OKIEM EKSPERTA

C#12 i .NET 8 dla programistów aplikacji wieloplatformowych

Twórz aplikacje, witryny WWW oraz serwisy sieciowe za pomocą ASP.NET Core 8, Blazor i EF Core 8

Wydanie VIII

Mark J. Price

Tytuł oryginału: C# 12 and .NET 8 - Modern Cross-Platform Development Fundamentals, 8th Edition

Tłumaczenie: Wojciech Moch

ISBN: 978-83-289-1455-1

Copyright ©Packt Publishing 2023. First published in the English language under the title 'C# 12 and .NET 8 – Modern Cross-Platform Development Fundamentals - Eighth Edition – (9781837635870)'
Polish edition copyright © 2024 by Helion S.A.

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

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

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

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/c12n88
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/c12n88.zip

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

O autorze ... 21

O korektorze merytorycznym ... 22

Wstęp ... 23

ROZDZIAŁ 1
Cześć, C#! Witaj, .NET! ... 29

- Wprowadzenie do książki i jej zawartości ... 29
  - Pobieranie kodu przygotowanego dla tej książki ... 29
  - Pojęcia związane z .NET używane w tej książce ... 30
  - Struktura książki i używany w niej styl ... 31
  - Tematy omawiane w tej książce ... 31
- Konfigurowanie środowiska programistycznego ... 32
  - Wybieranie narzędzia i typu aplikacji właściwych do nauki ... 32
  - Instalowanie na wielu platformach ... 36
  - Pobieranie i instalowanie Visual Studio 2022 dla Windows ... 36
  - Pobieranie i instalowanie Microsoft Visual Studio Code ... 38
- Poznawanie .NET ... 41
  - Plany obsługi platformy .NET ... 41
  - Poznawanie języka IL ... 45
  - Porównanie technologii .NET ... 45
  - Zarządzanie wieloma projektami w edytorach kodu ... 46
- Tworzenie aplikacji konsoli za pomocą Visual Studio 2022 ... 46
  - Pisanie kodu za pomocą Visual Studio 2022 ... 46
  - Kompilowanie i uruchamianie kodu w Visual Studio ... 48
  - Programy najwyższego poziomu ... 50
  - Wymagania programów najwyższego poziomu ... 51
  - Ujawnianie przestrzeni nazwy klasy Program ... 53
  - Dodawanie drugiego projektu w Visual Studio 2022 ... 53
- Tworzenie aplikacji konsoli za pomocą Visual Studio Code ... 56
  - Pisanie kodu za pomocą Visual Studio Code ... 56
  - Kompilowanie i uruchamianie kodu za pomocą polecenia dotnet ... 58
  - Dodawanie drugiego projektu w Visual Studio Code ... 59
  - Podsumowanie kroków wykonanych w Visual Studio Code ... 61
  - Podsumowanie innych typów projektów używanych w tej książce ... 62

Wykorzystywanie repozytorium GitHuba w tej książce 62
Kod aplikacji tworzonych w tej książce 63
Pobieranie kodu rozwiązań z repozytorium GitHuba 63
Używanie systemu Git w Visual Studio Code 64
Gdzie znaleźć pomoc? 65
Przeglądanie dokumentacji Microsoftu 65
Linki do dokumentacji używane w tej książce 65
Uzyskiwanie pomocy dla narzędzia dotnet 65
Przeglądanie definicji typów i ich elementów 67
Konfigurowanie wskazówek w tekście 70
Poszukiwanie odpowiedzi na Stack Overflow 71
Poszukiwanie odpowiedzi za pomocą Google 72
Przeszukiwanie kodu źródłowego .NET 72
Subskrybowanie blogów 73
Filmy Scotta Hanselmana 73
Narzędzia AI, takie jak ChatGPT i GitHub Copilot 74
Wyłączanie narzędzi, gdy zaczynają przeszkadzać 76
Praktyka i ćwiczenia 77
Ćwiczenie 1.1 — sprawdź swoją wiedzę 77
Ćwiczenie 1.2 — ćwicz C# gdzie się da 78
Ćwiczenie 1.3 — dalsza lektura 78
Ćwiczenie 1.4 — używanie notatników Polyglot 78
Ćwiczenie 1.5 — motywy kolorystyczne w nowoczesnym .NET 79
Ćwiczenie 1.6 — darmowa nauka C# i certyfikacja 79
Ćwiczenie 1.7 — wersje alfa platformy .NET 79
Podsumowanie 80

ROZDZIAŁ 2
Mówimy w C# 81
Wprowadzenie do języka C# 81
Rozpoznawanie wersji oraz funkcji języka 81
Standardy języka C# 82
Odczytywanie wersji używanego kompilatora C# 83
Wyświetlanie numeru wersji kompilatora 86
Poznawanie gramatyki i słownictwa języka C# 87
Gramatyka języka C# 87
Instrukcje 88
Komentarze 88
Bloki 89
Przykłady instrukcji i bloków 90
Słownictwo języka C# 92

Porównanie języków programowania do języków ludzkich 93
Zmiana schematu kolorów składni języka C# 93
Pomoc przy pisaniu kodu 94
Importowanie przestrzeni nazw 95
Niejawne i globalne importowanie przestrzeni nazw 95
Czasowniki jako metody 99
Rzeczowniki to typy, pola i zmienne 99
Ujawnienie wielkości słownika języka C# 100
Przykład prośby do ChatGPT o wyjaśnienie kodu 102
Praca ze zmiennymi 103
Nazywanie zmiennych i przypisanie wartości 104
Literały 105
Przechowywanie tekstu 105
Przechowywanie liczb 108
Przechowywanie liczb rzeczywistych 110
Przechowywanie wartości logicznych 115
Zapisywanie obiektów dowolnego typu 115
Przechowywanie typów dynamicznych 117
Deklarowanie zmiennych lokalnych 118
Odczytywanie i ustalanie domyślnych wartości typów 121
Dokładniejsze poznawanie aplikacji konsoli 122
Wyświetlanie informacji dla użytkownika 122
Pobieranie danych od użytkownika 128
Uproszczone korzystanie z konsoli 129
Odczytywanie naciśnięć klawiszy 130
Odczytywanie parametrów aplikacji konsoli 131
Ustalanie opcji za pomocą argumentów 135
Obsługiwanie platform nieobsługujących wybranych API 137
Instrukcje async i await 138
Poprawianie reakcji aplikacji konsoli 138
Praktyka i ćwiczenia 140
Ćwiczenie 2.1 — sprawdź swoją wiedzę 140
Ćwiczenie 2.2 — sprawdź swoją wiedzę o typach liczbowych 140
Ćwiczenie 2.3 — poznaj wielkości i zakresy liczb 141
Ćwiczenie 2.4 — dalsza lektura 141
Ćwiczenie 2.5 — Pakiet Spectre 142
Podsumowanie 142

Kup ksi k Pole ksi k

ROZDZIAŁ 3

Sterowanie przepływem, konwertowanie typów i obsługa wyjątków 143
Działania na zmiennych 143
Operatory dwuargumentowe 143
Operatory jednoargumentowe 144
Operatory trójargumentowe 144
Objaśnienie operatorów jednoargumentowych 145
Objaśnianie dwuargumentowych operatorów arytmetycznych 146
Operatory przypisania 147
Operatory pomijania wartości null 148
Operatory logiczne 148
Warunkowe operatory logiczne 149
Operatory bitowe i operatory przesunięć 150
Operatory różne 152
Instrukcje wyboru 153
Instrukcja if 153
Dopasowywanie wzorców z instrukcją if 154
Instrukcja switch 155
Dodawanie nowego elementu do projektu w Visual Studio 2022 157
Dopasowywanie wzorców z instrukcją switch 158
Upraszczanie instrukcji switch za pomocą wyrażeń switch 160
Instrukcje iteracji 161
Instrukcja while 161
Instrukcja do 162
Instrukcja for 163
Instrukcja foreach 163
Zapisywanie wielu wartości w tablicy 164
Praca z tablicami jednowymiarowymi 164
Praca z tablicami wielowymiarowymi 166
Praca z tablicami poszarpanymi 167
Dopasowywanie wzorców list w tablicach 169
Tablice inline 171
Podsumowanie tablic 172
Rzutowanie i konwertowanie między typami 172
Jawne i niejawne rzutowanie liczb 173
Zapisywanie liczb ujemnych w postaci binarnej 174
Używanie typu System.Convert 175
Zaokrąglanie liczb 176
Kontrolowanie sposobu zaokrąglania 178
Konwersja z dowolnego typu na ciąg znaków 178
Konwertowanie obiektu binarnego na ciąg znaków 179

Parsowanie ciągów znaków z liczbami, datami i czasem 180
Konwencja nazywania metod ze słowem Try 182
Obsługa wyjątków 183
Instrukcja try 183
Wykrywanie przepełnień 187
Instrukcja checked 187
Instrukcja unchecked 189
Praktyka i ćwiczenia 190
Ćwiczenie 3.1 — sprawdź swoją wiedzę 190
Ćwiczenie 3.2 — pętle i przepełnienia 190
Ćwiczenie 3.3 — sprawdź swoją wiedzę o operatorach 191
Ćwiczenie 3.4 — pętle i operatory 191
Ćwiczenie 3.5 — obsługa wyjątków 192
Ćwiczenie 3.6 — poznaj notatniki C# 192
Ćwiczenie 3.7 — dalsza lektura 193
Podsumowanie 193

ROZDZIAŁ 4
Pisanie, debugowanie i testowanie funkcji 194
Tworzenie funkcji 194
Programy i funkcje najwyższego poziomu 194
Co jest automatycznie generowane dla funkcji lokalnej? 195
Definiowanie częściowej klasy Program z funkcjami statycznymi 196
Co jest generowane automatycznie dla funkcji statycznej? 197
Przykład z tabliczką mnożenia 198
Dygresja na temat argumentów i parametrów 201
Pisanie funkcji zwracającej wartość 202
Rekurencyjne obliczanie silni 206
Dokumentowanie funkcji za pomocą komentarzy XML 209
Używanie wyrażeń lambda w implementacji funkcji 210
Debugowanie tworzonego programu 213
Tworzenie aplikacji z celowym błędem 213
Tworzenie punktu przerwania 213
Pasek narzędzi debugowania 216
Okna debugowania 217
Krokowe wykonywanie kodu 218
Używanie zintegrowanego terminala Visual Studio Code podczas debugowania 219
Dostosowywanie punktów przerwania 222
Przeładowywanie na gorąco w trakcie programowania 223
Przeładowywanie na gorąco w Visual Studio 2022 224

Kup ksi k Pole ksi k

Przeładowywanie na gorąco w Visual Studio Code i w wierszu poleceń ... 225
Protokołowanie ... 226
Dostępne opcje protokołowania ... 226
Wykorzystywanie typów Debug i Trace ... 227
Konfigurowanie obiektów nasłuchujących ... 228
Przełączanie poziomów śledzenia ... 230
Protokołowanie informacji o kodzie źródłowym ... 236
Testy jednostkowe ... 238
Różne rodzaje testów ... 239
Tworzenie biblioteki klas wymagającej testowania ... 239
Tworzenie testów jednostkowych ... 241
Rzucanie i wychwytywanie wyjątków w funkcjach ... 244
Rozróżnienie błędów użycia i błędów wykonania ... 244
Wyjątki często rzucane w funkcjach ... 245
Rzucanie wyjątków za pomocą klauzul ochronnych ... 245
Czym jest stos wywołań? ... 246
Gdzie należy wychwytywać wyjątki? ... 249
Ponowne rzucanie wyjątku ... 249
Implementowanie wzorca tester-wykonawca i wzorca próby ... 251
Praktyka i ćwiczenia ... 253
Ćwiczenie 4.1 — sprawdź swoją wiedzę ... 253
Ćwiczenie 4.2 — tworzenie funkcji z wykorzystaniem debugowania i testów jednostkowych ... 253
Ćwiczenie 4.3 — dalsza lektura ... 254
Podsumowanie ... 254

ROZDZIAŁ 5
Tworzenie własnych typów w programowaniu obiektowym ... 255

Programowanie obiektowe ... 255
Tworzenie bibliotek klas ... 257
Tworzenie biblioteki klas ... 257
Przestrzenie nazw o zasięgu całego pliku ... 259
Definiowanie klasy w przestrzeni nazw ... 259
Modyfikatory dostępu ... 260
Elementy klasy ... 261
Importowanie przestrzeni nazw, aby użyć wybranego typu ... 262
Tworzenie obiektów ... 263
Przechowywanie danych w polach ... 266
Definiowanie pól ... 266
Typy pól ... 267
Modyfikatory dostępu ... 267

Ustalanie i wypisywanie wartości pól 268
Ustalanie wartości pól za pomocą składni inicjalizacji obiektu 269
Zapisywanie wartości za pomocą słowa kluczowego enum 270
Przechowywanie wielu wartości w typie enum 271
Zapisywanie wielu wartości za pomocą kolekcji 272
Kolekcje generyczne 273
Tworzenie pól statycznych 274
Tworzenie stałych pól 275
Tworzenie pól tylko do odczytu 276
Wymaganie podania wartości właściwości przy tworzeniu obiektu 277
Inicjalizowanie pól w konstruktorach 278
Praca z metodami i krotkami 281
Zwracanie wartości z metody 281
Sterowanie przekazywaniem parametrów 282
Przeciążanie metod 283
Parametry opcjonalne i nazywane 283
Mieszanie parametrów opcjonalnych i wymaganych 285
Sposoby przekazywania parametrów 286
Zwracanie wartości ze słowem kluczowym ref 288
Łączenie wielu wartości za pomocą krotki 288
Implementowanie funkcji lokalnych 292
Dzielenie klas na części 293
Kontrola dostępu za pomocą właściwości i indekserów 294
Definiowanie właściwości tylko do odczytu 294
Definiowanie właściwości z możliwością przypisania 295
Ograniczanie wartości typów wyliczeniowych 298
Definiowanie indekserów 299
Dopasowywanie wzorców z obiektami 301
Definiowanie listy pasażerów 301
Rozszerzenia dopasowywania wzorców w C# 9 i nowszych 303
Praca z rekordami 304
Właściwości wyłącznie inicjalizowane 304
Rekordy 305
Równość typów rekordowych 306
Pozycyjne elementy danych w rekordach 307
Definiowanie podstawowego konstruktora klasy 308
Praktyka i ćwiczenia 310
Ćwiczenie 5.1 — sprawdź swoją wiedzę 310
Ćwiczenie 5.2 — modyfikatory dostępu 310
Ćwiczenie 5.3 — dalsza lektura 311
Podsumowanie 311

Kup ksi k Pole ksi k

ROZDZIAŁ 6

Implementowanie interfejsów i dziedziczenie klas 312
Konfigurowanie biblioteki klas i aplikacji konsoli 312
Metody statyczne i przeciążanie operatorów 314
Implementowanie działań w metodzie 315
Implementowanie działań za pomocą operatora 319
Wykorzystywanie typów generycznych 321
Praca z typami niegenerycznymi 322
Praca z typami generycznymi 323
Wywoływanie i obsługa zdarzeń 324
Wywoływanie metod za pomocą delegatów 325
Przykłady używania delegatów 326
Status: to skomplikowane! 326
Definiowanie i obsługa delegatów 326
Definiowanie i obsługa zdarzeń 329
Implementowanie interfejsów 330
Typowe interfejsy 330
Porównywanie obiektów podczas sortowania 331
Porównywanie obiektów za pomocą osobnej klasy 335
Jawne i niejawne implementowanie interfejsów 336
Definiowanie interfejsów z domyślnymi implementacjami 337
Zarządzanie pamięcią za pomocą typów referencyjnych i typów wartości 340
Pamięć stosu i sterty 340
Definiowanie typów referencyjnych i typów wartości 340
Sposób przechowywania w pamięci typów referencyjnych i typów wartości 341
Boxing 343
Równość typów 344
Definiowanie typu kategorii struct 346
Praca z typami record struct 347
Zwalnianie niezarządzanych zasobów 348
Wymuszanie wywołania metody Dispose 350
Praca z wartościami null 351
Przekształcanie typu wartości w typ nullowalny 351
Inicjalizowanie typów nullowalnych 353
Poznawanie nullowalnych typów referencyjnych 353
Sterowanie funkcją ostrzeżeń dla typów nullowalnych 354
Wyłączanie innych ostrzeżeń kompilatora 355
Deklarowanie nienullowalnych zmiennych i parametrów 356
Sprawdzanie wartości null 358
Kontrolowanie wartości null w parametrach metod 359

Kup ksi k Pole ksi k

Dziedziczenie klas ... 361
Rozbudowywanie klasy ... 361
Ukrywanie elementów ... 362
Słowa kluczowe this i base ... 363
Pokrywanie elementów klasy ... 363
Dziedziczenie po klasach abstrakcyjnych ... 364
Wybieranie między interfejsem a klasą abstrakcyjną ... 365
Blokowanie dziedziczenia i pokrywania ... 366
Polimorfizm ... 366
Rzutowanie w ramach hierarchii dziedziczenia ... 368
Rzutowanie niejawne ... 368
Rzutowanie jawne ... 368
Obsługa wyjątków rzutowania ... 369
Dziedziczenie i rozbudowywanie typów .NET ... 370
Dziedziczenie po wyjątku ... 370
Rozszerzanie typów, po których nie można dziedziczyć ... 372
Możliwości tworzenia własnych typów ... 374
Kategorie własnych typów i ich możliwości ... 375
Zmienność i rekordy ... 375
Dziedziczenie i implementowanie ... 377
Przeglądanie przykładowego kodu ... 377
Praktyka i ćwiczenia ... 379
Ćwiczenie 6.1 — sprawdź swoją wiedzę ... 379
Ćwiczenie 6.2 — tworzenie hierarchii dziedziczenia ... 380
Ćwiczenie 6.3 — pisanie lepszego kodu ... 380
Ćwiczenie 6.4 — dalsza lektura ... 381
Podsumowanie ... 381

ROZDZIAŁ 7

Poznawanie typów .NET ... 382
Wprowadzenie do .NET 8 ... 382
Sprawdzanie dostępności aktualizacji .NET SDK ... 383
Zestawy i przestrzenie nazw ... 384
Zestawy, pakiety i przestrzenie nazw ... 384
Poznawanie pakietów SDK dla projektów .NET ... 385
Przestrzenie nazw i typy w zestawach ... 386
Pakiety NuGet ... 386
Czym są frameworki? ... 387
Importowanie przestrzeni nazw w celu użycia typu ... 388
Związki słów kluczowych języka C# z typami .NET ... 389

Kup ksi k Pole ksi k

Wieloplatformowe współdzielenie kodu z bibliotekami klas .NET Standard 392
Domyślne ustawienia bibliotek klas w różnych wersjach SDK 393
Tworzenie biblioteki klas .NET Standard 394
Kontrolowanie wersji .NET SDK 394
Mieszanie pakietów SDK i docelowych frameworków 396
Publikowanie własnych aplikacji 397
Tworzenie aplikacji konsoli do publikacji 398
Poznawanie polecenia dotnet 399
Pobieranie informacji na temat platformy .NET i jej środowiska 400
Zarządzanie projektami 401
Publikowanie samodzielnej aplikacji 402
Publikowanie aplikacji jednoplikowej 404
Zmniejszanie wielkości aplikacji 405
Kontrolowanie miejsca tworzenia artefaktów 406
Kompilacja native AOT 407
Dekompilowanie zestawów 411
Dekompilowanie za pomocą rozszerzenia ILSpy w Visual Studio 2022 412
Przeglądanie oryginalnych źródeł w Visual Studio 2022 415
Nie, nie można zablokować możliwości dekompilowania 416
Przygotowywanie własnych pakietów NuGet 418
Dodawanie odwołania do pakietu 418
Tworzenie pakietu dla NuGet 419
Przeszukiwanie pakietów NuGet 424
Testowanie pakietu 425
Praca z proponowanymi funkcjami 425
Wymaganie proponowanych funkcji 427
Włączanie proponowanych funkcji 427
Interceptory metod 427
Praktyka i ćwiczenia 428
Ćwiczenie 7.1 — sprawdź swoją wiedzę 428
Ćwiczenie 7.2 — dalsza lektura 428
Ćwiczenie 7.3 — przenoszenie kodu z .NET Framework do nowoczesnego .NET 428
Ćwiczenie 7.4 — tworzenie generatorów kodu źródłowego 429
Ćwiczenie 7.5 — PowerShell 429
Ćwiczenie 7.6 — poprawianie wydajności w .NET 429
Podsumowanie 430

Kup ksi k Pole ksi k

ROZDZIAŁ 8

Używanie typów .NET 431
Praca z liczbami 431
Praca z wielkimi liczbami całkowitymi 432
Praca z liczbami zespolonymi 433
Generowanie liczb losowych na potrzeby gier i podobnych aplikacji 434
Generowanie identyfikatorów GUID 436
Praca z tekstem 437
Odczytywanie długości ciągu znaków 437
Odczytywanie znaków z ciągu 438
Dzielenie ciągu znaków 438
Pobieranie części ciągu znaków 439
Poszukiwanie tekstu w ciągu 439
Porównywanie ciągów znaków 440
Inne elementy klasy string 442
Wydajne tworzenie ciągów znaków 443
Dopasowywanie wzorców za pomocą wyrażeń regularnych 444
Kontrolowanie cyfr wprowadzonych jako tekst 444
Poprawianie wydajności wyrażeń regularnych 445
Składnia wyrażenia regularnego 446
Przykłady wyrażeń regularnych 446
Dzielenie złożonych ciągów znaków rozdzielanych przecinkami 447
Włączanie kolorowania składni wyrażeń regularnych 449
Poprawianie wydajności wyrażeń regularnych za pomocą generatorów kodu 452
Praca z kolekcjami 454
Wspólne funkcje wszystkich kolekcji 454
Sortowanie kolekcji 465
Używanie specjalizowanych kolekcji 466
Kolekcje tylko do odczytu, niezmienne i zamrożone 466
Inicjalizowanie kolekcji za pomocą wyrażeń kolekcji 471
Dobre praktyki w pracy z kolekcjami 471
Praca z typem Span, indeksami i zakresami 472
Wydajne korzystanie z pamięci za pomocą typu Span 472
Określanie pozycji za pomocą typu Index 473
Definiowanie zakresów za pomocą typu Range 473
Używanie indeksów i zakresów 474
Praktyka i ćwiczenia 475
Ćwiczenie 8.1 — sprawdź swoją wiedzę 475
Ćwiczenie 8.2 — wyrażenia regularne 475
Ćwiczenie 8.3 — metody rozszerzające 476

Kup ksi k Pole ksi k

Ćwiczenie 8.4 — praca z zasobami sieciowymi 476
Ćwiczenie 8.5 — dalsza lektura 476
Podsumowanie 476

ROZDZIAŁ 9
Hardening i bezpieczeństwo 477

Program antywirusowy Microsoft Defender 478
Instalacja programu antywirusowego Microsoft Defender 479
Wykorzystanie interfejsu użytkownika 479
Wyłączanie programu antywirusowego Microsoft Defender 481
Czym w ogóle jest ATP? 482
Windows Defender ATP Exploit Guard 483
Zapora systemu Windows Defender — bez żartów 485
Trzy konsole administracyjne zapory systemu Windows 485
Trzy różne profile zapory 489
Tworzenie w zaporze nowej reguły przychodzącej 491
Tworzenie reguły zezwalającej na wysyłanie pingów (ICMP) 493
Zarządzanie zaporą WFAS przy użyciu zasad grupy 495
Technologie szyfrowania 498
BitLocker i wirtualny układ TPM 499
Chronione maszyny wirtualne 500
Szyfrowane sieci wirtualne 500
Encrypting File System 501
Protokoły IPsec 501
Azure AD Password Protection 506
Szczegółowe zasady dotyczące haseł 506
Zaawansowana analiza zagrożeń — koniec wsparcia 510
Czym jest (była) ATA? 510
Microsoft Defender for Identity 512
Najważniejsze wskazówki dotyczące ogólnego bezpieczeństwa 513
Pozbycie się wiecznych administratorów 513
Korzystanie z odrębnych kont w celu uzyskania dostępu administracyjnego 514
Używanie innego komputera do wykonywania zadań administracyjnych 514
Nigdy nie przeglądaj internetu, będąc zalogowanym na serwerze 515
Kontrola dostępu oparta na rolach 515
Just Enough Administration 516
Zmiana portu 3389 połączenia pulpitu zdalnego 517
Natychmiast wyłącz zewnętrzne połączenia pulpitu zdalnego 519
Wyłącz niebezpieczne protokoły szyfrowania 520
Podsumowanie 521
Pytania 523

ROZDZIAŁ 10
Praca z bazami danych przy użyciu Entity Framework Core 524

Nowoczesne bazy danych 524
Czym jest Entity Framework? 525
Entity Framework Core 526
Co znaczy „najpierw baza danych” i „najpierw kod”? 526
Usprawnienia wydajności w EF Core 527
Używanie przykładowej relacyjnej bazy danych 527
Używanie SQLite 528
Konfigurowanie EF Core w projekcie .NET 530
Tworzenie aplikacji konsoli do pracy z EF Core 530
Tworzenie przykładowej bazy danych Northwind na serwerze SQLite 531
Zarządzanie przykładową bazą danych Northwind
za pomocą SQLiteStudio 532
Używanie lekkiego dostawcy ADO.NET dla SQLite 534
Wybieranie dostawcy danych EF Core 535
Łączenie z bazą danych 536
Definiowanie klasy kontekstu bazy danych Northwind 536
Definiowanie modeli EF Core 537
Konwencje w EF Core 538
Atrybuty EF Core 538
Płynne API EF Core 540
Tworzenie modelu w EF Core 541
Dodawanie tabel do klasy kontekstu bazy danych Northwind 544
Konfigurowanie narzędzia dotnet-ef 545
Tworzenie modeli na podstawie istniejącej bazy danych 546
Dostosowywanie szablonów wstecznej inżynierii 552
Konfigurowanie konwencji 552
Zapytania do modelu EF Core 553
Filtrowanie dołączanych encji 555
Filtrowanie i sortowanie produktów 557
Pobieranie generowanych instrukcji SQL 558
Protokołowanie w EF Core 559
Pobieranie pojedynczej encji 562
Dopasowywanie wzorców za pomocą instrukcji Like 564
Generowanie liczb losowych w zapytaniach 566
Definiowanie globalnych filtrów 567
Wzorce ładowania i śledzenia w EF Core 567
Chętne ładowanie encji za pomocą metody rozszerzającej Include 568
Włączenie leniwego ładowania 568
Jawne ładowanie encji za pomocą metody Load 569
Kontrolowanie śledzenia encji 572

Kup ksi k Pole ksi k

Manipulowanie danymi w EF Core 576
Wstawianie encji 577
Aktualizowanie encji 579
Usuwanie encji 581
Wydajniejsze aktualizowanie i usuwanie 582
Grupowanie kontekstów baz danych 585
Praktyka i ćwiczenia 586
Ćwiczenie 10.1 — sprawdź swoją wiedzę 586
Ćwiczenie 10.2 — eksportowanie danych z wykorzystaniem różnych formatów serializacji 586
Ćwiczenie 10.3 — praca z transakcjami 586
Ćwiczenie 10.4 — modele Code First w EF Code 587
Ćwiczenie 10.5 — sekrety aplikacji 587
Ćwiczenie 10.6 — dalsza lektura 587
Ćwiczenie 10.7 — poznawanie baz danych NoSQL 587
Podsumowanie 588

ROZDZIAŁ 11
Odczytywanie danych i manipulowanie nimi za pomocą LINQ 589

Tworzenie wyrażeń LINQ 589
Porównanie imperatywnych i deklaratywnych funkcji języka 589
Z czego składa się LINQ? 590
Rozbudowa sekwencji za pomocą klas wyliczeniowych 591
LINQ w praktyce 594
Czym jest opóźnione wykonanie? 594
Filtrowanie encji za pomocą metody Where 596
Korzystanie z metody nazwanej 598
Upraszczanie kodu przez usunięcie jawnego tworzenia delegata 599
Korzystanie z wyrażenia lambda 599
Sortowanie encji 600
Sortowanie według elementów 601
Deklarowanie zapytania za pomocą słowa kluczowego var lub określonego typu 601
Filtrowanie według typu 602
Praca ze zbiorami 604
Używanie LINQ z EF Core 606
Tworzenie aplikacji konsoli do nauki LINQ dla Encji 606
Tworzenie modelu danych EF Core 607
Filtrowanie i sortowanie sekwencji 610
Projekcje sekwencji na nowe typy 612
Łączenie i grupowanie 614

Grupowanie wyszukiwań 617
Agregowanie i stronicowanie sekwencji 620
Sprawdzanie, czy sekwencja nie jest pusta 621
Uważaj na właściwość Count! 622
Stronicowanie z LINQ 624
Upiększanie składni 627
Praktyka i ćwiczenia 629
Ćwiczenie 11.1 — sprawdź swoją wiedzę 629
Ćwiczenie 11.2 — zapytania LINQ 629
Ćwiczenie 11.3 — używanie wielu wątków w zapytaniach LINQ 630
Ćwiczenie 11.4 — praca z LINQ to XML 630
Ćwiczenie 11.5 — tworzenie własnych metod rozszerzających LINQ 630
Ćwiczenie 11.6 — dalsza lektura 630
Podsumowanie 630

ROZDZIAŁ 12
Wprowadzenie do aplikacji sieciowych w ASP.NET Core 632
Czym jest ASP.NET Core? 632
Klasyczna ASP.NET kontra ASP.NET Core 634
Tworzenie stron WWW za pomocą ASP.NET Core 634
Tworzenie serwisów sieciowych 637
Struktury projektów 638
Struktura projektów w rozwiązaniu 638
Tworzenie modelu encji używanego w tej książce 640
Tworzenie bazy danych Northwind 640
Tworzenie biblioteki klas dla modelu encji bazy SQLite 641
Dostosowanie modelu i definiowanie metod rozszerzających 645
Rejestrowanie zakresu zależnego serwisu 647
Tworzenie biblioteki klas modelu encji dla SQL Server 648
Testowanie bibliotek klas 651
Tworzenie w sieci WWW 653
Protokół http 653
Używanie Google Chrome do wykonywania żądań http 655
Tworzenie oprogramowania dla sieci WWW po stronie klienta 657
Praktyka i ćwiczenia 658
Ćwiczenie 12.1 — sprawdź swoją wiedzę 658
Ćwiczenie 12.2 — znasz te skrótowce? 658
Ćwiczenie 12.3 — dalsza lektura 659
Podsumowanie 659

Kup ksi k Pole ksi k

ROZDZIAŁ 13
Tworzenie witryn WWW przy użyciu ASP.NET Core Razor Pages 660

ASP.NET Core 660
Tworzenie pustego projektu ASP.NET Core 660
Testowanie i zabezpieczanie witryny 664
Kontrola środowiska hostingowego 669
Włączanie plików statycznych 670
Żądania w przeglądarce podczas tworzenia aplikacji 672
Technologia Razor Pages 673
Włączanie technologii Razor Pages 673
Definiowanie strony Razor 674
Używanie wspólnego układu w wielu stronach Razor 676
Tymczasowe przechowywanie danych 678
Używanie plików code-behind w stronach Razor 680
Konfigurowanie plików dołączanych do projektu ASP.NET Core 683
Operacje kompilowania pliku projektu 684
Używanie Entity Framework Core z ASP.NET Core 685
Konfigurowanie Entity Framework Core jako serwisu 685
Manipulowanie danymi na stronach Razor 688
Wstrzykiwanie zależnego serwisu na stronę Razor 689
Konfigurowanie serwisów i potoku obsługi żądań HTTP 690
Routowanie punktów końcowych 690
Konfigurowanie routowania punktów końcowych 691
Przeglądanie konfiguracji routowania punktów końcowych w naszym projekcie 691
Przygotowywanie potoku obsługi żądań HTTP 693
Podsumowanie najważniejszych metod rozszerzających oprogramowania pośredniczącego 694
Wizualizacja potoku HTTP 695
Implementowanie oprogramowania pośredniczącego jako anonimowego delegata 696
Praktyka i ćwiczenia 698
Ćwiczenie 13.1 — sprawdź swoją wiedzę 698
Ćwiczenie 13.2 — używanie bibliotek klas Razor 698
Ćwiczenie 13.3 — włączenie HTTP/3 i obsługa dekompresji żądań 698
Ćwiczenie 13.4 — tworzenie witryny obsługującej dane 699
Ćwiczenie 13.5 — zastępowanie aplikacji konsoli stronami WWW 699
Ćwiczenie 13.6 — wprowadzenie do biblioteki Bootstrap 700
Ćwiczenie 13.7 — dalsza lektura 700
Ćwiczenie 13.8 — tworzenie witryn WWW za pomocą wzorca Model-View-Controller 700
Podsumowanie 700

Kup ksi k Pole ksi k

ROZDZIAŁ 14

Tworzenie i używanie serwisów sieciowych 701
Tworzenie serwisów w technologii ASP.NET Core Web API 701
Skróty stosowane w serwisach sieciowych 701
Żądania i odpowiedzi HTTP w Web API 702
Tworzenie projektu ASP.NET Core Web API 705
Sprawdzanie funkcji serwisu sieciowego 709
Tworzenie serwisu internetowego dla bazy danych Northwind 710
Rejestrowanie serwisów zależnych 712
Tworzenie repozytorium danych dla encji 713
Routowanie w serwisach sieciowych 716
Konfigurowanie repozytorium klientów i kontrolera Web API 720
Podawanie szczegółów problemu 724
Kontrola nad serializacją XML 725
Dokumentowanie i testowanie serwisów 725
Testowanie żądań GET za pomocą przeglądarki 726
Testowanie żądań HTTP za pomocą narzędzi HTTP/REST 727
Włączanie narzędzia Swagger 731
Testowanie żądań w narzędziu SwaggerUI 732
Włączanie protokołowania HTTP 735
Obsługa protokołowania dodatkowych nagłówków żądań
w systemie W3CLogger 738
Używanie serwisu za pomocą klientów HTTP 738
Klasa HttpClient 738
Konfigurowanie klientów HTTP za pomocą klasy HttpClientFactory 739
Pobieranie w kontrolerze listy klientów w formacie JSON 739
Uruchamianie wielu projektów 742
Uruchamianie projektów serwisu sieciowego i klienta MVC 744
Praktyka i ćwiczenia 745
Ćwiczenie 14.1 — sprawdź swoją wiedzę 745
Ćwiczenie 14.2 — ćwiczenia w tworzeniu i usuwaniu klientów
za pomocą HttpClient 745
Ćwiczenie 14.3 — implementowanie zaawansowanych funkcji
serwisów sieciowych 746
Ćwiczenie 14.4 — tworzenie serwisów sieciowych
za pomocą minimalnego API 746
Ćwiczenie 14.5 — dalsza lektura 746
Podsumowanie 746

ROZDZIAŁ 15

Tworzenie interfejsów użytkownika w technologii Blazor 747
Technologia Blazor 747
JavaScript i podobne 747
Silverlight — C# i .NET w formie wtyczki 748
WebAssembly — podstawa technologii Blazor 748
Różne modele hostowania komponentów Blazora w .NET 7 i starszych 749
Unifikacja modeli hostowania Blazor w .NET 8 749
Omówienie komponentów tworzonych za pomocą Blazora 750
Czym różnią się Blazor i Razor? 751
Przeglądanie szablonu projektu Blazor Web App 752
Tworzenie projektu Blazor Web App 752
Routing, układy i nawigacja w aplikacji Blazor 754
Klasy komponentów bazowych 760
Uruchamianie szablonu projektu Blazor Web App 761
Tworzenie komponentów Blazor 762
Definiowanie i testowanie prostego komponentu 762
Używanie ikon Bootstrapa 763
Przekształcanie komponentu w routowalny komponent stronicowy 764
Dodawanie encji do komponentu 765
Tworzenie abstrakcji serwisu dla komponentu Blazora 765
Definiowanie formularzy za pomocą komponentu EditForm 771
Tworzenie i używanie komponentu formularza danych klienta 772
Tworzenie komponentów do tworzenia, edytowania i usuwania klientów 773
Włączanie interakcji po stronie serwera 775
Testowanie komponentu formularza danych klienta 776
Włączanie wykonywania po stronie klienta z wykorzystaniem WebAssembly ... 777
Praktyka i ćwiczenia 777
Ćwiczenie 15.1 — sprawdź swoją wiedzę 777
Ćwiczenie 15.2 — przygotowanie komponentu tabliczki mnożenia 778
Ćwiczenie 15.3 — przygotowanie elementu nawigowania według krajów 778
Ćwiczenie 15.4 — rozbudowywanie aplikacji Blazora 779
Ćwiczenie 15.5 — używanie otwartych bibliotek komponentów Blazora 779
Ćwiczenie 15.6 — dalsza lektura 779
Podsumowanie 779

Epilog 780

Kup ksi k Pole ksi k

Kup książkę

Poleć książkę

Rozdział 12 Wprowadzenie do aplikacji sieciowych w ASP.NET Core

Trzecia część tej książki została poświęcona tworzeniu aplikacji sieciowych za pomocą ASP.NET Core. Dowiesz się, jak tworzyć pełne wieloplatformowe aplikacje, takie jak witryny WWW albo serwisy internetowe, a także aplikacje dla komputerów stacjonarnych i dla urządzeń mobilnych.

Firma Microsoft nazywa platformy przeznaczone do budowania aplikacji **modelami aplikacji** (ang. *app models*).

Zalecam sekwencyjne przejście przez ten i kolejne rozdziały, ponieważ w dalszych rozdziałach będziemy korzystać z projektów przygotowanych we wcześniejszych rozdziałach. Poza tym podczas lektury kolejnych rozdziałów zgromadzisz informacje i umiejętności ułatwiające radzenie sobie z bardziej złożonymi problemami, które pojawią się w późniejszych rozdziałach.

W tym rozdziale omawiam następujące zagadnienia:

- ASP.NET Core;
- nowe funkcje w ASP.NET Core;
- struktury projektów;
- tworzenie modelu encji dla kolejnych rozdziałów;
- sposoby tworzenia aplikacji sieciowych.

Czym jest ASP.NET Core?

Ta książka jest poświęcona językowi C# i platformie .NET, dlatego przedstawię tutaj różne modele aplikacji, których można użyć do tworzenia praktycznych aplikacji, z jakimi zetkniemy się w pozostałych rozdziałach.

Uwaga

Firma Microsoft udostępnia rozbudowany przewodnik implementowania różnych modeli aplikacji. Jest to część dokumentacji .NET Application Architecture Guidance, dostępnej pod adresem: *https://dotnet.microsoft.com/en-us/learn/dotnet/architecture-guides*.

Kup książkę Poleć książkę

Microsoft ASP.NET Core jest częścią historii tworzonych przez Microsoft technologii, przeznaczonych do projektowania aplikacji i serwisów WWW, które ewoluowały przez całe lata:

- **Active Server Pages** (**ASP**) powstała już w 1996 r. jako pierwsza próba Microsoftu stworzenia platformy pozwalającej na dynamiczne wykonywanie kodu aplikacji WWW po stronie serwera. Pliki ASP tworzone były w języku VBScript.
- **ASP.NET Web Forms** została wydana w 2002 r. jako część .NET Framework. Biblioteka ta miała pozwolić programistom niezwiązanym z siecią WWW, ale np. znającym język Visual Basic, na szybkie tworzenie aplikacji WWW przez przeciąganie i upuszczanie komponentów graficznych i dopisywanie do nich kodu reagującego na zdarzenia, wykorzystującego język Visual Basic lub C#. W nowych projektach należy unikać stosowania tej technologii i wybierać raczej ASP.NET MVC.
- **Windows Communication Foundation** (**WCF**) została wydana w 2006 r. Umożliwia programistom tworzenie serwisów SOAP oraz REST. Technologia SOAP ma bardzo duże możliwości, ale jest też bardzo skomplikowana, dlatego należy unikać jej stosowania, chyba że naprawdę potrzebne są oferowane przez nią funkcje, takie jak transakcje rozproszone albo złożone topologie wysyłania wiadomości.
- **ASP.NET MVC** powstała w 2009 r. i została zaprojektowana tak, żeby tworzyć jasny podział pomiędzy zadaniami dla programistów. I tak: **model** przechowuje tymczasowo dane, **widok** prezentuje je, stosując różne formaty w ramach interfejsu użytkownika, natomiast **kontroler** pobiera dane z modelu i przekazuje je do widoku. Takie rozdzielenie zadań umożliwia skuteczniejsze ponowne wykorzystywanie kodu oraz ułatwia tworzenie testów jednostkowych.
- **ASP.NET Web API** została wydana w 2012 r. Pozwala programistom na tworzenie serwisów HTTP lub REST, które są znacznie prostsze i pozwalają się lepiej skalować niż serwisy SOAP.
- **ASP.NET SignalR** powstała w 2013 r. Umożliwia komunikację w czasie rzeczywistym w ramach aplikacji WWW, ukrywając wszystkie niższe technologie i techniki, takie jak *Web Sockets* lub *Long Polling*. Pozwala to implementować na stronach WWW takie funkcje jak komunikacja na żywo, aktualizowanie na bieżąco ważnych danych, np. cen akcji. To wszystko działa w wielu różnych przeglądarkach, nawet jeżeli te nie obsługują nowoczesnych technologii, takich jak WebSocket.
- **ASP.NET Core** została wydana w 2016 r. i łączy nowoczesne implementacje starszych technologii pochodzących z .NET Framework, takich jak MVC, Web API i SignalR, z nowszymi technologiami, takimi jak Razor Pages, gRPC lub Blazor, dzięki czemu wszystkie one mogą działać w ramach nowoczesnego .NET. W ten sposób ta technologia może być stosowana niezależnie od platformy. W ASP.NET Core istnieje wiele szablonów pozwalających na rozpoczęcie pracy z wieloma obsługiwanymi technologiami.

Kup ksi k Pole ksi k

Wskazówka

Dobra praktyka: Do tworzenia aplikacji i serwisów WWW wybieraj bibliotekę ASP.NET Core, ponieważ zawiera ona wiele technologii związanych z siecią WWW, jest nowoczesna i wieloplatformowa.

Klasyczna ASP.NET kontra ASP.NET Core

Jak dotąd cała biblioteka tworzona była jako jeden zestaw w ramach .NET Framework — *System.Web.dll*. Jest on ściśle powiązany z serwerem **IIS** (ang. *Internet Information Services*) tworzonym przez Microsoft wyłącznie dla systemów Windows. Przez lata zgromadziło się tam wiele różnych funkcji. Niestety, część z nich zupełnie nie nadaje się do tworzenia rozwiązań wieloplatformowych.

ASP.NET Core to całkowicie przebudowana ASP.NET. Usunięto z niej zależność od zestawu *System.Web.dll* i serwera IIS, a zamiast tego biblioteka składa się z modularnych, lekkich pakietów, tak jak całość nowoczesnego .NET. ASP.NET Core nadal pozwala używać IIS jako serwera WWW, ale udostępnia też znacznie lepsze rozwiązanie.

Wieloplatformowe aplikacje ASP.NET Core można tworzyć w systemach Windows, macOS oraz Linux. Microsoft przygotował nawet wieloplatformowy, bardzo wydajny serwer WWW o nazwie **Kestrel**. Całość stosu serwera została udostępniona w postaci otwartych źródeł.

Projekty tworzone na bazie ASP.NET Core 2.2 domyślnie stosują nowy wbudowany model hostowania stron. Pozwala to uzyskać zwiększenie wydajności o 400% w porównaniu do hostowania stron na serwerze Microsoft IIS, mimo to firma Microsoft zaleca stosowanie serwera Kestrel, gdyż pozwala on uzyskać jeszcze lepszą wydajność.

Tworzenie stron WWW za pomocą ASP.NET Core

Witryny WWW składają się z wielu stron ładowanych statycznie z systemu plików albo generowanych dynamicznie przez technologie działające po stronie serwera, takie jak ASP.NET Core. Przeglądarka WWW wysyła żądania `GET`, podając w nich adresy URL identyfikujące poszczególne strony. Może też manipulować danymi przechowywanymi na serwerze, stosując żądania `POST`, `PUT` i `DELETE`.

W przypadku witryn WWW przeglądarka traktowana jest jak warstwa prezentacji, a niemal wszystkie prace wykonywane są po stronie serwera. Po stronie klienta mogą się pojawiać niewielkie porcje kodu JavaScript, implementujące różne funkcje prezentacyjne, takie jak karuzele obrazów.

ASP.NET Core udostępnia kilka technologii tworzenia witryn WWW:

- **ASP.NET Core Razor Pages** i **biblioteki klas Razor** są metodą dynamicznego generowania kodu HTML dla prostych stron WWW. Więcej informacji na ten temat znajdziesz w rozdziale 13., „Tworzenie witryn WWW przy użyciu ASP.NET Core Razor Pages”.

Kup ksi k Pole ksi k

- **ASP.NET Core MVC** jest implementacją wzorca Model-View-Controller, czyli bardzo popularnej metody tworzenia złożonych witryn WWW. Więcej informacji na ten temat znajdziesz w rozdziale 14., „Tworzenie aplikacji WWW przy użyciu ASP.NET Core MVC”.
- **Technologia Blazor** umożliwia tworzenie serwerowych i klienckich komponentów i interfejsów użytkownika za pomocą języka C#, a nie za pomocą frameworków zbudowanych w języku JavaScript, takich jak Angular, React lub Vue. **Blazor Web Assembly** uruchamia nasz kod w przeglądarce, gdzie działa on tak samo jak kod z frameworków języka JavaScript. **Blazor Server** uruchamia kod na serwerze, gdzie dynamicznie aktualizuje zawartość strony WWW. Więcej informacji na temat technologii Blazor znajdziesz w rozdziale 16., „Tworzenie interfejsów użytkownika w technologii Blazor”.

Porównanie typów plików używanych przez ASP.NET Core

Dobrze jest podsumować wszystkie typy plików używane przez te technologie, ponieważ są one do siebie podobne, ale różnią się w szczegółach. Jeżeli nie zauważasz tych subtelnych, choć istotnych różnic, to możesz napotkać różne problemy podczas implementowania własnych projektów. Wszystkie te różnice zebrałem w tabeli 12.1.

Tabela 12.1. Porównanie typów plików używanych w ASP.NET Core

Technologia	Specjalna nazwa pliku	Rozszerzenie	Dyrektywa
Komponent Blazor (Blazor)		*.razor*	
Komponent Blazor (Blazor z trasowaniem)		*.razor*	`@page`
Strona Razor		*.cshtml*	`@page`
Widok Razor (MVC)		*.cshtml*	
Układ Razor		*.cshtml*	
Widok startowy Razor	`_ViewStart`	*.cshtml*	
Import widoku Razor	`_ViewImports`	*.cshtml*	

Takie dyrektywy jak `@page` dodawane są zawsze na początku pliku.

Jeżeli plik nie ma specjalnej nazwy, to może otrzymać dowolnie wybraną. Na przykład można utworzyć komponent Razor używany w projekcie Blazor i nadać mu nazwę *Klient.razor*. Można też utworzyć plik układu Razor do wykorzystania w projecie MVC lub Razor Pages i nadać mu nazwę *_UkładMobilny.cshtml*.

Uwaga

Konwencja nazywania współdzielonych plików Razor definiujących układy i widoki częściowe nakazuje poprzedzać ich nazwy znakiem podkreślenia. Przykładem mogą być pliki *_ViewStart.cshtml*, *_Layout.cshtml* lub *_Produkt.cshtml*. Ten ostatni plik może być widokiem częściowym wyświetlającym dane produktu.

Pliki układu Razor, takie jak *_MojWlasnyUklad.cshtml*, są definiowane tak samo jak widoki Razor. To, że plik jest traktowany jako definicja układu, wynika z przypisania tego pliku do właściwości `Layout` w innym pliku Razor, tak jak w poniższym kodzie:

```
@{
  Layout = "_MojWlasnyUklad"; // Nie potrzeba podawać rozszerzenia
}
```

Wskazówka

Ostrzeżenie! Pamiętaj o użyciu właściwego rozszerzenia pliku i dyrektywy na początku pliku. Wszystkie popełnione tu błędy będą skutkowały dziwnym zachowaniem w aplikacji.

Tworzenie witryn WWW za pomocą systemu zarządzania treścią

Niektóre witryny zawierają bardzo dużo treści, a każdorazowe angażowanie programistów przy zmianach tych treści byłoby bardzo niepraktyczne.

Systemy zarządzania treścią (ang. *content management system* — **CMS**) umożliwiają zdefiniowanie struktury i szablonów zapewniających spójność wyglądu, którymi później zarządzają właściciele witryny. Mogą oni tworzyć zupełnie nowe strony albo inne bloki treści, aktualizować już istniejące i mieć niezbitą pewność, że całość będzie świetnie wyglądać na ekranach urządzeń użytkowników.

Istnieje wiele różnych systemów zarządzania treścią działających na różnych platformach sieciowych, takich jak WordPress dla PHP albo Django dla Pythona. Rozbudowane systemy CMS zbudowane w nowoczesnym .NET to Optimizely Content Cloud, Piranha CMS albo Orchard Core.

Główną zaletą stosowania systemu CMS jest możliwość korzystania z przyjaznego interfejsu użytkownika do zarządzania treściami. Właściciele witryn mogą się zalogować do swojego systemu i sami zarządzać wszystkimi treściami. Tak przygotowany materiał jest następnie renderowany i pokazywany użytkownikom za pośrednictwem kontrolerów i widoków ASP.NET MVC albo serwisowych punktów końcowych w sieci WWW, nazywanych też **headless CMS**, które dostarczają treści dla aplikacji zaimplementowanych dla urządzeń mobilnych lub stacjonarnych oraz innych klientów tworzonych we frameworkach javascriptowych lub za pomocą technologii Blazor.

W tej książce nie omawiam systemów CMS tworzonych w .NET, dlatego pod poniższym adresem podaję linki do artykułów omawiających ten temat:

https://github.com/markjprice/cs12dotnet8/blob/main/docs/book-links.md#
↪net-content-management-systems

Tworzenie aplikacji WWW za pomocą framework SPA

Aplikacje WWW, znane też pod nazwą **aplikacji jednostronicowych** (ang. *Single-Page application* — **SPA**), składają się z pojedynczej strony WWW zbudowanej za pomocą technologii działającej w przeglądarce, takiej jak Blazor WebAssembly, Angular, React, Vue, albo innych własnościowych bibliotek JavaScript, które są w stanie wysyłać żądania do serwera sieciowego, aby pobierać dodatkowe informacje i przesyłać zaktualizowane

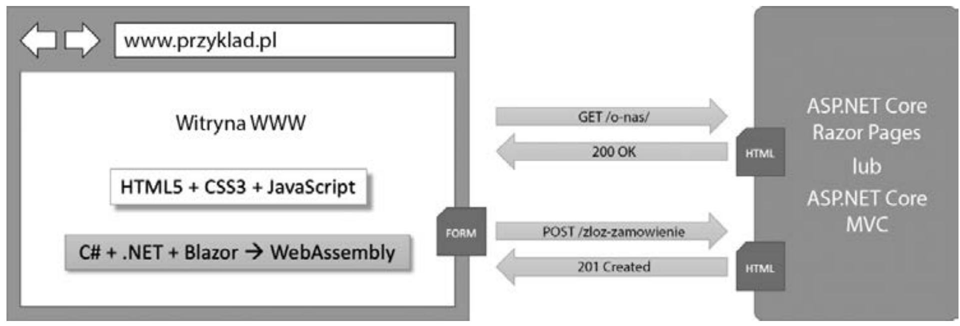
dane. Wykorzystują do tego standardowe formaty serializacji, takie jak XML lub JSON. Typowym przykładem tego rodzaju aplikacji mogą być aplikacje Google, np.: Gmail, Maps lub Docs.

W przypadku aplikacji WWW używane są biblioteki języka JavaScript działające po stronie klienta, gdzie obsługują złożone interakcje z użytkownikiem. Mimo to główne procesy przetwarzania i dostępu do danych realizowane są po stronie serwera, ponieważ przeglądarki mają tylko ograniczony dostęp do zasobów systemowych.

JavaScript jest językiem o luźnym typowaniu, który nie jest przygotowany do obsługi złożonych projektów, dlatego większość bibliotek JavaScriptu używa dzisiaj języka Microsoft TypeScript, który wprowadził silne typowanie oraz wiele funkcji nowoczesnych języków programowania umożliwiających realizację złożonych implementacji.

W .NET SDK dostępne są szablony projektów dla aplikacji SPA w języku JavaScript i Type-Script, jednak w tej książce nie będziemy zajmować się metodami tworzenia aplikacji SPA w tym języku, mimo że zazwyczaj wykorzystują one system ASP.NET Core po stronie serwera. To jest jednak książka o C#, a nie o innych językach programowania.

Podsumowując: język C# i platforma .NET mogą być używane zarówno po stronie serwera, jak i po stronie klienta, tworząc rozbudowane witryny, tak jak na rysunku 12.1.



Rysunek 12.1. Sposób użycia języka C# i środowiska .NET do tworzenia serwisów sieciowych po stronie serwera i po stronie klienta

Tworzenie serwisów sieciowych

Co prawda nie będziemy zajmować się tworzeniem aplikacji SPA w języku JavaScript lub TypeScript, ale nauczymy się budować serwisy sieciowe za pomocą **ASP.NET Core Web API** i wywoływać je z kodu działającego po stronie serwera w aplikacjach witryn ASP.NET Core. Później wywołamy przygotowaną usługę sieciową z komponentów tworzonych dla rozwiązań Blazor.

Nie istnieją żadne formalne definicje, ale usługi są często opisywane pod kątem stopnia ich złożoności:

- **Serwis** — wszystkie funkcje używane przez aplikację kliencką znajdują się w jednym, monolitycznym serwisie.

- **Mikroserwis** — wiele serwisów skupiających się na realizacji małego zbioru funkcji.
- **Nanoserwis** — dany serwis realizuje tylko jedną funkcję. W przeciwieństwie do serwisów i mikroserwisów, które muszą działać 24/7/365, nanoserwisy są nieaktywne i są aktywowane dopiero w momencie ich wywołania, co pozwala oszczędzać zasoby i koszty.

Na początku pierwszej części tej książki pokrótce przedstawiłem funkcje języka C#, podając przy tym wersję, w której pojawiła się dana funkcja. Na początku drugiej części książki pokrótce przedstawiłem funkcje biblioteki .NET, podając przy tym wersję, w której pojawiła się dana funkcja. Teraz, w trzeciej części książki przyjrzymy się różnym funkcjom ASP.NET Core. Tutaj również podawać będę numer wersji, w której pojawiły się poszczególne funkcje.

Wszystkie te informacje znajdziesz w dodatkowych materiałach do tej książki dostępnych pod adresem: *https//ftp.helion.pl/przykłady/C12N88.zip*.

Struktury projektów

Jak należy zatem przygotowywać struktury projektów? Dotychczas tworzyliśmy tylko proste aplikacje konsoli, aby zilustrować funkcje języka lub biblioteki. W dalszej części tej książki utworzymy wiele projektów wykorzystujących różne technologie, które będą ze sobą współpracowały tworząc jedno rozwiązanie.

W wielkich i złożonych rozwiązaniach nawigowanie w kodzie bywa kłopotliwe. Podstawowym powodem planowania struktury projektów jest ułatwienie wyszukiwania komponentów. Dobrze jest też wybrać ogólną nazwę rozwiązania odzwierciedlającą przeznaczenie tworzonej aplikacji.

Zbudujemy teraz kilka projektów dla fikcyjnej firmy **Northwind**. Rozwiązaniu lub przestrzeni roboczej nadamy nazwę `PraktyczneAplikacje`, natomiast nazwy projektów uzupełnimy przedrostkiem `Northwind`.

Istnieje wiele sposobów tworzenia struktury projektu i nadawania nazw projektom i rozwiązaniom. Na przykład można wykorzystywać hierarchię folderów albo stosować wybrane konwencje nazewnicze. Jeżeli pracujesz w zespole, to upewnij się, że wiesz, jak robi to cały zespół.

Struktura projektów w rozwiązaniu

Dobrze jest mieć konwencję nazywania projektów w rozwiązaniu lub przestrzeni roboczej, dzięki której każdy programista będzie od razu wiedzieć, do czego służy dany projekt. Często stosowaną metodą jest użycie typu projektu, na przykład biblioteka klas lub aplikacja konsoli, strona WWW itd.

Zazwyczaj chcemy uruchomić jednocześnie wiele projektów sieciowych, które mają pracować na lokalnym serwerze WWW. W takiej sytuacji trzeba rozdzielić poszczególne projekty, przydzielając im różne numery portów dla punktów końcowych HTTP i HTTPS.

Kup ksi k Pole ksi k

Najczęściej używany jest port numer 5000 dla punktu końcowego HTTP i port 5001 dla HTTPS. W tej książce przyjąłem konwencję wybierania portów zgodnie ze schematem 5<*rozdział*>0 dla HTTP i 5<*rozdział*>1 dla HTTPS. Na przykład witryna, którą przygotujemy w rozdziale 13., będzie używała portu 5130 dla HTTP i 5131 dla HTTPS.

W tabeli 12.2 zebrałem wszystkie nazwy projektów i używane przez nie numery portów.

Tabela 12.2. Przykładowe nazwy dla różnych typów projektów

Nazwa	Porty	Opis
`Northwind.Wspolne`	N.D.	Projekt biblioteki klas przechowującej wspólne typy, takie jak interfejsy, enumeracje, klasy, rekordy i struktury, które są używane w wielu projektach.
`Northwind.ModeleEncji`	N.D.	Projekt biblioteki klas przechowującej modele encji EF Core; modele encji zwykle są używane przez klienta i przez serwer, dlatego lepiej jest oddzielić zależności od konkretnych dostawców baz danych.
`Northwind.KontekstDanych`	N.D.	Projekt biblioteki klas przechowującej kontekst bazy danych EF Core z zależnościami od konkretnych dostawców.
`Northwind.Testy`	N.D.	Projekt xUnit zawierający testy dla całego rozwiązania.
`Northwind.Web`	http 5130 https 5131	Projekt ASP.NET Core tworzący prostą witrynę WWW wykorzystującą mieszankę statycznych plików HTML i dynamicznych stron Razor Pages.
`Northwind.MVC`	http 5140 https 5141	Projekt ASP.NET Core dla złożonej witryny zbudowanej zgodnie ze wzorcem MVC, ułatwiającym tworzenie testów jednostkowych.
`Northwind.WebAPI`	http 5150 https 5151	Projekt ASP.NET Core dla serwisu HTTP API; jest dobrym rozwiązaniem do zintegrowania z witrynami WWW, ponieważ umożliwia komunikację z dowolną biblioteką JavaScript albo Blazor.
`Northwind.MinimalApi`	http 5152	Projekt ASP.NET Core dla minimalnego serwisu HTTP. W przeciwieństwie do projektów Web API te projekty mogą być kompilowane z funkcją native AOT, co poprawia czas uruchamiania i zmniejsza zużycie pamięci.
`Northwind.Blazor`	http 5160 https 5161	Projekt serwera ASP.NET Core Blazor Server.

Tworzenie modelu encji używanego w tej książce

Praktyczne aplikacje zazwyczaj muszą pracować z danymi zapisanymi w relacyjnej bazie danych albo w innej składnicy danych. W tym rozdziale zdefiniujemy model danych dla bazy Northwind przechowywanej na serwerze SQL Server lub SQLite. Przygotowany tu model danych będzie używany w większości aplikacji, jakie będziemy tworzyć w kolejnych rozdziałach.

Tworzenie bazy danych Northwind

Pliki skryptów *Northwind4SQLServer.sql* i *Northwind4SQLite.sql* nie są identyczne. Skrypt dla SQL Server tworzy 13 tabel oraz powiązane z nimi widoki i procedury składowane. Skrypt dla SQLite jest jego uproszczoną wersją, tworzącą tylko 10 tabel. Wynika to z faktu, że SQLite nie obsługuje wielu funkcji dostępnych w SQL Server. Główne projekty z tej książki wykorzystują jednak tylko 10 tabel, dlatego wszystkie prezentowane tu zadania można wykonać z dowolnym serwerem baz danych.

Wszystkie skrypty SQL znajdziesz w repozytorium pod adresem *https://github.com/markjprice/cs12dotnet8/tree/main/scripts/sql-scripts*.

Dostępnych jest tam kilka skryptów:

- `Northwind4Sqlite.sql` — do użycia z lokalną bazą SQLite w systemach Windows, macOS lub Linux. Ten skrypt prawdopodobnie zadziała też z innymi systemami SLQ, takimi jak PostgreSQL lub MySQL, ale nie został z nimi jeszcze przetestowany.
- `Northwind4SqlServer.sql` — do użycia z lokalnym serwerem SQL Server w systemie Windows. Skrypt sprawdza, czy baza jest już utworzona na serwerze, a jeżeli jest, to ją usuwa i tworzy na nowo.
- `Northwind4AzureSqlDatabaseCloud.sql` — do użycia z serwerem SQL Server działającym jako zasób *Azure SQL Database* w chmurze Azure. Takie zasoby nie są darmowe przez cały czas swojego istnienia! Sam skrypt nie usuwa i nie tworzy bazy danych *Northwind*, ponieważ te operacje trzeba wykonać ręcznie w interfejsie użytkownika portalu Azure.
- `Northwind4AzureSqlEdgeDocker.sql` — do użycia z serwerem SQL Server działającym lokalnie jako kontener Dockera. Skrypt tworzy bazę danych Northwind. Nie próbuje usuwać istniejącej już bazy, ponieważ kontener po każdym uruchomieniu powinien być pusty i nie zawierać żadnych baz.

Dokładny opis procedury instalowania serwera SQL Server lub SQLite znajduje się w rozdziale 10., „Praca z bazami danych przy użyciu Entity Framework Core”. Znajdziesz w nim też instrukcje dotyczące instalowania narzędzia `dotnet-ef`, które wykorzystamy tutaj do utworzenia modelu encji na podstawie istniejącej bazy danych.

Kup ksi k Pole ksi k

Instrukcję instalowania serwera SQL Server Developer Edition (to wersja bezpłatna) można znaleźć w repozytorium GitHub dla tej książki pod adresem *https://github.com/markjprice/cs12dotnet8/blob/main/docs/sql-server/README.md*.

Instrukcję instalowania serwera Azure SQL Edge w Dockerze dla systemów Windows, macOS i Linux znajdziesz w repozytorium GitHub dla tej książki pod adresem *https://github.com/markjprice/apps-services-net8/blob/main/docs/ch02-sql-edge.md*.

Tworzenie biblioteki klas dla modelu encji bazy SQLite

Teraz zdefiniujemy model encji w bibliotece klas, tak aby można było go używać w innych rodzajach projektów, w tym w modelach aplikacji klienckich.

Wskazówka

Dobra praktyka: Dobrze jest utworzyć osobny projekt biblioteki klas przeznaczony na model encji. Znacznie ułatwia to współdzielenie modelu przez centralny serwer oraz wszystkie aplikacje klienckie, mobilne oraz klientów Blazor.

Wygenerujemy teraz automatycznie część modeli encji za pomocą narzędzi wiersza poleceń:

1. Użyj swojego edytora kodu, aby utworzyć nowy projekt, używając ustawień z następującej listy:
   - Szablon projektu: *Biblioteka klas* lub `classlib`.
   - Plik lub folder projektu: *Northwind.ModeleEncji.Sqlite*.
   - Plik i folder rozwiązania: *PraktyczneAplikacje*.
2. W pliku projektu *Northwind.ModeleEncji.Sqlite* dodaj referencję pakietu dostawcy danych SQLite oraz pakietu projektowania w EF Core:

```
<ItemGroup>
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.Sqlite"
    Version="8.0.0" />
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.Design"
    Version="8.0.0">
  <PrivateAssets>all</PrivateAssets>
  <IncludeAssets>runtime; build; native; contentfiles; analyzers;
  ↪buildtransitive</IncludeAssets>
  </PackageReference>
</ItemGroup>
```

3. Usuń plik *Class1.cs*.
4. Skompiluj projekt, aby pobrać niezbędne pakiety.
5. Skopiuj plik *Northwind4SQLite.sql* do folderu rozwiązania *PraktyczneAplikacje*, ale nie do folderu projektu!

Kup ksi k Pole ksi k

6. Utwórz plik *Northwind.db* wprowadzając poniższe polecenie w oknie terminala lub wiersza poleceń:

```
sqlite3 Northwind.db -init Northwind4SQLite.sql
```

7. Poczekaj cierpliwie, ponieważ ten skrypt potrzebuje kilku chwil na utworzenie całej struktury bazy danych.
8. Naciśnij klawisze *Ctrl+C* w systemie Windows lub *Cmd+D* w systemie macOS, aby zakończyć tryb poleceń SLQite.
9. Otwórz okno wiersza poleceń lub terminala w folderze *Northwind.ModeleEncji.Sqlite*. W wierszu poleceń wygeneruj model danych encji dla wszystkich tabel, używając poniższego polecenia:

```
dotnet ef dbcontext scaffold "Data Source=../Northwind.db"
↪Microsoft.EntityFramework Core.Sqlite --namespace Biblioteka.ModeleEncji
↪--data-annotations
```

Zwróć uwagę na następujące szczegóły:

- Uruchamiane jest polecenie `dbcontext scaffold`.
- Ciąg znaków połączenia to `"Data Source=../Northwind.db"`.
- Dostawca bazy danych to `Microsoft.EntityFrameworkCore.Sqlite`.
- Przestrzeń nazw to `--namespace Biblioteka.ModeleEncji`.
- Włączenie użycia atrybutów i płynnego API: `--data-annotations`.

Wskazówka

Ostrzeżenie! Polecenia `dotnet-ef` muszą być wprowadzane w jednym wierszu i w folderze zawierającym plik projektu. Jeżeli tego nie dopilnujesz, to pojawi się błąd: *No project was found. Change the current working directory or use the --project option*. Pamiętaj, że wszystkie polecenia znajdziesz na stronie pod adresem *https://github.com/markjprice/cs12dotnet8/blob/main/docs/command-lines.md*.

Uwaga

Jeżeli używasz bazy SQLite, to zobaczysz ostrzeżenie mówiące o niezgodnych mapowaniach typów pomiędzy kolumnami tabeli i właściwościami w klasach modeli encji, takie jak: *The column 'BirthDate' on table 'Employees' should map to a property of type 'DateOnly', but its values are in an incompatible format. Using a different type*. Wynika to z faktu, że SQLite używa dynamicznego typowania. Wszystkie te problemy rozwiążemy w następnym podrozdziale.

Tworzenie biblioteki klas z kontekstem bazy danych Northwind

Teraz zdefiniujemy kontekst bazy danych.

1. Do rozwiązania lub obszaru roboczego dodaj projekt biblioteki klas, używając następujących ustawień:
   - Szablon projektu: *Biblioteka klas* lub `classlib`.

- Plik lub folder projektu: *Northwind.KontekstDanych.Sqlite*.
- Plik lub folder rozwiązania: *PraktyczneAplikacje*.

2. Zmień zawartość pliku *Northwind.KontekstDanych.Sqlite.csproj*, statycznie i globalnie importując klasę `Console` oraz dodając referencję do projektu *Northwind.ModeleEncji.Sqlite* i pakietu *Entity Framework Core for SQLite*:

```
<ItemGroup>
  <Using Include="System.Console" Static="true" />
</ItemGroup>

<ItemGroup>
  <PackageReference
    Include="Microsoft.EntityFrameworkCore.SQLite"
    Version="8.0.0" />
</ItemGroup>

<ItemGroup>
  <ProjectReference

Include="..\Northwind.ModeleEncji.Sqlite\Northwind.ModeleEncji.Sqlite.csproj" />
</ItemGroup>
```

Wskazówka

Ostrzeżenie: W pliku projektu ścieżka zapisana w referencji projektu nie powinna zawierać znaków końca wiersza.

3. W projekcie *Northwind.KontekstDanych.Sqlite* usuń plik *Class1.cs*.
4. Skompiluj projekt *Northwind.KontekstDanych.Sqlite.*
5. W projekcie *Northwind.KontekstDanych* dodaj nowy plik klasy o nazwie *NorthwindProtokol.cs*.
6. Zmień zawartość nowego pliku, definiując w nim metodę statyczną o nazwie `WriteLine`, która będzie dopisywać ciąg znaków na końcu pliku tekstowego o nazwie *northwindlog.txt*:

```
using static System.Environment;

namespace Northwind.ModeleEncji;

public class NorthwindProtokol
{
  public static void WriteLine(string komunikat)
  {
    string sciezka = Path.Combine(GetFolderPath(
        SpecialFolder.DesktopDirectory), "northwindlog.txt");

    StreamWriter plikTekstowy = File.AppendText(sciezka);
    plikTekstowy.WriteLine(komunikat);
    plikTekstowy.Close();
  }
}
```

Kup ksi k Pole ksi k

7. Przenieś plik *NorthwindContext.cs* z folderu projektu *Northwind.ModeleEncji.Sqlite* do folderu projektu *Northwind.KontekstDanych.Sqlite*.

Wskazówka

W Visual Studio w okienku *Eksplorator rozwiązań* przeciągnięcie pliku między projektami spowoduje jego skopiowanie. Jeżeli podczas przeciągania przytrzymasz przycisk *Shift*, to plik zostanie przeniesiony. W Visual Studio Code w okienku *Explorer* przeciągnięcie pliku między projektami spowoduje jego przeniesienie. Aby skopiować plik, musisz go przeciągnąć, przytrzymując klawisz *Ctrl*.

8. W pliku *NorthwindContext.cs* zwróć uwagę na to, że drugi konstruktor pobiera parametr `options`, który umożliwia podanie ciągu znaków połączenia innego niż domyślny. Używać tego można w różnych projektach, na przykład w projekcie strony WWW, która musi pracować z bazą danych.

```
public NorthwindContext(DbContextOptions<NorthwindContext> options)
: base(options)
{
}
```

9. W pliku *NorthwindContext.cs*, w metodzie `OnConfiguring`, usuń ostrzeżenie kompilatora dotyczące ciągu znaków połączenia. Następnie dopisz instrukcje sprawdzające aktualny katalog, aby dopasować się do sytuacji, gdy aplikacja uruchamiana jest w Visual Studio 2022 lub w wierszu poleceń Visual Studio Code, zgodnie z wyróżnieniami w poniższym kodzie:

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
  if (!optionsBuilder.IsConfigured)
  {
    string bazaDanych = "Northwind.db";
    string katalog = Environment.CurrentDirectory;
    string sciezka = string.Empty;

    if (katalog.EndsWith("net8.0"))
    {
      // Uruchomienie w katalogu <projekt>\bin\<Debug|Release>\net8.0
      sciezka = Path.Combine("..", "..", "..", "..", bazaDanych);
    }
    else
    {
      // Uruchomienie w katalogu <projekt>
      sciezka = Path.Combine("..", bazaDanych);
    }

    sciezka = Path.GetFullPath(sciezka); // Zamiana na ścieżkę bezwzględną
    NorthwindProtokol.Wypisz($"Ścieżka do bazy danych: {sciezka}");

    if (!File.Exists(sciezka))
    {
      throw new FileNotFoundException(
        message: $"Nie znaleziono {sciezka}.", fileName: sciezka);
    }
```

```
      optionsBuilder.UseSqlite($"Data Source={sciezka}");

      optionsBuilder.LogTo(NorthwindProtokol.WriteLine,
        new[] { Microsoft.EntityFrameworkCore
          .Diagnostics.RelationalEventId.CommandExecuting });
    }
  }
```

Uwaga

Rzucenie wyjątku w tym miejscu jest bardzo ważne, ponieważ w przypadku gdy plik bazy danych nie istnieje, to dostawca danych SQLite utworzy nowy, pusty plik, a zatem próba połączenia z bazą danych się powiedzie. Jednak przy uruchamianiu zapytań otrzymamy wyjątki mówiące o brakujących tabelach, ponieważ nowo utworzona baza danych nie ma jeszcze żadnych tabel! Dzięki zamianie ścieżki względnej na bezwzględną łatwiej będzie podczas debugowania sprawdzić, gdzie znajduje się plik bazy danych.

Dostosowanie modelu i definiowanie metod rozszerzających

Teraz uprościmy metodę `OnModelCreating`. Pokrótce opiszę kolejne kroki, a na koniec zaprezentuję cały kod metody. Możesz zatem albo wykonywać działania krok po kroku, albo od razu skopiować całość metody.

1. W metodzie `OnModelCreating` usuń wszystkie instrukcje płynnego API wywołujące metodę `ValueGeneratorNever` w celu skonfigurowania właściwości klucza głównego, takich jak `CategoryId`, tak jak w poniższym kodzie. Chodzi o to, żeby nigdy nie generować automatycznie wartości klucza. W zamian można też wywołać metodę `HasDefaultValueSql`.

```
modelBuilder.Entity<Category>(entity =>
{
  entity.Property(e => e.CategoryId).ValueGeneratedNever();
});
```

Wskazówka

Jeżeli nie usuniemy z konfiguracji pokazanych wyżej instrukcji, to podczas dodawania nowego dostawcy właściwość `CategoryId` będzie zawsze miała wartość `0`, przez co będziemy mogli dodać tylko jednego dostawcę z takim identyfikatorem. Wszystkie kolejne próby dodania dostawcy będą kończyły się rzuceniem wyjątku. Możesz porównać swój plik *NorthwindContext.cs* z plikiem dostępnym w repozytorium GitHub pod adresem *https://github.com/markjprice/cs12dotnet8/blob/main/code/PracticalApps/Northwind.DataContext.Sqlite/NorthwindContext.cs*.

2. W definicji encji `Product` poinformuj serwer SQLite, że w przypadku kolumny `UnitPrice` można zmienić typ `decimal` na `double`, tak jak w poniższym kodzie:

```
entity.Property(product => product.UnitPrice)
  .HasConversion<double>();
```

Kup ksi k Pole ksi k

3. Teraz metoda `OnModelCreating` powinna być znacznie prostsza i wyglądać tak:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  modelBuilder.Entity<Order>(entity =>
  {
    entity.Property(e => e.Freight).HasDefaultValueSql("0");
  });

  modelBuilder.Entity<OrderDetail>(entity =>
  {
    entity.Property(e => e.Quantity).HasDefaultValueSql("1");
    entity.Property(e => e.UnitPrice).HasDefaultValueSql("0");
    entity.HasOne(d => d.Order).WithMany(p =>
      p.OrderDetails).OnDelete(DeleteBehavior.ClientSetNull);
    entity.HasOne(d => d.Product).WithMany(p =>
      p.OrderDetails).OnDelete(DeleteBehavior.ClientSetNull);
  });

  modelBuilder.Entity<Product>(entity =>
  {
    entity.Property(e => e.Discontinued).HasDefaultValueSql("0");
    entity.Property(e => e.ReorderLevel).HasDefaultValueSql("0");
    entity.Property(e => e.UnitPrice).HasDefaultValueSql("0");
    entity.Property(e => e.UnitsInStock).HasDefaultValueSql("0");
    entity.Property(e => e.UnitsOnOrder).HasDefaultValueSql("0");
    entity.Property(product => product.UnitPrice)
      .HasConversion<double>();
  });

  OnModelCreatingPartial(modelBuilder);
}
```

4. W projekcie *Northwind.KontekstDanych.Sqlite* dodaj klasę o nazwie *NorthwindContextExtensions.cs* i zmień jej zawartość tak, żeby zdefiniować metodę rozszerzającą, która doda kontekst bazy danych Northwind do kolekcji zależnych serwisów:

```
using Microsoft.EntityFrameworkCore; // UseSqlite
using Microsoft.Extensions.DependencyInjection; // IServiceCollection

namespace Biblioteka.ModeleEncji;

public static class NorthwindContextExtensions
{
  /// <summary>
  /// Dodaje obiekt NorthwindContext do wskazanej kolekcji typu
  /// IServiceCollection. Używa dostawcy bazy danych Sqlite.
  /// </summary>
  /// <param name="serwisy"></param>
  /// <param name="sciezkaWzgledna">Ta wartość podmienia domyślną
  /// ścieżkę ".."</param>
  /// <param name="bazaDanych">Domyślna nazwa to "Northwind.db"</param>
  /// <returns>Kolekcja typu IServiceCollection,
  /// której można użyć do dodawania kolejnych serwisów.</returns>
```

```
    public static IServiceCollection DodajKontekstNorthwind(
      this IServiceCollection serwisy, // Typ rozszerzany
      string sciezkaWzgledna = "..",
      string bazaDanych = "Northwind.db")
    {
      string sciezka = Path.Combine(sciezkaWzgledna, bazaDanych);
      sciezka = Path.GetFullPath(sciezka);
      NorthwindProtokol.WriteLine($"Ścieżka bazy danych: {sciezka}");

      if (!File.Exists(sciezka))
      {
        throw new FileNotFoundException(
          message: $"Nie znaleziono {sciezka}.", fileName: sciezka);
      }

      serwisy.AddDbContext<NorthwindContext>(opcje =>
      {
        // Dzisiaj zamiast parametru Filename używa się Data Source
        opcje.UseSqlite($"Data Source={sciezka}")

        opcje.LogTo(NorthwindProtokol.WriteLine,
          new[] { Microsoft.EntityFrameworkCore
          .Diagnostics.RelationalEventId.CommandExecuting });
      },
      // Zarejestruj z parametrem transient, aby uniknąć problemów
      // ze współbieżnością w serwerowych projektach Blazor.
      contextLifetime: ServiceLifetime.Transient,
      optionsLifetime: ServiceLifetime.Transient);

      return serwisy;
    }
  }
```

5. Skompiluj dwie biblioteki klas i popraw ewentualne błędy kompilatora.

Rejestrowanie zakresu zależnego serwisu

Domyślnie klasa `DbContext` jest rejestrowana z czasem życia ustalonym przez wartość `Scope`. Oznacza to, że wiele wątków może wspólnie korzystać z tego samego obiektu. Niestety klasa `DbContext` nie obsługuje pracy z wieloma wątkami. Jeżeli z klasy `NorthwindContext` będzie próbował korzystać więcej niż jeden wątek, to w trakcie pracy klasa rzuci wyjątek: *A second operation started on this context before a previous operation completed. This is usually caused by different threads using the same instance of a DbContext, however instance members are not guaranteed to be thread safe*.

Zdarza się to w projektach Blazor z komponentami działającymi po stronie serwera. W przypadku gdy użytkownik wykona operację po stronie klienta, protokół SignalR przekazuje informację do serwera, gdzie jedna instancja kontekstu bazy danych jest współdzielona przez wiele klientów. Ten problem nie pojawia się wcale, gdy ten sam komponent działa po stronie klienta.

Kup ksi k Pole ksi k

Tworzenie biblioteki klas modelu encji dla SQL Server

Jeżeli zamiast bazy danych SQLite chcesz użyć serwera SQL Server, to instrukcje podobne do powyższych znajdziesz pod adresem *https://github.com/markjprice/cs12dot ↪net8/blob/main/docs/sql-server/README.md#chapter-12---introducing-web-develop ↪ment-using-aspnet-core*.

Usprawnianie odwzorowania klas na tabele

Polecenie `dotnet-ef` generuje inny kod dla serwera SQL Server, a inny dla baz SQLite, ponieważ oba rozwiązania obsługują odmienne zestawy funkcji, a dodatkowo SQLite stosuje typowanie dynamiczne. Na przykład w EF Core 7 wszystkie kolumny typu `int` są odwzorowywane na nullowalny typ `long`, aby zachować najwyższy poziom elastyczności.

W EF Core 8 i nowszych wartości zapisywane w tych kolumnach są kontrolowane i jeżeli mieszczą się w typie `int`, to są odwzorowywane na właściwość typu `int`. Jeżeli wszystkie wartości z danej kolumny mieszczą się w typie `short`, to odwzorowująca właściwość otrzyma typ `short`.

W tym wydaniu mamy znacznie mniej pracy przy poprawianiu odwzorowań klas. Hura!

Kolejny przykład: w SQL Server można ograniczyć liczbę znaków w kolumnach tekstowych. SQLite nie obsługuje tej funkcji. W związku z tym polecenie `dotnet-ef` wygeneruje atrybuty kontrolne zapewniające, że właściwości typu `string` będą zgodne z ograniczeniem liczby znaków wyłączenie dla serwera SQL Serve, ale pominie je dla SQLite, tak jak w poniższym kodzie:

```
// Kod wygenerowany przez dostawcę SQLite
[Column(TypeName = "nvarchar (15)")]
public string CategoryName { get; set; } = null!;

// Kod wygenerowany przez dostawcę SQL Server
[StringLength(15)]
public string CategoryName { get; set; } = null!;
```

Wprowadzimy tu kilka małych zmian, aby usprawnić odwzorowania modelu encji i dodać reguły kontroli poprawności dla SQLite. Podobne instrukcje dla serwera SQL Server dostępne są w moim repozytorium GitHub.

Uwaga

Pamiętaj, że całość kodu jest dostępna na serwerze FTP wydawnictwa Helion. Samodzielnie wpisując całość kodu wiele się nauczysz, ale wcale nie musisz tego robić. Możesz też pobrać kod z tej książki pod adresem *https://ftp.helion.pl/przyklady/c12n88.zip*.

Po pierwsze, dodamy wyrażenie regularne sprawdzające, czy wartość pola `CustomerId` składa się z pięciu wielkich liter. Po drugie, wprowadzimy wymogi dotyczące długości ciągu znaków w tych modelach encji, co do których znamy już maksymalną dozwoloną długość wartości tekstowych.

Kup ksi k Pole ksi k

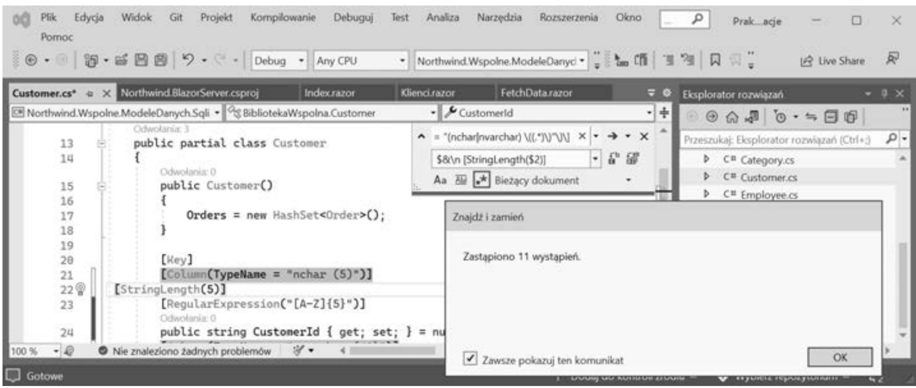
1. W swoim edytorze kodu wywołaj funkcję wyszukiwania i zamiany:
2. W Visual Studio 2022 wybierz z menu pozycję *Edycja/Znajdź i zamień/Szybkie zamienianie*) i włącz opcję *Użyj wyrażeń regularnych*.
3. W polu wyszukiwania wpisz poniższe wyrażenie regularne:

```
\[Column\(TypeName = "(nchar|nvarchar) \((.*)\)"\)\]
```

4. W polu *Zastąp...* wpisz nowe wyrażenie regularne:

```
$&\n    [StringLength($2)]
```

   Za znakiem nowego wiersza (`\n`) wstawiłem cztery znaki spacji, aby wprowadzić wcięcia właściwe dla mojego systemu, w którym używam dwóch spacji na każdy poziom wcięcia. W swoim kodzie możesz użyć tylu spacji, ile będzie Ci potrzebne.
5. Zaznacz opcję wyszukiwania i zamieniania we wszystkich plikach aktualnego projektu.
6. Uruchom wyszukiwanie i zamienianie, klikając przycisk *Zamień wszystkie*, tak jak na rysunku 12.2.



Rysunek 12.2. Wyszukaj i zamień wszystkie dopasowania wyrażenia regularnego w Visual Studio 2022

7. Zmień wszystkie właściwości związane z datą lub czasem, na przykład w pliku *Employee.cs*, tak żeby kod używał nullowalnego typu `DateTime`, a nie ciągu znaków:

```
// Przed zmianą
[Column(TypeName = "datetime")]
public string? BirthDate { get; set; }

// Po zmianie
[Column(TypeName = "datetime")]
public DateTime? BirthDate { get; set; }
```

Wskazówka

Więcej informacji: Użyj w swoim edytorze kodu funkcji *Wyszukaj i zamień*, aby poszukać tekstu „datetime” i znaleźć wszystkie właściwości wymagające modyfikacji. Dwa wystąpienia powinny znajdować się w pliku *Employee.cs*, a w pliku *Order.cs* jeszcze trzy.

Kup ksi k Pole ksi k

8. Zmień wszystkie właściwości związane z typem `money`, na przykład w pliku *Order.cs*, tak żeby kod używał nullowalnego typu `Decimal`, a nie typu `double`:

```
// Przed zmianą
[Column(TypeName = "money")]
public double? Freight { get; set; }

// Po zmianie
[Column(TypeName = "money")]
public decimal? Freight { get; set; }
```

Wskazówka

Więcej informacji: Użyj w swoim edytorze kodu funkcji *Wyszukaj i zamień*, aby poszukać tekstu „money” i znaleźć wszystkie właściwości wymagające modyfikacji. Jedno wystąpienie powinno znajdować się w pliku *Order.cs*, jedno w *OrderDetails.cs* i jedno w *Product.cs*.

9. W pliku *Category.cs* dopisz atrybut `Requred` do właściwości `CategoryName`:

```
[Required]
[Column(TypeName = "nvarchar (15)")]
[StringLength(15)]
public string CategoryName { get; set; }
```

10. W pliku *Customer.cs* dodaj wyrażenie regularne sprawdzające, czy klucz główny z pola `CustomerId` składa się wyłącznie z pięciu wielkich liter. Dopisz też atrybut `Requred` do właściwości `CompanyName`:

```
[Key]
[Column(TypeName = "nchar (5)")]
[StringLength(5)]
[RegularExpression("[A-Z]{5}")]
public string CustomerId { get; set; } = null!;

[Required]
[Column(TypeName = "nvarchar (40)")]
[StringLength(40)]
public string CompanyName { get; set; }
```

11. W pliku *Employee.cs* dopisz atrybut `Required` do właściwości `FirstName` i `LastName`.
12. W pliku *EmployeeTerritory.cs* dopisz atrybut `Required` do właściwości `TerritoryId`.
13. W pliku *Order.cs* dopisz do właściwości `CustomerId` atrybut wyrażenia regularnego wymuszającego stosowanie pięciu wielkich liter.
14. W pliku *Product.cs* dopisz atrybut `Required` do właściwości `ProductName`.
15. W pliku *Shipper.cs* dopisz atrybut `Required` do właściwości `CompanyName`.
16. W pliku *Supplier.cs* dopisz atrybut `Required` do właściwości `CompanyName`.
17. W pliku *Territory.cs* dopisz atrybut `Required` do właściwości `TerritoryId` i `TerritoryDescription`.

Kup ksi k Pole ksi k

Testowanie bibliotek klas

Przygotujmy teraz kilka testów jednostkowych, aby upewnić się, że nasze biblioteki klas działają poprawnie.

Wskazówka

Ostrzeżenie! Jeżeli używasz dostawcy danych SQLite, pamiętaj, że metoda `CanConnect` wywołana z nieprawidłowym lub nieistniejącym plikiem bazy danych sama utworzy pusty plik o podanej nazwie. To bardzo ważne! Z tego powodu nasza klasa `Kontekts Northwind` najpierw sprawdza, czy plik bazy danych istnieje, i rzuca wyjątek, jeżeli tego pliku brakuje. Chcemy w ten sposób uniknąć tworzenia pustych plików baz danych.

Napiszmy zatem kilka testów:

1. W swoim edytorze kodu dodaj nowy projekt o nazwie *Northwind.TestyJednostkowe*, umieszczając go w rozwiązaniu PraktyczneAplikacje. Do tworzenia projektu użyj szablonu *Projekt Testów xUnit [C#]* lub `xunit`.
2. W projekcie *Northwind.TestyJednostkowe* dodaj referencję projektu *Northwind.KontekstDanych* właściwą dla serwera SQLite lub SQL Server, zależnie od tego, którego z nich używasz. Zastosuj kod wyróżniony poniżej:

```
<ItemGroup>
  <!-- Możesz zmienić Sqlite na SqlServer -->
  <ProjectReference Include="..\Northwind.KontekstDanych.Sqlite\
Northwind.KontekstDanych.Sqlite.csproj" />
</ItemGroup>
```

Wskazówka

Ostrzeżenie: Referencja projektu nie powinna zawierać znaków końca wiersza.

3. Skompiluj projekt *Northwind.TestyJednostkowe*.
4. Zmień nazwę pliku *UnitTest1.cs* na *TestyModeluEncji.cs*.
5. Zmień zawartość tego pliku, wpisując do niego dwa testy. Pierwszy z nich będzie próbował połączyć się z bazą danych, a drugi ma potwierdzać, że w bazie danych znajduje się osiem kategorii. Użyj poniższego kodu:

```
using Biblioteka.ModeleEncji; // NorthwindContext

namespace Northwind.TestyJednostkowe
{
   public class TestyModeluEncji
   {
      [Fact]
      public void TestPolaczeniaZBazaDanych()
      {
         using NorthwindContext db = new();
         Assert.True(db.Database.CanConnect());
```

Kup książkę Poleć książkę

```
        }

        [Fact]
        public void TestLiczbyKategorii()
        {
           using NorthwindContext db = new();

           int oczekiwane = 8;
           int faktyczne = db.Categories.Count();

           Assert.Equal(oczekiwane, faktyczne);
        }

        [Fact]
        public void ProductId1IsChaiTest()
        {
           using NorthwindContext db = new();

           string oczekiwane = "Chai";
           Product? produkt = db.Products.Find(keyValues: 1);
           string faktyczne = product?.ProductName ?? string.Empty;
           Assert.Equal(oczekiwane, faktyczne);
        }
    }
}
```

6. Uruchom testy jednostkowe:
   - Jeżeli używasz Visual Studio 2022, wybierz z menu pozycję *Test/Uruchom wszystkie testy*, a ich wyniki przejrzyj w okienku *Eksplorator testów*.
   - Jeżeli używasz Visual Studio Code, w oknie terminala dla projektu *Northwind.TestyJednostkowe* uruchom test, wydając polecenie `dotnet test`. Możesz też skorzystać z okienka *TESTOWANIE*, jeżeli masz zainstalowane rozszerzenie *C# Dev Kit*.
7. W wynikach testów powinna znaleźć się informacja, że oba testy zostały uruchomione i oba zakończyły się poprawnie. Jeżeli jakiś test zgłosi błąd, to spróbuj usunąć jego przyczynę. Na przykład, jeżeli używasz serwera SQLite, możesz sprawdzić, czy plik *Northwind.db* rzeczywiście znajduje się w katalogu rozwiązania (poziom wyżej względem katalogów projektów). Sprawdź ścieżkę do bazy danych zapisaną w pliku *northwindlog.txt*. Ta ścieżka powinna zostać tam wypisana trzy razy, dla każdego z trzech uruchomionych testów, co widać na poniższym wydruku:

```
Ścieżka do bazy danych: C:\cs12dotnet8\PraktyczneAplikacje\Northwind.db
Ścieżka do bazy danych: C:\cs12dotnet8\PraktyczneAplikacje\Northwind.db
dbug: 18/09/2023 14:20:16.712 RelationalEventId.CommandExecuting[20100]
(Microsoft.EntityFrameworkCore.Database.Command)
      Executing DbCommand [Parameters=[@__p_0='?' (DbType = Int32)],
CommandType='Text', CommandTimeout='30']
      SELECT "p"."ProductId", "p"."CategoryId", "p"."Discontinued",
"p"."ProductName", "p"."QuantityPerUnit", "p"."ReorderLevel",
"p"."SupplierId", "p"."UnitPrice", "p"."UnitsInStock", "p"."UnitsOnOrder"
      FROM "Products" AS "p"
```

```
      WHERE "p"."ProductId" = @__p_0
      LIMIT 1
Ścieżka do bazy danych: C:\cs12dotnet8\PraktyczneAplikacje\Northwind.db
dbug: 18/09/2023 14:20:16.832 RelationalEventId.CommandExecuting[20100]
(Microsoft.EntityFrameworkCore.Database.Command)
      Executing DbCommand [Parameters=[], CommandType='Text',
CommandTimeout='30']
      SELECT COUNT(*)
      FROM "Categories" AS "c"
```

Na zakończenie tego rozdziału przyjrzymy się metodom tworzenia oprogramowania dla sieci WWW, aby przygotować się do pracy z technologią ASP.NET Core Razor Pages, którą zajmiemy się w następnym rozdziale.

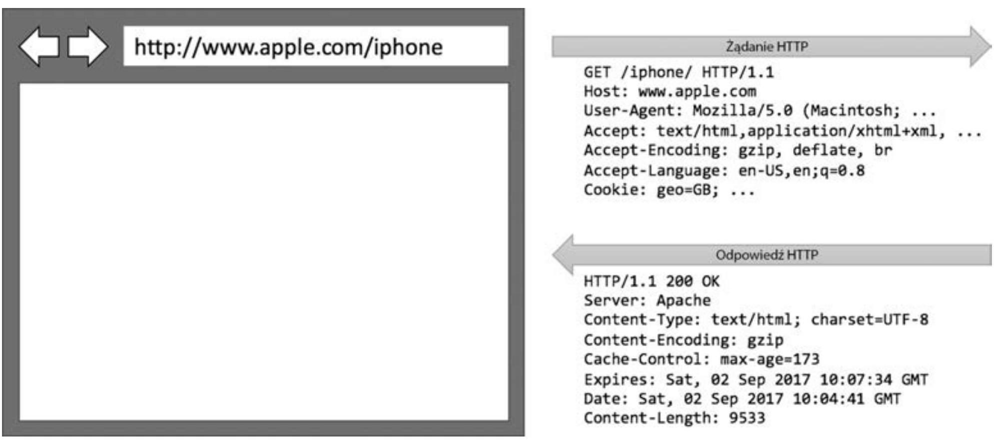
Tworzenie w sieci WWW

Tworzenie czegokolwiek w sieci WWW oznacza konieczność zaznajomienia się z protokołem **HTTP**, dlatego na początek omówię tę ważną technologię.

Protokół HTTP

Klient (lub **agent użytkownika**) komunikując się z serwerem WWW, wykonuje wywołania poprzez sieć, używając do tego protokołu **HTTP** (ang. *Hypertext Transfer Protocol*). Protokół HTTP stanowi techniczną bazę całej sieci WWW. Gdy zatem mówimy o aplikacjach i serwisach WWW, mamy na myśli, że komunikacja między klientem (często przeglądarką) a serwerem używa protokołu HTTP.

Klient wysyła żądanie HTTP dotyczące pewnego zasobu, takiego jak strona, identyfikując go adresem **URL** (ang. *Uniform Resource Locator*), a serwer odsyła odpowiedź HTTP. Całość odbywa się tak jak na rysunku 12.3.



Rysunek 12.3. Żądanie i odpowiedź HTTP

Kup ksi k Pole ksi k

Możesz skorzystać z przeglądarki Google Chrome lub innej, aby zapisać poszczególne żądania i odpowiedzi.

Wskazówka

Dobra praktyka: Google Chrome jest przeglądarką dostępną dla większej liczby systemów operacyjnych niż jakakolwiek inna. Ma też wbudowane zaawansowane narzędzia dla programistów, dlatego często wybierana jest jako podstawowa przeglądarka. Swoje aplikacje testuj zawsze w Chrome oraz przynajmniej w dwóch innych przeglądarkach, takich jak Firefox oraz Safari dla macOS i iOS. Mniej ważne jest testowanie stron w przeglądarce Microsoft Edge, ponieważ od 2019 r. działa ona z mechanizmem renderującym Chromium. Microsoft Internet Explorer jest już używany niemal wyłącznie w intranetach istniejących w niektórych organizacjach.

Składniki adresu URL

Każdy adres URL (Uniform Resource Locator) składa się z kilku elementów:

- **Schemat**: *http* (jawny tekst) lub *https* (szyfrowane).
- **Domena**: W przypadku produkcyjnej strony WWW lub serwisu domeną najwyższego poziomu (ang. *top-level domain*) może być *przykład.pl*. Można też stosować różne poddomeny, takie jak *www*, *praca* lub *extranet*. W trakcie prac rozwojowych nad projektem zazwyczaj adresem rozwijanych stron WWW i serwisów jest *localhost*.
- **Numer portu**: W przypadku produkcyjnych stron WWW lub serwisów używany jest port 80 dla protokołu HTTP albo 443 dla HTTPS. Te numery portów są zwykle wnioskowane na podstawie używanego schematu. W trakcie prac rozwojowych stosowane są przeważnie inne numery portów, takie jak na przykład 5000 lub 5001. W ten sposób odróżnia się od siebie poszczególne witryny i serwisy prowadzone pod lokalnym adresem *localhost*.
- **Ścieżka**: Względna ścieżka do zasobu, na przykład `/klienci/niemcy`.
- **Zapytanie**: Metoda przekazywania wartości parametrów, na przykład `?kraj=Niemcy&tekst=buty`.
- **Fragment**: Referencja elementu na stronie WWW wyznaczana za pomocą identyfikatora, na przykład `#spistresci`.

Uwaga

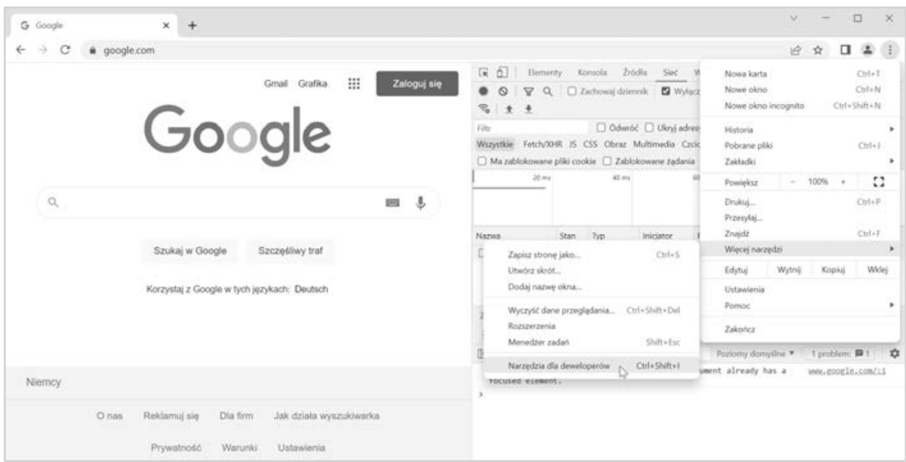
Adres URL jest podzbiorem większego adresu **URI** (ang. *Uniform Resource Identifier*). Adres URL określa, gdzie można znaleźć wybrany element, z kolei adres URI umożliwia identyfikację elementu za pomocą adresu URL lub **URN** (ang. *Uniform Resource Name*).

Kup książkę Poleć książkę

Używanie Google Chrome do wykonywania żądań HTTP

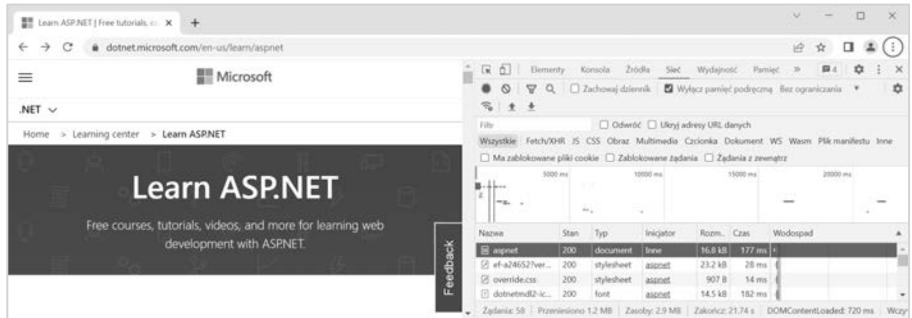
Sprawdźmy teraz, jak wykorzystać przeglądarkę Google Chrome do wykonywania żądań HTTP.

1. Uruchom Google Chrome.
2. Wybierz z menu pozycję *Więcej narzędzi/Narzędzia dla developerów*.
3. Kliknij kartę *Sieć*. Chrome powinna od razu zacząć rejestrować ruch sieciowy pomiędzy przeglądarką a dowolnym serwerem WWW (zwróć uwagę na czerwoną lampkę), tak jak na rysunku 12.4.



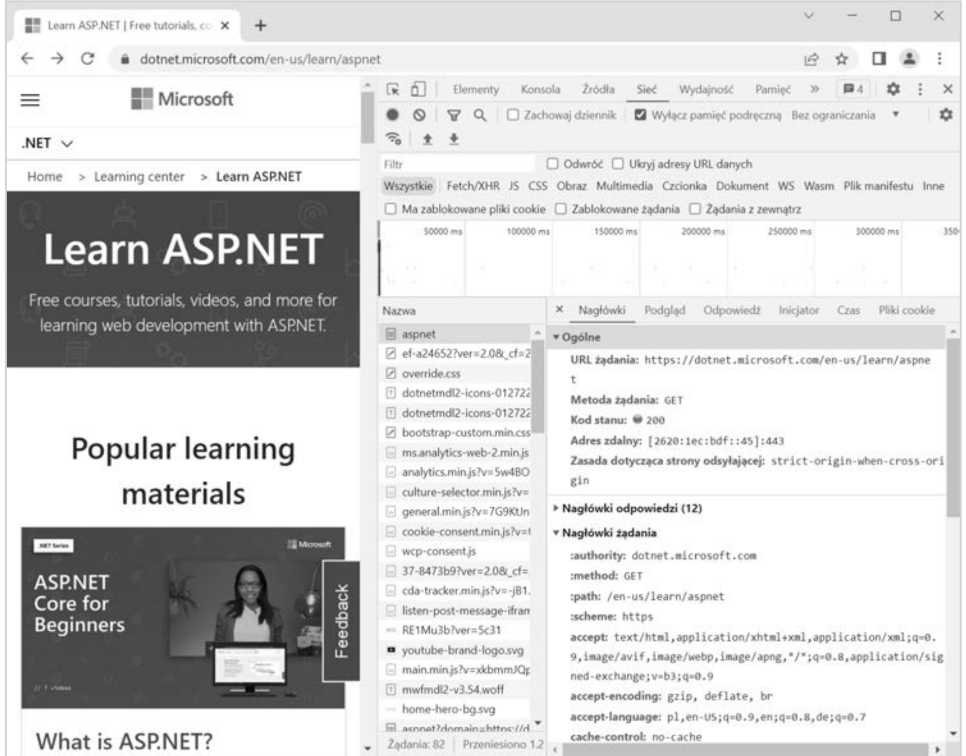
Rysunek 12.4. Narzędzia dla developerów w Google Chrome zapisują dane ruchu sieciowego

4. W pasku adresu Google Chrome wpisz adres URL witryny Microsoftu przeznaczonej do nauki korzystania z ASP.NET: *https://dotnet.microsoft.com/learn/aspnet*.
5. W okienku narzędzi dla programistów na liście zapisanych żądań kliknij pierwszą pozycję, w której w kolumnie *Typ* znajduje się wartość *dokument*, tak jak na rysunku 12.5.



Rysunek 12.5. Zapisane żądanie wyświetlone w oknie narzędzi dla developerów

6. Po prawej stronie kliknij kartę *Nagłówki*, a zobaczysz szczegóły wybranego żądania oraz odpowiedzi na nie, tak jak na rysunku 12.6.



Rysunek 12.6. Nagłówki żądania i odpowiedzi

Zwróć uwagę na następujące objaśnienia:

- **Metoda żądania** (ang. *Request Method*) to `GET`. Inne metody definiowane przez protokół HTTP to `POST`, `PUT`, `DELETE`, `HEAD` i `PATCH`.
- **Kod statusu** (ang. *Status Code*) to `200 OK`. Oznacza to, że serwer odszukał zasób żądany przez przeglądarkę i przesłał go w ciele odpowiedzi. Wśród innych kodów statusu znajdziemy też `301 Moved Permanently`, `400 Bad Request`, `401 Unauthorized` i `404 Not Found`.
- Sekcja **nagłówków żądania** (ang. *Request Headers*) zawiera:
  i. Nagłówek **Accept**, w której przeglądarka wymienia akceptowane formaty. W tym przypadku przeglądarka twierdzi, że może pracować z formatami HTML, XHTML, XML i pewnymi formatami obrazów, ale przyjmie też inne pliki oznaczone jako */*. Domyślną wagą dla formatów, zwaną też współczynnikiem jakości, jest wartość 1.0. Format XML otrzymał tu współczynnik jakości 0.9, co oznacza, że jest mniej pożądany od formatów HTML i XHTML. Wszystkie pozostałe typy plików otrzymały współczynnik jakości wynoszący 0.8, co oznacza, że są najmniej pożądanymi formatami danych.

Kup ksi k Pole ksi k

ii. Nagłówek **akceptowanych kodowań** (ang. *Accept-Encoding*) mówi nam, że przeglądarka przyjmie dane skompresowane algorytmami `GZIP`, `DEFLATE` lub `Brotli`.

iii. Nagłówek **accept-language**, w którym przeglądarka informuje serwer, jakie języki ludzkie będzie w stanie obsłużyć. Będzie to język polski (o domyślnym współczynniku jakości wynoszącym `1.0`), ale też angielski w wariancie amerykańskim (ze współczynnikiem jakości `0.9`) albo dowolny dialekt angielskiego (ze współczynnikiem jakości `0.8`).

- Sekcja **nagłówków odpowiedzi** (ang. *Response Headers*) i nagłówek **content-encoding** mówią nam, że serwer odesłał w odpowiedzi stronę HTML skompresowaną algorytmem GZIP, ponieważ wiedział, że klient może obsłużyć ten format. (Tej informacji nie widać na rysunku 12.6, gdyż brakuje na nim miejsca do rozwinięcie sekcji *Nagłówki odpowiedzi*).

7. Zamknij przeglądarkę Google Chrome.

Tworzenie oprogramowania dla sieci WWW po stronie klienta

Podczas tworzenia aplikacji WWW programista musi znać coś więcej niż tylko język C# i platformę .NET. Po stronie klienta (czyli przeglądarki) będziemy używali kombinacji następujących trzech elementów:

- **HTML5** — służy do definiowania treści i struktury strony WWW.
- **CSS3** — służy do definiowania stylów nakładanych na poszczególne elementy strony WWW.
- **JavaScript** — służy do zapisywania logiki biznesowej w ramach strony WWW, np. do sprawdzania poprawności danych wprowadzonych przez użytkownika albo do wywoływania funkcji w serwisach sieciowych w celu pobrania kolejnych danych używanych na stronie.

Mimo że HTML5, CSS3 i JavaScript są podstawowymi budulcami interfejsów użytkownika w sieci WWW, istnieje wiele bibliotek, które sprawiają, że tworzenie takich interfejsów jest znacznie bardziej produktywne. Można tu wymienić:

- **Bootstrap**, najpopularniejszy zestaw narzędzi frontendowych o otwartych źródłach.
- **SASS** i **LESS**, preprocesory stylów CSS.
- Język **TypeScript** zaprojektowany przez Microsoft, który pozwala na tworzenie bezpieczniejszego kodu.
- Biblioteki języka JavaScript: **jQuery**, **Angular**, **React** lub **Vue**.

Te technologie wyższego poziomu są na koniec przekształcane do jednej z trzech podstawowych technologii, które działają z wszystkimi nowoczesnymi przeglądarkami.

Kup ksi k Pole ksi k

W ramach procesu kompilowania i wdrażania swoich aplikacji najprawdopodobniej użyjesz takich narzędzi jak:

- **Node.js**, czyli serwerowy framework używający języka JavaScript.
- **NPM** (ang. *Node Package Manager*) lub **Yarn**, czyli menedżery pakietów działające po stronie klienta.
- **Webpack**, czyli popularne narzędzie do wiązania modułów, kompilowania, transformowania i wiązania ze sobą plików źródłowych witryny.

Praktyka i ćwiczenia

Sprawdź swoją wiedzę i wiadomości, odpowiadając na kilka prostych pytań. Zyskaj trochę doświadczenia w zakresie tematów omawianych w tym rozdziale.

Ćwiczenie 12.1 — sprawdź swoją wiedzę

1. Jak nazywała się pierwsza technologia Microsoftu umożliwiająca tworzenie dynamicznych stron WWW działających po stronie serwera i dlaczego jest ona użyteczna również dzisiaj?
2. Jak nazywają się dwa serwery WWW firmy Microsoft?
3. Na czym polegają różnice między mikroserwisami a nanoserwisami?
4. Czym jest Blazor?
5. Która wersja ASP.NET Core jako pierwsza nie mogła działać na platformie .NET Framework?
6. Co to jest agent użytkownika?
7. Jakie znaczenie dla programistów aplikacji sieciowych ma model komunikacji HTTP bazujący na żądaniach i odpowiedziach?
8. Nazwij i opisz cztery elementy adresu URL.
9. Jakie możliwości dają nam narzędzia dla deweloperów w przeglądarce?
10. Podaj trzy główne technologie używane przy tworzeniu klienckiej strony aplikacji WWW. Jakie jest ich zadanie?

Ćwiczenie 12.2 — znasz te skrótowce?

Co oznaczają poniższe skrótowce?

1. URI
2. URL
3. WCF
4. TLD
5. API

Kup ksi k Pole ksi k

6. SPA
7. CMS
8. Wasm
9. SASS
10. REST

Ćwiczenie 12.3 — dalsza lektura

Użyj linków udostępnionych na poniższej stronie, aby dowiedzieć się więcej na tematy poruszane w tym rozdziale:

https://github.com/markjprice/cs12dotnet8/blob/main/docs/book-links.md ↪#chapter-12---introducing-web-development-using-aspnet-core

Podsumowanie

W tym rozdziale:

- Przedstawiłem wprowadzenie do różnych modeli aplikacji, które można wykorzystać do tworzenia praktycznych aplikacji za pomocą języka C# i środowiska .NET.
- Przygotowaliśmy też dwie (lub cztery) biblioteki klas definiujących model encji umożliwiający pracę z bazą danych Northwind na serwerze SQL Server lub SQLite.

W kolejnych rozdziałach poznasz szczegóły tworzenia następujących projektów:

- proste strony WWW ze statycznym kodem HTML oraz z dynamicznym kodem Razor Pages;
- złożone witryny sieciowe budowane zgodnie ze wzorcem MVC (ang. *Model-View-Controller*);
- serwisy sieciowe, które można wywołać z dowolnej platformy zdolnej do generowania żądań HTTP, oraz witryny klienckie korzystające z tych serwisów;
- komponenty interfejsu użytkownika w technologii Blazor, które można hostować zarówno na serwerze, jak i w przeglądarce, a nawet wykorzystywać w aplikacjach dla urządzeń mobilnych i stacjonarnych.

Kup ksi k Pole ksi k

PROGRAM PARTNERSKI

GRUPY HELION

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

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

Nowa jakość programowania w Twoich rękach
Poznaj pełny potencjał C# 12 i .NET 8!

Microsoft może być dumny z języka C# i platformy .NET! Są one konsekwentnie rozwijane i wzbogacane, a z każdą kolejną wersją praca programisty staje się coraz efektywniejsza i bardziej satysfakcjonująca. Dzięki aktualnej wersji C# 12 i .NET 8 bez trudu będziesz tworzyć rozbudowane witryny internetowe czy aplikacje mobilne.

Ta książka jest kolejnym, starannie zaktualizowanym wydaniem cenionego i lubianego poradnika, dzięki któremu Twoja praca w języku C# stanie się przyjemna i wydajna. Znajdziesz tu liczne przykłady prezentujące nowe elementy .NET 8: aliasy typów i konstruktory podstawowe, zapewniające spójny i czytelny kod. Nauczysz się stosować klauzule ochronne i uproszczoną implementację pamięci podręcznej w ASP.NET Core 8. Poznasz też nową metodę kompilacji AOT, dzięki której publikowane serwisy zajmują mniej pamięci i szybciej się uruchamiają. Na zakończenie zaznajomisz się również z technologią Blazor Full Stack, będącą nowym, zunifikowanym modelem elastycznego projektowania aplikacji sieciowych.

Z tą książką nauczysz się:

- używać nowych funkcji języka C# 12
- stosować kompilację native AOT dla serwisów sieciowych z minimalnym API
- korzystać z technologii Blazor Full Stack, Razor Pages i innych funkcji ASP.NET Core
- tworzyć i aktualizować bazy danych w aplikacjach
- odczytywać dane i manipulować nimi za pomocą LINQ
- budować złożone serwisy za pomocą WebAPI lub Minimal API

Mark J. Price wielokrotnie otrzymywał tytuł MVP za pracę z serwerami i technologiami sieciowymi Microsoftu. Specjalizuje się w Microsoft DirectAccess i Always On VPN, regularnie publikuje artykuły na temat tych technologii. Kieruje zespołem inżynierów rozproszonych po całym kraju.

Cena: 179,00 zł