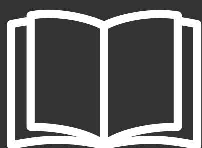
Ricardo Peres

Tajniki ASP.NET Core 2.0

Wzorzec MVC, konfiguracja, routing, wdrażanie i jeszcze więcej



Packt

Tajniki ASP.NET Core 2.0

Wzorzec MVC, konfiguracja, routing, wdrażanie i jeszcze więcej

Ricardo Peres

Przekład: Jakub Niedźwiedź

APN Promise
Warszawa 2018

Tajniki ASP.NET Core 2.0

Original English language edition © 2017 Packt Publishing

All rights reserved. Authorised translation from the English language edition book
Mastering ASP.NET Core 2.0
ISBN 978-1-78728-368-8, published by Packt Publishing.

© Polish edition by APN PROMISE SA, Warszawa 2018

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

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.

Książka ta przedstawia poglądy i opinie autorów. 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.

Wszystkie nazwy handlowe i towarowe występujące w niniejszej publikacji mogą być znakami towarowymi zastrzeżonymi lub nazwami zastrzeżonymi odpowiednich firm 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: 978-83-7541-353-3

Przekład: Jakub Niedźwiedź

Redakcja: Marek Włodarz
Korekta: Ewa Swędrowska
Skład i łamanie: MAWART Marek Włodarz

Zespół

Autor
Ricardo Peres

Recenzent
Alvin Ashcraft

Opracowanie indeksu
Tejal Daruwale Soni

Grafika
Abhinash Sahu

O autorze

Ricardo Peres jest portugalskim programistą, blogerem i od czasu do czasu autorem książek. Ma ponad 17-letnie doświadczenie w tworzeniu oprogramowania przy użyciu technologii, takich jak C/C++, Java, JavaScript i .NET. Interesuje się systemami rozproszonymi, architekturą oprogramowania, wzorcami projektowymi i ogólnym programowaniem w .NET.

Obecnie pracuje dla londyńskiej firmy Simplifydigital jako ewangelista techniczny i szczyci się tytułem MVP od 2015 roku.

Ricardo prowadzi bloga „*Developmenwt With A Dot*", gdzie regularnie pisze o sprawach technicznych. Można go poczytać pod adresem: *http://weblogs.asp.net/ricardoperes*.

Napisał książkę *Entity Framework Core Cookbook – Second Edition* i był recenzentem technicznym książki *Learning NHibernate 4* wydanej przez Packt.

Ricardo przyczynił się też do rozwoju kolekcji książek elektronicznych z serii *Succinctly* wydawanych przez Syncfusion, pracując nad tytułami dotyczącymi NHibernate, Entity Framework Code First, Entity Framework Core, aplikacji ASP.NET dla wielu użytkowników oraz Microsoft Unity.

Można skontaktować się z nim na Twitterze pod adresem *http://twitter.com/rjperes75*.

Chciałbym podziękować swojemu synowi Francisco i córce Madalenie za ich miłość; są oni powodem, dla którego napisałem tę książkę.

Nie mógłbym tego dokonać bez swoich przyjaciół GC i PJ; dziękuję Wam za przyjaźń i wsparcie!

Dla uczczenia pamięci o moich rodzicach: Irene i Jorge Peres.

O recenzencie

Alvin Ashcraft jest twórcą oprogramowania, mieszkającym w pobliżu Filadelfii w stanie Pensylwania. Poświęcił swoją 22-letnią karierę na budowanie oprogramowania przy pomocy C#, Visual Studio, WPF, ASP.NET, HTML/JavaScript, UWP, Xamarin i SQL Server. Ośmiokrotnie otrzymał nagrodę Microsoft MVP, raz za architekturę oprogramowania w 2009 roku oraz przez siedem następnych lat za C# oraz Visual Studio i narzędzia. Przygotowywaną przez niego codzienną porcję hiperłączy dla programistów .NET można znaleźć na jego blogu pod adresem alvinashcraft.com, a blog ze wskazówkami dla twórców aplikacji UWP można znaleźć pod adresem www.uwpapp.tips.

Obecnie pracuje jako główny inżynier oprogramowania w firmie Allscripts, opracowując oprogramowanie dla służby zdrowia. Wcześniej był zatrudniony w kilku dużych firmach programistycznych, w tym Oracle, Genzeon i Corporation Service Company.

Pomagał tam tworzyć rozwiązania programistyczne dla sektora finansowego, biznesu i służby zdrowia, korzystając z platform i rozwiązań Microsoft.

Był recenzentem technicznym książki *NuGet 2 Essentials* wydanej przez Packt.

Chciałbym podziękować za wsparcie swojej cudownej żonie Stelene i naszym trzem wspaniałym córkom. Były bardzo wyrozumiałe, gdy czytałem i recenzowałem te rozdziały wieczorami i podczas weekendów, aby pomóc w powstaniu przydatnej i doskonałej jakościowo książki dla programistów ASP.NET Core.

Spis treści

O autorze iii

O recenzencie iv

Spis treści v

Wstęp 1

Rozdział 1: Początki pracy z ASP.NET Core 7

- **Wprowadzenie do ASP.NET Core** 8
- **.NET Core** 11
- **Platformy systemowe** 16
- **Zależności i platformy** 17
  - Programowanie dla .NET Core lub pełnej platformy .NET Framework 20
- **Zrozumienie wzorca MVC** 21
- **Uzyskiwanie kontekstu** 25
- **Potok OWIN** 26
- **Hosting** 30
  - Kestrel 31
  - WebListener 32
  - IIS 33
  - Nginx 33
  - Apache 33
  - Konfiguracja 33
  - Funkcje 34
  - Konfiguracja uruchomieniowa 35
- **Odwrócenie sterowania i wstrzykiwanie zależności** 36
- **Środowiska uruchamiania** 41
- **Projekt przykładowy** 42
  - Struktura 42
- **Podsumowanie** 44

Rozdział 2: Konfiguracja 45

- **Konfiguracja w .NET Core** 45
- **Dostawcy** 48

Dostawcy oparci na plikach . . . 50
Dostawca JSON . . . 51
Dostawca XML . . . 52
Dostawca INI . . . 52
Inni dostawcy . . . 53
Sekrety użytkowników . . . 53
Azure Key Vault . . . 54
Wiersz poleceń . . . 54
Zmienne środowiskowe . . . 56
Pamięć . . . 56
Docker . . . 56
Tworzenie niestandardowego dostawcy . . . 57
Korzystanie z wartości konfiguracyjnych . . . 59
Jawne pobieranie i ustawianie wartości . . . 60
Sekcje konfiguracji . . . 60
Pobieranie wszystkich wartości . . . 62
Wiązanie z klasami . . . 62
Wstrzykiwanie wartości . . . 64
Przeładowywanie konfiguracji i obsługiwanie powiadomień o zmianach . . . 65
Łączenie wszystkiego razem . . . 66
Podsumowanie . . . 67

Rozdział 3: Routing . . . 69

Tabele routingu . . . 70
Szablony tras . . . 71
Dopasowywanie parametrów routingu . . . 73
Wybieranie trasy . . . 75
Trasy specjalne . . . 77
Domyślne trasy . . . 77
Procedury obsługi tras . . . 78
Routing do wewnętrznych procedur obsługi . . . 79
Ograniczenia tras . . . 80
Metody HTTP . . . 81
Ograniczenia domyślne . . . 82
Ograniczenia niestandardowe . . . 83
Tokeny danych tras . . . 85
Obszary . . . 86
Atrybuty routingu . . . 87

Trasy 87
Ustawienia domyślne 89
Ograniczenia 89
Obszary 90
Nazwy akcji..90
Brak akcji 91
Obostrzenia 91
Wartości tras..91
Radzenie sobie z błędami 92
Trasa przechwytująca wszystkie przypadki 92
Oprogramowanie pośrednie dla stron kodów stanu 92
Określone kody stanów 93
Dowolny kod stanu 93
Łączenie wszystkiego razem 94
Podsumowanie 94

Rozdział 4: Kontrolery i akcje..95

Korzystanie z kontrolerów 95
Klasy bazowe dla kontrolerów 96
Kontrolery POCO..98
Dodawanie kontekstu do kontrolerów POCO 98
Przechwytywanie akcji w kontrolerach POCO..100
Znajdowanie kontrolerów..100
Cykl życia kontrolera..102
Akcje 104
Znajdowanie akcji..104
Akcje synchroniczne i asynchroniczne 104
Uzyskiwanie kontekstu 105
Ograniczenia akcji..106
Parametry akcji 107
Wiązanie modelu 110
Treść żądania..111
Formularz..111
Nagłówek..112
Zapytanie 112
Trasa..112
Wstrzykiwanie zależności 112
Niestandardowe elementy wiążące 113

Elementy formatujące dane wejściowe 115
Jawne wiązanie 115
Anulowanie żądań....117
Sprawdzanie poprawności modelu....117
Wyniki akcji....119
Akcje API 124
Elementy formatujące dane wyjściowe....125
Negocjowanie zawartości 126
Wersjonowanie API 128
Wartości nagłówków 129
URL....129
Odwoływanie 130
Wersja domyślna 130
Mapowanie wersji 131
Nieprawidłowe wersje 131
Dokumentacja API 131
Pamięć podręczna dla odpowiedzi 136
Utrzymywanie stanu....139
Żądanie 139
Formularz 139
Łańcuch zapytania 139
Trasa 140
Pliki cookie 140
Sesja 142
Pamięć podręczna 144
Pamięć podręczna wewnątrz pamięci operacyjnej 144
Rozproszona pamięć podręczna 146
Redis 147
SQL Server 147
Dane tymczasowe 148
Porównanie technik utrzymywania stanu....149
Wstrzykiwanie zależności 149
Globalizacja i lokalizacja 150
Łączenie wszystkiego razem 156
Podsumowanie 156

Rozdział 5: Widoki 157

Klasy widoków....159

Lokalizowanie widoków 161
Ekspandery lokalizacji widoków 163
Silniki widoków 164
Rejestrowanie i diagnostyka 170
Kompilacja widoków 172
Model 177
ViewBag 178
Dane tymczasowe 179
Układy widoków 179
Widoki częściowe 181
Przekazywanie danych do widoków częściowych 182
Znajdowanie widoków częściowych 182
Pliki specjalne 183
Opcje widoków 183
Obszary 186
Wstrzykiwanie zależności 186
Zasoby 187
Widoki tłumaczone 189
Strony Razor 190
Model strony 191
Procedury obsługi strony 193
Wiązanie modelu 196
Sprawdzanie poprawności modelu 197
Utrzymywanie stanu 197
Układy widoków 197
Widoki częściowe 197
Pliki specjalne 198
Filtry 198
Wstrzykiwanie zależności 198
Konfigurowanie opcji 198
Trasy stron 199
Bezpieczeństwo 200
Łączenie wszystkiego razem 201
Podsumowanie 201

Rozdział 6: Korzystanie z formularzy i modeli 203

Kontekst formularza 204
Metadane modelu 204

Modele typów anonimowych . . . 208
Pomocnicy HTML . . . 209
Formularze . . . 209
Jednowierszowe pola tekstowe . . . 210
Wielowierszowe pola tekstowe . . . 211
Hasła . . . 211
Listy rozwijane . . . 211
Pola list . . . 212
Przyciski radiowe . . . 212
Pola wyboru . . . 213
Wartości ukryte . . . 213
Łącza . . . 213
Etykiety . . . 214
Surowy kod HTML . . . 214
Identyfikatory, nazwy i wartości . . . 214
Ogólne szablony do edycji i wyświetlania . . . 214
Metody i właściwości użytkowe . . . 215
Komunikaty sprawdzania poprawności . . . 215
Niestandardowi pomocnicy . . . 216
Szablony . . . 218
Wiązanie modelu . . . 220
Sprawdzanie poprawności modelu . . . 222
Sprawdzanie poprawności danych po stronie serwera . . . 222
Konfiguracja . . . 223
Sprawdzanie poprawności przez adnotacje danych . . . 224
Komunikaty o błędach . . . 227
Samoczynne sprawdzanie poprawności . . . 227
Niestandardowe sprawdzanie poprawności . . . 228
Sprawdzanie poprawności modelu po stronie klienta . . . 230
Konfiguracja . . . 230
Niestandardowe sprawdzanie poprawności . . . 231
AJAX 234
Sprawdzanie poprawności . . . 235
Obostrzenia . . . 236
Zawartość . . . 237
Ładowanie plików . . . 237
Dostęp bezpośredni . . . 238
Łączenie wszystkiego razem . . . 239

Podsumowanie 239

Rozdział 7: Bezpieczeństwo 241

Uwierzytelnianie 241
Uwierzytelnianie systemu Windows 242
Uwierzytelnianie niestandardowe 244
Uwierzytelnianie w ASP.NET Core 1.x. 244
Uwierzytelnianie w ASP.NET Core 2.x. 247
Identity 248
Korzystanie z JWT 252
IdentityServer 255
ASP.NET Core 1.x. 255
ASP.NET Core 2.x. 256
Azure AD 257
Logowanie przez serwisy społecznościowe 259
Twitter 259
Google. 260
Microsoft 260
Bezpieczeństwo wiązania 261
Bezpieczeństwo plików cookie 261
Autoryzacja 262
Role 262
Zasady 263
Procedury obsługi autoryzacji 264
Autoryzacja oparta na zasobach 267
Dostęp anonimowy 268
Przeciwdziałanie fałszerstwom 268
Kodowanie HTML 271
HTTPS 271
Hosting 272
IIS Express 272
Kestrel 273
ASP.NET Core 1.x. 273
ASP.NET Core 2.x. 273
WebListener. 274
Ograniczanie HTTP 274
Przekierowanie do HTTP 274
CORS 275

Ochrona danych 277
Pliki statyczne 278
Wykorzystanie akcji do pobierania plików 278
Wykorzystanie oprogramowania pośredniego do wymuszania bezpieczeństwa 279
Łączenie wszystkiego razem 281
Podsumowanie 281

Rozdział 8: Komponenty wielokrotnego użytku 283

Komponenty widoków 283
Odkrywanie komponentów widoków 284
Korzystanie z komponentów widoków 284
Wyniki komponentów widoków 285
Wstrzykiwanie zależności 286
Pomocnicy znacznikowi 287
Zrozumienie właściwości pomocnika znacznikowego 290
Ograniczanie stosowalności 291
Odkrywanie pomocników znacznikowych 292
Wstrzykiwanie zależności 293
Dołączeni pomocnicy znacznikowi 293
Znacznik <a> 294
Znacznik <cache> 295
Znacznik <distributed-cache> 296
Znacznik <environment> 296
Znacznik <form> 296
Znacznik <script> 297
Znacznik <link> 297
Znacznik <select> 298
Komunikat i podsumowanie sprawdzania poprawności 298
Komponenty pomocników znacznikowych 298
Widoki częściowe 300
Widoki częściowe a komponenty widoków 301
Łączenie wszystkiego razem 301
Podsumowanie 302

Rozdział 9: Filtry 303

Typy filtrów 303
Wersje synchroniczne i asynchroniczne 304
Zakresy filtrów 304

Kolejność wykonywania 305
Stosowanie filtrów poprzez atrybuty 306
Fabryki i dostawcy 307
Wstrzykiwanie zależności 308
Kontekst 309
Filtry autoryzacji 310
Zasady autoryzacji 312
Filtry zasobów 312
Zrozumienie filtrów akcji 314
Filtry wyników 315
Filtry wyjątków 317
Filtry stron 318
Łączenie wszystkiego razem 319
Podsumowanie 320

Rozdział 10: Rejestrowanie, śledzenie i diagnostyka 321

Wspólna platforma rejestrowania 321
Korzystanie z rejestrowania 322
Dostawcy rejestrowania 325
ASP.NET Core 1.x 326
ASP.NET Core 2.x 326
Filtrowanie rejestrowania 327
ASP.NET Core 1 327
ASP.NET Core 2 329
Niestandardowi dostawcy 330
Wstrzykiwanie zależności 332
Atrybuty rejestrowania 333
Niestandardowe oprogramowanie pośrednie do rejestrowania 335
Diagnostyka 336
Telemetria 339
ASP.NET Core 1.x 341
ASP.NET Core 2.x 342
Zdarzenia niestandardowe 342
Sprawdzanie kondycji 345
Łączenie wszystkiego razem 346
Podsumowanie 347

Rozdział 11: Testowanie 349
Testy jednostkowe 349
xUnit 351
Wstrzykiwanie zależności 355
Imitowanie obiektów 356
Twierdzenia 358
Interfejs użytkownika 359
Korzystanie z wiersza poleceń 361
Ograniczenia testów jednostkowych 362
Testowanie integracyjne 362
Łączenie wszystkiego razem 364
Podsumowanie 364

Rozdział 12: Programowanie po stronie klienta 365
Korzystanie z Bower 366
Korzystanie z Node.js 368
Wywoływanie Node z .NET Core 369
Wstępne kompilowanie widoków 370
Korzystanie z WebPack 371
Szablony SPA 372
Korzystanie z Gulp 373
Korzystanie z Grunt 373
Korzystanie z TypeScript 375
Korzystanie z Less 376
Łączenie wszystkiego razem 378
Podsumowanie 378

Rozdział 13: Poprawianie wydajności i skalowalności 379
Profilowanie 379
CoreProfiler 380
MiniProfiler 380
Stackify Prefix 381
Hosting 382
Wybieranie najlepszego hosta 382
Dostrajanie 383
Maksymalna liczba jednoczesnych połączeń 383
Ograniczenia 384

Limity czasu 385

Opakowywanie i minimalizowanie 386

Akcje asynchroniczne 387

Buforowanie w pamięci podręcznej 389

Buforowanie danych w pamięci podręcznej 389

Pamięć podręczna wewnątrz pamięci operacyjnej 389

Rozproszona pamięć podręczna 393

Buforowanie wyników akcji w pamięci podręcznej 395

Buforowanie widoków w pamięci podręcznej 398

Kompresowanie odpowiedzi 398

Łączenie wszystkiego razem 400

Podsumowanie 400

Rozdział 14: Komunikacja w czasie rzeczywistym 401

Konfigurowanie SignalR 401

Hosting koncentratora 402

Protokoły komunikacyjne 405

Kontekst SignalR 406

Cele komunikatów 406

Komunikowanie się z zewnątrz 407

Podsumowanie 408

Rozdział 15: Inne zagadnienia 409

Obszary 409

Użycie 410

Pomocnicy znacznikowi i HTML 411

Praca ze statycznymi plikami i folderami 411

Przeglądanie katalogów 411

Pliki statyczne 412

Dokumenty domyślne 414

Bezpieczeństwo 415

Zdarzenia czasu życia aplikacji 415

Osadzane zasoby 417

Rozszerzenia hostingu 418

Uruchamianie hostingu 418

Usługi hostowane 419

Konwencje niestandardowe 420

Przepisywanie adresów URL 424

Przekierowywanie URL 426
Przepisywanie adresów URL 426
Stosowanie reguł w czasie działania programu 427
HTTPS 429
Specyficzne dla platformy 429
Łączenie wszystkiego razem 429
Podsumowanie 430

Rozdział 16: Wdrażanie 431

Ręczne wdrażanie 431
Ustawianie platformy docelowej 433
Samodzielne wdrożenia i środowiska uruchomieniowe 434
Przebudowywanie w czasie rzeczywistym 434
Wdrażanie przy użyciu Visual Studio 435
IIS 437
Nginx440
Azure441
AWS 442
Docker 443
Usługa systemu Windows 445
Łączenie wszystkiego razem 446
Podsumowanie 446

Indeks 447

Wstęp

ASP.NET Core jest względnie nową technologią programistyczną, ale bazuje na swoim starszym kuzynie ASP.NET. Technologia ASP.NET istniała od tak wielu lat i była tak popularna, że można by się zastanawiać, po co ją zmieniać?

Czasem zmiany są jednak potrzebne; świat przedsiębiorstw się zmienił i pojawienie się chmur obliczeniowych dramatycznie zmieniło sytuację. Trudno jest być obecnie jedynie programistą dla systemu Windows lub programistą dla systemu Linux, ale często trzeba być jednym i drugim, żeby dostosować się do wymagań biznesu. Dlatego również świat .NET wymagał zmian.

Technologia ASP.NET Core została zbudowana od podstaw; nie jest to ulepszenie szacownej platformy ASP.NET, ale coś nowego. Nie ma już technologii Web Forms, gdyż trendy technologiczne wydają się zmierzać w kierunku paradygmatu Model-View-Controller (i to nie tylko w .NET). Środowiska programistów stosują nowe techniki takie jak programowanie sterowane testami (TDD – Test Driven Development), które idealnie pasują do MVC.

Ponieważ projekt .NET Core jest całkowicie oparty na otwartym kodzie, mógł też skorzystać ze wsparcia środowiska, które wpływało na wprowadzenie nowych funkcji, a czołowi programiści, pracujący nad tym projektem, otwarcie omawiają kolejne plany w Internecie. Co za ekscytujące czasy!

Ta książka stanowi przewodnik po funkcjach ASP.NET Core, z których niektóre mogą być już znane z wersji ASP.NET (sprzed wersji Core), ale wiele jest całkiem nowych. Nawet wcześniej istniejące funkcje są w pewnym sensie nowe, ponieważ są zbudowane na całkiem innej infrastrukturze, która jest modularna, lekka i rozszerzalna.

Tutaj będzie można znaleźć materiały na temat ASP.NET Core, ale również samej platformy .NET Core, na której osadzona jest technologia ASP.NET. Zarówno wytrawni programiści ASP.NET, jak i nowicjusze powinni być w stanie znaleźć coś dla siebie.

O czym jest ta książka

Rozdział 1: *Początki pracy z ASP.NET Core*, wprowadza podstawy .NET Core i ASP.NET Core oraz wyjaśnia kilka podstawowych pojęć.

Rozdział 2: *Konfiguracja*, omawia nowy system konfiguracji w .NET Core.

Rozdział 3: *Routing*, opisuje wykorzystanie routingu do dopasowywania nadchodzących żądań do konftrolerów i akcji.
Rozdział 4: *Kontrolery i akcje*, wyjaśnia wszystko na temat klas (kontrolerów) i metod (akcji), które są używane do przetwarzania żądań.
Rozdział 5: *Widoki*, zajmuje się widokami HTML i nową funkcjonalnością stron Razor w ASP.NET Core 2.
Rozdział 6: *Korzystanie z formularzy i modeli*, pokazuje, jak dane wejściowe z HTML są zamieniane na modele i jak generować kod HTML dla właściwości modelu.
Rozdział 7: *Bezpieczeństwo*, omawia kilka aspektów związanych z zabezpieczaniem ASP.NET Core, w tym autoryzację i uwierzytelnianie.
Rozdział 8: *Komponenty wielokrotnego użytku*, stanowi przewodnik po różnych sposobach pisania kodu wielokrotnego użytku dla różnych celów.
Rozdział 9: *Filtry*, przedstawia możliwości przechwytujące ASP.NET Core.
Rozdział 10: *Rejestrowanie, śledzenie i diagnostyka*, demonstruje, jak możemy przechwytywać informacje o swoich aplikacjach.
Rozdział 11: *Testowanie*, prowadzi nas przez pisanie testów, sprawdzających funkcjonalność naszych aplikacji i modułów.
Rozdział 12: *Programowanie po stronie klienta*, uczy, jak integrować Visual Studio i ASP.NET Core z popularnymi technologiami skryptowymi.
Rozdział 13: *Poprawianie wydajności i skalowalności*, przedstawia możliwości zwiększenia wydajności i skalowalności naszych aplikacji WWW.
Rozdział 14: *Komunikacja w czasie rzeczywistym*, pokazuje, jak korzystać z technologii SignalR do komunikacji pomiędzy serwerem a klientem.
Rozdział 15: *Inne zagadnienia*, jest rozdziałem, gdzie wyjaśniane są pewne funkcje ASP.NET Core, które nie znalazły miejsca w innych rozdziałach.
Rozdział 16: *Wdrażanie*, jest ostatnim rozdziałem. Poznamy w nim różne opcje wdrażania, dostępne dla aplikacji ASP.NET Core.

Co potrzeba do korzystania z tej książki

Do korzystania z kodu przykładowego z tej książki potrzebny będzie komputer działający pod kontrolą w miarę aktualnej wersji systemu Windows, macOS lub Linux. Potrzebny będzie też edytor tekstu do edycji kodu; polecam korzystanie z edytora pozwalającego otwierać wiele plików na raz i łatwo przełączać się między nimi. Na myśl od razu przychodzi mi Visual Studio 2015 lub 2017, albo Visual Studio Code. Ta ostatnia opcja jest lekkim edytorem, oferującym przy tym całkiem interesujące funkcje, opartym na otwartym kodzie źródłowym i dostępnym dla systemów Windows, macOS i Linux. Oprogramowanie Visual Studio powinno być dobrze znane wytrawnym

programistom .NET i oferuje rozbudowane funkcje takie jak bogate możliwości debugowania i refaktoringu kodu. Środowisko to jest dostępne w różnych wydaniach od Enterprise do Community, przy czym wersja Community jest darmowa, ale oferuje mniej funkcji niż płatne odpowiedniki. Jednakże przy korzystaniu z tej książki można spokojnie korzystać z dowolnej wersji beż żadnych problemów.

Dla kogo jest ta książka

Idealnym odbiorcą tej książki jest programista .NET, który pisał wcześniej aplikacje ASP.NET i chce przejść do tworzenia aplikacji dla .NET Core. Książka ta pozwoli poznać wszystkie funkcje z ostatnich wydań ASP.NET Core (1.1 i 2.0) w uporządkowany sposób, gdzie każdy rozdział omawia inne zagadnienie.

Ta książka zakłada, że czytelnik zna platformę .NET i język programowania C# oraz pisał już wcześniej jakiś kod. Zakłada też dobrą znajomość interfejsu wiersza poleceń w wybranym przez czytelnika systemie operacyjnym. Polecenia, które należy uruchamiać, będą podawane w tej książce.

Konwencje

W tej książce znajdziemy wiele stylów tekstu, którymi oznaczane są różne rodzaje informacji. Oto kilka przykładów tych stylów oraz wyjaśnienie ich znaczenia.

Elementy kodu w tekście, nazwy tabel bazodanowych, nazwy folderów, nazwy plików, rozszerzenia plików, nazwy ścieżek dostępu, adresy URL, dane wpisywane przez użytkowników oraz identyfikatory Twitter są przedstawiane następująco: „Najpierw wywołamy metodę `GetContacts()` dla wystąpienia klasy `ContactUtil`, aby uzyskać listę obiektów `Contact`.”

Blok kodu jest formatowany następująco:

```
if (this.TempData.ContainsKey("key"))
{
  var value = this.TempData["key"];
}
```

Wszelkie wejście lub wyjście wiersza polecenia jest zapisywane, jak w przykładzie poniżej. Polecenie wejściowe może być rozbite na kilka wierszy dla zapewnienia lepszej czytelności, ale musi być wpisane jako pojedyncze polecenie w wierszu poleceń:

```
dotnet restore
```

Nowe terminy i **ważne słowa** są pogrubione. Słowa, które są widoczne na ekranie, na przykład w menu lub oknach dialogowych, pojawiają się w tekście w następującej formie: „Dodajemy plik TypeScript do swojego projektu, klikając **Add New Item | Visual C# | ASP.NET Core | Web | Scripts | TypeScript File**."

Ostrzeżenia lub ważne uwagi oznaczone są takim symbolem.

Wskazówki i porady oznaczone są takim symbolem.

Informacje od czytelników

Informacje zwrotne od naszych czytelników są zawsze mile widziane. Prosimy o opinie na temat tej książki – co się w niej podoba, a co nie. Informacje od czytelników są dla nas ważne, żebyśmy mogli przygotowywać najbardziej pożądane tytuły.

W celu przekazania ogólnych uwag, wystarczy wysłać e-maila na adres `feedback@packtpub.com`, podając tytuł książki w temacie wiadomości.

Jeśli ktoś jest ekspertem w jakiejś dziedzinie i chciałby napisać lub uczestniczyć w pracach nad książką, może zajrzeć do naszego przewodnika dla autorów pod adresem `www.packtpub.com/authors`.

Obsługa klienta

Dla dumnego posiadacza książki wydawnictwa Packt mamy wiele rzeczy pomocnych w maksymalnym skorzystaniu ze swojego zakupu.

Pobieranie przykładowego kodu

Pliki przykładowego kodu dla tej książki można pobrać ze strony *http://www.ksiazki.promise.pl/aspx/produkt.aspx?pid=112130*. Po pobraniu plików należy je rozpakować za pomocą najnowszej wersji jednego z poniższych programów:

- WinRAR lub 7-Zip w systemie Windows
- Zipeg, iZip lub UnRarX w systemie Mac

- 7-Zip lub PeaZip w systemie Linux

Pakiet kodu dla tej książki jest też utrzymywany w serwisie GitHub pod adresem *https://github.com/PacktPublishing/Mastering-ASP.NET-Core-2.0*. Również inne pakiety kodu z naszego bogatego katalogu książek i filmów są dostępne pod adresem *https://github.com/PacktPublishing/*. Warto je sprawdzić!

Errata

Chociaż podjęliśmy wszelkie starania, aby zapewnić poprawność treści tej książki, błędy czasem się zdarzają. W razie znalezienia błędu w jednej z naszych książek – na przykład błędu w tekście lub w kodzie – bylibyśmy wdzięczni za zgłoszenie nam go. Dzięki temu możemy oszczędzić innym czytelnikom kłopotów i poprawić kolejne wersje tej książki. Wszelkie poprawki prosimy zgłaszać pod adresem *http://www.packtpub.com/submit-errata*, wybierając daną książkę, klikając łącze **Errata Submission Form** i wprowadzając szczegóły błędu. Po zatwierdzeniu zgłoszenia zostanie ono przyjęte i opublikowane na naszej stronie WWW lub dodane do listy istniejących poprawek w części Errata dla danego tytułu.

Wcześniej przesłane poprawki można znaleźć pod adresem *https://www.packtpub.com/books/content/support*, wpisując nazwę książki w polu wyszukiwania. Żądane informacje pojawią się w części **Errata**.

Piractwo

Internetowe piractwo materiałów chronionych prawem autorskim jest stałym problemem. W wydawnictwie Packt bardzo poważnie traktujemy ochronę naszych praw autorskich i licencji. W razie natrafienia w Internecie na nielegalne egzemplarze naszych prac w jakiejkolwiek formie prosimy o podanie nam adresu lub nazwy strony WWW, abyśmy mogli podjąć stosowne działania.

Prosimy o kontakt pod adresem `copyright@packtpub.com` z podaniem łącza do materiału podejrzanego o piractwo.

Dziękujemy za pomoc w ochronie naszych autorów i naszych możliwości dostarczania cennych treści.

Pytania

W przypadku problemów z dowolnym aspektem tej książki prosimy o kontakt pod adresem `questions@packtpub.com`, a postaramy się pomóc w miarę naszych możliwości.

1

Początki pracy z ASP.NET Core

Witam w mojej nowej książce na temat ASP.NET Core!

.NET Core i ASP.NET Core są względnie nowymi technologiami, jako że oficjalnie zostały wydane w sierpniu 2017 roku. Ponieważ mają w nazwie .NET, mogłoby się wydawać, że będą to jedynie nowe wersje niezwykle popularnej platformy .NET, ale tak nie jest – mówimy tu o czymś, co jest naprawdę nowe!

Nie oznacza to jedynie wsparcia dla wielu platform (witaj Linux!), ale znacznie więcej. Oznacza to nową modularność we wszystkich aspektach, przejrzysty sposób wprowadzania zmian, otwarcie dostępny kod źródłowy, zachęcający do udziału w jego dalszym rozwoju – to faktycznie znaczące różnice!

W tym pierwszym rozdziale porozmawiamy nieco o tym, co się zmieniło w wersjach Core technologii ASP.NET oraz .NET, a także o nowych, podstawowych pojęciach, takich jak OWIN, środowiska uruchomieniowe i wstrzykiwanie zależności. Następnie przedstawię krótki projekt przykładowy, który będzie nam towarzyszył przez całą książkę.

W tym rozdziale omówimy następujące zagadnienia:

- Historia ASP.NET Core
- Wprowadzenie do .NET Core
- Odwrócenie sterowania i wstrzykiwanie zależności
- OWIN
- Wzorzec MVC
- Hosting
- Środowiska
- Projekt przykładowy

Wprowadzenie do ASP.NET Core

Technologia Microsoft ASP.NET została wydana 15 lat temu, w roku 2002, jako część całkiem nowej wtedy platformy .NET Framework. Odziedziczyła nazwę **ASP** (**Active Server Pages**) po swojej poprzedniczce, z którą miała niewiele wspólnego poza tym, że też była technologią do opracowywania dynamicznej zawartości stron internetowych po stronie serwera i działała na platformach Windows.

Technologia ASP.NET zyskała ogromną popularność i konkurowała ramię w ramię z innymi popularnymi platformami WWW, takimi jak **Java Enterprise Edition** (**JEE**) i **PHP**. W istocie nadal jest popularna – serwis **BuiltWith** szacuje jej udział na 21% (łącznie ASP.NET i ASP.NET MVC), znacznie więcej niż Java (*https://trends.builtwith.com/framework*).

Technologia ASP.NET nie służyła jedynie do pisania dynamicznych stron WWW. Mogła też być używana do tworzenia usług WWW opartych na **XML** (**SOAP**), które były bardzo popularne na początku obecnego wieku. Czerpała korzyści z platformy .NET Framework oraz jej wielkiej biblioteki klas i składników wielokrotnego użytku, co sprawiało, że programowanie aplikacji dla przedsiębiorstw wydawało się niezwykle łatwe!

Pierwsza wersja ASP.NET 1 wprowadziła formularze **Web Forms**, które stanowiły próbę przeniesienia do WWW modelu programowania opartego na zdarzeniach i komponentach, znanego z aplikacji biurkowych, co odgradzało programistów od mniej przyjaznych aspektów HTML, HTTP i utrzymywania stanu aplikacji. W pewnym stopniu była to całkiem udana próba – korzystając z Visual Studio, można było łatwo w kilka minut utworzyć dynamiczną witrynę sterowaną danymi! Wiele można było osiągnąć przez same znaczniki, bez konieczności wprowadzania zmian w kodzie.

Wersja 2 pojawiła się kilka lat później, a wśród wielu ciekawych nowości przyniosła rozszerzalność w postaci modelu dostawców. Wiele aspektów funkcjonalności można było dostosowywać za pośrednictwem niestandardowych dostawców. Później pojawiły się rozszerzenia AJAX Extensions, które niezwykle ułatwiły programowanie w stylu AJAX. Technologia ta wyznaczyła standardy na kolejne lata, pozostawiając jedynie miejsce na dodatkowe składniki.

Kolejne wersje 3.5, 4 i 4.5 oferowały jedynie więcej tego samego w postaci nowych, wyspecjalizowanych kontrolek do wyświetlania danych i wykresów, a także kilka poprawek dotyczących zabezpieczeń. Dużą zmianę stanowił fakt, że niektóre z bibliotek platformy zostały wydane jako otwarty kod źródłowy.

Pomiędzy wersjami 3.5 i 4 firma Microsoft wypuściła całkiem nową platformę opartą na wzorcu **MVC** (**Model-View-Controller**) i w znacznym stopniu z otwartym kodem źródłowym. Choć bazowała ona na infrastrukturze przygotowanej przez

ASP.NET, to oferowała całkiem nowy paradygmat programowania, który tym razem w pełni obejmował możliwości HTTP i HTML. Wzorzec MVC stanowił ogólny trend w programowaniu WWW przy użyciu różnych technologii (PHP, Ruby i Java), a programiści .NET byli na ogół zadowoleni z jego wprowadzenia. Programiści ASP.NET mieli teraz do wyboru technologie Web Forms i MVC, które obie wykorzystywały potok przetwarzania ASP.NET oraz biblioteki .NET, ale oferowały dwa radykalnie odmienne podejścia do dostarczania zawartości do przeglądarki.

W międzyczasie platforma .NET Framework dojrzała we wciąż zmieniającym się świecie. W nowoczesnym przedsiębiorstwie potrzeby się zmieniają i stwierdzenia typu *„działa tylko w Windows"* albo *„trzeba czekać X lat na kolejną wersję"* stały się nie do przyjęcia. Dostrzegając to, firma Microsoft zaczęła pracować nad czymś nowym i całkiem innym na kolejne lata – i tak pojawiła się platforma .NET Core!

Pod koniec roku 2014 firma Microsoft ogłosiła .NET Core. Miało to być przepisanie platformy .NET Framework całkowicie od nowa z uwzględnieniem niezależności systemowej, możliwości korzystania z różnych języków oraz otwartego kodu źródłowego. Główne charakterystyki tej platformy były następujące:

- Biblioteki klas podstawowych .NET miały być przepisane od zera przy zachowaniu tych samych (uproszczonych) publicznych interfejsów API, co oznaczało, że początkowo nie wszystkie z nich byłyby dostępne.
- Możliwość uruchamiania na systemach operacyjnych innych niż Windows, w szczególności kilku odmianach systemów Linux i macOS oraz na urządzeniach mobilnych tak, że cały kod specyficzny dla Windows (oraz związane z nim interfejsy API) zostałby odrzucony.
- Wszystkie jej składniki miały być dostarczane jako pakiety NuGet, co oznacza, że na komputerze docelowym trzeba było instalować jedynie niewielki, binarny plik startowy.
- Nie było już zależności od IIS, można było ją uruchamiać bez procesu IIS.
- Miała mieć otwarty kod źródłowy, a programiści mieli mieć możliwość wpływania na jej kształt, zgłaszając swoje uwagi lub propozycje zmian.

Ostatecznie nastąpiło to w czerwcu 2016, gdy wydana została wersja 1.0 platformy .NET Core. Programiści .NET mogli teraz pisać kod raz i wdrażać go (niemal) wszędzie, a także mieć wpływ na dalsze kierunki rozwoju tej platformy!

Przepisanie całej platformy .NET Framework od nowa było epickim zadaniem, więc firma Microsoft musiała zdefiniować pewne priorytety i podjąć kilka decyzji. Jedną z nich było porzucenie ASP.NET Web Forms i skupienie się jedynie na MVC. Minęły więc dni, gdy ASP.NET i Web Forms były synonimami, to samo stało się z ASP.

NET Core i MVC; teraz mamy tylko ASP.NET Core! Dotychczas osobny interfejs ASP. NET Web API jest teraz również scalony z ASP.NET Core, co było mądrą decyzją ze strony Microsoft, ponieważ obie technologie, MVC i Web API, miały wiele wspólnych elementów, a nawet klasy o takich samych nazwach, które spełniały te same funkcje.

Co więc oznacza to dla programistów? Oto moje osobiste przemyślenia:

Tylko język C#, na razie brak obsługi Visual Basic.NET – w mojej opinii nie stanowi to większego problemu.

Otwarty kod źródłowy jest świetny! Jeśli ktoś chce coś zmienić, może po prostu pobrać kod z serwisu **GitHub** i samemu wprowadzić żądane zmiany! Jeśli będą wystarczająco dobre, można zainteresować nimi innych i zaproponować ich zintegrowanie z platformą.

Nie trzeba decydować z góry, czy chcemy korzystać z MVC, czy z Web API – to jedynie kwestia dodania w dowolnym momencie jednego lub dwóch pakietów NuGet oraz wpisania kilku wierszy w pliku `Startup.cs`; ten sam kontroler może obsługiwać zarówno interfejs API, jak i żądania WWW.

Wbudowany jest routing poprzez atrybuty, więc nie ma potrzeby jawnej konfiguracji.

ASP.NET Core wykorzystuje teraz konfigurację i oprogramowanie pośrednie (middleware) oparte na **OWIN** (**Open Web Interface for .NET**), więc trzeba będzie (znacząco) zmienić swoje moduły i procedury obsługowe, aby pasowały do tego modelu; filtry MVC/Web API są właściwie takie same.

Brak zależności od IIS albo systemu Windows, co oznacza, że możemy łatwo pisać swoje aplikacje w starym dobrym Windows/Visual Studio, a następnie po prostu wdrażać je na Azure/AWS/Docker/Linux/macOS. Całkiem fajne jest debugowanie aplikacji w Docker/Linux z poziomu Visual Studio! Witryna WWW może też być uruchamiana samodzielnie z poziomu aplikacji konsolowej.

Konsekwencją tego jest brak zarządzania przez **IIS Manager** i plików `web.config/machine.config`.

Nie wszystkie biblioteki są już dostępne dla .NET Core, co oznacza, że będziemy musieli znaleźć dla nich zastępniki lub sami zaimplementować potrzebne funkcje. Witryna *https://icanhasdot.net/Stats* prowadzi listę elementów, które są i nie są dostępne dla .NET Core, jest też lista zawarta w mapie drogowej projektu pod adresem *https://github.com/dotnet/core/blob/master/roadmap.md*.

Nawet w podstawowych klasach .NET Core nadal brakuje niektórych metod, przykładem jest klasa `System.Environment`.

Trzeba ręcznie wybierać pakiety NuGet dla bibliotek, z których chcemy skorzystać, w tym dla klas, które były dawniej dostępne automatycznie. W przypadku .NET obejmuje to na przykład kolekcje z `System.Collections` (*https://www.nuget.orgpackages/System.Collections)*, gdyż odwołanie do nich nie jest dodawane automatycznie.

Czasami ciężko jest znaleźć, który pakiet NuGet zawiera potrzebne nam klasy – w takiej sytuacji przydatny może być serwis *http://packagesearch.azurewebsites.net*.

Brak obsługi Web Forms (i projektowania wizualnego w Visual Studio), teraz wszystko opiera się na MVC!

.NET Core

Omówienie ASP.NET Core nie jest możliwe bez wyjaśnienia .NET Core. .NET Core jest platformą, o której wszyscy mówią nie bez powodu. ASP.NET jest chyba obecnie najciekawszym interfejsem API dla tej platformy, gdyż na razie nie zawiera ona żadnych innych bibliotek graficznego interfejsu użytkownika. Tak, tak – nie ma w niej Windows Forms, Windows Presentation Framework ani nawet Windows Services!

Jaka jest tego przyczyna? Wszystkie te interfejsy API polegały w dużym stopniu na rodzimych funkcjach Windows; w istocie formularze Windows Forms były jedynie opakowaniem dla interfejsów Win32 API, które towarzyszyły systemowi Windows od jego wczesnych lat. Ponieważ .NET Core jest przeznaczone dla wielu platform systemowych, niezwykle trudno byłoby opracować wersje tych interfejsów API dla wszystkich wspieranych platform. Nie oznacza to oczywiście, że ich nie będzie w przyszłości, a jedynie to, że obecnie ich nie ma.

W przypadku .NET Core komputer potrzebuje jedynie dość niewielkiego kodu startowego do uruchomienia aplikacji; sama aplikacja musi dołączyć wszystkie biblioteki, które są jej potrzebne do działania. Co ciekawe, możliwa jest kompilacja aplikacji .NET Core do rodzimego formatu, co daje plik wykonywalny specyficzny dla danego komputera, który zawiera wszystkie zależności i może być uruchamiany na nim bez kodu startowego .NET Core.

Jak wspominałem wcześniej, .NET Core napisano od podstaw, co niestety oznacza, że nie wszystkie interfejsy API, do których byliśmy przyzwyczajeni, zostały przeniesione. W szczególności w wersjach .NET Core 1.1 oraz 2.0 brakuje nadal następujących funkcji:

- ASP.NET Web Forms (`System.Web.UI`)
- XML Web Services (`System.Web.Services`)
- LINQ to SQL (`System.Data.Linq`)
- Windows Forms (`System.Windows.Forms`)
- Windows Presentation Foundation (`System.Windows` i `System.Xaml`)
- klasy po stronie serwera Windows Communication Foundation (`System.ServiceModel`)

- Windows Workflow Foundation (`System.Workflow` i `System.Activities`)
- .NET Remoting (`System.Runtime.Remoting`)
- Niektóre interfejsy API ADO.NET takie jak `DataSet`/`DataView` (`System.Data`) oraz części modelu dostawców ADO.NET (`System.Data.Common`)
- Generowanie kodu (`System.CodeDom`)
- Transakcje rozproszone (`System.Transactions`)
- Active Directory/LDAP (`System.DirectoryServices`)
- Enterprise Services (`System.EnterpriseServices`)
- Poczta elektroniczna (`System.Net.Mail`)
- XML i XSD (`System.Xml.Xsl` i `System.Xml.Schema`)
- Porty wejścia/wyjścia (`System.IO.Ports`)
- Zarządzana platforma wtyczek Addin (`System.Addin`)
- Mowa (`System.Speech`)
- Konfiguracja (`System.Configuration`), ten element został zastąpiony nowym interfejsem API dla konfiguracji
- Windows Management Instrumentation (`System.Management`)
- Funkcjonalność rysowania (`System.Drawing`), choć istnieją niektóre struktury
- Domeny aplikacji (`System.AppDomain`) i Rejestr systemu Windows (`Microsoft.Win32`)

Nie jest to bynajmniej wyczerpująca lista. Jak widać, brakuje wielu funkcji. Mimo to można osiągnąć wiele pod warunkiem, że będziemy działać w inny sposób i pokonamy dodatkowe przeszkody!

Poniższe interfejsy API są dostępne i można z nich bezpiecznie korzystać:

- MVC i Web API (`Microsoft.AspNetCore.Mvc`)
- Entity Framework Core (`Microsoft.EntityFrameworkCore`)
- Roslyn do generowania i analizowania kodu (`Microsoft.CodeAnalysis`)
- Wszystkie interfejsy API związane z Azure
- Managed Extensibility Framework (`System.Composition`)
- Szyfrowanie/odszyfrowywanie tekstu i przetwarzanie wyrażeń regularnych (`System.Text`)
- Serializacja JSON (`System.Runtime.Serialization.Json`)
- Niskopoziomowe generowanie kodu (`System.Reflection.Emit`)
- Większość ADO.NET (`System.Data`, `System.Data.Common`, `System.Data.SqlClient`, `System.Data.SqlTypes`)

- LINQ i Parallel LINQ (`System.Linq`)
- Kolekcje, w tym współbieżne (`System.Collections`, `System.Collections.Generic`, `System.Collections.ObjectModel`, `System.Collections.Specialized`, `System.Collections.Concurrent`)
- Wątki, komunikacja między procesami i zadania (`System.Threading`)
- Wejście/wyjście, kompresja, pamięć izolowana, pliki mapowane w pamięci, potoki (`System.IO`)
- XML (`System.Xml`)
- Klasy po stronie klienta Windows Communication Foundation (`System.ServiceModel`)
- Kryptografia (`System.Security.Cryptography`)
- Wywoływanie platformowe i COM Interop (`System.Runtime.InteropServices`)
- Universal Windows Platform (`Windows`)
- Śledzenie zdarzeń dla Windows (`System.Diagnostics.Tracing`)
- Adnotacje danych (`System.ComponentModel.DataAnnotations`)
- Obsługa sieci, w tym HTTP (`System.Net`)
- Refleksja (`System.Reflection`)
- Matematyka i obliczenia (`System.Numerics`)
- Reactive Extensions (`System.Reactive`)
- Globalizacja i lokalizacja (`System.Globalization`, `System.Resources`)
- Pamięć podręczna (w tym Redis) (`Microsoft.Extensions.Caching`)
- Rejestrowanie (`Microsoft.Extensions.Logging`)

Ta lista znów nie jest pełna, ale daje dobry obraz tego, co jest dostępne. Są to jedynie interfejsy API udostępniane przez Microsoft dla .NET Core; istnieją oczywiście tysiące innych od innych podmiotów.

Dlaczego te interfejsy API są obsługiwane? Są one określone w standardzie .NET (**.NET Standard**), a .NET Core implementuje ten standard! Więcej na ten temat za chwilę.

W .NET Core nie ma już globalnej pamięci podzespołów (**GAC – Global Assembly Cache**), ale istnieje scentralizowane miejsce (dla danego użytkownika) do przechowywania pakietów NuGet: `%HOMEPATH%.nugetpackages`, co zapobiega lokalnemu przechowywaniu zduplikowanych pakietów dla różnych projektów. Wersja .NET Core 2.0 wprowadziła magazyn uruchomieniowy (**runtime store**), który jest nieco podobny

do pamięci GAC. Jest to w zasadzie folder na lokalnym komputerze, gdzie udostępniane są pewne pakiety i kompilowane dla architektury danego komputera. Pakiety tam przechowywane nie są nigdy pobierane z NuGet; odwołania do nich są lokalne i nie trzeba ich dołączać do swojej aplikacji. Mile widziany dodatek! Więcej na temat metapakietów i magazynu uruchomieniowego można przeczytać tutaj: *https://docs.microsoft.com/en-us/aspnet/core/fundamentals/metapackage*.

Metapakiet (zestaw pakietów) `Microsoft.AspNetCore.All` zawiera:

- Wszystkie pakiety wspierane przez zespół ASP.NET Core
- Wszystkie pakiety wspierane przez Entity Framework Core
- Zależności wewnętrzne i pochodzące od firm trzecich, wykorzystywane przez ASP.NET Core i Entity Framework Core

Szablony Visual Studio dla .NET Core 2 zawierają już odwołanie do tego metapakietu.

Pakiety NuGet stanowią serce .NET Core, a niemal wszystko trzeba uzyskać z NuGet. Nawet projekty w tym samym rozwiązaniu Visual Studio odwołują się do siebie poprzez pakiety NuGet. Przy korzystaniu z .NET Core trzeba będzie jawnie dodawać pakiety NuGet, zawierające funkcjonalność, którą chcemy wykorzystać. Możemy prawdopodobnie natknąć się na niektóre z poniższych pakietów w swoich projektach:

Pakiet	**Zadanie**
`Microsoft.EntityFrameworkCore`	Entity Framework Core
`Microsoft.Extensions.Caching.Memory`	Pamięć podręczna
`Microsoft.Extensions.Caching.Redis`	Pamięć podręczna Redis
`Microsoft.Extensions.Configuration`	Ogólne klasy konfiguracyjne
`Microsoft.Extensions.Configuration.EnvironmentVariables`	Konfiguracja ze zmiennych środowiskowych
`Microsoft.Extensions.Configuration.Json`	Konfiguracja z plików JSON
`Microsoft.Extensions.Configuration.UserSecrets`	Konfiguracja tajemnic użytkownika (*https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets)*
`Microsoft.Extensions.Configuration.Xml`	Konfiguracja w XML
`Microsoft.Extensions.DependencyInjection`	Wbudowana platforma wstrzykiwania zależności
`Microsoft.Extensions.Logging`	Podstawowe klasy rejestrowania

Pakiet	Zadanie
`Microsoft.Extensions.Logging.Console`	Rejestrowanie na konsoli
`Microsoft.Extensions.Logging.Debug`	Rejestrowanie przy debugowaniu
`System.Collections`	Kolekcje
`System.ComponentModel`	Klasy i interfejsy wykorzystywane w definicji składników i źródeł danych
`System.ComponentModel.Annotations`	Adnotacje danych do sprawdzania poprawności i metadanych
`System.Data.Common`	ADO.NET
`System.Globalization`	Interfejsy API do globalizacji i lokalizacji
`System.IO`	Interfejsy API do obsługi wejścia/wyjścia
`System.Linq.Parallel`	Parallel LINQ
`System.Net`	Interfejsy API do obsługi sieci
`System.Reflection`	Refleksja
`System.Security.Claims`	Bezpieczeństwo oparte na poświadczeniach
`System.Threading.Tasks`	Implementacja zadań
`System.Xml.XDocument`	Interfejsy API do obsługi XML

Ta lista również nie jest wyczerpująca. Można nie dostrzec odwołań do wszystkich tych pakietów, ponieważ dodanie jednego pakietu, który ma zależności od innych, dołączy też wszystkie te zależności, a **duże** pakiety mają bardzo dużo zależności.

Nie ma już plików `.exe`; teraz wszystkie podzespoły są bibliotekami `.dll`, co oznacza, że trzeba je uruchamiać, korzystając z narzędzia wiersza poleceń `dotnet`. Wszystkie aplikacje .NET Core zaczynają się od metody statycznej `Main` tak jak *stare* aplikacje konsolowe i Windows Forms, ale teraz potrzebujemy narzędzia dotnet do ich uruchamiania. Narzędzie dotnet jest bardzo wszechstronne i może być używane do budowania, uruchamiania, wdrażania i przywracania pakietów NuGet, wykonywania testów jednostkowych oraz tworzenia pakietów NuGet z projektu. Jak mówiłem, możliwe też jest skompilowanie podzespołu do formatu rodzimego, ale nie będziemy tego tutaj omawiać.

Platforma .NET Core jest dostarczana z wbudowanym systemem **wstrzykiwania zależności** (**DI – dependency injection**), rejestrowaniem i elastyczną platformą konfigurowania, która pozwala dołączać własnych dostawców konfiguracji. Wszystkie

nowe interfejsy API (takie jak Entity Framework Core oraz ASP.NET Core) wykorzystują te usługi. Po raz pierwszy widzimy spójne zachowanie wielu interfejsów API.

Większość interfejsów API takich jak ASP.NET i Entity Framework pozwala zastępować usługi, na których są oparte, wersjami niestandardowymi, co umożliwia precyzyjne dostosowanie ich działania pod warunkiem oczywiście, że ma się odpowiednią wiedzę. Usługi te zwykle oparte są na interfejsach. Wszystko jest dużo bardziej modularne i przejrzyste.

Testy jednostkowe mają swoje miejsce w .NET Core. Większość nowych interfejsów API została zaprojektowana z uwzględnieniem łatwości testowania, a narzędzia (**dotnet**) mają jawną opcję wykonywania testów jednostkowych, które mogą być pisane na dowolnej platformie do obsługi testów (obecnie między innymi xUnit, NUnit, MbUnit oraz Microsoft udostępniły platformy do testów jednostkowych kompatybilne z .NET Core). Testowanie jednostkowe omówimy bardziej szczegółowo w dalszej części tej książki.

Platformy systemowe

.NET Core działa na następujących platformach:

- Windows 7 SP1 lub wyższa wersja
- Windows Server 2008 R2 SP1 lub wyższa wersja
- Red Hat Enterprise Linux 7.2 lub wyższa wersja
- Fedora 23 lub wyższa wersja
- Debian 8.2 lub wyższa wersja
- Ubuntu 14.04 LTS/16.04 LTS lub wyższa wersja
- Linux Mint 17 lub wyższa wersja
- openSUSE 13.2 lub wyższa wersja
- Centos 7.1 lub wyższa wersja
- Oracle Linux 7.1 lub wyższa wersja
- macOS X 10.11 lub wyższa wersja

Obejmuje to wszystkie nowoczesne dystrybucje systemów Windows, Linux i macOS (wersja Windows 7 SP1 była wydana w 2010 roku). Może też dobrze działać w innych dystrybucjach, ale te wymienione powyżej zostały dokładnie przetestowane przez Microsoft.

Jak więc to działa? Okazuje się, że gdy zgłaszamy żądanie pakietu NuGet, wymagającego bibliotek rodzimych, które nie są zawarte w systemie operacyjnym, są one

również dołączane w archiwum `.nupkg`. .NET Core wykorzystuje wywoływanie platformowe **P/Invoke** (**Platform Invoke**) do wywoływania bibliotek specyficznych dla systemu operacyjnego. Oznacza to, że nie musimy się tym martwić – proces znajdowania, dodawania pakietu NuGet i publikowania projektu jest taki sam niezależnie od tego, jaki jest docelowy system operacyjny.

Trzeba mieć na uwadze, że niezależność od platformy jest dla programistów niewidoczna, chyba że ktoś jest autorem biblioteki i musi o to samemu zadbać.

Zależności i platformy

Wewnątrz projektu .NET Core określamy platformy docelowe. Jakie to są platformy? No cóż, może to być sama platforma .NET Core, ale także *klasyczna* platforma .NET Framework, Xamarin, **Universal Windows Platform** (**UWP**), **Portable Class Libraries** (**PCL**), Mono, Windows Phone i jeszcze więcej.

We wczesnych dniach .NET Core można było określić jako platformę docelową samą platformę .NET Core i/lub jedną z tych innych platform. Obecnie zaleca się określanie standardów jako platform docelowych. Jak stwierdza Immo Landwerth z Microsoft (*https://blogs.msdn.microsoft.com/dotnet/2016/09/26/introducing-net-standard):*

.NET Standard rozwiązuje problem z udostępnianiem kodu przez programistów .NET dla wszystkich platform poprzez zapewnianie wszystkich oczekiwanych interfejsów API w żądanych środowiskach: aplikacjach biurkowych, aplikacjach i grach mobilnych oraz usługach w chmurze:

- ***.NET Standard** jest zestawem interfejsów API, które muszą być implementowane przez wszystkie platformy .NET. Unifikuje to platformy .NET i zapobiega przyszłej fragmentacji.*
- ***.NET Standard 2.0** jest implementowany przez platformy **.NET Framework**, **.NET Core**, **Mono** i **Xamarin**. W przypadku .NET Core powoduje to dodanie wielu istniejących i oczekiwanych interfejsów API.*
- ***.NET Standard 2.**0 zawiera warstwę kompatybilności z plikami binarnymi .NET Framework, znacząco zwiększając zestaw bibliotek, do których możemy odwoływać się ze swoich bibliotek .NET Standard.*
- ***.NET Standard** zastąpi biblioteki **Portable Class Libraries** (**PCLs**) jako narzędzie do budowania wieloplatformowych bibliotek .NET.*

Innymi słowy:

- **.NET Standard** jest specyfikacją określającą, które interfejsy API muszą być implementowane przez platformę .NET.

- **.NET Core** jest konkretną platformą .NET, która implementuje .NET Standard.
- Najnowszy .NET Standard będzie zawsze obejmował najnowszą wydaną pełną platformę .NET.

David Fowler (*https://twitter.com/davidfowl)* z firmy Microsoft przedstawił następującą analogię:

```
interface INetStandard10
{
   void Primitives();
   void Reflection();
   void Tasks();
   void Collections();
   void Linq();
}
interface INetStandard11 : INetStandard10
{
  void ConcurrentCollections();
   void InteropServices();
}
interface INetFramework45 : INetStandard11
{
  // Interfejsy API specyficzne dla platformy
   void AppDomain();
   void Xml();
   void Drawing();
   void SystemWeb();
   void WPF();
   void WindowsForms();
   void WCF();
}
```

Powinno to być łatwe do zrozumienia. Jak widać, wszystkie interfejsy API w .NET, które wymagają Windows (WPF, Windows Forms, Drawing), są dostępne jedynie na określonej platformie (.NET 4.5), a nie w standardzie. Standardy zapewniają funkcjonalność dla wielu platform.

Więcej informacji można uzyskać pod adresem *https://docs.microsoft.com/enus/dotnet/articles/standard/library*.

Zamiast więc określać jako cel konkretną wersję, taką jak .NET 4.5.1, .NET Core 1.0, Mono, Universal Windows Platform 10 lub Windows Phone 8.1, możemy określać jako cel standard .NET. Mamy gwarancję, że nasz projekt będzie działał na wszystkich platformach, które obsługują ten standard (lub wyższy), zarówno istniejących, jak i dopiero czekających na stworzenie. Należy starać się zachowywać zgodność

z najniższym możliwym standardem, aby zwiększyć liczbę platform, na których nasza aplikacja będzie działać, jeśli to ma dla nas znaczenie.

Poniższa tabela wypisuje wszystkie wersje .NET Standard oraz platformy wspierane przez poszczególne standardy.

.NET Standard	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**	**1.6**	**2.0**
.NET Core	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**	**2.0**
.NET Framework	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1 4.6.2	4.6.1 vNext	4.6.1
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4
Xamarin.iOS	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.14
Xamarin.Mac	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.8
Xamarin.Android	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.5
Universal Windows Platform	10.0	10.0	10.0	10.0	10.0	vNext	vNext	vNext
Windows	8.0	8.0	8.1					
.NET Standard	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**	**1.6**	**2.0**
Windows Phone	8.1	8.1	8.1					
Windows Phone Silverlight	8.0							

Ta tabela przedstawia aktualne dopasowanie pomiędzy różnymi platformami .NET i implementowanymi przez nie standardami .NET, a najnowsza jej wersja jest zawsze dostępna pod adresem: *https://github.com/dotnet/standard/blob/master/docs/versions.md*.

Każda wersja .NET Standard obsługuje pewne interfejsy API, a każda wyższa wersja obsługuje ich więcej, jak można zobaczyć w poniższej tabeli:

Wersja	**Liczba interfejsów API**	**Wzrost w %**
1.0	7949	
1.1	10239	+29%
1.2	10285	+0%
1.3	13122	+28%
1.4	13140	+0%
1.5	13355	+2%

Wersja	Liczba interfejsów API	Wzrost w %
1.6	13501	+1%
2.0	32638	+142%

Wersje .NET Core 2.0 i .NET Standard 2.0 zostały udostępnione w sierpniu 2017 i obecnie cztery platformy obsługują .NET Standard 2.0:

- .NET Framework (pełna platforma)
- .NET Core 2.0
- Xamarin
- Mono

Możemy określać swoje zależności osobno dla każdej platformy docelowej albo dla wszystkich platform docelowych. W tym pierwszym przypadku możemy mieć różne zależności dla każdej platformy docelowej, a w tym drugim wszystkie zależności muszą wspierać wszystkie platformy docelowe. Możemy mieszać oba podejścia, określając wspólne biblioteki jako zależności globalne, a bardziej wyspecjalizowane biblioteki określać tylko tam, gdzie są dostępne. Jeśli celem jest więcej niż jeden standard (lub platforma), trzeba uważać, gdyż niektóre funkcje mogą istnieć tylko w jednym z celów i trzeba uciekać się wtedy do definicji warunkowych (`#if`).

Zbiór najczęstszych pytań dotyczących .NET Standard jest dostępny w serwisie GitHub pod adresem: *https://github.com/dotnet/standard/blob/master/docs/faq.md*.

Programowanie dla .NET Core lub pełnej platformy .NET Framework

Trzeba wiedzieć, że możemy określać pełną platformę .NET Framework jako cel w swoich aplikacjach ASP.NET Core! Jeśli to zrobimy, stracimy niezależność od platformy, to znaczy, będziemy mogli uruchamiać aplikację tylko w systemie Windows.

Domyślnie projekty ASP.NET Core określają jako cele `netcoreapp1.0` lub `netcoreapp2.0`, w zależności od tego, czy stosujemy wersję ASP.NET Core 1.x, czy 2.x, ale możemy to zmienić w pliku `.csproj`. Jeśli chcemy pisać aplikację tylko dla jednej platformy, możemy zmodyfikować element `TargetFramework` następująco:

```
<TargetFramework>net461</TargetFramework>
```

Jeśli natomiast chcemy obsługiwać więcej niż jedną platformę, możemy zastąpić `TargetFramework` przez `TargetFrameworks`:

```
<TargetFrameworks>netcoreapp2.0;net461</TargetFrameworks>
```

Więcej informacji można znaleźć w dokumentacji firmy Microsoft: *https://docs.microsoft.com/en-us/dotnet/core/tools/csproj*.

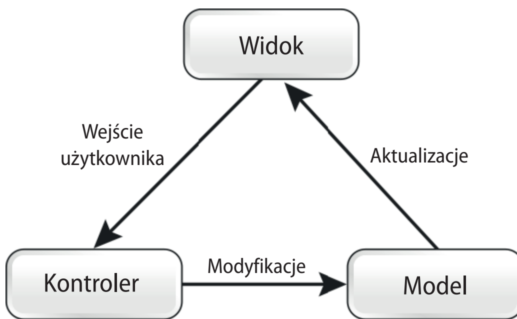
Zrozumienie wzorca MVC

Wracając do ASP.NET. Dla tych, którzy dotąd pracowali tylko z Web Forms, wyjaśnimy krótko, czym jest wzorzec MVC i skąd się wziął?

Mówiąc szczerze, w Web Forms łatwo było robić niewłaściwe rzeczy, na przykład dodawać kod na stronie (który nie byłby kompilowany aż do momentu, gdy przeglądarka zażądałaby danej strony), dodawać skomplikowaną logikę biznesową do klasy strony, utrzymywać kilka megabajtów danych w obiekcie ViewState i przesyłać je tam i z powrotem przy każdym żądaniu, itd. Nie było żadnych odgórnych mechanizmów wymuszających działanie we właściwy sposób. Ciężko też było przeprowadzać testy jednostkowe, ponieważ opierały się one na przesyłaniu danych przez przeglądarkę (`POST`) i kodzie JavaScript, żeby wszystko poprawnie działało (na przykład wiązanie akcji z procedurami obsługi zdarzeń i przesyłanych wartości z kontrolkami). Musiało być inne rozwiązanie.

Wzorzec projektowy **MVC** (**Model-View-Controller**) został zdefiniowany w późnych latach 70-ych i na początku lat 80-ych poprzedniego wieku. Powstał jako sposób na odpowiednie oddzielenie rzeczy, które nie pasują do siebie pojęciowo, na przykład kodu renderującego **interfejs użytkownika** (**UI** – user interface), kodu, który zawiera logikę biznesową i dostępu do danych, który ma dostarczać dane dla tego interfejsu użytkownika. W paradygmacie MVC (i jego potomkach) mamy kontrolery, które udostępniają publiczne akcje. Wewnątrz każdej akcji kontroler (controller) wykonuje wszelką potrzebną logikę biznesową, a następnie decyduje, który widok (view) powinien zostać wyrenderowany, przekazując mu odpowiednie informacje (model), aby mógł wykonać swoje zadanie. Kontroler nie wie nic na temat elementów interfejsu użytkownika, jedynie wykorzystuje kontekst wykonywania i dane potrzebne do wykonania danej akcji. Podobnie widok nie wie nic o bazach danych, usługach WWW, łańcuchach połączeń, SQL, itd. – po prostu renderuje dane i ewentualnie podejmuje proste decyzje na nich oparte. Jeśli chodzi o model, jest to pojemnik, zawierający informacje wymagane przez widok, takie jak listy rekordów, statyczne informacje o użytkowniku, itp. Ten ścisły podział ułatwia zarządzanie, testowanie i implementowanie. Wzorzec MVC nie jest specyficzny dla aplikacji WWW, może być używany wszędzie, gdzie takie rozdzielenie obowiązków jest przydatne, jeśli mamy interfejs użytkownika i jakiś kod, który nim steruje.

Poniższy diagram przedstawia relację pomiędzy widokami, kontrolerami i modelami:



Wzorzec MVC jest normalnie kojarzony z **programowaniem zorientowanym obiektowo**, ale istnieją implementacje w wielu językach, w tym JavaScript i PHP. Implementacja .NET MVC ma następujące podstawowe charakterystyki:

- Klasy kontrolerów (**Controller**) są albo zwykłymi obiektami **POCO (Plain Old CLR Object**), albo dziedziczą po klasie bazowej Controller. Dziedziczenie po klasie Controller nie jest wymagane (jak we wcześniejszych wersjach), ale nieco ułatwia sprawy. Obiekty klas kontrolerów są tworzone przez platformę wstrzykiwania zależności (**dependency injection**) w ASP.NET Core, co oznacza, że można do nich przekazywać usługi, od których zależą.
- **Akcje** są metodami publicznymi w kontrolerach; mogą przyjmować parametry zarówno prostych typów, jak i złożonych (obiekty POCO); MVC wykorzystuje wiązanie modelu do tłumaczenia informacji przesyłanych z przeglądarki (przez łańcuch zapytania, nagłówki, pliki cookie, formularze, wstrzykiwanie zależności, itd.) na parametry metod. Wybór metody kontrolera, którą należy wywołać w odpowiedzi na dane żądanie (URL i przesłane parametry), dokonywany jest przez kombinację tabeli routingu, konwencji i atrybutów pomocniczych.
- **Model** jest przesyłany z kontrolera do widoku w odpowiedzi na metodę akcji i może być czymkolwiek; oczywiście metody akcji dla wywołań API nie zwracają widoków, ale mogą zwracać model wraz z kodem stanu HTTP. Istnieją inne sposoby przekazywania danych do widoku, mianowicie pojemnik widoku (view bag), który jest w zasadzie ogólnym słownikiem danych; model z kolei ma zwykle określony typ. Poprawność modelu jest automatycznie sprawdzana, a jego zawartość jest wiązana z parametrami metod akcji.
- **Widoki** składają się z plików w językach specyficznych dla domeny, które są interpretowane przez silnik widoków i zamieniane na coś, co może być interpretowane

przez przeglądarkę, na przykład kod HTML. ASP.NET Core obsługuje rozszerzalną platformę silników widoków, ale zawiera pojedynczą implementację o nazwie **Razor**. Razor oferuje prostą składnię, która pozwala programistom mieszać kod HTML i C#, aby obsługiwać dane przekazane w modelu i podejmować decyzje, co z nimi robić. Widoki mogą być ograniczane przez układy (programiści Web Forms mogą je traktować jako strony szablonowe) i mogą zawierać inne widoki częściowe (podobne do kontrolek użytkownika w Web Forms). Widok dla silnika widoków Razor ma rozszerzenie `.cshtml` i nie jest dostępny bezpośrednio, a jedynie w wyniku wywołania akcji. Widoki mogą być wstępnie kompilowane, więc błędy składniowe są wykrywane wcześniej.

- **Filtry** są używane do przechwytywania, modyfikowania lub całkowitego zastępowania żądań; wbudowane filtry mogą zapobiegać dostępowi przez nieuwierzytelnionych użytkowników albo przekierowywać na stronę błędu w przypadku wystąpienia wyjątku.

Obecnie istnieją inne wzorce o podobnym przeznaczeniu co MVC, na przykład **Model-View-Presenter** (**MVP**) albo **Model-View-ViewModel** (**MVVM**). Skupimy się jedynie na implementacji MVC firmy Microsoft i jej szczegółach. W szczególności częścią ASP.NET Core jest wersja 6 MVC, opierająca się na wcześniejszej wersji 5, która była już dostępna dla pełnej platformy .NET. Ponieważ jest osadzona na platformie .NET Core, jest w pełni oparta na **OWIN**, więc nie ma już pliku `Global.asax.cs`. Więcej na ten temat za chwilę.

Sposób implementacji MVC w ASP.NET skupia się na:

- **Adresach URL**: Są one teraz bardziej treściwe i przyjazne dla technik optymalizacji dla silników wyszukiwawczych (**SEO – Search Engine Optimization**).
- **Poleceniach HTTP:** Polecenie teraz dokładnie określa, jaka operacja ma być wykonana, na przykład `GET` dla działań odczytujących, `POST` dla nowej zawartości, `PUT` dla pełnej aktualizacji zawartości, `PATCH` dla częściowej aktualizacji, `DELETE` dla usuwania, itd.
- **Kodach stanu HTTP:** Do zwracania kodów wyników operacji, które są ważniejsze w przypadku interfejsów Web API.

Na przykład użycie `GET` dla żądania *http://somehost/Product/120* prawdopodobnie zwróci widok dla produktu o identyfikatorze `120`, a żądanie `DELETE` dla tego samego adresu URL prawdopodobnie usunie ten produkt i zwróci jakiś kod stanu HTTP albo nowy widok informujący o tym fakcie. Adresy URL oraz ich powiązania z kontrolerami i akcjami mogą być konfigurowane przez routing i jest prawdopodobne, że ten adres URL będzie obsługiwany przez jakiś kontroler o nazwie `ProductController`

oraz jakieś metody akcji, które są skonfigurowane do obsługi żądań `GET` lub `DELETE`. Widoków nie da się wyciągnąć z adresów URL, ponieważ są ustalane wewnątrz metod akcji.

Implementację wzorca MVC firmy Microsoft omówimy dogłębnie w kolejnych rozdziałach. Ponieważ jest elementem .NET Core, oczywiście wszystkie składniki tej implementacji są dostępne jako pakiety NuGet. Kilka z nich wymieniono w poniższej tabeli:

Pakiet	Zadanie
`Microsoft.AspNetCore.Antiforgery`	Interfejsy API obsługujące przeciwdziałanie fałszowaniu
`Microsoft.AspNetCore.Authentication`	Klasy bazowe do uwierzytelniania
`Microsoft.AspNetCore.Authentication.Cookies`	Uwierzytelnianie przez pliki cookie
`Microsoft.AspNetCore.Authorization`	Interfejsy API do autoryzacji
`Microsoft.AspNetCore.Diagnostics`	Interfejsy API do diagnostyki
`Microsoft.AspNetCore.Hosting`	Podstawowe klasy hostingu
`Microsoft.AspNetCore.Identity`	Uwierzytelnianie tożsamości
`Microsoft.AspNetCore.Identity.EntityFrameworkCore`	Tożsamość wykorzystująca Entity Famework Core jako magazyn
`Microsoft.AspNetCore.Localization.Routing`	Lokalizacja poprzez routing
`Microsoft.AspNetCore.Mvc`	Podstawowe funkcje MVC
`Microsoft.AspNetCore.Mvc.Cors`	Wsparcie dla CORS (Cross Origin Request Scripting)
`Microsoft.AspNetCore.Mvc.DataAnnotations`	Sprawdzanie poprawności poprzez adnotacje danych
`Microsoft.AspNetCore.Mvc.Localization`	Interfejsy API dotyczące lokalizacji
`Microsoft.AspNetCore.Mvc.TagHelpers`	Funkcjonalność pomocników znacznikowych (tag helpers)
`Microsoft.AspNetCore.Mvc.Versioning`	Wersjonowanie Web API
`Microsoft.AspNetCore.ResponseCaching`	Pamięć podręczna dla odpowiedzi
`Microsoft.AspNetCore.Routing`	Routing
`Microsoft.AspNetCore.Server.IISIntegration`	Integracja z IIS

Pakiet	Zadanie
`Microsoft.AspNetCore.Server.Kestrel`	Serwer Kestrel
`Microsoft.AspNetCore.Server.WebListener` (`Microsoft.AspNetCore.Server.HttpSys` w ASP.NET Core 2)	Serwer WebListener (teraz zwany `HTTP.sys`). Zobacz *https://docs.microsoft.com/enus/aspnet/core/fundamentals/servers/httpsys.*
`Microsoft.AspNetCore.Session`	Funkcjonalność sesji
`Microsoft.AspNetCore.StaticFiles`	Możliwość obsługi plików statycznych

W konkretnej sytuacji nie wszystkie z tych pakietów muszą być potrzebne, ale warto się z nimi zapoznać.

W ASP.NET Core 2.0 istnieje metapakiet NuGet `Microsoft.AspNetCore.All`. Zawiera on większość potrzebnych pakietów, więc wystarczy dodać tylko ten jeden. Trzeba pamiętać, że wykorzystuje on `netcoreapp2.0`.

Uzyskiwanie kontekstu

Wszyscy pewnie pamiętają klasę `HttpContext` z ASP.NET. Wystąpienie tej klasy reprezentuje aktualny kontekst wykonywania, który zawiera informacje o zapytaniu oraz kanał odpowiedzi. Kontekst był zawsze obecny, choć w Web Forms był nieco ukryty. Był sposobem komunikacji aplikacji WWW z klientem.

Oczywiście ASP.NET Core również ma klasę `HttpContext`, ale istnieje duża różnica – nie ma już statycznej właściwości `Current`, która pozwalałaby na dostęp do bieżącego kontekstu, a proces ten jest nieco bardziej zagmatwany. W każdym razie wszystkie klasy infrastrukturalne – kontrolery, widoki, składniki widoków, pomocniki znacznikowe i filtry pozwalają na łatwy dostęp do bieżącego kontekstu.

Oprócz właściwości `Request` i `Response`, które są w większości podobne do swoich odpowiedników sprzed wersji Core, mamy też:

- Kolekcję `Features`, która udostępnia wszystkie funkcje implementowane przez bieżący serwer hostingowy (Kestrel, WebListener/`HTTP.sys` itd).
- Właściwość `RequestServices`, która daje nam dostęp do wbudowanej platformy wstrzykiwania zależności, o której więcej informacji będzie w następnych rozdziałach.

- Właściwość `TraceIdentifier`, która jednoznacznie identyfikuje żądanie w ASP.NET Core 2.x; we wcześniejszych wersjach musieliśmy uzyskać do niej dostęp przez osobną funkcję.
- Obiekt `Connection`, z którego możemy uzyskać odpowiednie informacje o połączeniu klienckim, na przykład wszelkie certyfikaty klienta.
  - Obiekt `Authentication`, dający łatwy dostęp do podstawowych elementów zabezpieczeń.
  - Obiekt `Session`, który jest implementowany przez funkcję `ISessionFeature`.

Kontekst jest istotną częścią aplikacji ASP.NET Core, jak wkrótce zobaczymy.

Potok OWIN

Poprzednie wersje ASP.NET były bardzo blisko związane z usługami **Internet Information Services (IIS)**, flagowym serwerem WWW firmy Microsoft dostarczanym z systemem Windows. W istocie IIS był jedynym wspieranym sposobem uruchamiania ASP.NET.

Aby to zmienić, firma Microsoft zdefiniowała specyfikację **Open Web Interface for .NET (OWIN)**, o której można poczytać pod adresem *http://owin.org*. Krótko mówiąc, jest to standard pozwalający oddzielić serwer od kodu aplikacji i określający potok wykonawczy dla żądań WWW. Ponieważ jest to standard, jest on niezależny od serwera WWW i może służyć do wyodrębniania jego funkcji.

.NET Core korzysta w znacznym stopniu ze specyfikacji OWIN. Nie mamy już plików konfiguracyjnych `Global.asax`, `web.config` i `machine.config` ani modułów. Mamy natomiast:

- Kod startowy deklarujący klasę, która zawiera metody zdefiniowane przez konwencję (jeśli żadna klasa nie jest zadeklarowana, automatycznie będzie używana klasa `Startup`).
- Konwencjonalna metoda, która powinna mieć nazwę `Configure` i otrzymuje jako jedyny parametr wskaźnik do wystąpienia `IApplicationBuilder`.
- Następnie dodajemy oprogramowanie pośrednie do obiektu `IApplicationBuilder`; to oprogramowanie pośrednie będzie zajmować się obsługą naszych żądań WWW.

Warto pokazać to na prostym przykładzie. Zacznijmy od klasy startowej, która mogłaby się nazywać `Program`:

```
public class Program
{
  public static void Main()
  {
    var host = new WebHostBuilder()
        .UseKestrel()
        .UseStartup<Startup>()
        .Build();
       host.Run();
  }
}
```

Kod ten może być bardziej skomplikowany, ale na razie nie będziemy się tym zbytnio przejmować. Później wyjaśnię, co to wszystko znaczy. Na razie wystarczy wiedzieć, że wykorzystujemy `WebHostBuilder`, aby zastosować serwer Kestrel i przekazujemy konwencjonalną klasę o nazwie `Startup`. Ta klasa `Startup` wygląda następująco:

```
public class Startup
{
  public void Configure(IApplicationBuilder app)
  {
    app.Run(async (context) =>
    {
      await context.Response.WriteAsync("Hello, OWIN World!");
    }
  }
}
```

Kilka spraw tutaj zasługuje na wyjaśnienie. Po pierwsze można zauważyć, że klasa `Startup` nie implementuje żadnego interfejsu ani nie dziedziczy po żadnej konkretnej klasie bazowej. Jest tak dlatego, ponieważ metoda `Configure` nie ma predefiniowanej sygnatury, poza swoją nazwą i przyjmowaniem jako pierwszy parametr wystąpienia `IApplicationBuilder`. Na przykład poniższa metoda również jest dozwolona:

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

Ta wersja daje nam nawet więcej niż potrzeba. Odchodzę jednak od tematu.

Interfejs `IApplicationBuilder` definiuje metodę `Run`. Ta metoda zwraca `Task` i przyjmuje parametr `RequestDelegate`, który jest definicją delegata przyjmującego `HttpContext` jako jedyny parametr. W moim przykładzie uczyniliśmy z niej metodę asynchroniczną, dodając słowa kluczowe `async` i `await`, ale nie jest to konieczne. Trzeba tylko zadbać o wyodrębnienie żądanych elementów z obiektu `HttpContext` i zapisanie w nim wymaganych elementów – to jest nasz potok WWW. Obejmuje

to zarówno obiekty żądania, jak i odpowiedzi HTTP i nazywamy to oprogramowaniem pośrednim (middleware).

Metoda Run jest samodzielnym potokiem, ale możemy podłączyć do niego inne kroki (oprogramowanie pośrednie), korzystając z metody Use:

```
app.Use(async (context, next) =>
{
  await context.Response.WriteAsync("Hello from a middleware!");
  await next.Invoke();
}
```

W ten sposób możemy dodać wiele kroków, a będą one wykonywane w kolejności, w jakiej zostały zdefiniowane:

```
app.Use(async (context, next) =>
{
  await context.Response.WriteAsync("step 1!");
  await next.Invoke();
}
app.Use(async (context, next) =>
{
  await context.Response.WriteAsync("step 2");
  await next.Invoke();
}
```

Trzeba mieć na uwadze, że kolejność ma tutaj znaczenie!

Metoda Use przyjmuje jako parametr wystąpienie HttpContext i zwraca Func<Task>, co normalnie jest wywołaniem następnej procedury obsługowej w potoku przetwarzania.

Moglibyśmy wyodrębnić wyrażenie lambda do własnej metody, jak poniżej:

```
async Task Process(HttpContext context, Func<Task> next)
{
  await context.Response.WriteAsync("Step 1");
  await next.Invoke();
}
app.Use(Process);
```

Możliwie jest nawet wyodrębnienie oprogramowania pośredniego do własnej klasy i zastosowanie go poprzez użycie ogólnej metody UseMiddleware:

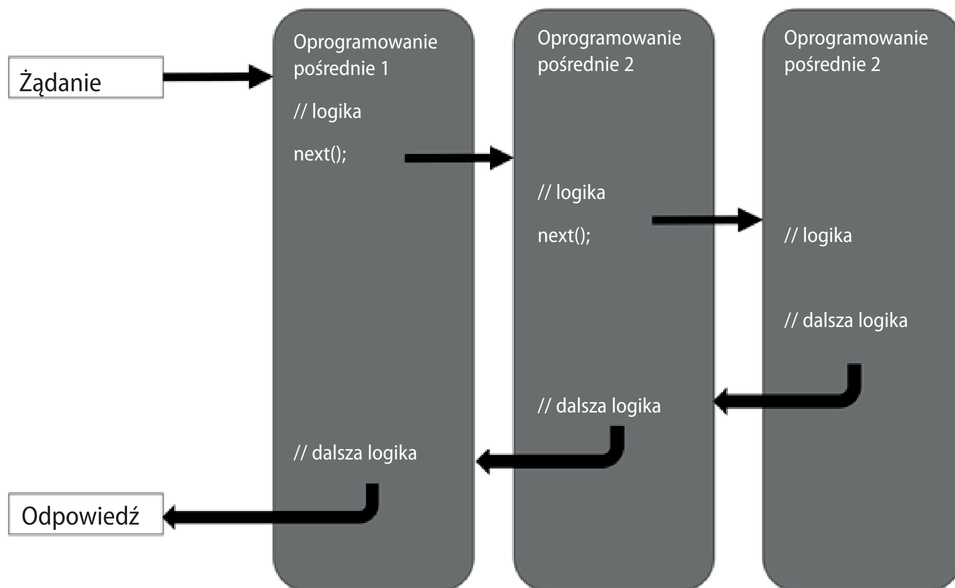
```
public class Middleware
{
  private readonly RequestDelegate _next;
  public Middleware(RequestDelegate next)
```

```
    {
      this._next = next;
    }
    public async Task Invoke(HttpContext context)
    {
      await context.Response.WriteAsync("This is a middleware class!");
      await this._next.Invoke(context);
    }
  }
  app.UseMiddleware<Middleware>();
```

W tym przypadku konstruktor musi przyjąć jako swój pierwszy parametr wskaźnik do następnego oprogramowania pośredniego w potoku w formie wystąpienia `RequestDelegate`.

Widać więc, że OWIN definiuje potok, do którego możemy dodawać procedury przetwarzania, które następnie są wykonywane sekwencyjnie. Różnica pomiędzy metodami `Run` i `Use` jest taka, że ta pierwsza kończy potok przetwarzania, to znaczy, że nie będzie wywoływać dalszych metod.

Poniższy diagram (pochodzący z dokumentacji Microsoft) jasno to pokazuje:



Pierwsze oprogramowanie pośrednie w pewnym sensie opakowuje wszystkie następne. Wyobraźmy sobie na przykład, że chcemy dodać obsługę wyjątków do wszystkich kroków w potoku przetwarzania; moglibyśmy zrobić coś takiego:

```
app.Use(async (context, next) =>
{
  try
  {
    await context.Response.WriteAsync("inside an exception handler");
    await next.Invoke();
  }
  catch (Exception ex)
  {
    // zrób coś w związku z wyjątkiem
  }
  await context.Response.WriteAsync("outside an exception handler");
}
```

Wywołanie `next.Invoke()` jest otoczone blokiem `try...catch`, więc przechwytywane będą wszelkie wyjątki, które mogłyby być wyrzucone przez inne (dodane później) oprogramowanie pośrednie w potoku.

Więcej na temat implementacji OWIN przez firmę Microsoft można przeczytać pod adresem: *https://docs.microsoft.com/en-us/aspnet/core/fundamentals/owin*.

Dlaczego standard OWIN jest tak ważny? Ponieważ ASP.NET Core i jego implementacja MVC są oparte na tym standardzie. Później zobaczymy, że aby uzyskać aplikację MVC, musimy dodać oprogramowanie pośrednie MVC do potoku OWIN w metodzie `Configure` klasy `Startup`, jak pokazano poniżej:

```
app.UseMvc();
```

Hosting

Gdy mówiliśmy o standardzie OWIN, wspomniałem, że aplikacja przykładowa jest obsługiwana przez serwer **Kestrel**. Kestrel jest nazwą niezależnego od platformy serwera WWW, który jest w pełni napisany w .NET Core (oczywiście z wykorzystaniem rodzimych bibliotek systemu operacyjnego). Musimy gdzieś uruchamiać swoją aplikację WWW, a .NET Core oferuje następujące opcje:

- **Kestrel**: Niezależny od platformy serwer, odpowiedni, jeśli chcemy uruchamiać swój kod na dowolnej platformie.

- **WebListener**: Serwer dostępny tylko dla Windows, oferujący znacznie lepszą wydajność niż Kestrel, ale wymaga systemu Windows; począwszy od wersji ASP.NET Core 2 nazywa się teraz `HTTP.sys`.
- **IIS**: Tak jak dawniej, możemy nadal uruchamiać swoje aplikacje WWW w IIS w systemie Windows, korzystając ze *starych* narzędzi konfiguracyjnych.

Serwer w tym kontekście jest po prostu implementacją `IServer`, interfejsu zdefiniowanego w pakiecie NuGet `Microsoft.AspNetCore.Hosting`. Definiuje to bazowy kontrakt oferowany przez serwer, który można opisać następująco:

- Metoda `Start`, od której wszystko się zaczyna, jest odpowiedzialna za utworzenie `HttpContext`, ustawienie właściwości `Request` i `Response` oraz wywołanie konwencjonalnej metody `Configure`.
- Kolekcja `Features`, zawierająca funkcje obsługiwane przez daną implementację. Istnieją dziesiątki funkcji, ale serwer musi obsługiwać co najmniej `IHttpRequestFeature` oraz `IHttpResponseFeature`.

Każda z tych implementacji serwerów jest zapewniana przez pakiet NuGet:

Serwer	Pakiet
Kestrel	`Microsoft.AspNetCore.Server.Kestrel`
WebListener	`Microsoft.AspNetCore.Server.WebListener` (`Microsoft.AspNetCore.Server.HttpSys` od wersji ASP.NET Core 2)
IIS	`Microsoft.AspNetCore.Server.IISIntegration`

IIS nie może być wykorzystywany samodzielnie. IIS jest oczywiście rodzimą aplikacją Windows i dlatego nie jest dostępny poprzez NuGet, a pakiet `Microsoft.AspNetCore.Server.IISIntegration` zawiera moduł IIS ASP.NET Core Module, który trzeba zainstalować w IIS, żeby móc uruchamiać w nim aplikacje ASP.NET Core poprzez Kestrel (WebListener nie jest kompatybilny z IIS). Istnieją oczywiście inne implementacje serwerów dostarczane przez firmy trzecie (na przykład Nowin dostępny pod adresem *https://github.com/Bobris/Nowin)*.

Co trzeba więc wiedzieć, aby wybrać jeden z tych serwerów hostingowych?

Kestrel

Kestrel jest *domyślnym*, wieloplatformowym serwerem WWW. Oferuje niezłą wydajność, ale brakuje mu wielu funkcji, które często są potrzebne w rzeczywistych aplikacjach: