

Nowoczesne API

Ewoluuujące aplikacje
sieciowe w technologii ASP.NET

ODKRYJ MOŻLIWOŚCI HTTP
NA NOWO!



Glenn Block, Pablo Cibraro, Pedro Felix,
Howard Dierking, Darrel Miller

Tytuł oryginału: Designing Evolvable Web APIs with ASP.NET

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-2391-9

© 2016 Helion S.A

Authorized Polish translation of the English edition of Designing Evolvable Web APIs with ASP.NET 9781449337711 © 2014 Glenn Block, Pablo Cibraro, Pedro Felix, Howard Dierking, and Darrel Miller

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

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

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

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

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

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Wprowadzenie	13
Wstęp	17
Dlaczego należy przeczytać tę książkę?	17
Co trzeba wiedzieć, aby jak najlepiej skorzystać z tej książki?	18
Jakie informacje zawiera ta książka?	19
Część I. Podstawy	19
Część II. Praktyczne programowanie API	20
Część III. Podstawowe zasady Web API	20
Konwencje zastosowane w książce	20
Użycie przykładowych kodów	21
Podziękowania	22
Rozdział 1. Internet, World Wide Web i HTTP	23
Architektura sieci	24
Zasób	25
URI	25
Cool URI	26
Reprezentacja	26
Typ danych	27
HTTP	29
Wykraczamy poza HTTP 1.1	29
Wymiana komunikatów HTTP	30
Pośredniki	32
Rodzaje pośredników	33
Metody HTTP	33
Nagłówki	36
Kody stanu HTTP	37
Negocjacja treści	38
Buforowanie	38
Uwierzytelnianie	42

Schematy uwierzytelniania	42
Dodatkowe schematy uwierzytelniania	43
Podsumowanie	44
Rozdział 2. Web API	45
Co to jest Web API?	45
Co w kwestii usług sieciowych SOAP?	45
Korzenie Web API	46
Początek rewolucji Web API	46
Zwrócenie uwagi na sieć	47
Wskazówki dla Web API	47
Typy danych charakterystyczne dla domeny	48
Profile typów danych	49
Wiele reprezentacji	50
Style API	51
Model dojrzałości Richardsona	52
RPC (poziom 0 w modelu dojrzałości Richardsona)	52
Zasoby (poziom 1 w modelu dojrzałości Richardsona)	54
Metody HTTP (poziom 2 w modelu dojrzałości Richardsona)	56
Pokonanie przepaści na drodze ku API zorientowanemu na zasoby	58
Hipermedia (poziom 3 w modelu dojrzałości Richardsona)	58
REST	63
Ograniczenia REST	63
Podsumowanie	65
Rozdział 3. Podstawy ASP.NET Web API	67
Najczęściej spotykane scenariusze	67
Pierwsza klasa programowania HTTP	68
Jednakowe możliwości podczas programowania po stronie klienta i serwera	69
Elastyczna obsługa różnych formatów	70
Nigdy więcej „tworzenia kodu z nawiasami ostrymi”	70
Możliwość stosowania testów jednostkowych	70
Wiele opcji w zakresie hostingu	71
Rozpoczęcie pracy z ASP.NET Web API	72
Nowy projekt Web API	75
Plik WebApiConfig.cs	76
Plik ValuesController.cs	78
„Witaj, Web API!”	79
Utworzenie usługi	79
Klient	86
Host	86
Podsumowanie	87

Rozdział 4. Architektura przetwarzania	89
Warstwa hostingu	91
Warstwa potoku procedur obsługi komunikatów	93
Obsługa tras	95
Warstwa obsługi kontrolera	96
Klasa bazowa ApiController	97
Podsumowanie	103
Rozdział 5. Aplikacja	105
Dlaczego ewolucja?	106
Bariery na drodze ewolucji	107
Jaki jest koszt zmiany?	108
Dlaczego po prostu nie wersja?	110
Przystępujemy do pracy	113
Cele aplikacji	113
Cele	113
Możliwości	114
Model informacji	114
Subdomeny	115
Powiązane zasoby	116
Grupy atrybutów	117
Kolekcje grup atrybutów	117
Model informacji kontra typ danych	118
Kolekcje zgłoszeń błędów	119
Modele zasobów	119
Zasób główny	119
Zasoby wyszukiwania danych	120
Kolekcja zasobów	120
Zasób elementu	120
Podsumowanie	123
Rozdział 6. Projekt i wybór typu danych	125
Samoopisywanie się	125
Rodzaje kontraktów	126
Typy danych	126
Proste formaty	126
Popularne formaty	128
Nowe formaty	129
Typy hipermediów	131
Eksplozja typów danych	132
Ogólne typy danych i profile	132
Inne typy hipermediów	136

Typy relacji łączy	137
Semantyka	137
Zastąpienie osadzonych zasobów	139
Warstwa pośrednia	139
Dane odwołania	140
Przepływ zdarzeń w aplikacji	141
Składnia	142
Doskonałe połączenie	145
Opracowanie kontraktu nowego typu danych	145
Wybór formatu	145
Włączenie obsługi hipermediów	146
Opcjonalne, obowiązkowe, pominięte, stosowalne	147
Metadane osadzone kontra zewnętrzne	148
Rozszerzalność	148
Rejestracja typu danych	150
Opracowanie nowych relacji łączy	151
Standardowe relacje łączy	151
Rozszerzenia relacji łączy	152
Osadzone relacje łączy	152
Rejestracja relacji łączy	153
Typy danych w domenie monitorowania zgłoszeń błędów	153
Lista zasobów	153
Zasób elementu	155
Zasób wykrycia	156
Zasób wyszukiwania	157
Podsumowanie	157

Rozdział 7. Tworzenie API 159

Projekt	159
Pobranie kodu źródłowego	160
Przygotowanie implementacji w stylu BDD	160
Nawigacja po rozwiązaniu	161
Pakiety i biblioteki	161
Samohostowanie	162
Modele i usługi	163
Zgłoszenie błędu i magazyn dla zgłoszeń błędów	163
Klasa IssueState	164
Klasa IssuesState	164
Klasa Link	166
Klasa IssueStateFactory	166
Klasa LinkFactory	167
Klasa IssueLinkFactory	168

Kryteria akceptacji	169
Funkcjonalność: pobieranie zgłoszeń błędów	172
Pobieranie zgłoszenia błędu	174
Pobieranie otwartych i zamkniętych zgłoszeń błędów	177
Pobieranie nieistniejącego zgłoszenia błędu	179
Pobieranie wszystkich zgłoszeń błędów	179
Pobieranie wszystkich zgłoszeń błędów jako danych w formacie Collection+Json	182
Zasoby wyszukiwania	184
Funkcjonalność: tworzenie zgłoszenia błędu	186
Funkcjonalność: uaktualnianie zgłoszenia błędu	188
Uaktualnianie zgłoszenia błędu	188
Uaktualnianie nieistniejącego zgłoszenia błędu	190
Funkcjonalność: usuwanie zgłoszenia błędu	191
Usuwanie zgłoszenia błędu	191
Usuwanie nieistniejącego zgłoszenia błędu	192
Funkcjonalność: przetwarzanie zgłoszenia błędu	193
Testy	193
Implementacja	194
Podsumowanie	195
Rozdział 8. Usprawnianie API	197
Kryteria akceptacji dla nowych funkcjonalności	197
Implementacja obsługi buforowania danych wyjściowych	198
Dodanie testów do sprawdzenia buforowania danych wyjściowych	200
Implementacja ponownego pobierania buforowanych danych	202
Implementacja warunkowych żądań GET	
do obsługi ponownego pobierania buforowanych danych	203
Wykrywanie konfliktów	206
Implementacja wykrywania konfliktów	206
Audyt zmiany	209
Implementacja audytu zmian za pomocą uwierzytelniania Hawk	210
Monitorowanie	213
Implementacja monitorowania	214
Podsumowanie	216
Rozdział 9. Tworzenie klienta	217
Biblioteki klienta	218
Biblioteki opakowujące	218
Łącza jako funkcje	222
Przebieg działania aplikacji	227
Warto wiedzieć	228

Klienty z misją	232
Stan klienta	234
Podsumowanie	235
Rozdział 10. Model programowania HTTP	237
Komunikaty	238
Nagłówki	242
Zawartość komunikatu	247
Wykorzystanie zawartości komunikatu	248
Tworzenie zawartości komunikatu	250
Podsumowanie	257
Rozdział 11. Hosting	259
Hosting WWW	260
Infrastruktura ASP.NET	260
Routing ASP.NET	262
Routing Web API	264
Konfiguracja globalna	266
Procedura obsługi Web API ASP.NET	268
Samohostowanie	270
Architektura WCF	271
Klasa HttpSelfHostServer	272
Klasa SelfHostConfiguration	273
Rezerwacja adresu URL i kontrola dostępu	275
Hosting Web API z użyciem OWIN i Katana	275
OWIN	276
Projekt Katana	277
Konfiguracja Web API	280
Oprogramowanie pośredniczące Web API	281
Środowisko OWIN	283
Hosting w pamięci	284
Hosting Azure Service Bus	284
Podsumowanie	290
Rozdział 12. Kontrolery i routing	291
Ogólny opis przepływu komunikatów HTTP	291
Potok procedur obsługi komunikatów	292
Dyspozytor	296
HttpControllerDispatcher	297
Wybór kontrolera	297
Aktywacja kontrolera	301

Potok kontrolera	302
ApiController	302
Model przetwarzania ApiController	302
Podsumowanie	314
Rozdział 13. Formatery i dołączanie modelu	315
Waga modeli w ASP.NET Web API	315
Jak działa dołączanie modelu?	316
Wbudowane bindery modelu	319
Implementacja ModelBindingParameterBinder	320
Dostawcy wartości	320
Bindery modelu	323
Dołączanie modelu tylko dla adresu URI	325
Implementacja FormatterParameterBinder	326
Domyślny wybór HttpParameterBinding	331
Sprawdzanie poprawności modelu	331
Zastosowanie w modelu atrybutów adnotacji danych	331
Przeglądanie wyników operacji sprawdzania poprawności	332
Podsumowanie	334
Rozdział 14. HttpClient	335
Klasa HttpClient	335
Cykl życiowy	335
Opakowanie	336
Wiele egzemplarzy	336
Bezpieczeństwo wątków	337
Metody pomocnicze	337
Zagłębiaamy się w kolejne warstwy	337
Ukończone żądania nie zgłaszają wyjątków	338
Zawartość jest wszystkim	338
Przerwanie na żądanie	339
Metoda SendAsync()	340
Procedury obsługi komunikatów klienta	341
Proxy dla procedur obsługi	342
Nieprawdziwe procedury obsługi odpowiedzi	343
Tworzenie wielokrotnego użytku procedur obsługi odpowiedzi	344
Podsumowanie	346
Rozdział 15. Bezpieczeństwo	347
Zapewnienie bezpieczeństwa transportu	347
Użycie TLS w ASP.NET Web API	349
Użycie TLS z hostingiem IIS	349
Użycie TLS z samohostowaniem	351

Uwierzytelnianie	351
Model oświadczeń	352
Pobieranie i przypisywanie aktualnego zleceniodawcy	356
Uwierzytelnienie oparte na transporcie	357
Uwierzytelnienie serwera	357
Uwierzytelnienie klienta	361
Framework uwierzytelniania HTTP	367
Implementacja uwierzytelniania opartego na HTTP	369
Katana, czyli oprogramowanie pośredniczące do uwierzytelniania	370
Aktywne i pasywne oprogramowanie pośredniczące odpowiedzialne za uwierzytelnianie	374
Filtry uwierzytelniania w Web API	375
Uwierzytelnianie oparte na tokenie	378
Schemat uwierzytelniania Hawk	385
Autoryzacja	386
Egzekwowanie autoryzacji	388
Współdzielenie zasobów między serwerami w różnych domenach	391
Obsługa mechanizmu CORS na platformie ASP.NET Web API	393
Podsumowanie	396
Rozdział 16. OAuth 2.0, czyli framework uwierzytelniania	397
Aplikacje klienta	399
Uzyskanie dostępu do chronionych zasobów	401
Pobranie tokenu dostępu	402
Uprawnienia kodu autoryzacji	404
Zakres	406
Kanał oficjalny kontra kanał nieoficjalny	407
Token refresh	409
Serwer zasobów i serwer autoryzacji	410
Przetwarzanie tokenów dostępu w ASP.NET Web API	411
OAuth 2.0 i uwierzytelnianie	413
Autoryzacja na podstawie zakresu	416
Podsumowanie	417
Rozdział 17. Testowanie	419
Testy jednostkowe	419
Frameworki testów jednostkowych	420
Rozpoczęcie pracy z testami jednostkowymi w Visual Studio	420
xUnit.NET	422
Rola testów jednostkowych w programowaniu TDD	423

Testy jednostkowe implementacji ASP.NET Web API	427
Testy jednostkowe ApiController	427
Testy jednostkowe MediaTypeFormatter	433
Testy jednostkowe HttpResponseMessage	436
Testy jednostkowe ActionFilterAttribute	437
Testy jednostkowe tras	440
Testy integracji w ASP.NET Web API	442
Podsumowanie	443
Dodatek A. Typy danych	445
Dodatek B. Nagłówki HTTP	447
Dodatek C. Negocjacja treści	451
Negocjacja proaktywna	451
Negocjacja reaktywna	452
Dodatek D. Buforowanie w działaniu	455
Dodatek E. Przepływ zdarzeń podczas uwierzytelniania	459
Dodatek F. Specyfikacja typu danych dla application/issue+json	463
Konwencje nazw	463
Dokument zgłoszenia błędu	463
Kwestie bezpieczeństwa	464
Kwestie interoperacyjności	464
Kwestie związane z IANA	464
Dodatek G. Certyfikaty i kryptografia klucza publicznego	465
Cofnięcie certyfikatu	471
Tworzenie testowych kluczy i certyfikatów	471
Skorowidz	475

Podstawy ASP.NET Web API

Łatwiej wyznaczyć kurs, gdy masz pod ręką mapę.

Skoro dostarczyliśmy kontekst i wyjaśniliśmy, dlaczego Web API ma tak duże znaczenie dla nowoczesnych aplikacji sieciowych, w tym rozdziale przejdziemy do pracy z ASP.NET Web API. Platforma ASP.NET Web API i nowy model programowania HTTP oferują różne możliwości w zakresie zarówno budowy, jak i wykorzystywania Web API. Na początku przedstawimy pewne najważniejsze cele Web API i sposoby umożliwiające ich osiągnięcie. Następnie, analizując model programistyczny, przekonasz się, jak te możliwości są udostępniane dla Twojego kodu w ASP.NET Web API. Czy może być tutaj lepszy sposób niż przeanalizowanie kodu dostarczanego przez szablon projektu Web API w Visual Studio? Na koniec wykroczymy poza domyślny kod szablonu i przygotowujemy nasze pierwsze Web API typu „Witaj, świecie!”.

Najczęściej spotykane scenariusze

ASP.NET Web API, w przeciwieństwie do wielu technologii, ma doskonale udokumentowaną i dostępną historię (fragment znajdziesz w serwisie *CodePlex*¹). Już na samym początku zespół odpowiedzialny za rozwój platformy podjął decyzję o zachowaniu maksymalnej możliwej przejrzystości, aby wpływ na powstawanie produktu mogła mieć społeczność ekspertów, którzy ostatecznie będą korzystać z tego produktu do budowy rzeczywistych systemów. Oto podstawowe cele, których spełnienie stało za utworzeniem ASP.NET:

- Pierwsza klasa programowania HTTP.
- Jednakowe możliwości podczas programowania po stronie klienta i serwera.
- Elastyczna obsługa różnych formatów.
- Nigdy więcej „tworzenia kodu z nawiasami ostrymi”.
- Możliwość stosowania testów jednostkowych.
- Wiele opcji w zakresie hostingu.

¹ <http://wcf.codeplex.com/>

To tylko najważniejsze cele, a więc nie jest to pełna lista wszystkich możliwości, jakie oferuje framework. ASP.NET Web API pozwala uzyskać najlepsze połączenie różnych technologii: WCF (*windows communication foundation*) z jego nieskończenie elastyczną architekturą, obsługą klienta i elastycznym modelem hostingu oraz ASP.NET MVC z oferowaną przez tę platformę obsługą konwencji zamiast konfiguracji, poprawionymi możliwościami w zakresie przeprowadzania testów i funkcjami zaawansowanymi, takimi jak dołączanie modelu i weryfikacja. Jak się przekonasz podczas lektury niniejszego rozdziału, w wyniku tego połączenia powstał framework, z którym łatwo rozpocząć pracę i który można bez problemu dostosować do własnych potrzeb w miarę ich ewolucji.

Pierwsza klasa programowania HTTP

Kiedy budujesz nowoczesne Web API — zwłaszcza przeznaczone dla prostszych klientów, takich jak urządzenia mobilne — sukces tego API jest bardzo często powiązany z jego ekspresyjnością. Z kolei ekspresyjność Web API zależy od tego, jak dobrze radzi sobie z użyciem HTTP jako protokołu aplikacji. Wykorzystanie HTTP w charakterze protokołu aplikacji wykracza poza prostą obsługę żądań HTTP i generowanie odpowiedzi HTTP. Oznacza, że zachowanie zarówno aplikacji, jak i stojącego za nią frameworku jest nadzorowane przez kontrolę przepływu zdarzeń w HTTP i elementy danych, a nie przez pewne dane dodatkowe, które są jedynie (i przypadkowo) przekazywane za pomocą HTTP. Spójrz na przykład na poniższe żądanie SOAP użyte do komunikacji z usługą WCF:

```
POST http://localhost/GreetingService.svc HTTP/1.1
Content-Type: text/xml; charset=utf-8
SOAPAction: "HelloWorld"
Content-Length: 154

<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
    <HelloWorld xmlns="http://localhost/wcf/greeting"/>
  </s:Body>
</s:Envelope>
```

W powyższym przykładzie klient wykonuje żądanie do serwera w celu otrzymania przyjaznego komunikatu powitalnego. Jak widzisz, żądanie zostało wysłane za pomocą HTTP. Jednak w tym miejscu tak naprawdę kończy się powiązanie z protokołem HTTP. Zamiast wykorzystać metody HTTP (czasami określane mianem *verbs*) do przedstawienia natury akcji żądanej usługi, przyjęte tutaj podejście polega na wysyłaniu wszystkich żądań za pomocą tej samej metody HTTP POST i umieszczeniu szczegółów charakterystycznych dla aplikacji zarówno w zawartości żądania HTTP, jak i w niestandardowym nagłówku SOAPAction. Jak zapewne się spodziewasz, ten sam wzorzec został powtórzony w odpowiedzi wygenerowanej przez tę usługę:

```
HTTP/1.1 200 OK
Content-Length: 984
Content-Type: text/xml; charset=utf-8
Date: Tue, 26 Apr 2011 01:22:53 GMT
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
    <HelloWorldResponse xmlns="http://localhost/wcf/greeting">
      ...
    </HelloWorldResponse>
  </s:Body>
</s:Envelope>
```

Podobnie jak w przypadku komunikatu żądania, elementy protokołu użyte do kontrolowania aplikacji — to znaczy określenia sposobu, w jaki aplikacje klienta i serwer mogą się wzajemnie rozumieć — zostały wyodrębnione z elementów HTTP i umieszczone w znacznikach XML części głównej odpowiednio żądania i odpowiedzi.

W tego rodzaju podejściu HTTP nie jest używany do wyrażenia protokołu aplikacji, ale raczej po prostu jako sposób transportu dla innego protokołu aplikacji — tutaj SOAP. Wprowadzie takie rozwiązanie może być dobre w sytuacji, gdzie pojedyncza usługa musi się komunikować z podobnymi klientami za pomocą wielu różnych protokołów, to jednak stanie się problematyczne, gdy zajdzie potrzeba komunikacji z wieloma różnymi klientami za pomocą pojedynczego protokołu. Problemy te są doskonale zilustrowane w przypadku Web API, gdzie różnorodność nie dotyczy jedynie klientów, ale także infrastruktury komunikacyjnej między klientami i usługami (na przykład internet), a ponadto ta różnorodność jest ogromna i ciągle się zmienia. W takim świecie należy dążyć do optymalizacji klientów i usług nie pod względem niezależności od protokołu, lecz raczej do zapewnienia pierwszorzędного wrażenia dotyczącego najczęściej stosowanego protokołu aplikacji. W przypadku aplikacji komunikujących się przez sieć tym protokołem jest HTTP.

Można powiedzieć, że ASP.NET Web API zbudowano na podstawie niewielkiego zbioru prostych obiektów HTTP. Dwa najważniejsze z nich to `HttpRequestMessage` i `HttpResponseMessage`. Przeznaczeniem tych obiektów jest dostarczenie silnie typowanego widoku rzeczywistego komunikatu HTTP. Spójrz na przedstawiony poniżej komunikat żądania HTTP:

```
GET http://localhost:50650/api/greeting HTTP/1.1
Host: localhost:50650
accept: application/json
if-none-match: "1"
```

Przyjmując założenie, że pokazane żądanie zostało otrzymane przez usługę ASP.NET Web API, możemy uzyskać dostęp do różnych elementów żądania i operować nimi za pomocą przedstawionego poniżej kodu w klasie kontrolera Web API:

```
var request = this.Request;
var requestedUri = request.RequestUri;
var requestedHost = request.Headers.Host;
var acceptHeaders = request.Headers.Accept;
var conditionalValue = request.Headers.IfNoneMatch;
```

Ten silnie typowany model przedstawia poprawny poziom abstrakcji poprzez HTTP, pozwala programiście na bezpośrednią pracę z żądaniem lub odpowiedzią HTTP i jednocześnie odciąża go od konieczności zajmowania się niskiego poziomu kwestiami, takimi jak analiza nieprzetworzonego komunikatu lub jego generowanie.

Jednakowe możliwości podczas programowania po stronie klienta i serwera

Jednym z najbardziej kuszących aspektów związanych ze zbudowaniem ASP.NET Web API na podstawie biblioteki opartej na HTTP jest to, że biblioteka ta może być używana nie tylko przez serwer, ale również w aplikacjach klienta utworzonych z wykorzystaniem platformy .NET. To oznacza, że przedstawione wcześniej żądanie HTTP można utworzyć za pomocą tych samych klas modelu programowania HTTP, które ostatecznie będą używane do pracy z żądaniem wewnątrz Web API, co pokażemy w dalszej części rozdziału.

Jak się przekonasz w rozdziale 10., model programowania HTTP to znacznie więcej niż operowanie różnymi elementami danych w żądaniach i odpowiedziach. Funkcje takie jak procedury obsługi komunikatów i negocjacja treści zostały bezpośrednio wbudowane w model programowania HTTP. To umożliwia programistom wykorzystanie ich po stronie zarówno serwera, jak i klienta w celu opracowania zaawansowanych interakcji klient – serwer przy jednoczesnej maksymalizacji wielokrotnego użycia tego samego kodu.

Elastyczna obsługa różnych formatów

Szczegółowe omówienie negocjacji treści znajdziesz w rozdziale 13. W tym miejscu powinieneś wiedzieć, że na wysokim poziomie to jest proces, w którym klient i serwer współpracują ze sobą w celu określenia odpowiedniego formatu, który będzie używany podczas wymiany reprezentacji przez HTTP. Istnieje kilka różnych podejść i technik stosowanych podczas negocjacji treści. Domyślnie ASP.NET Web API obsługuje podejście oparte na serwerze, używając nagłówka HTTP Accept, aby pozwolić klientowi na wybór między formatami XML i JSON. W przypadku braku nagłówka Accept ASP.NET Web API będzie domyślnie zwracać dane w formacie JSON (podobnie jak wiele innych aspektów frameworku, także to zachowanie domyślne można zmienić).

Na przykład spójrz na poniższe żądanie do usługi ASP.NET Web API:

```
GET http://localhost:50650/api/greeting HTTP/1.1
```

Ponieważ żądanie to nie zawiera nagłówka Accept wskazującego serwerowi preferowany format danych, serwer zwróci dane w formacie JSON. To zachowanie można zmienić przez umieszczenie w żądaniu nagłówka Accept i wskazanie, że preferowany typ danych to XML²:

```
GET http://localhost:50650/api/greeting HTTP/1.1
accept: application/xml
```

Nigdy więcej „tworzenia kodu z nawiasami ostrymi”

Wraz z dojrzewaniem frameworku .NET jedno z najczęściej powtarzających się zastrzeżeń ze strony programistów dotyczyło ilości kodu konfiguracyjnego XML koniecznego do stworzenia wydawałoby się prostych lub nawet domyślnych scenariuszy. Co gorsza, ponieważ konfiguracja kontrolowała aspekty takie jak typy wczytywane podczas uruchamiania aplikacji, zmiana konfiguracji mogła prowadzić do błędów w systemie niewychwytywanych przez kompilator, a ujawniających się dopiero po uruchomieniu aplikacji. Jeden z największych przykładów takich narzekań wiąże się z poprzednikiem ASP.NET Web API, czyli WCF. Wprawdzie w samym WCF znacznie poprawiono kwestię konfiguracji i ilości wymaganego przez nią kodu, ale zespół odpowiedzialny za rozwój ASP.NET Web API poszedł w zupełnie innym kierunku i wprowadził tryb konfiguracji całkowicie oparty na kodzie. Szczegółowe omówienie konfiguracji ASP.NET Web API znajdziesz w rozdziale 11.

² Utrzymywany przez IANA katalog publicznych typów danych znajdziesz tutaj: <http://www.iana.org/assignments/media-types/media-types.xhtml>.

Możliwość stosowania testów jednostkowych

Wraz z coraz większą popularnością technik takich jak **TDD** (*test-driven development*) i **BDD** (*behavior-driven development*) nastąpił także proporcjonalny wzrost frustracji związanej z faktem, że wiele popularnych usług i frameworków sieciowych nadal używa statycznych obiektów kontekstu, zamkniętych typów i obszernych drzew dziedziczenia. Te techniki znacznie utrudniają przeprowadzanie testów jednostkowych we wspomnianych obiektach, ponieważ trudno jest je utworzyć i zainicjalizować poza środowiskiem uruchomieniowym. Na dodatek bardzo trudno jest je zastąpić „imitacjami”, które mogłyby zapewnić lepszą izolację testu.

Na przykład ASP.NET bardzo mocno opiera się na obiekcie `HttpContext`, podczas gdy w przypadku WCF podstawą jest obiekt `OperationContext` (lub `WebOperationContext`, w zależności od rodzaju usługi). Podstawowy problem ze statycznymi obiektami kontekstu takimi jak wymienione polega na tym, że są one definiowane i używane przez środowisko uruchomieniowe frameworku. Dlatego też przetestowanie usługi opracowanej z użyciem tych obiektów kontekstu w rzeczywistości wymaga uruchomienia hosta usługi i udostępnienia tej usługi. Wprawdzie tego rodzaju techniki są ogólnie rzecz biorąc akceptowane podczas testów integracji, ale okazują się nieodpowiednie do stylu programowania takiego jak TDD, który opiera się na możliwości szybkiego wykonywania małych testów jednostkowych.

Jednym z celów w ASP.NET Web API jest znaczna poprawa obsługi wspomnianych powyżej stylów programowania. Mamy dwie cechy charakterystyczne frameworku pomagające w osiągnięciu tego celu. Pierwsza: ASP.NET Web API stosuje ten sam model programowania jak w przypadku frameworku MVC. Tym samym dostępne od kilku lat możliwości w zakresie przeprowadzania testów stają się dostępne także w ASP.NET Web API. Programiści mogą więc uniknąć konieczności użycia statycznych obiektów kontekstu i opakowań, aby egzemplarze „imitacji” mogły być wykorzystywane w testach jednostkowych.

Druga: pamiętaj o zbudowaniu ASP.NET Web API na podstawie modelu programowania HTTP. W tego rodzaju modelu obiekty są prostymi strukturami danych, które mogą być tworzone, konfigurowane, przekazywane metodzie akcji jako parametr i analizowane po ich otrzymaniu w odpowiedzi. Zyskujemy więc możliwość opracowywania testów jednostkowych w jasny i konkretny sposób. Wraz z ewolucją ASP.NET Web API testowanie stało się ważnym obszarem zainteresowania twórców frameworku, co znajduje odzwierciedlenie w klasie `HttpRequestContext` w Web API 2. Dokładne omówienie tematu testowania znajdziesz w rozdziale 17.

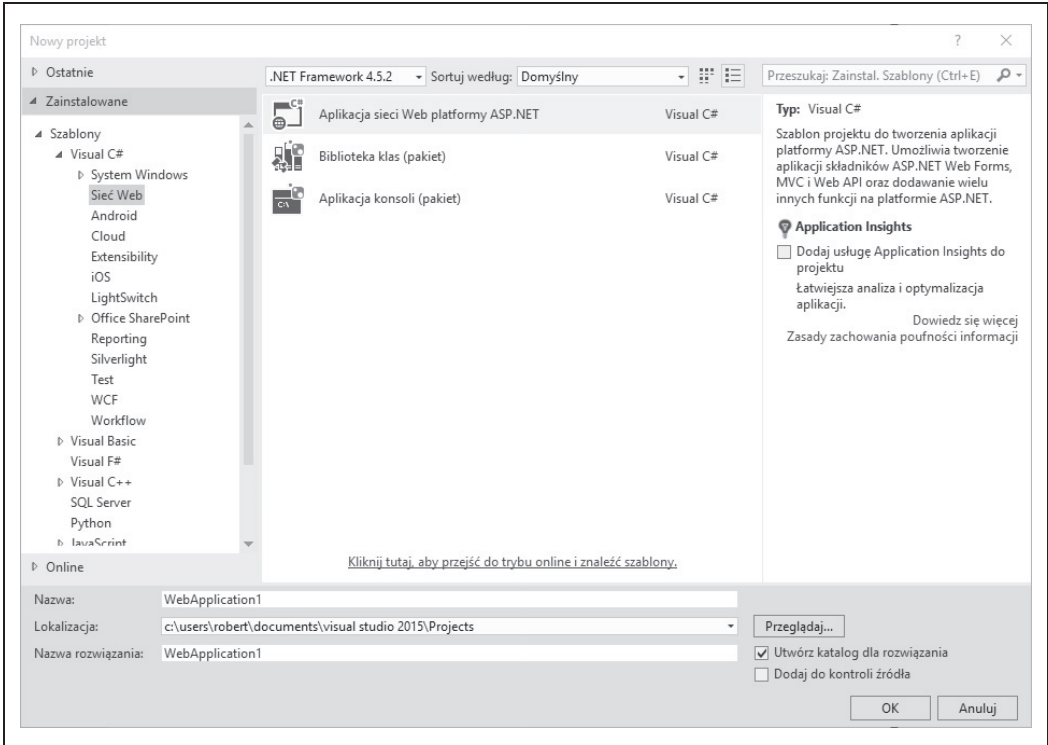
Wiele opcji w zakresie hostingu

Pomimo ogólnie mniejszych możliwości jednym z największych atrybutów WCF była możliwość „samohostowania”, czyli uruchomienia w dowolnym procesie, na przykład w usłudze Windows, aplikacji działającej w konsoli lub w serwerze **IIS** (*internet information services*). Tak naprawdę ta ogromna elastyczność w zakresie hostingu powodowała, że ograniczenia w zakresie testów jednostkowych były do zniesienia... prawie.

Podczas konsolidacji WCF Web API z ASP.NET w celu utworzenia ASP.NET Web API zespół odpowiedzialny za platformę chciał zachować ową możliwość samohostowania. Dlatego usługi ASP.NET Web API, podobnie jak usługi WCF, mogą być uruchamiane w dowolnym procesie. Tematem hostingu zajmiemy się w rozdziale 11.

Rozpoczęcie pracy z ASP.NET Web API

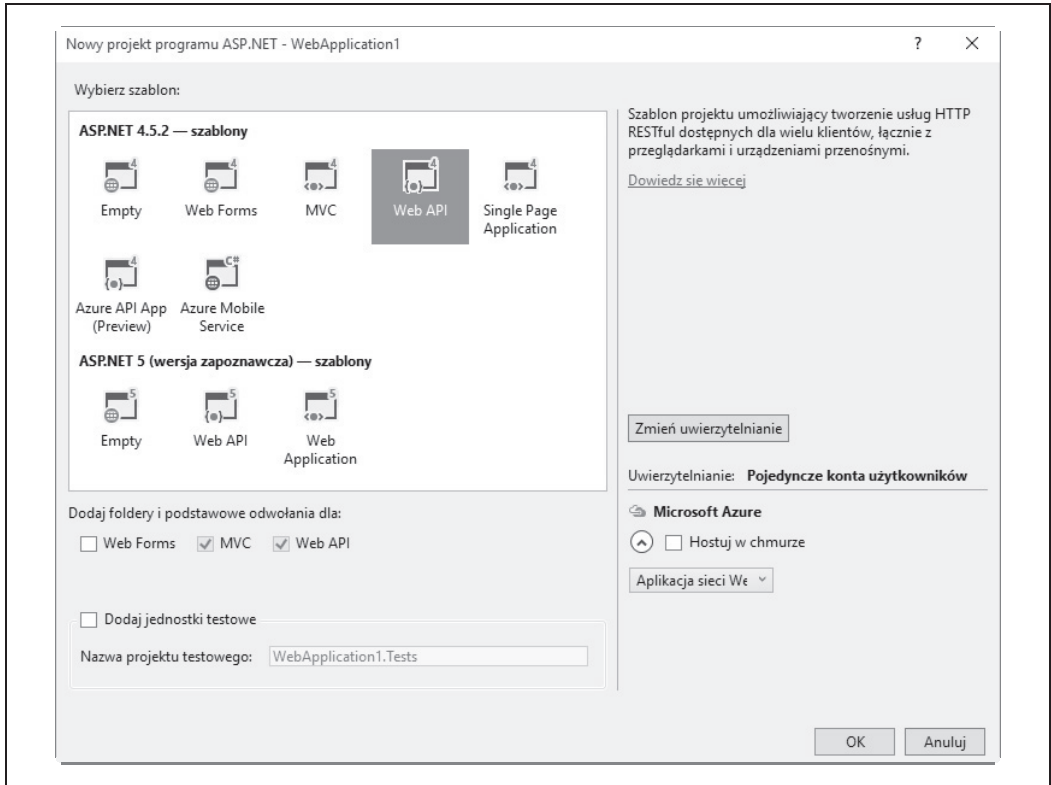
Po omówieniu celów kryjących się za opracowaniem ASP.NET Web API możemy przystąpić do zapoznawania się z wybranymi elementami, z którymi będziemy pracować podczas tworzenia własnych Web API. Jedno z najprostszych podejść polega na utworzeniu zupełnie nowego projektu ASP.NET Web API i przeanalizowaniu kodu dostarczanego przez szablon projektu. Aby utworzyć nowy projekt ASP.NET Web API, po uruchomieniu Visual Studio przejdź do kategorii *Visual C#/Sieć Web* w oknie dialogowym *Nowy projekt*, a następnie wybierz *Aplikacja sieci Web platformy ASP.NET* (rysunek 3.1).



Rysunek 3.1. Projekt aplikacji ASP.NET MVC w oknie dialogowym *Nowy projekt* wyświetlonym w *Visual Studio 2015*

Po kliknięciu przycisku *OK* zostanie wyświetlone kolejne okno dialogowe, w którym będzie można wybrać odpowiednią konfigurację projektu. W omawianym przykładzie wybierz utworzenie projektu *Web API* (rysunek 3.2).

Zwróć uwagę na fakt, że *Web API* to po prostu odmiana szablonu projektu z rodziny projektów ASP.NET. Oznacza to, że projekty *Web API* współdzielą te same podstawowe komponenty, jak w przypadku innych typów projektów sieciowych. Różnica polega jedynie na plikach generowanych przez szablon w chwili tworzenia projektu początkowego. Dlatego jest zarówno prawidłowe, jak i możliwe dołączanie *Web API* w innym dowolnym szablonie pokazanym na rysunku 3.2.



Rysunek 3.2. Wybór Web API jako typu dla tworzonej aplikacji ASP.NET MVC

Tak naprawdę ASP.NET Web API to po prostu zbiór klas utworzonych na podstawie komponentów frameworku Web API i hostowanych przez proces, w którym znajduje się środowisko uruchomieniowe ASP.NET. To może być ustawienie domyślne lub wskazany przez programistę niestandardowy host (do tego tematu jeszcze powrócimy w dalszej części rozdziału). Web API można więc dołączyć do dowolnego typu projektu, niezależnie od tego, czy to będzie projekt MVC, aplikacja działająca w konsoli czy nawet klasa biblioteki, do której odwołania będą pochodzić z wielu innych projektów.

Komponenty frameworku ASP.NET są dostępne dla projektów Web API za pomocą *aplikacji przeznaczonej do zarządzania pakietami NuGet*³. Pakiety NuGet wymienione w tabeli 3.1 są instalowane w domyślnym szablonie projektu. Jeżeli obsługę Web API chcesz dodać do innego projektu, musisz się po prostu upewnić, czy zainstalowałeś pakiety, które dostarczają wymaganą funkcjonalność.

³ <http://docs.nuget.org/>

Tabela 3.1. Pakiety NuGet dla ASP.NET Web API

Nazwa pakietu	Identyfikator pakietu ⁴	Opis	Zależności pakietu ⁵
Biblioteki klienta HTTP na platformie Microsoft .NET Framework 4	Microsoft.Net.Http	Zapewnia podstawowy model programowania HTTP, łącznie z obsługą obiektów HttpRequest ↳Message i HttpResponseMessage.	brak
Microsoft ASP.NET Web API	Microsoft.AspNet. ↳WebApi	Metapakiet ⁶ NuGet zapewnia pojedyncze odwołanie do instalacji wszystkiego, co jest potrzebne do tworzenia i hostingu Web API w ASP.NET.	Microsoft.AspNet. ↳WebApi.WebHost
Biblioteki klienta Microsoft ASP.NET Web API	Microsoft.AspNet. ↳WebApi.Client	Zawiera rozszerzenia dla podstawowych bibliotek klienta HTTP w NET Framework 4. Rozszerzenia te dostarczają obsługę funkcji takich jak formatowanie XML i JSON, a także możliwość przeprowadzania negocjacji treści.	Microsoft.Net. ↳HttpNewtonsoft.Json ⁷
Biblioteki podstawowe Microsoft ASP.NET Web API	Microsoft.AspNet. ↳WebApi.Core	Zawiera podstawowy model programowania Web API i komponenty środowiska uruchomieniowego, między innymi kluczową klasę ApiController.	Microsoft.AspNet. ↳WebApi.Client
Microsoft ASP.NET Web API Web Host	Microsoft.AspNet. ↳WebApi.WebHost	Zawiera wszystkie komponenty środowiska uruchomieniowego niezbędne do hostingu Web API w trakcie działania aplikacji ASP.NET.	Microsoft.Web. ↳Infrastructure Microsoft.AspNet. ↳WebApi.Core

Poza pakietami NuGet instalowanymi jako część domyślnych szablonów projektów dostępne są jeszcze inne pakiety NuGet, wymienione w tabeli 3.2.

Tabela 3.2. Dodatkowe pakiety NuGet dostępne dla ASP.NET Web API

Nazwa pakietu	Identyfikator pakietu	Opis	Zależność pakietu
Samohostowanie Microsoft ASP.NET Web API	Microsoft.AspNet.WebApi. ↳SelfHost	Zawiera wszystkie komponenty środowiska uruchomieniowego niezbędne do hostingu Web API w procesie klienta (na przykład aplikacji konsoli).	Microsoft.AspNet. ↳WebApi.Core
Microsoft ASP.NET Web API OWIN	Microsoft.AspNet.WebApi. ↳Owin	Umożliwia hosting ASP.NET Web API w obrębie serwera OWIN i dostęp do dodatkowych elementów serwera OWIN.	Microsoft.AspNet. ↳WebApi.Core, Microsoft.Owin, Owin

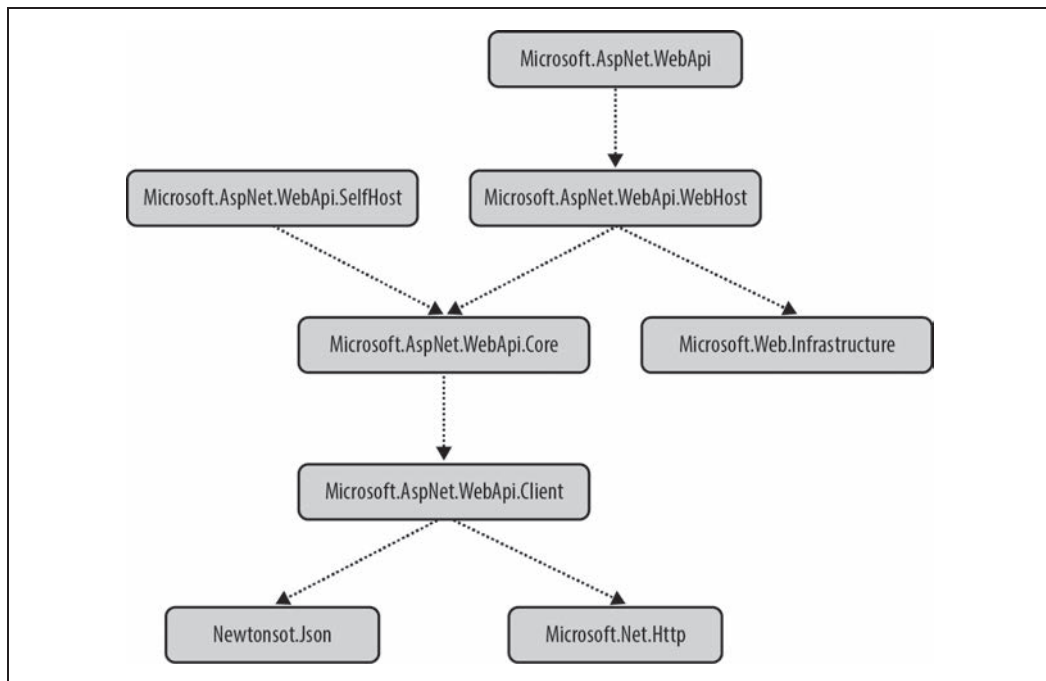
⁴ Identyfikator pakietu można wykorzystać do uzyskania dokładniejszych informacji o danym pakiecie. W tym celu identyfikator należy dołączyć do adresu URL.

⁵ Zależność pakietu NuGet oznacza, że podczas instalacji danego pakietu menedżer NuGet w pierwszej kolejności próbuje zainstalować wszystkie pakiety, na których opiera się działanie danego pakietu.

⁶ Metapakiet NuGet to pakiet pozbawiony rzeczywistej zawartości i mający jedynie zależności względem innych pakietów NuGet.

⁷ Wprowadzie używany w ASP.NET Web API, ale NewtonsoftJson to zewnętrzny komponent, który można pobrać bezpłatnie (<http://www.nuget.org/packages/newtonsoft.json>).

Przedstawienie zbioru pakietów NuGet w postaci graficznej może pomóc w jeszcze łatwiejszym ustaleniu, które pakiety należy zainstalować w projekcie w zależności od jego celów. Przykład znajdziesz na rysunku 3.3.

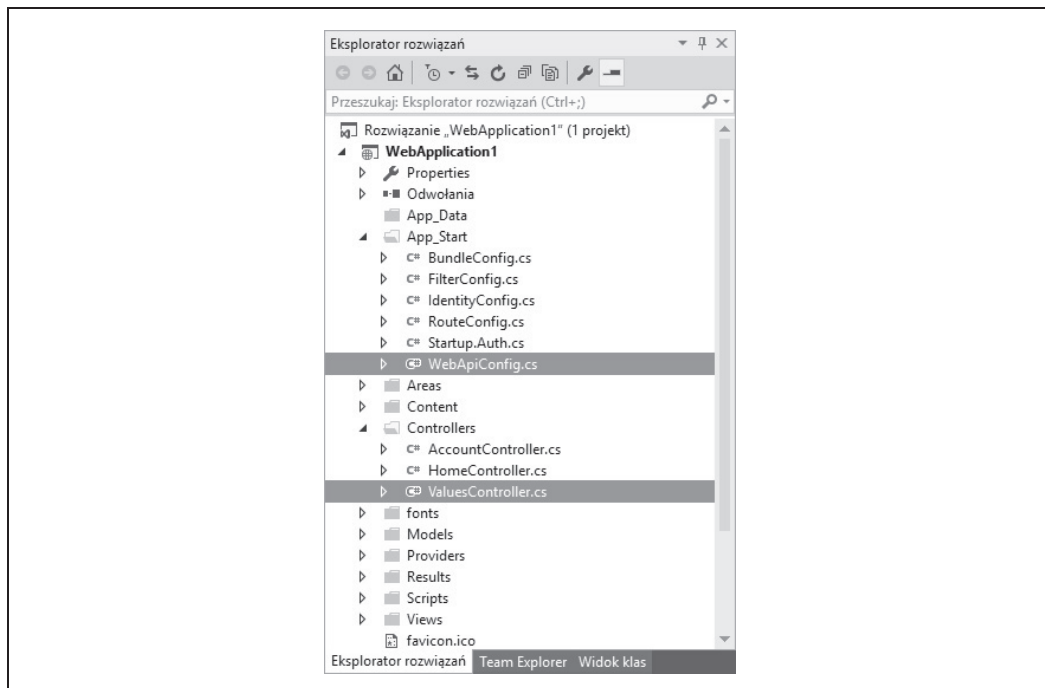


Rysunek 3.3. Hierarchia pakietów NuGet dla Web API

Jak widać na rysunku 3.3, instalacja dowolnego z tych pakietów NuGet automatycznie spowoduje instalację wszystkich bezpośrednio lub pośrednio powiązanych z nim pakietów. Na przykład `Microsoft.AspNet.WebApi` spowoduje instalację pakietów `Microsoft.AspNet.WebApi.WebHost`, `Microsoft.AspNet.WebApi.Core`, `Microsoft.Web.Infrastructure`, `Microsoft.AspNet.WebApi.Client`, `Newtonsoft.Json` i `Microsoft.Net.Http`.

Nowy projekt Web API

Po utworzeniu nowego projektu ASP.NET Web API warto przeanalizować kluczowe komponenty wygenerowane przez szablon projektu, które później dostosujemy do własnych potrzeb w celu utworzenia Web API. Skoncentrujemy się tutaj na dwóch ważnych plikach: `WebApiConfig.cs` i `ValuesController.cs` (rysunek 3.4).



Rysunek 3.4. Pliki `WebApiConfig.cs` i `ValuesController.cs` zaznaczone w oknie Eksploratora rozwiązania w Visual Studio 2015

Plik `WebApiConfig.cs`

Plik C# lub Visual Basic.NET znajduje się w katalogu `App_Start` i deklaruje klasę `WebApiConfig`. Ta klasa zawiera pojedynczą metodę o nazwie `Register()` i jest wywoływana przez kod metody `Application_Start()` w pliku `global.asax`. Jak sama nazwa wskazuje, klasa jest odpowiedzialna za rejestrację różnych aspektów konfiguracji Web API. Domyślnie podstawowy kod konfiguracyjny wygenerowany przez szablon projektu powoduje zarejestrowanie domyślnej trasy Web API. Trasa ta jest używana do mapowania przychodzących żądań HTTP na klasy kontrolerów, a także do przetwarzania elementów danych, które mogą być wysyłane jako część adresów URL i udostępniane innym klasom w potoku przetwarzania. Kod domyślnej klasy `WebApiConfig` przedstawiliśmy w listingu 3.1.

Listing 3.1. Kod domyślnej klasy `WebApiConfig`

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        // Konfiguracja i usługi składnika Web API
        // Skonfiguruj składnik Web API, aby korzystał tylko z uwierzytelniania za pomocą tokenów bearer
        config.SuppressDefaultHostAuthentication();
        config.Filters.Add(new
            HostAuthenticationFilter(0AuthDefaults.AuthenticationType));

        // Trasy składnika Web API
        config.MapHttpAttributeRoutes();
    }
}
```

```

        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );
    }
}

```

Jeżeli programowanie w stylu MVC nie jest Ci obce, wówczas możesz dostrzec, że ASP.NET Web API oferuje odmienny zbiór metod rozszerzających do zarejestrowania swoich tras innych niż domyślne trasy MVC. Na przykład ten sam nowy projekt zawierający klasę `WebApiConfig` zawiera także klasę `RouteConfig`, której kod przedstawia się następująco:

```

public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

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

```

Istnienie dwóch metod przeznaczonych do konfiguracji tras może się na początku wydawać niepotrzebne i dlatego też warto omówić występujące między nimi różnice na wysokim poziomie. Trzeba pamiętać, że te metody „mapujące” to po prostu metody rozszerzające odpowiedzialne za utworzenie egzemplarza trasy i jego dodanie do kolekcji tras powiązanej z hostem. Różnica między tymi metodami i zarazem powód istnienia obu sprowadza się do faktu, że ASP.NET MVC i ASP.NET Web API używają zupełnie różnych klas tras, a nawet typów kolekcji tras. Szczegóły dotyczące wspomnianych typów omówimy bardzo dokładnie w rozdziale 11. Powodem zastosowania innego typu tras niż w ASP.NET MVC było umożliwienie ASP.NET Web API odcięcie się od starego kodu znajdującego się w klasach `Route` i `RouteCollection` w podzespole `System.Web`, a tym samym zapewnienie znacznie większej elastyczności pod względem opcji dotyczących hostingu. Pośrednią korzyścią wynikającą z tej decyzji projektowej jest dostarczenie ASP.NET Web API możliwości samohostowania.

Konfiguracja routingu ASP.NET Web API wymaga zadeklarowania egzemplarzy `HttpRoute` i dodania ich do kolekcji tras. Pomimo tworzenia egzemplarzy `HttpRoute` za pomocą innej metody rozszerzającej niż w przypadku ASP.NET MVC semantyka pozostała praktycznie identyczna, między innymi w zakresie elementów takich jak nazwa trasy, szablon trasy, parametry domyślne, a nawet ograniczenia trasy. Jak możesz zobaczyć w listingu 3.1, wygenerowany przez szablon projektu kod konfiguracyjny tras definiuje trasę domyślną dla API zawierającą prefiks `api` w adresie URI, nazwę kontrolera i opcjonalny parametr w postaci identyfikatora. Bez konieczności wprowadzania jakichkolwiek modyfikacji ta deklaracja trasy zwykle okazuje się wystarczająca do tworzenia API pozwalającego na pobieranie, uaktualnianie i usuwanie danych. Taka możliwość wynika ze sposobu, w jaki klasa kontrolera ASP.NET Web API mapuje metody HTTP na metody akcji kontrolera. Mapowaniem metod HTTP zajmiemy się dokładniej w dalszej części rozdziału, natomiast szczegółowe omówienie tego tematu znajdziesz w rozdziale 12.

Plik ValuesController.cs

Klasa `ApiController` będąca klasą nadrzędną dla klasy `ValuesController` jest sercem ASP.NET Web API. Wprawdzie można utworzyć prawidłowy kontroler ASP.NET Web API przez implementację różnych elementów składowych interfejsu `IHttpController`, ale w praktyce większość kontrolerów ASP.NET Web API powstaje na podstawie klasy `ApiController`. Ta klasa odgrywa ważną rolę podczas koordynacji pozostałych klas w modelu obiektowym ASP.NET Web API w celu przeprowadzania kilku kluczowych zadań podczas przetwarzania żądania HTTP:

- wybór i uruchomienie metody akcji w klasie kontrolera;
- konwersja elementów komunikatu żądania HTTP na parametry w metodzie akcji kontrolera i konwersja wartości metody akcji kontrolera na prawidłową część głównej odpowiedzi HTTP;
- wykonanie różnego rodzaju filtrów skonfigurowanych dla metody akcji, kontrolera lub globalnie;
- udostępnienie odpowiedniego stanu kontekstu metodom akcji klasy kontrolera.

Oparcie na klasie bazowej `ApiController` i wykorzystanie zalet kluczowych zadań przetwarzania żądania powoduje, że klasa `ValuesController` dołączona do szablonu Web API stanowi wysokiego poziomu abstrakcję zbudowaną na podstawie `ApiController`. W listingu 3.2 przedstawiliśmy kod domyślnej klasy `ValuesController`.

Listing 3.2. Kod domyślnej klasy `ValuesController`

```
public class ValuesController : ApiController
{
    // GET api/values
    public IEnumerable<string> Get()
    {
        return new string[] { "value1", "value2" };
    }

    // GET api/values/5
    public string Get(int id)
    {
        return "value";
    }

    // POST api/values
    public void Post([FromBody]string value)
    {
    }

    // PUT api/values/5
    public void Put(int id, [FromBody]string value)
    {
    }

    // DELETE api/values/5
    public void Delete(int id)
    {
    }
}
```


Klasa `ValuesController` choć prosta, to jednak okazuje się niezwykle użyteczna, gdy po raz pierwszy spojłdasz na model programowania kontrolera.

Przede wszystkim zwróć uwagę na nazwy metod akcji kontrolera. Domyślnie w ASP.NET Web API stosowana jest konwencja polegająca na nadaniu metodzie akcji nazwy odpowiadającej metodzie HTTP. Ujmując rzecz dokładniej, klasa `ApiController` wyszukuje metodę akcji o nazwie rozpoczynającej się od odpowiedniej metody HTTP. Dlatego w kodzie przedstawionym w listingu 3.2 żądanie HTTP GET do zasobu `/api/values` będzie obsługiwane przez pozbawioną parametrów metodę `Get()`. Framework oferuje różne sposoby dostosowania tej domyślnej logiki dopasowywania nazw i zapewnia możliwość rozbudowy mechanizmu, a nawet jego całkowite zastąpienie, jeśli istnieje potrzeba. Więcej szczegółowych informacji na temat wyboru kontrolera i akcji znajdziesz w rozdziale 12.

Poza wyborem metody akcji na podstawie metody HTTP ASP.NET Web API pozwala również wybrać akcję na podstawie elementów dodatkowego żądania, takich jak parametry ciągu tekstowego zapytania. Co ważniejsze, framework obsługuje dołączanie tych elementów żądania do parametrów metody akcji. Domyślnie używane jest połączenie obu podejść w celu przeprowadzenia operacji dołączania parametru, a sam algorytm obsługuje zarówno proste, jak i złożone typy .NET. W przypadku odpowiedzi HTTP model programowania ASP.NET Web API umożliwia metodom akcji zwrot typów .NET i konwertuje te wartości na odpowiednią postać części głównej komunikatu odpowiedzi HTTP, używając negocjacji treści. Więcej szczegółowych informacji na temat dołączania parametrów i negocjacji treści znajdziesz w rozdziale 13.

Omówiliśmy już pewne aspekty projektu ASP.NET Web API, choć zaledwie dotknęliśmy tematu modelu programowania, analizując kod wygenerowany przez szablon projektu. Teraz możemy pójść nieco dalej i utworzyć nasze pierwsze Web API typu „Witaj, świecie!”.

„Witaj, Web API!”

Jako pierwszy przykład ASP.NET Web API utworzymy prostą usługę powitalną. Czy w świecie programowania istnieje bardziej znane powitanie niż „Witaj, świecie!”? Pracę rozpoczynamy więc od prostego, przeznaczonego tylko do odczytu API powitania, a następnie w pozostałej części rozdziału dodamy kilka uprawnień, aby tym samym zaprezentować inne aspekty modelu programowania ASP.NET Web API.

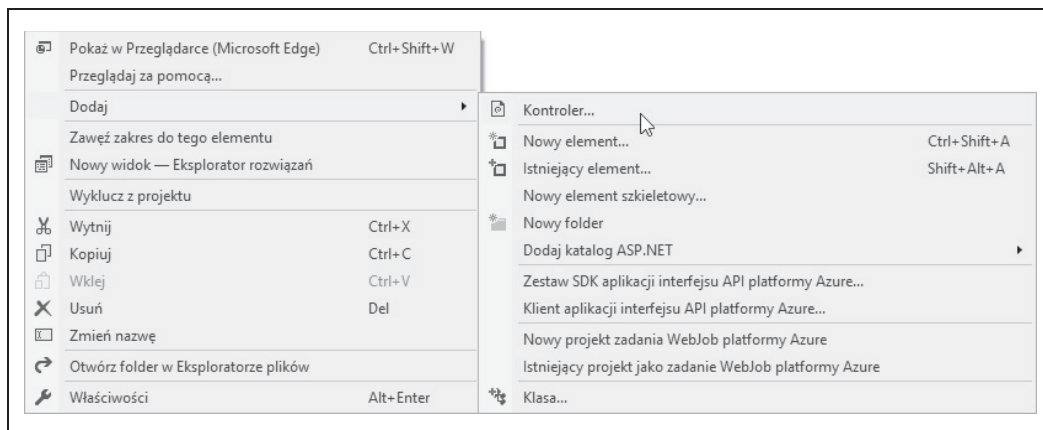
Utworzenie usługi

Aby zbudować usługę, zacznij od utworzenia nowej aplikacji ASP.NET z poziomu okna dialogowego *Nowy projekt* w Visual Studio. W kolejnym oknie dialogowym jako rodzaj projektu wybierz *Web API*. W ten sposób wygenerujesz nowy projekt ASP.NET Web API na podstawie szablonu domyślnego.

Usługa powitalna tylko do odczytu

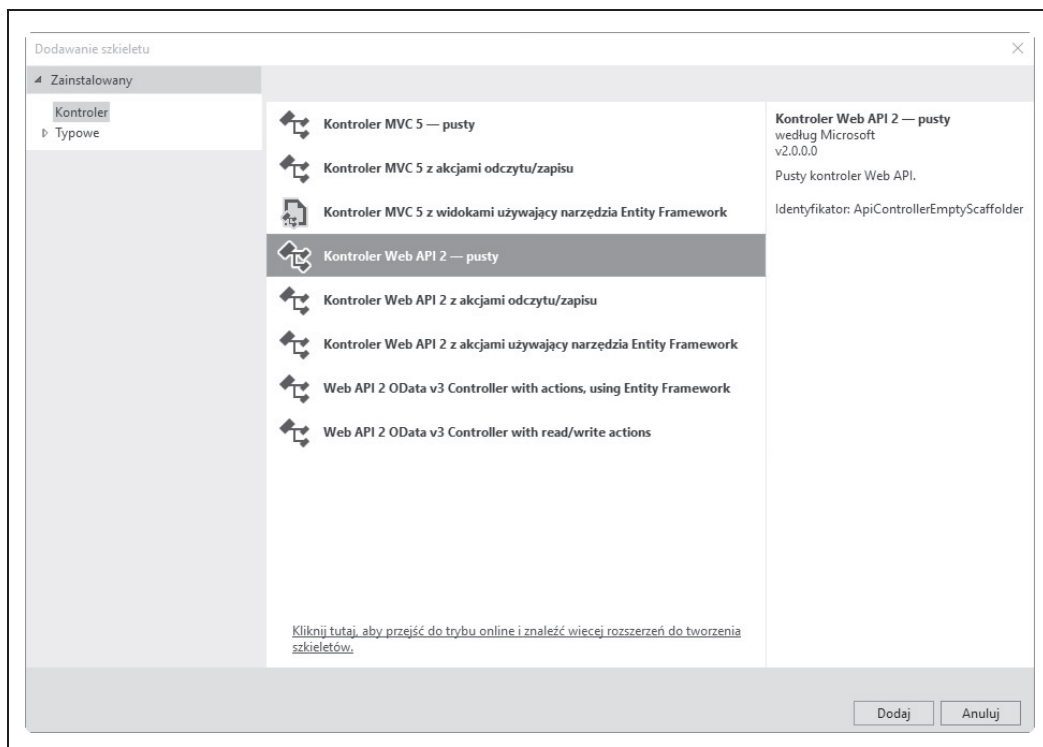
Gdy masz wygenerowany projekt Web API na podstawie szablonu domyślnego, pierwszym zadaniem jest dodanie kontrolera. Możesz w tym celu po prostu dodać nową klasę lub skorzystać z oferowanego przez Visual Studio szablonu kontrolera. Aby dodać kontroler za pomocą szablonu, kliknij

prawym przyciskiem myszy katalog *Controllers*, a następnie z menu kontekstowego wybierz opcję *Dodaj/Kontroler...* (rysunek 3.5).



Rysunek 3.5. Menu kontekstowe w Visual Studio umożliwiające dodanie nowego kontrolera

Na ekranie zostanie wyświetlone kolejne okno dialogowe, w którym podajesz dodatkowe informacje konfiguracyjne dla tworzonego kontrolera. Tworzymy kontroler o nazwie *GreetingController* i używamy do tego szablonu *Kontroler Web API 2 — pusty* (rysunek 3.6).

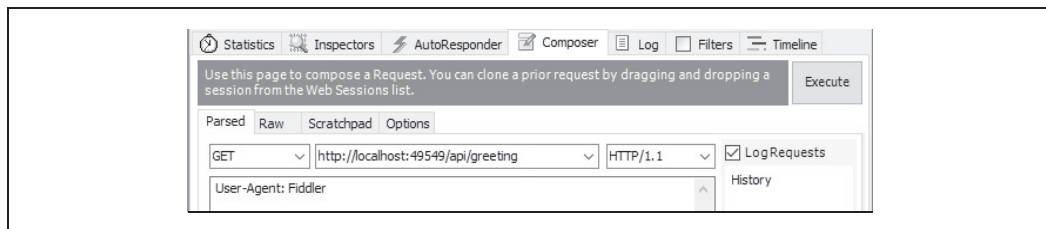


Rysunek 3.6. Szablon umożliwiający utworzenie kontrolera Web API

Po podaniu wymaganych informacji i kliknięciu przycisku OK w wyświetlonym oknie dialogowym nastąpi wygenerowanie nowej klasy `GreetingController` wywodzącej się z klasy `ApiController`. Aby nasze nowe API zwracało komunikat powitania, konieczne jest dodanie do tego kontrolera metody, która będzie potrafiła udzielać odpowiedzi na żądanie HTTP GET. Pamiętaj, że z powodu istnienia domyślnej reguły routingu kontroler `GreetingController` będzie wybierany dla żądania HTTP prowadzącego do zasobu `api/greeting`. Dlatego dodajemy prostą metodę przeznaczoną do obsługi żądań GET:

```
public class GreetingController : ApiController
{
    public string GetGreeting() {
        return "Witaj, świecie!";
    }
}
```

Teraz możemy przetestować nasze Web API i sprawdzić, czy faktycznie zwraca zdefiniowany komunikat powitania. W tym celu wykorzystamy narzędzie o nazwie *Fiddler*⁸ przeznaczone do debugowania proxy HTTP. Jedną ze szczególnie użytecznych funkcji narzędzia Fiddler podczas testowania Web API jest możliwość tworzenia komunikatów HTTP i ich wykonywania. Tej funkcji użyjemy do przetestowania naszego API (rysunek 3.7).



Rysunek 3.7. Utworzenie nowego żądania HTTP za pomocą narzędzia Fiddler

Po wykonaniu żądania można przeanalizować zarówno żądanie, jak i odpowiedź, używając do tego pokazanego na rysunku 3.8 panelu inspektora sesji w narzędziu Fiddler.

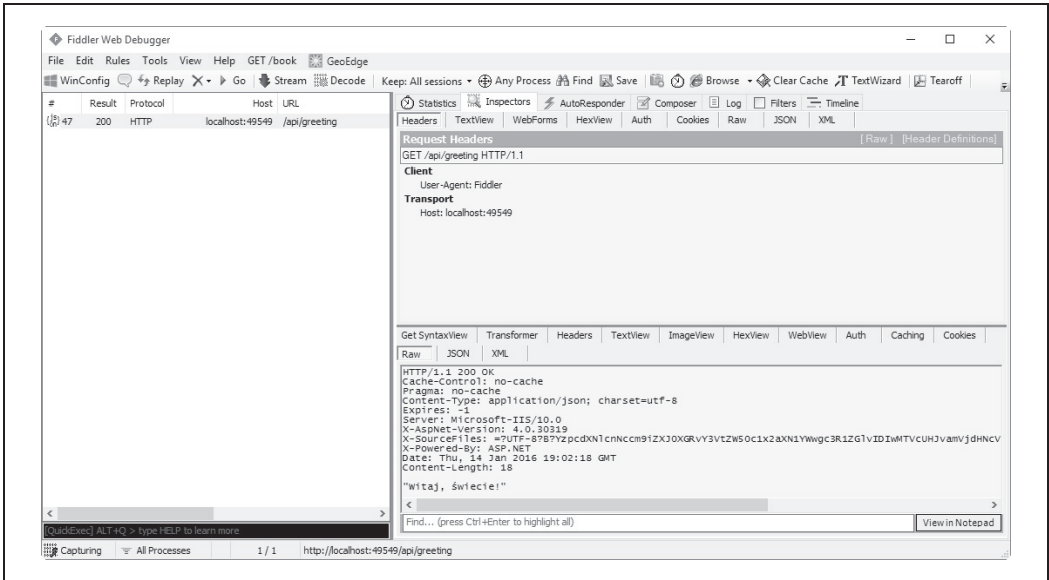
Zgodnie z oczekiwaniami przedstawione tutaj proste żądanie HTTP GET do naszej usługi powitalnej powoduje zwrot ciągu tekstowego „Witaj, świecie!”.

Negocjacja treści

Powracamy na chwilę do rysunku 3.8, aby nieco dokładniej przyjrzeć się nagłówkowi HTTP `Content-Type`. Domyślnie ASP.NET Web API spowoduje transformację wartości zwrrotnych metod akcji na format JSON za pomocą popularnej biblioteki `Json.NET`, o której po raz pierwszy wspomnieliśmy na rysunku 3.3. Zgodnie jednak z informacjami przedstawionymi we wcześniejszej części rozdziału ASP.NET Web API obsługuje opartą na serwerze negocjację treści, a domyślny wybór odbywa się między formatami JSON i XML. Aby zobaczyć to w działaniu, powróć do panelu tworzenia żądania w narzędziu Fiddler i dodaj poniższy wiersz kodu w polu tekstowym przeznaczonym dla nagłówków żądania:

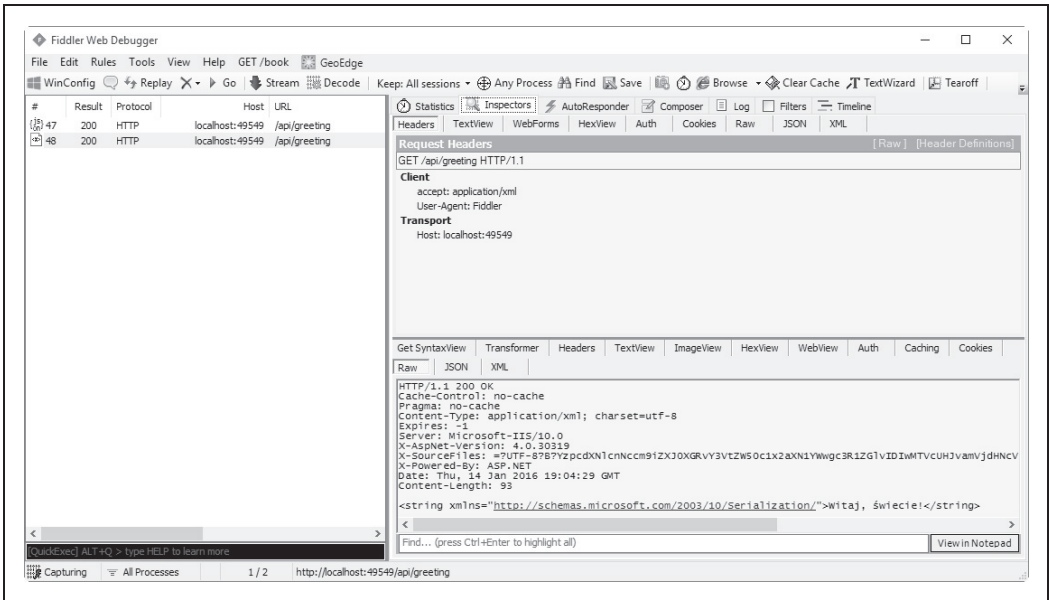
```
accept: application/xml
```

⁸ <http://www.telerik.com/fiddler>



Rysunek 3.8. Analizowanie żądań i odpowiedzi HTTP za pomocą narzędzia Fiddler

Teraz ponownie wykonaj żądanie; zauważysz, że odpowiedź zawiera nagłówek Content-Type: application/xml, a część główna odpowiedzi została sformatowana jako XML (rysunek 3.9).



Rysunek 3.9. Przykład negocjacji treści w przypadku żądania do naszej usługi powitalnej i odpowiedzi na to żądanie

Dodanie powitania

Wprawdzie interesująca może być możliwość otrzymywania powitania w różnych formatach, ale nieco bardziej skomplikowane API wymagają możliwości przeprowadzania operacji na stanie lub danych systemu. Dlatego rozbudujemy naszą usługę powitalną i umożliwimy klientom dodawanie nowych powitań. Idea polega na tym, aby klient mógł podać nazwę powitania i komunikat, dodać je do usługi, a następnie otrzymać w odpowiedzi na żądanie GET po podaniu w adresie URL nazwy interesującego go powitania. Ponadto trzeba zapewnić obsługę sytuacji, w której klient, podając nazwę powitania w adresie URL zrobi literówkę lub inny błąd. Wówczas powinien zostać zwrócony kod stanu HTTP 404 informujący o braku możliwości odnalezienia wskazanego zasobu.

Aby umożliwić klientowi utworzenie nowego powitania w serwerze, konieczne jest przygotowanie klasy modelu przeznaczonej do przechowywania nazwy powitania oraz właściwości komunikatu. Spełniamy ten wymóg przez dodanie przedstawionej poniżej klasy do katalogu *Models* projektu:

```
public class Greeting
{
    public string Name
    {
        get;
        set;
    }

    public string Message
    {
        get;
        set;
    }
}
```

Kolejnym krokiem jest utworzenie w kontrolerze *GreetingController* metody akcji odpowiedzialnej za obsługę żądania HTTP POST i akceptującej parametr w postaci egzemplarza *Greeting*.

Owa metoda akcji dodaje powitanie do statycznej listy powitań i zwraca kod stanu HTTP 201 wraz z nagłówkiem *Location* zawierającym adres URL prowadzący do nowo utworzonego zasobu powitania. Dodatkowy nagłówek *Location* pozwala klientom podążać za wartością łącza zamiast wymagać od nich przygotowania adresu URL dla nowego zasobu powitania. Tym samym rozwiązanie staje się bardziej niezawodne, ponieważ struktury adresów URL serwera mogą ulegać zmianie na przestrzeni czasu:

```
public static List<Greeting> _greetings = new List<Greeting>();

public HttpResponseMessage PostGreeting(Greeting greeting)
{
    _greetings.Add(greeting);

    var greetingLocation = new Uri(this.Request.RequestUri,
        "greeting/" + greeting.Name);
    var response = this.Request.CreateResponse(HttpStatusCode.Created);
    response.Headers.Location = greetingLocation;

    return response;
}
```

Po dodaniu nowego powitania do kolekcji statycznej tworzymy egzemplarz URI przedstawiający miejsce, gdzie nowe powitanie może być znalezione w trakcie kolejnych żądań. Kolejnym krokiem jest utworzenie nowego obiektu *HttpResponseMessage* za pomocą metody fabryki *CreateResponse* ()

egzemplarza `HttpRequestMessage` dostarczonego przez klasę `ApiController`. Możliwość pracy z poziomu metod akcji z egzemplarzami obiektów modelu HTTP stanowi kluczową cechę ASP.NET Web API, zapewnia dokładną kontrolę nad elementami komunikatów HTTP, takimi jak nagłówki `Location`, w sposób nieopierający się na statycznych obiektach kontekstu, na przykład `HttpContext` i `WebOperationContext`. To staje się szczególnie użyteczne podczas tworzenia testów jednostkowych dla kontrolerów Web API, czym zajmiemy się już wkrótce.

Na koniec musimy jeszcze dodać przeciążoną metodę `GetGreeting()`, która będzie potrafiła pobrać i zwrócić powitanie dostarczone przez klient:

```
public string GetGreeting(string id)
{
    var greeting = _greetings.FirstOrDefault(g => g.Name == id);
    return greeting.Message;
}
```

Powyższa metoda wyszukuje pierwsze powitanie, dla którego właściwość `Name` zostanie dopasowana do podanego parametru `id`, a następnie zwraca właściwość `Message`. Warto zwrócić uwagę, że na obecnym etapie nie są przeprowadzane żadne operacje weryfikacji danych wejściowych parametru `id`. Do tego zagadnienia powrócimy jednak w kolejnej sekcji.

Domyślnie część główna żądania HTTP POST jest obsługiwana przez obiekt `MediaTypeFormatter` wybrany na podstawie nagłówka żądania `Content-Type`. Odpowiednio, kolejne żądanie HTTP będzie obsługiwane przez domyślny formater JSON, który używa biblioteki `Json.NET` do deserializacji ciągu tekstowego JSON na postać egzemplarza klasy `Greeting`:

```
POST http://localhost:50650/api/greeting HTTP/1.1
Host: localhost:50650
Content-Type: application/json
Content-Length: 43
```

```
{"Name": "TestGreeting", "Message": "Witaj!"}
```

Otrzymany egzemplarz może być następnie przekazany metodzie `PostGreeting()`, za pomocą której zostanie dodany do kolekcji powitań. Po przetworzeniu żądania przez metodę `PostGreeting()` klient otrzyma następującą odpowiedź HTTP:

```
HTTP/1.1 201 Created
Location: http://localhost:50650/api/greeting/TestGreeting
```

Na podstawie informacji znajdujących się w nagłówku `Location` klient może wykonać żądanie do nowego powitania:

```
GET http://localhost:50650/api/greeting/TestGreeting HTTP/1.1
Host: localhost:50650
```

Podobnie jak w przypadku opracowanej na początku usługi powitania tylko do odczytu, klient może spodziewać się otrzymania następującej odpowiedzi:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 8
```

```
"Witaj!"
```

Obsługa błędów

Pokazany powyżej przykład wymiany komunikatów HTTP sprawdza się doskonale, o ile serwer nigdy nie napotka żadnych błędów, a wszystkie klienty będą korzystać z tych samych zasad i konwencji. Być może zastanawiasz się, co się stanie w przypadku wystąpienia błędu serwera lub otrzymania przez niego nieprawidłowego żądania. To jest kolejny obszar, na którym możliwość tworzenia egzemplarzy obiektów modelu HTTP i pracy z nimi okazuje się niezwykle użyteczna. Metoda akcji przedstawiona w listingu 3.3 zwraca ciąg tekstowy powitania pobrany na podstawie jego nazwy. Jednak w przypadku nieznalezienia nazwy wskazanego powitania zwrócona będzie odpowiedź wraz z kodem stanu HTTP 404. Do obsługi tego rodzaju sytuacji ASP.NET Web API oferuje wyjątek `HttpResponseException`.

Listing 3.3. Zwrot kodu stanu HTTP 404, gdy wskazane powitanie nie zostanie znalezione

```
public string GetGreeting(string id)
{
    var greeting = _greetings.FirstOrDefault(g => g.Name == id);
    if (greeting == null)
        throw new HttpResponseException(HttpStatusCode.NotFound);
    return greeting.Message;
}
```

Wprowadzić rozsądne będzie zwrócenie nowego obiektu `HttpResponseMessage` zawierającego kod stanu HTTP 404, ale to zawsze wymaga zwrotu `HttpResponseMessage` z poziomu metody akcji `GetGreeting()`, co oznacza niepotrzebne skomplikowanie ścieżki kodu pozbawionej obsługi wyjątku. Ponadto komunikat odpowiedzi musiałby zostać przekazany przez cały potok Web API, co w przypadku wyjątku jest prawdopodobnie niepotrzebne. Z tych powodów zdecydowaliśmy się na zgłoszenie wyjątku `HttpResponseException` zamiast zwracać obiekt `HttpResponseMessage` z metody akcji. W przypadku wyjątku zawierającego część główną odpowiedzi obsługującą negocjację treści zawsze można użyć metody `Request.CreateErrorResponse()` z klasy bazowej kontrolera i przekazać otrzymany obiekt `HttpResponseMessage` do konstruktora `HttpResponseException`.

Testowanie API

Kolejną zaletą bezpośredniej pracy z egzemplarzami obiektów modelu HTTP zamiast ze statycznymi obiektami kontekstu jest możliwość tworzenia odpowiednich testów jednostkowych dla kontrolerów Web API. Szczegółowe omówienie tematu testowania znajdziesz w rozdziale 17. Tutaj jednak, tytułem wprowadzenia do tego zagadnienia, utworzymy krótki test jednostkowy dla metody akcji `PostGreeting()` zdefiniowanej w kontrolerze `GreetingController`:

```
[Fact]
public void TestNewGreetingAdd()
{
    // Przygotowanie
    var greetingName = "newgreeting";
    var greetingMessage = "Witaj, test!";
    var fakeRequest = new HttpRequestMessage(HttpMethod.Post,
        "http://localhost:9000/api/greeting");
    var greeting = new Greeting { Name =
        greetingName, Message = greetingMessage };

    var service = new GreetingController();
    service.Request = fakeRequest;
    // Działanie
```

```

var response = service.PostGreeting(greeting);

//Asercja
Assert.NotNull(response);
Assert.Equal(HttpStatusCode.Created, response.StatusCode);
Assert.Equal(new Uri("http://localhost:9000/api/greeting/newgreeting"),
    response.Headers.Location);
}

```

W przedstawionym powyżej teście został zastosowany typowy wzorzec *przygotowanie, działanie i asercja*, powszechnie używany w testach jednostkowych. Możemy utworzyć pewien stan kontrolny, między innymi egzemplarz `HttpRequestMessage`, w celu przedstawienia całego żądania HTTP. Następnie wywołujemy metodę w teście, używając kontekstu, a na koniec przetwarzamy kilka asercji związanych z odpowiedzią. W omawianym przykładzie odpowiedź jest egzemplarzem `HttpResponseMessage`, natomiast w wyniku otrzymujemy możliwość przetworzenia asercji w elementach danych samej odpowiedzi.

Klient

Jak wspomnieliśmy na początku rozdziału, jedną z podstawowych korzyści płynących ze zbudowania ASP.NET Web API na podstawie modelu programowania HTTP jest fakt, że ten sam model programowania może być wykorzystany do tworzenia doskonałych aplikacji HTTP zarówno dla serwera, jak i dla klienta. Na przykład za pomocą przedstawionego poniżej kodu stworzymy żądanie, które zostanie obsłużone przez naszą pierwszą metodę akcji `GetGreeting()`:

```

class Program
{
    static void Main(string[] args)
    {
        var greetingServiceAddress =
            new Uri("http://localhost:50650/api/greeting");

        var client = new HttpClient();
        var result = client.GetAsync(greetingServiceAddress).Result;
        var greeting = result.Content.ReadAsStringAsync().Result;

        Console.WriteLine(greeting);
    }
}

```

Podobnie jak w przypadku serwera, kod działający po stronie klienta tworzy i przetwarza egzemplarze `HttpRequestMessage` i `HttpResponseMessage`. Ponadto komponenty rozszerzające ASP.NET Web API, takie jak formatery typów danych i procedury obsługi komunikatów, działają zarówno z klientami, jak i z serwerami.

Host

Opracowanie ASP.NET Web API z przeznaczeniem do hostingu w tradycyjnej aplikacji ASP.NET wydaje się niezwykle podobne do procesu opracowania aplikacji typu ASP.NET MVC. Ale jedną z doskonałych cech ASP.NET Web API jest możliwość hostingu w dowolnym procesie bez konieczności wykonywania w tym celu dodatkowej pracy. Listing 3.4 przedstawia kod niezbędny do

hostingu naszego kontrolera `GreetingController` w innym procesie hosta (w omawianym przykładzie jest to aplikacja działająca w konsoli).

Listing 3.4. Prosty hosting Web API w postaci aplikacji działającej w konsoli

```
class Program
{
    static void Main(string[] args)
    {
        var config = new HttpSelfHostConfiguration(
            new Uri("http://localhost:50651"));

        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional });

        var host = new HttpSelfHostServer(config);

        host.OpenAsync().Wait();

        Console.WriteLine("Naciśnij dowolny klawisz, aby zakończyć działanie.");
        Console.ReadKey();

        host.CloseAsync().Wait();
    }
}
```

Do hostingu Web API we własnym procesie nie musieliśmy modyfikować kontrolera ani dodawać żadnego magicznego kodu XML w pliku *app.config*. Zamiast tego po prostu utworzyliśmy egzemplarz `HttpSelfHostConfiguration`, skonfigurowaliśmy go, podając niezbędne informacje dotyczące adresu i routingu, a następnie otworzyliśmy `host`. Kiedy `host` jest otwarty i nasłuchuje żądań, blokujemy wątek główny konsoli, aby uniknąć zamknięcia serwera. Gdy użytkownik zdecyduje się na zamknięcie hosta (przez naciśnięcie dowolnego klawisza), wtedy zamykamy `host` Web API i kończymy działanie aplikacji w konsoli. Szczegółowe omówienie tematu hostingu znajdziesz w rozdziale 11.

Podsumowanie

W tym rozdziale omówiliśmy pewne kluczowe cele projektowe stojące za ASP.NET Web API. Wykorzystaliśmy domyślny szablon projektu Web API, aby zobaczyć, jak różne komponenty tworzące framework zostały zorganizowane i jak są rozprowadzane za pomocą menedżera pakietów NuGet. Na podstawie analizy kodu szablonu domyślnego rozpoczęliśmy także omawianie modelu programowania frameworku. Później utworzyliśmy naszą pierwszą usługę Web API typu „Witaj, świecie!” i wykorzystaliśmy możliwości ASP.NET Web API w zakresie samohostowania.

W kolejnych rozdziałach znajdziesz znacznie dokładniejsze omówienie tematów wprowadzonych w tym rozdziale. Zaczynamy od rozdziału 4., w którym przedstawiliśmy mechanizmy umożliwiające działanie ASP.NET Web API.

.NET Framework, 355

A

ActionFilterAttribute, 437

adres

URI, 25, 325

URL, 112, 275

akceptacja, 169

akcje, 309

aktywacja kontrolera, 301

ALPS, application-level profile semantics, 132

analizowanie żądań HTTP, 82

antywzorzec usługi, 223

API, 17, 85

GitHub, 140

homogeniczne, 153

tworzenie, 159

zorientowane na zasoby, 58

aplikacja, 105, 260

cele, 113

przebieg działania, 227

przepływ zdarzeń, 141

wersjonowanie, 110

aplikacje klienta, 399

architektura

hostingu ASP.NET, 270

przetwarzania, 89

przetwarzania ASP.NET, 90

sieci, 24

WCF, 271

ARPANET, 23

asercje, 380

SAML, 380

ASP.NET Web API, 72

ASP.NET Web API 101, 67

atak

CSRF, 408

typu MITM, 358, 466

atrybut, 117

AuthorizeAttribute, 388

EnableCorsAttribute, 394

atrybuty adnotacji danych, 331

audyt zmian, 209, 210

autoryzacja, 308, 386

egzekwowanie, 388

na podstawie zakresu, 416

podstawowa, 387

AWS, Amazon Web Services, 209

Azure Service Bus, 284

Azure Storage, 43

B

baza danych, 25

BDD, behavior-driven development, 14, 71, 160

bezpieczeństwo, 347, 464

transportu, 347

wątków, 337

biblioteka Thinkecture.IdentityModel, 390

biblioteki, 161

klienta, 218

opakowujące, 218

bindery modelu, 319, 323

buforowanie, 38, 41, 455–458

danych wyjściowych, 200

koncepcja weryfikacji, 39

koncepcja wygaśnięcia, 39

utrata ważności, 39

C

CDN, content delivery network, 33

cele aplikacji, 113

certyfiakat, 465, 468

klienta, 361

testowy, 471

CLR, common language runtime, 99

- cofnięcie certyfikatu, 471
- Cool URI, 26
- CORS, cross-origin resource sharing, 391
- CRL, certificate revocation lists, 471
- CRUD, create, read, update, delete, 169
- CSRF, cross-site request forgery, 374, 392, 408

D

- dane
 - odwołania, 140
 - wyjściowe, 198
- DARPA, 23
- delegowanie
 - ograniczonej autoryzacji, 397
 - uprawnień, 397
 - uwierzytelniania, 397
- deserializacja łączy, 225
- dokument zgłoszenia błędu, 463
- dołączanie, 321
 - modelu, 312–316, 325
 - parametru, 98, 99
- dostawca
 - tożsamości, 356
 - wartości, 320
- dostęp do chronionych zasobów, 401
- dyspozytor, 296
 - kontrolerów, 95
 - tras, 95
- działanie
 - aplikacji, 227
 - bufora, 41

E

- egzekwowanie autoryzacji, 388
- element `HttpControllerDispatcher`, 297
- encja, 25, 308
- ETag, entity tag, 40
- ewolucja, 106

F

- filtry, 306
 - akcji, 101, 309
 - autoryzacji, 101, 308
 - uwierzytelniania, 101, 307, 375
 - wyjątków, 310

- format, 145
 - Collection+Json, 182
 - JSON, 27, 149
 - vCard, 48
 - XML, 27
- formatery, 315
 - synchroniczne, 329
- formaty
 - nowe, 129
 - popularne, 128
 - proste, 126
- formularz rejestracji klienta, 400
- framework
 - ASP.NET, 73
 - OAuth 2.0, 397, 407
 - uwierzytelniania, 397
 - uwierzytelniania HTTP, 367
 - xUnit.NET, 422
- frameworki testów jednostkowych, 420
- funkcjonalności, 197

G

- generowanie podpisu komunikatu, 384
- globalna konfiguracja Web API, 267
- grupy atrybutów, 117

H

- HAL, hypertext application language, 27, 134
- Hawk, 385
- H-Factors, 60
- hipermedia, 58, 131, 136
- HMAC, hash-based message authentication code, 43, 209, 383
- host, 86
- hosting, 71
 - Azure Service Bus, 284
 - IIS, 349
 - OWIN, 92
 - w pamięci, 284, 285
 - Web API, 275
 - WWW, 92, 260
- HTTP, Hypertext Transfer Protocol, 14, 17, 24, 29
 - buforowanie, 38
 - kody stanu, 37
 - metody, 33, 36
 - pośredniki, 32
 - uwierzytelnianie, 42
 - wymiana komunikatów, 30
- HTTP 1.1, 29

I

- IANA, 27, 150, 464
- identyfikatory typów danych, 112
- IETF, Internet Engineering Task Force, 29, 137
- IIS, internet information services, 71, 92, 361
- imitacje, 426
- implementacja
 - ActionFilterAttribute, 333
 - ASP.NET Web API, 427
 - FormatterParameterBinder, 326
 - interfejsu IValueProvider, 321
 - IssueModelBinder, 324
 - JsonMediaTypeFormatter, 327
 - MediaTypeFormatter, 328
 - ModelBindingParameterBinder, 320
 - monitorowania, 214
 - obsługi zgłoszenia błędu, 212
 - uwierzytelniania, 369
 - w stylu BDD, 160
 - wykrywania konfliktów, 206
 - zgłoszenia błędu, 211
- infrastruktura ASP.NET, 260
- interfejs
 - ICorsPolicyProviderFactory, 395
 - IHttpRequestResult, 100, 432
 - IHttpController, 302
 - IHttpControllerSelector, 299
 - IValueProvider, 321
 - OWIN, 276
- interfejsy filtrów, 101
- interoperacyjność, 464
- ISAM, indexed sequential access method, 127

J

- JSON, 27, 149
- JWT, JSON web token, 380

K

- kanal
 - nieoficjalny, 407
 - oficjalny, 407
- Katana, 277
- klasa
 - ActionFilterAttribute, 437
 - ApiController, 84, 97, 100, 366
 - Assert, 423

- AuthorizationContext, 389
- AuthorizationFilterAttribute, 308
- Claim, 354
- ClaimsAuthorizationManager, 390
- ClaimsIdentity, 354
- DataContractSerializer, 327
- ExceptionHandlerAttribute, 310
- FormUrlEncodedContent, 250
- HttpClient, 335
 - bezpieczeństwo wątków, 337
 - cykl życiowy, 335
 - metody pomocnicze, 337
 - opakowanie, 336
 - wiele egzemplarzy, 336
 - wyjątki, 338
- HttpConfiguration, 216
- HttpContent, 247
- HttpContentHeaders, 243
- HttpHeaders, 246
- HttpMessageHandler, 436
- HttpMessageInvoker, 295
- HttpMethodOverrideHandler, 341
- HttpPropertyKeys, 242
- HttpRequestHeaders, 243
- HttpRequestMessage, 239
- HttpResponseHeaders, 243
- HttpResponseMessage, 239
- HttpSelfHostServer, 272
- HttpServer, 295
- HttpServiceBusServer, 287
- IssueController, 181
- IssueLinkFactory, 168
- IssuerNameRegistry, 364
- IssuesState, 164
- IssueState, 164
- IssueStateFactory, 166
- Link, 166
- LinkFactory, 167
- MediaTypeFormatter, 326, 327, 433
- SelfHostConfiguration, 273
- ValuesController, 79
- WebApiConfiguration, 173
- XmlSerializer, 327

- klasy
 - filtrów, 306
 - kontenerów nagłówków, 245
 - routingu w ASP.NET, 263
 - routingu Web API, 265

- klient, 86, 217, 232
 - poufny, 400
 - publiczny, 400
- klucze
 - prywatne, 465
 - publiczne, 361, 465
 - testowe, 471
- kody stanu HTTP, 38
- kolekcja
 - MediaTypeMappings, 327
 - zasobów, 120
 - grup atrybutów, 117
- kolekcje zgłoszeń błędów, 119
- komunikaty, 238, 247, 248
 - klienta, 341
 - samoopisujące się, 108
- koncepte projektu sieci, 24
- konfiguracja
 - globalna, 266
 - Web API, 280
- konflikty, 206
- konstruktor klasy IssueController, 181
- kontrakty, 126
- kontrola dostępu, 275
- kontroler, 291, 301
 - IssueController, 176
 - IssueProcessorController, 194
 - Web API, 80
- konwencje nazw, 463
- konwersja na HttpResponseMessage, 100
- konwerter wyniku akcji, 313
- koszt zmiany, 108
- kryptografia klucza publicznego, 465
- kryteria akceptacji, 169

L

- lista zasobów, 153
- logika wyboru kontrolera, 299

ł

- łańcuch certyfikatu, 469
- łącza, 139
 - jako funkcje, 222
 - jako zakładki, 226

M

- MAC, message authentication codes, 348
- magazyn
 - dla zgłoszeń błędów, 163
 - zaufania, 467
- mapowanie
 - konceptualne, 25
 - tożsamości, 363
- mechanizm
 - CORS, 393
 - o chunked transfer encoding, 254
- menedżer uwierzytelniania, 374
- menu kontekstowe, 80
- metadane
 - osadzone, 148
 - zewnętrzne, 148
- metoda
 - CanReadType(), 327
 - CanWriteType(), 327
 - CheckAccess(), 389
 - ExecuteBindingAsync(), 317
 - FindAsync(), 195
 - Get(), 201, 428
 - GetAsync(), 339
 - GetSearch(), 185
 - HTTP PATCH, 188
 - IsValidAction(), 195
 - Post(), 429
 - SelectController(), 299
 - SendAsync(), 340
 - SerializeToStreamAsync(), 256
 - Trace(), 214
 - TryComputeLength(), 256
 - UseWebApi(), 280
 - WriteToStreamAsync(), 434
- metody HTTP, 33, 36, 56
- MIME, 28
- MITM, man-in-the-middle, 358, 466
- model, 163, 315
 - autoryzacji, 387
 - dojrzałości Richardsons, 52
 - informacji, 114, 118
 - oświadczeń, 352
 - programowania HTTP, 237
 - przetwarzania ApiController, 302
 - przetwarzania ASP.NET Web API, 91
 - zasobu, 122
- modele zasobów, 119

moduły, 261
monitorowanie, 213
zgłoszeń błędów, 153

N

nagłówek
ETag, 40
Location, 83
nagłówki, 242
HTTP, 447
komunikatu, 36, 447
odpowiedzi, 37, 449
reprezentacji, 37, 450
żądania, 36, 448, 449
narzędzie
Fiddler, 82
makecert, 472
NAT, network address translatio, 285
nazwy domen, 351
negocjacja treści, 38, 81
odpowiedzi, 41
proaktywna, 451
reaktywna, 452
niestandardowe klasy zawartości, 255

O

OAuth 2.0, 397, 407
obiekt
HttpControllerDescriptor, 301
HttpRequestMessage, 241
IOwinContext, 281
obsługa
błędów, 85
buforowania, 198
dyspozytora trasy, 96
formatów, 70
hipermediów, 146, 182
komunikatów, 93, 292, 363, 368, 376
komunikatów klienta, 341
kontrolera, 96
mechanizmu CORS, 393
odpowiedzi, 343
pobierania buforowanych danych, 203
tokenów dostępu, 412
tras, 95, 298
Web API ASP.NET, 268
wersji, 230
wielu formatów, 156
zmian, 106

OCSF, online certificate status protocol, 471
odpowiedź, 225, 343
ograniczenia REST, 63
opcje uwierzytelniania, 371
OpenID Connect, 415, 416
operacje asynchroniczne, 225
oprogramowanie pośredniczące Web API, 281, 370
organizacja IANA, 27
organizacja testu jednostkowego, 423
ORM, object-relational mapping, 221
osadzanie łączy, 142
osadzone relacje łączy, 152
OWIN, 276, 283

P

pakiety, 161
NuGet, 73–75
parametry, 98
plik
app.config, 87
ValuesController.cs, 78
WebApiConfig.cs, 76
pliki .pfx, 473
pobieranie
buforowanych danych, 202, 203
tokenu dostępu, 402
wszystkich zgłoszeń błędów, 200
zgłoszenia błędu, 174, 179, 182
nieistniejącego, 179
otwartego, 177
zamkniętego, 177
pomiar kontrolerek hipermediów, 60
pośredniki, 32
potok
filtrów, 98
kontrolera, 302, 303
procedur obsługi komunikatów, 292, 293
powiązane zasoby, 116
procedura
CorsMessageHandler, 395
procedury obsługi, 261
komunikatów klienta, 341
odpowiedzi, 343, 344
Web API ASP.NET, 268
profile, 132
semantyczne, 135
typów danych, 49

programowanie
 BDD, 14, 160
 HTTP, 68
 po stronie klienta, 69
 po stronie serwera, 69
 TDD, 14, 423

projekt, 159
 Katana, 277
 Web API, 75

protokół
 HTTP, 14, 17
 OCSP, 471

proxy, 342

przepływ
 danych, 282
 kodu autoryzacji, 406, 416
 komunikatów HTTP, 291
 zdarzeń, 141, 459

przerwanie na żądanie, 339

przetwarzanie
 tokenów dostępu, 411
 zgłoszenia błędu, 193
przypisywanie zleceniodawcy, 356

R

RDF, resource description framework, 132

refaktoryzacja kodu, 424

rejestracja
 relacji łącza, 153
 typu danych, 27, 150

relacje łącza
 nowe, 151
 osadzone, 152
 rejestracja, 153
 rozszerzenia, 152
 standardowe, 151

REPL, read-eval-print loop, 233

reprezentacja, 26
 JSON, 50
 PNG, 51
 zasobu, 31

REST, representational state transfer, 59, 63
 ograniczenia, 63

rezerwacja adresu URL, 275

RIA, rich internet applications, 48

rodzaje
 kontraktów, 126
 pośredników, 33

routing, 291
 ASP.NET, 262
 Web API, 264
rozszerzalność, 148
rozszerzenia relacji łącza, 152
RPC, remote procedure call, 52
RSK, REST starter kit, 335

S

SAML, 380
samohostowanie, 92, 162, 270, 351
samoopisywanie się, 108, 125
schematy uwierzytelniania, 42, 43
 AWS, 43
 Azure Storage, 43
 Hawk, 385
 OAuth 2.0, 43

SDK, software development kit, 286

selektor akcji, 304

semantyka, 137

Service Bus, 286

serwer
 autoryzacji, 399, 402, 410
 CDN, 33
 zasobów, 410

serwis GitHub, 400

sieć
 ARPANET, 24
 CDN, 33

SOAP, 45, 107

SPA, single-page applications, 399

specyfikacja, 108
 formatu, 142
 OpenID Connect, 415
 typu danych, 463

sprawdzanie
 certyfikatu, 363
 poprawności, 332
 poprawności modelu, 331
 buforowania, 200

SSL, secure socket layer, 348

stan klienta, 234

stosowalność, 147

strumieniowanie, 254

styl
 API, 51
 REST, 63
 RPC, 52

subdomeny, 115
szablony, 80

Ś

ścieżka certyfikacji serwera, 470
środowisko
 OWIN, 283
 uruchomieniowe CORS, 394

T

TAP, task asynchronous pattern, 94
TDD, test-driven development, 14, 71, 160, 419
test jednostkowy, 421
testowanie, 419
 API, 85
test, 175
 integracji, 442
testy jednostkowe, 71, 419
 ApiController, 427
 cykle czerwone i zielone, 424
 HttpMessageHandler, 436
 imitacje, 426
 implementacji ASP.NET Web API, 427
 MediaTypeFormatter, 433
 metody Get(), 428
 metody Post(), 429
 metody WriteToStreamAsync(), 434
 programowanie TDD, 423
 refaktoryzacja kodu, 424
 tras, 440
 wstrzykiwanie zależności, 426
 xUnit.NET, 422
TLS, transport layer security, 348, 362
token, 378
 dostępu, 401, 402, 411
 refresh, 409
tożsamość serwera, 350
TPL, task parallel library, 327
trasy, 95
 oparte na atrybutach, 298
tworzenie
 API, 159
 klienta, 217
 komunikatu, 250
 testowych certyfikatów, 471
 testowych kluczy, 471
 usługi, 79

 wersji API, 110
 zgłoszenia błędu, 186
typ mediów
 Collection+Json, 136
 Siren, 136
typy danych, 27, 118, 126, 445
 charakterystyczne dla domeny, 48
 ogólne, 132
 rejestracja, 150
 specyfikacja, 463
 w domenie, 153
typy hipermediów, 131, 136
typy relacji łączy, 137

U

uaktualnianie zgłoszenia błędu, 188, 190
uprawnienia kodu autoryzacji, 404
URI, 25
URL, universal resource locator, 26
URN, universal resource name, 26
urząd certyfikacji, 350, 466
usługa, 163
 checkcode.example, 397
 domyślna, 300
 powitalna, 79, 83
 SOAP, 45, 187
 WCF, 68
usprawnianie API, 197
usuwanie
 nieistniejącego zgłoszenia błędu, 192
 zgłoszenia błędu, 191
uwierzytelnianie, 42, 351
 AWS, 43
 filtry, 375
 Hawk, 210, 385
 klienta, 361
 klienta w proxy, 460
 klienta w serwerze źródłowym, 459, 461
 OAuth 2.0, 413
 oparte na tokenie, 378
 oparte na HTTP, 369
 oparte na transporcie, 357
 oprogramowanie aktywne, 374
 oprogramowanie pasywne, 374
 oprogramowanie pośredniczące, 370
 podstawowe, 368, 373
 przepływ zdarzeń, 459
 serwera, 357

użycie
 atrybutu `AuthorizeAttribute`, 388
 procedur obsługi odpowiedzi, 346
 Service Bus, 286
 TLS, 349
 TLS z hostingiem IIS, 349
 TLS z samohostowaniem, 351
 tokenu JWT, 381

W

warstwa
 hostingu, 91
 obsługi kontrolera, 96
 pośrednia, 139
 potoku procedur obsługi komunikatów, 93
WCF, windows communication foundation, 68
Web API, 14, 17, 45–65, 72
wersjonowanie, 110
 oparte na treści, 111
 typu danych, 111
 w adresie URL, 112
weryfikacja modelu, 312
Windows Azure, 284
Windows Identity Foundation, 355
właściwości metody, 35
 bezpieczeństwo, 35
 buforowalność, 35
 idempotentność, 35
WSDL, web service description language, 45, 125
WSGI, 93
współdzielenie zasobów, 391
wstrzykiwanie zależności, 426
wybór
 akcji, 303
 formatu, 145
 `HttpParameterBinding`, 318
 kontrolera, 297, 299
 typu danych, 125
 Web API, 73
wyjątki, 310
wykrywanie konfliktów, 206
wymiana komunikatów HTTP, 30
wyniki weryfikacji, 332
wyszukiwanie zgłoszeń błędów, 184

wywołanie
 akcji, 313
 procedury obsługi komunikatu, 295

X

XML, 27
xUnit.NET, 422

Z

zakres, 406
zasoby, 54
 chronione, 401
 osadzone, 139
 wyszukiwania danych, 120, 184
zasób, 25
 elementu, 120, 155
 główny, 119
 wykrycia, 156
 wyszukiwania, 157
zawartość komunikatu, 247
zgłoszenie błędu, 163, 463
zmiana, 209, 230
znacznik `<form>`, 138

Ż

źródło typu danych, 28

Ż

żądania
 warunkowe, 35
 HTTP, 89
 uprawnienia autoryzacji, 405
żądanie, 225
 DELETE, 34
 GET, 33, 455–457
 HEAD, 34
 OPTIONS, 34
 PATCH, 34
 POST, 34
 PUT, 34, 457, 458
 SOAP, 68
 TRACE, 35

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



- 1. ZAREJESTRUJ SIĘ**
- 2. PREZENTUJ KSIĄŻKI**
- 3. ZBIERAJ PROWIZJĘ**

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 **Helion SA**

Większość z najpopularniejszych dużych witryn internetowych zbudowano z użyciem technologii Web API. Może ona działać na podstawie protokołu HTTP – umożliwia wówczas łatwą pracę nad nowymi funkcjonalnościami aplikacji, kwestiami bezpieczeństwa, skalowalnością rozwiązań, obsługą nowych urządzeń lub aplikacji klienckich. W takim modelu aplikacji kluczowymi komponentami są typy danych i hipermedia. Wielu twórców API nie wykorzystuje jednak możliwości protokołu HTTP i wdraża API silnie powiązane z aplikacją klienta, co w zasadzie zamyka drogę do bezproblemowego ewoluowania systemu. Na dłuższą metę tego rodzaju rozwiązania są sprzeczne z podstawami internetu.

Niniejsza książka to wartościowy podręcznik dla projektantów, którzy chcą tworzyć API adaptujące się do zachodzących zmian. Autorzy zaprezentowali tu wszystkie narzędzia niezbędne do tworzenia ewoluujących systemów, a także przedstawili informacje dotyczące sieci i programowania Web API. Dokładnie omówili proces tworzenia nowego API za pomocą platformy ASP.NET Web API, z uwzględnieniem takich zagadnień jak implementacja hipermediów z użyciem ASP.NET Web API oraz negocjowanie treści. Ten bardzo praktyczny podręcznik, napisany przez inżynierów oprogramowania, stanowi inspirację do projektowania najlepszych rozwiązań dla ewoluujących aplikacji internetowych.

W tej książce znajdziesz:

- zwięzłe podstawy budowy sieci, protokołu HTTP, programowania API oraz platformy ASP.NET Web API
- omówienie nowego modelu programowania HTTP na platformie .NET
- objaśnienia dotyczące różnych modeli hostingu, w tym samohostowania, IIS i modelu OWIN
- przedstawienie zasad działania routingu Web API i kontrolerów
- praktyczną prezentację zagadnień bezpieczeństwa aplikacji

Glenn Block – współtwórca ASP.NET, odpowiedzialny za przygotowanie jednej z poprzednich wersji ASP.NET Web API.

Pablo Cibraro – od czternastu lat projektuje i implementuje ogromne systemy rozproszone w centrach Microsoft Technologies.

Pedro Felix – naukowiec i inżynier oprogramowania, specjalizuje się w problematyce infrastruktury środowiska uruchomieniowego, w zarządzaniu tożsamością i w kwestiach związanych z kontrolą dostępu do danych.

Howard Dierking – menedżer programu w zespole WCF Web API w firmie Microsoft.

Darrel Miller – współzałożyciel Tavis Software, specjalizuje się w stosowaniu stylu architektonicznego REST w aplikacjach biznesowych.

Helion

44487 numer katalogowy

księgarnia internetowa

<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

Sprawdź najnowsze promocje:
 ● <http://helion.pl/promocje>
 Książki najchętniej czytane:
 ● <http://helion.pl/bestsellery>
 Zamów informacje o nowościach:
 ● <http://helion.pl/nowosci>

Helion SA
 ul. Kosciuszki 1c, 44-100 Gliwice
 tel.: 32 230 98 63
 e-mail: helion@helion.pl
<http://helion.pl>

sięgnij po WIĘCEJ



KOD KORZYŚCI

ISBN 978-83-283-2391-9



9 788328 323919

Informatyka w najlepszym wydaniu

cena: 79,00 zł