JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 23 (2022) 128–134
Received: 29 March 2022
Accepted: 27 April 2022

Comparative analysis of frameworks using TypeScript to build server applications

Analiza porównawcza szkieletów programistycznych wykorzystujących TypeScript do tworzenia aplikacji serwerowych

Marcin Golec*, Małgorzata Plechawska-Wójcik

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract

The subject of the research was a comparative analysis of programming frameworks that are intended for building applications. NestJS (version 8.1.1), FoalTS (version 2.5.0) and Ts.ED (version 6.69.1) were put together. An experiment was prepared based on scenarios that focused on the response time of test applications to requests. Each of them had the same set of functionalities. NestJS turned out to be the most efficient of the compared skeletons. It achieved the best results. The worst results in each scenario were achieved by Ts.ED, especially with higher loads. The biggest differences in the comparison can be seen in studies conducted according to a scenario based on GET-type requests, and in particular with more objects in response.

Keywords: Typescript frameworks; node.js; performance comparative analysis; server application

Streszczenie

Przedmiotem badań była analiza porównawcza szkieletów programistycznych, które są przeznaczone do budowania aplikacji. Zestawiono ze sobą NestJS (wersja 8.1.1), FoalTS (wersja 2.5.0) oraz Ts.ED (wersja 6.69.1). Został przygotowany eksperyment przeprowadzony według scenariuszy, które skupiały się na czasie odpowiedzi na żądania przez aplikacje testowe. Każda z nich posiadała ten sam zestaw funkcjonalności. Z porównywanych szkieletów najwydajniejszym okazał się NestJS. Osiągał on najlepsze wyniki. Najgorsze wyniki w każdym scenariuszu osiągał Ts.ED, a w szczególności przy większych obciążeniach. Największe różnice przy porównaniu widać w badaniach przeprowadzonych według scenariusza opierającego się na żądaniach typu GET, a w szczególności z większą ilością obiektów w odpowiedzi.

Słowa kluczowe: szkielety programistyczne TypeScript; node.js; porównanie wydajności; aplikacje serwerowe

*Corresponding author

Email address : marcin.golec@pollub.edu.pl (M. Golec)

©Published under Creative Common License (CC BY-SA v4.0)

1. Wstęp

Zainteresowanie aplikacjami internetowymi utrzymuje się na bardzo wysokim poziomie. Rezultatem tego trendu jest zwiększone zainteresowanie technologiami do tworzenia tego typu aplikacji. Przybywa narzędzi, które usprawniają pracę przy tworzeniu elementów składowych aplikacji internetowych.

Możliwość korzystania z tego samego języka programowania przy rozwijaniu zarówno części dla użytkownika jak i części do zadań serwerowych stanowi zdecydowane ułatwienie dla pracy programisty. Takim językiem programowania, popularnym od wielu lat, jest JavaScript. Jednak język ten może być trudny dla osób początkujących. Firma Microsoft wydała język programowania TypeScript, który jest kompilowany do JavaScript. Dzięki zastosowaniu TypeScript wytwarzanie kodu pozbawione jest błędów, które zostają wyłapane na etapie kompilacji. Obecnie wiele szkieletów programistycznych korzysta lub umożliwia wykorzystanie tego języka programowania.

Niniejszy artykuł powstał w odpowiedzi na rosnące zainteresowanie szkieletami programistycznymi tworzonymi w oparciu o TypeScript. Jego celem jest porównanie trzech wybranych rozwiązań: NestJS [1], FoalTS [2] oraz Ts.ED [3]. Rezultatem artykułu jest utworzenie rankingu ocenianych szkieletów, pod względem wydajności. W tym celu opracowano scenariusze badawcze, w których dokonano pomiarów czasów odpowiedzi na żądania przez serwerowe aplikacje testowe. Do realizacji tego zadania wykorzystano wybrane szkielety programistyczne. Z ich pomocą wykonano trzy aplikacje zawierające te same funkcjonalności.

Każdy z wybranych szkieletów wykorzystuje Node.js [4] jako środowisko uruchomieniowe. Jest ono już znane wśród twórców aplikacji internetowych od kilkunastu lat. Popularność zyskuje z powodu wykorzystywanego języka programowania JavaScript oraz uruchamiania na wydajnym silniku V8 utrzymywanego przez Google. Coraz więcej firm decyduje się na wykorzystanie Node.js w projektach komercyjnych.

Celem badań było odwzorowanie sytuacji, które są najczęściej spotykane podczas realizacji zadań, do których wybrane szkielety programistyczne zostały stworzone. Nieodłączny element aplikacji serwerowej to komunikacja z zewnętrznymi serwisami. Z tego powodu do badań zaimplementowano obsługę żądań typu GET,

POST, PUT i DELETE. Wykorzystane dane do badań były przechowywane w pamięci komputera.

Niniejszy artykuł poświęcony jest analizie szkieletów programistycznych wykorzystujących TypeScript. Głównym celem artykułu jest porównanie wydajności definiowanej czasem obsługi żądań wybranych szkieletów programistycznych.

2. Przegląd literatury

Język PHP był i wciąż jest najpopularniejszym językiem programowania wykorzystywanym do tworzenia aplikacji internetowych. W artykule [6] porównany został on z aplikacją napisaną w języku JavaScript, pracującą na platformie Node.js. Przedstawione zostały scenariusze badawcze, które opierały się na obliczeniach liczby Fibonacciego oraz odczycie dużych plików tekstowych. Wyniki badań były korzystniejsze dla Node.js. Jedynie w przypadku obliczeń matematycznych przegrywał on z PHP.

W artykule [7] autor porównuje szkielety programistyczne do tworzenia aplikacji serwerowych. Wykorzystują one język programowania JavaScript. Porównywanymi szkieletami są: Hapi, Koa i Express. W celu porównania zmierzone zostały czasy odpowiedzi na żądania. Danymi, które były wykorzystane przy badaniach to prosty ciąg znaków oraz zmienna liczba obiektów. W rezultacie można zauważyć, że przy większych obciążeniach różnice pomiędzy porównywanymi szkieletami zwiększają się. Przy mniejszej liczbie obiektów rozbieżności wyników nie były aż tak widoczne.

Porównanie Node.js, PHP i Python jest elementem artykułu [8]. Jego głównym założeniem było porównanie ich wydajności. Nie wykorzystano tutaj żadnych szkieletów programistycznych, a jedynie języki programowania. Autorzy porównują czasy odpowiedzi na żądania oraz czas potrzebny na wykonanie obliczenia liczby na danej pozycji w ciągu Fibonacciego. W scenariuszach badawczych była uwzględniona liczba użytkowników korzystających jednocześnie z aplikacji. W wyniku przeprowadzonych porównań stwierdzono, że najefektywniejszy był Node.js, ale nie radził on sobie przy obliczeniach matematycznych.

Tworzenie aplikacji serwerowej z wykorzystaniem NestJS jest przedstawione w artykule [9]. Autor porusza kwestie związane z istotnymi aspektami w procesie tworzenia kodu. Jako główne elementy poddawane ocenie zostały wybrane: architektura, testowanie, wydajność, dokumentacja, społeczność korzystająca z tego rozwiązania oraz tempo uczenia się tego narzędzia. Każdemu z tych tematów został poświęcony odpowiedni fragment analizy poparty zarówno wykresami jak i badaniami opartymi o doświadczenie innych programistów. Artykuł przedstawia aspekty związane bezpośrednio z NestJS oraz metody wykorzystywane do badania wydajności szkieletów programistycznych.

Wykorzystanie języka JavaScript do tworzenia prostej aplikacji serwerowej jest przedstawione w artykule [10]. Autor przedstawia listingi zawierające kod aplikacji uruchamianej na platformie Node.js realizującej operacje: zwracanie odpowiedzi tekstowej, wysyłanie obrazów przez serwer, odczytywanie strumieni z dysku oraz asynchroniczny i synchroniczny dostęp do plików. Kolejna część artykułu zawiera fragmenty kodu przedstawiające funkcjonalności takie jak: obsługa żądań, definiowanie tras (ang. routes) oraz obsługa plików statycznych. Zaimplementowane są wykorzystując szkielet programistycznego Express.js [11], który dostarcza interfejs do wybranych podstawowych funkcjonalności Node.js. Rezultatem artykułu było przedstawienie zalet korzystania z połączenia Node.js i Express.js do budowaniu aplikacji serwerowych napisanych w języku JavaScript.

3. Technologie i narzędzia

3.1. Node.js

Node.js [4] jest środowiskiem uruchomieniowym dla kodu napisanego w języku programowania JavaScript po stronie serwerowej. Stworzone w tym środowisku aplikacje, dzięki działaniu asynchronicznym, nie wymagają tworzenia osobnych wątków dla każdego z żądań przesyłanych do serwera. Zastosowanie architektury zdarzeń ograniczyło spowolnienie działania. Fragmenty kodu odpowiedzialne za wczytywanie lub zapisywanie danych nie powodują zatrzymania wykonywania pozostałego kodu całej aplikacji. Kolejne instrukcje są przetwarzane do momentu, kiedy zapis lub odczyt danych zostanie zakończony. Wtedy wykonywany jest kod, który został zdefiniowany do zrealizowania po zakończeniu tych zdarzeń.

3.2. Postman

Postman [5] jest to narzędzie, które służy do pracy z interfejsami programistycznymi aplikacji. Posiada dużą liczbę funkcji, które ułatwiają zarówno testy i rozwój tych interfejsów. Jest narzędziem bezpłatnym do użytku osobistego. Jedną z funkcji szczególnie przydatnych w kontekście tej pracy jest możliwość wielokrotnego wysyłania żądania do serwera. Każde z tak wysłanych zapytań po wykonaniu zawiera zarówno odpowiedź od serwera jak i czas potrzebny do przetworzenia.

Możliwość utworzenia żądania, które będzie symulacją oczekiwanego przez aplikację ułatwia testowanie i wykonywanie pomiarów. Dla każdego z zapytań można określić parametry zawarte w URL, nagłówek, ciało żądania lub elementów związanych z autoryzacją. Wszystkie obecnie panujące standardy (REST, SOAP, GraphQL, WebSocket) wykorzystywane do komunikacji z aplikacjami serwerowymi są możliwe do skonfigurowania w tym programie. Powtórzenie wielokrotne dokładnie tak samo skonstruowanego żądania i wyeksportowanie wyników daje pogląd na to jak wydajnie serwer przetwarza dane zadanie.

3.3. NestJS

Jest to szkielet programistyczny, który jest ciągle rozwijany i ulepszany oraz opiera się głównie na języku

TypeScript. Używając tego narzędzia, można także programować w języku JavaScript. Szkielet ten jest wykorzystywany do tworzenia aplikacji serwerowych. Kod pisany w szkielecie jest kombinacją programowania obiektowego (OOP), programowania funkcyjnego (FP) oraz programowania funkcyjnego reaktywnego (FRP). Jako serwera http NestJS używa solidnego szkieletu programistycznego jakim jest Express.js. Proces wytwarzania aplikacji jest przyspieszony dzięki wykorzystaniu CLI, które jest dostarczone wraz ze szkieletem. Dzięki temu aplikacja już na starcie posiada uschematyzowaną strukturę.

3.4. Ts.ED

Proces rozwoju tego szkieletu programistycznego jest na bardzo zaawansowanym etapie. Autorzy przygotowali wiele gotowych pakietów startowych dla tego szkieletu. Przykładami mogą być: pakiet z gotowymi elementami do pracy z AWS, MongoDB, Vue.js, React i wiele innych. Podobnie jak w przypadku szkieletu NestJS, głównym założeniem jest programowanie obiektowe oraz dekoratory. Domyślnym serwerem HTTP jest Express.js, ale istnieje możliwość zmiany w konfiguracji na szkielet Koa.js. Programista używający tego szkieletu ma dostęp do interfejsu, przez który może sobie skonfigurować dowolne inne rozwiązanie i dzięki temu może według swoich preferencji dostosować środowisko robocze.

3.5. FoalTS

FoalTS jest szkieletem programistycznym zawierającym bogaty zestaw komponentów, które ułatwiają implementację aplikacji. Wraz ze szkieletem dostarczane są narzędzia do testowania, CLI, narzędzia do integracji z aplikacjami internetowymi, skrypty, zaawansowana autentykacja, obsługa bazy danych, środowiska wdrożeniowe, GraphQL i Swagger API, narzędzia do AWS oraz wiele innych. Podstawowa architektura aplikacji stworzonej przy pomocy FoalTS jest zorientowana na trzy główne komponenty: kontrolery, serwisy oraz hooki. Każdy z elementów może zostać rozszerzony i odpowiednio dostosowany do wymagań implementacyjnych.

4. Metoda badań

4.1. Opis eksperymentu

Do przeprowadzenia porównania szkieletów programistycznych do budowania aplikacji serwerowych opracowano pięć scenariuszy badawczych. W czterech pierwszych scenariuszach wykonano pomiary czasów odpowiedzi na żądania typu: GET, POST, PUT i DELETE. Krótszy czas potrzebny do obsługi pojedynczego żądania oznaczał lepszy wynik. Pomiary były wykonywane przy pomocy wbudowanej w program Postman funkcji o nazwie „Runner", która każde żądanie wywoływała w seriach 1000 razy.

4.2. Aplikacje testowe

Stworzone zostały trzy aplikacje testowe posiadające ten sam zestaw funkcjonalności. Każda aplikacja była wygenerowana automatycznie poprzez CLI dostarczone wraz ze szkieletami programistycznymi. Żaden kod nie był usuwany, dodane zostały jedynie linie, które służyły do obsługi żądań. Stworzone aplikacje miały na celu obsłużyć serię żądań HTTP. Do badań został przygotowany zbiór danych zawierający obiekty w formacie JSON o strukturze przedstawionej na Listingu 1. Symulowało to bazodanowe obiekty, które są najczęściej wykorzystywane w tego typu aplikacjach. Zestaw danych był wczytywany przy starcie aplikacji i przechowywany był w pamięci komputera. Wczytanie danych było wykonywane w momencie startu aplikacji, który nie powodował zakłóceń przy pomiarach czasu potrzebnego na obsługę wysyłanych żądań. Realizacja funkcji wywoływanych w serii zapytań była całkowicie odseparowana od czynników zewnętrznych, które mogłyby mieć wpływ na uzyskany wynik.

Listing 1: Struktura obiektu wykorzystywanego do badań

```
export type PhotoDto = {
    albumId: number;
    id: number;
    title: string;
    url: string;
    thumbnailUrl: string;
}
```

4.3. Stanowisko badawcze

W Tabeli 1 znajdują się parametry stanowiska, które zostało wykorzystane do przeprowadzenia badań.

Tabela 1: Parametry stanowiska badawczego

Sprzęt	
Rodzaj sprzętu	Laptop podłączony do zasilania
Procesor	Intel(R) Core(TM) i7-7700HQ
Pamięć RAM	20 GB
Karta sieciowa	Realtek 8821AE Wireless LAN 802.11ac PCI-E NIC
System operacyjny	Windows 10 Education N
Oprogramowanie	
Node.js	16.13.2
NestJS	8.1.1
FoalTS	2.5.0
Ts.ED	6.69.1
Postman	9.08

5. Wyniki badań

Badania zostały przeprowadzone z podziałem na liczbę zwracanych obiektów. W przypadku żądań GET było to dla 1, 100, 500, 1000, 5000 oraz 10000 obiektów. Dla badań dotyczących żądań typu POST, PUT i DELETE była wykonywana jedna seria 1000 powtórzeń. Zbierano wyniki dotyczące czasu obsługi zapytania, który był wyznaczany jako czas od momentu otrzymania żądania do momentu zwrócenia odpowiedzi. Na podstawie uzyskanych danych policzone zostały takie wskaźniki jak: średnia arytmetyczna, odchylenie standardowe oraz wartości minimalne i maksymalne. Odchylenie standardowe jest wskaźnikiem, który

pozwala na zobrazowanie w postaci liczbowej tego, jak bardzo wyniki uzyskane w badaniach są oddalone od średniej arytmetycznej. Im mniejszy wynik uzyskany przy obliczaniu odchylenia standardowego, tym uzyskane wyniki są bliższe średniej. Dla sprawdzenia istotności różnic pomiędzy porównywanymi szkieletami przeprowadzono analizy statystyczne przy użyciu testów: jednoczynnikowej analizy wariancji (ANOVA) oraz testu Tukeya. Wybór testu ANOVA był podyktowany faktem, iż porównywane dane pochodziły z więcej niż dwóch (w tym przypadku trzech) niezależnych grup. Natomiast test Tukeya stosuje się w momencie, kiedy wykryte w teście ANOVA różnice są istotne statystycznie dla dokładniejszego wskazania par różnych od siebie. Przed przystąpieniem do testu ANOVA dokonano sprawdzenia i potwierdzono spełnienie założeń dotyczących normalności rozkładu i jednorodności wariancji. Rezultat przeprowadzenia testu ANOVA pozwolił na zweryfikowanie czy występują różnice pomiędzy którąkolwiek parą szkieletów, a test Tukeya umożliwił wskazanie, które z nich były istotnie różne względem siebie.

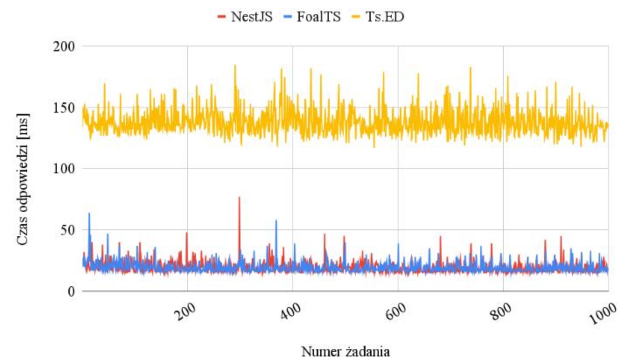
Na potrzeby tego artykułu z szeregu przeprowadzonych badań wybrano wyniki, które są najbardziej znaczące. Zawarta na końcu tego rozdziału tabela, stanowi podsumowanie zebranych wyników i jest próbą oceny badanych szkieletów programistycznych według ustalonych kryteriów.

Pierwsze przedstawione wyniki dotyczą czasu odpowiedzi serwera na żądania typu GET dla 10000 obiektów. Obrazuje to najlepiej różnice jakie wystąpiły pomiędzy badanymi szkieletami. Tabela 2 prezentuje wartości otrzymane z obliczeń statystycznych.

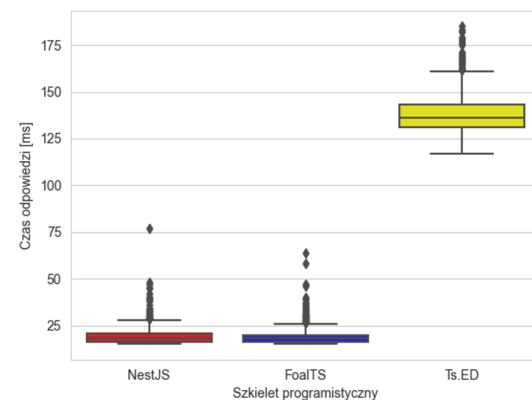
Tabela 2: Czasy odpowiedzi na żądania typu GET zwracającego tablicę 10000 obiektów

Szkielet	Czas min. [ms]	Czas maks. [ms]	Czas śr. [ms]	Odchylenie std. [ms]
NestJS	15	77	19,386	5,005
FoalTS	15	64	19,288	4,512
Ts.ED	117	185	138,217	10,584

Rysunki 2 i 3 przedstawiają wykresy dla uzyskanych wyników. Zaprezentowano wyniki w postaci dwóch typów wykresów: liniowego i pudełko-wąsy. Na wykresie liniowym został zawarty uzyskany czas odpowiedzi z każdego powtórzenia wykonanego w badaniu. Forma linii pozwala na zobrazowanie rozstrzału uzyskiwanych wartości – kształt linii odbiegający od prostego pokazuje, że wyniki są bardziej odległe względem siebie. Na wykresie pudełko-wąsy zostały zaprezentowane wartości pochodzące z obliczeń statystycznych. Rombami oznaczone są wartości odstające powyżej maksymalnych, które nie mieszczą się w ramach kwartyla. Górna linia „wąsa" odpowiada wartości maksymalnej. Górna krawędź pudełka jest rozpoczęciem trzeciego kwartyla, a odpowiednio dla pierwszego kwartyla jest to dolna krawędź. Pierwszy kwartyl jest odwzorowaniem wyniku, który jest powyżej 25% wszystkich obserwacji, a wartość trzeciego kwartylu jest powyżej 75% [12]. Istotną kwestią jest fakt, że wyniki są posortowane w kolejności rosnącej. Linia znajdująca się w środku pudełka reprezentuje medianę, a linia w dolnej części „wąsa" poniżej pudełka prezentuje wartość minimalną. W przypadku, kiedy na wykresie brakuje linii wewnątrz pudełka oznacza to, że wartość mediany jest taka sama jak jednego z dwóch kwartyli. Taką sytuację można zaobserwować przy wynikach badania czasu odpowiedzi na żądania POST, PUT i DELETE.



Rysunek 2: Czasy odpowiedzi na żądania typu GET zwracającego tablicę 10000 obiektów.



Rysunek 3: Czasy odpowiedzi na żądania typu GET zwracającego tablicę 10000 obiektów.

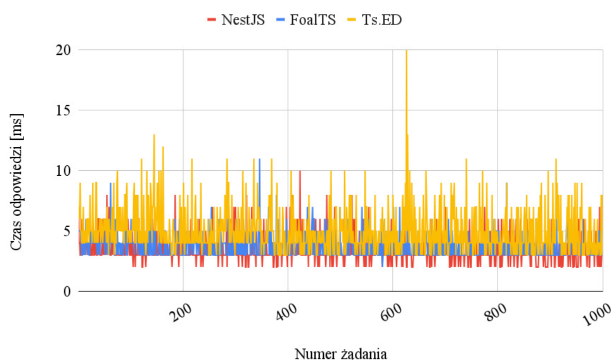
Jak widać na powyżej zaprezentowanych wynikach najwolniejszym szkieletem programistycznym okazał się *Ts.ED*. Natomiast na uwagę zasługuję tutaj fakt, że szkielety *NestJS* i *FoalTS* uzyskały wyniki zbliżone do siebie. Jest to o tyle ciekawe, że w badaniach na mniejszej liczbie zwracanych obiektów to pierwszy z nich uzyskiwał zdecydowanie lepsze wyniki. Przy tym badaniu test Tukeya nie wykazał istotnych różnic pomiędzy tymi dwoma szkieletami.

Kolejne badania dotyczyły żądań typu POST, PUT i DELETE. Wyniki dla nich zostały przedstawione w takiej samej formule jak te dla żądań typu GET, czyli najpierw tabela z wynikami liczbowymi, a potem graficzna prezentacja w postaci wykresów. Jako pierwsze zostały zaprezentowane rezultaty z badania dotyczącego żądań typu POST. Zmierzono czas od momentu wysłania do serwera zapytania zawierającego nowy obiekt JSON z taką samą strukturą jak te, które

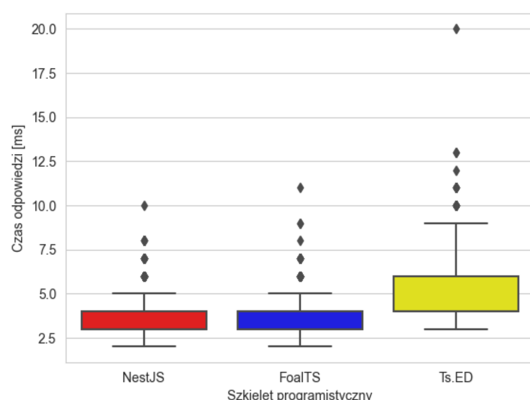
były wykorzystywane we wcześniejszych badaniach. Każdy serwer został poddany serii 1000 powtórzeń, a w każdym z nich był wykorzystany ten sam obiekt z tak samo zdefiniowanymi właściwościami. Ze względu na znacząco odstający wynik pierwszego pomiaru został on pominięty w prezentacji i obliczeniach. Wyniki zostały zaprezentowane w Tabeli 3 oraz na Rysunkach 4 i 5. Wyniki pokazują, że *NestJS* okazał się najszybszy, a *FoalTS* i *Ts.ED* były odpowiednio na drugim i trzecim miejscu. Przy tym badaniu testy statystyczne ANOVA i Tukeya wykazały istotne różnice dla każdej porównywanej pary.

Tabela 3: Czasy odpowiedzi na żądania typu POST

Szkielet	Czas min. [ms]	Czas maks. [ms]	Czas śr. [ms]	Odchylenie std. [ms]
NestJS	2	10	3,456	1,177
FoalTS	2	11	3,770	0,981
Ts.ED	3	20	4,947	1,823



Rysunek 4: Czasy odpowiedzi na żądania typu POST.



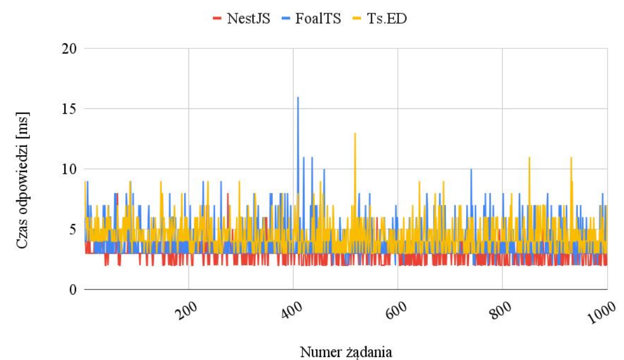
Rysunek 5: Czasy odpowiedzi na żądania typu POST.

Kolejnym badaniem było badanie czasu odpowiedzi na żądania typu PUT. Zapytanie składało się z identyfikatora istniejącego obiektu oraz nowych wartości dla obiektu, który miał zostać zaktualizowany. Czas był zmierzony od momentu wysłania żądania do momentu zwrócenia odpowiedzi przez serwer. Operacja została wykonana w serii 1000 powtórzeń aktualizujących całą tablicę wcześniej wczytanych obiektów. Podobnie jak w poprzednich porównaniach otrzymano wyraźnie odstający wynik w pierwszym pomiarze, przez co został on pominięty przy prezentacji i obliczeniach.

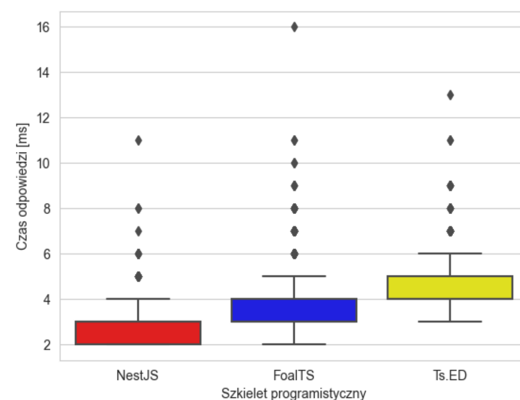
Wyniki zostały zaprezentowane w Tabeli 4 oraz na Rysunkach 6 i 7. Po przeanalizowaniu wyników można zauważyć, że najlepiej wypadł *NestJS*. Uzyskał on zarówno najszybszy średni czas oraz jego wyniki były najbardziej stabilne. Drugie miejsce zajął *FoalTS* ze średnim czasem gorszym o niecałą milisekundę. Najwolniejszy po raz kolejny był *Ts.ED,* uzyskując najsłabszy średni czas.

Tabela 4: Czasy odpowiedzi na żądania typu PUT

Szkielet	Czas min. [ms]	Czas maks. [ms]	Czas śr. [ms]	Odchylenie std. [ms]
NestJS	2	11	2,971	0,834
FoalTS	2	16	3,862	1,466
Ts.ED	3	13	4,425	1,260



Rysunek 6: Czasy odpowiedzi na żądania typu PUT.



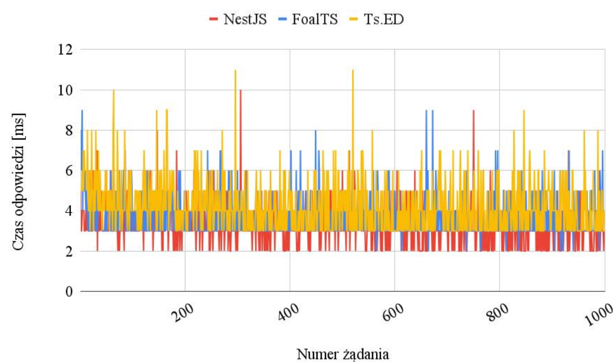
Rysunek 7: Czasy odpowiedzi na żądania typu PUT.

Ostatnim z porównań było porównanie czasów odpowiedzi na żądania typu DELETE. Pomiar czasu był wykonywany od momentu wysłania zapytania z identyfikatorem obiektu do usunięcia do momentu zwrócenia odpowiedzi przez serwer. Seria badań zaczynała się od usunięcia obiektu pierwszego aż do ostatniego, czyli tysięcznego. Po przeprowadzeniu 1000 powtórzeń pozostawała pusta tablica i badanie kończyło się. Odstający pierwszy pomiar został pominięty

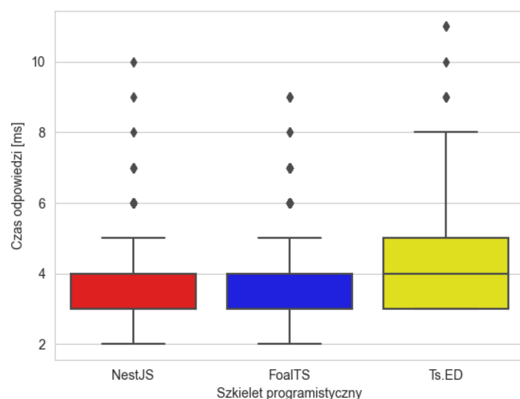
w analizach. Uzyskane wyniki zostały przedstawione w Tabeli 5 i na Rysunkach 8 i 9. Po przeanalizowaniu wyników można określić *NestJS* jako najlepszy szkielet w zestawieniu. Uzyskał on najszybszy czas oraz wykazał się najlepszą stabilnością czasu potrzebnego na obsłużenie żądania. *FoalTS* uzyskał czas nieco dłuższy. Przeprowadzone testy ANOVA i Tukeya pozwoliły na stwierdzenie, że były istotne różnice dla czasów uzyskanych przez wszystkie pary porównywanych szkieletów.

Tabela 5: Czasy odpowiedzi na żądania typu DELETE

Szkielet	Czas min. [ms]	Czas maks. [ms]	Czas śr. [ms]	Odchylenie std. [ms]
NestJS	2	10	3,236	0,999
FoalTS	2	9	3,687	1,018
Ts.ED	3	11	4,187	1,248



Rysunek 8: Czasy odpowiedzi na żądania typu DELETE.



Rysunek 9: Czasy odpowiedzi na żądania typu DELETE.

Tabela 6 zawiera podsumowanie uzyskanych rezultatów dla wszystkich przeprowadzonych badań. Wyniki dla poszczególnych szkieletów przedstawione są w kolejnych wierszach. Do przyznawania punktów zastosowano następujące kryteria:

- 0 pkt. – trzecie miejsce,
- 0,5 pkt. – drugie miejsce,
- 1 pkt. – pierwsze miejsce.

Powyższe kryteria oceniania były stosowane w momencie, kiedy występowały różnice istotne statystycznie. W przypadku, kiedy pomiędzy wszystkimi szkieletami nie występowały różnice, przyznawane było 0 punktów. W sytuacji, kiedy pomiędzy dwoma z trzech nie było istotnych statystycznie różnic, otrzymywały one punktację odpowiadającą uzyskanemu przez nie miejscu.

Tabela 6: Zestawienie wyników uzyskanych w badaniach

Badanie	NestJS	FoalTS	Ts.ED
Czas odpowiedzi żądanie typu GET, 1 obiekt	1	0,5	0,5
Czas odpowiedzi żądanie typu GET, 100 obiektów	1	0,5	0
Czas odpowiedzi żądanie typu GET, 500 obiektów	1	0,5	0
Czas odpowiedzi żądanie typu GET, 1000 obiektów	1	0,5	0
Czas odpowiedzi żądanie typu GET, 5000 obiektów	1	1	0,5
Czas odpowiedzi żądanie typu GET, 10000 obiektów	1	1	0,5
Czas odpowiedzi żądanie typu POST	1	0,5	0
Czas odpowiedzi żądanie typu PUT	1	0,5	0
Czas odpowiedzi żądanie typu DELETE	1	0,5	0
Suma	9	5,5	1,5

6. Wnioski

Celem tego artykuły było porównanie pod względem wydajności szkieletów programistycznych wykorzystywanych do budowy aplikacji serwerowych przy pomocy języka programowania TypeScript.

Najważniejszym czynnikiem wziętym pod uwagę w badaniach był czas odpowiedzi na żądania, ponieważ na jego podstawie była określana wydajność aplikacji. Odwzorowanie naturalnych warunków pracy, w których wykorzystywane są porównywane technologie zostało uwzględnione w stworzonych scenariuszach badawczych. Przeprowadzone testy ANOVA i Tukeya wykazały istotne statystycznie różnice w porównywanych czasach uzyskanych przez szkielety.

Na podstawie uzyskanych wyników można stwierdzić jednoznacznie, że wielkość przesyłanych danych ma wpływ na czas odpowiedzi. Najbardziej wyraźnie widać to w przypadku obsługi żądań typu GET. Jednocześnie, wraz ze wzrostem liczby zwracanych obiektów, zwiększały się różnice pomiędzy porównywanymi szkieletami. W pomiarach opartych o żądania typu GET to NestJS wypadł najlepiej uzyskując najkrótsze czasy obsługi. Najwolniejszy dla każdego rozmiaru danych był Ts.ED i różnica względem pozostałych konkurentów w kolejnych porównaniach wzrastała na niekorzyść tego szkieletu. Co ciekawe, FoalTS przy większym rozmiarze danych pomniejszał rozbieżność względem NestJS i przy zwracanych 5000 i 10000 obiektów różnice statystyczne nie występowały. W przypadku żądań typu POST, PUT

i DELETE to NestJS uzyskał najlepsze wyniki zajmując pierwsze, a drugie i trzecie miejsce zajęły kolejno FoalTS i Ts.ED.

Po przeprowadzeniu analiz można było stwierdzić, że najbardziej wydajnym szkieletem spośród wszystkich porównywanych był NestJS, a FoalTS i Ts.ED zajęły odpowiednio drugie i trzecie miejsce.

Literatura

[1] NestJS - szkielet programistyczny dla node.js, https://nestjs.com/, [26.11.2021].

[2] FoalTS - szkielet programistyczny dla node.js, https://foalts.org/, [26.11.2021].

[3] Ts.ED – szkielet programistyczny dla node.js, https://tsed.io/, [26.11.2021].

[4] Node.js - oficjalna strona, https://nodejs.org/, [26.11.2021].

[5] Postman - platforma do pracy z API, https://www.postman.com/, [26.11.2021].

[6] N. Chhetri, A comparative analysis of node.js (serverside javascript) (praca magisterska), Culminating Projects in Computer Science and Information Technology 5 (2016).

[7] B. Miłosierny, M. Dzieńkowski, The comparative analysis of web applications frameworks in the Node.js ecosystem, Journal of Computer Sciences Institute 18, (2021) 42–48.

[8] K. Lei, Y. Ma, Z. Tan, Performance comparison and evaluation of web development technologies in PHP, Python, and Node.js, Proceedings of 17th international conference on computational science and engineering, IEEE (2014) 661-668.

[9] A. D. Pham, Developing back-end of a web application with NestJS framework: Case: Integrify Oy's student management system (praca licencjacka), (2020).

[10] C. Peters, Building Rich Internet Applications with Node.js and Express.js, Rich Internet Applications w/HTML and Javascript Feb 6, (2017) 15-20.

[11] Express - szkielet programistyczny dla node.js, https://expressjs.com/, [26.04.2021].

[12] M. Major, J. Niezgoda, Elementy Statystyki. Część I. Statystyka opisowa, Oficyna Wydawnicza AFM, Kraków, 2003.