

Ovais Mehboob Ahmed Khan

C# 7 i .NET Core 2.0

Programowanie
wielowątkowych
i współbieżnych
aplikacji

Tytuł oryginału: C# 7 and .NET Core 2.0 High Performance:
Build multi threaded and concurrent applications using C# 7 and .Net Core 2.0

Tłumaczenie: Łukasz Świder

ISBN: 978-83-283-5044-1

Copyright © Packt Publishing 2018. First published in the English language
under the title 'C# 7 and .NET Core 2.0 High Performance – (9781788470049)'

Polish edition copyright © 2019 by Helion SA
All rights reserved.

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 Helion SA 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 Helion SA nie ponoszą również
żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/c7nc2p>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Spis treści

O autorze	7
O recenzencie	8
Wstęp	9
Rozdział 1. Co nowego w .NET Core 2 i C# 7?	13
Rozwój frameworka .NET	13
Nowości w .NET Core 2.0	15
Poprawki wydajności	15
Uproszczony system pakietów	17
Ścieżka aktualizacji z .NET Core 1.x do 2.0	17
1. Instalacja .NET Core 2.0	17
2. Zaktualizowanie TargetFramework	17
3. Aktualizacja wersji .NET Core SDK	18
4. Aktualizacja .NET Core CLI	18
Zmiany w ASP.NET Core Identity	18
Odkrywanie .NET Core CLI i szablonów nowych projektów	18
.NET Standard	22
Wersjonowanie .NET Standard	24
Nowości w .NET Standard 2.0	25
Tworzenie biblioteki .NET Standard	27
Co nowego w ASP.NET Core 2.0?	27
ASP.NET Core Razor Pages	27
Uproszczona konfiguracja Application Insights	28
Pule połączeń w Entity Framework Core 2.0	29
Nowe funkcje w C# 7.0	29
Krotki	30
Wzorce	31
Zwracanie referencji	32
Rozszerzone wyrażenia typu expression bodied member	32

Tworzenie lokalnych funkcji	33
Zmienne wyjściowe	33
Asynchroniczna metoda Main	34
Pisanie kodu wysokiej jakości	35
Podsumowanie	38
Rozdział 2. Mechanizmy wewnętrzne .NET Core i mierzenie wydajności	39
Mechanizmy wewnętrzne .NET Core	40
CoreFX	40
CoreCLR	40
Działanie MSIL, CLI, CTS i CLS	41
Jak działa CLR?	42
Od kompilacji do wykonania — pod maską	42
Mechanizm odzyskiwania pamięci (ang. garbage collection)	43
.NET Native i kompilacja JIT	46
Wykorzystywanie wielu rdzeni CPU dla większej wydajności	46
Jak kompilacje w trybie wydania zwiększają wydajność	48
Testy porównawcze aplikacji .NET Core 2.0	49
Poznanie BenchmarkDotNet	49
Jak to działa	51
Ustawianie parametrów	51
Diagnostyka pamięci z użyciem BenchmarkDotNet	53
Dodawanie konfiguracji	53
Podsumowanie	55
Rozdział 3. Wielowątkowość i programowanie asynchroniczne w .NET Core	57
Wielowątkowość kontra programowanie asynchroniczne	58
Wielowątkowość w .NET Core	60
Zastrzeżenia w wielowątkowości	60
Wątki w .NET Core	61
Synchronizacja wątków	64
Task parallel library (TPL)	70
Wzorce projektowe programowania równoległego	77
Podsumowanie	83
Rozdział 4. Struktury danych i pisanie zoptymalizowanego kodu C#	85
Czym są struktury danych?	86
Notacja wielkiego O do mierzenia wydajności i złożoności algorytmu	88
Logarytmy	90
Wybieranie odpowiedniej struktury danych do optymalizacji wydajności	91
Tablice	91
Listy	92
Stosy	93
Kolejka	94
Listy łączone	95
Słowniki, tablice haszujące i zbiory haszujące	96
Listy generyczne	96

Najlepsze praktyki pisania zoptymalizowanego kodu C#	97
Narzut pakowania i rozpakowywania	98
Konkatenacja łańcuchów znaków	100
Obsługa wyjątków	101
For i foreach	102
Delegaty	103
Podsumowanie	104
Rozdział 5. Wytyczne projektowania wydajnych aplikacji .NET Core	105
Zasady kodowania	106
Konwencje nazewnicze	106
Komentarze	107
Jedna klasa na plik	107
Jedna logika na metodę	107
Zasady projektowania	108
KISS (Keep It Simple, Stupid)	108
YAGNI (You Aren't Gonna Need It)	109
DRY (Don't Repeat Yourself)	109
Podział odpowiedzialności	109
Zasady SOLID	110
Buforowanie	121
Struktury danych	122
Komunikacja	122
Zarządzanie zasobami	123
Współbieżność	124
Podsumowanie	125
Rozdział 6. Techniki zarządzania pamięcią w .NET Core	127
Przegląd zarządzania alokacją pamięci	128
Analizowanie mechanizmów wewnętrznych CLR przez debugger SOS w .NET Core	128
Fragmentacja pamięci	132
Unikanie destruktorów	133
Najlepsze praktyki zwalniania obiektów w .NET Core	135
Wstęp do interfejsu IDisposable	135
Czym są niezarządzane zasoby?	135
Wykorzystywanie IDisposable	136
Kiedy implementować interfejs IDisposable?	137
Destruktor i Dispose	138
Podsumowanie	140
Rozdział 7. Stosowanie zabezpieczeń i implementowanie odporności na błędy w aplikacjach .NET Core	141
Wprowadzenie do aplikacji odpornych na błędy	142
Polityki odporności	142
Przechowywanie danych wrażliwych z wykorzystaniem Application Secrets	158
Zabezpieczanie API w ASP.NET Core	161
SSL (ang. Secure Socket Layer)	161
Zapobieganie atakom CSRF (ang. Cross-Site Request Forgery)	163

Wzmacnianie nagłówków bezpieczeństwa	163
Uwierzytelnianie i autoryzacja	168
Uwierzytelnianie	169
Autoryzacja	169
Implementacja uwierzytelniania i autoryzacji z użyciem frameworka ASP.NET Core Identity	169
Podsumowanie	173
Rozdział 8. Architektura mikrousług	175
Architektura mikrousług	176
Zalety architektury mikrousług	177
Standardowe praktyki podczas tworzenia mikrousług	178
Typy mikrousług	179
DDD	179
Manipulowanie danymi w mikrousługach	179
Spójność w różnych scenariuszach biznesowych	180
Komunikacja z mikrousługami	181
Architektura baz danych w mikrousługach	182
Czym jest kompozycja API?	183
CQRS	184
Tworzenie architektury mikrousług w .NET Core	185
Tworzenie przykładowej aplikacji w .NET Core z wykorzystaniem architektury mikrousług	185
Wdrażanie mikrousług w kontenerach Docker	210
Czym jest Docker?	211
Korzystanie z Dockera w .NET Core	212
Uruchamianie obrazów Dockera	214
Podsumowanie	214
Rozdział 9. Monitorowanie wydajności aplikacji z wykorzystaniem narzędzi	215
Kluczowe wskaźniki wydajności aplikacji	216
Średni czas odpowiedzi	216
Apdex	216
Odsetek błędów	216
Liczba żądań	216
Przepustowość/punkty końcowe	217
Wykorzystanie procesora i pamięci	217
Narzędzia i techniki monitorowania wydajności	217
Wstęp do App Metrics	217
Konfigurowanie App Metrics w ASP.NET Core	217
Śledzące oprogramowanie pośredniczące	218
Dodawanie raportów graficznych	220
Podsumowanie	229
Skorowidz	231

Mechanizmy wewnętrzne .NET Core i mierzenie wydajności

Podczas projektowania architektury aplikacji znajomość wewnętrznych mechanizmów platformy .NET jest kluczowa dla zagwarantowania jakości przyszłej aplikacji. W tym rozdziale skupimy się na mechanizmach wewnętrznych .NET Core, które mogą pomóc nam w pisaniu kodu wysokiej jakości i tworzeniu wysokiej jakości architektury dla dowolnej aplikacji. Ten rozdział poruszy główne pojęcia związane z mechanizmami wewnętrznymi .NET Core, takimi jak: proces kompilacji, mechanizm odzyskiwania pamięci i **biblioteka klas platformy** (ang. *Framework Class Library*, FCL). Zakończymy rozdział przeglądem narzędzia BenchmarkDotNet, które jest najczęściej używane do mierzenia wydajności aplikacji i jest zalecane do testowania fragmentów kodu w aplikacji.

W tym rozdziale poruszymy następujące zagadnienia:

- Mechanizmy wewnętrzne .NET Core
- Wykorzystywanie wielu rdzeni procesora w celu osiągnięcia wysokiej wydajności
- Jak budowanie w trybie wydania zwiększa wydajność
- Analiza porównawcza wydajności aplikacji .NET Core 2.0

Mechanizmy wewnętrzne .NET Core

.Net Core składa się z dwóch głównych komponentów — środowiska wykonawczego CoreCLR i bibliotek bazowych CoreFX. W tym podrozdziale poruszymy następujące tematy:

- CoreFX
- CoreCLR
- Działanie MSIL, CLI, CTS i CLS
- Jak działa CLR?
- Od kompilacji po wykonanie — pod maską
- Mechanizm odzyskiwania pamięci
- .NET Native i kompilacja JIT

CoreFX

CoreFX jest nazwą kodową bibliotek składających się na .NET Core. Zawiera wszystkie biblioteki zaczynające się słowami Microsoft.* i System.* oraz obejmuje kolekcje, mechanizmy I/O (wejścia-wyjścia), manipulowanie łańcuchami znaków, refleksję, zabezpieczenia i wiele innych.

CoreFX jest niezależne od środowiska wykonawczego i może być uruchamiane na każdej platformie bez względu na to, jakie interfejsy API ona wspiera.

Aby poznać więcej szczegółów, możesz odnieść się do kodu źródłowego .NET Core pod adresem <https://source.dot.net>.

CoreCLR

CoreCLR udostępnia wspólne środowisko wykonawcze dla aplikacji .NET Core i zarządza pełnym cyklem życia każdej aplikacji. Wykonuje wiele operacji podczas działania programu. Operacje takie jak alokowanie pamięci, mechanizm odzyskiwania pamięci, obsługa wyjątków, bezpieczeństwo typów, zarządzanie wątkami oraz bezpieczeństwo są częścią CoreCLR.

Środowisko uruchomieniowe .NET Core zawiera taki sam **mechanizm odzyskiwania pamięci** (ang. *Garbage Collection*, GC) jak .NET Framework oraz nowy, bardziej zoptymalizowany kompilator **Just In Time** (JIT), zwany *RyuJIT*. Podczas pierwszego wydania .NET Core obsługiwał tylko aplikacje w architekturze 64-bitowej, lecz wraz z wydaniem .NET Core 2.0 obsługuje także 32-bitowe architektury. Jednak wersja 32-bitowa obsługuje tylko system Windows.

Działanie MSIL, CLI, CTS i CLS

Podczas budowania projektu kod jest kompilowany do formy języka przejściowego (ang. *Intermediate Language*, IL), znanego także jako język przejściowy Microsoft (ang. *Microsoft Intermediate Language*, MSIL). MSIL jest zgodny ze wspólną infrastrukturą języka (ang. *Common Language Infrastructure*, CLI), gdzie CLI jest standardem definiującym wspólny system typów (*Common Type System*, CTS) oraz specyfikację języka (*Common Language Specification*, CLS).

CTS zawiera wspólny system typów i kompiluje typy specyficzne dla języka na typy zgodne z platformą. Standaryzuje wszystkie typy języków platformy .NET na typy wspólne dla zapewnienia współoperacyjności języka. Przykładowo, jeśli kod jest napisany w C#, zostanie przekonwertowany do właściwego CTS.

Przypuśćmy, że mamy dwie zmienne zdefiniowane w poniższym kodzie C#:

```
class Program
{
    static void Main(string[] args)
    {
        int minNo = 1;
        long maxThroughput = 99999;
    }
}
```

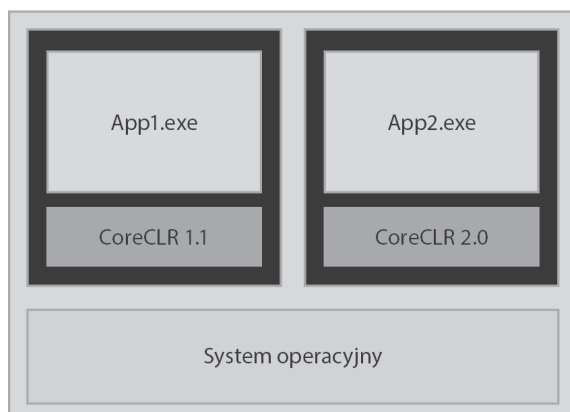
Podczas kompilacji kompilator wygeneruje do podzespołu (ang. *Assembly*) MSIL, które będzie dostępne dla CoreCLR, aby wykonać JIT i przekonwertować go na natywny kod maszynowy. Należy zauważyć, że typy `int` i `long` są konwertowane na, odpowiednio, `int32` i `int64`:

```
.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    // Code size      11 (0xb)
    .maxstack 1
    .locals init (int32 V_0,
                 int64 V_1)
    IL_0000: nop
    IL_0001: ldc.i4.1
    IL_0002: stloc.0
    IL_0003: ldc.i4      0x1869f
    IL_0008: conv.i8
    IL_0009: stloc.1
    IL_000a: ret
} // end of method Program::Main
```

Nie jest wymagane, aby każdy język był w pełni zgodny z CTS, może obsługiwać tylko mały wycinek CTS. Na przykład, gdy VB.NET został wydany po raz pierwszy w .NET Framework, obsługiwał tylko typy całkowitoliczbowe ze znakiem i nie było możliwe użycie typów całkowitoliczbowych bez znaku. Z kolejnymi wersjami .NET Framework i teraz z .NET Core 2.0 możemy do tworzenia aplikacji używać wszystkich zarządzanych języków, takich jak C#, F# i VB.NET, i łatwo odwoływać się do różnych podzespołów.

Jak działa CLR?

CLR jest zaimplementowane jako zestaw bibliotek wewnętrzprocesowych, które są ładowane wraz z aplikacją i działają w ramach kontekstu procesu aplikacji. Na poniższym rysunku widzimy dwie działające aplikacje, które zostały nazwane *App1.exe* i *App2.exe*. Każdy z czarnych kwadratów reprezentuje przestrzeń adresową aplikacji, wewnątrz których *App1.exe* i *App2.exe* mają uruchomione ich własne wersje CLR obok siebie:



Podczas pakowania aplikacji .NET Core możemy je opublikować albo jako **zależne od frameworka** (ang. *Framework Dependent Deployment*, FDD), albo jako **niezależne** (ang. *Self-Contained Deployments*, SCD). W FDD opublikowany pakiet nie posiada środowiska uruchomieniowego .NET Core i oczekuje, że .NET Core jest zainstalowane w docelowym systemie. Przy SCD wszystkie komponenty, takie jak środowisko uruchomieniowe .NET Core oraz biblioteki .NET Core, są dołączone do opublikowanego pakietu. Dzięki temu nie jest wymagane, aby docelowy system operacyjny posiadał zainstalowane .NET Core.

Więcej informacji o FDD i SCD znajdziesz na stronie pod adresem <https://docs.microsoft.com/en-us/dotnet/core/deploying/>.

Od kompilacji do wykonania — pod maską

Proces kompilacji .NET Core jest podobny do tego w .NET Framework. Podczas konstruowania projektu system MSBuild, który buduje projekt i generuje bibliotekę klas (*.dll*) lub plik wykonywalny (*.exe*), wywołuje wewnętrzne polecenie .NET CLI. Podzespół zawiera manifest posiadający metadane podzespołu, takie jak numer wersji, język, informacja o referencjach typów, informacja o referencjach do innych podzespółów oraz listę innych plików w podzespole wraz z ich lokalizacją. Ten manifest jest przechowywany w kodzie MSIL lub w samodzielnym **przenośnym pliku wykonywalnym** (ang. *Portable executable*, PE):



Gdy uruchamiany jest plik wykonywalny, tworzy się nowy proces i ładuje środowisko uruchomieniowe .NET Core, które potem inicjalizuje środowisko wykonawcze, konfiguruje sterter oraz pulę wątków i ładuje podzespół do przestrzeni adresowej procesu. Na podstawie kodu programu wywoływana jest metoda punktu startowego (metoda Main) i wykonywana jest kompilacja JIT. Od tego momentu kod jest wykonywany i obiekty zaczynają być alokowane na stercie, podczas gdy typy proste przechowywane są na stosie. Dla każdej metody wykonywana jest kompilacja JIT i generowany jest natywny kod maszynowy.

Po wykonaniu kompilacji JIT, a przed generowaniem natywnego kodu maszynowego wykonywanych jest kilka walidacji. Do tych walidacji zalicza się:

- weryfikowanie, czy MSIL został wygenerowany podczas procesu budowania,
- weryfikowanie, czy kod nie został zmodyfikowany lub czy podczas procesu kompilacji JIT nie zostały dodane nowe typy,
- weryfikowanie, czy został wygenerowany zoptymalizowany kod dla docelowej maszyny.

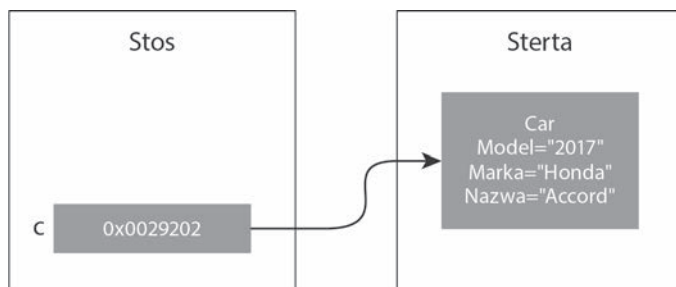
Mechanizm odzyskiwania pamięci (ang. garbage collection)

Jedną z najważniejszych cech CLR jest mechanizm odzyskiwania pamięci. Skoro aplikacje .NET Core są aplikacjami zarządzanymi, większość procesu odzyskiwania pamięci wykonywana jest automatycznie przez CLR. Alokacja obiektów w pamięci jest efektywnie wykonywana przez CLR. CLR nie tylko od czasu do czasu dostosowuje zasoby pamięci wirtualnej, lecz także zmniejsza fragmentację podległej pamięci wirtualnej, aby była bardziej wydajna pod względem dostępnej przestrzeni.

Gdy aplikacja jest wykonywana, obiekty zaczynają być alokowane na stercie, a adres każdego z obiektów przechowywany jest na stosie. Ten proces jest kontynuowany, dopóki pamięć nie osiągnie swojego maksymalnego limitu. Wtedy do gry wchodzi GC i zaczyna odzyskiwać pamięć poprzez usuwanie nieużywanych zarządzanych obiektów i alokowanie nowych obiektów. Wszystko to GC wykonuje automatycznie, lecz można też je wywołać, aby wykonało odzyskiwanie pamięci, poprzez wywołanie metody `GC.Collect`.

Na potrzeby przykładu założmy, że w metodzie Main mamy obiekt `c` typu `Car`. Podczas wykonywania metody obiekt `Car` będzie przez CLR zaalokowany w pamięci sterty, a referencja do obiektu `c` wskazująca na adres obiektu `Car` na stercie zostanie odłożona na stos.

Podczas działania mechanizmu odzyskiwania pamięci odzyskana zostanie pamięć ze sterty i referencja do obiektu ze stosu zostanie usunięta:



Warty odnotowania jest fakt, że odzyskiwanie pamięci wykonywane jest automatycznie przez GC tylko dla obiektów zarządzanych i jeśli mamy jakieś obiekty niezarządzane, jak połączenie z bazą danych, operacje wejścia-wyjścia itp., muszą być one zwolnione ręcznie. Inaczej mówiąc, GC działa efektywnie dla obiektów zarządzanych i zapewnia, że aplikacja nie odczuje spadku wydajności podczas wykonywania GC.

Generacje w GC

W odzyskiwaniu pamięci wyróżniamy trzy typy generacji, znane jako **Generacja 0**, **Generacja 1** i **Generacja 2**. W tym podrozdziale przyjrzymy się idei generacji i temu, jak wpływają one na wydajność odzyskiwania pamięci.

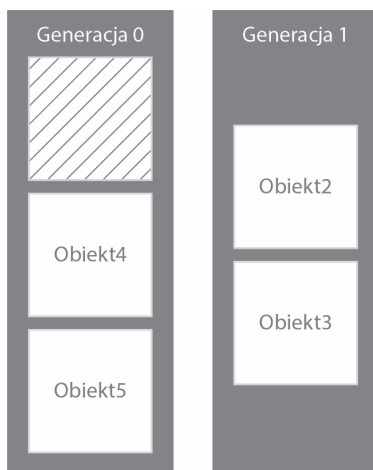
Przypuśćmy, że uruchamiamy aplikację tworzącą trzy obiekty nazwane Obiekt1, Obiekt2 i Obiekt3. Obiekty te będą miały zaalokowaną pamięć w **Generacji 0**:



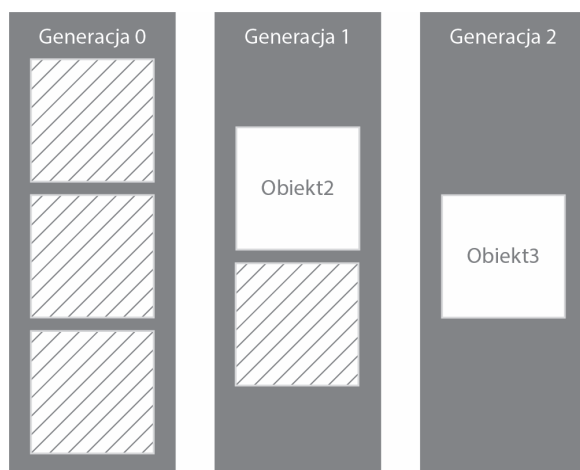
Gdy odzyskiwanie pamięci zostanie uruchomione (jest to proces automatyczny, chyba że samodzielnie wywołasz go z kodu), sprawdza, czy są obiekty, które nie są już potrzebne w aplikacji i nie ma do nich referencji w programie. Takie obiekty zostaną po prostu usunięte.

Przykładowo, jeżeli nigdzie nie ma odwołania do zakresu obiektu *Obiekt1*, pamięć zajmowana przez ten obiekt zostanie odzyskana. Jednakże dwa inne obiekty *Obiekt2* i *Obiekt3* nadal mają odwołania w kodzie programu i zostaną przeniesione do **Generacji 1**.

Teraz założmy, że są utworzone dwa kolejne obiekty, zwane *Obiekt4* i *Obiekt5*. Zostaną zakwalifikowane do **Generacji 0**, co pokazuje poniższy rysunek:



Gdy proces odzyskiwania pamięci zostanie uruchomiony po raz drugi, znajdzie dwa obiekty, zwane *Obiekt4* i *Obiekt5*, w **Generacji 0** i dwa obiekty, nazywające się *Obiekt2* i *Obiekt3*, w **Generacji 1**. GC sprawdzi najpierw referencje do obiektów w **Generacji 0** i jeśli nie będą używane przez aplikację, zostaną usunięte. Tak samo zostaną sprawdzone obiekty **Generacji 1**. Przykładowo, jeśli *Obiekt3* nadal jest używany, zostanie przesunięty do **Generacji 2**, a *Obiekt2* zostanie usunięty z **Generacji 1**, co pokazano na poniższym rysunku.



Koncept generacji optymalizuje pracę GC — jest wysoce prawdopodobne, że obiekty przechowywane w **Generacji 2** będą przechowywane przez dłuższy czas. GC wykonuje mniej operacji i dzięki temu zyskuje czas, zamiast za każdym razem sprawdzać obiekt po obiekcie. To samo tyczy się **Generacji 1**, dla której także odzyskanie pamięci jest mniej prawdopodobne niż w przypadku **Generacji 0**.

.NET Native i kompilacja JIT

Kompilacja JIT odbywa się w trakcie wykonania i konwertuje kod MSIL do natywnego kodu maszynowego. Odbywa się to podczas pierwszego uruchomienia kodu, które trwa trochę dłużej niż następne. W .NET Core tworzymy teraz aplikacje na platformy mobilne i urządzenia przenośne, które mają limitowane zasoby procesora oraz pamięci. W przypadku platform *Universal Windows Platform (UWP)* i *Xamarin* .NET Core generuje natywny kod w czasie kompilacji lub podczas generowania pakietów specyficznych dla platform. Nie jest przez to wymagana podczas działania kompilacja JIT, co zwiększa szybkość uruchamiania aplikacji. Ta natywna kompilacja wykonywana jest przez komponent zwany .NET Native.

.NET Native rozpoczyna proces kompilacji zaraz po kompilacji specyficznej dla języka podczas budowania aplikacji. Mechanizmy .NET Native odczytują kod MSIL generowany przez kompilator języka i przeprowadzają szereg operacji, m.in.:

- eliminują metadane z MSIL,
- podmieniają kod bazujący na refleksji i metadanych na natywny statyczny kod podczas porównywania wartości pól,
- sprawdzają kod, który ma być wykonany przez aplikację (i tylko on zostaje w wynikowym podzespole),
- zamieniają pełne CLR na zrefaktoryzowane środowisko uruchomieniowe z mechanizmem odzyskiwania pamięci i bez kompilatora JIT. Zrefaktoryzowane środowisko uruchomieniowe zostanie dostarczone wraz z aplikacją i jest zawarte w bibliotece o nazwie *mrt100_app.dll*.

Wykorzystywanie wielu rdzeni CPU dla większej wydajności

W obecnych czasach aplikacje bazują bardziej na łączności i w niektórych przypadkach operacje wykonywane przez aplikację trwają dłużej. Wiemy także, że obecnie wszystkie komputery posiadają wielordzeniowe procesory i efektywne wykorzystanie tych rdzeni zwiększa wydajność aplikacji. Operacje takie jak sieć bądź IO mają opóźnienia i synchroniczne wykonanie kodu aplikacji może prowadzić do długich czasów oczekiwania. Jeśli czasochłonne zadania są wykonywane w oddzielnym wątku lub w sposób asynchroniczny, wykonywana operacja będzie trwać krócej i wzrośnie responsywność aplikacji. Kolejną zaletą jest wykonywanie zadań na wielu rdzeniach procesora jednocześnie. W świecie .NET możemy zyskać responsywność i wydajność poprzez podział zadań na wiele wątków, wykorzystując klasyczne API progra-

mowania wielowątkowego lub korzystając z uproszczonego i zaawansowanego modelu znanego jako *task programming library* (TPL). TPL jest obsługiwane przez .NET Core 2.0 i wkrótce przyjrzymy się temu, jak może być użyte do wykonywania zadań na wielu rdzeniach.

Model programowania z wykorzystaniem TPL bazuje na pojęciu zadania. Zadanie jest jednostką pracy — obiektywnym przedstawieniem operacji w trakcie wykonywania.

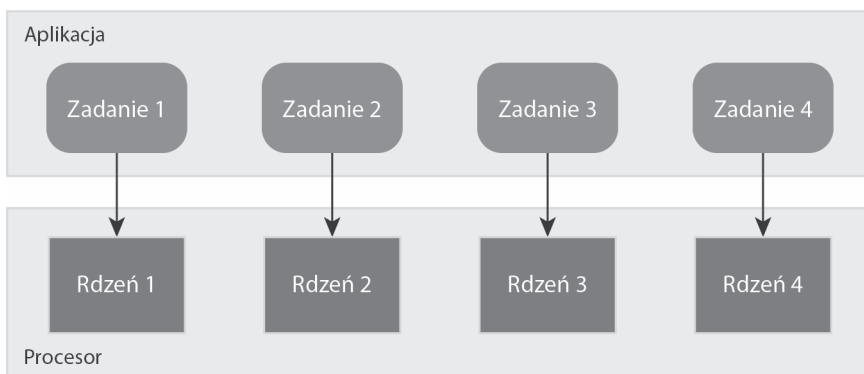
Proste zadanie możemy utworzyć poprzez napisanie np. poniższego kodu:

```
static void Main(string[] args)
{
    Task t = new Task(Execute);
    t.Start();
    t.Wait();
}

private static void Execute()
{
    for (int i = 0; i < 100; i++)
    {
        Console.WriteLine(i);
    }
}
```

W przedstawionym kodzie zadanie może być inicjalizowane z użyciem obiektu typu `Task`, gdzie `Execute` jest metodą obliczeniową, która zostanie wykonana po wywołaniu metody `Start`. Metoda `Start` informuje .NET Core, że zadanie może zostać wykonane, i kończy swoje działanie. Rozdziela wykonanie programu na dwa wątki, które działają współbieżnie. Pierwszy wątek to główny wątek aplikacji, a drugi wykonuje metodę `Execute`. Użyliśmy metody `t.Wait`, aby poczekać na wątek roboczy i wyświetlić wynik na konsoli. W innym przypadku po opuszczeniu przez program bloku kodu metody `Main` aplikacja kończy działanie.

Celem programowania równoległego jest efektywne wykorzystanie wielu rdzeni procesora. Na przykład założymy, że uruchamiamy podany kod na procesorze jednordzeniowym. Dwa wątki będą działać i dzielić ten sam procesor. Jeśli program byłby uruchomiony na procesorze wielordzeniowym, mógłby działać na kilku rdzeniach, wykorzystując każdy rdzeń oddzielnie, zwiększając wydajność i osiągając prawdziwą równoległość:

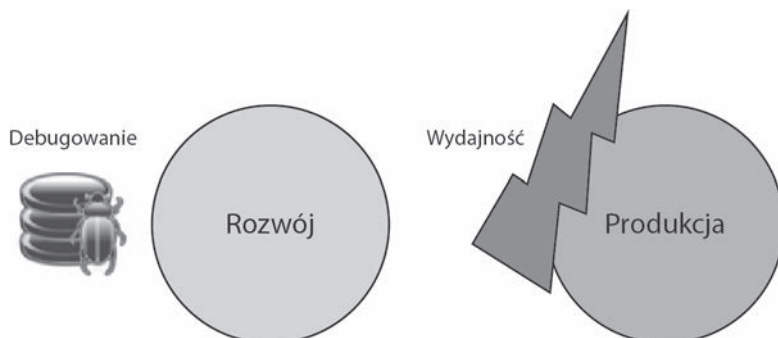


W przeciwieństwie do TPL, klasyczny obiekt typu Thread nie gwarantuje, że wątek będzie działał na różnych rdzeniach procesora. TPL daje pewność, że każdy wątek będzie wykonywany na innym rdzeniu, chyba że osiągnięta zostanie liczba zadań na rdzeń i rdzenie będą współdzielone.

Więcej informacji na temat korzyści stosowania TPL znajdziesz na stronie pod adresem <https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl>.

Jak kompilacje w trybie wydania zwiększają wydajność

Tryby kompilacji dla wydania (ang. *Release*) i debugowania są dwoma głównymi sposobami budowania aplikacji .NET. Tryb debugowania jest używany zazwyczaj podczas pisania kodu lub naprawiania błędów, podczas gdy tryb wydania jest częściej używany podczas pakowania aplikacji do wdrożenia na serwerach produkcyjnych. Podczas tworzenia paczki wdrożeniowej programiści często pomijają zmianę trybu budowania na tryb wydania, przez co po wdrożeniu aplikacji pojawiają się problemy z wydajnością.



Poniższa tabela opisuje różnice pomiędzy trybami debugowania i wydania.

Tryb debugowania (Debug)	Tryb wydania (Release)
Kompilator nie wykonuje optymalizacji kodu	Kod jest optymalizowany i zmniejszany jest jego rozmiar podczas budowania w trybie wydania
Ślad stosu jest łapany i wyrzucany w momencie wystąpienia wyjątku	Ślad stosu nie jest łapany
Symbole debugowania są przechowywane	Symbole debugowania i cały kod w ramach dyrektywy #debug są usuwane
Kod źródłowy zużywa więcej pamięci podczas wykonania	Kod źródłowy zużywa mniej pamięci podczas wykonania

Testy porównawcze aplikacji .NET Core 2.0

Testowanie porównawcze aplikacji jest procesem oceny i porównywania artefaktów z uzgodnionymi standardami. Aby wykonać testy porównawcze kodu aplikacji .NET Core 2.0, możemy użyć narzędzia BenchmarkDotNet, które zawiera bardzo proste API do oceny wydajności kodu aplikacji. Zazwyczaj analiza porównawcza w skali mikro, jak w przypadku klas i metod, nie jest prosta i zmierzenie wydajności wymaga wiele wysiłku, podczas gdy BenchmarkDotNet może wykonać całe to niskopoziomowe zadanie, jak i złożone zadania związane z analizą porównawczą.

Poznawanie BenchmarkDotNet

W tym podrozdziale poznamy BenchmarkDotNet i nauczymy się, jak efektywnie możemy użyć tego narzędzia do mierzenia wydajności aplikacji.

Możemy go zainstalować z użyciem konsoli menedżera pakietów NuGet lub poprzez sekcję *References* projektu. Aby zainstalować BenchmarkDotNet, należy wykonać polecenie:

```
Install-Package BenchmarkDotNet
```

Powyższe polecenie dodaje pakiet BenchmarkDotNet z *NuGet.org*.

Aby przetestować narzędzie BenchmarkDotNet, utworzymy prostą klasę, która zawiera dwie metody do generowania sekwencji 10 liczb ciągu Fibonacciego. Ciąg Fibonacciego może być zaimplementowany na kilka sposobów, dlatego użyjemy go do zmierzenia, który fragment kodu jest szybszy i bardziej wydajny.

Oto pierwsza metoda generująca ciąg Fibonacciego iteracyjnie:

```
public class TestBenchmark
{
    int len = 10;

    [Benchmark]
    public void Fibonacci()
    {
        int a = 0, b = 1, c = 0;
        Console.WriteLine("{0} {1}", a, b);
        for (int i = 2; i < len; i++)
        {
            c = a + b;
            Console.WriteLine(" {0}", c);
            a = b;
            b = c;
        }
    }
}
```

Poniżej druga metoda, która wykorzystuje podejście rekurencyjne do generowania ciągu Fibonacciego:

```
[Benchmark]
public void FibonacciRecursive()
{
    int len = 10;
    Fibonacci_Recursive(0, 1, 1, len);
}

private void Fibonacci_Recursive(int a, int b, int counter, int len)
{
    if (counter <= len)
    {
        Console.WriteLine("{0} ", a);
        Fibonacci_Recursive(b, a + b, counter + 1, len);
    }
}
```

Zauważ, że obie metody generujące liczby ciągu Fibonacciego zawierają atrybut `Benchmark`. To właśnie mówi obiektowi `BenchmarkRunner`, które metody mierzyć. Możemy też wywołać `BenchmarkRunner` z głównego punktu wejściowego aplikacji, który to zmierzy wydajność aplikacji i wygeneruje raport, co pokazano w poniższym fragmencie kodu:

```
static void Main(string[] args)
{
    BenchmarkRunner.Run<TestBenchmark>();
    Console.Read();
}
```

Po uruchomieniu analizy otrzymamy raport podobny do tego:

```
C:\Program Files\dotnet\dotnet.exe
// * Summary *
BenchmarkDotNet=v0.11.1, OS=Windows 10.0.17134.228 (1803/April2018Update/Redstone4)
Intel Core i7-5700HQ CPU 2.70GHz (Broadwell), 1 CPU, 8 logical and 4 physical cores
Frequency=2630639 Hz, Resolution=380.1358 ns, Timer=TSC
.NET Core SDK=2.1.400
[Host] : .NET Core 2.1.2 (CoreCLR 4.6.26628.05, CoreFX 4.6.26629.01), 64bit RyuJIT [AttachedDebugger]
DefaultJob : .NET Core 2.1.2 (CoreCLR 4.6.26628.05, CoreFX 4.6.26629.01), 64bit RyuJIT

          Method |      Mean |      Error |      StdDev |
-----|-----|-----|-----|
          Fibonacci | 19.84 us | 0.3938 us | 0.7587 us |
    FibonacciRecursive | 21.87 us | 0.4363 us | 0.7641 us |

// * Warnings *
Environment
  Summary -> Benchmark was executed with attached debugger

// * Legends *
  Mean : Arithmetic mean of all measurements
  Error : Half of 99.9% confidence interval
  StdDev : Standard deviation of all measurements
  1 us : 1 Microsecond (0.000001 sec)

// ***** BenchmarkRunner: End *****
Run time: 00:01:08 (68.88 sec), executed benchmarks: 2

// * Artifacts cleanup *
```

Zostaną również wygenerowane pliki w głównym folderze aplikacji uruchamiającej BenchmarkRunner: plik .html zawierający informacje o wersji BenchmarkDotNet, wersji systemu, procesorze, częstotliwości, rozkładzie, szczegółach licznika, wersji .NET (w naszym przypadku .NET Core SDK 2.1.4), hoście i tak dalej:

```
BenchmarkDotNet=v0.11.1, OS=Windows 10.0.17134.228 (1803/April2018Update/Redstone4)
Intel Core i7-5700HQ CPU 2.70GHz (Broadwell), 1 CPU, 8 logical and 4 physical cores
Frequency=2630639 Hz, Resolution=380.1358 ns, Timer=TSC
.NET Core SDK=2.1.400
[Host]      : .NET Core 2.1.2 (CoreCLR 4.6.26628.05, CoreFX 4.6.26629.01), 64bit RyuJIT [AttachedDebugger]
DefaultJob : .NET Core 2.1.2 (CoreCLR 4.6.26628.05, CoreFX 4.6.26629.01), 64bit RyuJIT
```

Method	Mean	Error	StdDev
Fibonacci	19.84 us	0.3938 us	0.7587 us
FibonacciRecursive	21.87 us	0.4363 us	0.7641 us

Tabela zawiera cztery kolumny. Możemy jednak dodać więcej kolumn, które domyślnie są opcjonalne. Możemy także dodawać niestandardowe kolumny. **Method** to nazwa metody zawierającej atrybut Benchmark, **Mean** jest średnim czasem, który zajmuje wykonanie wszystkich pomiarów (us oznacza milisekundy), **Error** jest czasem przeznaczonym na przetwarzanie błędów, **StdDev** jest standardowym odchyleniem pomiarów.

Po porównaniu obu metod można stwierdzić, że metoda FibonacciRecursive jest bardziej wydajna, jako że wartości dla **Mean**, **Error** i **StdDev** są mniejsze niż dla metody iteracyjnej.

Oprócz pliku HTML, tworzone są dwa dodatkowe pliki, plik CSV (ang. *Comma Separated Value*) oraz plik Markdown (MD), zawierające te same informacje.

Jak to działa

BenchmarkDotNet w czasie działania generuje projekt dla każdej analizowanej metody oraz buduje go w trybie wydania. Próbuje kilku kombinacji, aby zmierzyć wydajność metody poprzez wielokrotne jej wywołanie, po czym generuje raport składający się z plików i informacji o BenchmarkDotNet.

Ustawianie parametrów

W poprzednim przykładzie testowaliśmy metodę z tylko jedną wartością. W praktyce podczas testowania aplikacji klasy enterprise testujemy ją z różnymi wartościami, aby oszacować wydajność metody.

Po pierwsze, możemy zdefiniować właściwość dla każdego parametru poprzez dodanie atrybutu Params i podanie wartości, dla jakich chcielibyśmy, aby metoda była testowana. Później możemy użyć tej właściwości w kodzie. BenchmarkRun automatycznie testuje metodę ze wszystkimi parametrami i generuje raport. Poniżej kompletny kod klasy TestBenchmark:

```

public class TestBenchmark
{
    [Params(10,20,30)]
    public int Len { get; set; }

    [Benchmark]
    public void Fibonacci()
    {
        int a = 0, b = 1, c = 0;
        Console.WriteLine("{0} {1}", a, b);

        for (int i = 2; i < Len; i++)
        {
            c = a + b;
            Console.WriteLine(" {0}", c);
            a = b;
            b = c;
        }
    }

    [Benchmark]
    public void FibonacciRecursive()
    {
        Fibonacci_Recursive(0, 1, 1, Len);
    }

    private void Fibonacci_Recursive(int a, int b, int counter, int len)
    {
        if (counter <= len)
        {
            Console.WriteLine("{0} ", a);
            Fibonacci_Recursive(b, a + b, counter + 1, len);
        }
    }
}

```

Po uruchomieniu analizy zostanie wygenerowany raport podobny do poniższego:

```

BenchmarkDotNet=v0.11.1, OS=Windows 10.0.17134.345 (1803/April2018Update/Redstone4)
Intel Core i7-5700HQ CPU 2.70GHz (Broadwell), 1 CPU, 8 logical and 4 physical cores
Frequency=2630638 Hz, Resolution=380.1359 ns, Timer=TSC
.NET Core SDK=2.1.400
[Host] : .NET Core 2.1.2 (CoreCLR 4.6.26628.05, CoreFX 4.6.26629.01), 64bit RyuJIT [AttachedDebugger]
DefaultJob : .NET Core 2.1.2 (CoreCLR 4.6.26628.05, CoreFX 4.6.26629.01), 64bit RyuJIT

```

Method	Len	Mean	Error	StdDev
Fibonacci	10	19.44 us	0.3799 us	0.5915 us
FibonacciRecursive	10	21.96 us	0.4362 us	0.7166 us
Fibonacci	20	44.87 us	0.8831 us	1.6147 us
FibonacciRecursive	20	46.27 us	0.9088 us	1.6617 us
Fibonacci	30	73.30 us	1.4386 us	2.9710 us
FibonacciRecursive	30	72.49 us	1.4392 us	3.3356 us

Diagnostyka pamięci z użyciem BenchmarkDotNet

Z BenchmarkDotNet możemy także diagnozować dowolne problemy z pamięcią, procesem odzyskiwania pamięci oraz mierzyć liczbę alokowanych bajtów.

Możemy to zaimplementować, używając atrybutu `MemoryDiagnoser` na poziomie klasy. Dodajmy atrybut `MemoryDiagnoser` do klasy `TestBenchmark` utworzonej w poprzednim przykładzie:

```
[MemoryDiagnoser]
public class TestBenchmark {}
```

Przy ponownym uruchomieniu aplikacji zostaną zebrane dodatkowe dane związane z alokacją pamięci i procesem odzyskiwania pamięci, a następnie odpowiednio umieszczone w raporcie:

```
BenchmarkDotNet=v0.11.1, OS=Windows 10.0.17134.345 (1803/April2018Update/Redstone4)
Intel Core i7-5700HQ CPU 2.70GHz (Broadwell), 1 CPU, 8 logical and 4 physical cores
Frequency=2630638 Hz, Resolution=380.1359 ns, Timer=TSC
.NET Core SDK=2.1.400
[Host] : .NET Core 2.1.2 (CoreCLR 4.6.26628.05, CoreFX 4.6.26629.01), 64bit RyuJIT [AttachedDebugger]
DefaultJob : .NET Core 2.1.2 (CoreCLR 4.6.26628.05, CoreFX 4.6.26629.01), 64bit RyuJIT
```

Method	Len	Mean	Error	StdDev	Gen 0	Allocated
Fibonacci	10	22.69 us	0.4177 us	1.0557 us	0.1221	528 B
FibonacciRecursive	10	24.05 us	0.4300 us	0.9617 us	0.1526	560 B
Fibonacci	20	46.53 us	0.7754 us	1.6856 us	0.3052	1152 B
FibonacciRecursive	20	49.91 us	1.3673 us	3.9009 us	0.3662	1184 B
Fibonacci	30	80.66 us	1.3024 us	3.3619 us	0.4883	1792 B
FibonacciRecursive	30	76.86 us	1.3586 us	3.5791 us	0.4883	1824 B

W powyższej tabeli kolumna **Gen 0** oznacza liczbę generacji na 1000 operacji. Jeśli wartością jest 1, oznacza to, że proces odzyskiwania pamięci miał miejsce po 1000 operacji. Zauważ jednak, że w pierwszym wierszu jest wartość 0.1221, która oznacza, że odzyskiwanie pamięci zostało wykonane po 0,1221 sekundy, podczas gdy dla tego wiersza nie zostało przeprowadzone odzyskiwanie pamięci dla **Generacji 1** (w takim wypadku pojawiłaby się kolumna **Gen 1**). **Allocated** oznacza rozmiar pamięci zaalokowany podczas wywołania metody. Nie zawiera natywnych wywołań *Stackalloc/heap*.

Dodawanie konfiguracji

Konfiguracja analizy porównawczej może być zdefiniowana poprzez utworzenie niestandardowej klasy i dziedziczenie z klasy `ManualConfig`. Oto przykład klasy `TestBenchmark`, którą utworzyliśmy wcześniej, zawierającej metody testowe:

```
[Config(typeof(Config))]
public class TestBenchmark
{
    private class Config : ManualConfig
    {
```

```

// Będziemy analizować TYLKO metody mające w nazwie Recursive
public Config()
{
    Add(new DisjunctionFilter(
        new NameFilter(name => name.Contains("Recursive"))
    ));
}

[Params(10, 20, 30)]
public int Len { get; set; }

[Benchmark]
public void Fibonacci()
{
    int a = 0, b = 1, c = 0;
    Console.WriteLine("{0} {1}", a, b);

    for (int i = 2; i < Len; i++)
    {
        c = a + b;
        Console.WriteLine(" {0}", c);
        a = b;
        b = c;
    }
}

[Benchmark]
public void FibonacciRecursive()
{
    Fibonacci_Recursive(0, 1, 1, Len);
}

private void Fibonacci_Recursive(int a, int b, int counter, int len)
{
    if (counter <= len)
    {
        Console.WriteLine("{0} ", a);
        Fibonacci_Recursive(b, a + b, counter + 1, len);
    }
}
}

```

W powyższym kodzie zdefiniowaliśmy klasę `Config`, która dziedziczy z klasy `ManualConfig` dostarczanej wraz z biblioteką. Reguły mogą być zdefiniowane wewnątrz konstruktora klasy `Config`. W przykładzie występuje reguła określająca, że należy analizować metody zawierające w nazwie słowo *Recursive*. W naszym przypadku mamy tylko taką metodę, `FibonacciRecursive`, która zostanie wywołana i której wydajność będzie zmierzona.

Innym sposobem dodania konfiguracji jest użycie API fluent, w którym możemy pominąć tworzenie klasy Config i zaimplementować to w taki sposób:

```
static void Main(string[] args)
{
    var config = ManualConfig.Create(DefaultConfig.Instance);
    config.Add(new DisjunctionFilter(new NameFilter(
        name => name.Contains("Recursive"))));
    BenchmarkRunner.Run<TestBenchmark>(config);
}
```

Więcej informacji o BenchmarkDotNet znajdziesz pod adresem <http://benchmarkdotnet.org/Configs.htm>.

Podsumowanie

W niniejszym rozdziale poznaliśmy główne założenia .NET Core, w tym procesy kompilacji i odzyskiwania pamięci, dowiedzieliśmy się, jak tworzyć wysoko wydajne aplikacje .NET Core poprzez użycie wielu rdzeni procesora oraz jak publikować aplikacje w trybie wydania. Przyjrzelśmy się również narzędziu do analizy porównawczej, które jest najczęściej używane do optymalizacji kodu i które dostarcza wyników specyficznych dla klas.

W następnym rozdziale dowiemy się więcej o programowaniu wielowątkowym i współbieżnym w .NET Core.

Skorowidz

- .NET Core, 22
- .NET Core 2, 13
- .NET Core CLI, 18
 - instalacja, 19
 - polecenia, 20
- .NET Core SDK, 18
- .NET Framework, 22
- .NET Native, 46
- .NET Standard, 22
 - API, 25
 - tryb kompatybilności, 26
 - tworzenie biblioteki, 27
 - wersjonowanie, 24

A

- ACID, Atomicity, Consistency, Integrity, Durability, 181
- ACS, Azure Container Service, 175
- aktualizacja, 17
- alokacja pamięci, 128
- analiza porównawcza, 53
- Apdex, 216, 218
- API, 161
- aplikacje
 - ASP.NET Core MVC, 19
 - odporne na błędy, 142
 - responsywne, 57
 - SPAs, 194
- APM, Asynchronous Programming Model, 76
- App Metrics, 217
 - konfigurowanie, 217
 - monitorowanie wydajności, 218
 - śledzące oprogramowanie pośredniczące, 219

- Application
 - Insights, 28
 - Secrets, 158
- architektura
 - baz danych, 182
 - logiczna, 187
 - mikrouslug, 175, 185
 - zalety, 177
- ASP.NET Core, 161
 - Identity, 18, 169
 - Razor Pages, 27
- ASP.NET Core 2.0, 27
 - Application Insights, 28
 - automatyczna kompilacja widoku, 28
 - Entity Framework Core 2.0, 29
 - składnia Razor, 27
- asynchroniczna metoda Main, 34
- asynchroniczne operacje, 58
- atak CSRF, 163
- atomowość, 181
- automatyczna kompilacja widoku, 28
- autoryzacja użytkowników, 168, 192

B

- baza danych
 - InfluxDB, 220
 - per usługa, 183
- BCL, Base Class Libraries, 14
- BenchmarkDotNet, 49
 - diagnostyka pamięci, 53
 - mierzenie wydajności, 49
- bezpieczeństwo
 - wzmacnianie nagłówków, 163

biblioteka
 .NET Standard, 27
 PCL, 24
 biblioteki klas frameworka, FCL, 14
 błędy, 142
 buforowanie, 121

C

C#
 najlepsze praktyki, 97
 # 7.0, 13
 expression bodied members, 32
 krotki, 30
 metoda Main, 34
 nowe funkcje, 29
 tworzenie funkcji, 33
 wzorce, 31
 zmienne wyjściowe, 33
 zwracanie referencji, 32
 CAS, Central Authentication Server, 192
 centralny serwer autoryzacji, CAS, 192
 CLI, Command Line Interface, 18
 CLI, Common Language Infrastructure, 41
 CLR, Common Language Runtime, 14, 124, 128
 analiza mechanizmów wewnętrznych, 128
 CLS, Common Language Specification, 41
 CoreCLR, 40
 CoreFX, 40
 CORS, Cross-Origin Resource Sharing, 167, 184
 CQRS, Command Query Responsibility
 Segregation, 180
 CRUD, 179
 CSRF, Cross-Site Request Forgery, 163
 CSV, Comma Separated Value, 51
 CTS, Common Type System, 41
 cykl życia wątku, 63

D

dane wrażliwe, 158
 DDD, Domain Driven Design, 178
 debugger SOS, 128
 debugowanie, 129
 delegaty, 103
 denormalizacja danych, 180
 destruktor, 133, 138
 DI, Dependency Injection, 146, 191
 diagnostyka pamięci, 53
 Docker, 211
 uruchamianie obrazów, 214
 wdrażanie mikrousług, 210

doświadczenie użytkownika, 35
 DRY, 109
 działanie
 BenchmarkDotNet, 51
 CLR, 42
 dziedziczenie, 112

E

EAP, Event-based Asynchronous Pattern, 76
 Entity Framework Core 2.0, 29

F

FCL, Framework Class Libraries, 14
 FIFO, First In First Out, 94
 fragmentacja pamięci, 132
 framework .NET, 13
 funkcje lokalne, 33

G

GC, Garbage Collection, 40
 generacje, 44
 Grafana
 instalowanie narzędzia, 223
 pulpit nawigacyjny, 223, 225
 raportowanie, 227
 testowanie aplikacji, 227

I

IDE, Integrated Development Environment, 35
 IDisposable, 136
 implementowanie, 137
 implementacja
 buforowania, 155
 interfejsu IDisposable, 137
 limitu czasu, 154
 sprawdzania stanu zdrowia, 157
 TAP, 75
 TLS, 165
 usługi dostawców, 202
 uwierzytelniania i autoryzacji, 169
 Wyłącznika obwodu, 145
 wzorca Ponawianie, 143
 InfluxDB
 instalowanie, 222
 konfigurowanie, 220, 224
 pulpit nawigacyjny, 223

instalowanie
 .NET Core CLI, 19
 Grafana, 223
 InfluxDB, 222
 integralność, 181
 interfejs
 IDisposable, 135
 konsolowy, CLI, 18
 repozytorium, 190
 iterowanie, 102

K

KISS, 108
 klasa, 107
 BaseEntity, 189
 Startup, 29, 225
 TestBenchmark, 51
 kolejka, 87, 94
 FIFO, 94
 kolejkiwanie komunikacji, 123
 komentarze, 107
 kompilacja, 42
 w trybie wydania, 48
 kompilator
 JIT, 46
 RyuJIT, 16
 kompozycja, 113
 API, 180, 183
 komunikacja, 122
 z mikrousługami, 181
 konfiguracja
 analizy porównawczej, 53
 Application Insights, 28
 App Metrics, 217
 InfluxDB, 220, 224
 podsystemu Windows, 221
 konkatencja łańcuchów znaków, 100
 kontener Docker, 210
 konwencje nazewnictwa, 106
 krotki, 30

L

LINQ, 82
 listy, 92
 cykliczne, 95
 dwukierunkowe, 95
 generyczne, 96
 jednokierunkowe, 95
 łączone, 87, 95

logarytmy, 90
 logika, 107
 logowanie, 190

Ł

łańcuchy znaków, 100

M

mapowanie obiektowo-relacyjnego, ORM, 188
 maszyna wirtualna, VM, 176
 mechanizm odzyskiwania pamięci, GC, 40, 43
 mechanizmy wewnętrzne, 40
 metapakiety, 17
 metoda
 AddDbContextPool, 29
 Configure, 225
 ConfigureServices, 225
 Dispose, 138
 Finalize, 138
 Main, 34
 metody
 asynchroniczne, 72
 pakowania, 98
 synchroniczne, 72
 mierzenie wydajności, 39, 49, 88
 mikrousługi, 175
 architektura baz danych, 182
 bezstanowe, 179
 manipulowanie danymi, 179
 opakowywanie, 180
 stanowe, 179
 tworzenie, 178, 185
 wdrażanie, 210
 minimalizowanie rozmiaru wiadomości, 122
 model programowania asynchronicznego,
 APM, 76
 modele wdrażania, 15
 monitorowanie wydajności aplikacji, 217, 215
 monitory, 65
 MSIL, Microsoft Intermediate Language, 41
 MVC, Model View Controller, 14

N

nagłówek
 Content-Security-Policy, 166
 referrer-policy, 166
 strict transport security, 165

nagłówek
 X-Content-Type-Options, 165
 X-Frame-Options, 165
 X-Xss-Protection, 166
 najlepsze praktyki, 97
 zwalnianie obiektów, 135
 narzędzia
 do debugowania, 129
 do monitorowania wydajności, 217
 narzędzie
 .NET Core CLI, 18
 App Metrics, 217
 Grafana, 223
 narzut pakowania, 98
 niezarządzane zasoby, 135
 notacja wielkiego O, 88

O

obiekt
 DbContext, 29
 Task, 73
 obsługa wyjątków, 101
 odzyskiwanie pamięci, 43
 opakowywanie mikrousług, 180
 oprogramowanie pośredniczące śledzenia, 218
 optymalizacja wspierana profilem, PGO, 16
 ORM, Object-Relational Mapping, 188

P

pakiet, 17
 NuGet, 20
 pamięć
 fragmentacja, 132
 zarządzanie alokacją, 128
 Parallel LINQ, 82
 parametry, 111
 PCL, Portable Class Libraries, 23
 pętla
 for, 102
 foreach, 82, 102
 PGO, profile-guided optimization, 16
 platformy PCL, 24
 plik Markdown, 51
 pliki .cs, 107
 POCO, Plain Old CLR object, 122, 188
 podstawowe biblioteki klas, BCL, 14
 podział
 baz danych, 183
 odpowiedzialności, 109
 tabel, 183

polecenia
 .NET Core CLI, 20
 Entity Framework Core, 21
 serwerowe, 21
 polityka Fallback, 153
 polityki
 odporności, 142
 proaktywne, 154
 reaktywne, 142
 ponawianie, 143, 148, 153
 programowanie
 asynchroniczne, 58
 równoległe, 47
 wzorce projektowe, 77
 projekt
 infrastruktury, 189
 APIComponents, 191
 usługi tożsamości, 194
 projektowanie, 108
 przenośny plik wykonywalny, 42
 przepływy połączenia OpenIddict, 193
 pula wątków, 63

R

raportowanie, 227
 raporty graficzne, 220
 referencje, 32
 reguły, 37
 rozszerzenie SOS, 128
 RPM, Request Per Minute, 227
 RyuJIT, 16

S

słowniki, 96
 SOLID, 110
 SOS, 128
 spójność, 180, 181
 SSL, Secure Socket Layer, 161
 włączanie, 161
 sterta, 132
 stos, 87, 93
 struktura danych, 85, 122
 kolejka, 94
 lista, 92
 generyczna, 96
 łączona, 95
 słownik, 96
 stos, 93

- tablica, 91
- tablica haszująca, 96
- zbiór haszujący, 96
- synchronizacja wątków, 64
- system pakietów, 17
- szablon projektów, 18

Ś

- ścieżka aktualizacji, 17
- śledzące oprogramowanie pośredniczące, 218, 219

T

- tabela
 - per usługa, 182
 - użytkownika, 171
- tablica, 87, 91
 - haszująca, 96
- TAP, Task-based asynchronous pattern, 71
- TargetFramework, 17
- testowanie aplikacji, 227
- testy porównawcze, 49
- TLS, Transport Level Security, 165
- TPL, task parallel library, 70
 - tworzenie zadania, 70
- TPL, task programming library, 47
- trwałość, 181
- tryb
 - debugowania, 48
 - wydania, 48
- tworzenie
 - architektury mikrousług, 185
 - biblioteki .NET Standard, 27
 - domeny sprzedawców, 202
 - infrastruktury sprzedawców, 204
 - interfejsu repozytorium, 190
 - klasy, 189
 - lokalnych funkcji, 33
 - mikrousług, 178
 - usługi sprzedawców, 206
 - usługi tożsamości, 192
 - wątków, 61
 - zadania, 70
- typy
 - proste, 86
 - wiadomości, 206

U

- unikanie destruktorów, 133
- URL, Uniform Resource Locator, 194
- usługa
 - e-mail, 143
 - rejestracji użytkowników, 143
- uwierzytelnianie, 168

V

- VM, Virtual Machine, 176

W

- wątki
 - cykl życia, 63
 - synchronizacja, 64
 - w .NET Core, 61
- WDK, Windows Driver Kit, 128
- wdrażanie, 15
 - mikrousług, 210
- wersjonowanie .NET Standard, 24
- węzeł Analyzers, 36
- wielowątkowość, 58
 - w .NET Core, 60
- włączanie CORS, 167
- wskaźnik
 - aktywne żądania, 219
 - Apdex, 216, 218
 - błędy, 219
 - czas odpowiedzi, 219
 - liczba żądań, 216
 - odsetek błędów, 216
 - przepustowość/punkty końcowe, 217
 - średni czas odpowiedzi, 216
 - wydajności, 216
 - wykorzystanie procesora i pamięci, 217
- wspólne środowisko uruchomieniowe, CLR, 14
- współbieżność, 124
- wstrzykiwanie zależności, DI, 29, 146, 191
- wydajność, 15, 39
 - algorytmu, 88
 - BenchmarkDotNet, 49
 - kompilacje w trybie wydania, 48
 - monitorowanie, 215
 - narzędzia, 217
 - programowanie równoległe, 47
 - rdzenie CPU, 46

wydajność

RPM, 227

wskaźniki, 216

wybór struktury danych, 91

wyjątek, 72, 101

Wyłącznik obwodu, 148, 153

wyrażenia typu expression bodied member, 32

wzorce, 31

projektowe programowania równoległego, 77

stałych, 31

tworzenia obiektów, 120

wzorzec

DDD, 178, 179

Fabryka, 120

Jednostka pracy, 189

Mediator, 206

Ponawianie, 143

potoku, 77

producent-konsument, 80

przepływu danych, 78

typu, 31

var, 32

wstrzykiwanie zależności, 120

Wyłącznik obwodu, 145

X

Xamarin, 22

Y

YAGNI, 109

Z

zabezpieczenia, 141

zadania, 70

anulowanie, 73

raportowanie postępu, 74

status, 73

zajmowanie zasobów, 124

zarządzanie

pamięcią, 127, 128

zasobami, 123

zasada

jednej odpowiedzialności, 110

odwrócenia zależności, 120

otwarte-zamknięte, 111

podstawienia Liskov, 115

segregacji interfejsów, 117

zasady projektowania, 106, 108

DRY, 109

KISS, 108

SOLID, 110

YAGNI, 109

zbiory haszujące, 96

zintegrowane środowisko deweloperskie, IDE, 35

złożoność programu, 88

zmiennie wyjściowe, 33

zwalnianie obiektów, 124, 135

zwracanie referencji, 32

Ż

żądania na minutę, RPM, 227

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 

C# i .NET Core: wielowątkowość – współbieżność – wydajność!

W świecie programistów aplikacji panuje kult wydajności: najważniejsze są szybkość i efektywność działania kodu. Dostrajanie elementów dużych aplikacji staje się wirtuozerią; wymaga eliminowania wąskich gardeł, optymalizacji kodu, pilnowania każdego bitu pamięci. Niewielkie braki w rozwiązaniach w przypadku rozbudowanych systemów przeradzają się w wielkie problemy. Dla programisty oznacza to, że jeśli chce pracować na prawdziwie profesjonalnym poziomie, musi perfekcyjnie opanować zagadnienia związane ze skalarnością, z modularnością i efektywnością kodu.

Ta książka jest przeznaczona dla programistów .NET, którzy chcą przyspieszyć pracę swoich aplikacji. Opisuje nowe funkcje C# 7 i .NET Core 2.0 oraz ich wpływ na wydajność kodu. Przedstawia takie mechanizmy .NET Core jak proces kompilacji, odzyskiwanie pamięci czy wykorzystywanie wielu rdzeni procesora. Prezentuje koncepcje wielowątkowości i programowania asynchronicznego oraz wyjaśnia znaczenie optymalizacji struktur danych. Omawia też wzorce i najlepsze praktyki projektowania aplikacji w .NET Core, a także zagadnienia bezpieczeństwa i elastyczności oraz architektury mikrousług. Wiedza zawarta w książce pozwoli na pisanie modularnych, skalowalnych, bezpiecznych i niezależnie wdrażanych aplikacji.

W tej książce między innymi:

- nowości w C# 7 i .NET Core 2.0
- struktury danych i optymalizacja kodu w C#
- zarządzanie pamięcią i zapobieganie wyciekom pamięci
- zapewnianie odporności na błędy aplikacji
- narzędzia do monitorowania wydajności aplikacji: App Metrics, InfluxDB i Grafana
- wytyczne projektowania i dobre praktyki programistyczne

Ovais Mehboob Ahmed Khan jest architektem z 14-letnim doświadczeniem w programowaniu, a także autorem książek i innych publikacji technicznych. Pracował w kilku firmach informatycznych w Pakistanie, USA oraz na Bliskim Wschodzie. Obecnie jest zatrudniony w państwowej firmie w Dubaju. Otrzymał tytuł MVP. Specjalizuje się w takich technologiach jak Microsoft .NET, chmura i tworzenie aplikacji internetowych.

	<i>Sprawdź nasze szkolenia!</i>	KOD KORZYŚCI Sięgnij po więcej! ▶	
 helion.pl	 AKADEMIA IT & BUSINESS	ISBN 978-83-283-5044-1	
 0 801 339900		WWW.SZKOLENIA.HELION.PL	
 0 601 339900	INFORMATYKA W NAJLEPSZYM WYDANIU	Cena: 49,00 zł	

Packt