Informatyka techniczna i sztuczna inteligencja
Technical informatics and Artificial Intelligence
Engineer of XXI Century'2024

Performance comparison of modern backend programming languages

Grzegorz Kastelik [1], Marcel Gańczarczyk [2*], Łukasz Więcław [3], Tomasz Gancarczyk [4,]

[1] *mgr inż., Faculty of Mechanical Engineering and Computer Science, University of Bielsko-Biala, Willowa 2, 43-300 Bielsko-Biala, Poland, gk051414@student.ubb.edu.pl*
[2] *inż., Faculty of Mechanical Engineering and Computer Science, University of Bielsko-Biala, Willowa 2, 43-300 Bielsko-Biala, Poland, s57157@student.ubb.edu.pl*
[3] *dr, Department of Computer Science and Automatics, Faculty of Mechanical Engineering and Computer Science, University of Bielsko-Biala, Willowa 2, 43-300 Bielsko-Biala, Poland, lwieclaw@ubb.edu.pl*
[4] *dr inż., Department of Computer Science and Automatics, Faculty of Mechanical Engineering and Computer Science, University of Bielsko-Biala, Willowa 2, 43-300 Bielsko-Biala, Poland*
* *Corresponding author, s57157@student.ubb.edu.pl*

Abstract: This paper presents a comparative analysis of performance among four widely used backend programming languages: Golang, C#, JavaScript (Node.js), and PHP. To ensure a meaningful comparison, applications were developed in each language and tested using key metrics such as response time, the number of requests processed per second, and the complexity of their implementation. The testing aimed to evaluate how effectively each language handles multiple concurrent requests and performs under various system load conditions. The analysis offers insights into the strengths and weaknesses of each technology, particularly concerning performance and resource management. Additionally, the paper investigates how the choice of programming language can influence the efficiency and scalability of backend applications, depending on the project's specific requirements and available resources.

Keywords: Golang; backend development; performance analysis; C#; JavaScript; PHP; multithreading; web applications

Porównanie wydajności nowoczesnych języków programowania środowisk backend

Grzegorz Kastelik [1], Marcel Gańczarczyk [2*], Łukasz Więcław [3], Tomasz Gancarczyk [4,]

[1] *mgr inż., Wydział Budowy Maszyn i Informatyki, Uniwersytet Bielsko-Bialski, Willowa 2, 43-300 Bielsko-Biala, Polska, gk051414@student.ubb.edu.pl*
[2] *inż., Wydział Budowy Maszyn i Informatyki, Uniwersytet Bielsko-Bialski, Willowa 2, 43-300 Bielsko-Biala, Polska, s57157@student.ubb.edu.pl**
[3] *dr, Katedra Informatyki i Automatyki, Wydział Budowy Maszyn i Informatyki, Uniwersytet Bielsko-Bialski, Willowa 2, 43-300 Bielsko-Biala, Polska, lwieclaw@ubb.edu.pl*
[4] *dr inż., Katedra Informatyki i Automatyki, Wydział Budowy Maszyn i Informatyki, Uniwersytet Bielsko-Bialski, Willowa 2, 43-300 Bielsko-Biala, Polska, tgan@ubb.edu.pl*
* *Corresponding author, s57157@student.ubb.edu.pl*

Streszczenie: W artykule przeprowadzono analizę wydajności czterech popularnych języków programowania stosowanych w środowiskach backendowych: Golang, C#, JavaScript (Node.js) oraz PHP. Dla celów porównawczych stworzono aplikacje w każdym z tych języków i przetestowano je pod kątem kluczowych parametrów, takich jak czas odpowiedzi, liczba obsługiwanych zapytań na sekundę oraz złożoność ich implementacji. Przeprowadzone testy miały na celu ocenę, jak różne technologie radzą sobie z obsługą wielu równoczesnych żądań oraz w warunkach o zróżnicowanym obciążeniu systemu. Analiza wyników dostarcza wglądu w mocne i słabe strony każdego z tych języków, szczególnie w kontekście wydajności i zarządzania zasobami. Ponadto

https://doi.org/10.53052/9788367652292.12

artykuł bada, jak wybór języka programowania może wpływać na efektywność i skalowalność aplikacji backendowych w zależności od specyficznych wymagań projektu oraz dostępnych zasobów.

Słowa kluczowe: Golang; rozwój backendu; analiza wydajności; C#; JavaScript; PHP; wielowątkowość; aplikacje internetowe

1. Wstęp

W dzisiejszym świecie aplikacje internetowe stały się nieodłącznym elementem codziennego życia. Użytkownicy korzystają z aplikacji bankowych, platform społecznościowych czy serwisów zakupowych, oczekując od nich płynności działania i niezawodności. Za każdą z tych usług kryje się skomplikowana część zwana backendem, która odpowiada za przetwarzanie logiki biznesowej oraz zarządzanie danymi. Wydajność tej warstwy, choć niewidoczna dla użytkownika końcowego, ma kluczowe znaczenie dla ogólnego funkcjonowania aplikacji [1].

Tworzenie wydajnych rozwiązań backendowych wymaga zarówno zaawansowanej wiedzy, jak i odpowiednich narzędzi programistycznych. Język programowania, w którym tworzy się rozwiązanie, może znacząco wpłynąć na szybkość działania aplikacji, skalowalność oraz możliwość obsługi dużej liczby równoczesnych zapytań. W ostatnich latach język Golang (określany również mianem Go), opracowany przez firmę Google, zdobywa coraz większą popularność w branży backendowej dzięki swojej efektywności oraz wsparciu dla wielowątkowości [2]. Środowisko to zapoczątkowało rozwój języków programowania w kierunku zwiększania wydajności w zastosowaniach backendowych.

Celem niniejszej pracy jest analiza wydajności Golanga w porównaniu z innymi popularnymi językami backendowymi, takimi jak C#, JavaScript (Node.js) i PHP. Analizie poddano zarówno wydajność, jak i złożoność implementacji, co pozwala na wszechstronne porównanie tych technologii.

1.1. Historia aplikacji internetowych

Historia aplikacji internetowych sięga początku lat 90, kiedy powstawały pierwsze, głównie statyczne, strony tworzone za pomocą HTML. Przełomem w ich rozwoju było wprowadzenie JavaScriptu w 1995 roku, który umożliwił interaktywność stron. W kolejnych latach pojawiły się technologie takie jak Macromedia Flash (1996) do tworzenia gier w przeglądarkach oraz PHP (1997), który umożliwił dynamiczne generowanie stron i komunikację z bazami danych. W 1999 roku zaczęto budować bardziej złożone aplikacje internetowe w języku Java. Wprowadzenie technologii Ajax w 2005 roku znacznie zwiększyło szybkość i jakość aplikacji. Kolejnym przełomem był standard HTML5 (2014-2015), który usprawnił dotychczasowe technologie oraz umożliwił tworzenie aplikacji PWA. Obecnie frontend (czyli warstwa aplikacji działająca w przeglądarce) i backend (warstwa aplikacji działająca na serwerze) są zazwyczaj rozwijane oddzielnie z uwagi na ich rosnącą złożoność [3].

1.2. Bezpieczeństwo

Bezpieczeństwo jest kluczowym aspektem w tworzeniu zaawansowanych rozwiązań backendowych, ponieważ każda aplikacja wymienia dane z klientami, które mogą być poufne i podatne na nadużycia. Główne działania zabezpieczające koncentrują się na ochronie tych danych. Certyfikaty SSL są używane do szyfrowania połączeń, a mechanizm CORS chroni przed nieautoryzowanymi komunikacjami. Istnieje jednak wiele potencjalnych luk bezpieczeństwa, dlatego frameworki w różnych językach programowania pomagają rozwiązać te problemy i przyspieszają pracę. Ważnym elementem jest również uwierzytelnianie użytkowników, które może stanowić punkt podatny na ataki, dlatego jego poprawna implementacja jest kluczowa [4]. Z tego względu w niniejszej pracy zdecydowano się na skorzystanie z bibliotek rozwiązań backednowych dostarczanych dla każdego z języków.

1.3. Wydajność

Wydajność jest kluczowym czynnikiem przy tworzeniu rozwiązań backendowych, szczególnie gdy aplikacja musi obsługiwać tysiące użytkowników dziennie. Często, aby uzyskać lepszą wydajność, poświęca się intuicyjność i czytelność kodu. Na efektywność aplikacji wpływa wiele czynników, takich jak błędy logiczne w kodzie, wydajność maszyny serwerowej oraz specyfika języka programowania, np. obsługa wielowątkowości. Wszystkie te elementy wpływają na czas przetwarzania i odpowiedzi aplikacji na żądania użytkowników [5].

2. Założenia

W celu przeprowadzenia rzetelnego porównania wydajności, wybrano cztery popularne języki programowania backendowego: Golang, C#, JavaScript (Node.js) oraz PHP. Każdy z tych języków został wybrany na podstawie ich szerokiego zastosowania w branży oraz ich specyficznych zalet. Golang wyróżnia się efektywną obsługą wielowątkowości i niskimi wymaganiami zasobowymi [2], C# jest powszechnie stosowany w aplikacjach korporacyjnych [6], JavaScript wraz Node.js umożliwia obsługę zarówno aplikacji serwerowych, jak i przeglądarkowych [7], a PHP wciąż jest szeroko wykorzystywany do obsługi wielu stron internetowych, np. tych opartych na WordPressie [8].

Aplikacje zostały poddane testom w różnych scenariuszach obciążeniowych, takich jak liczba przetwarzanych żądań na sekundę, czas odpowiedzi oraz zarządzanie zasobami systemowymi, w tym wykorzystaniem pamięci RAM i procesora. Każda aplikacja obsługiwała identyczne punkty końcowe odpowiedzialne za operacje na plikach, zapytania do bazy danych oraz przetwarzanie obliczeń.

Testy przeprowadzono w jednolitym środowisku, aby wyeliminować zmienne mogące wpływać na wyniki. Aplikacje były uruchamiane na maszynach o tych samych specyfikacjach sprzętowych, a do izolacji środowisk testowych wykorzystano Dockera. Takie podejście do testowania zapewniło, że wyniki były rzetelne i łatwe do porównania.

Ponadto uwzględniono złożoność kodu, łatwość implementacji oraz utrzymania aplikacji. Każdy z języków wymagał innego nakładu pracy programistycznej do osiągnięcia tych samych funkcjonalności, co miało wpływ na ocenę złożoności implementacji.

2.1. Wybór języków do porównania

Do porównania wybrano cztery języki programowania: Golang, C#, JavaScript (Node.js) oraz PHP. Wybór ten wynika z ich powszechnego zastosowania w branży, szczególnie w rozwiązaniach non-enterprise.

W badaniu nie uwzględniono wszystkich dostępnych języków programowania backendowego, a wybór ograniczono do najpopularniejszych na rynku, z wykluczeniem języka Java, ze względu na jej charakter zastosowania głównie w rozwiązaniach enterprise. Ponadto, ze względu na ograniczenia czasowe i zasobowe, testy były prowadzone na ograniczonej liczbie zapytań i w specyficznym środowisku, co może wpływać na ich skalowalność w rzeczywistych warunkach produkcyjnych. Wyniki należy traktować jako miarodajne w odniesieniu do wybranego środowiska, z możliwością dostosowania w zależności od specyfiki rzeczywistych projektów.

2.2. Kryteria oceny wydajności

Głównymi kryteriami oceny wydajności były: czas odpowiedzi na zapytania (zarówno przy małym, jak i dużym obciążeniu), liczba przetwarzanych zapytań na sekundę oraz efektywność zarządzania zasobami (RAM i CPU). Wydajność mierzono poprzez testy obciążeniowe, w których każda aplikacja musiała obsłużyć serię zapytań GET oraz POST. Każdy język musiał poradzić sobie zarówno z prostymi operacjami na plikach, jak i bardziej złożonymi operacjami na bazach danych oraz obliczeniami.

2.3. Środowisko testowe

Testy wydajności przeprowadzono w jednorodnym środowisku, co pozwoliło na wyeliminowanie zmiennych mogących wpłynąć na wyniki. Każda aplikacja została uruchomiona na maszynie o odpowiedniej specyfikacji, zapewniając równe warunki dla wszystkich testów (Tabela 1).

Tabela 1. Spis podzespołów komputera, który posłużył do przeprowadzenia badań

Nazwa	Model
Procesor	Intel Core i7-10875H (maks. 5,10 GHz. 16 wątków, 8 rdzeni)
Pamięć RAM	16 GB
Dysk	1000 GB (SSD, M.2 PCIe)

Ponad to, napisane aplikacje zostały uruchomione z ograniczonym dostępem do zasobów, aby symulować typowe warunki serwerowe. Aplikacje były kontenerowane za pomocą obrazów Dockera, co zapewniło izolację poszczególnych środowisk testowych i uniknięcie potencjalnych konfliktów między nimi [9].

2.4. Zakres implementacji

Wszystkie porównywane aplikacje implementowały te same funkcjonalności, aby wyniki testów były jak najbardziej miarodajne. Każda aplikacja obsługiwała trzy punkty końcowe GET, odpowiedzialne za pobieranie danych z plików, bazy danych oraz za wykonanie złożonych obliczeń, a także dwa punkty końcowe POST do zapisywania danych. Implementacje różniły się jedynie technologią oraz narzędziami wykorzystywanymi w danym języku.

2.5. Podejście do złożoności kodu

Poza samą wydajnością aplikacji, uwzględniono również złożoność kodu i łatwość implementacji. Każdy język wymaga innego nakładu pracy programistycznej do osiągnięcia tych samych funkcji. Z tego względu zliczano liczbę linii kodu oraz oceniano, jak bardzo intuicyjne i łatwe do rozwinięcia są poszczególne implementacje. Ta część analizy pozwala na szersze spojrzenie na efektywność danego języka, nie tylko w kontekście wydajności, ale także z punktu widzenia utrzymania i rozwoju aplikacji.

3. Realizacja

W ramach realizacji badania, dla każdego z wybranych języków programowania stworzono aplikacje backendowe o identycznej funkcjonalności. Aplikacje te miały za zadanie obsługiwać zapytania GET i POST, w ramach których wykonywane były operacje na plikach, bazach danych oraz wykonywano obliczenia [10]. Poniżej znajduje się ogólny opis struktury kodu oraz wybranych rozwiązań implementacyjnych dla każdego języka.

3.1. Struktura aplikacji

Każda z aplikacji posiadała trzy punkty końcowe GET oraz dwa punkty POST:

- **GET** /api/files – zwracający dane z 10 plików. Testuje zdolność aplikacji do jednoczesnego otwierania i odczytu wielu plików.
- **GET** /api/database – zwracający dane z bazy danych. Testuje ocenę efektywności połączenia z bazą danych oraz wydajności zapytań do niej.
- **GET** /api/calc – zwracający wynik sortowania 5000 elementów za pomocą algorytmu sortowania bąbelkowego. Mierzy efektywność przetwarzania skomplikowanych operacji.
- **POST** /api/task – przyjmujący parametry i zapisujący do bazy danych. Testuje zapis do bazy oraz operacje CRUD.
- **POST** /api/measurement – przyjmujący listę liczb zmiennoprzecinkowych, sortuje je i zapisuje wynik do pliku.

3.2. Implementacja w językach programowania

W celu zapewnienia rzetelnego porównania wydajności, aplikacje miały identyczną funkcjonalność, jednak różniły się podejściem do implementacji, narzędziami oraz frameworkami specyficznymi dla danego języka. W przypadku C# wykorzystano framework ASP.NET Core, który umożliwia asynchroniczne przetwarzanie zapytań HTTP. Aplikacja w JavaScript została zbudowana z użyciem frameworka Express.js, popularnego w tworzeniu aplikacji serwerowych opartych na Node.js. Dla PHP zastosowano framework Laravel, który ułatwia zarządzanie bazami danych i operacjami CRUD. Z kolei dla Golang wykorzystano wbudowaną bibliotekę net/http, która zapewnia bezpośrednią obsługę zapytań HTTP i charakteryzuje się minimalizmem oraz wydajnością [11].

W poniższych podrozdziałach przedstawiono jeden z pięciu testowanych endpointów, celem wykazania różnic pomiędzy językami oraz implementacjami kodu.

3.2.1. Golang

W implementacji w Golang zastosowano mechanizmy do odczytu plików z systemu plików, aby pobrać zawartość dziesięciu plików tekstowych i połączyć ją w jeden string. W funkcji GetFiles wykorzystano pętlę, która iteruje od 1 do 10, budując dynamicznie ścieżki do plików za pomocą funkcji filepath.Join. Następnie, pliki są odczytywane synchronicznie za pomocą ioutil.ReadFile, a ich zawartość jest dodawana do obiektu strings.Builder, który umożliwia

efektywne łączenie tekstów. Po zakończeniu iteracji, pełny tekst jest zwracany, a ewentualne błędy związane z odczytem są obsługiwane przez zwrócenie odpowiedniego komunikatu błędu.

```
func GetFiles() (string, error) {
    var fullText strings.Builder

    for i := 1; i <= 10; i++ {
          filePath := filepath.Join("Data", fmt.Sprintf("test%d.txt", i))
          text, err := ioutil.ReadFile(filePath)
          if err != nil {
                return "", err
          }
          fullText.WriteString(string(text))
    }

    return fullText.String(), nil
}
```

3.2.2. C#

W implementacji w języku C# wykorzystano asynchroniczne podejście do odczytu zawartości plików z systemu plików. Metoda GetFiles odczytuje zawartość dziesięciu plików tekstowych znajdujących się w folderze Data, korzystając z metody File.ReadAllTextAsync, która umożliwia asynchroniczne wykonywanie operacji odczytu plików. W pętli for, każdy plik jest odczytywany i jego zawartość jest dodawana do obiektu StringBuilder, który efektywnie łączy teksty z plików. Po zakończeniu operacji, pełna zawartość plików jest zwracana jako jeden string, co umożliwia aplikacji jednoczesną obsługę innych żądań dzięki asynchronicznemu przetwarzaniu operacji I/O.

```
  [HttpGet("files")]
    public async Task<string> GetFiles()
    {
        StringBuilder fullText = new StringBuilder();

        for (int i = 1; i < 11; i++)
        {
            String text = await
System.IO.File.ReadAllTextAsync(Path.Combine(Environment.CurrentDirectory, "Data",
$"test{i}.txt"));
            fullText.Append(text);
        }

        return fullText.ToString();
    }
```

3.2.3. JavaScript (Node.JS)

Przedstawiony niżej kod napisany w języku JavaScript realizuje funkcję odczytu plików, analogicznie do wcześniejszych rozwiązań, z wykorzystaniem asynchronicznych operacji wejścia/wyjścia. W ramach funkcji obsługującej żądanie GET pod adresem '/api/files' zastosowano pętlę for, która iteruje przez dziesięć plików tekstowych znajdujących się w katalogu 'Data'. Ścieżki do tych plików są generowane dynamicznie za pomocą metody path.join. Zawartość każdego pliku jest odczytywana asynchronicznie przy użyciu funkcji fs.promises.readFile. Wynikowe dane są łączone w jeden ciąg znaków, który jest następnie zwracany w odpowiedzi HTTP do klienta. W przypadku napotkania błędu podczas odczytu któregokolwiek z plików, aplikacja zwraca odpowiedź z kodem statusu 500 oraz stosownym komunikatem błędów.

```
app.get('/api/files', async (req, res) => {
    try {
        let fullText = '';

        for (let i = 1; i <= 10; i++) {
            const filePath = path.join(__dirname, 'Data', `test${i}.txt`);
            const text = await fs.readFile(filePath, 'utf8');
            fullText += text;
        }
```

```
            res.send(fullText);
        } catch (err) {
            res.status(500).send('Error reading files');
        }
    });
```

3.2.4. PHP

W implementacji wykorzystującej język PHP oraz framework Laravel, funkcjonalność odczytu plików tekstowych została zaimplementowana w ramach endpointa typu GET pod adresem /files. Zastosowano pętlę for, która iteruje przez liczby od 1 do 10. W trakcie każdej iteracji dynamicznie generowane są ścieżki do plików za pomocą funkcji public_path(), która zwraca pełną ścieżkę do katalogu publicznego aplikacji. Do odczytu zawartości każdego pliku wykorzystano klasę File oraz jej metodę get(). Dane z poszczególnych plików są następnie dodawane do zmiennej $fullText. Po zakończeniu iteracji, pełna zawartość wszystkich plików jest zwracana do użytkownika w formacie JSON. Obsługa ewentualnych błędów związanych z odczytem plików jest automatycznie realizowana przez mechanizmy wyjątków wbudowane w framework Laravel, co zapewnia efektywne zarządzanie potencjalnymi błędami aplikacji.

```
Route::get('/files', function (Request $request) {
    $fullText = '';

    for ($i = 1; $i < 11; $i++)
    {
        $filename = public_path() . "/Data/test$i.txt";
        $fullText .= File::get($filename);
    }

    return response()->json($fullText);
});
```

4. Wyniki

Wyniki przeprowadzonych testów wydajnościowych dostarczyły istotnych danych na temat efektywności poszczególnych języków programowania w różnych scenariuszach obciążeniowych. W szczególności oceniano czas odpowiedzi aplikacji, liczbę przetwarzanych zapytań na sekundę oraz efektywność zarządzania zasobami systemowymi.

4.1. Zbieranie danych do analizy z rozwiązania napisanego w języku C#

Implementacja w języku C# zajmowała **18,6 MB** przestrzeni dyskowej i zawierała **141 linii kodu**. Podczas testów obciążeniowych z udziałem 100 użytkowników i 1000 zapytań aplikacja osiągnęła wydajność wynoszącą 421 zapytań na sekundę, z łącznym czasem wykonania 23 sekund i opóźnieniem sięgającym 238 sekund (Tabela 2). W testach dla pojedynczego użytkownika, powtórzonych sto razy, średni czas obsługi operacji na plikach wyniósł 15 sekund, dla zapytań do bazy danych 1394 sekundy, a dla operacji obliczeniowych 34 sekundy (Tabela 3). Przy obciążeniu 300 równoczesnych użytkowników średni czas odpowiedzi dla operacji na plikach wzrósł do 70 sekund, natomiast w przypadku zapytań do bazy danych aplikacja nie była w stanie obsłużyć wszystkich żądań (Tabela 4).

Tabela 2. Testy obciążeniowe zliczające wydajność w przetwarzaniu zapytań dla języka C#

Całkowity czas wykonywania	Zapytania/sekunda	Opóźnienie
23s	421	238s

Tabela 3. Testy obciążeniowe dla jednego użytkownika powtórzone sto razy dla języka C#

Endpoint	Średnio	Minimum	Maksimum
[GET] Files	15s	6s	120s
[GET] Database	1394s	1300s	2364s
[GET] Calc	34s	32s	41s
[POST] Task	12s	2s	85s
[POST] Measurement	2s	1s	18s

Tabela 4. Testy obciążeniowe dla 300 użytkowników jednocześnie dla języka C#

Endpoint	Średnio	Minimum	Maksimum
[GET] Files	70s	18s	143s
[GET] Database	N/A	N/A	N/A
[GET] Calc	4029s	47s	5238s
[POST] Task	3s	2s	38s
[POST] Measurement	2s	2s	5s

4.2. Zbieranie danych do analizy z rozwiązania napisanego w języku JavaScript (Node.js)

Rozwiązanie zaimplementowane w języku JavaScript zajmowało **3,1 MB** na dysku i składało się ze **100 linii kodu**. W testach obciążeniowych dla 100 użytkowników i 1000 zapytań, aplikacja osiągnęła wydajność na poziomie 227 zapytań na sekundę, z całkowitym czasem wykonania wynoszącym 44 sekundy i opóźnieniem 437 sekund (Tabela 5). Dodatkowo, testy dla jednego użytkownika, powtórzone sto razy, przyniosły średnie wyniki: czas obsługi plików wynosił 10 sekund, zapytań do bazy danych 1488 sekund, a operacji obliczeniowych 33 sekundy (Tabela 6). Przy zwiększeniu liczby użytkowników do 300, czas odpowiedzi dla operacji plikowych wynosił średnio 424 sekundy, a dla zapytań do bazy danych 168664 sekund (Tabela 7).

Tabela 5. Testy obciążeniowe zliczające wydajność w przetwarzaniu zapytań dla języka JavaScript (Node.js)

Całkowity czas wykonywania	Zapytania/sekunda	Opóźnienie
44s	227	437s

Tabela 6. Testy obciążeniowe dla jednego użytkownika powtórzone sto razy dla języka JavaScript (Node.js)

Endpoint	Średnio	Minimum	Maksimum
[GET] Files	10s	9s	15s
[GET] Database	1488s	1439s	1585s
[GET] Calc	33s	31s	38s
[POST] Task	6s	5s	14s
[POST] Measurement	2s	2s	5s

Tabela 7. Testy obciążeniowe dla 300 użytkowników jednocześnie dla języka JavaScript (Node.js)

Endpoint	Średnio	Minimum	Maksimum
[GET] Files	424s	15s	638s
[GET] Database	168664s	1825s	341305s
[GET] Calc	4821s	43s	9541s
[POST] Task	313s	12s	580s
[POST] Measurement	4s	3s	11s

4.3. Zbieranie danych do analizy z rozwiązania napisanego w języku PHP

Implementacja zrealizowana w języku PHP zajmowała **20,3 MB** przestrzeni dyskowej i zawierała **103 linie kodu**. W testach obciążeniowych przeprowadzonych dla 100 użytkowników i 1000 żądań aplikacja osiągnęła wydajność rzędu 10 zapytań na sekundę, z całkowitym czasem wykonania wynoszącym 960 sekund i opóźnieniem sięgającym 10 000 sekund (Tabela 8). Testy dla pojedynczego użytkownika, powtórzone sto razy, wykazały, że średni czas obsługi operacji na plikach wynosił 53 sekundy, zapytań do bazy danych 1 405 sekund, a operacji obliczeniowych 760 sekund (Tabela 9). Przy obciążeniu 300 równoczesnych użytkowników średni czas odpowiedzi dla operacji na plikach wyniósł 6 850 sekund, a dla zapytań do bazy danych 146 763 sekundy (Tabela 10).

Tabela 8. Testy obciążeniowe zliczające wydajność w przetwarzaniu zapytań dla języka PHP

Całkowity czas wykonywania	Zapytania/sekunda	Opóźnienie
960s	10	10000s

Tabela 9. Testy obciążeniowe dla jednego użytkownika powtórzone sto razy dla języka PHP

Endpoint	Średnio	Minimum	Maksimum
[GET] Files	53s	51s	56s
[GET] Database	1405s	1361s	1511s
[GET] Calc	760s	732s	789s
[POST] Task	51s	48s	56s
[POST] Measurement	52s	49s	60s

Tabela 10. Testy obciążeniowe dla 300 użytkowników jednocześnie dla języka PHP

Endpoint	Średnio	Minimum	Maksimum
[GET] Files	6850s	65s	13565s
[GET] Database	146763s	1576s	309852s
[GET] Calc	69745s	833s	127482s
[POST] Task	8205s	94s	15601s
[POST] Measurement	7237s	67s	14362s

4.4. Zbieranie danych do analizy z rozwiązania napisanego w języku Go

Rozwiązanie zaimplementowane w języku Go zajmowało **0,4 MB** na dysku i składało się z **156 linii kodu**. To jedno z najmniejszych rozwiązań pod względem rozmiaru plików, co wynika z minimalizmu biblioteki net/http oraz samego języka, który został zaprojektowany z myślą o prostocie i efektywności. W aplikacji zastosowano asynchroniczne przetwarzanie zapytań oraz efektywne zarządzanie pamięcią, co umożliwiło szybkie operacje na plikach oraz wykonywanie złożonych obliczeń w krótkim czasie. W testach obciążeniowych przeprowadzonych dla 100

użytkowników i 1000 zapytań, aplikacja napisana w Go osiągnęła wydajność na poziomie 392 zapytań na sekundę, co świadczy o wysokiej zdolności do równoczesnej obsługi wielu żądań. Całkowity czas wykonania wyniósł 25 sekund, z opóźnieniem wynoszącym 254 sekundy (Tabela 11).

Tabela 11. Testy obciążeniowe zliczające wydajność w przetwarzaniu zapytań dla języka Go

Całkowity czas wykonywania	Zapytania/sekunda	Opóźnienie
25s	392	254s

Testy przeprowadzone dla jednego użytkownika, powtórzone 100 razy, wykazały, że średni czas obsługi plików wynosił 6 sekund, zapytań do bazy danych 1340 sekund, a operacji obliczeniowych 31 sekund (Tabela 12). Tego typu wyniki pokazują, że Go radzi sobie doskonale zarówno z prostymi operacjami na plikach, jak i bardziej złożonymi operacjami na bazach danych.

Tabela 12. Testy obciążeniowe dla jednego użytkownika powtórzone sto razy dla języka Go

Endpoint	Średnio	Minimum	Maksimum
[GET] Files	6s	5s	11s
[GET] Database	1340s	1282s	1415s
[GET] Calc	31s	30s	34s
[POST] Task	9s	8s	13s
[POST] Measurement	1s	1s	3s

Przy 300 równoczesnych użytkownikach, średni czas odpowiedzi dla operacji plikowych wyniósł 52 sekundy, co pokazuje, że Go potrafi zachować względnie niskie czasy odpowiedzi nawet przy zwiększonym obciążeniu. Jednak w przypadku zapytań do bazy danych aplikacja napisana w Go nie była w stanie przetworzyć wszystkich żądań (Tabela 13).

Tabela 13. Testy obciążeniowe dla 300 użytkowników jednocześnie dla języka Go

Endpoint	Średnio	Minimum	Maksimum
[GET] Files	52s	25s	86s
[GET] Database	N/A	N/A	N/A
[GET] Calc	3390s	48s	7076a
[POST] Task	12s	9s	27s
[POST] Measurement	2s	1s	4s

4.5. Zbiorcze przedstawienie wyników i analiza

W tej części przedstawione zostały zbiorcze wyniki testów wydajnościowych dla aplikacji napisanych w językach C#, JavaScript, PHP oraz Go. Porównano kluczowe wskaźniki, takie jak liczba przetwarzanych zapytań na sekundę, czas odpowiedzi oraz złożoność kodu. Analiza wyników pozwala ocenić, jak każdy z języków radzi sobie zarówno przy niskim, jak i wysokim obciążeniu, co pozwala lepiej zrozumieć mocne i słabe strony poszczególnych technologii. Szczegółowa interpretacja danych umożliwia także wskazanie najlepszych rozwiązań do tworzenia wydajnych i skalowalnych aplikacji backendowych.

Tabela 14. Wyniki zbiorcze dla poszczególnych rozwiązań

Język	Rozmiar plików	Ilość linii kodu	Maksymalna ilość zapytań /s
C#	18.6MB	141	**421/s**
JavaScript	3.1MB	**100**	227/s
PHP	20.3MB	103	10/s
GO	**0.4MB**	156	392/s

Analizując wyniki, wyraźnie widoczna jest duża różnica w rozmiarze projektu między językiem Go a pozostałymi językami. Mimo że Go zajmował najmniej miejsca na dysku, wymagał największej liczby linii kodu, co wynika z jego niskopoziomowego charakteru (Tabela 14) [2]. Ciekawym spostrzeżeniem jest fakt, że liczba zapytań przetwarzanych na sekundę była bardzo podobna zarówno dla Go, jak i C#. Oznacza to, że oba te języki efektywnie radzą sobie z wielowątkowością oraz odpowiednim zarządzaniem zapytaniami. Zjawisko to podkreśla przewagę języków kompilowanych w tworzeniu wydajnych systemów przystosowanych do obsługi wielu użytkowników jednocześnie [12]. Aby lepiej zrozumieć mocne i słabe strony każdego z rozwiązań, warto przeanalizować podsumowanie wyników dla wszystkich punktów końcowych (Tabela 15).

Tabela 15. Wyniki zbiorcze dla poszczególnych rozwiązań dla jednego użytkownika powtórzone sto razy

Endpoint	C#	PHP	JavaScript	Go
[GET] Files	15s	53s	10s	**6s**
[GET] Database	1394s	1405s	1488s	**1340s**
[GET] Calc	34s	760s	33s	**31s**
[POST] Task	12s	51s	**6s**	9s
[POST] Measurement	2s	52s	2s	**1s**

Testy zostały przeprowadzone z udziałem jednego użytkownika i powtórzone sto razy, aby uzyskać bardziej precyzyjne średnie wyniki. Nawet przy pojedynczym żądaniu do API można było zaobserwować, że język Go przewyższał pozostałe rozwiązania, choć różnice były niewielkie. Jednak w sytuacji zwiększenia skali problemu różnice w czasach wykonania prawdopodobnie stałyby się bardziej znaczące. Również JavaScript i C# osiągnęły bardzo dobre wyniki w kontekście pojedynczych żądań do API, co świadczy o ich potencjale w efektywnej obsłudze tego typu operacji.

Tabela 16. Wyniki zbiorcze dla poszczególnych rozwiązań dla 300 użytkowników

Endpoint	C#	PHP	JavaScript	Go
[GET] Files	70s	6850s	424s	**52s**
[GET] Database	N/A	**146763s**	168664s	N/A
[GET] Calc	4029s	69745s	4821s	**3390s**
[POST] Task	**3s**	8205s	313s	9s
[POST] Measurement	2s	7237s	4s	**1s**

Wyniki testów przy większym obciążeniu ukazują istotne różnice w sposobie zarządzania zasobami przez poszczególne języki (Tabela 16). Należy zwrócić uwagę na oznaczenie "N/A", które wskazuje sytuacje, w których dana aplikacja nie była w stanie zakończyć zadania z powodu wyczerpania dostępnych zasobów. Widać, że przy dużym obciążeniu i ograniczonych zasobach, języki interpretowane radziły sobie lepiej. Wykorzystują wszystkie dostępne zasoby i kosztem dłuższego czasu przetwarzania odpowiadają na każde kolejne zapytanie. Natomiast języki kompilowane, po wykorzystaniu dostępnej mocy obliczeniowej, przestawały działać. Mimo tego w pozostałych przypadkach rozwiązanie oparte na języku Go ponownie okazało się najskuteczniejsze, a zwiększenie zasobów mogłoby znacząco poprawić jego wyniki. Przy większej liczbie zapytań widoczna była przewaga języków C# i Go,

natomiast JavaScript nie radził sobie równie dobrze z obsługą wielowątkowości, co spowodowało znaczące pogorszenie wyników.

5. Podsumowanie i wnioski

Przeprowadzone badania pozwoliły na szczegółowe porównanie wydajności czterech popularnych języków programowania backendowego: Golang, C#, JavaScript (Node.js) i PHP. Każdy z tych języków wykazał inne mocne i słabe strony, co daje cenny wgląd w to, jak te technologie sprawdzają się w różnych scenariuszach obciążeniowych i implementacyjnych.

Język Golang okazał się szczególnie wydajny w obsłudze dużych obciążeń, co wynika z jego efektywnej obsługi wielowątkowości oraz niewielkich wymagań dotyczących zasobów systemowych. Mimo że wymaga on większej liczby linii kodu ze względu na niższy poziom abstrakcji, jego szybkość i stabilność sprawiają, że jest to jeden z najlepszych wyborów do budowy skalowalnych aplikacji o dużym ruchu.

C# również wykazał się wysoką wydajnością, zwłaszcza w zakresie wielowątkowości. Jego zaletą jest również łatwiejsza składnia i duża liczba dostępnych narzędzi wspierających programistów, co czyni go dobrym wyborem dla dużych korporacyjnych aplikacji. Mimo nieco większego zapotrzebowania na zasoby w porównaniu do Golanga, język ten okazał się bardzo konkurencyjny.

JavaScript, mimo swojej popularności, radził sobie gorzej w testach obciążeniowych, szczególnie w sytuacjach wymagających obsługi wielu równoczesnych zapytań. Jego asynchroniczna natura dobrze sprawdza się w mniej wymagających środowiskach, jednak w warunkach dużego obciążenia wydajność była zauważalnie niższa niż w przypadku Golanga i C#.

PHP, mimo swojej popularności i długiej historii na rynku, wykazał najniższą wydajność, szczególnie przy obsłudze wielu równoczesnych użytkowników. Wymagał większych zasobów i dłuższego czasu na przetwarzanie zapytań, co może ograniczać jego zastosowanie w nowoczesnych, wysoko obciążonych aplikacjach.

Dalsze prowadzenie badań nad tym zagadnieniem może obejmować następujące aspekty:

- Przeprowadzenie testów wydajnościowych dla innych popularnych języków backendowych, takich jak Ruby, Python, Rust czy Elixir, aby uzyskać pełniejszy obraz możliwości technologicznych.
- Analizę wpływu różnych środowisk chmurowych (AWS, Google Cloud, Microsoft Azure) na wydajność aplikacji backendowych w poszczególnych językach [13].
- Przeprowadzenie badań nad wpływem różnych technik optymalizacyjnych na wydajność aplikacji, takich jak zmiany w algorytmach, strukturyzacja kodu oraz wykorzystanie narzędzi takich jak kompilatory Just-In-Time (JIT) [14]
- Zbadanie wydajności w kontekście bardziej złożonych aplikacji, które wykorzystują zaawansowane operacje na danych, obsługę dużych baz danych czy skomplikowane algorytmy.
- Analizę, jak każdy z języków radzi sobie z różnymi formami skalowania (dodawanie zasobów w pionie i poziomie), w tym obsługą mikrousług i konteneryzacji.
- Rozszerzenie badań o aspekty bezpieczeństwa, np. analiza odporności aplikacji na ataki, szybkość wprowadzenia poprawek bezpieczeństwa oraz zarządzanie uwierzytelnianiem i autoryzacją.
- Zbadanie, jak różne architektury systemów, takie jak mikroserwisy, monolity czy serwerless, wpływają na wydajność w każdym z języków [1].
- Badania nad kosztami utrzymania i rozwoju aplikacji backendowych w różnych językach, z uwzględnieniem zasobów potrzebnych do utrzymania wydajności na wysokim poziomie.

Reference

1. S. Newman. Building Microservices: Designing Fine-Grained Systems. ISBN: 978-1492034025, 2021
2. A. Rios. System Programming Essentials with Go: System calls, networking, efficiency, and security practices with practical projects in Golang. ISBN: 978-1837634132, 2024
3. D. Kopec. History of Web Programming. DOI: 10.1007/978-1-4302-6482-8_20, 2014
4. A. Hoffman. Web Application Security: Exploitation and Countermeasures for Modern Web Applications (2nd edition). ISBN: 978-1098143930, 2024
5. M. Abbott, M. Fisher. Art of Scalability, The: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise (2nd edition). ISBN: 978-0134032801, 2015

6. RB Whitaker. The C# Player's Guide (5th edition), ISBN: 978-0985580155, 2022
7. J. Wilson. Node.js the Right Way: Practical, Server-Side JavaScript That Scales, ISBN: 978-1937785734, 2013
8. M. MacDonald. WordPress: The Missing Manual: The Book That Should Have Been in the Box (3rd edition). ISBN: 978-1492074168, 2020
9. N. Poulton. Docker Deep Dive: Zero to Docker in a single book. ISBN: 978-1916585256, 2023
10. S. Allamaraju. RESTful Web Services Cookbook: Solutions for Improving Scalability and Simplicity. ISBN: 978-0596801687, 2010
11. N. Yellavula. Hands-On RESTful Web Services with Go: Develop elegant RESTful APIs with Golang for microservices and the cloud (2nd edition), ISBN: 978-1838643577, 2020
12. M. Scott. Programming Language Pragmatics. ISBN: 978-0124104099, 2015
13. R. McKendrick. Infrastructure as Code for Beginners: Deploy and manage your cloud-based services with Terraform and Ansible. ISBN: 978-1837631636, 2023
14. J. Aycock. A brief history of just-in-time. DOI: 10.1145/857076.857077, 2003