.asax*) z wykorzystaniem metody `Routes.MapRoute`. Dla osób używających dawniej MVC od 1 do 5 lub WebAPI 1 i 2 poniższy fragment kodu będzie wyglądał znajomo:

```
Routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index",
        ↪ id = UrlParameter.Optional }
);
```

Ten sposób definiowania routingu bazował w całości na technikach dopasowywania do wzorca, konkretny adres URL był wiązany z konkretną akcją wybranego kontrolera. Był to tak zwany **routing bazujący na konwencjach**.

Dopiero w **ASP.NET MVC5** po raz pierwszy pojawił się **routing bazujący na atrybutach**. Ten sposób działania dawał programistom większą swobodę. Każdy, kto go używał choć raz, z pewnością zgodzi się, że stanowił on istotny dodatek do frameworka, bo pozwalał określić routing w pliku kontrolera. Nawet osoby, które standardowo stosowały routing bazujący na konwencjach, korzystały z routingu bazującego na atrybutach do nadpisania niektórych ścieżek bez uciekania się do skomplikowanych wyrażeń regularnych:

```
[RoutePrefix("v2Products")]
public class ProductsController : Controller
{
    [Route("v2Index")]
    public ActionResult Index()
    {
        return View();
    }
}
```

W **ASP.NET Core MVC** (dawniej **MVC6**), gdzie ponownie całkowicie przepisano cały mechanizm routingu, **routing bazujący na atrybutach** stał się *de facto* standardem, zastępując podejście **bazujące na konwencjach** w większości przykładów i szablonów. Warto jednak pamiętać, że metoda `Routes.MapRoute()` nadal jest dostępna i stanowi sensowną opcję, jeśli chcemy zdefiniować pewien ogólny mechanizm routingu wysokiego poziomu. Bardzo dobrze widać to w pliku `Startup.cs` (istotne wiersze zostały pogrubione):

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
    routes.MapSpaFallbackRoute(
        name: "spa-fallback",
        defaults: new { controller = "Home", action = "Index" });
});
```

Powyższy fragment kodu pochodzi z metody `Configure` i doskonale pokazuje, że routing bazujący na konwencjach nadal stanowi sensowne rozwiązanie w trakcie prac nad aplikacją SPA. Jedyną różnicą względem ASP.NET 4.x i wcześniejszych wersji polega na tym, że ścieżki są kierowane bezpośrednio do middleware MVC, gdy dodamy je do potoku żądań HTTP jako część konfiguracji. Zapewnia to większą spójność.

Obsługa routingu w .NET Core

Nowa implementacja routingu w zasadzie bazuje na dwóch metodach — `services.AddMvc()` i `app.UseMvc()` — wywoływanych w pliku `Startup.cs`. Wykonują one następujące zadania:

- rejestrację MVC przy użyciu mechanizmu wstrzykiwania zależności wbudowanego w ASP.NET Core;
- dodanie wymaganego middleware do potoku żądań HTTP, a także (opcjonalnie) konfigurację routingu domyślnego.

Możemy zobaczyć, co tak naprawdę dzieje się w środku aktualnej implementacji metody `app.UseMvc()` w kodzie frameworka (istotne wiersze zostały pogrubione):

```
public static IApplicationBuilder UseMvc(
    [NotNull] this IApplicationBuilder app,
    [NotNull] Action<IRouteBuilder> configureRoutes)
{
    // Verify if AddMvc was done before calling UseMvc
    // We use the MvcMarkerService to make sure if all the services were added.
    MvcServicesHelper.ThrowIfMvcNotRegistered(app.ApplicationServices);

    var routes = new RouteBuilder
    {
        DefaultHandler = new MvcRouteHandler(),
        ServiceProvider = app.ApplicationServices
    };

    configureRoutes(routes);

    // Adding the attribute route comes after running the user-code because
    // we want to respect any changes to the DefaultHandler.
    routes.Routes.Insert(0, AttributeRouting.CreateAttributeMegaRoute(
        routes.DefaultHandler,
        app.ApplicationServices));
    return app.UseRouter(routes.Build());
}
```

Zaletą obecnego podejścia jest to, że framework zajmuje się całą żmudną pracą dotyczącą konfiguracji domyślnych routingów dla akcji kontrolerów. Warto zauważyć, że domyślne reguły stosują powszechnie przyjęte konwencje REST, nazwy akcji będą więc ograniczone do Get, Post, Put i Delete. Możemy powiedzieć, że w tej materii ASP.NET Core wprowadza ściśle podejście znane z **WebAPI**, co jednak nie powinno dziwić, bo nowy framework ASP.NET Core zawiera wszystkie elementy poprzednich systemów.

Podejście typu REST to w wielu przypadkach dobre rozwiązanie, szczególnie jeśli zależy nam na pragmatycznych, publicznych API dostępnych dla innych programistów. Jeżeli jednak stworzymy własną aplikację i API nie musi być publiczne, własny system routingu jest równie dobrą opcją. Co więcej, własne ścieżki routingu mogą nas nawet uchronić przed najprostszyimi

formami ataku typu DDoS. Warto pamiętać, że **routing bazujący na konwencjach** i **routing bazujący na atrybutach** nadal jest dostępny, co pozwala szybko zdefiniować własny standard. Aby wymusić ten pierwszy, wystarczy rozbudować kod już istniejący w pliku *Startup.cs*. Możemy też kontynuować definiowanie routingu w kodzie źródłowym kontrolerów, **gdzie routing bazujący na atrybutach** stosuje się najczęściej na poziomie kontrolera:

```
[Route("api/[controller]")]
public class ItemsController : Controller
```

Można go jednak dodać również na poziomie **metody akcji**:

```
[HttpGet("GetLatest")]
public JsonResult GetLatest()
```

Trzy sposoby routingu

Podsumujmy — ASP.NET Core zapewnia nam wybór spośród trzech sposobów obsługi routingu: wymuszenia stosowania **konwencji zgodnych z REST**, powrotu do starego **routingu bazującego na konwencjach** i dekorowania plików kontrolerów przy użyciu **routingu bazującego na atrybutach**. W naszej przykładowej aplikacji wykorzystamy kombinację wszystkich trzech, abyś mógł dobrze poznać każdy z nich.

Warto pamiętać, że ścieżki zdefiniowane za pomocą **routingu na poziomie atrybutów** spowodują nadpisanie dowolnych wzorców **bazujących na konwencjach**. Oba te mechanizmy, jeśli zostaną użyte, wywołają nadpisanie domyślnego routingu zgodnego z REST utworzonego przez wbudowaną metodę `app.UseMvc()`.

Dodawanie nowych ścieżek

Powróćmy do kontrolera `QuizController`. Skoro znasz już różne wzorce routingu, wykorzystamy je do zaimplementowania brakujących wywołań API.

Otwórz plik *QuizController.cs* i dodaj poniższy kod (nowe fragmenty są pogrubione):

```
using System;
using Microsoft.AspNetCore.Mvc;
using Newtonsoft.Json;
using TestMakerFreeWebApp.ViewModels;
using System.Collections.Generic;
using System.Linq;

namespace TestMakerFreeWebApp.Controllers
{
    [Route("api/[controller]")]
    public class QuizController : Controller
    {
```

```

#region Metody dostosowujące do konwencji REST
/// <summary>
/// GET: api/quiz/{id}
/// Pobiera quiz o podanym {id}
/// </summary>
/// <param name="id">Identyfikator istniejącego quizu</param>
/// <returns>Quiz o podanym {id}</returns>
[HttpGet("{id}")]
public IActionResult Get(int id)
{
    // Tworzy przykładowy quiz pasujący do żądania
    var v = new QuizViewModel()
    {
        Id = id,
        Title = String.Format("Przykładowy quiz o identyfikatorze {0}", id),
        Description = "To nie jest prawdziwy quiz - to tylko przykład!",
        CreatedDate = DateTime.Now,
        LastModifiedDate = DateTime.Now
    };

    // Przekaż wyniki w formacie JSON
    return new JsonResult(
        v,
        new JsonSerializerSettings()
        {
            Formatting = Formatting.Indented
        }
    );
}
#endregion

#region Metody routingu bazujące na atrybutach
/// <summary>
/// GET: api/quiz/latest
/// Pobiera {num} najnowszych quizów
/// </summary>
/// <param name="num">liczba quizów do pobrania</param>
/// <returns>{num} najnowszych quizów</returns>
[HttpGet("Latest/{num?}")]
public IActionResult Latest(int num = 10)
{
    var sampleQuizzes = new List<QuizViewModel>();

    // Dodaj pierwszy przykładowy quiz
    sampleQuizzes.Add(new QuizViewModel()
    {
        Id = 1,
        Title = "Którą postacią z Shingeki No Kyojin (Atak tytanów) jesteś?",
        Description = "Test osobowości bazujący na anime",
        CreatedDate = DateTime.Now,
    });
}

```

```

        LastModifiedDate = DateTime.Now
    });

    // Dodaj kilka następnych przykładowych quizów
    for (int i = 2; i <= num; i++)
    {
        sampleQuizzes.Add(new QuizViewModel()
        {
            Id = i,
            Title = String.Format("Przykładowy quiz {0}", i),
            Description = "To jest przykładowy quiz",
            CreatedDate = DateTime.Now,
            LastModifiedDate = DateTime.Now
        });
    }

    // Przekaż wyniki w formacie JSON
    return new JsonResult(
        sampleQuizzes,
        new JsonSerializerSettings()
        {
            Formatting = Formatting.Indented
        });
}

/// <summary>
/// GET: api/quiz/ByTitle
/// Pobiera {num} quizów posortowanych po tytule (od A do Z)
/// </summary>
/// <param name="num">liczba quizów do pobrania</param>
/// <returns>{num} quizów posortowanych po tytule</returns>
[HttpGet("ByTitle/{num:int?}")]
public IActionResult ByTitle(int num = 10)
{
    var sampleQuizzes = ((JsonResult)Latest(num)).Value
        as List<QuizViewModel>;

    return new JsonResult(
        sampleQuizzes.OrderBy(t => t.Title),
        new JsonSerializerSettings()
        {
            Formatting = Formatting.Indented
        });
}

/// <summary>
/// GET: api/quiz/mostViewed
/// Pobiera {num} losowych quizów
/// </summary>

```

```

/// <param name="num">liczba quizów do pobrania</param>
/// <returns>{num} losowych quizów</returns>
[HttpGet("Random/{num:int?}")]
public IActionResult Random(int num = 10)
{
    var sampleQuizzes = ((JsonResult)Latest(num)).Value
        as List<QuizViewModel>;

    return new JsonResult(
        sampleQuizzes.OrderBy(t => Guid.NewGuid()),
        new JsonSerializerSettings()
        {
            Formatting = Formatting.Indented
        });
}
#endregion
}
}

```

Wprowadziliśmy kilka istotnych ulepszeń:

- Dodaliśmy metodę `Get`, która stosuje opisywaną wcześniej **konwencję zgodną z REST**. Metody tej będziemy z pewnością potrzebować, aby pobierać konkretny quiz na podstawie jego identyfikatora.
- Każdy z elementów klasy udekorowaliśmy **znacznikiem dokumentacyjnym <summary>** wyjaśniającym działanie metody i zwracany przez nią wynik. Ze znaczników tych korzysta mechanizm Intellisense z Visual Studio, by wyświetlać w interfejsie graficznym podpowiedzi. Dokumentacja tego typu przyda się również przy automatycznym generowaniu całej dokumentacji projektu za pomocą standardowych narzędzi dokumentacyjnych, takich jak **Sandcastle**.
- Dodaliśmy również dyrektywy preprocesora `#region` i `#endregion`, by podzielić kod na kilka części. Od tej pory będziemy często tak robić, aby poprawić czytelność i użyteczność kodu źródłowego. Zastosowane dyrektywy ułatwiają rozwijanie powiązanych ze sobą fragmentów, kiedy nad nimi pracujemy, i chowanie ich, gdy nie są już aktywnie edytowane.

Więcej informacji na temat znaczników dokumentacyjnych znajdziesz w oficjalnej dokumentacji MSDN na stronie <https://docs.microsoft.com/pl-pl/dotnet/csharp/programming-guide/xml-doc/xml-documentation-comments>.

Aby dowiedzieć się więcej o dyrektywach preprocesora w C#, zajrzyj do dokumentacji dostępnej na stronie <https://docs.microsoft.com/pl-pl/dotnet/csharp/language-reference/preprocessor-directives/preprocessor-region>.

Atrapa dostawcy danych

W kodzie wyraźnie widać, że **emulujemy** działanie **dostawcy danych** zwracającego jeden lub więcej przykładowych quizów. Powód jest oczywisty — na razie nie zdefiniowaliśmy żadnego dostawcy danych. Zrobimy to dopiero w rozdziale 4. Oznacza to, że to jedyny sposób, aby wyświetlić w przykładowej aplikacji treści otrzymywane z serwera.

Zwróć uwagę, że kod został napisany w taki sposób, by zawsze zwracać te same elementy, jeśli tylko parametr `num` pozostanie taki sam.

- Generowane wartości `Id` zmieniają się liniowo od 1 do `num`.
- Każdy wygenerowany element będzie miał nowsze wartości w polach `CreatedDate` i `LastModifiedDate`, czyli im wyższa wartość `Id`, tym nowsze będą quizy. W ten sposób symulujemy standardowe działanie baz danych, w których to przy automatycznej inkrementacji kluczy nowsze wpisy mają wyższe wartości `Id`.

Choć oczywiście zastosowane rozwiązanie nie umożliwi dodawania, edycji i usuwania quizów, pozwoli testować podstawowy kod aplikacji klienckiej do momentu, w którym dodamy trwałego **dostawcę danych**.

Od strony technicznej z pewnością moglibyśmy tutaj działać o wiele więcej, na przykład wykorzystując jeden z dostępnych w **NuGet** frameworków do tworzenia atrap: **Moq**, **NMock3**, **NSubstitute** lub **Rhino**. Te frameworki to idealne rozwiązanie, gdy używamy techniki **TDD** (ang. *Test-Driven Development*). W tej książce nie będziemy się nią zajmować. Atrapa dostawców danych to dobre wyjście w sytuacji rozpoczynania prac nad nowym projektem, bo pozwala szybko sprawdzić interakcję między ASP.NET Core i Angularem.

Obsługa pojedynczych elementów

Uaktualniona klasa `QuizController` pozwala nam pobrać pojedynczy element. Z pewnością będzie to potrzebna operacja, bo dzięki niej będzie można pobrać szczegóły quizu, gdy użytkownik wybierze go z listy najnowszych quizów lub też bezpośrednio przejdzie do strony ze szczegółami quizu. Oczywiście pobieranie pojedynczego elementu przyda się również przy operacjach **CRUD** takich jak aktualizacja lub usunięcie.

Ponieważ jeszcze nie zajmowaliśmy się kodem części klienckiej, nie wiemy, **jak** przedstawimy te operacje użytkownikowi. Z drugiej strony wiemy, **czego** będziemy potrzebować — metod `Get`, `Put`, `Post` i `Delete` dla każdego z rodzajów obiektów (**quizów**, **pytań**, **odpowiedzi** i **wyników**), bo operacje te będzie trzeba realizować dla każdego z nich.

Na szczęście nie musimy ich teraz implementować. Ponieważ właśnie zajmujemy się kontrolerami, zdefiniujmy odpowiednie metody bez ich implementowania.

```
#region Metody dostosowujące do konwencji REST
/// <summary>
/// Pobiera odpowiedź o podanym {id}
/// </summary>
/// <param name="id">identyfikator istniejącej odpowiedzi</param>
/// <returns>odpowiedź o podanym {id}</returns>
[HttpGet("{id}")]
public IActionResult Get(int id)
{
    return Content("(Jeszcze) niezaimplementowane!");
}

/// <summary>
/// Dodaje nową odpowiedź do bazy danych
/// </summary>
/// <param name="model">obiekt AnswerViewModel z danymi do wstawienia</param>
[HttpPut]
public IActionResult Put(AnswerViewModel model)
{
    throw new NotImplementedException();
}

/// <summary>
/// Modyfikuje odpowiedź o podanym {id}
/// </summary>
/// <param name="model">obiekt AnswerViewModel z danymi do uaktualnienia</param>
[HttpPost]
public IActionResult Post(AnswerViewModel model)
{
    throw new NotImplementedException();
}

/// <summary>
/// Usuwa odpowiedź o podanym {id} z bazy danych
/// </summary>
/// <param name="id">identyfikator istniejącej odpowiedzi</param>
[HttpDelete("{id}")]
public IActionResult Delete(int id)
{
    throw new NotImplementedException();
}
}
#endregion
```

Oto kod, który należy dodać w klasie AnswerController. Bardzo podobny kod musimy dodać również do pozostałych kontrolerów: QuestionController, ResultController i QuizController

(w tym przypadku poza metodą `Get`, którą dodaliśmy wcześniej). Pamiętaj o odpowiedniej zmianie treści komentarzy i użyciu właściwej klasy modelu widoku (każdy **kontroler** musi korzystać z referencji do dotyczącego go **modelu widoku**).

Nowe metody wykorzystujące konwencję REST obecnie tak naprawdę nic nie robią poza zwróceniem zwykłego tekstu lub wyjątku `NotImplementedException`. Właściwą implementację tych metod zapewnimy w dalszej części książki.

Aby sprawdzić nowe metody, wybierz z menu Visual Studio polecenie *Debugowanie/Rozpocznij debugowanie* (*Debug/Start Debugging*) lub naciśnij klawisz *F5*. Wpisz w przeglądarce poniższe adresy URL.

- Dla quizu wpisz `/api/quiz/1`.

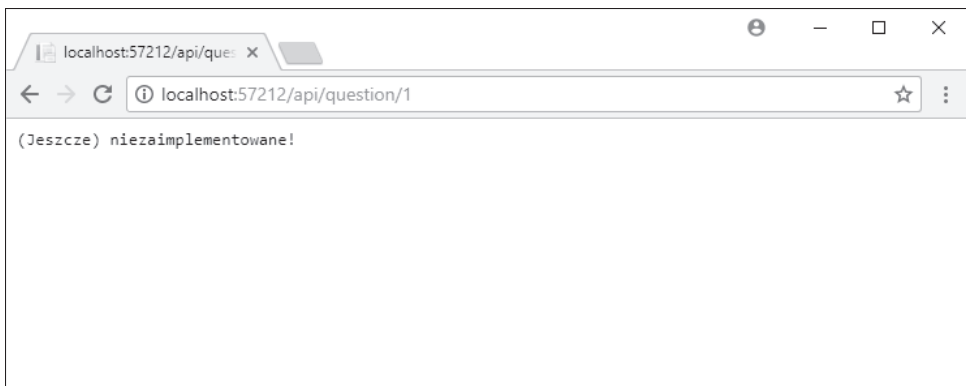


```

{
  "Id": 1,
  "Title": "Przykładowy quiz o identyfikatorze 1",
  "Description": "To nie jest prawdziwy quiz - to tylko przykład!",
  "Text": null,
  "Notes": null,
  "Type": 0,
  "Flags": 0,
  "UserId": null,
  "CreateDate": "2018-03-29T19:52:10.6356278+02:00",
  "LastModifiedDate": "2018-03-29T19:52:10.6358442+02:00"
}

```

- Dla pytania wpisz `/api/question/1`.

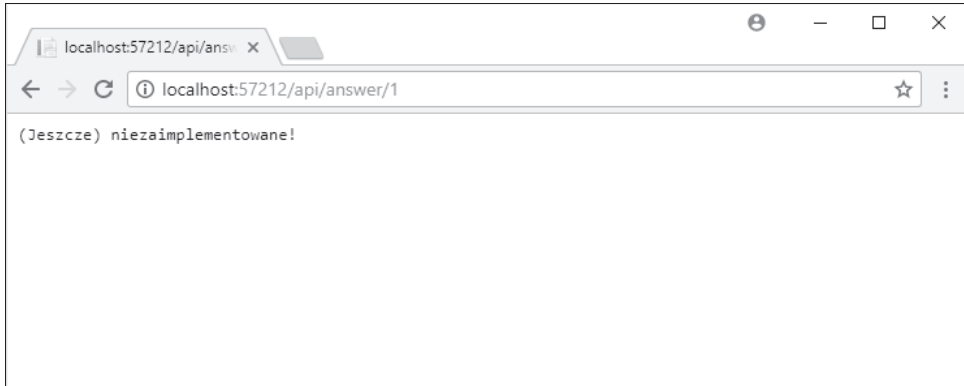


```

(Jeszcze) niezaimplementowane!

```

- Dla odpowiedzi wpisz `/api/answer/1`.



- Dla wyniku wpisz `/api/result/1`.



Wszystko działa dokładnie tak, jak powinno. Jak wcześniej wspomniałem, informacja o braku implementacji jest na razie odpowiednia. Ponieważ standardowo przeglądarka używa żądań typu GET, nie sprawdzimy działania wyjątku `NotImplementedException`. Nie jest to jednak istotne — najważniejsze jest to, że przy braku implementacji zwracamy klientowi dane lub informację o wyjątku bez żadnych negatywnych konsekwencji dla pozostałej części aplikacji.

Przygotowaliśmy zestaw kilku serwerowych API, które zwracają klientowi tablice w formacie JSON wypełnione wskazaną (lub domyślną) listą najnowszych elementów, a także pozwalają pobrać pojedynczy element na podstawie jego identyfikatora. Wszystkie wymienione operacje przydadzą się w następnym rozdziale, gdy będziemy tworzyć w Angularze komponenty części klienckiej.

Poruszana tematyka

Żądanie HTTP, odpowiedź HTTP, routing bazujący na konwencjach, routing bazujący na atrybutach, konwencje REST, obiekty będące atrapami, programowanie w stylu TDD, znaczniki dokumentacyjne XML i dyrektywy preprocesora języka C#.

Podsumowanie

Poświęciliśmy nieco czasu na utworzenie standardowego przepływu danych w aplikacji, bazującego na dwukierunkowej komunikacji klienta i serwera przy użyciu protokołu HTTP. Przyjęliśmy, że dane będą przesyłane w formacie JSON. Dla quizów wykonaliśmy serwerową klasę `QuizViewModel`, która będzie serializowana, a także klasę `QuizController` odpowiadającą za przygotowanie i przesłanie danych.

Zaczęliśmy budowę interfejsu Web API bazującego na MVC6 od najważniejszych metod wymaganych przez interfejs użytkownika części klienckiej. Zastosowaliśmy **routing bazujący na atrybutach**, bo w tej sytuacji to najlepsze rozwiązanie.

Ponieważ zajmowaliśmy się kontrolerami, przy okazji utworzyliśmy metody związane z dodawaniem, pobieraniem edycją i usuwaniem pojedynczych wpisów. Tym razem przy implementacji metod `Get`, `Put`, `Post` i `Delete` posłużyliśmy się **routingiem używającym konwencji REST**.

W następnym rozdziale zaczniemy korzystać z przygotowanego API, budując w Angularze interaktywny interfejs użytkownika.

Skorowidz

- .NET Core, 79
 - część serwerowa, 79
 - mechanizmy uwierzytelniania, 335
 - obsługa routingu, 98
- .NET Core Identity, 335

A

- ADAL, Azure AD
 - Authentication Library, 382
- adres URL, 447
- aktualizacja
 - Angulara, 52
 - bazy danych, 343
 - encji użytkownika, 388
 - komponentów, 315
 - kontrolerów, 355
 - pakietów, 53
 - plików AppSettings, 352
- aktywacja trybu edycji, 204
- analiza błędów, 457
- Angular, 27, 109
 - część kliencka, 109
 - formularze, 302, 361
- Angular Universal, 364
- API, 26
- aplikacja TestMakerFree, 33
 - funkcjonalności, 34
 - wymagania, 34
- aplikacje SPA, 19, 29
 - oczekiwania właściciela produktu, 30
 - projekt, 32
 - typowe funkcjonalności, 29
- arkusze stylów, 251

- ASP.NET Core, 452
 - dla IIS, 452
 - moduł logowania, 461
- atrapa dostawcy danych, 103
- automatyczna aktualizacja pakietów, 51
- autoryzacja, 329, 332, 376
 - udział strony trzeciej, 333
- AWS, Amazon Web Services, 349

B

- backend, *Patrz* strona serwerowa
- baza danych, 157
 - aktualizacja, 343
 - dodanie nowego użytkownika, 445
 - konfiguracja, 443
 - konfiguracja połączenia, 446
 - migracja, 343
 - powiązanie loginu, 445
 - strategie inicjalizacji, 172
 - tworzenie, 174, 444
 - uwierzytelnianie, 444
 - wybór, 172
 - wypełnianie danymi, 177, 186, 345
- biblioteka
 - ADAL, 382
 - Mapster, 188
 - RxJS, 116
- blokadry wzajemne, 342

- błąd, 457
 - braku pliku, 176
 - HTTP o kodzie 401, 358
 - HTTP o kodzie 500, 358
- błędy TypeScript, 72, 73
- Bootstrap, 267
 - zmiana motywu, 267

C

- CompileSass, 260
- CoreCLR, 26
- CRUD, Create-Read-Update-Delete, 80
- CSRF, Cross-Site Request Forgery, 382
- CSS, Cascading Style Sheets, 252
- cykl
 - żądanie-odpowiedź, 81
 - życia, 129
- czyszczenie aplikacji klienckiej, 71

D

- debugowanie, 128, 321
 - po stronie klienta, 326
- definiowanie relacji, 168
- routingu, 96
- dodanie
 - encji dla tokena, 387
 - interfejsów, 228
 - komponentów, 118, 122, 143, 200

dodanie
 listy, 124
 migracji, 343
 migracji początkowej, 174
 nowej witryny, 453
 obiektu HttpClient, 141
 plików komponentu, 121
 pliku szablonu, 426
 referencji, 233
 skrótów, 313
 ścieżki edycji, 205
 tokena, 373
 walidatorów, 312

dokumentacja LESS, 259

domieszki, 257

dostawca danych, 191

dwukierunkowe dowiązanie
 danych, 133

dynamiczne pakowanie
 modułów, 59

dyrektywy importu, 256

działanie
 mechanizmu migracji, 177
 routingu, 96
 stref, 424
 uwierzytelniania OAuth2,
 409

dziennik aktywności, 326

E

EDM, Entity Data Model, 154

EDMX, Entity Designer
 Model XML, 156

element
 app, 63
 div, 296
 img, 277
 input, 205
 link, 63
 script, 63
 summary, 102

emulowanie dostawcy danych,
 103

encje, 160
 użytkownika, 388

enkapsulacja, 281
 CSS, 278
 natywna, 280

Entity Framework Core, 153
 instalacja, 154
 model danych, 153

Event Viewer, 461

F

Facebook, 412

folder
 /ClientApp/, 44
 app/, 64, 72
 dist/, 63
 test/, 63
 /components/, 64
 /Controllers/, 44
 /counter/, 72
 /fetchdata/, 72
 /home/, 72
 /navmenu/, 72
 /Views/, 44, 62
 /wwwroot/, 44
 dist/, 268
 Zależności, Dependencies, 44

format JSON, 114

formularze, 297, 301
 reaktywne, 307
 sterowane modelem, 304
 sterowane szablonami, 302
 testy jednostkowe, 327
 w Angularze, 302, 361

framework
 Angular, 25, 109
 ASP.NET Core, 25
 CSS, 265

frontend, *Patrz* strona kiencka

G

graf zależności, 56, 60

H

host, 45

I

identyfikator, 139
 aktualnego użytkownika, 380

ikony, 295

implementacja
 funkcjonalności usuwania,
 206
 LESS, 260
 tokena odświeżania, 389
 uwierzytelniania JWT, 350

inicjalizacja bazy danych, 172

instalacja
 Entity Framework Core, 154
 kompilatora LESS, 261
 modułu ASP.NET Core, 452
 pakietu
 Authentication.Facebook,
 428
 SQL Server 2017 Express
 Edition, 442
 SQL Server Management
 Studio, 442

instancje komponentu, 125

integracja z IIS, 46

Intellisense, 52

interakcja, 195
 klienta z serwerem, 207

interfejs
 OnInit, 129
 graficzny, 251
 sieciowy, 26
 TokenResponse, 361, 395
 User, 403
 użytkownika
 test, 287, 292, 298
 zmiana struktury, 271

izomorficzny JavaScript, 364

J

język TypeScript, 54

języki arkuszy stylów, 252

JWT, JSON Web Token, 349

K

kaskadowe arkusze stylów, CSS,
 252

klasa
 Answer, 165
 AnswerController, 92, 217
 AnswerViewModel, 91
 ApplicationDbContext, 388

- ApplicationUser, 160, 336
 - AppModule, 119, 146, 365
 - AppModuleShared, 74
 - AuthInterceptor, 373, 374
 - AuthResponseInterceptor, 396, 399
 - AuthService, 361, 395
 - BaseApiController, 225, 226, 354
 - DbContext, 337
 - DbSeeder, 178, 184, 338
 - HttpClient, 114, 115
 - Question, 164
 - QuestionController, 90, 213
 - QuestionViewModel, 89
 - Quiz, 161
 - QuizComponent, 121
 - QuizController, 84, 103, 186, 196
 - QuizViewModel, 83
 - RegisterComponent, 403
 - Result, 166
 - ResultController, 94, 221
 - ResultViewModel, 93
 - SignInManager, 431
 - Startup, 46, 47
 - TokenController, 353, 355, 390, 416, 430
 - TokenRequestViewModel, 358
 - TokenResponseViewModel, 359, 389
 - UserController, 400
 - UserViewModel, 402
 - ViewModel, 82
 - klasy
 - AppModule, 74
 - potomne, 336
 - kliencka część interfejsu, 111
 - klucz prywatny, 349
 - kod
 - po stronie klienckiej, 63
 - po stronie serwerowej, 61
 - kompilacja plików LESS, 263
 - kompilator
 - TSC, 55
 - komponent
 - .NET Core, 26
 - AboutComponent, 144
 - AnswerEditComponent, 240, 316
 - AnswerListComponent, 237
 - AppComponent, 271
 - HomeController, 282
 - LoginComponent, 145, 365, 407
 - LoginExternalProvidersComponent, 435, 438
 - LoginFacebookComponent, 421, 427
 - NavMenu, 407
 - NavMenuComponent, 273, 377
 - PageNotFoundComponent, 145
 - QuestionEditComponent, 234, 315
 - QuestionListComponent, 228
 - QuizComponent, 139, 289, 379
 - QuizEditComponent, 294, 295, 308
 - QuizEditController, 200
 - QuizListComponent, 112, 142, 283
 - QuizSearchComponent, 275, 276
 - ResultEditComponent, 243, 319
 - ResultListComponent, 241
 - komponenty
 - Angulara, 35
 - firm trzecich, 36
 - stylowanie, 278
 - komunikacja między klientem i serwerem, 79, 211
 - konfiguracja
 - .NET Core Identity, 335
 - bazy danych, 443
 - foldera treści, 46
 - integracji z IIS, 46
 - mechanizmu pakowania modułów, 59
 - obiektu DbContext, 170
 - połączenia z bazą danych, 447
 - potoku żądań HTTP, 47
 - puli aplikacji, 455
 - serwera, 452
 - serwera Kestrel, 46
 - środowiska, 38, 42
 - usług, 47
 - usługi Identity, 335
 - usługi uwierzytelniania, 428
 - wstrzykiwania zależności, 47
 - z wykorzystaniem wiersza poleceń, 42
 - kontrola nad plikami statycznymi, 57
 - kontroler, 35
 - QuizController, 99
 - QuizEditController, 232
 - kontrolery API, 35, 61
- ## L
- leniwe wczytywanie danych, 170, 388
 - LESS, 252, 254
 - dokumentacja, 259
 - domieszki, 257
 - dyrektywy importu, 256
 - implementacja, 260
 - instalacja kompilatora, 261
 - kompilacja plików, 263
 - pseudoklasy, 258
 - zagnieżdżanie selektorów, 256
 - zmienne, 255
 - LESS Compiler, 260
 - liczba komponentów, 72
 - listy, 124
 - literal tekstowy, 50
 - logo, 277
 - logowanie
 - do Facebooka, 412
 - w Angularze, 361
- ## M
- Mapster, 187
 - instalacja, 187
 - mechanizm
 - Intellisense, 52
 - interakcji klient-serwer, 79
 - janowy, 410, 427

- mechanizm
 - middleware, 48
 - migracji, 177
 - niejawny, 410, 415
 - OpenID, 331
 - pakowania modułów, 59
 - Shadow DOM, 280
 - task runner, 42
 - Web Forms, 96
 - mechanizmy uwierzytelniania, 335
 - metoda
 - app.UseMvc(), 48
 - app.UseStaticFiles(), 48
 - ByTitle(), 87
 - Configure(), 48, 68
 - ExternalLogin(), 430
 - ExternalLoginCallback(), 431
 - GenerateRandomPassword(), 419
 - ngOnChanges(), 231
 - ngOnInit(), 131
 - onSelect(), 117
 - Random(), 87
 - UseWebpackDevMiddle
 - ware(), 50, 57
 - WebHost.CreateDefault
 - Builder(), 46
 - metody akcji, 86
 - metodyka SCRUM, 31
 - middleware, 48, 58
 - Microsoft.AspNet.Core
 - StaticFiles, 48
 - MVC, 48
 - StaticFiles, 49, 50
 - migracja, 343, 344
 - początkowa, 174
 - minimalistyczny szkielet strony, 62
 - model, 156
 - danych, 34, 109, 153
 - formularza, 321
 - widoku, 82
 - modelowanie danych
 - najpierw baza danych, 157
 - najpierw kod, 158
 - najpierw model, 156
 - moduł
 - aplikacji, 74
 - ASP.NET Core dla IIS, 452
 - korzenia, 74
 - logowania, 461
 - moduły Angulara, 74
 - motyw Flatly, 268, 270
 - MVCApplication, 45
- ## N
- narzędzia NPM, 51
 - narzędzie
 - Event Viewer, 461
 - Mapster, 187
 - Postman, 359
 - Webpack, 50, 56, 263
 - NavMenu, 75
 - nawigacja, 110
 - Node.js, 51
 - nowy projekt, 38
 - NPM, Node Package Manager, 51
- ## O
- OAuth2, 333, 409
 - obiekt
 - clientBundleConfig, 57
 - DbContext, 170
 - HttpClient, 141
 - IConfiguration, 71
 - IWebHost, 46
 - Observable, 207, 323
 - serverBundleConfig, 57
 - sharedConfig, 57
 - obsługa
 - pojedynczych elementów, 103
 - routingu, 98
 - SSR, 65
 - zdarzeń, 205
 - obstylowanie, 35
 - ochrona serwera, 380
 - oczekiwania właściciela produktu, 30
 - odświeżanie, 385
 - sesji, 383
 - plików, 60
 - odwrócony serwer
 - pośredniczący, 452
 - opcja HotModuleReplacement, 50
 - OpenID, 331
 - OpenID Connect, 332
 - operacje CRUD, 195
 - operator potoku, 322
 - oprogramowanie, 22, 38
 - optymalizacja SEO, 42
 - OWIN, Open Web Interface for .NET, 47
- ## P
- pakiet
 - .NET Core Windows Server Hosting, 453
 - Authentication.Facebook, 428
 - Microsoft Visual C++ 2015 Redistributable, 453
 - pakiety NPM, 51
 - pamięć podręczna, 66
 - para klucz-wartość, 51
 - plik
 - app.component.ts, 64
 - app.module.browser.ts, 64
 - app.module.shared.ts, 65, 72, 73
 - AppModule, 406
 - appsettings.Development
 - json, 51
 - appsettings.json, 46, 50, 69, 71, 173, 429
 - arkusza stylów, 118
 - boot.browser.ts, 63
 - boot.server.ts, 63
 - boot.ts, 64
 - Global.aspx, 47
 - home.component.html, 76
 - HomeController.cs, 61
 - launchSettings.json, 448
 - modułu, 119
 - navmenu.component.html, 75
 - package.json, 51, 53
 - Program.cs, 45
 - SampleDataController.cs, 61

Startup.cs, 47, 50, 68, 174
 szablonu, 117, 233, 240
 tsconfig.json, 54
 update-webpack.bat, 269
 web.config, 67, 459
 webpack.config.js, 56, 58
 webpack.config.vendor.js, 56, 59

pliki
 .csproj, 269
 .js, 44
 .json, 44
 .ts, 44
 AppSettings, 352
 konfiguracyjne, 45
 konfiguracyjne Webpack, 56
 statyczne, 44, 50, 66
 SVG, 277

pobieranie
 danych, 141
 identyfikatora, 139

poła tekstowe, 297

połączenie z bazą SQL Server, 446

Postman, 359

potok żądań HTTP, 47

powiązanie ogóln-szczegóły, 111

proces uruchamiania aplikacji, 64

profil publikacji, 449
 do folderu, 451

programowanie ekstremalne, XP, 30

projekt
 część kliencka, 28, 44
 część serwerowa, 28, 44

protokół FTP, 450

przepływ
 danych, 79
 komunikacji, 211

przyciski, 295

pseudoklasa
 extend, 258

publikacja
 aplikacji internetowej, 448
 poprzez protokół FTP, 450

PWA, Progressive Web Apps, 19

R

reagowanie na zmiany, 323

reaktywne formularze, 307

redukcja liczby żądań HTTP, 56

refaktoryzacja aplikacji, 137

referencja, 237, 241, 242
 do ReactiveFormsModule, 307

rejestracja
 komponentu, 276
 nowej ścieżki, 138
 użytkownika, 400

relacja jeden-do-wielu, 168, 170

renderowanie, 464
 po stronie serwera, 42, 383

Roslyn, 26

routing, 34, 96, 237, 241, 242
 bazujący na atrybutach, 97
 bazujący na konwencjach, 99
 po stronie klienckiej, 135
 używający konwencji REST, 107
 wewnątrz aplikacji klienckiej, 111

RxJS, ReactiveX JavaScript, 116

RyuJIT, 26

ryzyko konfliktu, 56

S

Sass, 260

SCRUM, 31

serwer
 IIS, 35
 IIS Express, 35
 Kestrel, 46, 462, 463

sesje, 346

Shadow DOM, 280

silne typowanie, 50, 71, 115

silnik, 456

skrót, 313

SoC, Separation of Concerns, 252

SPA, Single-Page Application, 19, 29

SSL, Secure Socket Layer, 382

SQL Server, 441, 446
 2017 Express Edition, 442
 Management Studio, 442

SSR, Server-Side Rendering, 42, 65, 383

strategia
 HashLocationStrategy, 136
 PathLocationStrategy, 136

strefy, 424

strona kliencka, 55, 63, 109
 debugowanie, 326
 framework Angular, 109
 interakcje, 195
 zadania, 228, 394, 402

strona serwerowa, 61
 .NET Core, 79
 zadania, 213, 387, 400

struktura interfejsu użytkownika, 271

style
 framework CSS, 264
 samodzielne definiowanie, 264

stylowanie komponentów, 278

Stylus, 260

Switch CSS, 260

sygnatury, 349

system kontroli wersji, 35
 Git, 35
 Mercurial, 35
 Team Foundation, 35

system Webpack, 45

szablon
 Angular SPA, 50
 ASP.NET Core, 41
 komponentu
 HomeComponent, 120

szablony Visual Studio, 40

Ś

ścieżka edycji, 205

środowisko pracy, 35

T

tabele, 296

testowanie, 120, 123, 128, 133, 321
 dostawcy danych, 191
 działania aplikacji, 148, 245, 399, 407, 427

testowanie

- interfejsu użytkownika, 287, 292, 298
- jednostkowe formularzy, 327
- konfiguracji środowiska, 42
- logowania, 370
- routingu, 143

token, 373, 348

- dostępowy, 392
- dostępowy OAuth2, 418
- odświeżania, 385–389

tokeny JWT, 383

tryb

- debugowania, 49, 70, 75
- edycji, 204

tworzenie

- aplikacji Facebooka, 412
- bazy danych, 174
- encji, 160
- profilu publikacji, 449

TypeScript, 54

U

układ interfejsu graficznego, 251

uruchamianie

- silnika, 456
- serwera Kestrel, 463

usługa

- Identity, 335, 343
- IIS, 452
- uwierzytelniania, 350

usuwanie, 206

uwierzytelnianie, 35, 195, 329, 346

- bazujące na sesji, 347
- bazujące na tokenie, 348
- bazy danych, 444
- dodanie usługi, 350
- dwuetapowe, 349
- dzięki firmom trzecim, 331, 409
- JWT, 350
- na styku klient-serwer, 381
- OAuth2, 409
- poprzez Facebooka, 428
- sygnatury, 349
- w .NET Core, 335

V

- Visual Studio 2017, 35
- nowy projekt, 38

W

- walidacja danych, 302
- walidatory, 312, 406
- warunkowa dyrektywa kompilacji, 180
- wdrożenie, 441, 457
- Web Compiler, 260
- Web Components, 280
- Web Forms, 96
- Web Host, 45
- Webpack, 45, 263
 - dynamiczne pakowanie modułów, 59
 - kompilacja plików LESS, 263
 - odświeżenie plików, 60
 - pliki konfiguracyjne, 56
- wersje oprogramowania, 38
- weryfikacja danych, 301
- widok, View, 45
 - strony głównej, 61
- wielokrotne użycie komponentów, 125
- wiersz poleceń
 - konfiguracja środowiska, 42
 - narzędzie NPM, 51
- wirtualny folder Zależności, 44, 51
- witryny jednostronowe, 61
- własne walidatory, 406
- własny system siatki, 265
- właściciel produktu, 30
- właściwość encji, 170
- wstępne renderowanie, 364
- wstrzykiwanie zależności, 47, 50, 116
- wybór bazy danych, 172
- wygasanie tokena, 385
- wyłączenie enkapsulacji, 281
- renderowania, 464
- wymuszenie autoryzacji, 376

wzorzec

- nawigacji, 110
- routingu, 96

X

- XP, Extreme Programming, 30
- XSD, XML DataSet Schema, 156

Z

zadania

- asynchroniczne, 342
- modelu widoku, 82
- po stronie klienckiej, 228, 394, 402
- po stronie serwerowej, 213, 387, 400

zagnieżdżanie selektorów, 256

zarządzanie stanem sesji, 382

zasada podziału

- odpowiedzialności, SoC, 252

zastosowanie migracji, 344

zdarzenia cyklu życia, 129

zgodność wsteczna, 114

zmiana

- motywu, 267
- struktury interfejsu użytkownika, 271

zmienna, 255

- środowiskowa ASPNETCORE_ENVIRONMENT, 461

znacznik, *Patrz* element

znak

- karety, 51
- tyldy, 51

Ż

żądanie-odpowiedź, 80

żądanie

- GET, 81, 82
- POST, 310

PROGRAM PARTNERSKI

— GRUPY 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
Helion

Nowoczesne aplikacje: oszałamiająca wydajność, wszechobecna prostota!

Framework ASP.NET Core odzwierciedla zmianę podejścia do technologii strony klienta: niezależność od platformy sprzętowej, łatwiejsze prowadzenie testów jednostkowych i rozbudowa tworzonych systemów. Kolejna wersja frameworka odzwierciedla dalszą ewolucję koncepcji: od aplikacji, które miały być reaktywne i responsywne, do progresywnej realizacji zadań. Również technologia strony serwera poczyniła postępy w zakresie stabilności i wydajności pracy, co w widoczny sposób przyczyniło się do radykalnych zmian kolejnych wersji Angulara. Poprzednie wersje ASP.NET Core i Angulara były znakomitą propozycją dla projektantów całościowych rozwiązań. Czy bezproblemowe stosowanie obu tych narzędzi będzie możliwe w przypadku ich najnowszych wersji?

Dzięki tej książce dowiesz się, w jaki sposób zrealizować kompleksowy projekt aplikacji internetowej, zapewniając efektywną pracę jej części klienckiej i serwerowej za pomocą frameworków ASP.NET Core 2 i Angular 5. Dowiesz się, jak zapewnić obsługę wywołań API oraz routingu po stronie serwera, czym jest dowiązanie dwukierunkowe, jak wykorzystać obiekty Observable i jak wstrzykiwać zależności. Nauczysz się stosować Entity Framework Core do tworzenia modelu danych, a framework Bootstrap i narzędzie LESS do nadania odpowiednich stylów. Poznasz różne techniki uwierzytelniania klientów, w tym protokół OAuth 2. Dowiesz się też, jak poprawnie skonfigurować mechanizm odwrotnego pośrednika między serwerami IIS i Kestrel.

W książce między innymi:

- solidne wprowadzenie do frameworków i przygotowanie środowiska pracy
- Entity Framework Core i implementacja modelu danych
- nowoczesne podejście do interfejsu graficznego
- zaawansowane funkcje formularzy, w tym walidacja i weryfikacja danych
- uwierzytelnianie i autoryzacja oraz praca z tokenami
- wdrażanie aplikacji w środowisku produkcyjnym

Valerio De Sanctis od kilkunastu lat zarządza projektami budowania profesjonalnych witryn internetowych. Zdobył również spore doświadczenie w branży finansowej i w ubezpieczeniach. Specjalizuje się w implementacji i utrzymaniu rozwiązań wykorzystujących technologię .NET. Współpracował między innymi z London Stock Exchange Group, Zurich Insurance Group, Allianz, Generali i wieloma innymi firmami.

 helion.pl	<i>Sprawdź nasze szkolenia</i> SZKOLENIA  AKADEMIA IT & BUSINESS WWW.SZKOLENIA.HELION.PL	KOD KORZYŚCI Słęgnij po więcej! ▶  ISBN 978-83-283-4643-7  9 788328 346437
 helion.pl		
 HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl		
INFORMATYKA W NAJLEPSZYM WYDANIU		Cena: 79,00 zł

Packt