JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 35 (2025) 121–128
Received: 07 January 2025
Accepted: 28 January 2025

Performance comparison of development frameworks in selected environments in REST API architecture

Porównanie wydajności szkieletów programistycznych w wybranych środowiskach w architekturze REST API

Mateusz Szewczyk*, Maria Skublewska-Paszkowska

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

Abstract
This paper presents a performance comparison of five popular REST API frameworks: ASP.NET, Spring Boot, Express.js, Laravel and Django REST Framework. The analysis took into account response times, resource consumption, Docker image sizes and code complexity. ASP.NET showed the shortest response times and smallest images, Express.js stood out for its stable resource management, while Django and Laravel, although less efficient, featured by compact code. Spring Boot, on the other hand, showed similar, though slightly worse, efficiency compared to ASP.NET. The results underscore the importance of matching the platform to specific project requirements.

Keywords: REST API performance; comparison of frameworks; web applications architecture

Streszczenie
Artykuł przedstawia porównanie wydajności pięciu popularnych szkieletów programistycznych REST API: ASP.NET, Spring Boot, Express.js, Laravel i Django REST Framework. Analiza uwzględniała czasy odpowiedzi, zużycie zasobów, rozmiary obrazów Docker oraz złożoność kodu. ASP.NET wykazał najkrótsze czasy odpowiedzi i najmniejsze obrazy, Express.js wyróżniał się stabilnym zarządzaniem zasobami, a Django REST Framework i Laravel, choć mniej wydajne, charakteryzowały się zwartym kodem. Spring Boot natomiast wykazał zbliżoną, lecz nieco gorszą, wydajność w stosunku do ASP.NET. Wyniki podkreślają znaczenie dopasowania platformy do specyficznych wymagań projektu.

Słowa kluczowe: wydajność REST API; porównanie szkieletów programistycznych; architektura aplikacji internetowych

*Corresponding author
Email address: mateusz.szewczyk@pollub.edu.pl (M. Szewczyk)

1. Introduction

Nowadays, IT systems are increasingly being designed in a Service Oriented Architecture (SOA). This carries many advantages, such as the ability to build loosely coupled reusable components, which allows for more efficient management of business processes. However, SOA architecture also involves the problem of coherently linking the various application blocks together so that they form a single, cohesive whole [1].

As a result of the above problem, a variety of API (Application Programming Interfaces) protocols have been developed to integrate and connect multiple independent programs. The context of the most common client-server web applications should be mentioned a group of so-called WebAPIs, which can include SOAP, REST, gRPC, JSON-RPC and GraphQL protocols [2].

The oldest and the most popular of the WebAPIs are SOAP and REST. However, while SOAP may be better for certain applications due to the preservation of connection state and greater security of data transmission, the REST API works better for most solutions due to its greater performance flexibility and scalability [3].

The main aim of the paper is to compare the performance of popular programming frameworks in REST API architecture in selected environments. The concept of performance is defined as the interdependence of response times, computing power and memory usage, and source code size.

For the purpose of this study, the following research hypotheses were established:

- H1: Spring Boot and ASP.NET do not differ in performance - average scores of each metric do not differ by more than 15%.
- H2: Express.js is the fastest development framework in terms of response times for returning small portions of data (at most 100 records at the same time) compared to other technologies tested, such as ASP.NET, Spring Boot, Laravel, Django REST Framework.
- H3: Django REST Framework achieves the slowest response times when compared with the other examined platforms in the REST API architecture, that is, ASP.NET, Spring Boot, Express.js and Laravel.

2. Related works

REST architecture is extremely important in the operation of modern applications. This can be evidenced, for example, by continuous attempts to develop and improve REST API technology. Li and his co-authors in two publications [4, 5] research the creation of tools and methods to improve the scalability and flexibility of the architecture in question. Among other things, the researchers are directing their work towards software-defined networks

(SDNs), pointing out opportunities for REST development in cloud and distributed environments. Their results indicated that both proposed projects are flexible and scalable to the further development and upgrades. Similar attempts to improve the aforementioned architecture were also made by Haijun Gu et al. [6] describing a novel way of composing RESTful web services. The authors suggested replacing classical endpoints (endpoints) with natural speech expressions, which resulted in better search precision and relevance according to the keywords-based searching.

Imam Ahmad et al. [7] developed a RESTful application in PHP language for order processing and delivery companies demonstrating the superiority of REST technology over SOAP. Specifically, their findings highlighted that REST is more efficient, providing faster response times for requests. Also, Xianjun Chen's team [8] positively evaluated a test program written in PHP language using Laravel framework in the context of modern applications, demonstrating flexibility, scalability and functionality of REST architecture in that implementation.

The issue of testing RESTful API programs has also been widely studied. Adeel Ehsan et al. [9] performed a literature review system covering 16 items on this topic, concluding that the number of publications on testing RESTful APIs increased between 2014 and 2021. Amid Golmohammadi et al. [10], on the other hand, reviewed 92 publications related to automated testing of REST technologies, finding that the quantity of published papers on RESTful APIs testing has increased dramatically from 2017 to 2023.

The large number of constantly developing software frameworks in the REST architecture and the constantly emerging new solutions provide ample room for comparative studies between them. Such analyses are sorely needed, as they allow software developers to choose the technology that best fits their project.

Grzegorz Blinowski et al. [11] conducted a study comparing the performance and scalability of monolithic and microservices architectures using the example of applications written in Spring Boot and ASP.NET. While the authors focused more on the analysis of the architectures studied, the experiments conducted showed that the choice of program implementation technology has negligible significance in the performance of a cloud application. Additionally, researchers conducted that monolithic applications work better on single machine than microservices, and vertical scaling cost less than horizontal scaling in Azure cloud services.

Erdem Kemer and Ruya Samli [1] made the most extensive analysis of all the publications discussed in this chapter. They compared as many as five technologies: Go, C#, Java, Python and Node.js. They designed a separate minimalist application for each programming language avoiding the use of unnecessary external libraries. Load tests were performed on the finished applications using k6 and Locust tools. The tests included different numbers of virtual users. The results showed that Go outperformed other technologies. C# and Java had similar performance and Node.js was slightly worse. Python reached the worst handling of requests.

Marin Kaluža et al. [12] also made an extensive comparison of Laravel, Ruby on Rails, Django and Spring frameworks. Moreover, the researchers defined a wide range of research metrics aimed at analyzing each technology as thoroughly as possible. In addition, they were the only ones of all the publications analyzed to create a point system, which was used after converting the aforementioned metrics into a more comprehensible ranking list, with conclusion that Django and Spring are suitable for large, developing projects, in contrast to Laravel and Ruby on Rails, which are better for small, compact applications.

Table 1: Summary of selected publications

Results/conclusions	Ref.
Express.js is minimalistic and appropriate for less complex applications. NestJS is better for extensive and complicated projects.	[13]
Spring Boot has better performance and requires writing fewer files than Laravel, but Laravel require less code lines.	[14]
Koa framework has the best performance for GET HTTP requests. For other requests, Koa and Hapi were better than Express.js, which was 30% worse.	[15]
NestJS reached better average request handling times than .NET.	[16]
ASP.NET and Java EE are not very different in performance context, and the choice of libraries/algorithms may be more important than the platform itself.	[17]
Express.js may be a more suitable choice for high-performance applications due to its faster request handling times compared to Hapi.	[18]
Express.js has better performance than Spring Boot. Request handling times were up to 249% better for Express.js.	[19]
The .Net API excelled in PUT operations, processing 11.6% faster than Java, while Java outperformed .Net in GET operations, handling 80.36% more records. Java was also 58.1% better for DELETE operations, but .Net led in POST performance.	[20]
Core and Node.js had similar overall performance, with Core excelling under heavy workloads. In CRUD tests, Node.js was better at POST operations, while Core outperformed in GET operations. Both were comparable in PUT and DELETE operations.	[21]
For low load, NodeJS was significantly faster than Spring in most cases. On the other hand, in almost all high-load tests, Spring proved to be more efficient.	[22]

The other papers in this literature review made technology comparisons in fairly similar ways. Each of them developed small applications for each technology,

prepared relevant research scenarios and defined metrics that often overlapped between these publications. Moreover, it can be seen that many of the technologies studied are repeated between articles. The Table 1 and Table 2 present a list of the compared frameworks in the respective publications.

Table 2: Comparison of used frameworks and metrics in studies

Frameworks	Metrics	Ref.
NestJS, Express.js	Number of lines of code and files	[13]
	HTTP request processing time and performance	
	Quality and completeness of documentation	
	Popularity and availability of advice among the community	
Spring Boot, Laravel	Request processing time for different scenarios	[14]
	Size of the project on disk	
	Number of files in the project	
	Number of lines of code	
	Community support	
Express.js, Hapi, Koa	Response time for HTTP requests	[15]
	Number of lines of code	
.NET, NestJS	Number of authentication and authorization functionalities available	[16]
	Maximum number of users performing login request depending on hardware configuration and hashing algorithms used	
	Response times for different hashing algorithms	
ASP.NET Core, Java EE	Test execution time	[17]
	Percentage of success of execution of scenarios	
	Network response time for given scenarios	
	Time of execution of scenarios	
Express.js, Hapi	Query execution time	[18]
Spring Boot, Express.js	Query execution time	[19]
Java, .NET Framework	Response time to requests	[20]
	Number of lines of code	
	Execution time of the request	
	Amount of data processed per unit of time	
.NET Core, Node.js	Response time to requests	[21]
Spring Boot, Node.js	Response time to requests	[22]

3. Materials and methods

As a result of the literature analysis, it was decided to compare the technologies of ASP.NET [23], Spring Boot [24], Express.js [25], Laravel [26] and Django REST Framework (DRF) [27].

In order to benchmark the considered REST API technologies, applications based on each of the considered programming frameworks were designed and implemented. They are designed to perform the same functionalities with particular emphasis on minimizing the use of external libraries and using the most standard solutions.

Appropriate research scenarios were developed and the most meaningful metrics were selected. The research was then conducted according to the following scheme:

- load tests using the Postman tool,
- averaging the results,
- analysis of other research metrics.

Relevant research scenarios were developed and the most meaningful metrics were selected.

3.1. Study objects

Five applications written in some of the most popular programming frameworks were studied [28, 29]. As a result of the literature review and online sources, the technologies chosen were Express.js 4.19.2, Laravel 11.9, ASP.NET 8.0, Spring Boot 3.3.2 and the Django REST Framework 3.15.2.

All programs are implemented according to the same OpenAPI document describing the list of REST service endpoints [28]. Available API queries implement JSON Web Tokens (JWT) authentication and authorization, as well as simple CRUD operations (create, read, update, delete). The focus was on using native REST API tools available in the given development frameworks. Only necessary additional libraries and packages unavailable immediately after initialization of a new project in a given environment were installed or imported into the application.

In addition, all the tested applications use identical copies of the same MySQL database version 9.0. The database consists of four entities containing user credentials and fictitious student data, respectively.

All applications, after eventual compilation, were built as Docker images and then placed in the cloud service.

3.2. Database and dataset

The database consists of four tables. The “users” entity contains user access data for authorizing API requests. The “students” table contain basic data related to students, such as name, email address and date of birth. The “locations” and “pictures” entities contain, respectively, accurate information about their location including residential address and URL addresses to their pictures. The “students” entity is further connected to the “locations” and “pictures” tables by one-to-many relationships. The structure of the database is shown in the Figure 1.

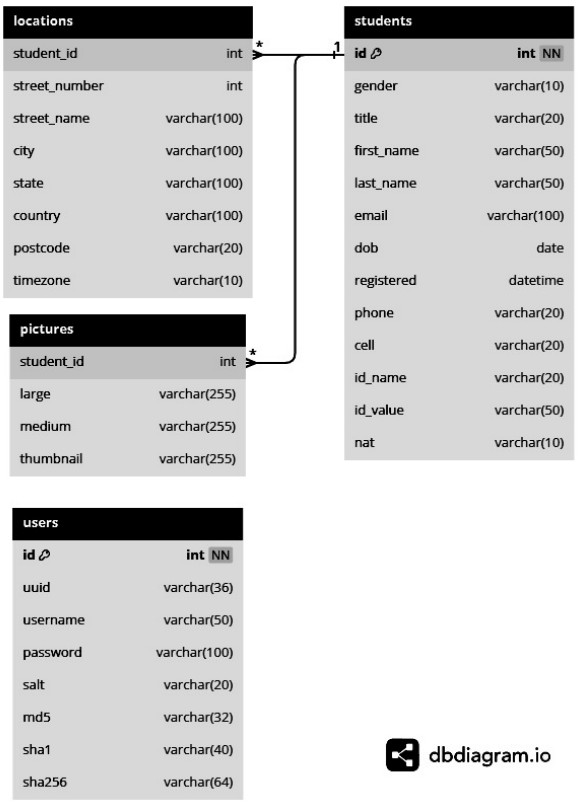

Figure 1: Structure of the database used in the research experiment.

The database was filled with fictitious data from the randomuser.me generator. Using the API provided by the service, the data of 1000 people was downloaded. Then, from the set so obtained, the data of 100 people were copied at random and used to populate the "users" table in the database. In turn, the data from the primary set was placed in the other entities corresponding to the students' data.

3.3. Research stand

The experiment was conducted on an MSI GL73 8RD laptop and the Microsoft Azure cloud platform. On the computer, the client in the form of the Postman tool version 11.1.14 was launched, while in the cloud, containers were launched based on the images of the various applications under study.

The laptop was equipped with an Intel Core i7-8750H processor (2.20 GHz), 16 GB RAM, NVIDIA GeForce GTX 1050 Ti graphics card and two hard drives – a GOODRAM 128 GB SSD containing the system partition and a WDC 1 TB HDD storing data. The machine was running Windows 10 Home 22H2 operating system.

Docker containers, as server part, were run in the Microsoft cloud using the Azure App Service in plan B2 (2 CPU cores, 3.5 GB RAM memory) with other settings retained with default values.

In order to increase the reliability of the test and to stabilize the Internet connection, a fiber-optic connection with a claimed capacity of 300 Mbps download and 50 Mbps upload was used. In addition, the Wi-Fi connection was abandoned, so the computer was connected to the Funbox 6 router directly via an Ethernet cable.

For the duration of the experiment, all unnecessary programs were disabled on the laptop and the downloading of system updates was blocked. In addition, all other devices were disconnected from the local computer network.

3.4. Research scenarios

Eleven research scenarios have been prepared to be run in performance testing mode. Requests should be performed alternately continuously for 10 minutes.

All developed scenarios are included in the Table 3.

Table 3: Research scenarios for benchmarking applications

Scenario	Description
1.	Displaying the static caption "Hello World"
2.	GET request retrieving 10 records
3.	GET request retrieving 100 records
4.	GET request retrieving 1000 records
5.	POST request adding 1 record
6.	POST request adding 10 records
7.	POST request adding 100 records
8.	PUT query updating 1 record
9.	PATCH request updating one field in 1 record
10.	PATCH request updating three fields in 1 record
11.	DELETE request deleting 1 record

3.5. Comparative metrics

The following metrics were used to benchmark the tested REST API technologies:

- the average response time of the server application for each of the test scenarios,
- the number of lines of code and the number of files that make up the source code of a given program,
- Docker image size of each application,
- average CPU load and RAM consumption for each scenario tested.

3.6. Description of the experiment

Prior to the experiment, a research stand was prepared and configured. Thanks to the use of Docker images, there was no need for advanced configuration of the cloud environment. Launching the applications was based solely on uploading the appropriate docker-compose file to the Azure service [30]. The experiment was then proceeded to run according to the developed scenarios.

In order to simulate virtual users, the "Runner – Performance test" tool available in Postman was used. It allows for continuous querying of specific API access points over a set period of time and generating rich reports on the performed tests.

Each of the test scenarios was performed for each of the applications for 10 minutes. The Listing 1 contains HTTP requests in the form of platform-independent pseudocode based on cURL.

Listing 1: All HTTP requests

```
# HELLO WORLD
curl -location '{{baseUrl}}/hello' \
--header 'Accept: text/plain'

# GET {{number}} students
curl -location
'{{baseUrl}}/student?results={{number}
}' \
--header 'Accept: application/json' \
--header 'Authorization: Bearer
{{jwtToken}}'

# POST students
curl -location '{{baseUrl}}/student' \
--header 'Content-Type:
application/json' \
--header 'Accept: application/json' \
--data-raw '[
     {
     "gender": "...", "title": "...",
     "first_name": "...",
     "last_name": "...",
     "email": "example@ex.com",
     "dob": "1900-01-01",
     "phone": "...", "id_name": "...",
     "id_value": "...", "nat": "ES",
     "location": {
          "street_number": 1234,
          "street_name": "...",
          "city": "...", "state": ...",
          "country": "...",
          "postcode": "...",
          "timezone": "-9:00"
     },
     "picture": {
          "large":
"https://example.com/img.jpg",
          "medium": "<url>",
          "thumbnail": "<url>"
       }
     },
     ...
]'

   # PUT student
   curl --location --request PUT
   '{{baseUrl}}/student/{{id}}' \
   --header 'Content-Type:
   application/json' \
   --header 'Accept: application/json' \
   --header 'Authorization: Bearer
   {{jwtToken}}' \
   --data-raw '{
        <the same student object structure
        as in the POST request>
   }'

   # PATCH student fields
   curl --location --request PATCH
   '{{baseUrl}}/student/{{id}}' \
   --header 'Content-Type:
   application/json' \
   --header 'Accept: application/json' \
   --header 'Authorization: Bearer
   {{jwtToken}}' \
   --data '{
        "<any_field>": "<value>",
        ...,
        "location": {
             ...
        },
        "picture": {
             ...
        }
   }'
  # DELETE student
  curl --location --globoff --request DELETE
  '{{baseUrl}}/student/{{studentid}}' \
  --header 'Authorization: Bearer
  {{jwtToken}}'
```

After each application was tested, REST API load test results were collected and compiled. In addition, resource consumption statistics were downloaded from the Azure service. The CLOC application available for installation via any package manager on Windows Linux or Mac OS was used to count lines of code and number of files. Finally, the remaining measurement metrics were determined and calculated.

4. Study results

The collected results of the conducted measurements according to the assumed scenarios and comparative metrics are presented below.

4.1. Applications response times to API requests

Measurements via the Postman tool resulted in average response times for individual API queries (Table 4) and the 90th percentiles of these queries (Table 5). In addition, average times for all queries together were calculated (Figure 2).

Table 4: Average response times for each scenario (where ASP. – ASP.NET, Lar. – Laravel, Exp. – Express.js, Spr. – Spring Boot)

Scenario no.	ASP. [ms]	Lar. [ms]	DRF [ms]	Exp. [ms]	Spr. [ms]
1.	34	357	44	35	39
2.	44	444	110	43	86
3.	61	491	654	66	218
4.	248	1074	6089	271	1563
5.	61	483	88	68	65
6.	110	577	268	264	121
7.	636	1950	2084	2118	656
8.	59	489	91	77	64
9.	54	478	71	62	59
10.	58	485	84	68	61
11.	42	431	50	41	47

Table 5: 90th percentiles of response times for each scenario

Scenario no.	ASP. [ms]	Lar. [ms]	DRF [ms]	Exp. [ms]	Spr. [ms]
1.	41	456	45	40	47
2.	54	500	127	53	129
3.	76	510	763	85	314
4.	364	1235	6588	345	1956
5.	77	529	118	92	87
6.	145	614	314	339	178
7.	875	2163	2300	2530	991
8.	74	514	124	104	81
9.	70	511	90	78	85
10.	75	530	127	88	86
11.	53	466	64	51	66

It can be seen that Laravel proved to be the slowest in most cases. Furthermore, in some cases it was as much as ten times slower than the fastest platform, which is ASP.NET.

It is also worth noting that the response time of the Django REST Framework for scenario 4 (downloading data of 1000 students) increased by leaps and bounds with respect to the response times of the other applications.

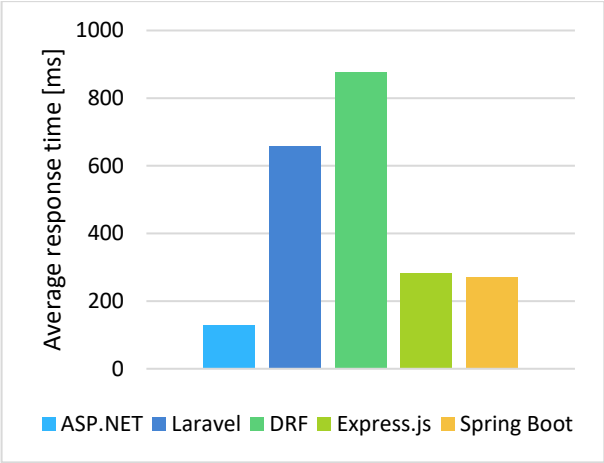

Figure 2: Average response times for all scenarios combined.

4.2. Number of lines of code and number of source code files

Using the CLOC tool, reports were generated and used to create the graphs in the Figures 3-4.

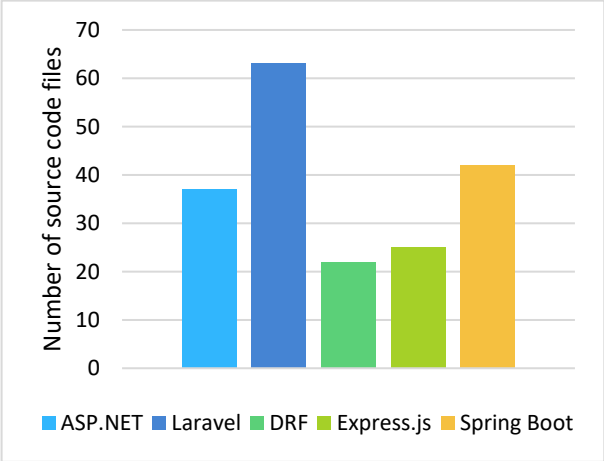

Figure 3: Number of source code files per application.

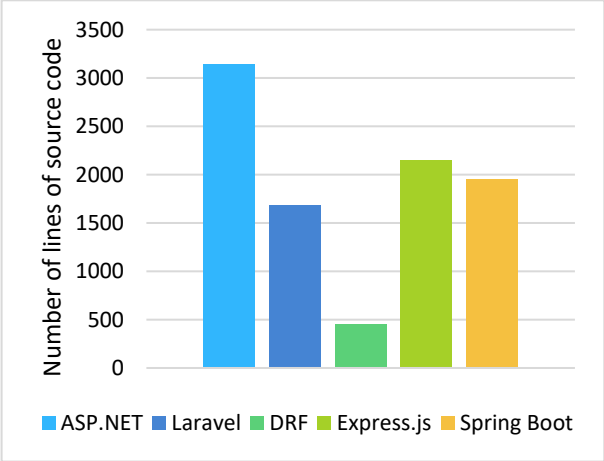

Figure 4: Number of lines of source code per application.

The most compact application in terms of source code volume turned out to be the one written using the Django REST Framework. On the other hand, it is possible to distinguish a program based on ASP.NET, whose code was the longest, and Laravel, where the project consisted of the largest number of files.

4.3. Docker image size

An assessment of the size of the Docker images for each application was carried out (Figure 5). Analyzing the resulting graph, it is possible to divide the studied applications into three groups. The smallest sizes were achieved by applications based on ASP.NET and Spring Boot. Laravel and Express.js applications turned out to be the largest, while Django REST Framework was placed in the middle.

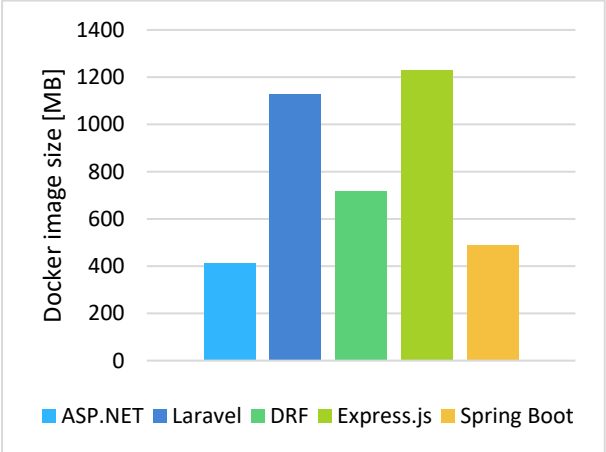

Figure 5: Sizes of particular Docker images.

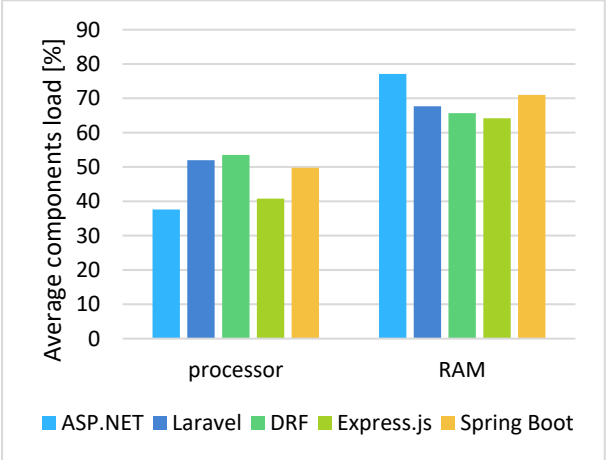

Figure 6: Average processor and RAM load per application.

4.4. Processor (CPU) load and RAM consumption

At the end of the study, statistics downloaded from the Azure platform were analyzed. Mean values and standard deviation were calculated for the data obtained (Figure 6, Table 6 and Table 7).

Table 6: Average processor and RAM load per application

Framework	Processor [%]	RAM [%]
ASP.NET	37.6	77.1
Laravel	52	67.7
DRF	53.5	65.7
Express.js	40.8	64.2
Spring Boot	49.8	71

Table 7: Standard deviation for average CPU and RAM load per application

Framework	Processor [%]	RAM [%]
ASP.NET	5.1	1
Laravel	5.3	0.6
DRF	3.6	0.5
Express.js	3.7	0.6
Spring Boot	7.6	0.4

5. Discussion

Analyzing the results of comparative studies of the aforementioned programming frameworks, it is possible to some correlations can be observed.

The results showed significant differences in response times (Figure 2 and Table 4-5). ASP.NET proved to be the most efficient in most cases, offering the lowest response times. It was especially visible while processing large numbers of records. Laravel and the Django REST Framework showed significant latency, especially in more demanding scenarios, such as retrieving 1000 records. The Django REST Framework reported furthermore an unusually long response time in this case, which may indicate the limitations of this technology in handling larger volumes of data. These results support hypothesis H3.

Performance disparities can also be seen in the consumption of system resources. ASP.NET, despite offering fast processing, put more strain on the CPU than other technologies.

In the context of system resources, a correlation can also be observed in that the least CPU-intensive applications used the most RAM and, conversely, the most RAM-saving programs put the heaviest load on the CPU. Such a phenomenon may be indicative of the different ways in which the various programming frameworks were implemented. Express.js stood out as a technology that provides equivalent CPU and RAM usage, suggesting that it may be the optimal choice for applications requiring stability and even resource distribution (Table 6-7).

The number of lines of code and source files is an important factor for development teams, as more compact code can contribute to easier software maintenance and development. The Django REST Framework was found to be the most compact in terms of number of code, while ASP.NET and Laravel required the highest number of files and/or lines of code. This discrepancy may indicate that ASP.NET and Laravel technologies are more complex and potentially more difficult to maintain, but may provide greater flexibility in terms of extending functionality (Figure 3-4).

Another important performance indicator is the size of the Docker image, which affects application deployment time and server resource consumption. ASP.NET and Spring Boot proved to be the most efficient in this respect, while Laravel and Express.js, on the other hand, generated the largest Docker images. The smaller image sizes suggest that ASP.NET and Spring Boot are more optimal for cloud environments where resource minimization is key (Figure 5).

After analyzing all metrics, it can be concluded that hypothesis H1 has been partially confirmed, as the assumption of a difference in rates of no more than 15% has been met for several of them. Most of the response time measurements for the individual scenarios are within the assumed limits. In addition, only the average RAM load and the number of lines of code and number of files are consistent with the hypothesis, the rest of the metrics contradict it.

In the case of hypothesis H2, it was rejected because for scenarios 1-3. the results of Express.js and ASP.NET were very even and only in second case Express.js was slightly faster (Figure 2 and Table 4-5).

The results obtained are similar to those in the literature. The response times, especially for Python application, coincide with the conclusions of E. Kemer and R. Samali [1]. The inferior optimization of ASP.NET in terms of RAM usage compared to Java applications was also revealed, as was done by K. Kornis and M. Uhanova [17]. They also confirmed the conclusions of A. Poniszewska-Marańska et al. who found that the NodeJS platform is significantly faster than Spring for low load, but Spring is more efficient for high load [22].

6. Conclusions and future works

In this paper, the performance of five popular development frameworks (ASP.NET, Spring Boot, Express.js, Laravel, and Django REST Framework) was compared in the context of REST API architecture. API query response times, system resource usage, Docker image sizes and source code complexity were analyzed. The results showed that ASP.NET was the most efficient, while Express.js stood out for its balanced use of resources. Django REST Framework and Laravel offered a compact code structure, but were less efficient in handling large workloads. Spring Boot, on the other hand, stood out for its relatively small Docker image size and relatively short response times. The obtained results show how many criteria should be considered in order to choose the most optimal platform for a given project.

The research was limited to five platforms and conducted in specific hardware and cloud environments. Results may vary depending on test conditions, such as different server configurations, types of workloads or use of alternative databases. Additionally, security and scalability aspects of the technology were not considered.

Future works may include comparison of additional development frameworks, such as NestJS or Ruby on Rails, and testing in different cloud environments (for example AWS or Google Cloud). In addition, another database systems may be used as well as other containerization tools, such as Podman. It should be also consider analyzing scalability under dynamic user growth and comparing the effectiveness of platforms in microservices applications. In addition, it can be measured with other tools, such as Grafana k6.

References

[1] E. Kemer, R. Samali, Performance comparison of scalable rest application programming, Computer Standards & Interfaces 66 (2019) 103355-103369.

[2] What Is an API (Application Programming Interface)? Meaning, Working, Types, Protocols, and Examples, https://www.spiceworks.com/tech/devops/articles/application-programming-interface/, [14 06 2024].

[3] H. Subramanian, P. Raj, Hands-On RESTful API Design Patterns and Best Practices: Design, develop, and deploy highly adaptable, scalable, and secure RESTful web APIs, Packt Publishing Ltd., Birmingham, 2019.

[4] L. Li, T. Tang, W. Chou, A XML Based Monadic Framework for REST Service Compositions, In IEEE International Conference on Web Services (2015) 487-494.

[5] L. Li, W. Chou, W. Zhou, M. Luo, Design Patterns and Extensibility of REST API, In IEEE Transactions on Network and Service Management 13(1) (2016) 154-167.

[6] H. Gu, Y. Ma, S. Wang, X. Chen, W. Su, Semantically realizing discovery and composition, Computing 1 (2024) 1-27.

[7] I. Ahmad, et al., Implementation of RESTful API Web Services, In 1st International Conference on Electronic and Electrical Engineering and Intelligent System (2021) 132-137.

[8] X. Chen, Z. Ji, Y. Fan, Y. Zhan, Restful API Architecture Based on Laravel Framework, In Journal of Physics: Conference Series 910(1) (2017) 12016-12022.

[9] A. Ehsan, M. A. M. E. Abuhaliqa, C. Catal, D. Mishra, RESTful API Testing Methodologies: Rationale, Challenges, and Solution Directions, Applied Sciences 12(9) (2022) 4369-4385.

[10] A. Golmohammadi, M. Zhang, A. Arcuri, Testing RESTful APIs: A Survey, ACM Transactions on Software Engineering and Methodology 33(1) (2023) 1-41.

[11] G. Blinowski, A. Ojdowska, A. Przybyłek, Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation, IEEE Access 10 (2022) 20357-20374.

[12] M. Kaluža, M. Kalanj, B. Vukelić, A comparison of back-end frameworks for web application development, Journal of the Polytechnic of Rijeka 7(1) (2019) 317-332.

[13] B. Zima, Comparative analysis of NodeJs frameworks, Master thesis, Lublin University of Technology, Lublin, 2024.

[14] J. Suchanowski, M. Plechawska-Wójcik, Performance analysis of web applications created in the Spring and Laravel frameworks, Journal of Computer Sciences Institute 29 (2023) 304-311.

[15] 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.

[16] P. S. Rodzik, Comparative Analysis of the Net 6 and NestJS Programming Frameworks in Terms of their Suitability for User Authentication and Authorization, Journal of Computer Sciences Institute 27 (2023) 104-111.

[17] K. Kronis, M. Uhanova, Performance Comparison of Java EE and ASP.NET, Applied Computer Systems 23(1) (2018) 37-44.

[18] M. Grudniak, REST API performance comparison of web applications based on JavaScript programming frameworks, Master thesis, Lublin University of Technology, Lublin, 2021.

[19] M. Wicha, Performance analysis of REST API technologies using Spring and Express.js examples, Master thesis, Lublin University of Technology, Lublin, 2023.

[20] K. Munonye, P. Martinek, Performance Analysis of the Microsoft .Net- and Java-Based Implementation of REST Web Services, In IEEE 16th International Symposium on Intelligent Systems and Informatics (2018) 191-196.

[21] E. Shkodra, E. Jajaga, M. Shala, Development and Performance Analysis of RESTful APIs in Core and Node.js using MongoDB Database, Proceedings of the 17th International Conference on Web Information Systems and Technologies WEBIST (2021) 227-234.

[22] A. Poniszewska-Marańda, K. Stepień, M. Głowiński, Function Analysis of Web Services Based on REST Protocol with Selected Frameworks, In International Conference on Software, Telecommunications and Computer Networks (SoftCOM) (2021) 1-6.

[23] ASP.NET technical documentation, https://dotnet.microsoft.com/en-us/apps/aspnet, [16.10.2024].

[24] Spring Boot technical documentation, https://spring.io/projects/spring-boot, [16.10.2024].

[25] Express.js technical documentation, https://expressjs.com/, [16.10.2024].

[26] Laravel documentation, https://laravel.com, [16.10.2024].

[27] Django REST Framework technical documentation, https://www.django-rest-framework.org/, [16.10.2024].

[28] Top 100 Development Frameworks, https://www.bairesdev.com/blog/top-development-frameworks/, [07.06.2024].

[29] A Comparison of 10 Popular Options for Building RESTful APIs, https://www.linkedin.com/pulse/choosing-right-restful-api-framework-comparison-10-popular-vanam-bbmue/, [07.06.2024].

[30] Source code and OpenAPI specification, https://github.com/MAtt5816/api-servers-comparison, [18.10.2024].