

Aplikacje i usługi na platformie .NET 7

Tworzenie praktycznych projektów opartych na programach Blazor, .NET MAUI, gRPC, GraphQL i innych zaawansowanych technologiach



Tytuł oryginału: Apps and Services with .NET 7: Build practical projects with Blazor, .NET MAUI, gRPC, GraphQL, and other enterprise technologies

Tłumaczenie: Andrzej Watrak

ISBN: 978-83-8322-716-0

Copyright © Packt Publishing 2022. First published in the English language under the title 'Apps and Services with .NET 7 – (9781801813433)'

Polish edition copyright © 2023 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 jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

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

Autor oraz 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/aplius>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

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 recenzentach.....	22
Przedmowa.....	23
ROZDZIAŁ 1	
Aplikacje i usługi oparte na platformie .NET	29
Podsumowanie treści książki	30
Kontynuacja edukacyjnej przygody	30
Czego się dowiesz z tej książki?	31
Moja filozofia uczenia się	31
Naprawianie moich błędów	31
Konwencje nazewnictwa projektów i numeracji portów	32
Traktowanie ostrzeżeń jako błędów	34
Technologie do tworzenia aplikacji i usług	35
Tworzenie witryn internetowych i aplikacji z użyciem platformy ASP.NET Core	35
Tworzenie usług internetowych i nie tylko	36
Windows Communication Foundation (WCF)	37
Podsumowanie rodzajów usług	37
Tworzenie aplikacji wyłącznie dla systemu Windows	39
Tworzenie aplikacji mobilnych i stacjonarnych dla różnych systemów operacyjnych	41
Opcje alternatywne do .NET MAUI	41
Przygotowanie środowiska programistycznego	43
Wybór odpowiedniego do nauki narzędzia i typu aplikacji	43
Tworzenie uniwersalnych aplikacji	46
Pobranie i instalacja środowiska Visual Studio 2022 dla systemu Windows	46
Pobranie i instalacja środowiska Visual Studio Code	48
Pliki z rozwiązaniami zadań	51
Wykorzystywane zasoby Azure	51

Jak używać analizatora, aby pisać lepszy kod	52
Ukrywanie ostrzeżeń	55
Korekta kodu	56
Co nowego w C# i .NET?	58
Utrzymanie platformy .NET	58
Wersje .NET Runtime i .NET SDK	59
Co nowego w C# 8 i .NET Core 3?	60
Co nowego w C# 9 i .NET 5?	64
Co nowego w C# 10 i .NET 6?	66
Co nowego w C# 11 i .NET 7?	69
Jak najlepiej wykorzystać repozytorium GitHub dla tej książki?	72
Zgłaszanie problemów podczas pracy z książką	72
Zgłaszanie opinii	73
Kody z rozwiązaniami zadań	73
Gdzie szukać pomocy?	73
Dokumentacja Microsoft	73
Narzędzie dotnet	73
Wyszukiwarka Google	75
Oficjalny blog .NET	75
Filmy Scotta Hanselmana	75
Nauka i praktyka	75
Ćwiczenie 1.1. Sprawdź swoją wiedzę	76
Ćwiczenie 1.2. Zbadaj temat	76
Podsumowanie	76

ROZDZIAŁ 2

Zarządzanie relacyjną bazą danych SQL Server	78
Nowoczesne bazy danych	78
Prosta relacyjna baza danych	79
Nawiązanie połączenia z bazą SQL Server	80
Instalacja i konfiguracja oprogramowania SQL Server	82
Tworzenie przykładowej bazy danych Northwind w systemie Windows	85
Konfiguracja usługi Azure SQL Database	86
Instalacja oprogramowania Azure SQL Edge w środowisku Docker	91
Przetwarzanie danych za pomocą języka Transact-SQL	94
Typy danych w języku T-SQL	94
Język DML	96
Język DDL	99
Zarządzanie bazą SQL Server za pomocą niskopoziomowych interfejsów API	100
Typy danych w ADO.NET	101
Utworzenie aplikacji konsolowej opartej na sterowniku ADO.NET	102

Uruchamianie zapytań i odczytywanie danych za pomocą sterownika ADO.NET	107
Operacje asynchroniczne w sterowniku ADO.NET	109
Uruchamianie procedur składowanych za pomocą sterownika ADO.NET	110
Zarządzanie bazą SQL Server z wykorzystaniem technologii EF Core	113
Informacje o EF Core	114
Konstruowanie modelu jednostek z wykorzystaniem istniejącej bazy danych	114
Konfiguracja narzędzia dotnet-ef	114
Definiowanie modeli jednostek EF Core	115
Definiowanie modelu jednostek EF Core z użyciem konwencji	115
Definiowanie modelu z użyciem atrybutów adnotacyjnych	116
Definiowanie modelu z użyciem interfejsu Fluent API	118
Inicjowanie bazy danych za pomocą interfejsu Fluent API	118
Definiowanie modelu bazy Northwind	119
Wysyłanie zapytań do modelu Northwind	124
Mapowanie hierarchii dziedziczenia klas z wykorzystaniem technologii EF Core	127
Strategia TPH	128
Strategia TPT	129
Strategia TPC	130
Konfigurowanie strategii mapowania hierarchii	131
Przykład zastosowania strategii mapowania hierarchii klas	132
Tworzenie współdzielonego modelu jednostek danych	139
Tworzenie biblioteki klas dla modelu jednostek bazy SQL Server	140
Tworzenie biblioteki klas dla kontekstu danych bazy SQL Server	141
Tworzenie jednostek za pomocą wyliczanych właściwości	144
Tworzenie projektu testującego integrację bibliotek klas	146
Tworzenie testów jednostkowych modelu jednostek	146
Wykonanie testów jednostkowych w środowisku Visual Studio 2022	148
Wykonanie testów jednostkowych w środowisku Visual Studio Code	148
Zwalnianie zasobów baz danych	149
Zwalnianie zasobów Azure	149
Zwalnianie zasobów Docker	149
Nauka i praktyka	150
Ćwiczenie 2.1. Sprawdź swoją wiedzę	150
Ćwiczenie 2.2. Porównanie wydajności sterownika ADO.NET i technologii EF Core	151
Ćwiczenie 2.3. Zbadaj temat	151
Ćwiczenie 2.4. Zbadaj oprogramowanie Dapper	151
Podsumowanie	152

ROZDZIAŁ 3

Zarządzanie bazą NoSQL z użyciem Azure Cosmos DB	153
Bazy NoSQL	153
Baza Cosmos DB i jej interfejsy API	154
Modelowanie dokumentów	155
Poziomy spójności	157
Hierarchia komponentów	158
Apro wizacja przepływności	159
Strategie partycjonowania	160
Projektowanie magazynu danych	160
Przenoszenie danych do bazy Cosmos DB	161
Tworzenie zasobów bazy Cosmos DB	161
Tworzenie zasobów z użyciem emulatora	161
Tworzenie zasobów w portalu Azure	167
Tworzenie zasobów za pomocą aplikacji .NET	171
Przetwarzanie danych za pomocą interfejsu Core (SQL) API	176
Wykonywanie operacji CRUD za pomocą interfejsu Core (SQL) API	176
Zapytania SQL	185
Programowanie serwerów	188
Przetwarzanie danych grafowych za pomocą interfejsu Gremlin API	190
Usuwanie zasobów Azure	190
Nauka i praktyka	190
Ćwiczenie 3.1. Sprawdź swoją wiedzę	190
Ćwiczenie 3.2. Przecwicz modelowanie i partycjonowanie danych	191
Ćwiczenie 3.3. Zbadaj temat	191
Ćwiczenie 3.4. Zbadaj bazy NoSQL	191
Ćwiczenie 3.5. Pobierz ścią gawki	192
Ćwiczenie 3.6. Przeczytaj podręcznik do interfejsu Gremlin API	192
Podsumowanie	192

ROZDZIAŁ 4

Ocena wydajności kodu, wielozadaniowość i współbieżność	193
Procesy, wątki, zadania	193
Monitorowanie wydajności aplikacji i wykorzystania zasobów	195
Ocena efektywności typów danych	195
Monitorowanie wydajności kodu i zajętości pamięci za pomocą przestrzeni Diagnostics	196
Monitorowanie wydajności przetwarzania ciągów znaków	199
Monitorowanie wydajności kodu i zajętości pamięci za pomocą pakietu Benchmark.NET	201

Asynchroniczne wykonywanie zadań	205
Synchroniczne wykonywanie operacji	205
Asynchroniczne wykonywanie operacji za pomocą zadań	208
Oczekiwanie na wykonanie zadań	209
Sekwencyjne wykonanie innego zadania	210
Zadania zagnieżdżone i podrzędne	212
Opakowanie zadania w obiekcie	213
Synchronizacja dostępu do współdzielonych zasobów	215
Synchronizacja dostępu do zasobu	215
Zakładanie na muszlę blokady dostępu na wyłączność	217
Synchronizacja zdarzeń	220
Kodowanie atomicznych operacji procesora	221
Inne techniki synchronizacji dostępu	222
Instrukcje async i await	222
Poprawa responsywności aplikacji konsolowej	223
Operacje na strumieniach asynchronicznych	224
Poprawa responsywności aplikacji graficznej	225
Poprawa skalowalności aplikacji i usług internetowych	230
Popularne typy obsługujące wielozadaniowość	231
Instrukcja await w bloku catch	231
Nauka i praktyka	231
Ćwiczenie 4.1. Sprawdź swoją wiedzę	231
Ćwiczenie 4.2. Zbadaj temat	232
Ćwiczenie 4.3. Dowiedz się więcej o programowaniu równoległym	232
Podsumowanie	232

ROZDZIAŁ 5

Korzystanie z popularnych zewnętrznych bibliotek	234
Najpopularniejsze zewnętrzne biblioteki	234
Co opisuję w swoich książkach?	235
Co powinienem opisać w swoich książkach?	236
Przetwarzanie obrazów	236
Tworzenie czarno-białych miniatur	236
Biblioteki do rysowania i przekształcania obrazów w internecie	240
Rejestrowanie działań	240
Strukturalne dane o zdarzeniach	240
Zlewy	241
Rejestrowanie działań w konsoli i cyklicznym pliku	241
Mapowanie obiektów	243
Testowanie konfiguracji mapowania	244
Mapowanie obiektów na bieżąco między modelami	248

Kodowanie asercji w testach jednostkowych	250
Asercja testująca ciąg znaków	251
Asercja testująca kolekcję lub tablicę	252
Asercja testująca datę lub czas	252
Sprawdzanie poprawności danych	253
Wbudowane weryfikatory	254
Niestandardowa weryfikacja	254
Dostosowywanie komunikatów	254
Definiowanie modelu i weryfikatora	254
Test weryfikatora	256
Integracja z ASP.NET Core	259
Generowanie plików PDF	260
Utworzenie biblioteki klas do generowania plików PDF	260
Utworzenie aplikacji generującej plik PDF	263
Nauka i praktyka	266
Ćwiczenie 5.1. Sprawdź swoją wiedzę	266
Ćwiczenie 5.2. Zbadaj temat	266
Podsumowanie	267

ROZDZIAŁ 6

Dynamiczne monitorowanie i modyfikowanie kodu	268
Refleksje i atrybuty	268
Numerowanie wersji zestawu	269
Odczytanie metadanych zestawu	269
Tworzenie niestandardowych atrybutów	272
Standardowe typy generowane przez kompilator i ich elementy	274
Oznaczanie przestarzałych typów i ich elementów	274
Dynamiczne ładowanie zestawów i wywoływanie metod	275
Do czego jeszcze służą refleksje?	280
Drzewa wyrażeń	280
Komponenty drzewa wyrażeń	281
Uruchomienie najprostszego drzewa wyrażeń	282
Generatory kodu źródłowego	283
Implementacja najprostszego generatora	283
Co jeszcze można uzyskać za pomocą generatora kodu źródłowego?	287
Nauka i praktyka	288
Ćwiczenie 6.1. Sprawdź swoją wiedzę	288
Ćwiczenie 6.2. Zbadaj temat	288
Podsumowanie	289

ROZDZIAŁ 7

Daty, godziny i internacjonalizacja	290
Operacje na datach i czasie	290
Definiowanie daty i czasu	291
Formatowanie daty i czasu	292
Działania na datach i czasie	293
Mikro- i nanosekundy	294
Globalizacja daty i czasu	294
Lokalizowanie typu wyliczeniowego DayOfWeek	296
Działania wyłącznie na dacie lub czasie	297
Operacje na strefach czasowych	298
Typy DateTime i TimeZoneInfo	298
Badanie typów DateTime i TimeZoneInfo	300
Operacje na kulturach	305
Identyfikacja i zmiana kultury	306
Tymczasowe stosowanie niezmiennych kultur	312
Lokalizowanie interfejsu użytkownika	313
Definiowanie i ładowanie zasobów	314
Test globalizacji i lokalizacji	317
Nauka i praktyka	323
Ćwiczenie 7.1. Sprawdź swoją wiedzę	323
Ćwiczenie 7.2. Zbadaj temat	324
Ćwiczenie 7.3. Ekspert Jon Skeet radzi	324
Podsumowanie	324

ROZDZIAŁ 8

Ochrona danych i aplikacji	325
Terminologia bezpieczeństwa danych	326
Klucze i ich długości	326
Wektory inicjujące i bloki danych	327
Sól	328
Generowanie klucza i wektora inicjującego	328
Szyfrowanie i deszyfrowanie danych	329
Szyfrowanie symetryczne za pomocą algorytmu AES	330
Skracanie danych	336
Skracanie danych z użyciem popularnego algorytmu SHA256	337
Podpisywanie danych	340
Podpisywanie danych za pomocą algorytmów SHA256 i RSA	341
Generowanie liczb losowych	343
Generowanie liczb losowych w grach i prostych aplikacjach	344
Generowanie liczb losowych w kryptografii	345

Uwierzytelnianie i autoryzowanie użytkowników	346
Mechanizmy uwierzytelnienia i autoryzacji	347
Implementacja uwierzytelnienia i autoryzacji	349
Ochrona funkcjonalności aplikacji	352
Uwierzytelnienie i autoryzacja w praktyce	353
Nauka i praktyka	354
Ćwiczenie 8.1. Sprawdź swoją wiedzę	354
Ćwiczenie 8.2. Zabezpiecz dane przez zaszyfrowanie ich i podpisanie	354
Ćwiczenie 8.3. Odszyfruj dane	355
Ćwiczenie 8.4. Zbadaj temat	355
Ćwiczenie 8.5. Zapoznaj się z zaleceniami Microsoft dotyczącymi szyfrowania danych	355
Podsumowanie	355

ROZDZIAŁ 9

Tworzenie i zabezpieczanie usług internetowych

z użyciem minimalistycznych interfejsów API

Tworzenie usług internetowych	
z użyciem interfejsów ASP.NET Core Minimal API	357
Mapowanie tras za pomocą interfejsów Minimal API	359
Mapowanie parametrów	359
Zwracane wartości	360
Dokumentowanie usługi	360
Przygotowanie projektu usługi ASP.NET Core Web API	361
Testowanie usług internetowych	
z użyciem rozszerzenia Visual Studio Code	367
Wykluczanie tras z dokumentacji OpenAPI	371
Omijanie zasady tego samego pochodzenia przez współdzielenie zasobów	371
Rejestrowanie zapytań HTTP w usłudze internetowej	372
Utworzenie klienckiego skryptu JavaScript	373
Utworzenie klienckiej aplikacji .NET	377
Współdzielenie zasobów CORS	380
Włączenie współdzielenia CORS dla określonych tras	382
Opcje współdzielenia CORS	383
Ochrona przed atakami DoS przez ograniczanie liczby zapytań	383
Ograniczanie liczby zapytań za pomocą pakietu <code>AspNetCoreRateLimit</code>	384
Utworzenie aplikacji klienckiej objętej limitem zapytań	387
Ograniczanie liczby zapytań	
za pomocą komponentu pośredniczącego ASP.NET Core	392
Usługi kontroli tożsamości	394
Autoryzacja z użyciem tokenu JWT	395
Uwierzytelnianie z użyciem tokenu JWT	395

Nauka i praktyka	398
Ćwiczenie 9.1. Sprawdź swoją wiedzę	398
Ćwiczenie 9.2. Poznaj zasady Microsoft dotyczące tworzenia interfejsów HTTP API	399
Ćwiczenie 9.3. Zbadaj temat	399
Podsumowanie	399
ROZDZIAŁ 10	
Udostępnianie danych w internecie za pomocą OData	400
Protokół OData	400
Standard OData	401
Zapytania OData	401
Tworzenie usługi internetowej obsługującej protokół OData	402
Zdefiniowanie modeli OData dla modeli jednostek EF Core	404
Testy modeli OData	406
Utworzenie i testowanie kontrolerów OData	407
Testowanie usługi OData za pomocą rozszerzenia Visual Studio Code	410
Wysyłanie zapytań do usługi OData za pomocą rozszerzenia REST Client	412
Weryfikacja wydajności zapytań OData na podstawie dziennika	416
Wersjonowanie kontrolerów OData	417
Dodawanie, modyfikowanie i usuwanie jednostek	419
Tworzenie klienta usługi OData	422
Wywoływanie usług w witrynie Northwind MVC	424
Ponowna analiza początkowego zapytania	427
Nauka i praktyka	429
Ćwiczenie 10.1. Sprawdź swoją wiedzę	429
Ćwiczenie 10.2. Zbadaj temat	429
Podsumowanie	429
ROZDZIAŁ 11	
Łączenie źródeł danych za pomocą GraphQL	430
Język GraphQL	430
Format zapytania GraphQL	431
Inne funkcjonalności języka GraphQL	433
Platforma ChilliCream GraphQL	433
Tworzenie usługi obsługującej język GraphQL	434
Zdefiniowanie schematu GraphQL	435
Tworzenie i uruchamianie zapytań GraphQL	438
Nadawanie nazw zapytaniom GraphQL	439
Konwencja nazewnictwa pól	439

Definiowanie zapytań GraphQL w modelu EF Core	441
Implementacja obsługi technologii EF Core	441
Wysyłanie zapytań GraphQL do bazy Northwind	443
Tworzenie klienta .NET dla usługi GraphQL	448
Odpowiedzi na zapytania GraphQL	449
Rozszerzenie REST Client jako klient usługi GraphQL	449
Projekt ASP.NET Core MVC jako klient usługi GraphQL	452
Test klienta .NET	459
Pakiet Strawberry Shake	460
Implementowanie mutacji w języku GraphQL	464
Nauka i praktyka	468
Ćwiczenie 11.1. Sprawdź swoją wiedzę	468
Ćwiczenie 11.2. Zbadaj temat	469
Ćwiczenie 11.3. Przećwicz tworzenie klientów .NET	469
Podsumowanie	469

ROZDZIAŁ 12

Tworzenie wydajnych mikrousług za pomocą gRPC	470
Platforma gRPC	471
Jak funkcjonuje platforma gRPC?	471
Definiowanie kontraktów za pomocą plików .proto	471
Zalety platformy gRPC	472
Wady platformy gRPC	472
Rodzaje metod gRPC	472
Pakiety Microsoft gRPC	473
Utworzenie usługi i klienta gRPC	474
Utworzenie prostej usługi gRPC	474
Utworzenie prostego klienta gRPC	477
Test usługi i klienta gRPC	481
Implementacja usługi gRPC dla modelu EF Core	482
Implementacja usługi gRPC	482
Implementacja klienta gRPC	485
Uzyskiwanie metadanych o zapytaniach i odpowiedziach	488
Limit czasowy zwiększający stabilność usługi	489
Implementacja transkodowania gRPC JSON	492
Włączenie transkodowania gRPC JSON	493
Test transkodowania gRPC JSON	494
Porównanie transkodowania gRPC JSON i technologii gRPC-Web	496

Nauka i praktyka	496
Ćwiczenie 12.1. Sprawdź swoją wiedzę	496
Ćwiczenie 12.2. Zbadaj temat	497
Podsumowanie	497

ROZDZIAŁ 13

Rozgłaszanie komunikatów w czasie rzeczywistym z użyciem SignalR 498

Technologia SignalR	498
Historia komunikacji w czasie rzeczywistym w internecie	499
AJAX	499
WebSocket	499
Pakiet SignalR	500
Azure SignalR Service	500
Definiowanie sygnatur metod	501
Tworzenie usługi komunikacji w czasie rzeczywistym z użyciem SignalR	502
Zdefiniowanie współdzielonych modeli	502
Utworzenie koncentratora po stronie serwera	503
Tworzenie klienta internetowego z użyciem biblioteki SignalR JavaScript	507
Dodanie strony czatu do projektu MVC	508
Test czatu	512
Tworzenie klienckiej aplikacji konsolowej .NET	515
Utworzenie klienta .NET dla usługi SignalR	515
Test konsolowej aplikacji .NET	517
Strumieniowanie danych z użyciem SignalR	518
Utworzenie koncentratora strumieniowego	518
Utworzenie konsolowej strumieniowej aplikacji .NET	519
Test usługi strumieniowej i klienta	522
Nauka i praktyka	523
Ćwiczenie 13.1. Sprawdź swoją wiedzę	523
Ćwiczenie 13.2. Zbadaj temat	524
Podsumowanie	524

ROZDZIAŁ 14

Tworzenie nanousług bezserwerowych z użyciem funkcji Azure 525

Funkcje Azure	525
Wyzwalacze i powiązania funkcji Azure	526
Wyrażenia NCRONTAB	527
Wersje środowisk i języki funkcji Azure	532
Modele hostingu funkcji Azure	533
Plany hostingu	533

Wymagania magazynu Azure	534
Lokalne testowanie funkcji za pomocą Azurite	534
Poziomy autoryzacji funkcji Azure	535
Wstrzykiwanie zależności w funkcjach Azure	535
Instalacja Azure Functions Core Tools	536
Tworzenie projektu funkcji Azure	536
Tworzenie projektu w środowisku Visual Studio 2022	536
Tworzenie projektu w środowisku Visual Studio Code	537
Tworzenie projektu za pomocą wiersza poleceń	539
Przegląd projektu funkcji Azure	540
Implementacja prostej funkcji	542
Test funkcji	543
Reagowanie na zdarzenia czasomierza i zasobów	545
Implementacja funkcji wyzwalanej czasomierzem	545
Test funkcji wywoływanej przez czasomierz	548
Implementacja funkcji operującej na kolejkach i obiektach BLOB	552
Test funkcji operującej na kolejce i obiektach BLOB	558
Publikowanie projektu funkcji Azure w chmurze	560
Publikowanie projektu za pomocą środowiska Visual Studio 2022	560
Publikowanie projektu za pomocą środowiska Visual Studio Code	564
Usuwanie zasobów funkcji Azure	565
Nauka i praktyka	565
Ćwiczenie 14.1. Sprawdź swoją wiedzę	566
Ćwiczenie 14.2. Zbadaj temat	566
Podsumowanie	566

ROZDZIAŁ 15

Tworzenie interfejsów internetowych z użyciem ASP .NET Core	567
Tworzenie witryny opartej na wzorcu ASP.NET Core MVC	567
Tworzenie witryny ASP.NET Core MVC	568
Eksploracja domyślnej witryny ASP.NET Core MVC	569
Rejestracja użytkowników	571
Przegląd struktury projektu MVC	572
Odwołanie do biblioteki klas EF Core i kontekstu bazy danych	574
Definiowanie interfejsu użytkownika za pomocą widoków Razor	575
Widoki Razor	575
Prototypowanie strony z wykorzystaniem biblioteki Bootstrap	577
Składnia i wyrażenia języka Razor	584
Metody HTML Helpers	584
Zdefiniowanie silnie typowanego widoku Razor	585

Lokalizowanie i globalizowanie witryny za pomocą platformy ASP.NET Core	588
Utworzenie plików zasobów	589
Lokalizowanie widoków Razor za pomocą wstrzykniętego lokalizatora	591
Nagłówki Accept-Language	595
Definiowanie interfejsu użytkownika za pomocą metod Tag Helpers	595
Porównanie metod HTML Helpers i Tag Helpers	596
Metody Tag Helpers dotyczące odnośników	596
Metody Tag Helpers dotyczące bufora	601
Metoda Tag Helper dotycząca środowiska	604
Omijanie buforowania z użyciem metod Tag Helpers	606
Metody Tag Helpers dotyczące formularza	606
Nauka i praktyka	610
Ćwiczenie 15.1. Sprawdź swoją wiedzę	610
Ćwiczenie 15.2. Przećwicz tworzenie interfejsu użytkownika z użyciem biblioteki Bootstrap	610
Ćwiczenie 15.3. Zbadaj temat	611
Podsumowanie	611

ROZDZIAŁ 16

Tworzenie komponentów internetowych z użyciem Blazor WebAssembly 612

Platforma Blazor	612
Modele hostingu w platformie Blazor	613
Scenariusze wdrażania aplikacji w modelu Blazor WebAssembly	614
Analityczny komponent przeglądarki w modelu Blazor WebAssembly	614
Izolowanie kodów CSS i JavaScript	615
Komponenty platformy Blazor	615
Kierowanie zapytań do komponentów Blazor	616
Przekazywanie parametrów ścieżki	618
Bazowa klasa komponentów	620
Odnosiące do komponentów Blazor	621
Utworzenie komponentów Blazor	622
Utworzenie i przetestowanie komponentu paska postępu	627
Utworzenie i przetestowanie komponentu okna dialogowego	629
Utworzenie i przetestowanie komponentu alarmu	632
Utworzenie komponentu danych	635
Utworzenie komponentu strony z powiązaną ścieżką	636
Utworzenie usługi internetowej pobierającej dane z bazy	638
Pobieranie z usługi internetowej danych dla komponentu	640
Implementacja bufora w lokalnej pamięci przeglądarki	644
Interakcje z modułami JavaScript	644
Utworzenie usługi wykorzystującej pamięć lokalną	645

Utworzenie aplikacji PWA	651
Implementacja trybu offline w aplikacji PWA	655
Nauka i praktyka	656
Ćwiczenie 16.1. Sprawdź swoją wiedzę	656
Ćwiczenie 16.2. Przecwicz tworzenie komponentów Blazor	656
Ćwiczenie 16.3. Przecwicz tworzenie usługi interakcji z bazą danych	657
Ćwiczenie 16.4. Zbadaj temat	657
Podsumowanie	657

ROZDZIAŁ 17

Otwarte biblioteki komponentów Blazor 658

Otwarte biblioteki komponentów Blazor	658
Komponenty biblioteki Radzen Blazor	659
Aktywacja komponentów reprezentujących okna dialogowe, powiadomienia, menu kontekstowe i dymki	662
Komponenty reprezentujące dymki i menu kontekstowe	663
Komponenty reprezentujące powiadomienia i okna dialogowe	665
Utworzenie usługi internetowej opartej na bazie Northwind	668
Komponenty reprezentujące zakładki, obrazy i ikony	669
Komponent reprezentujący edytor HTML	674
Komponent reprezentujący wykres	676
Komponent reprezentujący formularz	680
Test komponentu strony Pracownicy	688
Nauka i praktyka	690
Ćwiczenie 17.1. Sprawdź swoją wiedzę	690
Ćwiczenie 17.2. Zbadaj bibliotekę komponentów MudBlazor	691
Ćwiczenie 17.3. Zbadaj temat	691
Podsumowanie	691

ROZDZIAŁ 18

Tworzenie aplikacji mobilnych i stacjonarnych z użyciem .NET MAUI 693

Język XAML	694
Prostszy kod dzięki XAML	694
Przestrzenie nazw w platformie .NET MAUI	696
Konwertery typów danych	697
Formanty platformy .NET MAUI	697
Rozszerzenia języka	698
Platforma .NET MAUI	699
Kodowanie przede wszystkim mobilne i chmurowe	700
Ręczna instalacja platformy .NET MAUI	700

Komponenty interfejsu graficznego .NET MAUI	702
Moduły obsługi	704
Tworzenie kodu dla określonego systemu operacyjnego	704
Tworzenie aplikacji mobilnych i stacjonarnych z użyciem platformy .NET MAUI	705
Utworzenie wirtualnego urządzenia Android do lokalnego testowania aplikacji	705
Włączenie trybu programisty w systemie Windows	707
Utworzenie rozwiązania .NET MAUI	707
Dodanie nawigacji i stron z treścią	712
Implementacja nowych stron	718
Współdzielenie zasobów	720
Definiowanie współdzielonych zasobów na poziomie aplikacji	721
Odwołania do współdzielonych zasobów	722
Dynamiczna wymiana zasobów	722
Wiązanie danych	727
Wiązanie elementów	727
Wzorzec MVVM	728
Interfejs INotifyPropertyChanged	729
Klasa ObservableCollection	731
Utworzenie modelu widoku z dwukierunkowym wiązaniem danych	731
Utworzenie widoków listy i szczegółów klientów	735
Test aplikacji .NET MAUI	741
Komunikacja między aplikacją mobilną a usługą internetową	744
Utworzenie usługi internetowej opartej na interfejsach Minimal API	744
Przystosowanie usługi do niezabezpieczonych połączeń	748
Lokalne połączenie z usługą na potrzeby testów	749
Konfiguracja niezabezpieczonych połączeń w aplikacji dla systemu iOS	749
Konfiguracja niezabezpieczonych połączeń w aplikacji dla systemu Android	750
Odczytywanie danych klientów z usługi internetowej	751
Nauka i praktyka	754
Ćwiczenie 18.1. Sprawdź swoją wiedzę	754
Ćwiczenie 18.2. Zbadaj temat	755
Podsumowanie	755

ROZDZIAŁ 19**Integracja aplikacji .NET MAUI z Blazor i natywnymi platformami 756**

Tworzenie hybrydowych aplikacji .NET MAUI Blazor	757
Utworzenie projektu .NET MAUI Blazor	758
Utworzenie powłoki i stron .NET MAUI	759
Utworzenie usługi internetowej opartej na interfejsach Minimal API	764
Przystosowanie aplikacji .NET MAUI do niezabezpieczonych połączeń	767
Implementacja wzorca MVVM	768
Odczytanie kategorii produktów z usługi internetowej	773
Interakcje z natywnymi platformami	777
Korzystanie z systemowego schowka	778
Wybieranie plików z lokalnego systemu	781
Utworzenie nowych okien	788
Uzyskanie informacji o urządzeniu	790
Integracja z paskiem menu urządzenia stacjonarnego	794
Wyskakujące powiadomienia	797
Zewnętrzne biblioteki formantów	798
Nauka i praktyka	800
Ćwiczenie 19.1. Sprawdź swoją wiedzę	800
Ćwiczenie 19.2. Przeglądaj przykładowe kody	800
Ćwiczenie 19.3. Zbadaj temat	801
Podsumowanie	801

ROZDZIAŁ 20**Konkurs na projekt narzędzia ankietowego 802**

Czego się nauczyłeś z tej książki?	802
Opisane technologie	802
Dlaczego właśnie projekt narzędzia ankietowego?	803
Jaki sposób uczenia się jest najlepszy?	804
Cechy dobrego projektu edukacyjnego	804
Inne pomysły na projekty	805
Funkcjonalności narzędzi ankietowych i sondażowych	805
Rodzaje pytań	806
Sondaże i quizy	806
Analiza odpowiedzi	807
Jakie są wymagania dotyczące narzędzia?	807
Przegląd minimalnych wymagań	807
Propozycje dodatkowych wymagań	809
Reklamuj swoje umiejętności	811
Podsumowanie	812

ROZDZIAŁ 21

Epilog	813
Drugie wydanie książki w listopadzie 2023 r.	813
Kolejne kroki w podróży po języku C# i platformie .NET	813
Powodzenia!	814

DODATEK

Odpowiedzi na pytania „Sprawdź swoją wiedzę”	815
Rozdział 1., „Aplikacje i usługi oparte na platformie .NET”	815
Rozdział 2., „Zarządzanie relacyjną bazą danych SQL Server”	817
Rozdział 3., „Zarządzanie bazą NoSQL z użyciem Azure Cosmos DB”	819
Rozdział 4., „Ocena wydajności, wielozadaniowość i współbieżność”	820
Rozdział 5., „Korzystanie z popularnych zewnętrznych bibliotek”	821
Rozdział 6., „Dynamiczne monitorowanie i modyfikowanie kodu”	822
Rozdział 7., „Daty, godziny i internacjonalizacja”	824
Rozdział 8., „Ochrona danych i aplikacji”	826
Rozdział 9., „Tworzenie i zabezpieczanie usług internetowych z użyciem minimalistycznych interfejsów API”	827
Rozdział 10., „Udostępnianie danych w internecie za pomocą OData”	829
Rozdział 11., „Łączenie źródeł danych za pomocą GraphQL”	830
Rozdział 12., „Tworzenie wydajnych mikrousług za pomocą gRPC”	832
Rozdział 13., „Rozgłaszanie komunikatów w czasie rzeczywistym z użyciem SignalR”	832
Rozdział 14., „Tworzenie nanousług bezserwerowych z użyciem funkcji Azure”	833
Rozdział 15., „Tworzenie interfejsów internetowych z użyciem ASP .NET Core”	835
Rozdział 16., „Tworzenie komponentów internetowych z użyciem Blazor WebAssembly”	837
Rozdział 17., „Otwarte biblioteki komponentów Blazor”	839
Rozdział 18., „Tworzenie aplikacji mobilnych i stacjonarnych z użyciem .NET MAUI”	840
Rozdział 19., „Integracja aplikacji .NET MAUI z Blazor i natywnymi platformami”	842
Skorowidz	845

Ocena wydajności kodu, wielozadaniowość i współbieżność

Rozdział

4

Ten rozdział jest poświęcony jednoczesnemu wykonywaniu wielu operacji przez aplikację, aby praca jej użytkownika była bardziej wydajna, skalowalna i efektywna.

W tym rozdziale są opisane następujące zagadnienia:

- procesy, wątki, zadania,
- monitorowanie wydajności aplikacji i wykorzystania zasobów,
- asynchroniczne wykonywanie zadań,
- synchronizacja dostępu do współdzielonych zasobów,
- instrukcje `async` i `await`.

Procesy, wątki, zadania

Proces to na przykład aplikacja konsolowa, taka jak jedna z utworzonych wcześniej, której są przydzielane zasoby, m.in. pamięć i wątki.

Wątek to kod wykonywany instrukcja po instrukcji. Domyślnie proces ma tylko jeden wątek, co może powodować problemy, gdy kilka zadań musi być wykonanych w tym samym czasie. Wątki są również wykorzystywane do kontrolowania różnych rzeczy, na przykład uwierzytelnionego w danym czasie użytkownika lub zasad internacjonalizacji właściwych dla bieżącego języka i regionu.

Większość nowoczesnych systemów operacyjnych, w tym Windows, wykorzystuje **wielozadaniowość z wywłaszczaniem** (ang. *preemptive multitasking*), która symuluje równoległe wykonywanie zadań. Polega ona na przydzielaniu czasu procesora każdemu wątkowi, jednemu po drugim. Procesor zawiesza bieżący wątek po upływie przydzielonego mu czasu, po czym wykonuje następny wątek przez zadany okres.

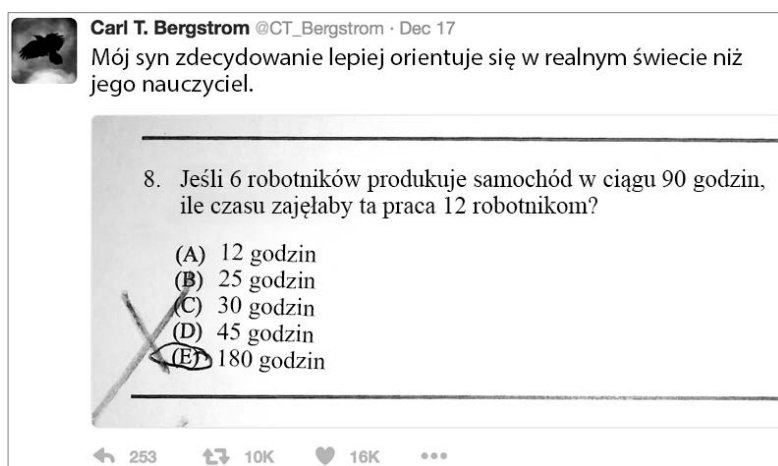
System Windows podczas przełączania umieszczonych w kolejce wątków zapisuje ich konteksty, a następnie ponownie je ładuje. Wymaga to zarówno czasu, jak i zasobów.

Jeżeli programista musi zaimplementować niewielką liczbę złożonych operacji i chce mieć nad nimi pełną kontrolę, może utworzyć kilka instancji klasy `Thread` i zarządzać nimi. Jeżeli jest jeden główny wątek i wiele małych operacji, które mogą być wykonywane w tle, można użyć klasy `ThreadPool`, utworzyć instancje delegatów wskazujących te operacje zaimplementowane jako metody i umieścić je w kolejce. Instancje te są automatycznie przydzielane do wątków znajdujących się w puli.

W tym rozdziale użyjesz typu `Task` do zarządzania wątkami na wyższym poziomie abstrakcji.

Wątki mogą konkurować między sobą o dostęp do współdzielonych zasobów, takich jak zmienne, pliki czy obiekty bazy danych. Można nimi zarządzać, o czym się przekonasz w praktyce w dalszej części rozdziału.

Nie zawsze podwojenie liczby wątków skraca czas wykonania danej operacji o połowę. To zależy od rodzaju operacji. Może się zdarzyć, że ten czas nawet się wydłuży, co ilustruje rysunek 4.1.



Rysunek 4.1. Tweet na temat wykonywania pracy w prawdziwym świecie

Wskazówka

Dobra praktyka: nie zakładaj, że zwiększenie liczby wątków poprawi wydajność aplikacji. Sprawdź wydajność bazowej implementacji z jednym wątkiem, a następnie porównaj ją z implementacją wielowątkową. Oprócz tego mierz wydajność w środowisku testowym możliwie zbliżonym do produkcyjnego.

Monitorowanie wydajności aplikacji i wykorzystania zasobów

Abyś mógł zwiększać wydajność kodu, musisz umieć go monitorować i tworzyć linię odniesienia, która pozwoli Ci mierzyć postępy.

Ocena efektywności typów danych

Jaki typ danych jest najlepszy w konkretnym przypadku? Aby odpowiedzieć na to pytanie, trzeba dokładnie rozważyć, co oznacza pojęcie *najlepszy*. W związku z tym należy wziąć pod uwagę następujące czynniki:

- **Funkcjonalność:** czy dany typ zapewnia potrzebne funkcje?
- **Wielkość:** ile bajtów pamięci zajmuje dany typ?
- **Wydajność:** jak szybki jest dany typ?
- **Przyszłe potrzeby:** czynnik zależny od zmian w wymaganiach i od możliwości utrzymania.

W niektórych przypadkach, na przykład przetwarzania liczb, te same funkcjonalności są dostępne w kilku typach. Wówczas podczas dokonywania wyboru trzeba wziąć pod uwagę ich wielkość i wydajność.

Jeśli liczb jest kilka milionów i trzeba je zapisać, najlepszym typem będzie ten, który zajmuje najmniej pamięci. Jeżeli natomiast liczb jest zaledwie kilka, ale trzeba na nich wykonywać wiele obliczeń, najlepszy będzie taki typ, który jest najszybszy dla danego procesora.

Funkcja `sizeof` zwraca liczbę bajtów pamięci zajmowanych przez pojedynczą instancję danego typu. Jeżeli wiele wartości ma być przechowywanych w bardziej złożonych strukturach, na przykład tablicach lub listach, potrzebny jest lepszy sposób określania zajętości pamięci.

W książkach i w internecie można znaleźć wiele porad na ten temat, jednak najlepszym rozwiązaniem jest samodzielne porównanie typów i wybór najlepszego dla danego kodu.

W następnym punkcie dowiesz się, jak pisać kod, aby monitorować rzeczywistą zajętość pamięci i wydajność różnych typów.

Na przykład dzisiaj najlepszy może się wydawać typ `short`. Jednak typ `int`, mimo że zajmuje dwa razy więcej miejsca w pamięci, będzie jeszcze lepszym wyborem, ponieważ w przyszłości może być konieczne przechowywanie szerszego zakresu wartości.

Jak wspomniałem, jest jeszcze jeden czynnik, o którym programiści często zapominają: możliwości utrzymania kodu. Jest to miara wysiłku, jaki musi włożyć inny programista, aby zrozumieć i zmodyfikować Twój kod. Jeśli dokonasz nieoczywistego wyboru i nie umieścisz komentarza z uzasadnieniem, programista, który będzie musiał później na przykład poprawić błąd lub dodać nową funkcję, będzie zdezorientowany.

Monitorowanie wydajności kodu i zajętości pamięci za pomocą przestrzeni Diagnostics

Przestrzeń nazw `System.Diagnostics` oferuje wiele przydatnych typów do monitorowania kodu. Pierwszym, któremu się przyjrzymy, jest `Stopwatch`.

1. W wybranym edytorze kodu utwórz projekt biblioteki klas z następującymi ustawieniami:
 - szablon projektu: *Biblioteka klas (Class Library)/classlib*,
 - plik/katalog rozwiązania/ obszaru roboczego: *Chapter04*,
 - plik/katalog projektu: *MonitoringLib*.
2. Utwórz projekt aplikacji konsolowej z następującymi ustawieniami:
 - szablon projektu: *Aplikacja konsoli (Console App)/console*,
 - plik/katalog rozwiązania/ obszaru roboczego: *Chapter04*,
 - plik/katalog projektu: *MonitoringApp*.
3. Wskaż aktywny projekt:
 - W środowisku Visual Studio 2022 oznacz bieżący projekt jako startowy.
 - W środowisku Visual Studio Code ustaw *MonitoringApp* jako aktywny projekt OmniSharp.
4. W projekcie *MonitoringLib* zmień nazwę pliku *Class1.cs* na *Recorder.cs*.
5. W tym samym projekcie zaimportuj statycznie i globalnie przestrzeń nazw `System.Console`.
6. W projekcie *MonitoringApp* zaimportuj statycznie i globalnie przestrzeń nazw `System.Console` i dodaj odwołanie do biblioteki klas *MonitoringLib*, jak niżej:

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

<ItemGroup>
  <ProjectReference
    Include="..\MonitoringLib\MonitoringLib.csproj" />
</ItemGroup>
```
7. Skompiluj projekt *MonitoringApp*.

Przydatne elementy typów Stopwatch i Process

Typ Stopwatch ma kilka przydatnych elementów opisanych w poniższej tabeli.

Element	Opis
Metoda Restart	Metoda zerująca pomiar czasu i uruchamiająca stoper
Metoda Stop	Metoda zatrzymująca stoper
Właściwość Elapsed	Czas, który upłynął, zapisany w formacie TimeSpan (np. godziny:minuty:sekundy)
Właściwość ElapsedMilliseconds	Czas w milisekundach, który upłynął, zapisany jako wartość Int64

Typ Process ma kilka przydatnych elementów opisanych w poniższej tabeli.

Element	Opis
Właściwość VirtualMemorySize64	Liczba bajtów wirtualnej pamięci, która została przydzielona procesowi
Właściwość WorkingSet64	Liczba bajtów fizycznej pamięci, która została przydzielona procesowi

Implementacja typu Recorder

Utwórz typ Recorder, który ułatwi Ci monitorowanie czasu działania kodu i zajętości pamięci. Wykorzystaj w tym celu typy Stopwatch i Process.

1. W pliku *Recorder.cs* wpisz poniższy kod, który za pomocą instancji typu Stopwatch i Process mierzy, odpowiednio, czas działania kodu i zajętość pamięci:

```
using System.Diagnostics; // Stopwatch

using static System.Diagnostics.Process; // GetCurrentProcess

namespace Packt.Shared;

public static class Recorder
{
    private static Stopwatch timer = new();

    private static long bytesPhysicalBefore = 0;
    private static long bytesVirtualBefore = 0;

    public static void Start()
    {
        // Jawne uporządkowanie pamięci w celu usunięcia obiektów, które nie są już używane.
```

```
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();

// Zapisanie bieżącego wykorzystania pamięci wirtualnej i fizycznej.
bytesPhysicalBefore = GetCurrentProcess().WorkingSet64;
bytesVirtualBefore = GetCurrentProcess().VirtualMemorySize64;

timer.Restart();
}

public static void Stop()
{
    timer.Stop();

    long bytesPhysicalAfter =
        GetCurrentProcess().WorkingSet64;

    long bytesVirtualAfter =
        GetCurrentProcess().VirtualMemorySize64;

    WriteLine("Liczba zajętych bajtów pamięci fizycznej: {0:N0}.",
        bytesPhysicalAfter - bytesPhysicalBefore);

    WriteLine("Liczba zajętych bajtów pamięci wirtualnej: {0:N0}.",
        bytesVirtualAfter - bytesVirtualBefore);

    WriteLine("Czas działania: {0}.", timer.Elapsed);

    WriteLine("Czas działania w milisekundach: {0:N0}.",
        timer.ElapsedMilliseconds);
}
}
```

Uwaga

Metoda Start klasy Recorder, zanim odczyta ilość zajętej pamięci, usuwa z niej wszystkie przydzielone, ale nie wykorzystane obszary z użyciem klasy GC (ang. *garbage collector*, kolektor śmieci). Jest to zaawansowana operacja, której nie należy implementować w kodzie, ponieważ mechanizm GC więcej „wie” o wykorzystaniu pamięci niż programista i jemu należy powierzyć podejmowanie decyzji o porządkowaniu niewykorzystywanych obszarów. Konieczność przejęcia kontroli przez kod jest w tym przykładzie wyjątkiem.

2. Usuń z pliku *Program.cs* istniejące instrukcje, a następnie wpisz poniższe, które uruchamiają i zatrzymują rejestrator czasu oraz tworzą tablicę 10 000 liczb całkowitych:

```
using Packt.Shared; // Recorder

WriteLine("Przetwarzanie danych. Czekaj...");
Recorder.Start();

// Symulacja procesu zajmującego pewną ilość pamięci...
int[] largeArrayOfInts = Enumerable.Range(
    start: 1, count: 10_000).ToArray();

// ...i trwającego jakiś czas.
Thread.Sleep(new Random().Next(5, 10) * 1000);
Recorder.Stop();
```

3. Uruchom kod. Uzyskasz wynik podobny do poniższego:

```
Przetwarzanie danych. Czekaj...
Liczba zajętych bajtów pamięci fizycznej: 889 824.
Liczba zajętych bajtów pamięci wirtualnej: 131 072.
Czas działania: 00:00:06.0074221.
Czas działania w milisekundach: 6 007.
```

Uwaga

Zwróć uwagę, że w programie jest wprowadzana zwłoka o losowej długości od 5 do 10 sekund. Ponadto wielokrotnie uruchamiając ten kod na tym samym komputerze, za każdym razem uzyskasz różne wyniki. Na przykład na moim laptopie Mac Mini M1 program zajął mniej pamięci fizycznej, ale więcej wirtualnej:

```
Przetwarzanie danych. Czekaj...
Liczba zajętych bajtów pamięci fizycznej: 294 912.
Liczba zajętych bajtów pamięci wirtualnej: 10 485 760.
Czas działania: 00:00:06.0016951.
Czas działania w milisekundach: 6 001.
```

Monitorowanie wydajności przetwarzania ciągów znaków

Teraz, gdy wiesz już, jak używać typów `Stopper` i `Process` do monitorowania wydajności kodu, wykorzystaj je do wybrania najlepszego sposobu przetwarzania ciągów znaków.

1. W projekcie *MonitoringApp* utwórz plik klasy *Program.Helpers.cs*.
2. W nowym pliku zdefiniuj częściową klasę `Program` z metodą wyświetlającą tytuł sekcji w ciemnożółtym kolorze, jak niżej:

```
partial class Program
{
    static void SectionTitle(string title)
    {
        ConsoleColor previousColor = ConsoleColor;
        ConsoleColor = ConsoleColor.DarkYellow;
        WriteLine("*");
        WriteLine($"* {title}");
        WriteLine("*");
        ConsoleColor = previousColor;
    }
}
```

3. W pliku *Program.cs* oznacz jako komentarz wpisane wcześniej instrukcje przez umieszczenie ich wewnątrz znaków `/* */`.
4. Wpisz poniższe instrukcje, które tworzą tablicę 50 000 liczb całkowitych, a następnie rozdzielają je przecinkami i dołączają do ciągu znaków:

```
int[] numbers = Enumerable.Range(
    start: 1, count: 50_000).ToArray();

SectionTitle("Klasa StringBuilder");
Recorder.Start();

System.Text.StringBuilder builder = new();
for (int i = 0; i < numbers.Length; i++)
{
    builder.Append(numbers[i]);
    builder.Append(", ");
}

Recorder.Stop();
WriteLine();

SectionTitle("Typ string i operator +");
Recorder.Start();

string s = string.Empty; // Ciąg ""

for (int i = 0; i < numbers.Length; i++)
{
    s += numbers[i] + ", ";
}

Recorder.Stop();
```

5. Uruchom kod. Uzyskasz wynik podobny do poniższego:

```
*
* Klasa StringBuilder
*
```

```
Liczba zajętych bajtów pamięci fizycznej: 1 150 976.  
Liczba zajętych bajtów pamięci wirtualnej: 0.  
Czas działania: 00:00:00.0010796.  
Czas działania w milisekundach: 1.
```

```
*  
* Typ string i operator +  
*  
Liczba zajętych bajtów pamięci fizycznej: 11 849 728  
Liczba zajętych bajtów pamięci wirtualnej: 1 638 400.  
Czas działania: 00:00:01.7754252.  
Czas działania w milisekundach: 1 775.
```

Powyższe wyniki można podsumować następująco:

- Kod wykorzystujący klasę `StringBuilder` zajął ok. 1 MB pamięci fizycznej, pamięci wirtualnej nie zajął w ogóle, a opisaną operację wykonywał przez mniej więcej 1 milisekundę.
- Kod wykorzystujący typ `string` i operator `+` zajął ok. 11 MB pamięci fizycznej, 1,5 MB pamięci wirtualnej, a opisaną operację wykonywał przez mniej więcej 1,7 milisekundy.

W tym przykładzie klasa `StringBuilder` jest ponad 1000 razy szybsza i zajmuje ok. 10 razy mniej pamięci niż typ `string`. Wynika to stąd, że typ ten jest niemutowalny i podczas każdorazowego dołączania ciągu znaków jest tworzony nowy ciąg. Natomiast klasa `StringBuilder` tworzy w pamięci jeden bufor, w którym umieszcza dołączane znaki.

Wskazówka

Dobra praktyka: nie stosuj w pętlach metody `String.Concat` ani operatora `+`. Zamiast nich używaj klasy `StringBuilder`.

Teraz, gdy wiesz już, jak mierzyć wydajność kodu i efektywność wykorzystania zasobów z użyciem wbudowanych typów platformy .NET, poznaj pakiet NuGet, który oferuje bardziej zaawansowane techniki pomiarów.

Monitorowanie wydajności kodu i zajętości pamięci za pomocą pakietu Benchmark.NET

Benchmark.NET to popularny pakiet NuGet do wykonywania testów porównawczych aplikacji opartych na platformie .NET. Microsoft propaguje go na swoim blogu we wpisach poświęconych poprawianiu wydajności kodu. Dlatego warto, abyś wiedział,

jak działa ten pakiet, i mógł go używać we własnych testach wydajności. Zobacz teraz, jak możesz go użyć do porównania wydajności operacji łączenia ciągów znaków za pomocą typu `string` i klasy `StringBuilder`.

1. W wybranym edytorze kodu otwórz rozwiązanie/ obszar roboczy o nazwie *Chapter04* i dodaj do niego aplikację konsolową o nazwie *Benchmarking*.
 - W środowisku Visual Studio 2022 oznacz bieżący projekt jako startowy.
 - W środowisku Visual Studio Code ustaw *Benchmarking* jako aktywny projekt OmniSharp.

2. W pliku konfiguracyjnym projektu dodaj odwołanie do pakietu `Benchmark.NET`, jak niżej. Pamiętaj, aby użyć jego najnowszej wersji:

```
<ItemGroup>
  <PackageReference Include="BenchmarkDotNet" Version="0.13.1" />
</ItemGroup>
```

3. Skompiluj projekt, aby został pobrany wymagany pakiet.

4. Utwórz plik klasy *StringBenchmarks.cs*.

5. Wpisz poniższy kod definiujący klasę z metodami, których użyjesz w pomiarach porównawczych. Metody te wykorzystują, odpowiednio, typ `string` i klasę `StringBuilder` do utworzenia ciągu 20 liczb oddzielonych przecinkami.

```
using BenchmarkDotNet.Attributes; // [Benchmark]
```

```
public class StringBenchmarks
{
    int[] numbers;

    public StringBenchmarks()
    {
        numbers = Enumerable.Range(
            start: 1, count: 20).ToArray();
    }

    [Benchmark(Baseline = true)]
    public string StringConcatenationTest()
    {
        string s = string.Empty; // Ciąg ""

        for (int i = 0; i < numbers.Length; i++)
        {
            s += numbers[i] + ", ";
        }

        return s;
    }
}
```

```
[Benchmark]
public string StringBuilderTest()
{
    System.Text.StringBuilder builder = new();

    for (int i = 0; i < numbers.Length; i++)
    {
        builder.Append(numbers[i]);
        builder.Append(", ");
    }

    return builder.ToString();
}
}
```

6. Usuń z pliku *Program.cs* istniejące instrukcje, a następnie wpisz poniższe, które importują przestrzeń nazw `BenchmarkDotNet.Running` i wywołują metodę klasy pomiarowej:

```
using BenchmarkDotNet.Running;
BenchmarkRunner.Run<StringBenchmarks>();
```

7. Uruchom aplikację w trybie *Release*:

- W środowisku Visual Studio 2022 wybierz z rozwijanej listy *Konfiguracje rozwiązania (Solution Configurations)* pozycję *Release*, a następnie kliknij polecenie menu *Debug/Start Without Debugging (Debuguj/Uruchom bez debugowania)*.
- W środowisku Visual Studio Code otwórz terminal i wpisz w nim polecenie **run --configuration Release**.

8. Uzyskasz wyniki zawierające kilka nieistotnych informacji, m.in. nazwy plików z raportami. Ważna jest natomiast tabela w sekcji *Summary* (podsumowanie), według której łączenie ciągów z użyciem typu `string` i klasy `StringBuilder` średnio zajęło, odpowiednio, 413 ns i 275,1 ns. Ilustruje to poniższy częściowy wynik.

```
// ***** BenchmarkRunner: Finish *****

// * Export *
BenchmarkDotNet.Artifacts\results\StringBenchmarks-report.csv
BenchmarkDotNet.Artifacts\results\StringBenchmarks-report-github.md
BenchmarkDotNet.Artifacts\results\StringBenchmarks-report.html

// * Detailed results *
StringBenchmarks.StringConcatenationTest: DefaultJob
Runtime = .NET 7.0.0 (7.0.22.22904), X64 RyuJIT; GC = Concurrent
Workstation
Mean = 412.990 ns, StdErr = 2.353 ns (0.57%), N = 46, StdDev = 15.957 ns
Min = 373.636 ns, Q1 = 413.341 ns, Median = 417.665 ns, Q3 = 420.775 ns,
```

```

Max = 434.504 ns
IQR = 7.433 ns, LowerFence = 402.191 ns, UpperFence = 431.925 ns
ConfidenceInterval = [404.708 ns; 421.273 ns] (CI 99.9%), Margin = 8.282 ns
(2.01% of Mean)

```

```
Skewness = -1.51, Kurtosis = 4.09, MValue = 2
```

```

----- Histogram -----
[370.520 ns ; 382.211 ns) | @@@@@@
[382.211 ns ; 394.583 ns) | @
[394.583 ns ; 411.300 ns) | @@
[411.300 ns ; 422.990 ns) | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
[422.990 ns ; 436.095 ns) | @@@@
-----

```

```
StringBenchmarks.StringBuilderTest: DefaultJob
```

```
Runtime = .NET 7.0.0 (7.0.22.22904), X64 RyuJIT; GC = Concurrent
Workstation
```

```

Mean = 275.082 ns, StdErr = 0.558 ns (0.20%), N = 15, StdDev = 2.163 ns
Min = 271.059 ns, Q1 = 274.495 ns, Median = 275.403 ns, Q3 = 276.553 ns,
Max = 278.030 ns

```

```

IQR = 2.058 ns, LowerFence = 271.409 ns, UpperFence = 279.639 ns
ConfidenceInterval = [272.770 ns; 277.394 ns] (CI 99.9%), Margin = 2.312
ns (0.84% of Mean)

```

```
Skewness = -0.69, Kurtosis = 2.2, MValue = 2
```

```

----- Histogram -----
[269.908 ns ; 278.682 ns) | @@@@@@@@@@@@@@@@@@
-----

```

```
// * Summary *
```

```
BenchmarkDotNet=v0.13.1, OS=Windows 10.0.22000
```

```
11th Gen Intel Core i7-1165G7 2.80GHz, 1 CPU, 8 logical and 4 physical cores
.NET SDK=7.0.100
```

```
[Host] : .NET 7.0.0 (7.0.22.22904), X64 RyuJIT
```

```
DefaultJob : .NET 7.0.0 (7.0.22.22904), X64 RyuJIT
```

	Method	Mean	Error	StdDev	Ratio	
RatioSD						
	StringConcatenationTest	413.0 ns	8.28 ns	15.96 ns	1.00	0.00
	StringBuilderTest	275.1 ns	2.31 ns	2.16 ns	0.69	0.04

```
// * Hints *
```

```
Outliers
```

```
StringBenchmarks.StringConcatenationTest: Default -> 7 outliers
```

```
↳ were removed, 14 outliers were detected (376.78 ns..391.88 ns, 440.79
```

```
↳ ns..506.41 ns)
```

```
StringBenchmarks.StringBuilderTest: Default -> 2 outliers
```

```
↳ were detected (274.68 ns, 274.69 ns)
```



```
// * Legends *
Mean      : Arithmetic mean of all measurements
Error     : Half of 99.9% confidence interval
StdDev    : Standard deviation of all measurements
Ratio     : Mean of the ratio distribution ([Current]/[Baseline])
RatioSD   : Standard deviation of the ratio distribution
([Current]/[Baseline])
1 ns     : 1 Nanosecond (0.000000001 sec)

// ***** BenchmarkRunner: End *****
// ** Remained 0 benchmark(s) to run **

Run time: 00:01:13 (73.35 sec), executed benchmarks: 2
Global total time: 00:01:29 (89.71 sec), executed benchmarks: 2
// * Artifacts cleanup *
```

Szczególnie ciekawa jest sekcja *Outliers* (wartości odstające), która pokazuje, że typ `string` jest nie tylko wolniejszy od klasy `StringBuilder`, ale również uzyskiwanie przy jego użyciu wyniki są mniej spójne. Twoje wyniki mogą być oczywiście inne. Wiedz również, że gdyby wyniki nie zawierały wartości odstających, nie byłoby sekcji *Hints* (wskazówki) i *Outliers*,

Poznałeś dwa sposoby mierzenia wydajności kodu. Teraz dowiedz się, jak możesz ją poprawić poprzez asynchroniczne wykonywanie zadań.

Asynchroniczne wykonywanie zadań

Aby zrozumieć, w jaki sposób kilka zadań może być wykonywanych w *tym samym czasie*, napiszesz kod, który wywoła trzy metody. Pierwsza będzie wykonywana przez 3 sekundy, druga przez 2 sekundy, a trzecia przez 1 sekundę. Użyjesz w tym celu klasy `Thread`, umożliwiającej m.in. wprowadzenie bieżącego wątku w stan uśpienia na zadaną liczbę milisekund.

Synchroniczne wykonywanie operacji

Zanim sprawisz, że zadania zostaną wykonane jednocześnie, uruchom je synchronicznie, czyli jedno po drugim.

1. W wybranym edytorze kodu otwórz rozwiązanie/ obszar roboczy o nazwie *Chapter04* i dodaj do niego aplikację konsolową o nazwie *WorkingWithTasks*.
 - W środowisku Visual Studio 2022 oznacz bieżący projekt jako startowy.
 - W środowisku Visual Studio Code ustaw *WorkingWithTasks* jako aktywny projekt OmniSharp.

2. W tym samym projekcie zaimportuj statycznie i globalnie przestrzeń nazw `System.Console`.
3. Utwórz plik klasy *Program.Helpers.cs*.
4. W nowym pliku zdefiniuj częściową klasę `Program` z metodami wyświetlającymi, dla poprawy czytelności w różnych kolorach, odpowiednio tytuł sekcji, nazwę zadania i informacje o bieżącym wątku, jak niżej:

```
partial class Program
{
    static void SectionTitle(string title)
    {
        ConsoleColor previousColor = ForegroundColor;
        ForegroundColor = ConsoleColor.DarkYellow;
        WriteLine("*");
        WriteLine($"* {title}");
        WriteLine("*");
        ForegroundColor = previousColor;
    }

    static void TaskTitle(string title)
    {
        ConsoleColor previousColor = ForegroundColor;
        ForegroundColor = ConsoleColor.Green;
        WriteLine($"*{title}");
        ForegroundColor = previousColor;
    }

    static void OutputThreadInfo()
    {
        Thread t = Thread.CurrentThread;
        ConsoleColor previousColor = ForegroundColor;
        ForegroundColor = ConsoleColor.DarkCyan;

        WriteLine(
            "ID wątku: {0}, priorytet: {1}, w tle: {2}, nazwa: {3}",
            t.ManagedThreadId, t.Priority, t.IsBackground, t.Name ?? "null");

        ForegroundColor = previousColor;
    }
}
```

5. Utwórz plik klasy *Program.Methods.cs*.
6. W nowym pliku zdefiniuj trzy metody symulujące pracę, jak niżej:

```
partial class Program
{
    static void MethodA()
    {
        TaskTitle("Rozpoczęcie metody A...");
    }
}
```

```

        OutputThreadInfo();
        Thread.Sleep(3000); // Symulacja 3 sekund pracy.
        TaskTitle("Zakończenie metody A.");
    }

    static void MethodB()
    {
        TaskTitle("Rozpoczęcie metody B...");
        OutputThreadInfo();
        Thread.Sleep(2000); // Symulacja 2 sekund pracy.
        TaskTitle("Zakończenie metody B.");
    }

    static void MethodC()
    {
        TaskTitle("Rozpoczęcie metody C...");
        OutputThreadInfo();
        Thread.Sleep(1000); // Symulacja 1 sekundy pracy.
        TaskTitle("Zakończenie metody C.");
    }
}

```

7. Usuń z pliku *Program.cs* istniejące instrukcje, a następnie wpisz poniższe, które wywołują metodę pomocniczą (wyświetlającą informacje o wątku), definiują i uruchamiają czasomierz oraz wywołują trzy metody symulujące pracę i metodę wyświetlającą czas działania kodu:

```

using System.Diagnostics; // Stopwatch

OutputThreadInfo();
Stopwatch timer = Stopwatch.StartNew();

SectionTitle("Synchroniczne wywołanie metod w jednym wątku.");
MethodA();
MethodB();
MethodC();

WriteLine($"Czas działania: {timer.ElapsedMilliseconds:# ##0} ms.");

```

8. Uruchom kod i zwróć uwagę, że na pierwszym planie zostanie uruchomiony tylko jeden wątek bez nazwy, który będzie działał przez 6 sekund:

```

ID wątku: 1, priorytet: Normal, w tle: False, nazwa: null
*
* Synchroniczne wywołanie metod w jednym wątku.
*
Rozpoczęcie metody A...
ID wątku: 1, priorytet: Normal, w tle: False, nazwa: null
Zakończenie metody A.
Rozpoczęcie metody B...
ID wątku: 1, priorytet: Normal, w tle: False, nazwa: null
Zakończenie metody B.
Rozpoczęcie metody C...

```

```
ID wątku: 1, priorytet: Normal, w tle: False, nazwa: null
Zakończenie metody C.
Czas działania: 6 012 ms.
```

Asynchroniczne wykonywanie operacji za pomocą zadań

Klasa `Thread` (wątek) pojawiła się w pierwszej wersji platformy .NET w 2002 r. Służy ona do tworzenia wątków i zarządzania nimi, ale bezpośrednie korzystanie z niej jest dość trudne. W 2010 r. w wersji platformy .NET Framework 4.0 wprowadzono klasę `Task` (zadania), która reprezentuje operację asynchroniczną. **Zadanie** to wyższego rzędu abstrakcja wątku, który wykonuje operację. Tworzenie zadań i zarządzanie nimi jest łatwiejsze niż w przypadku wątków. Dzięki wątkom opakowanym w zadania kod może wykonywać kilka operacji jednocześnie, czyli asynchronicznie.

Klasa `Task` ma właściwości `Status` i `CreationOptions` (opcje utworzenia), jak również metodę `ContinueWith`, którą można dostosowywać przez użycie wartości wyliczeniowej `TaskContinuationOptions` i którą można zarządzać za pomocą klasy `TaskFactory`.

Uruchomienie zadania

Poznaj trzy sposoby wywoływania metody za pomocą instancji klasy `Task`. W repozytorium GitHub przypisanym tej książce znajdziesz odnośniki do artykułów opisujących zalety i wady tych sposobów.

Każdy z tych sposobów wymaga zdefiniowania i uruchomienia zadania z użyciem innej składni.

1. W pliku `Program.cs` dopisz poniższe instrukcje, które definiują i wywołują trzy metody wykonujące poszczególne zadania:

```
SectionTitle("Asynchroniczne wywołanie metod w kilku wątkach.");
timer.Restart();

Task taskA = new(MethodA);
taskA.Start();
Task taskB = Task.Factory.StartNew(MethodB);
Task taskC = Task.Run(MethodC);

WriteLine($"Czas działania: {timer.ElapsedMilliseconds:# ##0} ms.");
```

2. Uruchom kod i zwróć uwagę, że zakończy działanie niemal natychmiast, ponieważ każda metoda zostanie wywoływana w tle w osobnym wątku przydzielonym z puli.

```
ID wątku: 1, priorytet: Normal, w tle: False, nazwa: null
*
```

```
* Asynchroniczne wywołanie metod w kilku wątkach.
*
Rozpoczęcie metody A...
ID wątku: 6, priorytet: Normal, w tle: True, nazwa: .NET ThreadPool
Worker
Rozpoczęcie metody C...
ID wątku: 8, priorytet: Normal, w tle: True, nazwa: .NET ThreadPool
Worker
Rozpoczęcie metody B...
ID wątku: 9, priorytet: Normal, w tle: True, nazwa: .NET ThreadPool
Worker
Czas działania: 6 ms.
```

Uwaga

Może się zdarzyć, że aplikacja zakończy działanie, zanim jedno, czy nawet wszystkie zadania zostaną wykonane i coś wyświetli w terminalu!

Oczekiwanie na wykonanie zadań

Czasami trzeba wstrzymać działanie kodu do czasu wykonania zadania. W tym celu można użyć metody `Wait` instancji klasy `Task` lub statycznej metody `WaitAll` albo `WaitAny` z tablicą zadań w argumentcie, jak opisano w poniższej tabeli:

Metoda	Opis
<code>t.Wait()</code>	Oczekiwanie na zakończenie zadania reprezentowanego przez instancję <code>t</code> klasy <code>Task</code>
<code>Task.WaitAny(Task[])</code>	Oczekiwanie na zakończenie dowolnego zadania umieszczonego w tablicy
<code>Task.WaitAll(Task[])</code>	Oczekiwanie na zakończenie wszystkich zadań umieszczonych w tablicy

Korzystanie z metod oczekujących na wykonanie zadań

Zobacz teraz, jak możesz użyć powyższych metod do rozwiązania problemu z aplikacją.

1. W pliku `Program.cs`, poniżej instrukcji uruchamiającej trzecie zadanie, ale przed instrukcją wyświetlającą czas działania, wpisz poniższy kod, który tworzy tablicę zadań i umieszcza ją w argumentcie metody `WaitAll`:

```
Task[] tasks = { taskA, taskB, taskC };
Task.WaitAll(tasks);
```

2. Uruchom kod i zwróć uwagę, że po wywołaniu metody `WaitAll` wątek zostanie wstrzymany do czasu wykonania wszystkich zadań. Dopiero potem pojawi się informacja o czasie działania kodu, nieco przekraczającym 3 sekundy:

```
Rozpoczęcie metody A...
Rozpoczęcie metody B...
ID wątku: 8, priorytet: Normal, w tle: True, nazwa: .NET ThreadPool
Worker
ID wątku: 6, priorytet: Normal, w tle: True, nazwa: .NET ThreadPool
Worker
Rozpoczęcie metody C...
ID wątku: 4, priorytet: Normal, w tle: True, nazwa: .NET ThreadPool
Worker
Zakończenie metody C.
Zakończenie metody B.
Zakończenie metody A.
Czas działania: 3 005 ms.
```

Trzy nowe wątki uruchamiają się niemal jednocześnie, potencjalnie w dowolnej kolejności. Jako pierwsza kończy działanie metoda C, ponieważ trwa tylko 1 sekundę. Następna jest metoda B, zajmująca 2 sekundy, a na końcu metoda A, działająca przez 3 sekundy.

Duży wpływ na uzyskiwane wyniki ma procesor, który przydziela interwały czasu każdemu wątkowi. Programista nie ma możliwości kontrolowania chwili wywołania poszczególnych metod.

Sekwencyjne wykonanie innego zadania

Po jednoczesnym uruchomieniu kilku zadań wystarczy poczekać, aż wszystkie zostaną wykonane. Jednak często kolejne zadanie jest uzależnione od wyników poprzedniego. W takich sytuacjach definiuje się **zadanie kontynuacyjne**.

W kolejnym przykładzie utworzysz dwie metody symulujące usługę internetową. Pierwsza będzie zwracać pewną kwotę, a druga liczbę produktów, które kosztują więcej, niż wynosi ta kwota. Wynik zwrócony przez pierwszą metodę musi być przekazany drugiej metodzie.

Tym razem żadna z metod nie będzie wstrzymywała działania wątków na stały przedział czasu. Użyjesz klasy `Random`, dzięki której każda z nich będzie symulowała pracę przez losowy okres od 2 do 4 sekund.

1. W pliku `Program.Methods.cs` wpisz poniższy kod definiujący dwie metody. Pierwsza symuluje wywołanie usługi internetowej, a druga wywołanie procedury składowanej w bazie danych.

```

static decimal CallWebService()
{
    TaskTitle("Wywołanie usługi internetowej");
    OutputThreadInfo();
    Thread.Sleep((new Random()).Next(2000, 4000));
    TaskTitle("Koniec wywołania usługi internetowej.");
    return 89.99M;
}

static string CallStoredProcedure(decimal amount)
{
    TaskTitle("Wywołanie procedury składowanej...");
    OutputThreadInfo();
    Thread.Sleep((new Random()).Next(2000, 4000));
    TaskTitle("Koniec wywołania procedury składowanej.");
    return $"12 produktów kosztuje więcej niż {amount:C}.";
}

```

2. W pliku *Program.cs* wpisz poniższy kod, który uruchamia zadanie odwołujące się do usługi internetowej, a zwrócony wynik przekazuje zadaniu wywołującemu procedurę składowaną:

```

SectionTitle("Przekazanie wyniku jednego zadania na wejście innego.");
timer.Restart();

```

```

Task<string> taskServiceThenSProc = Task.Factory
    .StartNew(CallWebService) // Zwraca wynik typu Task<decimal>.
    .ContinueWith(previousTask => // Zwraca wynik typu Task<string>.
        CallStoredProcedure(previousTask.Result));

```

```

WriteLine($"Wynik: {taskServiceThenSProc.Result}");

```

```

WriteLine($"Czas działania: {timer.ElapsedMilliseconds:# ##0} ms.");

```

3. Uruchom kod. Uzyskasz wynik podobny do poniższego:

```

Wywołanie usługi internetowej...
ID wątku: 4, priorytet: Normal, w tle: True, nazwa: .NET ThreadPool Worker
Koniec wywołania usługi internetowej.
Wywołanie procedury składowanej...
ID wątku: 6, priorytet: Normal, w tle: True, nazwa: .NET ThreadPool Worker
Koniec wywołania procedury składowanej.
Wynik: 12 produktów kosztuje więcej niż 89,99 zł.
Czas działania: 5 005 ms.

```

Usługa internetowa i procedura składowana mogą zostać wywołane w dwóch różnych wątkach, w powyższym przykładzie oznaczonych numerami 4 i 6, albo w jednym, jeżeli będzie wolny w chwili rozpoczęcia drugiego zadania.

Zadania zagnieżdżone i podrzędne

Oprócz zależności między zadaniami można definiować **zadania zagnieżdżone i podrzędne**. Zadanie zagnieżdżone jest wykonywane wewnątrz innego zadania. Zadanie podrzędne jest podobne, ale musi zostać wykonane przed zakończeniem zadania nadrzędnego.

Przyjrzyj się obu rodzajom zadań.

1. W pliku *Program.Methods.cs* wpisz poniższy kod definiujący dwie metody. Pierwsza uruchamia zadanie, które inicjuje inne zadanie.

```
static void OuterMethod()
{
    TaskTitle("Wywołanie metody zewnętrznej...");
    Task innerTask = Task.Factory.StartNew(InnerMethod);
    TaskTitle("Koniec metody zewnętrznej.");
}

static void InnerMethod()
{
    TaskTitle("Wywołanie metody wewnętrznej...");
    Thread.Sleep(2000);
    TaskTitle("Koniec metody wewnętrznej.");
}
```

2. W pliku *Program.cs* wpisz poniższy kod, który wywołuje metodę zewnętrzną, czeka na jej wykonanie, po czym kończy działanie:

```
SectionTitle("Zadania zagnieżdżone i podrzędne");

Task outerTask = Task.Factory.StartNew(OuterMethod);
outerTask.Wait();
WriteLine("Zakończenie działania aplikacji.");
```

3. Uruchom kod. Uzyskasz wynik podobny do poniższego:

```
Wywołanie metody zewnętrznej...
Wywołanie metody wewnętrznej...
Koniec metody zewnętrznej.
Zakończenie działania aplikacji.
```

Uwaga

Chociaż kod czeka na zakończenie zewnętrznego zadania, jego wewnętrzne zadanie nie musi być wykonane do końca. Co więcej, zewnętrzne zadanie może zostać wykonane, a aplikacja może zakończyć działanie, jeszcze zanim wewnętrzne zadanie zostanie rozpoczęte!

4. Aby połączyć zadania relacją nadrzędne-podrzędne, trzeba użyć specjalnej opcji. Dodaj wyróżniony w poniższym kodzie argument `TaskCreationOptions.AttachedToParent`:


```
Task innerTask = Task.Factory.StartNew(InnerMethod,
    TaskCreationOptions.AttachedToParent);
```

5. Ponownie uruchom kod. Zwróć uwagę, że tym razem zewnętrzna metoda zaczeka, aż zostanie wykonana metoda wewnętrzna:

Wywołanie metody zewnętrznej...

Wywołanie metody wewnętrznej...

Koniec metody zewnętrznej.

Koniec metody wewnętrznej.

Zakończenie działania aplikacji.

Uwaga

Metoda zewnętrzna może zakończyć działanie przed metodą wewnętrzną, jak pokazuje powyższy wynik, ale jej zadanie musi czekać na wykonanie zadania wewnętrznego.

Opakowanie zadania w obiekcie

Czasami potrzebna jest metoda asynchroniczna, której wynik nie jest zadaniem. W takim przypadku wynik można umieścić w pomyślnie wykonanym zadaniu, a następnie zwrócić wyjątek lub wskazać z użyciem jednej ze statycznych metod wymienionych w poniższej tabeli, że zadanie zostało anulowane.

Metoda	Opis
<code>FromResult<TResult>(TResult)</code>	Utworzenie obiektu <code>Task<TResult></code> , którego właściwość <code>Result</code> zawiera wynik niebędący zadaniem, a właściwość <code>Status</code> zawiera wartość <code>RanToCompletion</code>
<code>FromException<TResult>(Exception)</code>	Utworzenie obiektu <code>Task<TResult></code> reprezentującego zadanie, po którego wykonaniu jest zgłaszany zadany wyjątek
<code>FromCanceled<TResult>(CancellationToken)</code>	Utworzenie obiektu <code>Task<TResult></code> reprezentującego zadanie, którego wykonanie można przerwać za pomocą zadanego tokenu

Powyższe metody mają następujące zastosowania:

- Synchroniczna implementacja interfejsu zawierającego metody asynchroniczne. Jest to typowe podejście stosowane w witrynach internetowych i usługach.
- Symulowanie asynchronicznych implementacji w testach jednostkowych.

Założmy, że trzeba utworzyć metodę sprawdzającą poprawność danych wejściowych XML. Metoda musi być zgodna z interfejsem i zwracać wynik typu `Task<T>`, jak niżej:

```
public interface IValidation
{
    Task<bool> IsValidXmlTagAsync(this string input);
}
```

Aby zwrócić wyniki opakowane w zadanie, można użyć pomocnych metod `FromX`, jak w poniższym kodzie:

```
using System.Text.RegularExpressions;

namespace Packt.Shared;

public static class StringExtensions : IValidation
{
    public static Task<bool> IsValidXmlTagAsync(this string input)
    {
        if (input == null)
        {
            return Task.FromException<bool>(
                new ArgumentNullException($"Brak parametru {nameof(input)}"));
        }

        if (input.Length == 0)
        {
            return Task.FromException<bool>(
                new ArgumentException($"Parameter {nameof(input)} jest pusty."));
        }

        return Task.FromResult(Regex.IsMatch(input,
            @"^<([a-z]+)([^\<]+)*(?:>(.*)<\/\1>|\s+\/>)$$"));
    }
}
```

Jeżeli zaimplementowana metoda zwraca wynik typu `Task` (odpowiednik `void` w metodzie synchronicznej), może to być predefiniowany obiekt reprezentujący ukończone zadanie, jak w poniższym kodzie:

```
public Task DeleteCustomerAsync()
{
    //...
    return Task.CompletedTask;
}
```

Synchronizacja dostępu do współdzielonych zasobów

Jeżeli kilka wątków jest wykonywanych w tym samym czasie, może się zdarzyć, że jednocześnie odwołają się do tej samej zmiennej lub zasobu i spowodują problem. Dlatego trzeba zwracać baczną uwagę, aby kod był **wątkowo bezpieczny**. Najprostszym rozwiązaniem jest użycie zmiennej obiektowej w charakterze flagi oznaczającej blokadę współdzielonego zasobu i uzyskanie do niego dostępu na wyłączność.

W książce *Władca much* Williama Goldinga Prosiaczek i Ralph znajdują muszlę i używają jej do przeprowadzenia zebrania. Określają przy tym „zasadę muszli”, że mówić może tylko ten, kto trzyma muszlę.

Obiekt, którego używam do implementacji kodu wątkowo bezpiecznego, mam zwyczaj nazywać muszlą. Gdy jakiś wątek „ma muszlę”, żaden inny wątek nie powinien uzyskać dostępu do reprezentowanych przez nią współdzielonych zasobów. Zwróć uwagę, że użyłem tu zwrotu „nie powinien”. Zsynchronizowany dostęp jest możliwy tylko wtedy, gdy w kodzie jest respektowana „zasada muszli”. Muszla sama w sobie nie jest blokadą.

Przyjrzymy się dwóm typom obiektów używanych do synchronizowania dostępu do współdzielonych zasobów:

- **Monitor**: obiekt wykorzystywany przez wiele wątków do sprawdzania, czy mogą uzyskać dostęp do zasobu w ramach tego samego procesu.
- **Interlocked**: obiekt wykorzystywany do wykonywania operacji na prostych typach liczbowych na poziomie procesora.

Synchronizacja dostępu do zasobu

Utwórz aplikację konsolową, która pozwoli Ci dowiedzieć się, jak zasoby są współdzielone między wątkami.

1. W wybranym edytorze kodu otwórz rozwiązanie/ obszar roboczy o nazwie *Chapter04* i dodaj do niego aplikację konsolową o nazwie *SynchronizingResourceAccess*.
 - W środowisku Visual Studio 2022 oznacz bieżący projekt jako startowy.
 - W środowisku Visual Studio Code ustaw *SynchronizingResourceAccess* jako aktywny projekt OmniSharp.
2. W tym samym projekcie zaimportuj statycznie i globalnie przestrzeń nazw `System.Console`.
3. Utwórz plik klasy *SharedObjects.cs*.

4. W nowym pliku zdefiniuj statyczną klasę zawierającą pole `Message`, które będzie współdzielonym zasobem, jak niżej:

```
static class SharedObjects
{
    public static string? Message; // Współdzielony zasób.
}
```

5. Utwórz plik klasy `Program.Methods.cs`.

6. W nowym pliku wpisz poniższy kod definiujący dwie metody. Każda z nich zawiera pętlę, która pięciokrotnie dołącza do pola `Message` literę A lub B:

```
partial class Program
{
    static void MethodA()
    {
        for (int i = 0; i < 5; i++)
        {
            Thread.Sleep(Random.Shared.Next(2000));
            SharedObjects.Message += "A";
            Write(".");
        }
    }

    static void MethodB()
    {
        for (int i = 0; i < 5; i++)
        {
            Thread.Sleep(Random.Shared.Next(2000));
            SharedObjects.Message += "B";
            Write(".");
        }
    }
}
```

7. Usuń z pliku `Program.cs` istniejące instrukcje, a następnie wpisz poniższy kod, który importuje przestrzeń nazw `System.Diagnostics`, wywołuje obie powyższe metody wykonujące zadania w osobnych wątkach, oczekuje na zakończenie ich działania i wyświetla czas, jaki upłynął:

```
using System.Diagnostics; // Stopwatch

WriteLine("Oczekiwanie na zakończenie zadań.");
Stopwatch watch = Stopwatch.StartNew();
Task a = Task.Factory.StartNew(MethodA);
Task b = Task.Factory.StartNew(MethodB);

Task.WaitAll(new Task[] { a, b });
WriteLine();
WriteLine($"Wynik: {SharedObjects.Message}.");
WriteLine($"Czas działania: {timer.ElapsedMilliseconds:NO} ms.");
```

8. Uruchom kod. Uzyskasz wynik podobny do poniższego:

Oczekiwanie na zakończenie zadań.

.....

Wynik: BABABAABBA.

Czas działania: 6 740 ms.

Wynik dowodzi, że dwa wątki jednocześnie modyfikowały właściwość `Message`. W aplikacji z prawdziwego zdarzenia oznaczałoby to problem. Jednoczesnemu odwoływaniu się do zasobu można zapobiec przez założenie na obiekt muszli blokady dostępu na wyłączność oraz dodanie do obu metod kodu sprawdzającego tę blokadę przed próbą modyfikacji zasobu. Tę operację zaimplementujesz w następnym podrozdziale.

Zakładanie na muszlę blokady dostępu na wyłączność

Teraz użyj obiektu muszli, aby tylko jeden wątek w danej chwili miał dostęp do zasobu.

1. W pliku `SharedObjects.cs` wpisz poniższy kod tworzący obiekt, który będzie pełnił funkcję muszli:

```
public static object Conch = new();
```

2. W pliku `Program.Methods.cs` wpisz w metodach `MethodA` i `MethodB` instrukcję `lock` przed instrukcją `for`:

```
lock (SharedObjects.Conch)
{
    for (int i = 0; i < 5; i++)
    {
        Thread.Sleep(Random.Shared.Next(2000));
        SharedObjects.Message += "A";
        Write(".");
    }
}
```

Wskazówka

Dobra praktyka: zwróć uwagę, że sprawdzanie blokady muszli jest kwestią „dobrej woli”. Jeśli instrukcja `lock` zostanie użyta tylko w jednej z dwóch metod, obie będą miały jednoczesny dostęp do współdzielonego zasobu.

3. Uruchom kod. Uzyskasz wynik podobny do poniższego:

Oczekiwanie na zakończenie zadań.

.....

Wynik: BBBBBAAAAA.

Czas działania: 10 345 ms.

Tym razem kod działał dłużej, ale w każdej chwili tylko jedna z metod miała dostęp do współdzielonego zasobu. Każda z nich może być wywołana w pierwszej kolejności. Gdy jedna kończy przetwarzanie współdzielonego zasobu, zwalnia blokadę muszli, aby druga metoda mogła zacząć wykonywać swoje operacje.

Instrukcja lock

Zapewne się zastanawiasz, co robi poniższa instrukcja lock, gdy blokuje zmienną obiektową (podpowiedź: nie blokuje zasobu):

```
lock (SharedObjects.Conch)
{
    // Operacje wykonywane na współdzielonym zasobie.
}
```

Kompilator języka C# zmienia instrukcję lock na instrukcje try-finally, które używają klasy Monitor do wejścia do obiektu muszli i wyjścia z niego (mam zwyczaj nazywać to założeniem i zwolnieniem blokady muszli), jak niżej:

```
try
{
    Monitor.Enter(SharedObjects.Conch);
    // Operacje wykonywane na współdzielonym zasobie.
}
finally
{
    Monitor.Exit(SharedObjects.Conch);
}
```

Metoda Monitor.Enter sprawdza, czy wątek inny niż jej własny założył blokadę na obiekcie muszli umieszczonym w jej argumencie. Jeżeli tak, wstrzymuje działanie. W przeciwnym razie zakłada blokadę i wątek może wykonywać operacje na współdzielonym zasobie. Gdy kończy, wywołuje metodę Monitor.Exit, która zwalnia blokadę muszli. Jeżeli w tym czasie czekał inny wątek, może teraz założyć swoją blokadę i wykonać operacje. Wymagane jest przy tym, aby wszystkie wątki przestrzegały „zasady muszli”, a więc wywoływały metody, odpowiednio, Monitor.Enter i Monitor.Exit.

Wskazówka

Dobra praktyka: nie można w charakterze muszli stosować typów wartości (struct). Metoda Monitor.Enter wymaga typu referencyjnego, ponieważ blokuje adres obszaru pamięci.

Zapobieganie zakleszczeniu blokad

Sposób, w jaki kompilator zamienia instrukcję lock na wywołania metod klasy Monitor, warto znać również dlatego, aby wiedzieć, jak zapobiec zakleszczeniu blokad.

Zakleszczenie ma miejsce wtedy, gdy współdzielonych zasobów jest kilka, z każdym jest powiązana muszla wykorzystywana do kontrolowania, który wątek może wykonywać na nim operacje, i zachodzi następująca sekwencja zdarzeń:

- Wątek X blokuje muszlę A i zaczyna wykonywać operacje na współdzielonym zasobie A.
- Wątek Y blokuje muszlę B i zaczyna wykonywać operacje na współdzielonym zasobie B.
- Wątek X musi w trakcie przetwarzania zasobu A wykonać operację na zasobie B i w tym celu usiłuje założyć blokadę na muszli B. Muszla ta jednak jest zablokowana przez wątek Y.
- Wątek Y musi w trakcie przetwarzania zasobu B wykonać operację na zasobie A i w tym celu usiłuje założyć blokadę na muszli A. Muszla ta jednak jest zablokowana przez wątek X.

Jednym ze sposobów uniknięcia zakleszczenia blokad jest określenie maksymalnego czasu oczekiwania na zwolnienie blokady. W tym celu zamiast instrukcji `lock` należy jawnie użyć klasy `Monitor`.

1. W pliku `Program.Methods.cs` zastąp instrukcję `lock` wyróżnionym niżej kodem, który podejmuje próbę wejścia do obiektu muszli na określony czas. Jeżeli to się nie uda, kod wyświetla komunikat o błędzie. W obiekt może wtedy wejść inny wątek.

```
try
{
    if (Monitor.TryEnter(SharedObjects.Conch, TimeSpan.FromSeconds(15)))
    {
        for (int i = 0; i < 5; i++)
        {
            Thread.Sleep(Random.Shared.Next(2000));
            SharedObjects.Message += "A";
            Write(".");
        }
    }
    else
    {
        WriteLine("Przekroczony czas blokowania muszli przez metodę A.");
    }
}
finally
{
    Monitor.Exit(SharedObjects.Conch);
}
```

2. Uruchom kod. Powinieneś uzyskać taki sam wynik jak poprzednio (aczkolwiek muszlę jako pierwsza może przejąć każda z metod). Ten kod jest jednak lepszy od poprzedniego, ponieważ nie powoduje zakleszczenia blokad.

Wskazówka

Dobra praktyka: używaj instrukcji `lock` tylko wtedy, gdy możesz napisać kod, nie narażając się na zakleszczenia. Jeżeli nie jest to możliwe, stosuj kombinację metody `Monitor.TryEnter` i instrukcji `try-finally`, dzięki którym możesz określić maksymalny czas oczekiwania wątku na zwolnienie blokady i uniknąć potencjalnego zakleszczenia. Więcej dobrych praktyk pisania wielowątkowego kodu znajdziesz na stronie <https://learn.microsoft.com/pl-pl/dotnet/standard/threading/managed-threading-best-practices>.

Synchronizacja zdarzeń

Zdarzenia generowane przez platformę .NET nie są wątkowo bezpieczne, dlatego nie należy z nich korzystać w wielowątkowym kodzie.

Czasami programiści próbują zakładać blokady na wyłączność podczas dodawania i usuwania procedur obsługi zdarzeń, jak również podczas zgłaszania zdarzeń, na przykład:

```
// Pole delegata zdarzenia.
public event EventHandler? Shout;

// Muszla.
private object eventConch = new();

// Metoda.
public void Poke()
{
    lock (eventConch) // Zły pomysł.
    {
        // Jeżeli coś nasłuchuje...
        if (Shout != null)
        {
            // ...wywołaj delegata i zgłoś zdarzenie.
            Shout(this, EventArgs.Empty);
        }
    }
}
```

Wskazówka

Dobra praktyka: Czy to dobrze, czy źle, że programiści tak robią? Zależy to od wielu skomplikowanych czynników, więc nie sposób wydać jednoznacznej oceny. Więcej o zdarzeniach i bezpieczeństwie wątków dowiesz się na stronie <https://docs.microsoft.com/en-us/archive/blogs/cburrows/field-like-events-considered-harmful>.

O tym, jak złożone jest to zagadnienie, pisze Stephen Cleary na swoim blogu pod adresem <https://blog.stephencleary.com/2009/06/threadsafe-events.html>.

Kodowanie atomicznych operacji procesora

Termin *atomiczny* pochodzi od greckiego słowa *atomos* — niepodzielny. Znajomość pojęcia atomiczności w wielowątkowym kodzie jest ważna, ponieważ operacja nieatomiczna może zostać w trakcie jej wykonywania przerwana przez inny wątek. Na przykład czy operacja inkrementacji w poniższym kodzie jest atomiczna?

```
int x = 3;
x++; // Czy jest to atomiczna operacja procesora?
```

Nie jest! Inkrementacja liczby całkowitej wymaga wykonania przez procesor następujących kroków:

1. Załadowanie zmiennej do rejestru.
2. Inkrementacja wartości.
3. Zapisanie wartości zmiennej.

Wątek po wykonaniu dwóch pierwszych kroków może zostać przerwany przez inny wątek, który wykona wszystkie kroki. Gdy pierwszy wznowi działanie, nadpisze wartość zmiennej i w ten sposób zniweczy inkrementację dokonaną przez drugi wątek.

Istnieje typ `Interlocked`, umożliwiający wykonywanie atomicznych operacji `Add`, `Increment`, `Decrement`, `Exchange`, `CompareExchange`, `And`, `Or` i `Read` na liczbach całkowitych następujących typów:

- `System.Int32 (int)`, `System.UInt32 (uint)`,
- `System.Int64 (long)`, `System.UInt64 (ulong)`.

Nie można go jednak używać z liczbami typu `byte`, `sbyte`, `short`, `ushort` i `decimal`.

Operacje `Exchange` i `CompareExchange`, polegające na zamianie wartości zapisanych w pamięci, można wykonywać na danych następujących typów:

- `System.Single (float)`, `System.Double (double)`,
- `nint`, `nuint`,
- `T`, `System.Object (object)`.

Zobacz teraz typ `Interlocked` w akcji.

1. W klasie `SharedObjects` zadeklaruj pole, w którym będzie zapisywana liczba wykonanych operacji:

```
public static int Counter; // Inny współdzielony zasób.
```

2. W metodach `MethodA` i `MethodB` wpisz wewnątrz pętli `for`, pod instrukcją modyfikującą ciąg znaków, poniższy kod, który w bezpieczny sposób inkrementuje licznik:

```
Interlocked.Increment(ref SharedObjects.Counter);
```

3. W pliku *Program.cs*, pod instrukcją wyświetlającą czas działania programu, wpisz poniższy kod wyświetlający wartość licznika:

```
WriteLine($"Liczba modyfikacji ciągu znaków:{SharedObjects.Counter}.");
```

4. Uruchom kod. Uzyskasz wynik podobny do poniższego:

```
Oczekiwanie na zakończenie zadań.
.....
Wynik: BBBBBAAAA.
Czas działania: 12 129 ms.
Liczba modyfikacji ciągu znaków: 10.
```

Gdy się uważnie przyjrzyjiesz, zauważysz, że chronione są wszystkie zasoby, do których kod odwołuje się w bloku zabezpieczonym z użyciem obiektu *muszli*. Zatem w tym konkretnym przypadku użycie typu *Interlocked* nie jest konieczne. Inaczej byłoby, gdyby nie był chroniony inny współdzielony zasób, na przykład pole *Message*.

Inne techniki synchronizacji dostępu

Monitor i *Interlocked* to wzajemnie wykluczające się blokady dostępu na wyłączność. Są proste i skuteczne, ale czasami potrzebne są bardziej zaawansowane techniki synchronizowania dostępu do współdzielonego zasobu. Są one opisane w poniższej tabeli.

Typ	Opis
<i>ReaderWriterLock</i> , <i>ReaderWriterLockSlim</i>	Kilka wątków może działać w trybie odczytu. Jeden wątek może działać w trybie zapisu z blokadą zasobu na wyłączność. Wątek działający w trybie odczytu może przejść w tryb zapisu bez rezygnacji z dostępu do odczytu
<i>Mutex</i>	Typ ten, podobnie jak <i>Monitor</i> , daje wyłączny dostęp do udostępnionego zasobu, chyba że jest on używany do synchronizowania dostępu między procesami
<i>Semaphore</i> , <i>SemaphoreSlim</i>	Ograniczenie za pomocą slotów liczby wątków, które mogą jednocześnie uzyskiwać dostęp do zasobu lub puli zasobów
<i>AutoResetEvent</i> , <i>ManualResetEvent</i>	Procedury obsługi zdarzeń wykorzystywane przez wątki do synchronizowania swoich działań przez wysyłanie do siebie nawzajem sygnałów i oczekiwanie na nie

Instrukcje *async* i *await*

W wersji języka C# 5 wprowadzono dwie instrukcje wykorzystujące typ *Task*, szczególnie użyteczne w następujących sytuacjach:

- implementacja wielozadaniowości w graficznym interfejsie użytkownika,
- skalowanie aplikacji i usług internetowych.

W rozdziale 18., „Tworzenie aplikacji mobilnych i stacjonarnych z użyciem .NET MAUI”, dowiesz się, jak z użyciem instrukcji `async` i `await` implementować wielozadaniowość w graficznym interfejsie użytkownika.

Teraz poznaj podstawy teoretyczne uzasadniające wprowadzenie tych instrukcji, a później zastosuj je w praktyce.

Poprawa responsywności aplikacji konsolowej

Instrukcję `await` można stosować wyłącznie w metodach asynchronicznych. W wersjach języka C# 7 i starszych nie można było oznaczać instrukcją `async` metody `Main` w aplikacji konsolowej. Na szczęście jedną z nowości wprowadzonych w wersji C# 7.1 jest możliwość deklarowania tej metody jako asynchronicznej.

1. W wybranym edytorze kodu otwórz rozwiązanie/ obszar roboczy o nazwie *Chapter04* i dodaj do niego aplikację konsolową o nazwie *AsyncConsole*.
 - W środowisku Visual Studio 2022 oznacz bieżący projekt jako startowy.
 - W środowisku Visual Studio Code ustaw *AsyncConsole* jako aktywny projekt OmniSharp.

2. Usuń z pliku *Program.cs* istniejące instrukcje, zaimportuj statycznie przestrzeń nazw `System.Console` i wpisz poniższy kod, który tworzy instancję klasy `HttpClient`, wysyła zapytanie na adres strony Apple i wyświetla liczbę odebranych bajtów:

```
using static System.Console;

HttpClient client = new();

HttpResponseMessage response =
    await client.GetAsync("http://www.apple.com/");

WriteLine("Wielkość strony głównej Apple w bajtach: {0:N0}.",
    response.Content.Headers.ContentLength);
```

3. Uruchom kompilację projektu i zwróć uwagę, że zostanie wykonana pomyślnie. W wersjach platformy .NET 5 i starszych szablon projektu zawierał jawną klasę `Program` z synchroniczną metodą `Main`, więc próba kompilacji kończyła się wyświetleniem następującego komunikatu:

```
Program.cs(14,9): error CS4033: The 'await' operator can only be used
within an async method. Consider marking this method with the 'async'
modifier and changing its return type to 'Task'. [/Users/markjprice/
appsservices-net7/Chapter04/AsyncConsole/AsyncConsole.csproj]
```

(Operatora 'await' można używać tylko w metodzie asynchronicznej. Oznacz tę metodę modyfikatorem 'async' i zmień typ zwracanego przez nią wyniku na 'Task').

4. Począwszy od wersji .NET 6 szablon aplikacji konsolowej wykorzystuje funkcje najwyższego poziomu i automatycznie definiuje klasę Program z asynchroniczną metodą <Main>\$.
5. Uruchom kod. Możesz uzyskać inny wynik od poniższego, ponieważ Apple często zmienia swoją stronę główną.

Wielkość strony głównej Apple w bajtach: 162 480.

Operacje na strumieniach asynchronicznych

Wraz z wersją platformy .NET Core 3.0 pojawiła się możliwość wykonywania asynchronicznych operacji na strumieniach.

Uwaga

Na stronie <https://learn.microsoft.com/pl-pl/dotnet/csharp/asynchronous-programming/generate-consume-asynchronous-stream> znajduje się przewodnik po strumieniach asynchronicznych.

W starszych wersjach języka i platformy niż, odpowiednio, C# 8.0 i .NET Core 3.0 instrukcję `await` można było stosować tylko w metodach zwracających wartości skalarne. W platformie .NET Standard 2.1 metoda asynchroniczna może zwracać wartości z użyciem strumienia.

Przeanalizujemy przykładową metodę, która zwraca trzy losowe liczby całkowite jako strumień asynchroniczny.

1. W wybranym edytorze kodu otwórz rozwiązanie/ obszar roboczy o nazwie *Chapter04* i dodaj do niego aplikację konsolową o nazwie *AsyncEnumerable*.
 - W środowisku Visual Studio 2022 oznacz bieżący projekt jako startowy.
 - W środowisku Visual Studio Code ustaw *AsyncEnumerable* jako aktywny projekt OmniSharp.
2. Zaimportuj globalnie i statycznie przestrzeń nazw `System.Console`.
3. Usuń z pliku *Program.cs* istniejące instrukcje i wpisz poniższy kod, który definiuje metodę zwracającą asynchronicznie za pomocą instrukcji `yield` trzy losowe liczby całkowite:

```
async static IEnumerable<int> GetNumbersAsync()  
{  
    Random r = Random.Shared;
```

```
// Symulacja działania.  
await Task.Delay(r.Next(1500, 3000));  
yield return r.Next(0, 1001);  
  
await Task.Delay(r.Next(1500, 3000));  
yield return r.Next(0, 1001);  
await Task.Delay(r.Next(1500, 3000));  
yield return r.Next(0, 1001);  
}
```

4. Powyżej metody `GetNumbersAsync` wpisz kod wyświetlający sekwencję liczb:

```
await foreach (int number in GetNumbersAsync())  
{  
    WriteLine($"Liczba: {number}");  
}
```

5. Uruchom kod. Uzyskasz wynik podobny do poniższego:

```
Liczba: 820  
Liczba: 70  
Liczba: 629
```

Poprawa responsywności aplikacji graficznej

Do tej pory podczas pracy z tą książką tworzyłeś wyłącznie aplikacje konsolowe. Życie programisty jest trudniejsze, gdy musi kodować aplikacje i usługi internetowe, programy z graficznym interfejsem użytkownika, na przykład dla komputerów z systemem Windows i urządzeń przenośnych. Jest tak m.in. dlatego, że aplikacja graficzna wykorzystuje specjalny wątek UI (ang. *user interface*, interfejs użytkownika). Obowiązują przy tym dwie zasady:

- Wątek ten nie może wykonywać długotrwałych zadań.
- Do obiektów graficznych może się odwoływać tylko wątek UI.

Wcześniej, aby spełnić te wymagania, programiści musieli pisać złożony kod, w którym długotrwałe zadania były wykonywane przez dodatkowy wątek, a ten po zakończeniu działania przekazywał wyniki wątkowi UI w celu przedstawienia ich użytkownikowi. W takim kodzie bałagan pojawiał się bardzo szybko.

Na szczęście począwszy od wersji języka C# 5 można korzystać z instrukcji `async` i `await`. Dzięki nim można pisać kod, który wygląda jak synchroniczny, jest czysty i zrozumiały. Jednak pod spodem kompilator C# tworzy złożoną maszynę stanów i śledzi uruchomione wątki. To niemal czary! Kombinacja tych dwóch instrukcji sprawia, że metoda asynchroniczna jest wywoływana w wątku roboczym, a po zakończeniu działania zwraca wyniki do wątku UI.

Przeanalizujemy przykład aplikacji przeznaczonej dla komputerów stacjonarnych, opartej na platformie WPF. Aplikacja ta będzie odczytywała z bazy SQL Server *Northwind* dane pracowników z wykorzystaniem niskopoziomowych typów `SqlConnection`, `SqlCommand` i `SqlDataReader`.

Uwaga

Northwind jest średnio złożoną bazą danych, zawierającą przyzwoitą liczbę przykładowych rekordów. Poznałeś ją, skonfigurowałeś i intensywnie wykorzystywałeś w rozdziale 2., „Zarządzanie relacyjną bazą danych z użyciem oprogramowania SQL Server”.

Wskazówka

Opisany przykład będziesz mógł wykonać, jeżeli korzystasz z systemu Microsoft Windows i masz bazę Microsoft SQL Server *Northwind*. To jedyny rozdział książki, w którym aplikacja nie jest wieloplatformowa i nowoczesna (platforma WPF ma już 17 lat!). Możesz użyć środowiska Visual Studio 2022 lub Visual Studio Code.

Teraz skupimy się na responsywności aplikacji graficznej. Język XAML i techniki tworzenia uniwersalnych aplikacji graficznych poznasz w rozdziale 18., „Tworzenie aplikacji mobilnych i stacjonarnych z użyciem .NET MAUI”. Ponieważ nigdzie indziej w tej książce nie będziemy się zajmować platformą WPF, uznałem, że teraz będzie dobra okazja, aby poznać — choćby ogólnie — opartą na niej przykładową aplikację.

Do dzieła!

1. Jeżeli używasz środowiska Visual Studio 2022, utwórz w rozwiązaniu/obszarze roboczym *Chapter04* projekt aplikacji WPF (C#) o nazwie *WpfResponsive*. Jeżeli zaś korzystasz ze środowiska Visual Studio Code, utwórz nowy projekt za pomocą polecenia `dotnet new wpf` i ustaw go jako aktywny projekt OmniSharp.
2. Dodaj w projekcie odwołanie do pakietu `Microsoft.Data.SqlClient`.
3. Otwórz plik konfiguracyjny projektu i zwróć uwagę, że docelowa aplikacja jest typu Windows EXE oraz że wykorzystuje ona platformy .NET 7 dla Windows (nie uruchomi się w systemie macOS ani Linux) i WPF.

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net7.0-windows</TargetFramework>
    <Nullable>enable</Nullable>
    <UseWPF>true</UseWPF>
  </PropertyGroup>
```

```
<ItemGroup>
  <PackageReference Include="Microsoft.Data.SqlClient" Version="5.0.0" />
</ItemGroup>

</Project>
```

4. Skompiluj projekt, aby został pobrany wymagany pakiet.
5. Otwórz plik *MainWindow.xaml* i w sekcji <Grid> wpisz poniższy kod definiujący dwa przyciski, pole tekstowe i listę. Elementy te będą ułożone pionowo, jeden nad drugim:

```
<StackPanel>
  <Button Name="GetEmployeesSyncButton"
    Click="GetEmployeesSyncButton_Click">

    Synchroniczny odczyt danych pracowników</Button>
  <Button Name="GetEmployeesAsyncButton"
    Click="GetEmployeesAsyncButton_Click">

    Asynchroniczny odczyt danych pracowników </Button>
  <TextBox HorizontalAlignment="Stretch" Text="Wpisz tutaj" />
  <ListBox Name="EmployeesListBox" Height="350" />
</StackPanel>
```

Uwaga

Środowisko Visual Studio 2022 dla systemu Windows, w odróżnieniu od Visual Studio Code, dobrze współpracuje z platformą WPF i oferuje funkcję IntelliSense ułatwiającą edytowanie kodu i znaczników XAML.

6. W pliku *MainWindow.xaml.cs* wpisz instrukcje importujące przestrzenie nazw **System.Diagnostics** i **Microsoft.Data.SqlClient**.
7. W klasie *MainWindow* wpisz poniższy kod definiujący dwie stałe tekstowe zawierające, odpowiednio, ciąg połączenia z bazą danych oraz polecenie SQL:

```
private const string connectionString =
  "Data Source=.;" +
  "Initial Catalog=Northwind;" +
  "Integrated Security=true;" +
  "Encrypt=false;" +
  "MultipleActiveResultSets=true;";

private const string sql =
  "WAITFOR DELAY '00:00:05';" +
  "SELECT EmployeeId, FirstName, LastName FROM Employees";
```

8. Wpisz poniższy kod, który definiuje metody obsługujące zdarzenia kliknięcia przycisków. Metody te wykorzystują powyższe stałe tekstowe do połączenia się z bazą *Northwind*, odczytania identyfikatorów i nazwisk pracowników, a następnie wyświetlenia ich w liście.

```
private void GetEmployeesSyncButton_Click(object sender, RoutedEventArgs e)
{
    Stopwatch timer = Stopwatch.StartNew();
    using (SqlConnection connection = new(connectionString))
    {
        try
        {
            connection.Open();

            SqlCommand command = new(sql, connection);
            SqlDataReader reader = command.ExecuteReader();

            while (reader.Read())
            {
                string employee = string.Format("{0}: {1} {2}",
                    reader.GetInt32(0), reader.GetString(1), reader.GetString(2));

                EmployeesListBox.Items.Add(employee);
            }

            reader.Close();
            connection.Close();
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }
    EmployeesListBox.Items.Add($"Czas odczytu synchronicznego:
{timer.ElapsedMilliseconds:N0} ms");
}

private async void GetEmployeesAsyncButton_Click(
    object sender, RoutedEventArgs e)
{
    Stopwatch timer = Stopwatch.StartNew();

    using (SqlConnection connection = new(connectionString))
    {
        try
        {
            await connection.OpenAsync();

            SqlCommand command = new(sql, connection);
            SqlDataReader reader = await command.ExecuteReaderAsync();

            while (await reader.ReadAsync())
            {
                string employee = string.Format("{0}: {1} {2}",
                    await reader.GetFieldValueAsync<int>(0),
                    await reader.GetFieldValueAsync<string>(1),
```



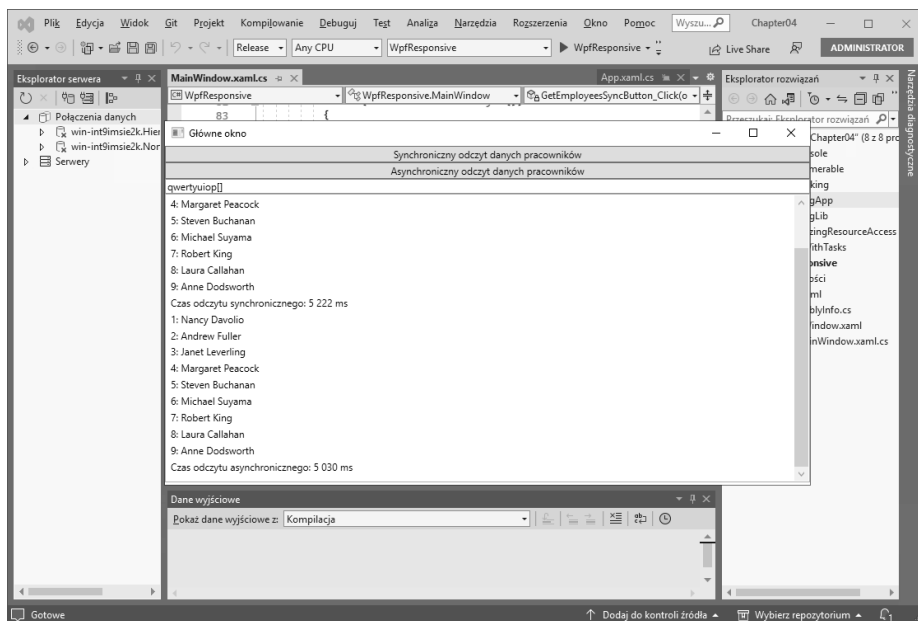
```
        await reader.GetFieldValueAsync<string>(2);

        EmployeesListBox.Items.Add(employee);
    }
    await reader.CloseAsync();
    await connection.CloseAsync();
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
}
EmployeesListBox.Items.Add($"Czas odczytu asynchronicznego:
{timer.ElapsedMilliseconds:NO} ms");
}
```

Zwróć uwagę na następujące kwestie:

- Definiowanie metody `async void` jest złą praktyką „wywołaj i zapomnij”. Nie uzyska się powiadomienia o jej zakończeniu i nie ma możliwości przerwania jej działania, ponieważ nie zwraca ona wyniku typu `Task` ani `Task<T>`, który dawałby nad nią kontrolę.
 - Polecenie SQL symuluje pracę przez 5 sekund z wykorzystaniem instrukcji `WAITFOR DELAY`, a następnie odczytuje rekordy z trzech kolumn tabeli `Employees`.
 - Metoda `GetEmployeesSyncButton_Click`, obsługująca zdarzenie kliknięcia przycisku, wykorzystuje synchroniczne metody do nawiązania połączenia z bazą i odczytania danych pracowników.
 - Metoda `GetEmployeesAsyncButton_Click`, obsługująca zdarzenie kliknięcia przycisku, wykorzystuje asynchroniczne metody oznaczone instrukcją `async` do nawiązania połączenia z bazą i odczytania danych pracowników.
 - Obie powyższe metody wykorzystują klasę `Stopwatch` do zarejestrowania czasu działania, a następnie wyświetlenia go w liście.
9. Uruchom aplikację w trybie *Release* bez debugowania.
 10. Kliknij pole tekstowe i wpisz w nim dowolny ciąg. Zwróć uwagę, że interfejs graficzny jest responsywny.
 11. Kliknij przycisk *Synchroniczny odczyt danych pracowników*.
 12. Kliknij pole tekstowe i zwróć uwagę, że interfejs graficzny nie jest responsywny.
 13. Zaczekaj 5 sekund, aż w liście pojawią się dane pracowników.
 14. Ponownie kliknij pole tekstowe i wpisz w nim dowolną treść. Interfejs z powrotem jest responsywny.
 15. Kliknij przycisk *Asynchroniczny odczyt danych pracowników*.

16. Kliknij pole tekstowe i wpisz w nim dowolną treść. Zwróć uwagę, że w trakcie wykonywania operacji interfejs graficzny jest responsywny. Wpisuj znaki, aż pojawią się dane pracowników (patrz rysunek 4.2).



Rysunek 4.2. Synchroniczny i asynchroniczny odczyt danych w aplikacji WPF

17. Zwróć uwagę na różne czasy wykonania obu operacji. Na czas synchronicznego odczytywania danych interfejs graficzny jest blokowany, natomiast w trakcie odczytu asynchronicznego jest responsywny.
18. Zamknij aplikację.

Poprawa skalowalności aplikacji i usług internetowych

Instrukcje `async` i `await` można również stosować w kodzie generującym strony internetowe, w aplikacjach i usługach uruchamianych na serwerach. Z perspektywy aplikacji klienckiej nic się nie zmienia (może z wyjątkiem nieznacznego wydłużenia czasu przetwarzania zapytania). Natomiast efektem implementacji wielozadaniowości na serwerze za pomocą instrukcji `async` i `await` mogą być gorsze wrażenia użytkownika.

Po stronie serwera tworzone są dodatkowe, tańsze wątki robocze, które czekają na zakończenie wykonania długotrwałych zadań. Dzięki temu drogie wątki wykonujące operacje wejścia/wyjścia nie są blokowane i mogą obsługiwać inne żądania klientów.

Poprawia to ogólną skalowalność aplikacji lub usługi internetowej, która może obsługiwać większą liczbę klientów jednocześnie.

Popularne typy obsługujące wielozadaniowość

Istnieje wiele popularnych typów zawierających metody asynchroniczne, które można stosować z instrukcją `await`. Przedstawia je poniższa tabela.

Typ	Metody
<code>DbContext<T></code>	<code>AddAsync</code> , <code>AddRangeAsync</code> , <code>FindAsync</code> , <code>SaveChangesAsync</code>
<code>DbSet<T></code>	<code>AddAsync</code> , <code>AddRangeAsync</code> , <code>ForEachAsync</code> , <code>SumAsync</code> , <code>ToListAsync</code> , <code>ToDictionaryAsync</code> , <code>AverageAsync</code> , <code>CountAsync</code>
<code>HttpClient</code>	<code>GetAsync</code> , <code>PostAsync</code> , <code>PutAsync</code> , <code>DeleteAsync</code> , <code>SendAsync</code>
<code>StreamReader</code>	<code>ReadAsync</code> , <code>ReadLineAsync</code> , <code>ReadToEndAsync</code>
<code>StreamWriter</code>	<code>WriteAsync</code> , <code>WriteLineAsync</code> , <code>FlushAsync</code>

Wskazówka

Dobra praktyka: widząc metodę o nazwie zakończonej przyrostkiem `Async`, zawsze sprawdzaj, czy zwraca ona wynik typu `Task` lub `Task<T>`. Jeżeli tak, możesz jej używać zamiast synchronicznej metody z innym sufiksem w nazwie. Pamiętaj, aby wywoływać ją z użyciem instrukcji `await`, a swoją metodę oznaczać instrukcją `async`.

Instrukcja `await` w bloku `catch`

W wersji języka C# 5 instrukcję `await` można było stosować w bloku `try`, ale nie w `catch`. Począwszy od wersji C# 6 można to robić w obu blokach.

Nauka i praktyka

Sprawdź swoją wiedzę i umiejętności: odpowiedz na kilka pytań, wykonaj praktyczne ćwiczenia i pogłęb wiedzę na tematy omówione w tym rozdziale.

Ćwiczenie 4.1. Sprawdź swoją wiedzę

Odpowiedz na poniższe pytania:

1. Jakie informacje można pozyskać o procesie?
2. Jak dokładna jest klasa `Stopwatch`?

3. Jaki sufiks powinna mieć zgodnie z przyjętą konwencją nazwa metody zwracającej wynik typu `Task` lub `Task<T>`?
4. Jakiej instrukcji należy użyć w deklaracji metody, aby w jej wnętrzu można było stosować instrukcję `await`?
5. Jak się tworzy zadanie podrzędne?
6. Dlaczego należy unikać używania instrukcji `lock`?
7. Kiedy należy stosować klasę `Interlocked`?
8. Kiedy należy stosować klasę `Mutex` zamiast `Monitor`?
9. Jakie są zalety stosowania instrukcji `async` i `await` w kodzie aplikacji lub usługi internetowej?
10. Czy można przerwać zadanie? Jeżeli tak, to w jaki sposób?

Ćwiczenie 4.2. Zbadaj temat

Odwiedź stronę <https://github.com/markjprice/apps-services-net7/blob/main/book-links.md#chapter-4---improving-performance-and-scalability-using-multitasking>, aby dowiedzieć się więcej na tematy poruszone w tym rozdziale.

Ćwiczenie 4.3. Dowiedz się więcej o programowaniu równoległym

Tematy poruszone w tym rozdziale są dokładniej omówione w książce *Parallel Programming and Concurrency with C# 10 and .NET 6: A modern approach to building faster, more responsive, and asynchronous .NET applications using C#* (Programowanie równoległe i współbieżność w C# 10 i .NET 6: nowoczesne techniki tworzenia szybszych, responsywnych i asynchronicznych aplikacji .NET z użyciem C#), Alvin Ashcraft, wyd. Packt (<https://www.packtpub.com/product/parallel-programming-and-concurrency-with-c-10-andnet-6/9781803243672>).

Podsumowanie

W tym rozdziale nauczyłeś się:

- definiować i uruchamiać zadania,
- kodować oczekiwanie na zakończenie jednego lub kilku zadań,
- kontrolować kolejność wykonywania zadań,

- synchronizować dostęp do współdzielonych zasobów,
- stosować magiczne instrukcje `async` i `await`.

W następnym rozdziale dowiesz się, jak korzystać z kilku popularnych zewnętrznych bibliotek.

Skorowidz |

- .NET, 58
 - technologie tworzenia aplikacji, 802
 - utrzymanie platformy, 58
- .NET 5, 64
- .NET 6, 66
- .NET 7, 69
- .NET Core 3, 60
- .NET MAUI, 41, 693
 - dostęp
 - do informacji o urządzeniu, 790
 - do lokalnego systemu plików, 781
 - do systemowego schowka, 778
 - formanty, 697
 - instalacja ręczna, 700
 - integracja z paskiem menu, 794
 - komponenty interfejsu graficznego, 702
 - obsługa wyskakujących powiadomień, 797
 - przestrzenie nazw, 696
 - tworzenie
 - okien, 788
 - powłoki i stron, 759
 - tworzenie aplikacji, 705
 - dodanie nawigacji, 712
 - dodanie stron z treścią, 712
 - implementacja nowych stron, 718
 - mobilnych, 699
 - współdzielenie zasobów, 720
 - aplikacji, 721
 - dynamiczna wymiana, 722
 - formantu, 720
 - odwołania, 722
 - strony, 720
 - zewnętrzne biblioteki formantów, 798
- .NET MAUI Blazor
 - aplikacje uniwersalne, 758
 - aplikacje hybrydowe, 757
- .NET MAUI Community Toolkit, 797
- .NET Runtime, 59
- .NET SDK, 59

A

- ACID, Atomic, Consistent, Isolated, Durable, 188
- ADO.NET
 - odczytywanie danych, 107
 - operacje asynchroniczne, 109
 - tworzenie aplikacji konsolowych, 102
 - typy danych, 101
 - uruchamianie procedur składowanych, 110
 - zapytania, 107
- adres URL, 35
- AJAX, 499
- alarm
 - tworzenie komponentu, 632
- algorytmy
 - szyfrowania
 - AES, 330
 - asymetryczne, 327
 - RSA, 341
 - SHA256, 337, 341
 - symetryczne, 327
 - ochrony danych, 326
 - skraccające dane, 336
- analizator
 - kodu, 52
 - kompatybilności przeglądarki, 614
- Android SDK, 705
- aplikacje
 - .NET MAUI
 - testowanie, 741
 - asynchroniczne wykonywanie zadań, 205
 - chmurowe, 44
 - hybrydowe, 756
 - konsolowe, 102
 - lokalizacja, 313
 - lokalne testowanie, 705
 - mobilne, 699
 - komunikacja z usługą, 744

- aplikacje
 - monitorowanie wydajności, 195
 - ochrona, 325
 - ochrona funkcjonalności, 352
 - poprawa responsywności, 225
 - poprawa skalowalności, 230
 - PWA, 651
 - tryb offline, 655
 - responsywność, 223, 225
 - strumieniowe .NET, 519
 - synchronizacja dostępu, 215
 - uniwersalne, 44, 46, 707, 758
 - wdrażanie, 614
 - ARS, Atom-Record-Sequence, 154
 - asercja, 250
 - testująca ciąg znaków, 251
 - testująca datę, 252
 - testująca kolekcję, 252
 - ASP.NET Core, 35
 - Blazor WebAssembly, 33
 - Identity, 574
 - lokalizowanie i globalizowanie witryny, 588
 - Minimal API, 357, 359
 - MVC, 36
 - biblioteka klas EF Core, 574
 - eksploracja witryny, 569
 - interfejs użytkownika, 575
 - klient usługi GraphQL, 452
 - kontekst bazy danych, 574
 - metody HTML Helpers, 584
 - przegląd struktury projektu, 572
 - rejestracja użytkowników, 571
 - tworzenie witryny, 567
 - Razor Pages, 35
 - SignalR, 500
 - Web API, 361
 - tworzenie usługi, 361
 - asynchroniczne
 - wykonywanie operacji, 208
 - odczytywanie danych, 230
 - atak
 - DDoS, 383
 - DoS, 383
 - atomiczne operacje procesora, 221
 - atom-rekord-sekwencja, 154
 - atrybuty, 268
 - tworzenie, 272
 - ATS, App Transport Security, 744
 - AutoMapper, 243
 - autoryzacja, 326, 346, 347, 353
 - funkcji Azure, 535
 - implementacja, 349
 - użycie tokenu JWT, 395
 - Avalonia, 42
 - Azure
 - funkcje, 525
 - zasoby, 51
 - usuwanie zasobów, 190, 565
 - Azure Cosmos DB, 154
 - aprowizacja przepływności, 159
 - Data Migration, 161
 - Emulator, 162
 - hierarchia komponentów, 158
 - interfejsy API, 154
 - modelowanie dokumentów, 155
 - poziomy spójności, 157
 - programowanie serwera, 188
 - przenoszenie danych do bazy, 161
 - strategie partycjonowania, 160
 - tworzenie zasobów
 - w chmurze, 167
 - z użyciem emulatora, 161
 - za pomocą aplikacji .NET, 171
 - zapytania SQL, 185
 - zarządzanie bazą NoSQL, 153
 - Azure Functions Core Tools, 536
 - Azure SignalR Service, 500
 - Azure SQL Database
 - konfiguracja usługi, 86
 - nawiązywanie połączenia, 90
 - tworzenie serwera bazy danych, 87
 - zwalnianie zasobów, 149
 - Azure SQL Edge
 - instalacja, 91
 - nawiązywanie połączenia, 93
 - Azurite, 534
- ## B
- Banana Cake Pop, 433, 437, 438, 448
 - ustawienia połączenia, 438
 - baza danych
 - Azure Cosmos DB, 154
 - NoSQL, 153
 - Benchmark.NET, 201
 - bezpieczeństwo danych, 326
 - biblioteka, 32
 - AutoMapper, 244
 - Bootstrap, 577
 - FluentValidation, 254
 - QuestPDF, 260

- Radzen Blazor, 659
 - Serilog, 240
 - SignalR JavaScript, 507
 - Skiasharp, 260
 - biblioteki
 - dla kontekstu danych SQL Server, 141
 - dla modelu jednostek SQL Server, 140
 - do generowania plików PDF, 260
 - do przekształcania obrazów, 240
 - do rysowania, 240
 - formantów dla .NET MAUI Blazor, 798
 - otwarte komponentów Blazor, 658
 - popularnych
 - modeli jednostek EF Core, 33
 - typów danych, 33
 - testowanie integracji, 146
 - zewnętrzne, 234
 - binarne ramkowanie danych, 472
 - Blazor, 36
 - Hybrid, 613
 - implementacja bufora w pamięci przeglądarki, 644
 - interakcje ze skryptami JavaScript, 615, 644
 - izolowanie kodów CSS, 615
 - klasa komponentów, 620
 - komponent, 615
 - alarmu, 632
 - danych, 635
 - okna dialogowego, 629
 - paska postępu, 627
 - strony, 636
 - tworzenie, 622
 - modele hostingu, 613
 - odczytywanie wartości parametrów, 619
 - odnośniki do komponentów, 621
 - parametry w ścieżce, 618, 619
 - Server, 613
 - usługa
 - pobierająca dane, 638
 - wykorzystująca pamięć lokalną, 645
 - WebAssembly, 612, 613
 - sprawdzanie zgodności przeglądarki, 614
 - wdrażanie aplikacji, 614
 - zapytania do komponentów, 616
 - blokada dostępu, 217
 - zapobieganie zakleszczeniu, 218
 - bloki danych, 327
 - błędy, 34
 - Bootstrap
 - alarmy, 583
 - motywy kolorów, 579
 - plakietki, 582
 - prototypowanie strony, 577
 - przyciski i odnośniki, 581
 - punkty graniczne, 577
 - tabele, 580
 - wiersze i kolumny, 578
 - bufor, 601
 - buforowanie, 606
- ## C
- C#, 58
 - C# 8
 - deklaracje, 61
 - indeksy i zakresy, 63
 - wyrażenie switch, 60
 - zerowalne typy referencyjne, 62
 - C# 9
 - celowa instrukcja new, 66
 - instrukcja init, 64
 - instrukcje najwyższego poziomu, 65
 - typy rekordowe, 64
 - C# 10
 - instrukcje najwyższego poziomu, 66
 - niejawnie importowane przestrzenie nazw, 66
 - sprawdzanie pustych argumentów metod, 68
 - C# 11
 - operacje matematyczne na typach generycznych, 71
 - podstawowe literały tekstowe, 70
 - Cassandra API, 154
 - ChilliCream GraphQL, 433
 - chmura
 - publikowanie projektu funkcji Azure, 560
 - Core (SQL) API, 154
 - operacje CRUD, 176
 - CORS, Cross-Origin Resource Sharing, 380
 - CRUD, Create, Read, Update, Delete, 159, 176
 - czat, 508
 - rejestracja nowego użytkownika, 512
 - test, 512

D

dane grafowe, 190
 data i czas, 290
 definiowanie, 291
 działania, 293, 297
 formatowanie, 292
 globalizacja, 294
 DDL, Data Definition Language, 99
 DDoS, Distributed Denial of Service, 383
 definiowanie
 interfejsu użytkownika, 575, 595
 kontraktów, 471
 deszyfrowanie, 329
 DHTML, Dynamic Hypertext Markup Language, 499
 Diagnostics, 196
 DML, Data Manipulation Language, 96
 instrukcja SELECT, 96
 zapytania, 99
 Docker, 91
 Azure SQL Edge, 92
 zwalnianie zasobów, 149
 dokument
 JSON, 155, 166
 wymagań produktowych, 804
 dokumentacja OpenAPI, 371
 DoS, Denial of Service, 383
 dostęp
 blokowanie, 217
 do współdzielonych zasobów, 215
 synchronizowanie, 215
 techniki synchronizacji, 222
 drzewo wyrażeń, 280
 komponenty, 281
 uruchamianie, 282
 DTO, Data Transfer Object, 243
 dymek, 662, 663
 dynamiczna wymiana zasobów, 722
 dynamiczne
 ładowanie zestawów, 275
 monitorowanie, 268
 dynamiczny hipertekstowy język znaczników, DHTML, 499
 działania
 na dacie, 297
 na datach i czasie, 293
 dziedziczenie, 127
 dziennik, 416

E

edytor HTML, 674
 EF Core, Entity Framework Core, 78, 114
 definiowanie modelu
 bazy, 119
 jednostek, 115
 OData, 404
 mapowanie hierarchii dziedziczenia, 127
 tworzenie modelu jednostek, 114
 użycie
 attributów adnotacyjnych, 116
 konwencji, 115
 wysyłanie zapytań, 124
 zapytania GraphQL, 441
 zarządzanie bazą SQL Server, 113

F

Fluent API
 definiowanie modelu jednostek, 118
 inicjowanie bazy danych, 118
 formant
 Editor, 703
 Entry, 703
 ListView, 703
 Shell, 702
 formanty platformy .NET MAUI, 697
 format ARS, 154
 formatowanie daty i czasu, 292
 formularze, 606, 680
 funkcja PBKDF2, 328
 funkcje Azure, 37, 525
 implementacja, 542
 modele hostingu, 533
 obsługa języków, 532
 operowanie na kolejkach, 552
 plany hostingu, 533
 powiązania, 526
 poziomy autoryzacji, 535
 przeгляд projektów, 540
 publikowanie w chmurze, 560
 test funkcji, 534, 543, 548, 558
 usługi magazynowe, 534
 usuwanie zasobów, 565
 używanie wiersza poleceń, 539
 w środowisku
 Visual Studio 2022, 536
 Visual Studio Code, 537
 wersje środowiska, 532

wstrzykiwanie zależności, 535
wyrażenie NCRONTAB, 527
wyzwalacze, 526
wyzwalanie czasomierzem, 545

G

generator kodu źródłowego, 283
 implementacja, 283
generowanie
 klucza, 328
 liczb losowych, 343, 345
 plików PDF, 260
 aplikacja, 263
 biblioteka klas, 260
 wektora inicjującego, 328
GitHub, 72, 100
GitHub Codespaces, 44
 tworzenie aplikacji chmurowych, 44
globalizacja, 305
 daty i czasu, 294
 witryny, 588
GraphQL, 37, 38, 430
 definiowanie modeli danych, 435
 format zapytania, 431
 funkcjonalności języka, 433
 implementowanie mutacji, 464
 klient
 .NET, 448
 ASP.NET Core MVC, 452
 nazwy
 pól, 439
 zapytań, 439
 obsługa języka, 434
 uruchamianie zapytań, 438
 zapytania w modelu EF Core, 441
Green Donut, 433
Gremlin API, 155
 przetwarzanie danych grafowych, 190
gRPC, Google Remote Procedure Call, 36, 38, 471
 czasowy limit wywołania, 489
 implementacja
 klienta, 485
 usługi, 482
 metadane o wywołaniu usługi, 488
 pakiety, 473
 rodzaje metod, 472
 testowanie usługi i klienta, 481
 tworzenie
 klienta, 477

 mikrousługi, 470
 usługi, 474
gRPC JSON, 493, 494
 implementacja transkodowania, 492
 porównanie transkodowania z gRPC-Web, 496
 test transkodowania, 494
 włączenie transkodowania, 493
gRPC-Web, 496
gwarantowany poziom świadczenia usług, 157

H

hosting funkcji Azure, 533
Hot Chocolate, 433, 441
HTML Helpers, 584, 596

I

IDE, Integrated Development Environment, 43
ikona, 669
IL, Intermediate Language, 268
implementacja
 bufora w pamięci przeglądarki, 644
 funkcji
 Azure, 542
 operującej na kolejkach, 552
 wyzwalanej czasomierzem, 545
 generatora kodu, 283
 klienta gRPC, 485
 mutacji, 464
 stron, 718
 transkodowania gRPC JSON, 492
 trybu offline, 655
 usługi gRPC, 482
 uwierzytelnienia i autoryzacji, 349
 wzorca MVVM, 768
informacje o urządzeniu Android, 790, 795
instalacja
 Azure Functions Core Tools, 536
 Azure SQL Edge, 91
 platformy .NET MAUI, 700
instrukcja
 async, 222
 await, 222
 lock, 218
 SELECT, 98
IntelliSense, 53

interfejs

- API, 36, 154
 - minimalistyczny, 358, 638
 - Cassandra API, 154
 - Core (SQL) API, 154, 176
 - Fluent API, 118
 - Gremlin API, 155, 190
 - INotifyPropertyChanged, 729
 - Minimal API, 744, 764
 - MongoDB API, 154
 - użytkownika
 - aplikacji wieloplatformowej, MAUI, 693
 - prototypowanie, 577
 - tworzenie, 575, 595
- internacjonalizacja, 290, 305

J

język

- C#, 58
 - DDL, 99
 - DHTML, 499
 - DML, 96
 - GraphQL, 430
 - Transact-SQL, 94
 - XAML, 40, 693
- JWT, JSON Web Token, 395

K

klasa

- ComponentBase, 620
- ObservableCollection, 731
- Task, 208
- Thread, 205, 208
- WebApplication, 359

klient

- .NET
 - dla usługi GraphQL, 448
 - dla usługi SignalR, 515
- ASP.NET Core MVC
 - dla usługi GraphQL, 452
- gRPC, 33, 477

klucze, 326

- współdzielone, 327

kod IL, 268

kolejki, 552

koligacja sesji, 601

komponent

- reprezentujący
 - dymek, 662, 663

edytor HTML, 674

- formularz, 680
- ikonę, 669
- menu kontekstowe, 662, 663
- obraz, 669
- okno dialogowe, 662, 665
- powiadomienia, 662, 665
- stronę Pracownicy, 688
- wykres, 676
- zakładkę, 669
- typu handler, 704

komponenty

- bazy Azure Cosmos DB, 158
- drzewa wyrażeń, 281
- interfejsu graficznego .NET MAUI, 702
- platformy Blazor, 615
- witryny internetowej, 612

komunikacja

- między aplikacją mobilną a usługą internetową, 744
- w czasie rzeczywistym, 499, 502

komunikat, 254

- koncentrator, hub, 502
- strumieniowy, 518

konfiguracja

- mapowania, 244
- niezabezpieczonych połączeń, 749, 750

kontrakty, 471

konwencja

- nazewnictwa pól, 439
- nazewnictwa projektów, 32
- tworzenia modelu jednostek, 115

korekta kodu, 56

kryptografia, 345

kultury, 305, 595

identyfikacja, 306

tymczasowe stosowanie, 312

zmiana, 306

L

liczby

- losowe, 344
- pseudolosowe, 345
- liniowość, linearizability, 157
- lokalizator, 591
- lokalizowanie, 305, 313
 - widoków Razor, 591
- witryny, 588

Ł

ładowanie zasobów, 314

M

magazyn danych, 160
manifest, 268
mapowanie

- hierarchii dziedziczenia, 127
- obiektów, 243
- między modelami, 248
- testowanie konfiguracji, 244

ORM, 113
parametrów, 359
tras, 359
menedżer pakietów NPM, 502
menu kontekstowe, 662, 663
metadane

- o wywołaniu usługi, 488
- typów, 268
- zapytania gRPC, 489
- zestawu, 269

metody

- gRPC, 472
- HTML Helpers, 584, 596
- Tag Helpers, 595, 596

mikrousługi, 36, 470
miniatury czarno-białe, 236
Minimal API, 744, 764

- mapowanie parametrów, 359
- mapowanie tras, 359
- zwracane wartości, 360

model jednostek, 114, 115, 118, 243

- danych, 139
- EF Core, 404

moduł obsługi, handler, 704
MongoDB API, 154
monitorowanie

- wydajności kodu, 196, 201
- wydajności przetwarzania ciągów znaków, 199
- zajętości pamięci, 196, 201

multipleksacja, 472
mutacje, 464
MVC, Model-View-Controller, 567
MVP, Minimal Viable Product, 804
MVU, Model-View-Update, 41
MVVM, Model-View-ViewModel, 41, 728

N

nagłówek Accept-Language, 595
nanosekunda, 294
nanousługi, 36, 525
narzędzie

- Data Migration, 161
- dotnet-ef, 114

NoSQL, 153

- baza Cosmos DB, 154

numerowanie wersji zestawu, 269

O

obiekty

- BLOB, 552, 558
- DTO, 243
- przenoszące dane, 243

obrazy, 236, 669
ochrona

- aplikacji, 325
- funkcjonalności aplikacji, 352
- przed atakami DoS, 383

OData, Open Data Protocol, 37, 38, 401

- funkcje, 415
- klient usługi, 422
- obsługa
 - protokołu, 402
 - zapytania, 427
- opcje zapytań, 413
- operacje CRUD, 419
- operatory, 414
- testowanie, 406, 407, 410
- udostępnianie danych, 400
- wersjonowanie kontrolerów, 417
- wydajność zapytań, 416
- wywoływanie usługi, 424
- zapytania, 401
- zapytania do usługi, 415

odnośniki, 596

- do komponentów Blazor, 621

okno dialogowe, 662, 665

- tworzenie komponentu, 629

OpenAPI

- wykluczanie tras z dokumentacji, 371

operacje

- asynchroniczne, 109, 208
- atomiczne, 221
- CRUD, 159, 176, 419
- na datach i czasie, 290
- na kulturach, 305

operacje
 na strefach czasowych, 298
 na strumieniach asynchronicznych, 224
 synchroniczne, 205
 ORM, Object/Relational Mapping, 113
 osadzone zasoby, 268
 ostrzeżenia, 34
 ukrywanie, 55
 oświadczenie, claim, 346

P

pakiet
 Android SDK, 705
 AspNetCoreRateLimit, 384
 Benchmark.NET, 201
 Green Donut, 433
 Grpc.AspNetCore, 473, 476
 Grpc.Net.Client, 473
 Grpc.Net.ClientFactory, 473
 Hot Chocolate, 433, 441
 Microsoft.Data.SqlClient, 100
 MVVM Toolkit, 769
 NuGet, 392
 SDK, 67
 SignalR, 500
 Strawberry Shake, 433, 460
 pamięć
 lokalna, 644, 645
 sesji, 644
 partycjonowanie, 160
 pasek postępu
 tworzenie komponentu, 627
 platforma
 .NET, 58
 .NET MAUI, 41, 693
 ASP.NET Core, 35
 Blazor, 36, 612
 Blazor WebAssembly, 33
 ChilliCream GraphQL, 433
 gRPC, 471
 pliki
 .cshtml, 567
 .pdf
 generowanie, 260, 263
 .proto, 471
 .resx, 589
 pobieranie danych
 z usługi internetowej, 640
 POCO, Plain Old CLR Object, 434
 podpisywanie, 326, 340

połączenie
 lokalne z usługą, 749
 niezabezpieczone
 aplikacji .NET MAUI, 767
 konfiguracja, 749, 750
 z usługą, 748
 z bazą danych
 Azure SQL, 90
 Azure SQL Edge, 93
 SQL Server, 80, 100
 pomoc
 blog .NET, 75
 dokumentacja Microsoft, 73
 kanał Scotta Hanselmana, 75
 narzędzie dotnet, 73
 zaawansowane opcje wyszukiwarki
 Google, 75
 powiadomienia, 662, 665
 powiązanie, binding, 526
 procedury składowane, 110
 proces, 193
 projekt narzędzia ankietowego, 802
 funkcjonalności, 805
 rodzaje pytań, 806
 sondaże i quizy, 806
 wymagania, 807
 wymagania dodatkowe, 809
 protokół
 OData, 400
 WebSocket, 499
 przekształcanie obrazów, 240
 przepływność bazy danych, 159
 przestrzeń nazw, 67
 Microsoft.Maui, 696
 Microsoft.Maui.Controls, 696, 704
 Microsoft.Maui.Graphics, 696
 System.Diagnostics, 196
 przetwarzanie obrazów, 236
 PWA, Progressive Web Application, 651

R

Radzen Blazor
 aktywacja komponentów, 662
 komponenty biblioteki, 659
 Razor, 36
 definiowanie interfejsu użytkownika, 575
 lokalizowanie widoków, 591
 silnie typowany widok, 585
 składnia, 584
 RDBMS, Relational Database Management
 System, 78, 153

refleksje, 268, 280
rejestrowanie
 działań, 240
 w cyklicznym pliku, 241
 w konsoli, 241
 zapytań HTTP, 372
relacyjna baza danych, 79
 SQL Server, 78
 tabele i relacje, 79
repozytorium GitHub, 72
responsywność aplikacji
 graficznej, 225
 konsolowej, 223
REST Client, 367, 410
 klient usługi GraphQL, 449
 wysyłanie zapytań do usługi, 412
rozszerzenia środowiska Visual Studio
 Code, 49
rozszerzenie
 REST Client, 367, 410
 SQL Server, 84
RPC, Remote Procedure Call, 37

S

scaffolding, 114
SignalR, 37, 38, 498
 definiowanie sygnatur metod, 501
 klient .NET, 515
 komunikacja w czasie rzeczywistym, 502
 koncentrator, 502
 strumieniowanie danych, 518
 tworzenie klienta internetowego, 507
skalowalność
 aplikacji, 230
 usług internetowych, 230
skracanie danych, 336, 337
skrót, 326
skrypt JavaScript, 373
SLA, Service Level Agreement, 157
sól, salt, 328
spójność danych, 157
sprawdzanie poprawności danych, 253
SQL Server, 78
 biblioteka klas, 140, 141
 Developer Edition, 82
 instalacja, 82
 interfejsy API, 100
 konfiguracja, 82
 Management Studio, SSMS, 84
 nawiązywanie połączeń, 80, 100

 wersje oprogramowania, 82
 zarządzanie bazą, 100
 zwalnianie zasobów, 149
SSMS, SQL Server Management Studio, 84
 tworzenie bazy danych, 85
 zapytania T-SQL, 98
sterownik ADO.NET, 101, 102, 107–110
strategia
 TPC, 130
 TPH, 128
 TPT, 129
strategie mapowania hierarchii
 konfigurowanie, 131
 zastosowanie, 132
Strawberry Shake, 433
 tworzenie aplikacji klienckich .NET, 460
strefy czasowe, 298
strona czatu, 508
strukturalne dane o zdarzeniach, 240
strumienie asynchroniczne, 224
strumieniowanie danych, 518
StyleCop, 52
sygnatury metod, 501
symbol @, 575
synchroniczne wykonywanie operacji, 205
synchroniczny odczyt danych, 230
synchronizacja
 dostępu, 215, 222
 zdarzeń, 220
szyfrowanie, 326, 329
 symetryczne, 330

Ś

środowisko
 Banana Cake Pop, 433, 437
 Docker, 91
 GitHub Codespaces, 43
 Visual Studio 2022, 43
 Visual Studio Code, 43

T

Tag Helpers, 596
 bufor, 601
 definiowanie interfejsu użytkownika, 595
 formularz, 606
 odnośniki, 596
 omijanie buforowania, 606
technologie tworzenia aplikacji i usług, 802

- test jednostkowy
 - kodowanie asercji, 250
 - modelu jednostek, 146
 - w Visual Studio Code, 148
- testowanie
 - aplikacji, 705
 - .NET MAUI, 741
 - konsolowej .NET, 517
 - czatu, 512
 - funkcji
 - Azure, 543
 - operującej na kolejce, 558
 - wywoływanej przez czasomierz, 548
 - globalizacji, 317
 - integracji bibliotek klas, 146
 - klienta
 - .NET, 459
 - gRPC, 481
 - komponentu
 - alarmu, 632
 - okna dialogowego, 629
 - paska postępu, 627
 - strony Pracownicy, 688
 - konfiguracji mapowania, 244
 - kontrolerów OData, 407
 - lokalizacji, 317
 - modelu OData, 406
 - transkodowania gRPC JSON, 494
 - usługi, 367
 - gRPC, 481
 - OData, 410
 - strumieniowej i klienta, 522
 - weryfikatora, 256
- token JWT, 395
- transakcje ACID, 188
- transkodowanie gRPC JSON, 492
- tryb programisty w systemie Windows, 707
- T-SQL, Transact-SQL, 94
 - typy danych, 94
- tworzenie
 - aplikacji, 35
 - .NET, 377
 - chmurowych, 44
 - hybrydowych, 756
 - dla różnych systemów
 - operacyjnych, 41
 - dla systemu Windows, 39
 - generującej plik PDF, 263
 - klienckich .NET, 460
 - konsolowych, 102
 - konsolowych .NET, 519
 - mobilnych, 699, 705
 - PWA, 651
 - stacjonarnych, 705
 - uniwersalnych, 44, 46, 707
 - bazy danych, 85
 - biblioteki klas, 140, 141
 - czarno-białych miniatur, 236
 - klienta
 - .NET, 448, 515
 - gRPC, 477
 - internetowego, 507
 - usługi OData, 422
 - komponentów
 - Blazor, 622
 - witryny internetowej, 612, 636
 - komponentu
 - alarmu, 632
 - danych, 635
 - okna dialogowego, 629
 - paska postępu, 627
 - koncentratora
 - SignalR, 503
 - strumieniowego, 518
 - kontrolerów OData, 407
 - mikrouslug, 470
 - modelu
 - jednostek, 114–118, 140
 - widoku, 731
 - współdzielonego, 139
 - nanouslug, 525
 - niestandardowych atrybutów, 272
 - plików zasobów, 589
 - powłoki i stron, 759
 - projektu funkcji Azure, 536
 - serwera bazy danych SQL, 87
 - skryptu JavaScript, 373
 - testów jednostkowych, 146
 - usług internetowych, 36, 357, 402
 - usługi
 - ASP.NET Core Web API, 361
 - dla GraphQL, 434
 - gRPC, 474
 - komunikacji, 502
 - opartej na Minimal API, 744, 764
 - wykorzystującej pamięć lokalną, 645
 - widoku
 - listy, 735
 - szczegółów klientów, 735
 - wirtualnego urządzenia, 705
 - witryny, *Patrz* ASP.NET Core MVC
 - zapytań GraphQL, 438
 - zasobów bazy Cosmos DB, 161

typ danych

- CultureInfo, 306
- DateTime, 298, 300
- IStringLocalizer<T>, 314
- IStringLocalizerFactory, 314
- Process, 197
- Recorder, 197
- RegionInfo, 306
- Stopwatch, 197
- TimeZoneInfo, 299, 300
- wyliczeniowy DayOfWeek, 296

typy danych

- obsługujące wielozadaniowość, 231
- ocena efektywności, 195
- przestarzałe, 274
- standardowe, 274
- w ADO.NET, 101
- w języku T-SQL, 94

U

- udostępnianie danych, 400
- Universal Windows Platform, UWP, 39
- Uno, 42
- URL, Uniform Resource Locator, 35
- usługa, 36
 - Azure SQL Database, 86
 - GraphQL, 38, 448
 - gRPC, 33, 38, 474
 - HTTP API, 33
 - OData, 38, 410
 - SignalR, 38, 500
 - Web API, 38
- usługi internetowe
 - aplikacja z limitem zapytań, 387
 - dokumentowanie, 360
 - interfejs
 - API, 361
 - Minimal API, 744
 - kliencka aplikacja .NET, 377
 - kliencki skrypt JavaScript, 373
 - komunikacja z aplikacją mobilną, 744
 - kontroli tożsamości, 394
 - magazynowe, 534
 - obsługa
 - języka GraphQL, 434
 - protokołu OData, 402
 - odczytywanie danych, 668, 751, 773
 - ograniczanie liczby zapytań
 - komponent ograniczający, 392
 - pakiet AspNetCoreRateLimit, 384

- oparte na Minimal API, 764
- pobieranie danych, 638
- rejestrwanie zapytań HTTP, 372
- skalalność, 230
- testowanie, 367
- tworzenie, 36, 357
 - wykorzystujące pamięć lokalną, 645
- usuwanie zasobów Azure, 190, 565
- UTC, Universal Time Coordinated, 290
- uwierzytelnianie, 326, 346, 347, 353
 - implementacja, 349
 - użycie tokenu JWT, 395

V

- Visual Studio 2022
 - baza danych Azure SQL Edge, 93
 - instalacja, 46
 - projekt funkcji Azure, 536
 - publikowanie projektu, 560
 - testy jednostkowe, 148
 - tworzenie aplikacji, 45
 - tworzenie plików zasobów, 589
- Visual Studio Code
 - edytor zasobów, 590
 - instalacja, 48
 - projekt funkcji Azure, 537
 - publikowanie projektu, 564
 - rozszerzenia, 49
 - REST Client, 367, 410
 - SQL Server, 84
 - testowanie
 - usług internetowych, 367
 - usługi OData, 410
 - testy jednostkowe, 148
 - tworzenie aplikacji uniwersalnych, 44
 - wersje środowiska, 50
 - zapytania T-SQL, 98

W

- warstwa
 - danych, 803
 - prezentacji, 803
 - transportu i formatu wymiany danych, 803
 - usług, 803
- wątek, 193
- WCF, Windows Communication Foundation, 37
- Web API, 38

- WebSocket, 499
- wektory inicjujące, 327
- wersjonowanie kontrolerów OData, 417
- weryfikator, 254
 - definiowanie, 254
 - testowanie, 256
 - modelu, 254
- wiązanie danych, 727
- widok, view, 575
- widoki Razor, 575
- wielozadaniowość, 231
 - z wyłączeniem, 193
- wiersz poleceń
 - projekt funkcji Azure, 539
- Windows
 - App SDK, 39
 - Forms, 39
 - Presentation Foundation, WPF, 39, 694
 - Store, 39
 - Workflow, 40
- współdzielenie
 - CORS, 380
 - dla określonych tras, 382
 - opcje, 383
 - zasobów, 371, 720
- wstrzykiwanie zależności
 - w funkcjach Azure, 535
- wydajność
 - kodu, 196, 201
 - przetwarzania ciągów znaków, 199
 - zapytań OData, 416
- wyjątek PlatformNotSupportedException, 614
- wykres, 676
- wyliczane właściwości, 144
- wyrażenie NCRONTAB, 527
- wysyłanie zapytań, 124, 368, 412, 443
 - T-SQL, 98
- wywołanie gRPC, 489
- wywoływanie
 - metod, 275
 - usług, 424
- wyzwalacz, trigger, 526
- wzorzec projektowy
 - MVC, 33, 567
 - MVVM, 728, 768

X

- XAML, eXtensible Application Markup Language, 40, 693
 - konwertery typów danych, 697
 - rozszerzenia języka, 698
- XMLHttpRequest, 499

Z

- zadania, 193
 - kontynuacyjne, 210
 - oczekujące na wykonanie, 209
 - opakowanie, 213
 - podrzędne, 212
 - sekwencyjne wykonanie, 210
 - uruchamianie, 208
 - zagnieżdżone, 212
- zajętość pamięci, 196, 201
- zakleszczenie blokad, 218
- zakładka, 669
- zapytania
 - ADO.NET, 107
 - DDL, 99
 - do komponentów Blazor, 616
 - do usługi OData, 412, 415, 427
 - do usługi Web API, 368
 - GraphQL, 431, 438, 441–445, 448
 - HTTP, 372
 - OData, 401
 - SQL, 124, 185
 - T-SQL, 98
- zarządzanie bazą danych
 - NoSQL, 153
 - SQL Server, 113
 - relacyjną bazą danych, 78
- zasada tego samego pochodzenia, 371
- zdalne wywoływanie procedur, 37, 471
- zdarzenia
 - synchronizowanie, 220
- zestaw, assembly, 268
 - satelicki, 313, 322
- zestawy
 - ładowanie dynamiczne, 275
 - numerowanie wersji, 269
 - odczytanie metadanych, 269
- zintegrowane środowisko
 - programistyczne, IDE, 43
- zlew, sink, 241
- zwalnianie zasobów baz danych, 149

PROGRAM PARTNERSKI

— GRUPY HELION —

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

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

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

GRUPA
Helion

Aplikacje w .NET, wydajne, skalowalne, bezpieczne — poznaj najnowsze rozwiązania!

C# i wieloplatformowy framework .NET sprawiają, że praca programisty jest efektywna i satysfakcjonująca. Podobnie jak w wypadku innych zaawansowanych technologii, nauka obsługi .NET może przysporzyć sporo trudności. Wielu deweloperów odkrywa, że oficjalna dokumentacja nie wystarczy do nabrania wprawy w tworzeniu złożonych projektów.

To książka przeznaczona dla programistów zaznajomionych z podstawami języka C# i platformy .NET, chcących zdobyć umiejętność tworzenia rzeczywistych aplikacji i usług. Opisuje wyspecjalizowane biblioteki, które umożliwiają monitorowanie i zwiększanie wydajności aplikacji, zabezpieczanie ich wraz z danymi, a także internacjonalizowanie ich kodu. Zawiera również omówienie najnowszych rozwiązań, bibliotek i technologii w połączeniu z ich praktycznym zastosowaniem — między innymi Web API, OData, gRPC, GraphQL, SignalR i Azure Functions. Nie zabrakło prezentacji technik pracy z .NET MAUI, programem, który służy do tworzenia aplikacji mobilnych dla systemów iOS i Android, a także stacjonarnych dla systemów Windows i macOS.

Najciekawsze zagadnienia:

- wydajność, bezpieczeństwo i skalowalność aplikacji i usług
- specjalistyczne biblioteki .NET i biblioteki zewnętrzne, takie jak Serilog i FluentValidation
- tworzenie wielosystemowych aplikacji i ich integracja z natywnymi funkcjami mobilnych systemów operacyjnych
- stosowanie różnych technologii, między innymi bibliotek komponentów Blazor
- praca z danymi w bazach SQL Server i Cosmos DB

Mark J. Price specjalizuje się w programowaniu w języku C#. Pracuje w Microsoftzie, tworzy rozwiązania dla MS Azure. Zdął ponad 80 egzaminów Microsoftu. Zajmuje się też dydaktyką — specjalizuje się w przygotowywaniu kandydatów do egzaminów certyfikacyjnych.

	KOD KORZYŚCI Sięgnij po więcej! ▶	
 helion.pl	ISBN 978-83-8322-716-0	
 HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl	 9 788383 227160	
Cena: 169,00 zł		