JOURNAL of COMPUTER SCIENCES INSTITUTE

JCSI 18 (2021) 42–48
Received: 07 December 2020
Accepted: 20 December 2020

The comparative analysis of web applications frameworks in the Node.js ecosystem

Analiza porównawcza szkieletów do budowy aplikacji internetowych w ekosystemie Node.js

Bartosz Miłosierny*, Mariusz Dzieńkowski

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

Abstract

The subject of the research was the comparative analysis of three frameworks for building web applications, i.e. Express (version 4.17.1), Hapi (version 20.0.1) and Koa (version 2.13.0), operating in the Node.js ecosystem. An experiment was prepared consisting of a number of scenarios, during which the server response times to incoming requests from the client were measured. As part of the work, server test applications handling HTTP requests (GET, POST, PUT, DELETE) performing typical data operations were implemented. The applications contained the same functionalities and were built using the three tested frameworks. In individual scenarios, 1,000 requests of a given type were sent from independent clients, the times of successive responses were measured and their averages were calculated. On the basis of the obtained results, Hapi and Koa frameworks for GET, POST, PUT, DELETE requests, operating on one object or a string *Hello World!* have achieved the best, although very similar, response times. In the case of the GET request, the Koa framework proved to be the best for higher loads, achieving response times approximately 20% better than the Express framework. For high loads, the Hapi framework achieved the worst results, reaching response times over 2 times longer than the Koa framework.

Keywords: JavaScript frameworks; Node.js environment; performance comparative analysis; server applications

Streszczenie

Przedmiotem badań była analiza porównawcza trzech szkieletów do budowy aplikacji internetowych działających w ekosystemie Node.js: Express (wersja 4.17.1), Hapi (wersja 20.0.1) oraz Koa (wersja 2.13.0). Przygotowano eksperyment składający się z szeregu scenariuszy, podczas których dokonano pomiarów czasów odpowiedzi serwera na żądania przychodzące ze strony klienta. W ramach pracy zaimplementowano serwerowe aplikacje testowe obsługujące żądania HTTP (GET, POST, PUT, DELETE) realizujące typowe operacje na bazie danych. Aplikacje zawierały te same funkcjonalności i zostały zbudowane przy pomocy trzech testowanych szkieletów. W poszczególnych scenariuszach wysyłano od niezależnych klientów po 1000 żądań danego typu, dokonywano pomiarów czasów kolejnych odpowiedzi oraz obliczano ich średnie. Na podstawie uzyskanych wyników okazało się, że szkielety Hapi i Koa dla żądań typu GET, POST, PUT, DELETE, operujące na jednym obiekcie lub ciągu znaków *Hello World!* osiągnęły najlepsze, choć bardzo zbliżone czasy odpowiedzi. W przypadku żądania GET, przy większych obciążeniach zdecydowanie najlepszym okazał się szkielet Koa, uzyskując czasy odpowiedzi w przybliżeniu o 20% lepsze niż Express. Przy dużych obciążeniach zdecydowanie najgorzej wypadł szkielet Hapi osiągający ponad 2 razy dłuższe czasy odpowiedzi niż szkielet Koa.

Słowa kluczowe: szkielety programistyczne JavaScript; środowisko Node.js; analiza porównawcza wydajności; aplikacje serwerowe

*Corresponding author

Email address: bartosz.milosierny@pollub.edu.pl (B. Miłosierny)

1. Wstęp

Szybki rozwój technologii informatycznych, w tym w szczególności urządzeń mobilnych powoduje, że wiele firm przenosi prowadzoną działalność do internetu, dając w ten sposób klientom natychmiastowy i nieograniczony dostęp do swoich usług z różnych urządzeń zarówno stacjonarnych jak i mobilnych. Korzyści płynące z rozwoju usług prowadzonych z wykorzystaniem technologii informatycznych w świecie wirtualnym powodują, że aplikacje internetowe są obecnie najszybciej rozwijającą się klasą systemów oprogramowania. Brak konieczności tworzenia oddzielnych programów dostosowanych do danego rodzaju urządzenia i różnych systemów operacyjnych, a także uniknięcie problemu synchronizacji danych to tylko niektóre przykłady zalet oprogramowania działającego w internecie.

Aplikacje webowe to skomplikowane systemy składające się na ogół z dużej liczby komponentów opracowanych w różnych technologiach. Obecnie do ich budowy często stosuje się szkielety programistyczne, których używanie przyczynia się do zwiększenia wydajności programowania. Wynika to z tego, że szkielety wykorzystują wbudowane metody, których samodzielna

implementacja byłaby czasochłonna [1]. Szkielety programistyczne są na ogół udostępniane bezpłatnie. Ponadto ich zastosowanie obniża końcowe koszty wytworzenia aplikacji, ponieważ oprogramowanie powstaje szybciej i zawiera mniej błędów. Za używaniem szkieletów przemawiają także względy bezpieczeństwa. Aktywna społeczność wykorzystująca w swoich aplikacjach dany framework, natychmiast zgłasza napotkane problemy, które są niezwłocznie naprawiane.

Bogaty wachlarz technologii oferujących podobne możliwości sprawia, że już sam wybór szkieletu programistycznego jest dużym wyzwaniem. W tym celu przeprowadza się różnego typu badania, na specjalnie do tego celu przygotowanych aplikacjach testowych, budowanych na podstawie porównywanych frameworków. Wykorzystując do tego celu różne dodatkowe narzędzia, stwarzane są warunki, w których opracowywane oprogramowanie może się w przyszłości znaleźć.

Celem niniejszej pracy jest wykonanie analizy porównawczej szkieletów do budowy aplikacji webowych działających na platformie Node.js pod kątem wydajności czasowej oraz objętości kodu.

Do przeprowadzenia testów wykorzystano trzy identyczne pod względem funkcjonalności aplikacje serwerowe wysyłające żądania typu GET, POST, PUT, DELETE, do budowy których wykorzystano testowane szkielety programistyczne: Express (wersja 4.17.1), Hapi (wersja 20.0.1) oraz Koa (wersja 2.13.0). Aplikacje napisane zostały w języku JavaScript, który jest natywnie wspierany poprzez środowisko Node.js. Wykorzystano je do porównania wymienionych w tytule trzech szkieletów, biorąc pod uwagę czasy odpowiedzi serwera w reakcji na żądania przychodzące ze strony aplikacji klienckiej. Dodatkowo w analizie wzięto pod uwagę objętości fragmentów kodu odpowiedzialnych za realizację określonych funkcjonalności utworzonych aplikacji.

Analizowane szkielety oparte są na istniejącym od stosunkowo długiego czasu środowisku Node.js. Jednak w związku z zastosowaniem do budowy aplikacji języka JavaScript i rozwijanego przez firmę Google bardzo wydajnego silnika V8, Node.js wciąż zyskuje na popularności, a coraz więcej różnej wielkości firm decyduje się na przejście właśnie na tę platformę [2].

Do przeprowadzenia testów wydajnościowych wykorzystano zaimplementowane funkcje, obsługujące poszczególne rodzaje żądań, bibliotekę Perfy oraz zewnętrzne oprogramowanie o nazwie Postman, służące do testowania interfejsów komunikacyjnych aplikacji, poprzez wysyłanie do serwera żądań HTTP.

2. Przegląd literatury

W przypadku aplikacji internetowych, które mają wielowarstwową architekturę, są heterogeniczne i jednocześnie może z nich korzystać duża grupa użytkowników, bardzo ważną kwestią jest ich jakość i związana z nią wydajność. W związku z tym testowanie tego typu aplikacji jest procesem trudnym i pracochłonnym. W artykule [3], jego autorzy podkreślają wagę testowania aplikacji pod względem wydajności przy zastosowaniu różnych obciążeń uzyskiwanych poprzez zwiększanie liczby zapytań oraz poprzez wysyłanie różnej wielkości danych. Podstawową miarą wykorzystywaną w tego typu badaniach jest czas wykonania poszczególnych czynności i odpowiedzi serwera.

Magnus Greiff i André Johansson w swojej pracy licencjackiej [4] porównali szkielety programistyczne Symfony i Express przeznaczone do budowy aplikacji po stronie serwera. Badania przeprowadzano wysyłając 100, 1000, 10000 i 100000 zapytań do programów działających na serwerze, zbudowanych przy użyciu tych dwóch szkieletów, testując przy tym i porównując poprawność realizacji zapytań, czas potrzebny na ich wykonanie oraz poziom wykorzystania procesora. Wyniki testów wykazały, że aplikacja serwerowa zbudowana przy użyciu szkieletu programistycznego Express uzyskiwała krótsze czasy odpowiedzi, realizując przy tym poprawnie większą liczbę zapytań niż to było w przypadku programu zbudowanego na bazie Symfony. Aplikacja serwerowa oparta na szkielecie Express wykazała większe zużycie procesora niż aplikacja bazująca na Symfony, jednak podczas wykonania poszczególnych operacji potrzebowała ona mniej pamięci operacyjnej.

W artykule [5] autorzy Kai Lei, Yining Ma oraz Zhi Tan porównali i ocenili technologie służące do budowy aplikacji internetowych: PHP, Python i Node.js. Badania wykonano, przeprowadzając te same testy na aplikacjach zawierających takie same funkcjonalności, ale zbudowanych przy użyciu tych trzech popularnych dzisiaj technologii. Testy aplikacji: wyświetlającej tekst *Hello World!*, obliczających i zwracających dziesiąty, dwudziesty i trzydziesty wyraz ciągu Fibonacciego, jak i przeprowadzających operacje na bazie danych, w większości przypadków wykazały największą wydajność środowiska Node.js w stosunku do pozostałych, testowanych technologii.

Z kolei w artykule [6] porównano trzy technologie serwerowe Node.js, PHP/Apache oraz Nginx. Przeprowadzone badania pokazały, że aplikacja serwerowa utworzona na platformie Node.js charakteryzowała się większą wydajnością od aplikacji serwerowych Apache i Nginx. Wynikało to z tego, że Node.js lepiej wykorzystuje dostępne zasoby serwera, co przekładało się na obsługę większej liczby zapytań na sekundę. Jednak serwer Node.js okazał się być mniej wydajnym od serwera Nginx w operacjach na plikach statycznych, w których Nginx osiągał lepsze wyniki od niego i od serwera PHP/Apache.

3. Wykorzystane technologie i narzędzia

3.1. Node.js

Node.js jest wieloplatformowym środowiskiem uruchomieniowym o otwartym źródle, umożliwiającym wykonywanie kodu napisanego w języku JavaScript po stronie serwerowej. Aplikacje utworzone w środowisku Node.js, dzięki asynchronicznym operacjom wejścia/wyjścia, mogą pracować na pojedynczym procesie bez potrzeby tworzenia oddzielnych wątków dla każdego z żądań przychodzących do serwera. Środowisko, operując na architekturze zbudowanej na zdarzeniach,

uniemożliwia zatrzymywanie się kodu aplikacji podczas wczytywania danych, przechodząc do następnych instrukcji. Po ukończeniu pobierania danych, aplikacja wraca do pominiętego kodu oczekującego na wymagane dane [7].

3.2. REST

Representational State Transfer (REST) jest to styl architektury oprogramowania, wprowadzający określony schemat budowy serwisów internetowych i usług sieciowych. REST stał się wzorcem budowy aplikacji internetowych, który narzucił programistom określony sposób tworzenia programów serwerowych, ułatwiając równocześnie implementację części klienckich, które podążając za tym samym stylem, mogą w prosty sposób komunikować się z serwerem.

Architekturę REST można opisać za pomocą sześciu cech [8-10]:

- model klient-serwer – aplikacja kliencka jest odseparowana od serwerowej,
- bezstanowość – żadna informacja o stanie nie jest przechowywana przez serwer i musi być każdorazowo przesyłana przez klienta,
- przechowywanie danych w pamięci podręcznej – każde żądanie zawiera informację o tym, czy powinno być ono buforowane po stronie klienta,
- jednolity interfejs – każdy zasób powinien być dostępny przez unikalny dla niego adres URI, a odpowiedź na żądanie powinna zawierać informacje, które jasno definiują format danych, jakie określona odpowiedź zawiera,
- warstwowość - dane, jakie klient otrzymuje od serwera muszą być odseparowane od logiki systemu, która stoi za wygenerowaniem tych danych,
- kod na żądanie - serwer może udostępniać aplikacji klienckiej kod w postaci skryptów lub apletów do operowania na zasobach po stronie klienta.

3.3. Postman

Postman jest narzędziem służącym do testowania i rozwijania interfejsów programistycznych aplikacji (API), udostępnionym bezpłatnie do realizacji małych projektów przez grupy kilkuosobowe. Pozwala on na wielokrotne wysyłanie żądań HTTP do serwera, który następnie zwraca odpowiedzi wraz ze wszystkimi danymi i informacją o czasie, jaki upłynął od wysłania żądania do otrzymania odpowiedzi.

Aplikacja pozwala na zapisywanie żądań w celu ich późniejszego wykorzystania, a także użycia przez inne osoby pracujące wspólnie nad tym samym projektem [11].

Postman daje możliwość spreparowania wszystkich składników żądania według potrzeb użytkownika testującego daną aplikację. Mogą to być na przykład parametry załączane do adresu URL, nagłówek, ciało żądania, elementy autoryzacji w postaci różnych kluczy czy tokenów. Ponadto Postman działając w środowisku przeglądarki internetowej pozwala na przeglądanie danych cookie [12].

3.4. Biblioteka Perfy

Perfy jest bardzo prostym narzędziem, udostępnionym na darmowej licencji MIT, służącym do pomiaru z dokładnością do nanosekundy, w czasie rzeczywistym, wydajności wykonywania kodu aplikacji tworzonych w środowisku Node.js [13].

Użycie biblioteki Perfy polega na utworzeniu instancji wydajnościowej poprzez wywołanie metody *start* z parametrem będącym nazwą tej instancji, która jednocześnie rozpoczyna odmierzanie czasu. Zakończenie pomiaru czasu następuje w momencie wywołania metody *end*, również z parametrem będącym nazwą wcześniej utworzonej instancji. Metoda ta zwraca obiekt wraz z informacją zawierającą dokładny czas, który upłynął pomiędzy wywołaniami metod *start* i *end* [14].

4. Porównywane szkielety programistyczne

Platforma Node.js będąca środowiskiem uruchomieniowym języka JavaScript jest dojrzałą technologią, cieszącą się coraz większą popularnością wśród programistów odpowiedzialnych zarówno za część kliencką jak i serwerową. Obecnie istnieje bardzo wiele szkieletów programistycznych opartych na języku JavaScirpt, które usprawniają realizację typowych funkcjonalności aplikacji www. Do analizy, zrealizowanej w ramach tej pracy, wybrano trzy szkielety programistyczne działające w ekosystemie Node.js: Express, Hapi oraz Koa.

4.1. Express

Express.js jest szkieletem programistycznym służącym do budowy aplikacji internetowych oraz interfejsów programistycznych, który powstał jako projekt fundacji OpenJS Foundation [15]. Jest on jednym z najbardziej popularnych szkieletów do budowy interfejsów programowania aplikacji na platformie Node.js [16], a na jego bazie powstało wiele innych frameworków. Express z założenia jest bardzo minimalistyczny, co przyczynia się do zwiększenia wydajności tworzonych przy jego użyciu aplikacji, a wszystkie potrzebne funkcjonalności można dodawać za pomocą dostarczanych przez menedżera pakietów *npm* oraz wtyczek.

4.2. Hapi

Szkielet Hapi pierwotnie został stworzony do obsługi dużego ruchu sieciowego jednej z największej sieci supermarketów na świecie – Walmart. Powstał on z myślą o budowie dużych i elastycznych aplikacji za pomocą jak najmniejszej ilości kodu i nakładu pracy. Jako jeden z niewielu szkieletów, Hapi nie wykorzystuje funkcji pośredniczących zwanych „middleware". Za poświadczenia, autoryzację oraz walidację treści żądań przychodzących do serwera, odpowiadają wewnętrzne wbudowane funkcje oraz procedury zawarte w samym szkielecie [17]. Pomimo braku wsparcia dla funkcji pośredniczących, Hapi w pełni współpracuje z innymi wtyczkami oraz zewnętrznymi pakietami stworzonymi przez społeczność do pracy w Node.js.

4.3. Koa

Koa jest minimalistycznym szkieletem, który powstał z przeznaczeniem do tworzenia wydajnych aplikacji internetowych oraz interfejsów programistycznych. Aplikacja wykonana za pomocą tego szkieletu jest obiektem zawierającym tablicę funkcji pośredniczących, które na żądanie są składane i wywoływane w sposób podobny do stosu. Dzięki funkcjom pośredniczącym nie ma potrzeby używania wywołań zwrotnych [18]. Szkielet ten zapewnia prostą obsługę błędów oraz ułatwia zarządzanie kodem. Duża wydajność jest osiągana poprzez mały objętościowo kod, uruchamianie wielu zadań w sposób równoległy, używanie w kodzie asynchronicznych interfejsów API oraz kompresję gzip.

5. Metoda badań

5.1. Opis eksperymentu

Do przeprowadzenia analizy porównawczej wybranych szkieletów programistycznych służących do budowy aplikacji internetowych działających na platformie Node.js opracowano 5 scenariuszy badawczych, w ramach których wykonano:

1. pomiary czasów odpowiedzi na żądania GET,
2. pomiary czasów odpowiedzi na żądania POST,
3. pomiary czasów odpowiedzi na żądania PUT,
4. pomiary czasów odpowiedzi na żądania DELETE,
5. pomiar objętości fragmentów kodu.

Wyzwaniem dla każdej aplikacji jest możliwie najkrótszy czas, który upłynie od wysłania żądania, poprzez jego przetworzenie i otrzymanie odpowiedzi przez klienta. W ramach pracy opracowano eksperyment, podczas którego badano wydajność trzech aplikacji zawierających te same funkcjonalności i zbudowanych przy pomocy jednego z trzech testowanych szkieletów programistycznych. Zadaniem każdej aplikacji była odpowiedź na serię tysiąca żądań HTTP odpowiedniego typu (GET, POST, PUT, DELETE). Głównym parametrem branym pod uwagę do oceny wydajności danego szkieletu był średni czas cyklu życia żądania.

Na potrzeby przeprowadzenia testów przygotowany został zbiór danych w postaci obiektów JSON, składających się z losowych danych imitujących wpisy bazodanowe. Każdy taki wpis składał się z numeru identyfikacyjnego użytkownika, identyfikatora wpisu, tytułu oraz treści. Dla żądania GET, PUT oraz DELETE do pamięci komputera, na którym przeprowadzany był eksperyment, załadowano od 1 do 10000 obiektów pobranych wcześniej z serwisu JSONPlaceholder [19]. Czasy odpowiedzi serwera na żądania typu GET, POST, PUT oraz DELETE, zostały pozyskane dzięki wewnętrznej funkcji o nazwie „Runner”, wbudowanej w aplikację Postman oraz bibliotece „Perfy”, służącej do pomiarów czasu w programach tworzonych w środowisku Node.js.

5.2. Środowisko testowe

W tabeli 1 znajduje się charakterystyka środowiska testowego, wykorzystanego do realizacji badań. Zawiera ona parametry sprzętu oraz wersje oprogramowania.

Tabela 1: Parametry środowiska testowego.

Sprzęt	
Procesor	Intel(R) Core(TM) i7-3770K
Pamięć RAM	16 GB
Karta sieciowa	Atheros GbE LAN
System operacyjny	Ubuntu 20.04.1 LTS
Oprogramowanie	
Express	4.17.1
Hapi	20.0.1
Koa	2.13.0
Perfy	1.1.5
Postman	7.34.0

5.3. Scenariusz 1 - pomiar czasów odpowiedzi serwera na żądania GET

Czasy mierzono od momentu przyjścia żądania do serwera, wysłanego przez aplikację kliencką, do momentu zwrócenia klientowi odpowiedniej liczby obiektów razem z kodem „200” oznaczającym pomyślne wykonanie operacji. Zmierzone zostały czasy zwrócenia 1, 10, 100, 500, 1000, 5000 oraz 10000 obiektów jako pojedynczej odpowiedzi. Dodatkowo dokonano pomiaru czasu odpowiedzi bardzo prostej aplikacji, zwracającej ciąg znaków *Hello World!*.

Na listingu 1 zaprezentowano fragment kodu aplikacji testowej utworzonej za pomocą szkieletu Express, która po otrzymaniu żądania GET pod adresem *http://localhost/api/hello*, zwraca klientowi ciąg znaków o treści *Hello World!*.

Listing 1: Fragment kodu aplikacji testowej *Hello World!* zbudowanej za pomocą szkieletu programistycznego Express.

```
app.get('/api/hello', async (req, res) => {
    perfy.start( name: 'get-time');
    await res.send('Hello World!');
    const time = perfy.end( name: 'get-time').fullMilliseconds;
    await times.push(time);
});

app.listen(3000, () => console.log('Express listens on 3000...'));
```

W przypadku aplikacji zwracających obiekty JSON, za pośrednictwem funkcji *data_loader* klientowi zwracana jest odpowiednia liczba obiektów. Funkcjonalność tą, zrealizowaną za pomocą szkieletu Hapi, pokazano na listingu 2.

Listing 2: Fragment kodu aplikacji testowej obsługującej żądania GET zbudowanej za pomocą szkieletu Hapi.

```
(async () => {
    data = await dataLoader(/*ilość obiektów*/);
    await server.start();
    console.log('Hapi listens on 4000...')
})();
```

```
server.route({
    method: 'GET',
    path: '/api/posts',
    handler: (request, h) => {
        perfy.start( name: 'get-time');
        return data;
    }
})

server.events.on('response', async () => {
    const time = perfy.end( name: 'get-time').fullMilliseconds;
    await times.push(time);
});
```

5.4. Scenariusz 2 - pomiar czasów odpowiedzi serwera na żądania POST

W tym scenariuszu czas jest odmierzany od momentu otrzymania żądania zawierającego zagnieżdżony w swoim ciele obiekt. Następnie dodawany jest ten obiekt do pamięci, a aplikacji klienckiej zwracany jest kod „201” - oznaczający utworzenie przez serwer nowego zasobu. Po wykonaniu tych operacji następuje zakończenie pomiaru czasu i zapis wyniku.

Listing 3 przedstawia fragment kodu aplikacji testowej utworzonej za pomocą szkieletu Koa.

Listing 3: Fragment kodu aplikacji testowej obsługującej żądanie POST, zrealizowanej za pomocą szkieletu Koa.

```
app.use( fn: async (ctx, next) => {
    perfy.start( name: 'get-time');
    await next();
    ctx.res.on('finish', async () => {
        const time = perfy.end( name: 'get-time').fullMilliseconds;
        await times.push(time);
    });
});

router.post('/api/posts', async (ctx) => {
    await data.push(ctx.request.body.post);
    ctx.response.status = 201;
});
```

Po utworzeniu i zainicjowaniu zmiennych oraz przygotowaniu serwera do obsługi żądania POST, zawierającego obiekt JSON, wraz z uruchomieniem serwera, rozpoczyna się nasłuchiwanie na wskazanym porcie. Po otrzymaniu od klienta żądania POST zawierającego obiekt, rozpoczyna się odmierzanie czasu realizowane za pomocą funkcji *perfy*. Serwer umieszcza otrzymany obiekt w jednowymiarowej tablicy *data*, po czym wysyła klientowi kod „201”, oznaczający pomyślne utworzenie nowego zasobu. W tym momencie kończy się odmierzanie czasu, wynik zostaje zapisany do tablicy *times*, która następnie jest przekazywana do funkcji *logger*, a po zakończeniu pracy serwera zostaje zapisana w pliku tekstowym.

5.5. Scenariusz 3 - pomiar czasów odpowiedzi serwera na żądania PUT

W tym przypadku pomiar czasu rozpoczyna się w momencie przyjścia żądania do serwera, które w adresie URL zawiera numer modyfikowanego zasobu, a w swoim ciele przechowuje obiekt, przeznaczony do modyfikacji, która może być dokonana po stronie klienta. Po wykonaniu zmian na tym, przechowywanym w pamięci serwera obiekcie, aplikacja zwraca klientowi kod „204” - oznaczający zakończone sukcesem wykonanie operacji i nie dołącza żadnych dodatkowych danych zwrotnych. Potem następuje zaprzestanie mierzenia czasu i zapis wyniku.

Na listingu 4 przedstawiono fragment kodu aplikacji testowej utworzonej za pomocą szkieletu Express.

Listing 4: Fragment kodu aplikacji testowej obsługującej żądania PUT zbudowanej za pomocą szkieletu Express.

```
app.put('/api/posts/:id', async (req :Koa.Context , res) => {
    perfy.start( name: 'get-time');
    const id = req.params.id;
    data[id - 1] = await req.body.post;
    await res.sendStatus( statusCode: 204);
    const time = perfy.end( name: 'get-time').fullMilliseconds;
    await times.push(time);
```

Po inicjalizacji zmiennych, aplikacja serwerowa gotowa jest do obsługi żądania PUT. W momencie otrzymania od klienta żądania zawierającego numer przechowywanego w pamięci aplikacji obiektu, przeznaczonego do modyfikacji, rozpoczyna się odmierzanie czasu. W adresie *http://localhost/api/posts/id*, id jest identyfikatorem modyfikowanego obiektu. Następnie zachodzi podmiana wskazanego obiektu, obiektem zawartym w żądaniu przesłanym przez klienta. Po dokonaniu modyfikacji serwer zwraca kod „204”, który oznacza pomyślne wykonanie operacji, i w tym momencie kończy się odmierzanie czasu, który jest zapisywany w tablicy *times*. Po zakończeniu pracy serwera, ogłoszonym sygnałem *SIGINT*, funkcja *logger* zapisuje czasy odpowiedzi w pliku tekstowym.

5.6. Scenariusz 4 - pomiar czasów odpowiedzi serwera na żądania DELETE

W tym scenariuszu celem było zmierzenie czasu, jaki zajmuje aplikacji serwerowej obsługa żądania DELETE. Pomiar czasu rozpoczynał się w momencie przyjścia żądania zawierającego w adresie URL numer usuwanego zasobu. Po wykonaniu tej operacji i zwróceniu kodu „204”, kończył się pomiar czasu i następował jego zapis.

Listing 5 przedstawia kod aplikacji testowej utworzonej za pomocą szkieletu Hapi.

Listing 5: Fragment kodu aplikacji testowej obsługującej żądania DELETE zbudowanym za pomocą szkieletu Hapi.

```
server.route({
    method: 'DELETE',
    path: '/api/posts/{id}',
    handler: async (request, h) => {
        perfy.start( name: 'get-time');
        const id = request.params.id;
        await data.splice(id - 1, 1);
        const response = h.response();
        response.statusCode = 204;
        return response;
    }
})

server.events.on('response', async () => {
    const time = perfy.end( name: 'get-time').fullMilliseconds;
    await times.push(time);
});
```

Po utworzeniu i inicjalizacji zmiennych, przygotowywana jest odpowiedź serwera na żądanie DELETE. Po otrzymaniu takiego żądania, co następuje pod adresem *http://localhost/api/posts/id*, w którym *id* oznacza numer usuwanego obiektu, aplikacja rozpoczyna odmierzanie czasu. Następnie zostaje usunięty z pamięci wskazany przez użytkownika obiekt i w końcu aplikacja odpowiada klientowi kodem „204", który oznacza poprawne wykonanie operacji.

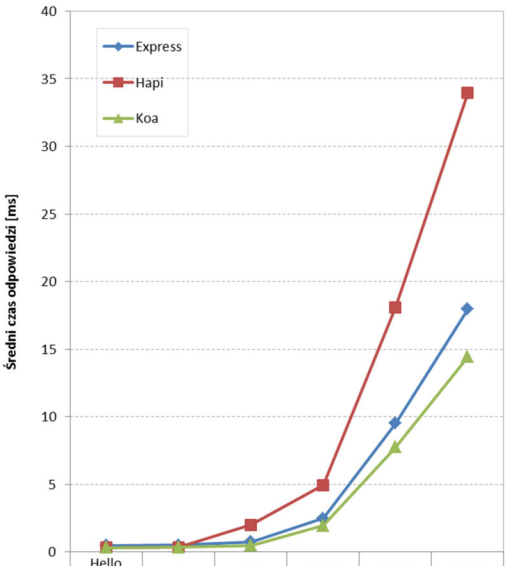
5.7. Scenariusz 5 - pomiar objętości kodu

Objętość kodu była dodatkowym wskaźnikiem, który obok wydajności, został wzięty pod uwagę podczas analizy porównawczej. Pomiar objętości był prostą czynnością polegającą na zliczeniu linii kodu, odpowiednich fragmentów stanowiących implementację danej funkcjonalności w danym szkielecie. Pustych wierszy lub wierszy zawierających wyłącznie komentarze nie uwzględniano w obliczeniach.

6. Wyniki badań

6.1. Analiza czasów odpowiedzi serwera dla żądania GET

Rysunek 1 przedstawia wyniki uzyskane po zrealizowaniu scenariusza badawczego nr 1, podczas którego wykonano pomiary czasów obsługi żądania GET przy różnych obciążeniach: 1, 100, 1000, 5000, 10000 obiektów oraz krótkiego tekstu *Hello World!*. Pomiary powtarzano 1000 razy dla każdego typu obciążenia. Następnie otrzymane wyniki uśredniono.



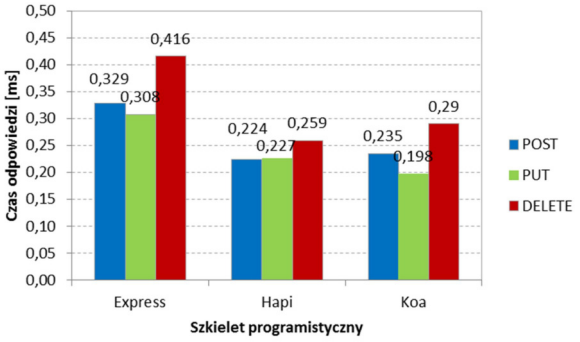
	Hello World!	1	100	1000	5000	10000
Express	0,474	0,503	0,728	2,488	9,513	17,954
Hapi	0,311	0,349	1,987	4,924	18,096	33,954
Koa	0,314	0,332	0,476	1,94	7,737	14,416

Rysunek 1: Wykres przedstawiający średnie czasy obsługi żądania GET przy różnych rodzajach obciążeń dla 3 szkieletów.

W przypadku aplikacji, która w odpowiedzi na żądanie zwracała tekst *Hello World!*, najkrótszy średni czas obsługi żądania GET miały programy zbudowane na bazie szkieletów Hapi oraz Koa. Również dla małych obciążeń, gdy w odpowiedzi zwracany był pojedynczy obiekt, czasy obsługi żądań były najmniejsze dla szkieletów Hapi i Koa. Dla obciążeń, gdy zwracanych było jednocześnie 100 obiektów, najkrótszy średni czas odpowiedzi uzyskał szkielet Koa. Na przygotowanie i wysłanie odpowiedzi szkielet Express potrzebował dwa razy dłuższego czasu, natomiast szkielet Hapi aż 5 razy dłuższego czasu niż Koa. W przypadku dużych obciążeń (1000, 5000 i 10000 zwracanych obiektów JSON) różnice w czasie obsługi żądania GET, między testowanymi szkieletami były jeszcze większe.

6.2. Analiza czasów odpowiedzi serwera dla żądań POST, PUT i DELETE

Na rysunku 2 zaprezentowano wyniki uzyskane po przeprowadzeniu badań według scenariuszy nr 2, 3 i 4. Zawierają one średnie czasy obsługi trzech typów żądań (POST, PUT i DELETE) realizowanych za pomocą aplikacji testowych opracowanych na podstawie trzech testowanych szkieletów.



Rysunek 2: Wykres przedstawiający średnie czasy obsługi żądań POST, PUT i DELETE przy różnych obciążeniach dla 3 szkieletów.

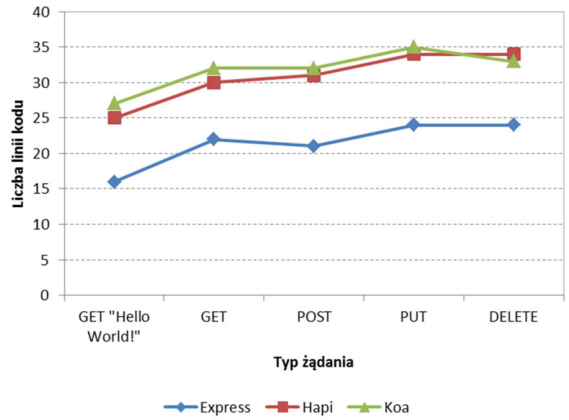
Z wykresu przedstawionego na rysunku 2 można wysnuć następujące wnioski:

1. Dla żądania POST najkrótszy średni czas odpowiedzi miał szkielet Hapi, niewiele dłuższy Koa, a najdłuższy Express.
2. Dla żądania PUT, najkrótszy średni czas odpowiedzi miał szkielet Koa, nieznacznie dłuższy Hapi i najdłuższy, również w tym przypadku szkielet Express.
3. Dla żądania DELETE, najkrótszy średni czas uzyskał szkielet Hapi, trochę dłuższy Koa i znacznie dłuższy Express.

6.3. Analiza objętości kodów

Rysunek 3 obrazuje wyniki pomiarów objętości kodu (wyrażonej w liczbie linii) zrealizowanych wg scenariusza nr 5. Biorąc pod uwagę liczbę linii kodu, fragmentów programu obsługujących poszczególne rodzaje żądań, zdecydowanie najlepszym okazał się szkielet Express. Aplikacja testowa wykonana na podstawie tej platformy programistycznej, dla każdej operacji powiązanej z określonym żądaniem, miała najmniejszą objętość. W przypadku Hapi oraz Koa wyniki były zbliżone, choć znacznie gorsze w porównaniu do szkieletu Express. Większa ilość wierszy kodu w przypadku tych dwóch szkieletów wynikała po pierwsze z potrzeby dołączania dodatkowych bibliotek odpowiedzialnych na

przykład za dodanie możliwości dostępu do ciała żądania, oraz po drugie ze specyfiki implementacji funkcji obsługujących żądania w poszczególnych szkieletach.



Rysunek 3: Wykres prezentujący objętość kodu odpowiedzialnego za obsługę poszczególnych typów żądań przez aplikacje wykonane na podstawie danego szkieletu.

7. Wnioski

Przed przystąpieniem do budowy aplikacji, wiedząc jakie będzie jej przeznaczenie, przewidując w jakich warunkach obciążeniowych będzie ona funkcjonowała oraz jakie będą jej stawiane wymagania, należy w sposób przemyślany, oparty na wynikach badań lub na wynikach badań innych ekspertów, dokonać optymalnego wyboru technologii. Praca ta powstała z myślą, aby wspomóc programistów i ułatwić im podjęcie decyzji dotyczącej wyboru szkieletu programistycznego opartego na języku JavaScript i funkcjonującego w ekosystemie Node.js.

Eksperyment, przeprowadzony na podstawie czterech scenariuszy, dał wyniki pozwalające na wykonanie analiz ilościowych. W porównaniach wzięto pod uwagę dwie miary: średni czas obsługi czterech typów żądań oraz objętość kodu wyrażoną w liczbie linii. Porównanie szkieletów było możliwe po utworzeniu aplikacji testowych, z których każda, choć posiadała dokładnie te same funkcjonalności, zbudowana została za pomocą innego szkieletu.

Analiza czasu obsługi żądania GET była rozszerzona, gdyż uwzględniała różne rodzaje i wielkości obciążeń. Zatem odpowiedzi generowane przez serwer miały różne wielkości. Dla żądań POST, PUT i DELETE odpowiedzi miały postać kodu liczbowego, informującego o zakończonym sukcesem wykonaniem operacji.

Na podstawie otrzymanych wyników można sformułować dwa generalne wnioski:

1. W przypadku żądania GET, najkrótsze czasy obsługi żądań uzyskiwała aplikacja zbudowana za pomocą szkieletu Koa. Nieco gorszą wydajność osiągała aplikacja oparta na szkielecie Express.
2. Dla żądań POST, PUT i DELETE najlepsze czasy miały Koa i Hapi. Wyniki aplikacji opartej na szkielecie Express były o około 30% gorsze.

W ramach pracy wykonano również analizę objętości kodów odpowiedzialnych za obsługę poszczególnego typu żądania, aplikacji testowych zbudowanych za pomocą różnych szkieletów. Wyniki jednoznacznie pokazały, że najkrótszy kod miała aplikacja zbudowana na bazie platformy programistycznej Express.

Literatura

[1] A. Wójcik, M. Wolski, J. B. Smołka, Performance analysis of the Symfony framework for creating modern web application based on selected versions, Journal of Computer Sciences Institute 13 (2019) 293-297.

[2] Global companies using Node.js in production, https://www.tothenew.com/blog/how-are-10-global-companies-using-node-js-in-production/, [04.11.2020]

[3] G. A. Di Lucca, A. R. Fasolino, Testing web-based applications: The state of the art and future trends, Information and Software Technology 48 (2006) 1172-1186.

[4] M. Greiff, A. Johansson, Symfony vs Express: A Server-Side Framework Comparison, Dissertation 2019.

[5] K. Lei, Y. Ma, Z. Tan, Performance comparison and evaluation of web development technologies in PHP, Python and Node.js, IEEE Computer Society (2014) 661-668.

[6] I. K. Chaniotis, K. D. Kyriakou, N. D. Tselikas, Is Node.js a viable option for building modern web applications? A performance evaluation study, Computing 97 (10) (2015) 1023-1044.

[7] Rozpoczęcie pracy z Node.js, https://riptutorial.com/pl/node-js, [07.11.2020].

[8] R. T. Fielding, Architectural Styles and the Design of Network-based Software Architectures, Doctoral dissertation, University of California, 2000.

[9] Wstęp do REST API, https://devszczepaniak.pl/wstep-do-rest-api/, [04.11.2020].

[10] Mastering REST, https://medium.com/@ahmetozlu93/mastering-rest-architecture-rest-architecture-details-e47ec659f6bc, [04.11.2020].

[11] Postman, https://www.postman.com/, [04.11.2020].

[12] B. Mehta, RESTful Java Patterns and Best Practices, Packt Publishing, 2014.

[13] Perfy, https://www.npmjs.com/package/perfy, [04.11.2020].

[14] Dokumentacja Perfy, https://github.com/onury/perfy, [04.11.2020].

[15] Dokumentacja Express, https://expressjs.com/, [04.11.2020].

[16] Zestawienie szkieletów Node.js, https://www.simform.com/best-nodejs-frameworks/, [04.11.2020].

[17] Dokumentacja Hapi, https://hapi.dev/, [04.11.2020].

[18] Dokumentacja Koa, https://github.com/koajs/koa, [04.11.2020].

[19] JSONPlaceholder, https://jsonplaceholder.typicode.com/, [04.11.2020].