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

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.

Spis treści

ROZDZIAŁ 2

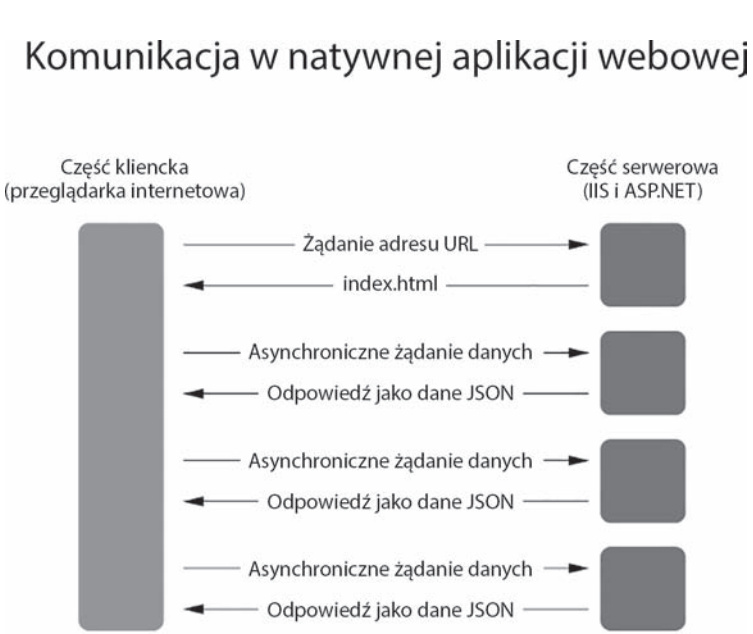
Część serwerowa wykorzystująca .NET Core

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:



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.

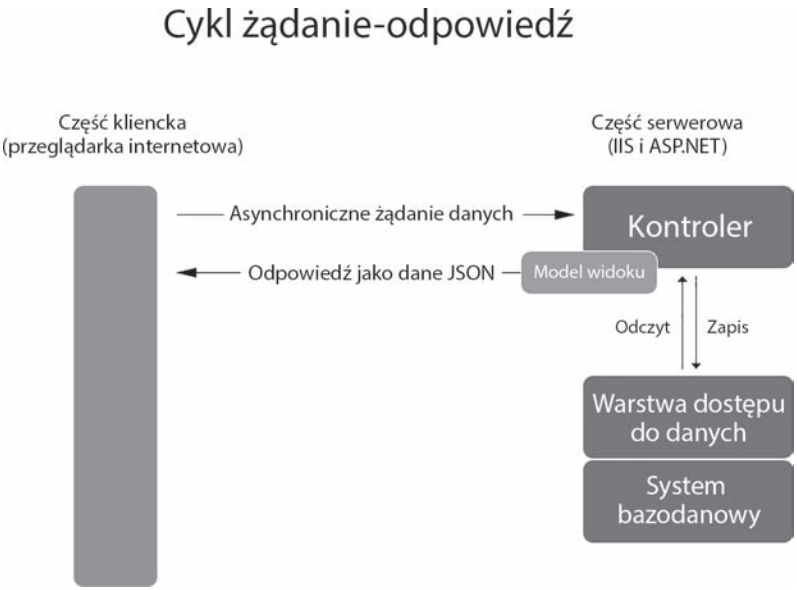
Przedstawiony schemat pokazuje, co musimy zrobić — zdefiniować i zaimplementować wzorzec obsługi żądań bazujących na formacie JSON i przesyłać do klienta wymagane dane. Ponieważ zdecydowaliśmy się na aplikację wymagającą wielu różnych rodzajów informacji, z pewnością wykonamy kilka standardowych zestawów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



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 dziedzinowych 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 kliencka 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 rozwiać 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 *Eksplorator rozwiązań* (*Solution Explorer*) otwórz folder */Controllers/*.

2. 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.

3. 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 dokładniej 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()`:

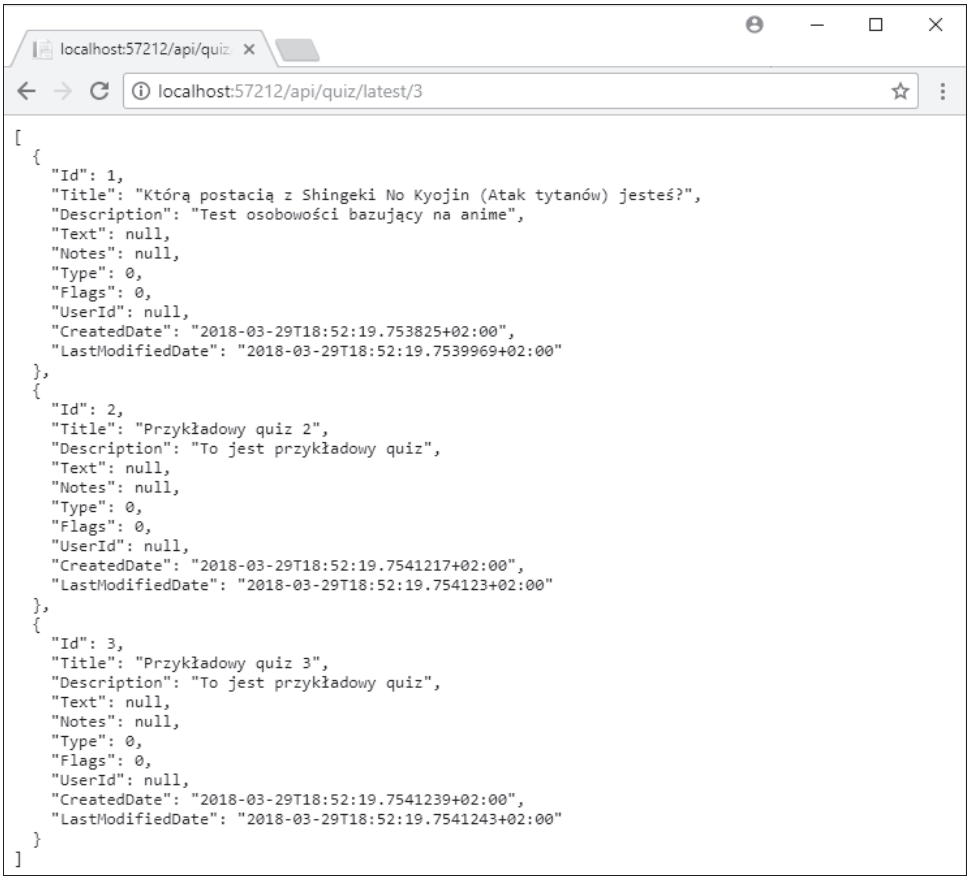
```
/// <summary>
/// GET: api/quiz/mostViewed
/// Pobiera {num} losowych quizów
/// </summary>
/// <param name="num">liczba quizów do pobrania</param>
/// <returns>{num} losowych quizów</returns>
[HttpGet("Random/{num:int?}")]
public IActionResult Random(int num = 10)
{
   var sampleQuizzes = ((JsonResult)Latest(num)).Value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



> 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. Doceń 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, pominę 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 pojawią 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. wspomniałem 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. Jedyna różnica 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 ścisłe 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 tworzymy 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 najprostszymi

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/xmldoc/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 umożliwia 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 zdziałać 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`

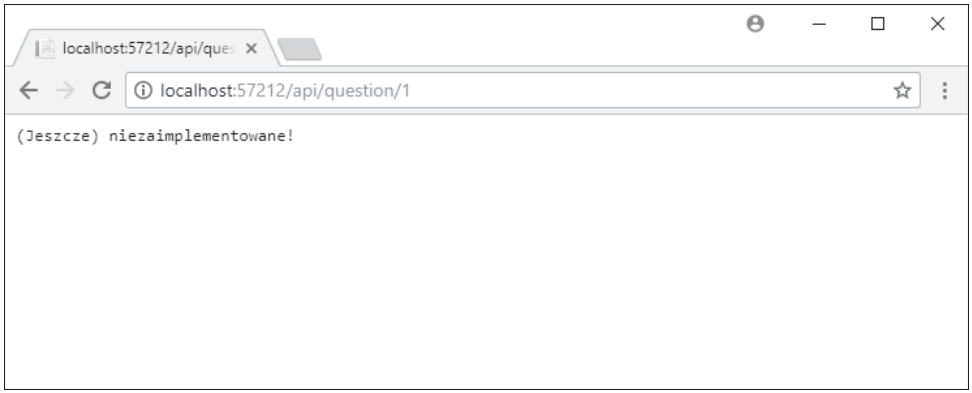
(w tym przypadku poza metodą `Get`, którą dodaliśmy wcześniej). Pamiętaj o odpowiedniej zmianie treści komentarzy i użyciu właściwej klasy modelu widoku (każdy **kontroler** musi korzystać z referencji do dotyczącego go **modelu widoku**).

> 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.

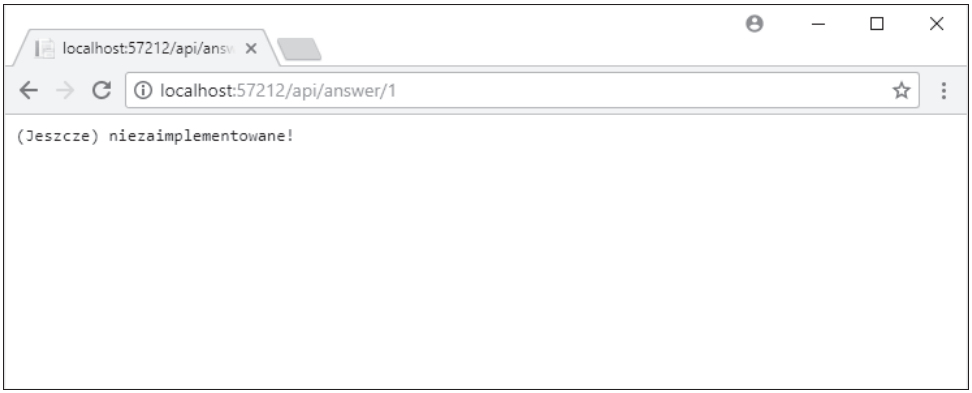
Aby sprawdzić nowe metody, wybierz z menu Visual Studio polecenie *Debugowanie/Rozpocznij debugowanie* (*Debug/Start Debugging*) lub naciśnij klawisz *F5*. Wpisz w przeglądarce poniższe adresy URL.

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

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



- 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

E

F

G

H

I

J

K

L

M

N

O

P

R

S

Ś

T

U

V

W

X

Z

Ż


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