Microsoft

Programowanie w ASP.NET Core



Professional

Dino Esposito

Microsoft

Dino Esposito

Programowanie w ASP.NET Core

Przekład: Jakub Niedźwiedź

APN Promise, Warszawa 2018

Programowanie w ASP.NET Core

Authorized Polish translation of the English language edition entitled Programming ASP.NET Core, by Dino Esposito, ISBN: 978-1-50-930441-7

Copyright © 2017 by Pearson Education, Inc.

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 Pearson Education, Inc.

Polish language edition published by APN PROMISE SA Copyright © 2017

Autoryzowany przekład z wydania w języku angielskim, zatytułowanego: Programming ASP.NET Core, by Dino Esposito, ISBN: 978-1-50-930441-7

Wszystkie prawa zastrzeżone. Żadna część niniejszej książki nie może być powielana ani rozpowszechniana w jakiejkolwiek formie i w jakikolwiek sposób (elektroniczny, mechaniczny), włącznie z fotokopiowaniem, nagrywaniem na taśmy lub przy użyciu innych systemów bez pisemnej zgody wydawcy.

APN PROMISE SA, ul. Domaniewska 44a, 02-672 Warszawa
tel. +48 22 35 51 600, fax +48 22 35 51 699
e-mail: mspress@promise.pl

Książka ta przedstawia poglądy i opinie autora. Przykłady firm, produktów, osób i wydarzeń opisane w niniejszej książce są fikcyjne i nie odnoszą się do żadnych konkretnych firm, produktów, osób i wydarzeń, chyba że zostanie jednoznacznie stwierdzone, że jest inaczej. Ewentualne podobieństwo do jakiejkolwiek rzeczywistej firmy, organizacji, produktu, nazwy domeny, adresu poczty elektronicznej, logo, osoby, miejsca lub zdarzenia jest przypadkowe i niezamierzone.

Microsoft oraz znaki towarowe wymienione na stronie *http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx* są zastrzeżonymi znakami towarowymi grupy Microsoft. Wszystkie inne znaki towarowe są własnością ich odnośnych właścicieli.

APN PROMISE SA dołożyła wszelkich starań, aby zapewnić najwyższą jakość tej publikacji. Jednakże nikomu nie udziela się rękojmi ani gwarancji.
APN PROMISE SA nie jest w żadnym wypadku odpowiedzialna za jakiekolwiek szkody będące następstwem korzystania z informacji zawartych w niniejszej publikacji, nawet jeśli APN PROMISE została powiadomiona o możliwości wystąpienia szkód.

ISBN: 978-83-7541-364-9

Przekład: Jakub Niedźwiedź
Redakcja: Marek Włodarz
Korekta: Ewa Swędrowska
Skład i łamanie: MAWart Marek Włodarz

Spis treści

O autorze x
Wprowadzenie xi

Część I: Rzut oka na nową technologię ASP.NET

1 Dlaczego kolejna wersja ASP.NET? 3
Obecna platforma .NET 4
Najważniejsze cechy platformy .NET 4
.NET Framework 4
Platforma ASP.NET 5
Platforma Web API 7
Potrzeba niezwykle prostych usług Web 7
.NET 15 lat później 8
Bardziej zwarta platforma .NET Framework 9
Oddzielenie ASP.NET od serwera 10
Nowa platforma ASP.NET Core 11
Narzędzia wiersza pleceń .NET Core 12
Instalowanie narzędzi CLI 12
Narzędzie sterownika *dotnet* 13
Predefiniowane polecenia *dotnet* 14
Podsumowanie 15

2 Pierwszy projekt ASP.NET Core 17
Anatomia projektu ASP.NET Core 17
Struktura projektu 18
Interakcja ze środowiskiem uruchomieniowym 25
Podsystem wstrzykiwania zależności 33
Rzut oka na wstrzykiwanie zależności 33
Wstrzykiwanie zależności w ASP.NET Core 34
Integracja z zewnętrznymi bibliotekami wstrzykiwania zależności 38
Budowanie mini witryny Web 40
Tworzenie witryny Web z pojedynczym punktem końcowym 41
Dostęp do plików na serwerze Web 48
Podsumowanie 53

Część II: Model aplikacji ASP.NET MVC

3 Ładowanie ASP.NET MVC . . . 57

Włączanie modelu aplikacji MVC . . . 57
Rejestrowanie usługi MVC . . . 58
Włączanie konwencjonalnego routingu . . . 61
Konfigurowanie tabeli routingu . . . 65
Anatomia trasy . . . 65
Zaawansowane aspekty routingu . . . 72
Mapa maszynerii ASP.NET MVC . . . 76
Element wywołujący akcje . . . 77
Przetwarzanie wyników akcji . . . 78
Filtry akcji . . . 78
Podsumowanie . . . 79

4 Kontrolery ASP.NET MVC . . . 81

Klasy kontrolerów . . . 81
Ustalanie nazwy kontrolera . . . 81
Dziedziczone kontrolery . . . 83
Kontrolery POCO . . . 85
Akcje kontrolerów . . . 89
Mapowanie akcji do metod . . . 89
Routing oparty na atrybutach . . . 93
Implementacja metod akcji . . . 98
Podstawowe pobieranie danych . . . 98
Wiązanie modelu . . . 99
Wyniki akcji . . . 107
Filtry akcji . . . 111
Anatomia filtrów akcji . . . 111
Mała galeria filtrów akcji . . . 115
Podsumowanie . . . 119

5 Widoki ASP.NET MVC . . . 121

Serwowanie zawartości HTML . . . 121
Serwowanie HTML z kończącego oprogramowania pośredniego . . . 121
Serwowanie HTML z kontrolerów . . . 122
Serwowanie HTML ze stron Razor . . . 123
Silnik widoków . . . 124

Wywoływanie silnika widoków 124
Silnik widoków Razor 126
Dodawanie niestandardowego silnika widoków 133
Struktura widoku Razor 134
Przekazywanie danych do widoku 139
Wbudowane słowniki 139
Silnie typowane modele widoków 143
Wstrzykiwanie danych poprzez system wstrzykiwania zależności 145
Strony Razor 146
Uzasadnienie dla stron Razor 146
Implementacja stron Razor 146
Przesyłanie danych ze strony Razor 148
Podsumowanie 150

6 Składnia Razor 151

Elementy składni 151
Przetwarzanie wyrażeń kodu 152
Szablony układów 158
Widoki częściowe 161
Pomocnicy znacznikowi w Razor 163
Korzystanie z pomocników znacznikowych 164
Wbudowani pomocnicy znacznikowi 165
Pisanie niestandardowych pomocników znacznikowych 170
Komponenty widoków Razor 174
Pisanie komponentu widoku 174
Wzorzec Composition UI 176
Podsumowanie 178

Część III: Najważniejsze zagadnienia

7 Uwarunkowania projektowe 181

Infrastruktura wstrzykiwania zależności 181
Refaktoring w celu wyizolowania zależności 181
Ogólne zasady systemu wstrzykiwania zależności w ASP.NET Core . . 185
Aspekty pojemnika wstrzykiwania zależności 189
Wstrzykiwanie danych i usług w warstwach 189
Zbieranie danych konfiguracyjnych 191
Obsługiwani dostawcy danych 192

Budowanie modelu obiektów dokumentu dla danych konfiguracyjnych . . . 194
Przekazywanie danych konfiguracyjnych . . . 198
Architektura warstwowa . . . 200
Warstwa prezentacyjna . . . 202
Warstwa aplikacyjna . . . 203
Warstwa domenowa . . . 204
Warstwa infrastrukturalna . . . 205
Obsługa wyjątków . . . 205
Oprogramowanie pośrednie do obsługi wyjątków . . . 205
Filtry wyjątków . . . 209
Rejestrowanie wyjątków . . . 211
Podsumowanie . . . 213

8 Zabezpieczanie aplikacji . . . 215

Infrastruktura zabezpieczeń Web . . . 215
Protokół HTTPS . . . 215
Obsługa certyfikatów zabezpieczeń . . . 216
Stosowanie szyfrowana w HTTPS . . . 216
Uwierzytelnianie w ASP.NET Core . . . 217
Uwierzytelnianie oparte na plikach cookie . . . 217
Obsługa wielu schematów uwierzytelniania . . . 219
Modelowanie tożsamości użytkowników . . . 222
Uwierzytelnianie zewnętrzne . . . 227
Uwierzytelnianie użytkowników poprzez ASP.NET Identity . . . 233
Ogólne zasady ASP.NET Identity . . . 233
Praca z menedżerem użytkowników . . . 239
Zasady autoryzacji . . . 244
Autoryzacja oparta na rolach . . . 244
Autoryzacja oparta na zasadach . . . 248
Podsumowanie . . . 255

9 Dostęp do danych aplikacji . . . 257

W stronę ogólnego mechanizmu wewnętrznego aplikacji . . . 258
Aplikacje monolityczne . . . 259
Podejście wykorzystujące CQRS . . . 260
Wewnątrz warstwy infrastrukturalnej . . . 262

Dostęp do danych w .NET Core 264
Entity Framework 6.x 264
Adaptery ADO.NET 266
Wykorzystanie mikroplatform O/RM 269
Wykorzystanie magazynów NoSQL 271
Typowe zadania EF Core 273
Modelowanie bazy danych 273
Praca z danymi tabelarycznymi 278
Obsługa transakcji 284
Kilka słów na temat asynchronicznego przetwarzania danych 286
Podsumowanie 289

Część IV: Po stronie klienta

10 Projektowanie interfejsu Web API 293

Budowanie interfejsu Web API w ASP.NET Core 293
Udostępnianie punktów końcowych HTTP 294
Serwery plików 297
Projektowanie interfejsu typu REST 298
Rzut oka na REST 299
REST w ASP.NET Core 303
Zabezpieczanie interfejsu Web API 307
Planowanie naprawdę potrzebnych zabezpieczeń 307
Prostsze metody kontroli dostępu 308
Wykorzystanie serwera zarządzania tożsamością 310
Podsumowanie 318

11 Przekazywanie danych ze strony klienta 319

Organizowanie formularzy HTML 319
Definiowanie formularza HTML 320
Wzorzec Post-Redirect-Get 325
Przesyłanie formularzy przez JavaScript 328
Ładowanie zawartości formularza 328
Odświeżanie fragmentów bieżącego ekranu 334
Ładowanie plików do serwera Web 335
Podsumowanie 339

12 Wiązanie danych po stronie klienta 341
Odświeżanie widoku poprzez HTML 341
Przygotowywanie gruntu 342
Definiowanie obszarów, które mogą być odświeżane 342
Łączenie wszystkiego razem 343
Odświeżanie widoku poprzez JSON 348
Wprowadzenie do biblioteki Mustache.JS 349
Wprowadzenie do biblioteki KnockoutJS 354
Budowanie aplikacji Web przy pomocy Angular 361
Podsumowanie 361

13 Budowanie widoków przyjaznych dla różnych urządzeń 363
Adaptowanie widoków do faktycznego urządzenia 364
Scenariusze najlepszego wykorzystania funkcji HTML5 dla urządzeń 364
Wykrywanie funkcji 368
Wykrywanie urządzeń po stronie klienta 369
Client Hints 373
Obrazy przyjazne dla urządzeń 374
Element PICTURE 374
Platforma ImageEngine 376
Automatyczne zmienianie rozmiarów obrazów 376
Strategie programowania zorientowanego na urządzenia 379
Strategie koncentrujące się na kliencie 379
Strategie koncentrujące się na serwerze 383
Podsumowanie 385

Część V: Ekosystem ASP.NET Core

14 Środowisko uruchomieniowe ASP.NET Core 389
Host ASP.NET Core 389
Klasa *WebHost* 390
Niestandardowe ustawienia hostingu 394
Wbudowany serwer HTTP 400
Wybór serwera HTTP 400
Konfigurowanie odwróconego serwera proxy 403
Parametry konfiguracyjne serwera Kestrel 406
Oprogramowanie pośrednie ASP.NET Core 408
Architektura potoku przetwarzania 408

Pisanie składników oprogramowania pośredniego 412
Pakowanie składników oprogramowania pośredniego 416
Podsumowanie 419

15 Wdrażanie aplikacji ASP.NET Core 421

Publikowanie aplikacji 422
Publikowanie z poziomu Visual Studio. 422
Publikowanie przy użyciu narzędzi CLI. 428
Wdrażanie aplikacji 430
Wdrażanie na serwerze IIS 430
Wdrażanie w Microsoft Azure 433
Wdrażanie na serwerach Linux 438
Kontenery Docker 441
Kontenery kontra maszyny wirtualne 441
Od kontenerów do architektury mikrousług 442
Docker i Visual Studio 2017 442
Podsumowanie 444

16 Strategie migracji i przystosowania 445

W poszukiwaniu wartości biznesowej 446
Szukanie korzyści 446
Rozwijanie starszych aplikacji 451
Rozwijanie nowych aplikacji 452
Strategia pośrednia 455
Radzenie sobie z brakującymi zależnościami 456
.NET Portability Analyzer. 456
Windows Compatibility Pack 458
Odkładanie obsługi wieloplatformowej na później 458
Droga do architektury mikrousług 460
Podsumowanie 462

Indeks 463

O autorze

Dino Esposito jest cyfrowym strategiem w firmie BaxEnergy i jest autorem ponad 20 książek i 1000 artykułów. Jego kariera programistyczna trwa od 25 lat. Powszechnie uważa się, że jego książki i artykuły pomogły w zawodowym rozwoju wielu tysiącom programistów i architektów .NET na całym świecie. Dino zaczął jako programista C w roku 1992 i był świadkiem debiutu .NET, rozwoju i wycofania Silverlight oraz wzlotów i upadków różnych wzorców architektonicznych. Obecnie przygląda się Artificial Intelligence 2.0 oraz Blockchain i jest autorem sztuki teatralnej „The Sabbatical Break", w której łączy swoje zainteresowania oprogramowaniem, literaturą, nauką, sportem, technologią i sztuką. Można się z nim skontaktować pod adresami:

http://youbiquitous.net
http://twitter.com/despos
http://instagram.com/desposofficial
http://facebook.com/desposofficial

Wprowadzenie

„Potrzeba nam ludzi, którzy marzą o rzeczach nieistniejących i pytają, dlaczego one nie istnieją."

– Prezydent John F. Kennedy,
przemówienie w irlandzkim parlamencie,
czerwiec 1963

Niektóre aspekty historii ASP.NET Core przypominają mi początki przygody z ASP.NET ponad 15 lat temu. Bardzo młody wtedy Scott Guthrie (obecnie wiceprezes Microsoft) prezentował nową technologię o nazwie ASP+ przed niewielką widownią programistów Web w Londynie jesienią 1999 roku. Były to czasy technologii Active Server Pages, a technologia ASP+ próbowała wprowadzić nową składnię, aby przenieść kod VBScript z powrotem na serwer i wyrazić go poprzez język kompilowany. Technologia ASP+ była prawdziwym przełomem.

Podczas tej prezentacji platforma .NET nie była jeszcze znana szerszej publiczności, miała być dopiero publicznie ujawniona następnego lata. Kod demonstracyjny pokazywany przez Scotta, w tym przykład usługi Web, pochodził z samodzielnego środowiska uruchomieniowego opartego na niestandardowym procesie roboczym (aplikacji konsolowej) mogącym nasłuchiwać na porcie 80. Pierwsze programy demonstracyjne wykorzystywały zwykły kod Visual Basic i C++ z użyciem interfejsu API Win32. W krótkim czasie cały pomysł ASP+ został szybko wchłonięty przez nową platformę .NET Framework i ostatecznie przekształcił się w ASP.NET.

Technologia ASP.NET Core również została najpierw przedstawiona jako nowa, samodzielna platforma napisana od podstaw w celu przeniesienia stosu Web firmy Microsoft na wyższy poziom skalowalności i wydajności. Jednakże podczas tego procesu zespół dostrzegł kuszącą okazję udostępnienia ASP.NET Core na wielu platformach docelowych. Aby zrealizować ten zamiar, podzbiór .NET Framework musiał być udostępniony na platformach docelowych, a to oznaczało konieczność utworzenia nowej platformy .NET Framework. Tak właśnie w końcu się stało.

Przez długi czas cel technologii ASP.NET Core nie był dokładnie sprecyzowany, a mechanizmy określające ten cel nie były zbyt jasne i nie zawsze były odpowiednio przedstawiane. Dwadzieścia lat temu brakowało nam (na szczęście?) podejścia

związanego z natychmiastowym udostępnianiem wszystkiego, co wymuszają obecnie media społecznościowe. Cel ASP+ prawdopodobnie też nie był dokładnie określony, ale nikt poza zaangażowanymi osobami w firmie Microsoft o tym nie wiedział.

Podczas gdy filary historii ASP.NET i ASP.NET Core mogą wydawać się takie same, to warunki działania są dosyć różne. Sieć Web przed ASP.NET była jeszcze w powijakach, z ograniczoną dostępnością skalowalnych technologii po stronie serwera, a sama skalowalność nie była tak poważnym problemem, jak obecnie. Jednocześnie wiele aplikacji potencjalnie było gotowych do przepisania na platformę Web i oczekiwało na niezawodną platformę od solidnego dostawcy.

Obecnie istnieje wiele platform, które można by wykorzystać zamiast ASP.NET Core. Jednakże technologia ASP.NET Core nie jest wykorzystywana jedynie od strony klienta, działa też po stronie serwera, jako interfejs Web API oraz kompaktowe monolity wdrażane samodzielnie lub w ramach struktury usług. Technologia ASP.NET Core może być też wykorzystywana na wielu platformach sprzętowych i programowych.

Trudno naprawdę powiedzieć, czy technologia ASP.NET Core jest koniecznością w najbliższej przyszłości (lub nawet obecnie) w każdej firmie i w każdym zespole. Na pewno jest to naturalny wybór dla programistów ASP.NET i jest kolejnym wcieleniem pełnego rozwiązania stosu programistycznego Web dla różnych platform.

Kto powinien przeczytać tę książkę

Ta książka nie jest dla całkiem początkujących, bez co najmniej ogólnego zrozumienia programowania Web. Jest skrojona na miarę dla programistów ASP.NET, zwłaszcza tych, którzy korzystali z MVC. Jednocześnie książka ta może się dobrze nadawać dla zaawansowanych programistów Web znających wzorzec MVC, ale bez znajomości ASP.NET. Choć platforma ASP.NET Core jest całkiem nowa, to ma wiele wspólnych punktów z ASP.NET MVC (i w mniejszym stopniu z Web Forms).

Korzystającym ze stosu Web firmy Microsoft, platforma ASP.NET Core oferuje świetny wybór dla całego stosu, w tym ścisłe powiązanie z chmurą Azure.

Założenia

Autor książki oczekuje, że Czytelnik rozumie programowanie Web w stopniu co najmniej podstawowym, najlepiej (ale niekoniecznie) z wykorzystaniem stosu oprogramowania firmy Microsoft.

Może nie być odpowiednia, jeśli...

Jeśli ktoś jest całkowitym nowicjuszem w programowaniu Web, który nigdy nie słyszał o ASP.NET i szuka podstawowego przewodnika ASP.NET Core, ta książka nie będzie idealnym wyborem.

Organizacja tej książki

Książka ta jest podzielona na pięć części:

- Część I „Rzut oka na nową technologię ASP.NET" zapewnia szybki przegląd podstaw ASP.NET Core i wprowadza przykładową aplikację typu „hello world".
- Część II „Model aplikacji ASP.NET MVC" skupia się na modelu aplikacyjnym MVC i zarysowuje jego podstawowe elementy, takie jak kontrolery i widoki.
- Część III „Najważniejsze zagadnienia" dotyczy typowych aspektów programowania Web, takich jak uwierzytelnianie, konfiguracja i dostęp do danych.
- Część IV „Po stronie klienta" jest poświęcona technologiom i dodatkowym platformom służącym do budowania użytecznej i skutecznej warstwy prezentacyjnej.
- Część V „Ekosystem ASP.NET Core" zajmuje się środowiskiem uruchomieniowym, wdrażaniem i strategiami migracji.

Wymagania systemowe

Do ukończenia ćwiczeń praktycznych z tej książki potrzebny będzie następujący sprzęt i oprogramowanie:

- System Windows 7 lub nowszy albo MacOS 10.12 lub nowszy.
- Alternatywnie można wykorzystać jedną z wielu dystrybucji systemu Linux, zgodnie z opisem dostępnym pod adresem: *https://docs.microsoft.com/en-us/dotnet/core/linux-prerequisites*.
- Visual Studio 2015 (dowolne wydanie) lub nowsza wersja; Visual Studio Code.
- Połączenie internetowe do pobierania oprogramowania lub przykładów kodu.

Pobieranie: przykłady kodu

Cały kod przedstawiony w tej książce, włącznie z ewentualnymi erratami i rozszerzeniami można znaleźć pod adresem *https://aka.ms/ASPNetCore/downloads*.

Errata, aktualizacje i wsparcie

Dokonaliśmy wszelkich starań, aby zapewnić dokładność informacji w tej książce i jej materiałach towarzyszących. Aktualizacje tej książki mogą być dostępne w formie errata i listy poprawek pod adresem:

https://aka.ms/ASPNetCore/errata

W przypadku odkrycia błędu, który nie jest wymieniony na tej liście, prosimy o przesłanie nam informacji, korzystając z formularza na tej samej stronie.

Jeśli potrzebne jest dodatkowe wsparcie, wystarczy wysłać e-maila do Microsoft Press Book Support pod adres *mspinput@microsoft.com*.

Należy zwrócić uwagę, że wsparcie techniczne dla produktów programowych i sprzętowych firmy Microsoft nie jest oferowane pod powyższymi adresami. Pomoc dotyczącą oprogramowania lub sprzętu firmy Microsoft można uzyskać pod adresem *http://support.microsoft.com*.

Bądźmy w kontakcie

Zachęcamy do podzielenia się swoimi uwagami na temat tej książki. Jesteśmy na Twitterze: *http://twitter.com/MicrosoftPress*.

CZĘŚĆ I

Rzut oka na nową technologię ASP.NET

Witamy w ASP.NET Core!

Minęło już ponad 15 lat, odkąd firma Microsoft wprowadziła technologię ASP.NET i platformę .NET Framework. Oczywiście programowanie Web znacznie zmieniło się w tym czasie. Programiści wiele się nauczyli, a klienci chcą radykalnie innych rozwiązań dostarczanych na nowe sposoby dla nowych urządzeń. ASP.NET Core odpowiada na wszystkie te zapotrzebowania i przewiduje, co może się zdarzyć w przyszłości. Część I pokazuje ASP.NET Core w szerszym kontekście i pomaga szybko rozpocząć pracę w tej technologii.

Rozdział 1, *Dlaczego kolejna wersja ASP.NET?* wyjaśnia, dlaczego technologia ASP.NET Core w ogóle powstała, gdzie może wydawać się znajoma (szczególnie dla programistów ASP.NET MVC) i gdzie jest znacząco różna. Zbadamy ASP.NET Core w kontekście kompaktowej, modularnej, opartej na otwartym kodzie i wieloplatformowej technologii .NET Core Framework oraz zobaczymy, jak pozwala ona lepiej obsługiwać zarówno minimalne usługi Web, jak i rozbudowane witryny internetowe. Przyjrzymy się też nieco narzędziom programistycznym działającym w wierszu poleceń.

Następnie w rozdziale 2, *Pierwszy projekt w ASP.NET Core*, szybko utworzymy swoją pierwszą aplikację. Niektóre rzeczy nigdy się nie zmieniają, więc nasza pierwsza aplikacja będzie utrzymana w znajomej konwencji „Hello World". Nawet tutaj jednak spotkamy się z uderzającym minimalizmem ASP.NET Core i zobaczymy, na co nam pozwala.

ROZDZIAŁ 1

Dlaczego kolejna wersja ASP.NET?

Jeśli chcemy, aby rzeczy zostały po staremu, będą musiały się zmienić.

– Giuseppe Tomasi di Lampedusa, „Lampart"

Było to chyba latem 1999 roku. Pisanie oprogramowania dla systemu operacyjnego Windows wymagało wtedy umiejętności programowania w C/C++, a wielkie biblioteki, takie jak Microsoft Foundation Classes (MFC) i ActiveX Template Library (ATL) ułatwiały programowanie. Model COM (Component Object Model) stawał się podstawą każdej aplikacji działającej w systemie Windows. Wszystko, włącznie z dostępem do danych, miało być przeprojektowane na zgodność z COM. Jednakże wybór języka programowania i narzędzi programistycznych nadal był istotnym wyróżnikiem, zwłaszcza jeśli w aplikacji Windows konieczny był dostęp do danych lub złożony interfejs użytkownika. Wybierając Visual Basic, można było mieć trywialnie prosty dostęp do baz danych oraz szybki i ładny interfejs użytkownika, ale nie dało się korzystać ze wskaźników do funkcji i nie można było mieć łatwego i niezawodnego dostępu do wszystkich funkcji w Windows SDK. Z drugiej strony, wybierając C lub C++, brakowało wysokopoziomowych udogodnień w dostępie do danych, a budowanie menu lub paska narzędzi było znacznie trudniejsze w porównaniu do Visual Basic.

Dla profesjonalnych programistów nie były to łatwe czasy, ale każdy starał się znaleźć swoją wygodną niszę i udawało nam się dość dobrze rozwijać swoje przedsięwzięcia. Nagle jednak pojawiła się platforma .NET i wszystko się zmieniło. Na szczęście zmieniło się na lepsze.

Obecna platforma .NET

Platforma .NET została ogłoszona latem 2000 roku i osiągnęła etap beta rok później. Wersja 1.0 została wydana na początku 2002 roku, choć w skali rozwoju oprogramowania można to traktować, jakby minęło kilka epok geologicznych.

Najważniejsze cechy platformy .NET

Platforma .NET składa się z zestawu klas i maszyny wirtualnej zwanej CLR (Common Language Runtime – środowisko uruchomieniowe wspólnego języka). CLR jest w zasadzie środowiskiem wykonawczym dla kodu zapisanego koncepcyjnie w języku pośrednim (IL) podobnie jak w przypadku kodu bajtowego języka Java. CLR zapewnia uruchamianemu kodowi rozmaite usługi, takie jak zarządzanie pamięcią i odśmiecanie pamięci, obsługę wyjątków, bezpieczeństwo, wersjonowanie, debugowanie i profilowanie. Przede wszystkim jednak CLR może zapewniać te usługi w sposób niezależny od języka programowania.

Na bazie CLR działają kompilatory języków oraz istnieje pojęcie „języka zarządzanego". Język zarządzany jest zwykłym językiem programowania, dla którego istnieje kompilator, który może generować kod IL wykorzystywany przez CLR. Każdy kompilator .NET generuje kod IL, ale kod IL nie jest uruchamiany bezpośrednio w docelowym systemie operacyjnym Windows. Potrzebne jest kolejne narzędzie – kompilator just-in-time (JIT). Kompilator ten zamienia kod IL na kod binarny, który może być wykonywany na danej platformie sprzętowej/programowej.

.NET Framework

Aspektem .NET, który od początku mnie najmocniej poruszył, była możliwość mieszania różnych języków programowania w tym samym projekcie. Można było łatwo utworzyć bibliotekę powiedzmy w Visual Basic i wywoływać ją z kodu napisanego w dowolnym innym języku zarządzanym. Zaoferowany został też nowy, niezwykle mocny język programowania – obecnie szeroko rozpowszechniony język C#, który narodził się jak legendarny feniks z popiołów języka Java.

Największą zmianą dla programistów była możliwość dostępu z klas do większości elementów Windows SDK. Biblioteka klas bazowych (BCL – Base Class Library) stanowi wspólne podłoże kodu dla dowolnej aplikacji .NET. Biblioteka BCL jest zbiorem typów, które są ściśle zintegrowane z CLR, takich jak typy prymitywne, LINQ oraz klasy i typy pomocne w najczęstszych operacjach, takich jak wejście/wyjście, daty, kolekcje i diagnostyka.

Biblioteka BCL jest uzupełniana przez zestaw dodatkowych, wysoko wyspecjalizowanych bibliotek, takich jak ADO.NET do obsługi dostępu do baz danych, Windows

Forms do tworzenia aplikacji pulpitowych dla Windows, ASP.NET do tworzenia aplikacji Web, XML, itd. Przez lata ten zestaw dodatkowych bibliotek rozrósł się i obejmuje gigantyczne zestawy klas, takie jak Windows Presentation Foundation (WPF), Windows Communication Foundation (WCF) oraz Entity Framework (EF).

Biblioteka BCL wraz z dodatkowymi bibliotekami razem tworzą platformę .NET Framework.

Platforma ASP.NET

Jesienią 1999 roku firma Microsoft zaczęła ujawniać nową platformę Web, mającą zastąpić Active Server Pages (ASP). W pierwszych publicznych demonstracjach platforma ta była nazywana ASP+ i była oparta na swoim własnym silniku C/C++, który następnie został zastąpiony przez .NET, co dało obecną platformę ASP.NET.

Platforma ASP.NET składa się z rozszerzenia usług IIS (Internet Information Services), mogącego przechwytywać przychodzące żądania HTTP i przetwarzać je w środowisku uruchomieniowym ASP.NET. W tym środowisku uruchomieniowym żądanie jest obsługiwane przez znajdowanie specjalnego komponentu, który może przetworzyć dane żądanie oraz przygotowanie pakietu odpowiedzi HTTP dla przeglądarki. Struktura środowiska uruchomieniowego jest potokowa. Żądanie przechodzi przez różne etapy aż zostanie w pełni przetworzone, a odpowiedź jest zwracana z powrotem do strumienia wyjściowego.

W odróżnieniu od swoich konkurentów, platforma ASP.NET zapewniała oparty na zdarzeniach i zachowujący stan model programowania, który pozwalał na przekazywanie kontekstu pomiędzy kolejnymi żądaniami. Model ten był dobrze znany programistom aplikacji biurkowych i otworzył świat programowania Web przed wieloma programistami, którzy dotąd nie znali HTML i JavaScript. Dzięki grubej warstwie abstrakcji dla HTTP i HTML dostępnej w ASP.NET technologia ta przyciągnęła tłumy programistów Visual Basic, Delphi, C/C+, a nawet Java.

Model Web Forms

Pierwotnie środowisko uruchomieniowe ASP.NET było opracowywane z uwzględnieniem dwóch głównych celów:

- Pierwszym celem było zapewnienie modelu programistycznego, który mógłby jak najbardziej odgrodzić programistów od HTML i JavaScript. Model Web Forms, który był głęboko inspirowany klasycznym modelem żądań klient/serwer, działał świetnie i stworzył ekosystem darmowych i komercyjnych składników serwerowych oferujących coraz więcej zaawansowanych możliwości, takich jak inteligentne siatki danych, formularze do wprowadzania danych, kreatory, kalendarze do wyboru dat itd.

- Drugim celem było jak największe połączenie ASP.NET oraz IIS. Platforma ASP.NET miała być operacyjnym skrzydłem IIS, a nie jedynie wtyczką. Jej środowisko uruchomieniowe miało stać się strukturalną częścią IIS. Ten kamień milowy został w pełni osiągnięty wraz z wydaniem IIS 7 w 2008 roku. Tryb Integrated Pipeline w IIS 7 (i nowszych wersjach) jest trybem roboczym, w którym IIS oraz ASP.NET wykorzystują ten sam potok przetwarzania. Ścieżka, przez którą przechodzi żądanie, które trafia do IIS, jest tą samą ścieżką, przez którą przechodziłoby ono w ASP.NET. Kod ASP.NET jest po prostu odpowiedzialny za przetwarzanie żądania i za przechwytywanie i wstępne przetwarzanie dodatkowych wymaganych żądań.

Około roku 2009 model programowania Web Forms został uzupełniony przez platformę ASP.NET MVC, która została zainspirowana całkiem inną zasadą, stanowiąc zupełne odwrócenie pierwotnych celów ASP.NET. W modelu Web Forms strony ASP.NET generują kod HTML poprzez kontrolki serwerowe, które stanowią główny powód sukcesu i szybkiego rozpowszechnienia się ASP.NET. Te kontrolki serwerowe są „czarnymi skrzynkami" (konfigurowanymi deklaratywnie lub programowo), które generują kod HTML i JavaScript dla przeglądarki. Programista ma jednak ograniczoną kontrolę nad generowanym kodem HTML, a powszechne wymagania zmieniały się z biegiem czasu.

Model ASP.NET MVC

Platforma ASP.NET MVC została zaprojektowana od podstaw tak, aby blisko współdziałała z protokołem HTTP; nie próbuje ukrywać żadnych funkcji HTTP i wymaga od programistów świadomości mechanizmów żądań i odpowiedzi HTTP. Programiści wykorzystujący ASP.NET MVC powinni mieć opanowane umiejętności związane z JavaScript i CSS. ASP.NET MVC jest wynikiem gruntownej przeróbki modelu programistycznego opartej na nowych wymaganiach, takich jak rozdzielenie odpowiedzialności, modularyzacja i podatność na testowanie.

Prawdopodobnie była to ciężka decyzja, ale technologia ASP.NET MVC nie otrzymała swojego własnego środowiska uruchomieniowego i została zakodowana jako wtyczka dla istniejącego środowiska ASP.NET. To jednocześnie dobra i zła wiadomość. To dobra wiadomość, ponieważ możemy obsługiwać przychodzące żądania albo poprzez model Web Forms, albo model ASP.NET MVC, co ułatwia rozpoczęcie prac od istniejącej aplikacji Web Forms i stopniowe przenoszenie jej do ASP.NET MVC. To jednak zła wiadomość, ponieważ można było usunąć bardzo niewiele strukturalnych wad ASP.NET (z punktu widzenia nowoczesnych wymagań). Zespołowi ASP.NET MVC udało się na przykład umożliwić wykorzystanie atrap dla całego kontekstu HTTP, ale nie można było wbudować w tę platformę pełnej i kanonicznej infrastruktury wstrzykiwania zależności.

Model programowania ASP.NET MVC jest przy tym najbardziej elastycznym i zrozumiałym sposobem obsługi żądań Web, które muszą zwracać zawartość HTML. Jednak

w pewnym momencie przy szalonym rozwoju platform mobilnych zawartość HTML przestała być jedynym możliwym wynikiem żądania HTTP.

Platforma Web API

W szczególności żądanie dla danego punktu końcowego Web może dotyczyć dowolnego typu zawartości (na przykład JSON, XML, obrazy, PDF). Dowolny fragment kodu, który może zgłaszać żądania HTTP, jest potencjalnym klientem punktu końcowego Web. Poziom skalowalności pewnych rozwiązań stał się elementem krytycznym.

W dziedzinie ASP.NET nie dało się już wiele zrobić w celu rozszerzenia infrastruktury, aby dobrze funkcjonowała w nowych scenariuszach, takich jak niezwykła skalowalność, chmura i niezależność od platformy. Platforma Web API była próbą zaoferowania tymczasowego rozwiązania dla dużego zapotrzebowania na cienkie serwery mogące udostępniać interfejs REST i będące w stanie prowadzić dialog z dowolnym klientem HTTP bez żadnych założeń i ograniczeń. Platforma Web API jest alternatywnym zestawem klas do tworzenia punktów końcowych HTTP, które mają być świadome pełnej składni i semantyki HTTP. Platforma Web API oferuje interfejs programistyczny niemal identyczny jak ASP.NET MVC; zawiera kontrolery, routing i wiązanie modeli, ale wykonuje je w całkiem nowym środowisku uruchomieniowym.

W przypadku ASP.NET Web API możemy tworzyć platformę Web oddzieloną od serwera Web, co doprowadziło do zdefiniowania otwartego interfejsu Web dla standardu .NET (OWIN). OWIN jest specyfikacją, która ustanawia reguły dla współdziałania serwera Web i aplikacji Web. W przypadku OWIN drugi spośród oryginalnych celów ASP.NET (silne i ciasne powiązanie pomiędzy hostem Web a aplikacją Web) został porzucony jako nieaktualny.

Web API może być obsługiwany w dowolnej aplikacji, która jest zgodna ze standardem OWIN. Jednakże użyteczność wymaga, aby interfejs Web API był obsługiwany w IIS, co wymaga aplikacji ASP.NET. Użycie Web API wewnątrz aplikacji ASP.NET (Web Forms lub MVC) zwiększa zużycie pamięci przez aplikację, ponieważ wykorzystywane są dwa środowiska uruchomieniowe.

Potrzeba niezwykle prostych usług Web

Inną znaczącą zmianą w krajobrazie branży oprogramowania, która pojawiła się w ostatnich latach, jest potrzeba minimalnych, niezwykle prostych usług Web – jedynie cienkiej warstwy serwera Web wokół elementu logiki biznesowej.

Minimalny serwer Web jest punktem końcowym HTTP, który może być wywoływany przez klienta w celu uzyskania niezwykle podstawowej, najczęściej tekstowej zawartości. Taki serwer Web nie musi uruchamiać skomplikowanego i konfigurowalnego potoku przetwarzania. Musi jedynie odebrać żądanie HTTP, odpowiednio je przetworzyć i zwrócić odpowiedź HTTP. Wszystko to powinno się odbywać bez

dodatkowych narzutów albo jedynie z narzutami wymaganymi przez kontekst. Zastosowanie modeli programowania po stronie klienta (takich jak Angular) napędza potrzebę takich usług Web.

Platforma ASP.NET i wszystkie jej środowiska uruchomieniowe nie są zaprojektowane dla tego rodzaju scenariuszy. Choć środowisko uruchomieniowe ASP.NET (które obsługuje zarówno aplikacje Web Forms, jak i MVC) można do pewnego stopnia dostosowywać (wyłączać obsługę sesji, pamięci podręcznej danych wyjściowych a nawet uwierzytelnianie), to nie ma takiego poziomu szczegółowości i kontroli, jakie są wymagane obecnie przez niektóre scenariusze biznesowe. Dla przykładu niemal niemożliwe jest zamienienie ASP.NET w skuteczny serwer plików statycznych.

.NET 15 lat później

Piętnaście lat to całkiem sporo dla każdego rodzaju oprogramowania i .NET Framework nie jest tu wyjątkiem. Platforma ASP.NET została opracowana w późnych latach 90-ych a Web ewoluuje bardzo szybko. Około roku 2014 zespół ASP.NET zaczął planować nową wersję ASP.NET i opracował całkiem nowe środowisko uruchomieniowe spełniające dość ściśle specyfikację OWIN.

Usunięcie wszelkich zależności od starego środowiska uruchomieniowego ASP.NET – symbolizowanego przez podzespół *System.Web* – stało się głównym celem zespołu. Jednak kolejnym istotnym celem zespołu było danie programistom pełnej kontroli nad potokiem przetwarzania, aby możliwe stało się zbudowanie zarówno minimalnej usługi Web, jak i pełnej witryny Web. Robiąc to, zespół stanął przed kolejnym nietrywialnym problemem: jak zapewnić przepustowość i uczynić każde rozwiązanie wydajnym kosztowo w chmurze obliczeniowej – w tym celu rozmiar aplikacji musiał być drastycznie ograniczony. Również platforma .NET Framework musiała przejść specjalną kurację odchudzającą.

Wytyczne dla nowej platformy ASP.NET można by podsumować następująco:

- Sprawić, aby platforma ASP.NET była w stanie zarówno mieć dostęp do pełnej, istniejącej platformy .NET Framework, jak i do uproszczonej jej wersji pozbawionej wszelkich rzadziej używanych i mniej przydatnych dla programistów Web zależności.
- Odłączenie nowego środowiska ASP.NET od serwera Web.

Gdy jednak zaimplementowano ten plan, pojawiło się sporo nowych problemów i możliwości. Postanowiono się więc nimi zająć.

Bardziej zwarta platforma .NET Framework

Nowa platforma ASP.NET była projektowana jednocześnie z nową platformą .NET Framework, która w końcu została nazwana .NET Core Framework. Tę nową platformę można traktować jako podzbiór oryginalnej platformy .NET Framework zaprojektowany specjalnie tak, aby był bardziej szczegółowy, zwarty i wieloplatformowy. Ten cel projektowy został osiągnięty na dwa sposoby: przez porzucenie pewnych funkcjonalności i ponowne przepisanie innych funkcjonalności w celu poprawienia wydajności w pewnych przypadkach i zastąpienia istniejących zależności od porzuconych funkcjonalności.

Platforma .NET Core Framework była przede wszystkim projektowana do działania z aplikacjami ASP.NET. Kierowało to wyborem, które biblioteki dołączyć, a które porzucić. Platforma .NET Core Framework posiada nowe środowisko uruchomieniowe do wykonywania aplikacji zwane CoreCLR. CoreCLR stosuje ten sam układ i architekturę, co obecne środowisko .NET CLR i zajmuje się takimi rzeczami, jak ładowanie kodu IL, kompilowanie do kodu maszynowego oraz odśmiecanie pamięci. CoreCLR nie obsługuje pewnych funkcji obecnego środowiska CLR, takich jak domeny aplikacji oraz bezpieczeństwo dostępu do kodu, które okazały się niepotrzebne lub zbyt specyficzne dla platformy Windows i trudne do przeniesienia na inne platformy sprzętowo-programowe. Co więcej, zestaw bibliotek klas w .NET Core Framework jest podzielony na pakiety, a pakiety te mają duży poziom szczegółowości i są znacznie mniejsze niż obecna platforma .NET Framework.

Cała platforma .NET Core jest w pełni oparta na otwartym kodzie źródłowym. Łącza do repozytoriów są pokazane w tabeli 1-1.

Tabela 1-1 Łącza do kodu źródłowego .NET Core w serwisie Github

Platforma	Opis	Łącze
CoreCLR	CLR i powiązane narzędzia	*http://github.com/dotnet/coreclr*
CoreFX	.NET Core Framework	*http://github.com/dotnet/corefx*

Krótko mówiąc, różnice pomiędzy pełną platformą .NET Framework a .NET Core Framework można podsumować w następujących punktach:

- Platforma .NET Core Framework jest bardziej zwarta i modularna.
- Platforma .NET Core Framework (i powiązane z nią narzędzia) jest oparta na otwartym kodzie źródłowym.
- Platforma .NET Core Framework nie może być używana do pisania kodu innego niż aplikacje ASP.NET i aplikacje konsolowe.
- Platforma .NET Core Framework może być wdrażana razem z aplikacją, natomiast pełna platforma .NET Framework może być jedynie instalowana na komputerze

docelowym i wspólnie wykorzystywana przez wszystkie aplikacje. Jak widać sprawia to nietrywialny problem związany z wersjonowaniem.

Pozbawiona zależności od konkretnej platformy sprzętowo-programowej, nowa i bardziej zwarta platforma .NET stanowi też kod, który można adaptować do działania w różnych, alternatywnych systemach operacyjnych. To sprawia kolejną, wielką różnicę pomiędzy platformą .NET Core Framework a istniejącą platformą .NET Framework. Platforma .NET Core Framework może być wykorzystywana do pisania aplikacji międzyplatformowych, które mogą też działać w systemach operacyjnych Linux i Mac.



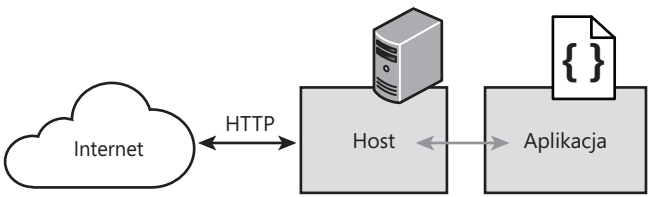
Uwaga Wraz z wydaniem wersji .NET Core 2.0, luka funkcjonalna pomiędzy pełną platformą .NET Framework a .NET Core Framework zmniejsza się, ponieważ więcej klas i przestrzeni nazw zostało przeniesionych do platformy Core Framework (na przykład *System.Drawing* oraz klasy dla tabel danych). Jednak traktowanie .NET Core Framework jako kopii pełnej platformy .NET Framework jest błędem. Jest to osobna platforma zaprojektowana od podstaw, która wygląda bardzo podobnie i działa w sposób wieloplatformowy.

Oddzielenie ASP.NET od serwera

Aby sprostać wymaganiu, żeby model aplikacji Web mógł być wykorzystywany do pisania zarówno minimalnych usług Web, jak i pełnych witryn Web, odłączenie ASP.NET od IIS okazało się koniecznym krokiem. Cała filozofia OWIN (zobacz *http://owin.org*) oznacza:

- Oddzielenie funkcji serwera Web od funkcji aplikacji Web.
- Zachęcanie do programowania prostszych modułów dla .NET, które połączone razem mogą osiągnąć pełne wymagania rzeczywistej witryny Web.

Rysunek 1-1 pokazuje ogólną architekturę OWIN.



Rysunek 1-1 Architektura otwartego interfejsu Web

Gdy mamy architekturę opartą na OWIN, serwerem Web nie musi być już koniecznie IIS. Interfejs serwera może być też zaimplementowany przez aplikację konsolową lub usługę systemu Windows. Poza tymi ograniczonymi scenariuszami, pełna moc modelu

aplikacji Web zainspirowanego przez otwarty interfejs OWIN przejawia się w tym, że ta sama aplikacja może być obsługiwana przez dowolny zgodny serwer Web niezależnie od platformy systemowej.

HTTP jest protokołem niezależnym od platformy i gdy nowa wersja platformy .NET Framework została zbudowana bez ścisłych zależności od konkretnej platformy systemowej (takiej jak Windows), budowanie modelu aplikacji Web, który działa w sposób wieloplatformowy, stało się nagle realistycznym i atrakcyjnym projektem.



Ważne W roku 2008, gdy serwer IIS zaczął obsługiwać tryb Integrated Pipeline, wizja Web firmy Microsoft była całkiem odmienna od obecnej wizji. Do pewnego stopnia cały świat był wtedy inny. Zgodnie z wizją Integrated Pipeline serwer IIS oraz ASP.NET musiały współpracować ze sobą i stanowiły połączony silnik. Model zbudowany dla nowej platformy ASP.NET przewraca tę wizję do góry nogami, określając, że ASP.NET jest autonomicznym środowiskiem, które może być obsługiwane przez dowolny serwer Web. Ten model stwierdza, że to autonomiczne środowisko mogłoby nawet działać (w pewnych sytuacjach), będąc bezpośrednio dostępnym publicznie.

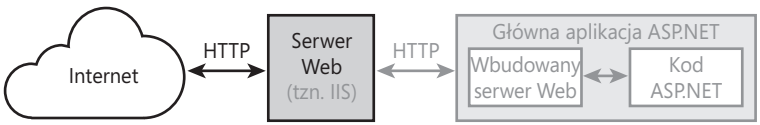
Nowa platforma ASP.NET Core

ASP.NET Core jest nową platformą do budowania rozmaitych aplikacji opartych na Internecie, przede wszystkim (ale nie tylko) aplikacji Web. W istocie specjalne odmiany aplikacji Web mogą być związane z serwerami osadzonymi w *Internecie rzeczy* (IoT) oraz usługami Web, na przykład serwerowe części aplikacji mobilnych.

Aplikacje ASP.NET Core mogą być pisane dla platformy .NET Core Framework lub dla istniejącej pełnej platformy .NET Framework. ASP.NET Core jest wieloplatformowe, więc programiści mogą tworzyć aplikacje działające w systemach Windows, Mac i Linux. ASP.NET Core składa się z osadzonego serwera Web oraz środowiska uruchomieniowego wykonującego kod aplikacji. Kod aplikacji jest pisany przy użyciu nieco przerobionej platformy ASP.NET MVC i opiera się na kolekcji modułów systemowych, które mają być niezwykle małe, co daje większą możliwość budowania aplikacji o minimalnych narzutach. Rysunek 1-2 przedstawia ogólną architekturę ASP.NET Core.



Uwaga Serwer Web, taki jak IIS lub Apache nie jest ściśle wymagany, ponieważ wbudowany serwer Web (Kestrel) może być bezpośrednio udostępniany. Zapotrzebowanie na osobny serwer Web zależy głównie od tego, czy Kestrel spełnia nasze wymagania, czy nie.



Rysunek 1-2 Ogólna architektura ASP.NET Core

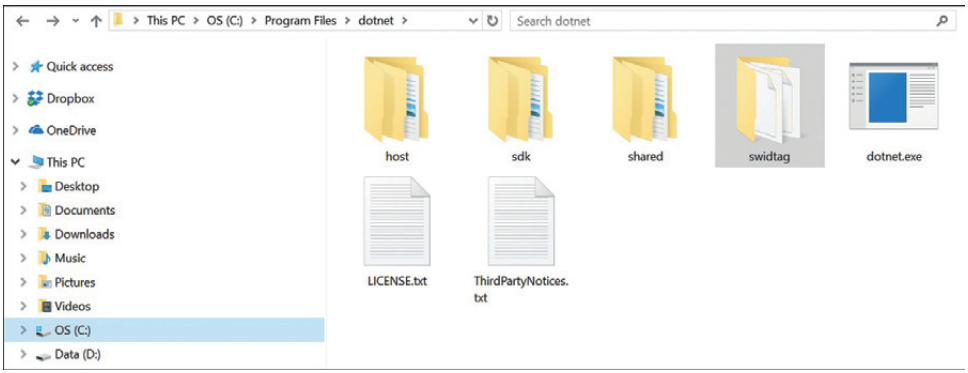
Nowa platforma ASP.NET opiera się na narzędziach .NET Core SDK przy budowaniu i uruchamianiu aplikacji. W dalszej części rozdziału poznamy dokładniej .NET SDK i narzędzia wiersza poleceń. Środowisko uruchomieniowe ASP.NET Core omówię dokładnie w rozdziale14 „Środowisko uruchomieniowe ASP.NET Core".

Narzędzia wiersza pleceń .NET Core

W .NET Core cały zestaw podstawowych narzędzi programistycznych (używanych do budowania, testowania, uruchamiania i publikowania aplikacji) jest też dostępny jako aplikacje wiersza poleceń. Aplikacje te są łącznie określane mianem interfejsu wiersza poleceń .NET Core (CLI – Command-line Interface).

Instalowanie narzędzi CLI

Narzędzia CLI są dostępne dla wszystkich platform programistycznych i wdrożeniowych, które mogą być celem aplikacji .NET Core. Zwykle mają pakiet instalacyjny dostosowany do danej platformy, jak na przykład pakiety RPM lub DEB dla systemu Linux oraz pakiety MSI dla systemu Windows. Po uruchomieniu instalatora narzędzia CLI są bezpiecznie zachowywane w globalnie dostępnym miejscu na dysku. Rysunek 1-3 pokazuje folder z narzędziami CLI na komputerze z systemem Windows.



Rysunek 1-3 Zainstalowane narzędzia CLI

Warto zwrócić uwagę, że możemy mieć wiele wersji narzędzi CLI działających obok siebie. Gdy zainstalowanych jest wiele wersji, to domyślnie uruchamiane są najnowsze.

Narzędzie sterownika *dotnet*

Narzędzia CLI zwykle oznaczają zbiór narzędzi, ale jest to właściwie kolekcja poleceń wykonywanych przez narzędzie hosta zwane sterownikiem. Tym narzędziem jest *dotnet.exe* (zobacz rysunek 1-3). Instrukcja wiersza poleceń przyjmuje następującą postać:

```
dotnet [opcje-hosta] [polecenie] [argumenty] [opcje-ogólne]
```

[polecenie] oznacza polecenie do wykonania w narzędziu sterownika, natomiast *[argumenty]* oznacza argumenty przekazywane do tego polecenia. Opcje hosta i ogólne są opisane dokładnie poniżej.

Gdy zainstalowanych jest wiele wersji CLI, a nie chcemy uruchamiać najnowszej, możemy utworzyć plik *global.json* w folderze aplikacji i umieścić w nim następującą opcję:

```
{
  "sdk": {
    "version": "2.0.0"
  }
}
```

Wartość właściwości *version* określa wersję narzędzia CLI, z której chcemy korzystać.



Uwaga Ta wersja narzędzi CLI nie jest taka sama, jak wersja środowiska uruchomieniowego .NET Core, z którego będzie korzystać aplikacja. Wersja środowiska uruchomieniowego jest określana w pliku projektu i możemy ją swobodnie edytować z poziomu interfejsu użytkownika wybranego środowiska programistycznego. Jeśli natomiast chcemy edytować plik projektu ręcznie, to wystarczy edytować plik XML *.csproj* i zmienić wartość elementu *TargetFramework*. Wartość ta odnosi się do oznaczenia identyfikującego wersję (na przykład *netcoreapp2.0)*.

Opcje hosta

W wierszu polecenia narzędzia *dotnet* opcje hosta są przekazywane przed nazwą polecenia i odnoszą się do konfiguracji narzędzia *dotnet*. Obsługiwane są trzy wartości związane z uzyskiwaniem informacji o narzędziu i środowisku uruchomieniowym, wyświetlaniem numeru wersji CLI oraz włączaniem diagnostyki (zobacz tabela 1-2).

Tabela 1-2 Opcje hosta w CLI

Opcja	Opis
`-d` lub `--diagnostics`	Włącza wyświetlanie informacji diagnostycznych.
`--info`	Wyświetla informacje o środowisku uruchomieniowym oraz .NET CLI.
`--version`	Wyświetla numer wersji .NET CLI.

Opcje ogólne

Ogólne opcje CLI w tabeli 1-3 oznaczają wspólne opcje dla wszystkich poleceń, takie jak uzyskiwanie pomocy lub włączanie pełnych danych wyjściowych.

Tabela 1-3 Opcje ogólne CLI

Opcja	Opis
`-v` lub `--verbose`	Włącza pełne dane wyjściowe.
`-h` lub `--help`	Pokazuje ogólną pomoc dotyczącą korzystania z narzędzia *dotnet*.

Predefiniowane polecenia *dotnet*

Domyślnie zainstalowanie narzędzi CLI udostępnia polecenia wymienione w tabeli 1-4. Kolejność poleceń w tej tabeli stara się odzwierciedlać realistyczną kolejność ich użycia.

Tabela 1-4 Typowe polecenia CLI

Polecenie	Opis
`new`	Tworzy nową aplikację .NET Core na bazie jednego z dostępnych szablonów. Domyślne szablony obejmują aplikacje konsolowe, a także aplikacje ASP.NET MVC, projekty testowe i biblioteki klas. Dodatkowe opcje pozwalają nam wskazać język docelowy i nazwę projektu.
`restore`	Przywraca wszystkie zależności dla danego projektu. Zależności są odczytywane z pliku projektu i przywracane jako pakiety NuGet pochodzące z pakietu konfiguracyjnego.
`build`	Buduje projekt i wszystkie jego zależności. Parametry dla kompilatorów (na przykład, czy budować bibliotekę, czy aplikację) powinny być określone w pliku projektu.
`run`	Kompiluje kod źródłowy, jeśli to konieczne; generuje plik wykonywalny i go uruchamia. Polega w pierwszym kroku na poleceniu *build*.

Tabela 1-4 Typowe polecenia CLI

Polecenie	Opis
`test`	Uruchamia testy jednostkowe w projekcie, wykorzystując skonfigurowane narzędzie do uruchamiania testów. Testy jednostkowe są bibliotekami klas z zależnością od określonej platformy testów jednostkowych i jej aplikacji wykonawczej.
`publish`	Kompiluje aplikację, jeśli to konieczne, wczytuje listę zależności z pliku projektu, a następnie publikuje wynikowy zestaw plików w katalogu docelowym.
`pack`	Tworzy pakiet NuGet z plików binarnych projektu.
`migrate`	Migruje stary projekt oparty na *project.json* do projektu opartego na *msbuild*.
`clean`	Czyści folder wyjściowy projektu.

Aby dowiedzieć się więcej na temat szczegółowego sposobu wywoływania któregoś z powyższych poleceń, można wpisać następujące polecenie w wierszu poleceń:

```
dotnet <polecenie> --help
```

Więcej poleceń można dodać, odwołując się do przenośnych aplikacji konsolowych w projekcie lub globalnie, kopiując plik wykonywalny do katalogu skojarzonego ze zmienną środowiskową *PATH*.

Podsumowanie

Platforma .NET istnieje od ponad 15 lat, a w całym tym czasie przyciągnęła wiele inwestycji i stała się bardzo popularna. Świat jednak stale się zmienia, a w pełni oddaje to słynny cytat z powieści „Lampart" Giuseppe Tomasi di Lampedusa na początku tego rozdziału – „Jeśli chcemy, aby rzeczy zostały po staremu, będą musiały się zmienić". Pierwotna platforma .NET, skupiona wokół pojedynczej, rozbudowanej biblioteki klas i kilku modeli aplikacji (ASP.NET, Windows Forms oraz WPF), przechodzi teraz znaczące przeprojektowanie. To przeprojektowanie, które rozpoczęło się w roku 2014, osiągnęło pierwszy znaczący kamień milowy w wersji 2.0 i będzie z pewnością trwało dalej w przyszłości.

Z punktu widzenia biznesu możemy czuć (lub nie) potrzebę przyjęcia tej nowej platformy, a myślę, że ta nowa platforma w ciągu kilku lat stanie się najczęściej wybieraną opcją. Najważniejszymi elementami tej nowej platformy są jej niezwykła modularność i natura wieloplatformowa. Dowolny kod pisany przez nas dla .NET Core będzie działał w systemach Linux, Mac lub Windows, choć w różnych środowiskach

uruchomieniowych. Ze względu na silną orientację na programowanie wieloplatformowe, wszystkie narzędzia działające na tej platformie (do budowania, uruchamiania, testowania i publikowania) są udostępniane jako narzędzia wiersza poleceń i są wykorzystywane przez środowiska IDE. Interfejs wiersza poleceń dla .NET Core nosi nazwę narzędzi CLI.

W następnym rozdziale zaczniemy skupiać się na głównym temacie tej książki – ASP.NET i programowaniu Web.

ROZDZIAŁ 2

Pierwszy projekt ASP.NET Core

Wszystkie zwierzęta są sobie równe, ale niektóre zwierzęta są równiejsze od innych.

– George Orwell, „Folwark zwierzęcy”

ASP.NET Core jest modelem aplikacji zorientowanych na Web, który działa na bazie platformy .NET Core. Choć nazwa tego modelu aplikacji zawiera stare i znane określenie ASP.NET, to tak naprawdę nic w ASP.NET Core nie jest takie samo, jak w poprzedniej wersji ASP.NET. Przede wszystkim ASP.NET Core ma całkiem nowe środowisko uruchomieniowe, które obsługuje pojedynczy model aplikacyjny – ASP.NET MVC. To oznacza, że ta nowa platforma Web nie ma nic podobnego do Web Forms ani nawet nic, takiego jak Web API. Wszystko jest całkiem nowe, a ponowne wykorzystanie kodu oraz dotychczasowych umiejętności jest możliwe jedynie w obszarze modelu programowania ASP.NET MVC – kontrolerach, widokach i trasach.



Ważne W tym rozdziale i w pozostałej części książki będziemy odwoływać się do funkcji i aspektów implementacyjnych ASP.NET niezwiązanych z .NET Core (w tym Web Forms, ASP.NET MVC oraz Web API) i porównywać je z funkcjami ASP.NET Core. Aby uniknąć nieporozumień, będziemy używać terminu *klasyczne ASP.NET* przy odwoływaniu się do modelu aplikacyjnego ASP.NET dostępnego przed ASP.NET Core.

Anatomia projektu ASP.NET Core

Istnieje kilka opcji utworzenia nowego projektu ASP.NET Core. Po pierwsze możemy skorzystać z jednego z kanonicznych szablonów projektów dostępnych w Visual Studio. Alternatywnie możemy skorzystać z polecenia New w narzędziu CLI. Jeśli wybierzemy inne zintegrowane środowisko programistyczne, takie jak JetBrains Rider, to mamy garść innych szablonów projektów ASP.NET do wyboru. Wreszcie, jeśli

chcemy po prostu mieć całkowitą kontrolę nad plikami wygenerowanymi dla projektu, to najlepszą opcją jest chyba użycie generatora ASP.NET w narzędziu Yeoman.

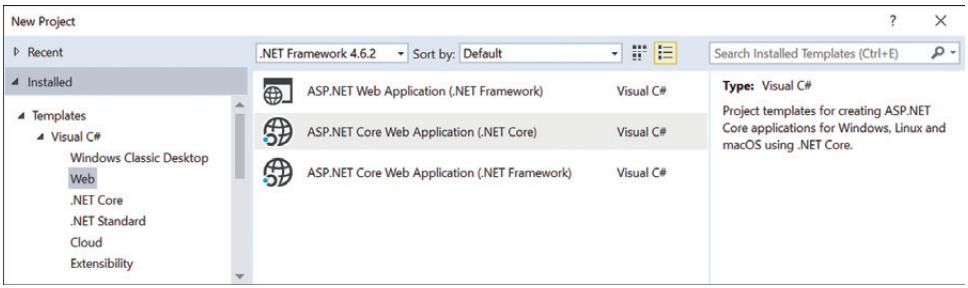
Yeoman jest niezależnym od języka generatorem projektów, który przy odpowiedniej konfiguracji może wygenerować wszystkie pliki składające się na szkielet aplikacji Web, w tym aplikacji ASP.NET Core (więcej informacji można uzyskać pod adresem *http://yeoman.io/learning*).



Uwaga Pliki projektów, które uzyskujemy, korzystając z Visual Studio, Rider, narzędzi CLI i Yeoman różnią się nieco między sobą. Visual Studio oferuje dwie opcje – projekt podstawowy i projekt pełny zawierający obsługę użytkowników oraz Bootstrap. Polecenie New w narzędziu CLI również generuje złożony projekt ASP.NET. Domyślna aplikacja ASP.NET Core w Rider jest czymś pośrednim pomiędzy pustym projektem a w pełni skonfigurowanym projektem pozbawionym logiki aplikacji. Yeoman jest chyba najbardziej elastycznym generatorem i oferuje kilka opcji do wyboru.

Struktura projektu

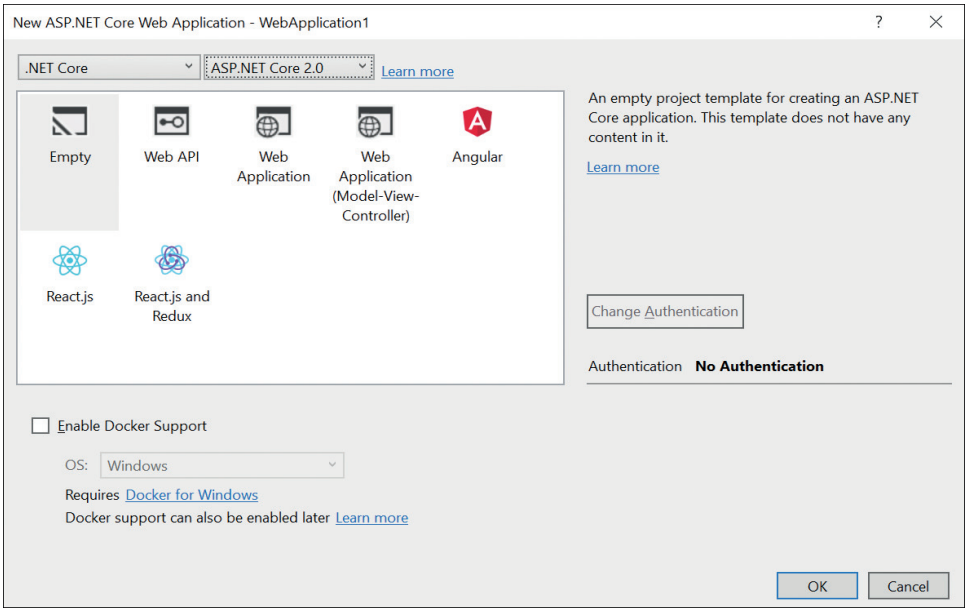
Jak można zobaczyć na rysunku 2-1, Visual Studio zawiera predefiniowane szablony do tworzenia klasycznych aplikacji Web (nie .NET Core) działających na pełnej platformie .NET Framework, a także aplikacji ASP.NET Core. Opcja zaznaczona na rysunku – ASP.NET Core Web Application (.NET Core) – tworzy aplikację ASP.NET Core działającą na platformie .NET Core.



Rysunek 2-1 Tworzenie nowego projektu ASP.NET Core w Visual Studio

Następny krok w tym kreatorze wymaga określenia zakresu kodu, który chcemy wygenerować dla początkowej aplikacji. Uważam, że przy poznawaniu platformy najlepszym podejściem jest zaczęcie od podstawowego, ale działającego projektu. W tym przypadku opcja Empty w Visual Studio jest opcją idealną (rysunek 2-2).

Po zatwierdzeniu wyboru Visual Studio utworzy kilka plików i skonfiguruje nowy projekt. W tym momencie możemy zbadać te pliki i spróbować zbudować z nich wykonywalną aplikację.



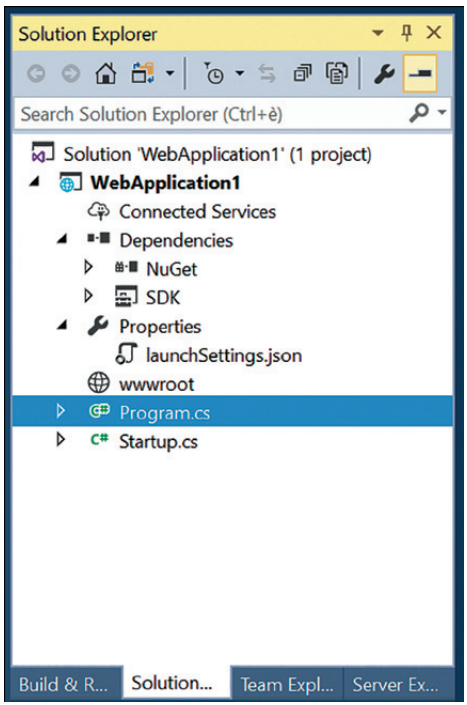
Rysunek 2-2 Wybieranie pustego projektu (Empty)

Pierwsze spojrzenie na pusty projekt

Zawartość tego rozwiązania może wywołać różne reakcje w zależności od posiadanego doświadczenia programistycznego. Na przykład programista ASP.NET zauważy nietypowy folder *wwwroot* i brak jednego z fundamentalnych plików we wcześniejszych aplikacjach ASP.NET: *global.asax*. Inny kluczowy plik konfiguracyjny dotychczasowych aplikacji ASP.NET – *web.config* – jest nadal obecny, ale jego zawartość znacznie różni się od oczekiwanej (zobacz rysunek 2-3).

Jak pokazuje rysunek 2-3, rozwiązanie zawiera dwa nowe pliki: *startup.cs* i *program.cs*. Dostępność pliku startup.cs może nie być zupełną niespodzianką, jeśli ktoś miał nieco do czynienia z platformami opartymi na OWIN, takimi jak Web API lub ASP.NET SignalR. Jednakże obecność pliku program.cs w aplikacji Web może być zaskoczeniem. Plik z programów konsolowych w aplikacji Web? Jak to możliwe?

Wszystko to jest związane z nową infrastrukturą uruchomieniową, która wykonuje aplikacje ASP.NET Core. Dowiedzmy się więcej na temat nowych elementów podstawowego projektu ASP.NET Core.



Rysunek 2-3 Zawartość eksploratora rozwiązania dla pustego projektu

Przeznaczenie foldera *wwwroot*

Jeśli chodzi o pliki statyczne, środowisko uruchomieniowe ASP.NET Core rozróżnia folder główny dla zawartości (content root) oraz folder główny dla Web (web root).

Folder główny dla zawartości jest zwykle bieżącym katalogiem projektu, a w środowisku produkcyjnym jest to główny folder, w którym wdrażana jest aplikacja. Reprezentuje ścieżkę bazową dla wszelkich wyszukiwań i dostępów do plików, które mogą być wymagane w kodzie. Natomiast *folder główny dla Web* jest ścieżką bazową dla wszelkich plików statycznych, które aplikacja może serwować klientom Web. Zazwyczaj folder główny dla Web jest podfolderem foldera głównego dla zawartości i ma nazwę *wwwroot*.

Ciekawą rzeczą jest, że folder główny dla Web musi być utworzony na komputerze produkcyjnym, ale jest on całkowicie przeźroczysty dla przeglądarek klienckich żądających plików statycznych. Innymi słowy, jeśli wewnątrz foldera *wwwroot* mamy podfolder *images* z plikiem o nazwie *banner.jpg*, to prawidłowym adresem URL pozwalającym na dostęp do tego obrazu jest:

```
/images/banner.jpg
```

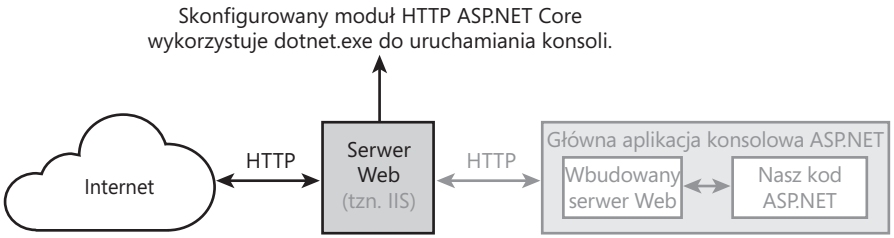
Fizyczny plik obrazu musi jednak znajdować się gdzieś wewnątrz foldera *wwwroot* na serwerze; w przeciwnym razie nie zostanie pobrany. Lokalizację obu folderów głównych można programowo zmienić w pliku *program.cs* (więcej na ten temat za chwilę).



Uwaga Wyraźne rozróżnienie na poziomie systemu pomiędzy folderem głównym dla zawartości a folderem głównym dla Web nie istnieje w klasycznym ASP.NET. W klasycznym ASP.NET folder główny dla zawartości jest automatycznie definiowany jako główny folder, w którym instalowana jest aplikacja. Posiadanie jasno zdefiniowanego foldera głównego dla Web jest jednak dobrą praktyką, która była implementowana przez większość zespołów i została przekształcona w funkcję systemową w ASP.NET Core. Osobiście lubię nazywać swój folder główny dla Web *Content*, ale widziałem, że wielu innych programistów nazywa go *Assets*. W każdym razie w klasycznym ASP.NET definicja foldera głównego dla Web jest wirtualna, a folder ten musi być zawarty w każdym adresie URL, który wskazuje na przechowywany wewnątrz plik statyczny.

Przeznaczenie pliku program.cs

Choć może to brzmieć dziwnie, aplikacja ASP.NET Core jest właściwie aplikacją konsolową uruchamianą przez narzędzie sterownika *dotnet*, z którym spotkaliśmy się już w rozdziale 1. Kod źródłowy tej (wymaganej) aplikacji konsolowej znajduje się w pliku *program.cs*. Rola tej aplikacji konsolowej jest dobrze zilustrowana na rysunku 2-4.



Rysunek 2-4 Ogólne przedstawienie sposobu działania aplikacji ASP.NET Core

Serwer Web (na przykład IIS) komunikuje się z plikiem wykonywalnym przez skonfigurowany port i przekazuje przychodzące żądanie do aplikacji konsolowej. Aplikacja konsolowa jest uruchamiana w przestrzeni procesów IIS pod kontrolą wymaganego modułu HTTP, który umożliwia IIS obsługę ASP.NET Core. Analogiczne moduły rozszerzeniowe są wymagane do obsługi aplikacji ASP.NET Core na innych serwerach Web, takich jak Apache lub NGINX.



Ważne Warto zauważyć, że architektura ASP.NET Core przedstawiona na rysunku 2-4 ma pewną analogię do oryginalnej architektury pochodzącej z roku 2003 i łączącej ASP.NET 1.x oraz IIS. Wtedy technologia ASP.NET miała swój własny proces roboczy komunikujący się z IIS poprzez nazwane potoki. Później zadania procesu roboczego ASP.NET zostały przejęte przez wbudowany proces roboczy IIS (w3wp.exe), tworząc pojęcie puli aplikacji. W ASP.NET Core dwa niezależne, niepowiązane i w pełni rozłączone elementy wykonywalne komunikują się ze sobą, ale element wykonywalny ASP.NET nie jest procesem roboczym obsługującym wiele aplikacji. Jest to po prostu wystąpienie aplikacji, które obsługuje podstawowy, asynchroniczny serwer Web w celu przetwarzania nadchodzących żądań.

Wewnętrznie aplikacja konsolowa jest zbudowana wokół następujących kilku wierszy kodu wziętych z pliku *program.cs*.

```
public static void Main(string[] args)
{
    var host = new WebHostBuilder()
         .UseKestrel()
         .UseContentRoot(Directory.GetCurrentDirectory())
         .UseIISIntegration()
         .UseStartup<Startup>()
         .Build();
    host.Run();
}
```

Aplikacja ASP.NET Core wymaga hosta, w którym będzie wykonywana. Host ten odpowiada za uruchomienie aplikacji i zarządzanie jej czasem życia. *WebHostBuilder* jest klasą odpowiedzialną za zbudowanie w pełni skonfigurowanego wystąpienia prawidłowego hosta ASP.NET Core. Tabela 2-1 krótko wyjaśnia zadania wykonywane przez metody wywoływane w powyższym fragmencie kodu.

Tabela 2-1 Metody rozszerzeniowe dla hosta ASP.NET Core

Metoda	Efekt
UseKestrel	Instruuje hosta, z jakiego wbudowanego serwera Web ma korzystać. Wbudowany serwer Web jest odpowiedzialny za przyjmowanie i przetwarzanie żądań HTTP w kontekście hosta. Kestrel jest nazwą domyślnego, wieloplatformowego, wbudowanego serwera ASP.NET.
UseContentRoot	Instruuje hosta, gdzie znajduje się folder główny zawartości.

Tabela 2-1 Metody rozszerzeniowe dla hosta ASP.NET Core

Metoda	Efekt
UseIISIntegration	Instruuje hosta, aby korzystał z IIS jako odwróconego serwera proxy, który będzie przechwytywał żądania z publicznego Internetu i przekazywał je do wbudowanego serwera. *Warto zauważyć, że w przypadku aplikacji ASP.NET Core posiadanie odwróconego serwera proxy może być zalecane ze względów bezpieczeństwa i utrzymania ruchu, ale nie jest konieczne z czysto funkcjonalnego punktu widzenia.*
UseStartup<T>	Instruuje hosta, jaki jest typ przechowujący ustawienia inicjujące aplikację.
Build	Buduje wystąpienie typu hosta ASP.NET Core.

Klasa *WebHostBuilder* ma całkiem sporo metod rozszerzeniowych, które mogą dostosowywać jej zachowanie.

Wersja ASP.NET Core 2.0 oferuje prostszy sposób na budowanie wystąpienia hosta Web. Korzystając z „domyślnego" elementu budującego, pojedyncze wywołanie może zwrócić świeżo utworzone wystąpienie hosta Web. Oto, jak można by przepisać plik *program.cs*.

```
public class Program
{
    public static void Main(string[] args)
    {
        BuildWebHostInstance(args).Run();
    }

    public static IWebHost BuildWebHostInstance(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>()
            .Build();
}
```

Statyczna metoda *CreateDefaultBuilder* wykonuje za nas całą pracę i dodaje serwer Kestrel, konfigurację IIS oraz folder główny zawartości, a także inne opcje, takie jak dostawców rejestrowania i dane konfiguracyjne (które do wersji ASP.NET Core 1.1 mogły być dodawane tylko w klasie *startup*). Najlepszym sposobem na zrozumienie, co robi dla nas metoda *CreateDefaultBuilder*, jest przyjrzenie się jej kodowi źródłowemu: *http://github.com/aspnet/MetaPackages/blob/dev/src/Microsoft.AspNetCore/WebHost.cs#L150*.

Przeznaczenie pliku startup.cs

Plik *startup.cs* zawiera klasę przeznaczoną do konfigurowania potoku przetwarzania żądań, obsługującego wszystkie żądania kierowane do aplikacji. Klasa ta zawiera co najmniej kilka metod, które host będzie wywoływać zwrotnie podczas inicjowania aplikacji. Pierwsza metoda nazywa się *ConfigureServices* i jest wykorzystywana do dodawania usług mechanizmu wstrzykiwania zależności, który będzie wykorzystywany przez aplikację. Obecność metody *ConfigureServices* w klasie startowej jest opcjonalna, ale w najbardziej realistycznych scenariuszach jest ona konieczna.

Druga metoda nazywa się *Configure* i jak sugeruje jej nazwa, służy do konfigurowania wcześniej żądanych usług. Na przykład, jeśli w metodzie *ConfigureServices* zadeklarowaliśmy zamiar wykorzystania usługi ASP.NET MVC, to w metodzie *Configure* możemy określić listę prawidłowych tras, które zamierzamy obsługiwać, wywołując metodę UseMvc z parametrem IApplicationBuilder. Metoda *Configure* jest wymagana. Klasa startowa nie musi implementować żadnego interfejsu ani dziedziczyć po żadnej klasie bazowej. Obie metody *Configure* oraz *ConfigureServices* są w istocie wykrywane i wywoływane poprzez refleksję.



Uwaga Choć może się to wydawać dziwne, sam ASP.NET Core pozwala nam pisać aplikacje Web, ale niekoniecznie aplikacje ASP.NET MVC z kontrolerami, widokami i trasami. Jeśli więc zamierzamy pisać kanoniczną aplikację ASP.NET MVC, musimy najpierw zażądać usług specyficznych dla MVC.

W pewnym sensie działania przeprowadzane w ramach klasy startowej przypominają operacje, które w klasycznym ASP.NET kodowalibyśmy w metodzie *Application_Start* w pliku *global.asax* oraz w niektórych sekcjach pliku *web.config*.

Warto zauważyć, że nazwa klasy startowej nie jest ściśle ustalona. Nazwa *Startup* jest sensownym wyborem, ale można ją zmienić, dopasowując do swoich upodobań. Jasne jest, że jeśli zmienimy nazwę klasy startowej, to musimy przekazać właściwy typ w wywołaniu *UseStartup<T>*. Warto też zwrócić uwagę, że metoda rozszerzeniowa *UseStartup* oferuje kilka dodatkowych przeciążeń, pozwalających wskazać klasę startową. Na przykład możemy przekazać jej nazwę jako łańcuch podzespołu klasy albo jako obiekt *Type*, jak pokazano poniżej.

```
// Wykorzystanie niekonwencjonalnej i nostalgicznej nazwy
// dla klasy startowej (GlobalAsax)
// ...
var host = new WebHostBuilder()
         .UseKestrel()
         .UseContentRoot(Directory.GetCurrentDirectory())
         .UseIISIntegration()
```

```
.UseStartup<GlobalAsax>()
.Build();
```



Ważne Jak wspominałem wcześniej, w tym rozdziale poruszamy tylko podstawowe zagadnienia związane ze środowiskiem uruchomieniowym ASP.NET i środowiskiem hosta. Celem jest przejście od razu do sedna budowania aplikacji i sprawienie, aby zachowywały się zgodnie z naszymi oczekiwaniami. Jednakże dogłębne przyjrzenie się środowisku uruchomieniowemu ASP.NET Core jest konieczne dla zrozumienia potencjału tej platformy oraz najlepszych sposobów jej wykorzystania nawet w różnych systemach operacyjnych. Przegląd systemu ASP.NET jest więc na miejscu i odbędzie się w rozdziale 14 „Środowisko uruchomieniowe ASP.NET Core".

Interakcja ze środowiskiem uruchomieniowym

Wszystkie aplikacje ASP.NET Core są obsługiwane w środowisku uruchomieniowym i wykorzystują kilka dostępnych usług. Dobra wiadomość jest taka, że liczba i jakość tych usług całkowicie zależy od zespołu programującego aplikację. Nie dostaniemy żadnych usług, których nie chcemy. Musimy też jawnie zadeklarować wszystkie usługi, które są nam potrzebne do działania aplikacji.



Uwaga Błędem, który często popełniałem na początku przy korzystaniu z platformy ASP.NET Core, było zapominanie o wymogu dołączenia usługi plików statycznych, co powodowało, że system odmawiał serwowania obrazów, czy plików JavaScript, nawet gdy były odpowiednio wdrożone w folderze głównym Web.

Następnie zajmiemy się interakcją pomiędzy aplikacją a środowiskiem hosta.

Ustalanie typu klasy startowej

Jednym z pierwszych zadań podejmowanych przez hosta jest ustalenie typu klasy startowej. Jawnie wskazujemy typ startowy jako dowolną nazwę albo poprzez ogólną metodę rozszerzeniową *UseStartup<T>*, albo przekazując ten typ jako parametr do wersji nieogólnej tej metody. Możliwe jest też przekazanie nazwy podzespołu zawierającego typ klasy startowej.

Konwencjonalną nazwą klasy startowej jest *Startup*, ale możemy ją zmienić zgodnie z własnymi upodobaniami. Jeśli jednak zachowamy konwencjonalną nazwę, uzyskamy kilka dodatkowych korzyści. W szczególności możemy w aplikacji mieć skonfigurowanych wiele klas startowych, po jednej dla każdego środowiska. Możemy mieć klasę

startową na użytek podczas programowania aplikacji i inne dla środowiska pośredniego i produkcyjnego. Dodatkowo możemy też definiować własne środowiska, jeśli chcemy.

Powiedzmy, że mamy w swoim projekcie klasy o nazwach *StartupDevelopment* oraz *StartupProduction* i wykorzystujemy następujący kod do utworzenia hosta:

```
var host = new WebHostBuilder()
    .UseKestrel()
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseIISIntegration()
    .UseStartup(Assembly.GetEntryAssembly().GetName().Name)
    .Build();
```

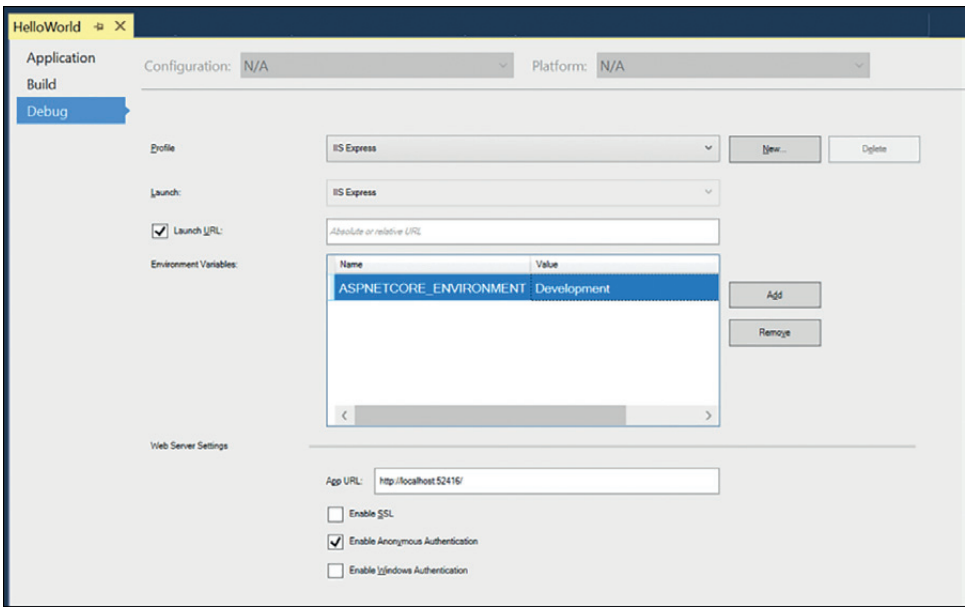
Nakazujemy teraz hostowi ustalenie klasy startowej na podstawie bieżącego podzespołu. W tym przypadku host spróbuje znaleźć nadającą się do załadowania klasę, która pasuje do następującego wzorca: *StartupXXX*, gdzie *XXX* jest nazwą aktualnego środowiska hosta. Domyślnie środowisko hosta jest ustawione na Production, ale może zostać zmienione na dowolny łańcuch tekstowy. Na przykład może to być Staging lub Development albo cokolwiek innego, co będzie miało dla nas sens. Jeśli środowisko hosta nie jest ustawione, to system po prostu spróbuje zlokalizować zwykłą klasę *Startup* i wyrzuci błąd, jeśli jej nie znajdzie.

Krótko mówiąc, zdecydowanie możemy zmienić nazwę klasy startowej, ale w bardziej realistycznej sytuacji chcielibyśmy, aby host ustalał nazwę tej klasy na podstawie bieżącego środowiska hosta. Świetnie, ale jak ustawić bieżące środowisko hosta?

Środowisko hosta

Środowisko Development wynika z wartości zmiennej środowiskowej o nazwie ASPNETCORE_ENVIRONMENT. W projekcie Visual Studio zmienna ta jest ustawiana domyślnie na *Development*, ale może być ustawiona na dowolny żądany łańcuch tekstowy, taki jak *Production* lub *Staging*.

Zmienna ASPNETCORE_ENVIRONMENT może być ustawiona w dowolny sposób, w jaki możemy ustawić zmienną środowiskową w danym systemie operacyjnym. Na przykład w systemie Windows możemy skorzystać z Panelu sterowania, konsoli PowerShell lub narzędzia *set* z wiersza poleceń. Oczywiście możemy też ustawić zmienną programowo lub z poziomu Visual Studio w oknie dialogowym Properties dla danego projektu, jak pokazano na rysunku 2-5. Trzeba mieć na uwadze, że jeśli z jakiegokolwiek powodu zmienna ASPNETCORE_ENVIRONMENT nie będzie ustawiona, to jako środowisko hosta przyjmowane jest Production.



Rysunek 2-5 Ustawianie zmiennej środowiskowej w Visual Studio

Konfiguracja środowiska hosta jest dostępna programowo poprzez elementy interfejsu *IHostingEnvironment* (tabela 2-2).

Tabela 2-2 Interfejs IHostingEnvironment

Element	Opis
ApplicationName	Zwraca lub ustawia nazwę aplikacji. Host ustawia wartość tej właściwości na nazwę podzespołu, który zawiera punkt wejścia do aplikacji.
EnvironmentName	Zwraca lub ustawia nazwę środowiska, nadpisując wartość zmiennej ASPNETCORE_ENVIRONMENT. Można użyć tej właściwości do programowego ustawienia środowiska.
ContentRootPath	Zwraca lub ustawia bezwzględną ścieżkę do katalogu, który zawiera pliki aplikacji. Ta właściwość jest zwykle ustawiana na główny folder ścieżki instalacyjnej.
ContentRootFileProvider	Zwraca lub ustawia składnik, który musi być używany do pobierania plików zawartości. Składnik ten może być dowolną klasą, która implementuje interfejs *IFileProvider*. Domyślny dostawca plików wykorzystuje system plików do pobierania plików.
WebRootPath	Zwraca lub ustawia bezwzględną ścieżkę do katalogu, który zawiera pliki statyczne, których klienci mogą żądać poprzez adresy URL.

Tabela 2-2 Interfejs IHostingEnvironment

Element	Opis
WebRootFileProvider	Zwraca lub ustawia składnik, który musi być używany do pobierania plików witryny Web. Składnik ten może być dowolną klasą, która implementuje interfejs *IFileProvider*. Domyślny dostawca plików wykorzystuje system plików do pobierania plików.

Interfejs *IFileProvider* reprezentuje dostawcę plików tylko do odczytu i działa, wykorzystując łańcuch tekstowy opisujący nazwę pliku lub katalogu i zwracając abstrakcję z żądaną zawartością. Interesująca, alternatywna implementacja interfejsu *IFileProvider* zwraca zawartość plików i katalogów z bazy danych.

Obiekt, który implementuje interfejs *IHostingEnvironment*, jest tworzony przez hosta i udostępniany publicznie klasie startowej oraz wszystkim innym klasom w aplikacji poprzez wstrzykiwanie zależności (więcej na ten temat w kolejnej części rozdziału „Włączanie usług systemowych i aplikacyjnych").



Uwaga Konstruktor klasy startowej może opcjonalnie otrzymywać odwołanie do pary usług systemowych: *IHostingEnvironment* oraz *ILoggerFactory*. W szczególności ta druga jest abstrakcją ASP.NET Core do tworzenia wystąpień składnika rejestrowania.

Włączanie usług systemowych i aplikacyjnych

Jeśli jest zdefiniowana, metoda *ConfigureServices* jest wywoływana przed metodą *Configure*, aby dać programistom szansę na podłączenie usług systemowych i aplikacyjnych do potoku przetwarzania żądań. Konfiguracja podłączonych usług może odbywać się bezpośrednio w metodzie *ConfigureServices* lub może być odłożona aż do wywołania metody *Configure*. Zależy to ostatecznie od interfejsu programistycznego danej usługi. Oto prototyp metody *ConfigureServices*.

```
public void ConfigureServices(IServiceCollection services)
```

Jak widać, metoda ta otrzymuje kolekcję usług i po prostu dodaje swoje własne usługi. W ogólności usługi, które mają znaczącą fazę przygotowawczą, zapewniają metodę rozszerzeniową *AddXXX* dla *IServiceCollection* i przyjmują kilka parametrów. W poniższym fragmencie kodu widać, jak dodać kontekst *DbContext* z Entity Framework do listy dostępnych usług. Metoda *AddDbContext* przyjmuje kilka opcji, takich jak dostawca baz danych oraz faktyczny łańcuch połączenia.

```
public void ConfigureServices(IServiceCollection services)
{
    var connString = "...";
    services.AddDbContext<YourDbContext>(options => options.
UseSqlServer(connString));
}
```

Dodanie usługi do pojemnika *IServicesCollection* udostępnia tę usługę w innych częściach aplikacji poprzez wbudowany w ASP.NET Core system wstrzykiwania zależności.

Konfigurowanie usług systemowych i aplikacyjnych

Metoda *Configure* jest używana do konfigurowania potoku przetwarzania żądań HTTP oraz określania modułów, które będą miały możliwość przetwarzania nadchodzących żądań HTTP. Moduły i luźny kod, które mogą być dodawane do potoku przetwarzania żądań HTTP, są zbiorczo nazywane oprogramowaniem pośrednim (*middleware*).

Metoda *Configure* otrzymuje wystąpienie obiektu systemowego, który implementuje interfejs *IApplicationBuilder* i będzie dodawać oprogramowanie pośrednie poprzez metody rozszerzeniowe tego interfejsu. Metoda *Configure* może też otrzymać wystąpienie składników *IHostingEnvironment* oraz *ILoggerFactory*. Oto możliwy sposób zadeklarowania tej metody.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
   ...
}
```

Bardzo powszechnym działaniem podejmowanym w metodzie *Configure* jest włączenie możliwości serwowania plików statycznych oraz scentralizowanej obsługi błędów.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
   app.UseExceptionHandler("/error/view");
   app.UseStaticFiles();
}
```

Metoda rozszerzeniowa *UseExceptionHandler* działa jako scentralizowany moduł obsługi błędów i przekierowuje żądanie do określonego adresu URL w przypadku wystąpienia nieobsłużonych wyjątków. Jej ogólne zachowanie jest analogiczne do metody *Application_Error* z *global.asax* w klasycznym ASP.NET. Aby otrzymywać w przypadku wyjątków komunikaty przydatne dla programisty, możemy zamiast tego użyć metody *UseDeveloperExceptionPage*. Moglibyśmy też chcieć oglądać komunikaty przydatne dla programisty tylko w trybie programowania. Ten scenariusz przedstawia świetny przypadek użycia dla pewnych metod rozszerzeniowych interfejsu *IHostingEnvironment*.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error/View");
    }

    app.UseStaticFiles();
}
```

Metody rozszerzeniowe, takie jak *IsDevelopment*, *IsProduction* oraz *IsStaging* są predefiniowanymi metodami sprawdzającymi aktualny tryb hosta. Jeśli zdefiniujemy niestandardowe środowisko, możemy je sprawdzać poprzez metodę *IsEnvironment*. W systemach Windows i Mac wielkość liter w nazwach środowisk nie ma znaczenia, ale ma znaczenie w systemie Linux.

Ponieważ kod pisany w metodzie *Configure* ostatecznie konfiguruje potok przetwarzania w środowisku uruchomieniowym, kolejność konfigurowania usług ma znaczenie. Z tego powodu pierwszą rzeczą, którą chcemy zrobić w metodzie *Configure*, jest ustawienie obsługi błędów oraz serwowania plików statycznych.

Metody konfiguracyjne specyficzne dla środowiska

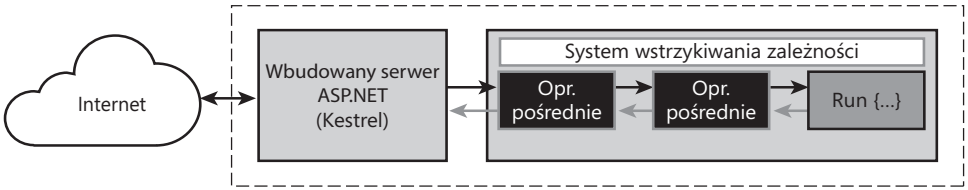
W klasie startowej nazwy metod *Configure* i *ConfigureServices* mogą mieć też postać specyficzną dla środowiska. Wzorzec wygląda następująco *ConfigureXxx* i *ConfigureXxxServices;* gdzie Xxx oznacza nazwę środowiska.

Utworzenie pojedynczej klasy startowej z wykorzystaniem domyślnej nazwy *Startup* i zarejestrowanie jej w hoście poprzez metodę *UseStartup<T>* jest chyba idealnym sposobem skonfigurowania uruchamiania aplikacji ASP.NET Core. Następnie w ciele tej klasy tworzymy metody specyficzne dla środowiska, takie jak *ConfigureDevelopment* i *ConfigureProduction*.

Host zajmie się ustaleniem odpowiedniej nazwy metody w oparciu o aktualnie ustawione środowisko. Warto zwrócić uwagę, że jeśli zmienimy nazwę klasy startowej na coś innego niż *Startup,* to ta wbudowana logika automatycznego ustalania nazw typów zawiedzie.

Potok przetwarzania ASP.NET

Interfejs *IApplicationBuilder* zapewnia sposoby na zdefiniowanie struktury potoku przetwarzania ASP.NET. Potok przetwarzania jest łańcuchem opcjonalnych modułów, które przetwarzają nadchodzące żądania HTTP, jak pokazano na rysunku 2-6.



Rysunek 2-6 Potok przetwarzania ASP.NET Core

Potok przetwarzania składa się ze składników oprogramowania pośredniego zarejestrowanych w metodzie *Configure* i wywoływanych w kolejności zarejestrowania dla każdego żądania. Każdy składnik oprogramowania pośredniego jest zbudowany na bazie następującego wzorca:

```
app.Use(async (httpContext, next) =>
{
    // Wstępne przetwarzanie żądania
    ...

    // Przekazanie przetwarzania do następnego modułu oprogramowania
    // pośredniego w łańcuchu
    await next();

    // Końcowe przetwarzanie żądania
    ...
});
```

Wszystkie składniki oprogramowania pośredniego mają szansę na przetworzenie żądania, zanim będzie ono faktycznie obsługiwane przez kod ASP.NET. Wywołując następny moduł, każdy składnik oprogramowania pośredniego przepycha żądanie dalej do następnego modułu w łańcuchu. Gdy ostatni zarejestrowany moduł zakończy wstępne przetwarzanie żądania, żądanie jest wykonywane. Następnie żądanie wraca przez łańcuch składników oprogramowania pośredniego w odwrotnej kolejności i wszystkie zarejestrowane moduły mają szansę na końcowe przetworzenie żądania, zwykle sprawdzając przy tym zaktualizowany kontekst i odpowiedź. W drodze z powrotem do klienta moduły oprogramowania pośredniego są wywoływane w odwrotnej kolejności.

Możemy zarejestrować swoje własne oprogramowanie pośrednie przy pomocy takiego fragmentu kodu, jak pokazano powyżej i wskazać ten kod poprzez wyrażenie lambda. Alternatywnie możemy opakować logikę w osobną klasę i utworzyć metodę *UseXxx* do rejestrowania jej w potoku przetwarzania wewnątrz metody *Configure*. Wrócimy do potoku przetwarzania ASP.NET i jego dostosowania w rozdziale 14.

Łańcuch składników oprogramowania pośredniego kończy się modułem uruchamiającym żądanie, czyli kodem, który faktycznie wykona działanie zamierzone przez to żądanie. Kod ten jest też nazywany *kończącym oprogramowaniem pośrednim*. W klasycznym ASP.NET MVC moduł wykonywania żądania wywołuje akcję, która wybiera odpowiednią klasę kontrolera, ustala właściwą metodę i ją wywołuje. Jak jednak wspominaliśmy, w ASP.NET Core model programowania MVC jest tylko jedną z opcji. Oznacza to, że moduł wykonujący żądanie przyjmuje bardziej abstrakcyjną postać:

```
app.Run(async context =>
{
    await
      context.Response.WriteAsync("Courtesy of 'Programming ASP.NET Core'");
});
```

Kod wykonywany przez kończące oprogramowanie pośrednie ma formę następującego delegata:

```
public delegate Task RequestDelegate(HttpContext context);
```

Kończące oprogramowanie pośrednie przyjmuje wystąpienie obiektu *HttpContext* i zwraca obiekt zadania *Task*. Obiekt kontekstu HTTP jest pojemnikiem na informacje związane z HTTP, w tym strumień odpowiedzi, poświadczenia uwierzytelniania, parametry wejściowe, stan sesji i informacje o połączeniu.

Jeśli kończące oprogramowanie pośrednie jest zdefiniowane jawnie poprzez metodę *Run*, to każde żądanie jest obsługiwane bezpośrednio z tego miejsca bez potrzeby obecności kontrolerów i widoków. Przy zaimplementowanej metodzie oprogramowania pośredniego *Run* każde żądanie będzie serwowane niemal bez dodatkowych kosztów w najszybszy możliwy sposób i przy całkowicie minimalnym zużyciu pamięci. Zademonstruję tę funkcję w następnej części tego rozdziału.

Podsystem wstrzykiwania zależności

Przegląd środowiska uruchomieniowego ASP.NET nie byłby kompletny bez przyjrzenia się wbudowanemu podsystemowi wstrzykiwania zależności.

Rzut oka na wstrzykiwanie zależności

Wstrzykiwanie zależności jest zasadą projektową, która promuje luźne powiązania pomiędzy klasami. Powiedzmy na przykład, że mamy następującą klasę:

```
public class FlagService
{
   private FlagRepository _repository;

   public FlagService()
   {
      _repository = new FlagRepository();
   }

   public Flag GetFlagForCountry(string country)
   {
      return _repository.GetFlag(country);
   }
}
```

Klasa *FlagService* jest zależna od klasy *FlagRepository* i biorąc pod uwagę zadania obu klas, ścisła relacja jest nie do uniknięcia. Zasada wstrzykiwania zależności pomaga utrzymywać luźne powiązania pomiędzy klasą *FlagService* i jej zależnościami. Podstawą wstrzykiwania zależności jest uzależnienie klasy *FlagService* jedynie od abstrakcji funkcji zapewnianych przez klasę *FlagRepository*. Mając wstrzykiwanie zależności na uwadze, możemy przepisać tę klasę w następujący sposób:

```
public class FlagService
{
   private IFlagRepository _repository;

   public FlagService(IFlagRepository repository)
   {
      _repository = repository;
   }

   public Flag GetFlagForCountry(string country)
   {
      return _repository.GetFlag(country);
   }
}
```

Teraz dowolna klasa, która implementuje *IFlagRepository*, może bezpiecznie działać z wystąpieniem klasy *FlagService*. Dzięki wykorzystaniu wstrzykiwania zależności zamieniliśmy ścisłą zależność pomiędzy klasami *FlagService* i *FlagRepository* na luźniejszą relację pomiędzy klasą *FlagService* i abstrakcją usług, które są jej potrzebne z zewnątrz. Odpowiedzialność za utworzenie wystąpienia abstrakcji repozytorium została przeniesiona poza klasę usługi. Oznacza to, że jakiś inny kod jest teraz odpowiedzialny za wzięcie odwołania do interfejsu (abstrakcji) i zwrócenie użytecznego wystąpienia konkretnego typu (klasy). Kod ten może być ręcznie napisany za każdym razem, gdy jest potrzebny.

```
var repository = new FlagRepository();
var service = new FlagService(repository);
```

Ten kod może też być uruchamiany przez warstwę „ad hoc" kodu, która bada konstruktora usługi i ustala wszystkie jej zależności.

```
var service = DependencyInjectionSubsystem.Resolve(FlagService);
```

Refaktoring typów dzięki stosowaniu tego wzorca wstrzykiwania pomoże też nam łatwiej pisać testy jednostkowe, ponieważ imitowana implementacja może być w dowolnym momencie przekazywana do konstruktorów.

ASP.NET Core zawiera swój własny podsystem wstrzykiwania zależności tak, że każda klasa może po prostu deklarować w konstruktorze (lub innych elementach) wszystkie konieczne zależności; system ten będzie zapewniać, że prawidłowe wystąpienia tych zależności będą tworzone i przekazywane.

Wstrzykiwanie zależności w ASP.NET Core

Aby skorzystać z systemu wstrzykiwania zależności, musimy rejestrować typy, dla których ten system ma być w stanie tworzyć za nas wystąpienia. System wstrzykiwania zależności ASP.NET Core domyślnie zna już niektóre typy, takie jak *IHostingEnvironment* oraz *ILoggerFactory,* ale musi poznać typy specyficzne dla danej aplikacji. Zobaczmy, co wiąże się z dodawaniem nowych typów do systemu wstrzykiwania zależności.

Rejestrowanie typów w systemie wstrzykiwania zależności

Parametr *IServicesCollection*, który nasz kod otrzymuje w metodzie *ConfigureServices*, jest dojściem pozwalającym na dostęp do wszystkich typów aktualnie zarejestrowanych w systemie wstrzykiwania zależności. Aby zarejestrować nowy typ, dodajemy kod do metody *ConfigureServices*.

```
public void ConfigureServices(IServiceCollection services)
{
    // Zarejestruj niestandardowy typ w systemie wstrzykiwania
    // zależności
    services.AddTransient<IFlagRepository, FlagRepository>();
}
```

Metoda *AddTransient* instruuje system wstrzykiwania zależności, aby serwował świeże, nowe wystąpienie typu *FlagRepository* za każdym razem, gdy żądana będzie abstrakcja, taka jak interfejs *IFlagRepository*. Po dodaniu tego wiersza kodu każda klasa, dla której tworzenie wystąpień jest zarządzane przez ASP.NET Core, może po prostu deklarować parametr typu *IFlagRepository*, aby otrzymywać dla niego od systemu świeże wystąpienie danej klasy. Oto typowy przykład użycia systemu wstrzykiwania zależności:

```
public class FlagController
{
    private IFlagRepository _flagRepository;
    public FlagController(IFlagRepository flagRepository)
    {
        _flagRepository = flagRepository;
    }

    ...
}
```

Klasy kontrolerów i widoków są bardzo typowymi przykładami klas ASP.NET Core, które wykorzystują system wstrzykiwania zależności.

Ustalanie typów w oparciu o warunki w trakcie działania programu

Czasami chcemy zarejestrować typ abstrakcyjny w systemie wstrzykiwania zależności, ale musimy zdecydować się na konkretny typ dopiero po sprawdzeniu pewnych warunków działania programu (na przykład dołączone pliki cookie, nagłówki HTTP, parametry łańcucha zapytania). Oto, jak to zrobić:

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddTransient<IFlagRepository>(provider =>
   {
      // Utwórz wystąpienie faktycznego typu do zwrócenia
      // w oparciu o tożsamość aktualnie zalogowanego użytkownika.
      var context = provider.GetRequiredService<IHttpContextAccessor>();
      return new FlagRepositoryForUser(context.HttpContext.User);
   });
}
```

Można zauważyć, że możemy uzyskać wstrzyknięty kontekst HTTP w kontekście programu, gdzie nie jest on normalnie dostępny, prosząc pojemnik wstrzykiwania zależności o wstrzyknięcie wystąpienia *IHttpContextAccessor*.

Ustalanie typów na żądanie

W niektórych przypadkach musimy tworzyć wystąpienia typów, które mają swoje własne zależności. Bardzo dobrym przykładem jest klasa *FlagService*, którą przedstawiliśmy wcześniej, demonstrując pojęcie wstrzykiwania zależności (zobacz „Rzut oka na wstrzykiwanie zależności" wcześniej w tym rozdziale).

```
public class FlagService
{
   public FlagService(IFlagRepository repository)
   {
      _repository = repository;
   }
   ...
}
```

Jak mielibyśmy utworzyć wystąpienie klasy bez wcześniejszego, ręcznego ustalenia wszystkich jej zależności? Trzeba zwrócić uwagę, że zależności mogą być zagnieżdżone na wielu poziomach, więc może być tak, że do utworzenia wystąpienia typu implementującego *IFlagRepository* trzeba najpierw utworzyć wystąpienia wielu innych typów. Każdy system wstrzykiwania zależności może nam pomóc w rozwiązaniu tego problemu, a system istniejący w ASP.NET Core nie jest tutaj wyjątkiem.

Zwykle system wstrzykiwania zależności skupia się wokół obiektu głównego znanego jako pojemnik, który przechodzi przez drzewo zależności i ustala konkretne typy dla typów abstrakcyjnych. W systemie ASP.NET Core pojemnik ten jest reprezentowany przez interfejs *IServiceProvider*. Aby ustalić wystąpienie *FlagService*, mamy dwie opcje: zastosować klasyczny operator new i zapewnić prawidłowe wystąpienie implementacji interfejsu *IFlagRepository* albo wykorzystać *IServiceProvider*, jak pokazano poniżej:

```
var flagService = provider.GetService<FlagService>();
```

Aby uzyskać wystąpienie pojemnika IServiceProvider, po prostu definiujemy *IServiceProvider* jako parametr konstruktora, gdzie nam będzie potrzebny, a system wstrzykiwania zależności wstrzyknie oczekiwane wystąpienie. Oto przykład kontrolera:

```
public class FlagController
{
```