JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 39 (2026) 176-182
Received: 16 April 2026
Accepted: 10 May 2026

Comparative Performance Analysis of Express.js and Spring Boot in CRUD-Oriented Web Applications

Wojciech Wnuk*, Małgorzata Plechawska-Wójcik

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

Abstract

This study presents a comparative performance analysis of Express.js and Spring Boot in CRUD-oriented web applications. Three independent benchmark runs were executed for both frameworks using identical hardware, request patterns and load profiles. The results indicate that Express.js achieved lower global latency (avg ~10.5 ms vs. ~14.6 ms) and comparable throughput with slightly higher stability. CRUD-level measurements showed lower latency values for Express.js across create, read and update operations, with the most notable differences observed for delete requests. System-level metrics indicate lower CPU and memory consumption for Express.js. The findings are consistent with the characteristics of lightweight, event-driven architectures compared to JVM-based solutions in high-frequency CRUD workloads.

Keywords: web frameworks performance comparison; benchmarking; global latency; CRUD workloads; Express.js; Spring Boot

*Corresponding author

Email address: wojciech.wnukk@gmail.com (W. Wnuk)

Published under Creative Common License (CC BY 4.0 Int.)

1. Introduction

Web application performance remains a critical research topic due to the increasing demand for high-responsiveness systems operating under intensive request loads [1, 2]. Modern frameworks differ not only in architectural principles, but also in execution models, runtime environments and resource utilisation patterns. Among the most widely adopted technologies, Express.js represents a lightweight, event-driven JavaScript framework running on Node.js [3, 4], while Spring Boot delivers a comprehensive, JVM-based ecosystem for large-scale enterprise applications [5, 6].

Although both frameworks are commonly used to implement RESTful services, their performance characteristics under CRUD-oriented workloads are not widely addressed in the literature. Existing studies typically focus on theoretical comparisons or evaluate single metrics without analysing stability across multiple runs or observing system-level behaviour under load [7]. Meanwhile, modern applications frequently rely on high-volume interactions with databases [8], making precise measurements of latency, throughput and resource consumption essential for selecting an appropriate technology stack.

Previous studies have investigated the performance of REST APIs and backend frameworks under controlled workloads. Wicha and Pańczyk reported measurable differences between Spring- and Express-based REST API implementations in terms of response time and resource utilisation [9]. Szewczyk and Skublewska-Paszkowska compared five REST API frameworks and showed that Express.js stood out for stable resource management, while Spring Boot achieved efficiency close to the leading solutions, although slightly below ASP.NET [10]. Choma et al. analysed Express, Django and Spring Boot under increasing load and concluded that Spring Boot provided the best request processing time and high reliability, but under extreme load it also produced a higher percentage of incorrectly processed requests than Express [11]. Shkodra et al. showed that the relative advantage of Node.js- and .NET-based REST APIs depends strongly on workload profile and load intensity, with .NET Core outperforming Node.js mainly in large-load scenarios [12]. These studies indicate that framework performance is highly sensitive to experimental setup and therefore should be assessed using controlled and statistically rigorous benchmarking procedures [7, 13].

The objective of this study is to provide a quantitative comparison of Express.js and Spring Boot using a unified benchmarking procedure. Three independent test runs were executed for each framework, capturing global latency metrics (average, p95, maximum), CRUD-specific measurements, request throughput and system-level utilisation (CPU, memory). The experiments were conducted in a controlled environment to improve comparability, and the results were aggregated to evaluate both absolute performance and run-to-run stability. The findings contribute to the understanding of performance characteristics of event-driven and JVM-based architectures in latency-sensitive and high-frequency CRUD scenaios.

2. Methodology

2.1. Test environment

All experiments were conducted on a single physical machine to maintain consistent conditions across all benchmark runs. The hardware and software configuration used during the experiments is summarised in Table 1.

The selection of a single test environment was intended to limit external variability and improve the repeatability of the experiments. All benchmarks were executed on the same physical machine without concurrent background processes that could affect performance measurements.

Attention was paid to maintaining stable operating conditions throughout all test runs, including identical system configuration, fixed hardware parameters, and consistent software versions. This approach supports the interpretation that the observed performance differences are related mainly to the evaluated frameworks rather than to changes in the execution environment.

Table 1: Test environment specification

Operating System	Windows 10 Education (64-bit)
Processor (CPU)	AMD Ryzen 7 7735HS (8 cores / 16 threads, 3.20 GHz)
RAM	16 GB DDR5 (5600 MHz)
Storage	512 GB SSD (Micron_2400)
Benchmarking tool	k6 (v1.0.0)
Backend frameworks	Express.js (v4.21.2), Spring Boot (v3.3.2)

No additional background workloads were intentionally executed during benchmarking to minimise external interference.

2.2. Application setup

Two functionally equivalent RESTful web applications were implemented using **Express.js** and **Spring Boot**, respectively. Both applications exposed identical HTTP endpoints supporting CRUD operations (Create, Read, Update, Delete). The purpose of this setup was to analyse the impact of the underlying framework and runtime environment on performance, while keeping application logic consistent.

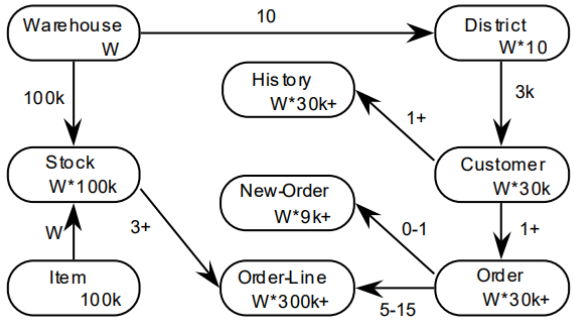

Figure 1: Conceptual database schema based on the TPC-C benchmark specification [14]

Both applications used the same relational data model implemented in PostgreSQL. The schema was based on the TPC-C benchmark structure and included the main entities warehouse, district, customer, order, order line, stock, item, new order, and history, as shown in Figure 1. The structure was used to represent a simplified transactional workload typical of OLTP systems. The tested operations involved creating orders, updating customer and warehouse-related data, retrieving the most recent customer order, and processing delivery-related records. Although the data model and transaction logic were inspired by TPC-C, the implemented workload was not a full TPC-C benchmark and should not be interpreted as an official TPC-C test [14].

The **Express.js** application was executed using **Node.js (v22.14.0)** and relied on the Node.js runtime environment and its event-driven, non-blocking I/O model [3]. The Spring Boot application was implemented using **Spring Boot** and executed on the **Java Virtual Machine (JVM)**, relying on an embedded application server [6].

Both applications were started and stopped separately for each benchmark run. Each benchmark iteration was preceded by a fresh application startup, followed by a controlled cooldown period after completion. This approach reduced the likelihood of cross-run effects such as warm caches, residual memory allocation, or runtime optimisation artefacts. No application instance was reused across benchmark repetitions.

2.3. Benchmarking tool and load configuration

Performance testing was conducted using **k6** [15], an open-source load testing tool designed for modern web applications. The tool was selected because it supports controlled workload generation, constant-arrival-rate execution, scenario-based test design, and detailed reporting of latency and throughput metrics. These capabilities make it suitable for conducting repeatable HTTP performance experiments under stable load conditions.

Each framework was subjected to **three independent benchmark runs**, executed sequentially under identical load conditions. A cooldown period of 90 seconds was applied between consecutive benchmark runs to allow system resources to stabilize before the next measurement. The same request patterns, endpoints and test duration were used for both applications to support a comparable test procedure.

The generated workload consisted of HTTP requests targeting CRUD endpoints. Requests were issued at a constant rate to maintain stable load conditions throughout the benchmark execution. The applied workload represents a custom HTTP-based transactional scenario inspired by the TPC-C transaction mix, including New-Order, Payment, Order-Status, and Delivery operations exposed as REST endpoints. However, the workload does not implement the full TPC-C specification and should not be interpreted as an official TPC-C benchmark [14].

During the measurement phase, the workload was generated at a constant rate of 96 requests per second in total, distributed across CRUD operations as follows: CREATE – 45 req/s, UPDATE – 43 req/s, READ – 4 req/s, and DELETE – 4 req/s. This workload level was maintained for both analysed frameworks under identical test conditions.

The experimental procedure applied in each benchmark run is illustrated in Figure 2.

Figure 2: Experimental procedure workflow

This procedure was repeated three times for each framework. The collected data were then used to analyse global latency metrics, CRUD-specific latency metrics, throughput, run-to-run stability, and system resource utilisation.

2.4. Performance metrics

During each benchmark run, the following metrics were collected and analysed:

- **global latency metrics**: average response time (avg), minimum response time (min), maximum response time (max), and 95th percentile latency (p95);
- **CRUD-specific latency metrics**: average and p95 response times for Create, Read, Update and Delete operations;
- **throughput**: number of processed requests per second (RPS).

In addition to application-level metrics, **system-level performance data** was collected to evaluate runtime resource utilisation:

- **CPU usage (%);**
- **Memory consumption**, including working set and private memory.

System metrics were sampled continuously during benchmark execution and later aggregated for analysis.

2.5. Data processing and analysis

All raw benchmark results were exported from k6 in CSV format and processed using spreadsheet-based analysis. For each framework, metrics from three benchmark runs were aggregated using arithmetic means to obtain representative values. In addition, selected metrics were supplemented with minimum, maximum, and standard deviation values in order to describe run-to-run variability.

The analysis focused on both absolute performance and stability across runs. Differences between Express.js and Spring Boot were evaluated based on global latency metrics, CRUD-specific latency characteristics, throughput consistency, and system resource utilisation.

3. Results

This section presents the quantitative results obtained from the performance evaluation of Express.js and Spring Boot. Each framework was tested in three independent runs under identical conditions to ensure stability and repeatability of observations. We report global latency metrics, CRUD-level behaviour, throughput characteristics and system-level resource utilisation.

3.1. Global latency comparison

The results presented in Figure 3 and Figure 4 indicate consistent differences in latency between Express.js and Spring Boot.

Express.js achieved an average global latency of approximately 10.5-10.6 ms across all runs, while Spring Boot showed higher and more variable results, ranging from 12 ms to nearly 20 ms.

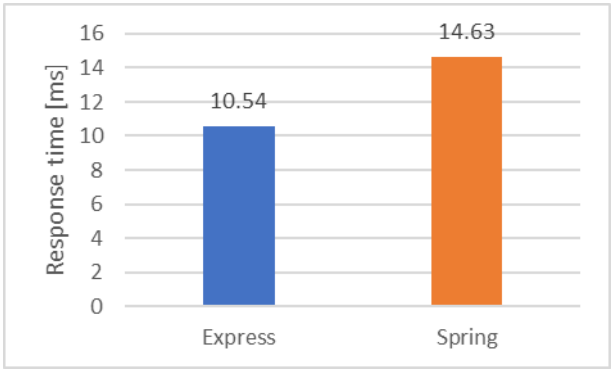

Figure 3: Average global response latency for Express.js and Spring Boot (mean values from three benchmark runs)

The difference becomes more pronounced for the p95 metric, where Spring Boot exhibited higher tail latency and greater variability, especially visible in p95 and maximum values. Figure 4 illustrates higher tail-latency values observed for Spring Boot.

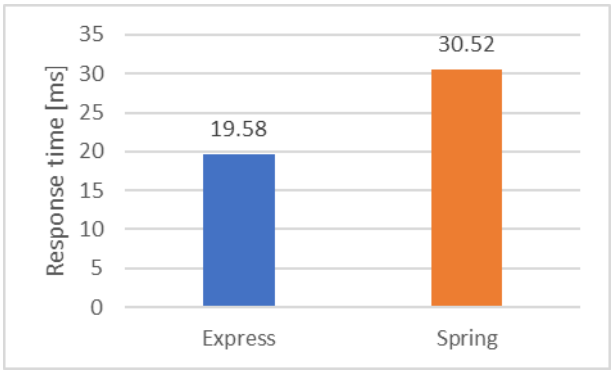

Figure 4: Comparison of 95th percentile global latency for Express.js and Spring Boot (mean values from three benchmark runs)

Express.js maintained stable p95 latency values below 20 ms, indicating a more predictable response time profile under load.

The results indicate that Express.js is associated with lower latency and lower variability, which may be relevant in latency-sensitive applications, as also reflected in the variability metrics presented in Table 2.

Table 2: Variability metrics for global performance indicators

Metric	Framework	MIN	MAX	STD
Avg RPS [req/s]	Express.js	89.07	89.08	0.002
	Spring Boot	89.04	89.08	0.02
Avg response time [ms]	Express.js	10.52	10.57	0.03
	Spring Boot	12.05	19.73	4.42
P95 response time [ms]	Express.js	19.53	19.64	0.06
	Spring Boot	23.32	44.91	12.46

3.2. CRUD performance analysis

Although the operations are grouped under CRUD categories for analytical clarity, they internally correspond to transactional operations inspired by the TPC-C model, such as New-Order, Payment, Order-Status, and Delivery. A detailed breakdown of CRUD operations indicates that Express.js exhibits lower latency values than Spring Boot across all operations. The average CRUD operation latencies are presented in Figure 5. The latency levels observed for individual CRUD operations should be interpreted in the context of the adopted workload distribution, in which write-oriented operations (CREATE and UPDATE) accounted for most of the generated requests.

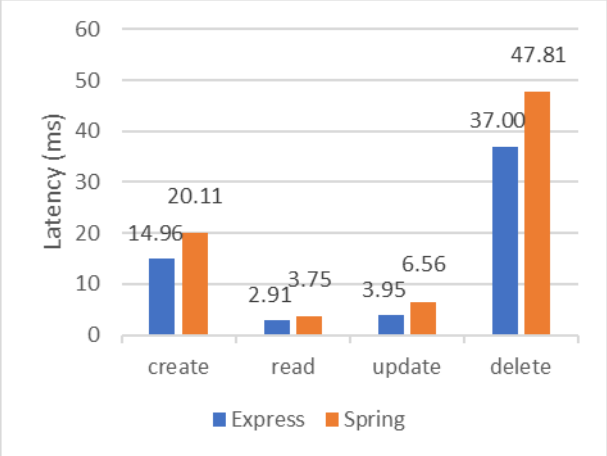

Figure 5: Average latency of CRUD operations for Express.js and Spring Boot (mean values from three benchmark runs)

- Create - Express.js shows lower average and p95 latencies compared to Spring Boot. Spring Boot exhibits noticeable tail latency spikes.
- Read - Express.js shows lower latency values (approximately 3-6 ms) than Spring Boot.
- Update - Express.js shows stable and consistently lower latency across all runs compared to Spring Boot.
- Delete - Express.js achieves lower tail latency and more consistent performance than Spring Boot.

The tail latency distribution for CRUD operations is further summarized in Figure 6.

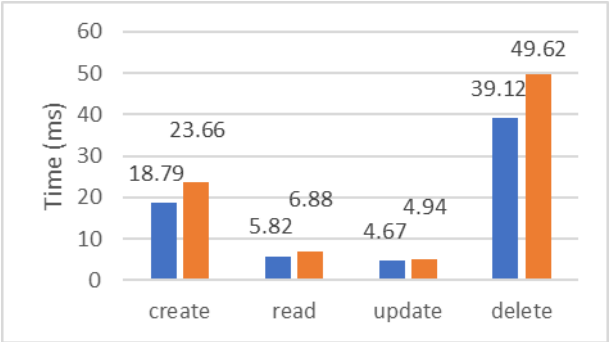

Figure 6: Comparison of 95th percentile latency for CRUD operations

The results confirm that the event-driven architecture of Express.js handles CRUD workflows with lower overhead, especially in high-frequency request scenarios. These observations are further supported by the variability metrics presented in Table 3.

The differences are particularly visible for the DELETE operation, where Spring Boot presented higher maximum values and more pronounced tail behaviour, as also reflected in the higher standard deviation values shown in Table 3.

Table 3: Variability metrics for CRUD latency

Metric	Framework	MIN	MAX	STD
Create [ms]	Express.js	14.91	15.01	0.05
	Spring Boot	17.70	24.87	4.12
Read [ms]	Express.js	2.88	2.96	0.04
	Spring Boot	2.79	5.61	1.61
Update [ms]	Express.js	3.93	3.97	0.02
	Spring Boot	3.76	12.14	4.84
Delete [ms]	Express.js	36.83	37.31	0.26
	Spring Boot	43.89	54.93	6.12

3.3. Throughput comparison

Throughput was measured as the number of processed requests per second (RPS) and analysed in the context of the applied constant-arrival-rate workload. This metric reflects the ability of the system to sustain a given request rate under stable load conditions.

The applied workload used a constant request generation rate, which effectively bounded the maximum achievable throughput. As a result, both frameworks reached a similar requests-per-second level, as shown in Figure 7, allowing the comparison to focus primarily on response latency and stability rather than raw throughput capacity.

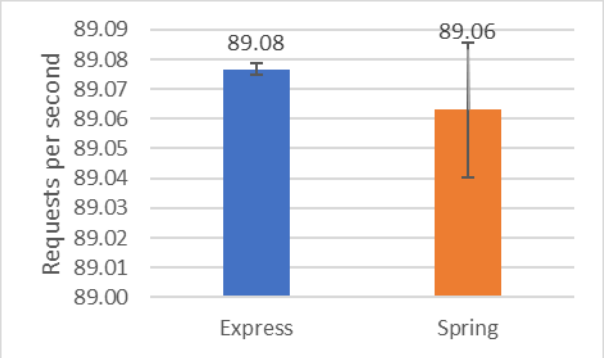

Figure 7: Throughput comparison (requests per second) for Express.js and Spring Boot (mean values from three benchmark runs)

Although throughput values were very similar for both frameworks, Express.js showed slightly lower run-to-run variability, as reflected in the values reported in Table 2. Given the bounded constant-arrival-rate workload, the throughput comparison should be interpreted mainly as an indicator of stability rather than maximum throughput capacity.

3.4. System-level metrics

System metrics collected during benchmarking provide insight into the runtime characteristics of both frameworks.

CPU utilisation remained low for both frameworks, indicating that CPU was not a performance bottleneck, as shown in Figure 8.

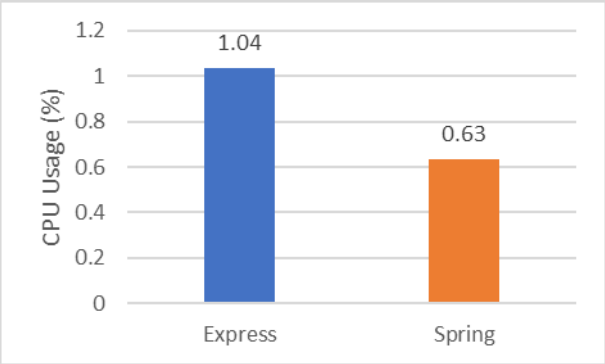

Figure 8: Average CPU utilisation during benchmark execution (mean values from three benchmark runs)

Spring Boot's memory footprint was substantially larger, reflecting the inherent overhead of the Java Virtual Machine. Express.js used only a fraction of Spring Boot's memory, as illustrated in Figure 9, which may be relevant in resource-constrained environments. These observations are consistent with differences between lightweight JavaScript runtimes and JVM-based frameworks.

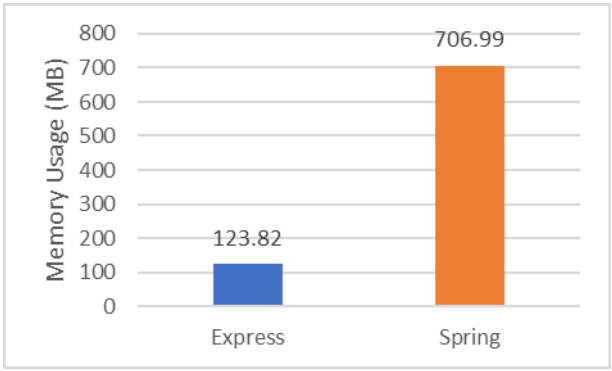

Figure 9: Average memory usage (working set) for Express.js and Spring Boot (mean values from three benchmark runs)

4. Discussion

The experimental results indicate consistent differences in performance between Express.js and Spring Boot, particularly in latency-related metrics and memory utilisation. The lower latency observed for Express.js is consistent with its event-driven, non-blocking I/O architecture, which reduces per-request overhead and avoids thread-management mechanisms typical of JVM-based servers. This design allows Express.js to process high-frequency CRUD operations with reduced scheduling overhead and relatively stable latency, as observed in both global and CRUD-specific metrics.

In contrast, the performance characteristics of Spring Boot reflect the architectural complexity of the Java Virtual Machine [6]. While the JVM provides capabilities such as just-in-time compilation, automatic memory management, and extensive library support, these mechanisms introduce additional runtime overhead. Garbage collection cycles [16], thread-pool scheduling, and object lifecycle management may contribute to increased latency, particularly under sustained load. This effect is visible in higher p95 and maximum latency values observed for Spring Boot, indicating the presence of long-tail delays that may affect latency-sensitive applications [1].

The results obtained in this study are broadly consistent with previous reports on framework-level performance differences. Wicha and Pańczyk also observed measurable differences in response time and resource utilisation between Spring- and Express-based REST API implementations [9]. Similar trends can be seen in the study by Szewczyk and Skublewska-Paszkowska, in which Express.js was characterised by stable resource management, while Spring Boot achieved competitive efficiency among the analysed frameworks [10]. At the same time, the present results differ from those reported by Choma et al., who found Spring Boot to be highly efficient under their adopted load scenarios, although they also noted reliability limitations for some technologies under extreme request volumes [11]. This discrepancy may be related to differences in workload design, measurement scope, and evaluation criteria. A similar dependence on workload profile was reported by Shkodra et al., who showed that relative performance advantages may change with load intensity and request characteristics [12].

The differences in CPU and memory utilisation are also consistent with this interpretation. Spring Boot's higher resource consumption reflects the requirements of the JVM, which maintains baseline memory allocation and executes background processes. Express.js, by contrast, demonstrates a lightweight runtime model associated with lower resource usage. These characteristics may be relevant in resource-constrained deployments, containerised environments, or horizontally scalable systems. Despite these differences, both frameworks achieved comparable throughput under the applied constant-arrival-rate workload. This indicates that the adopted load profile primarily exposed differences in latency and stability rather than in maximum throughput capacity.

An additional aspect worth noting is run-to-run stability. Across three benchmark repetitions, Express.js showed lower variability in global latency, CRUD-level latency, and throughput, while Spring Boot exhibited larger deviations in selected latency metrics. This observation is consistent with the variability values reported in Tables 2 and 3 and suggests that the Node.js-based implementation behaved in a more predictable manner under the applied workload.

Overall, the findings indicate a trade-off between runtime flexibility and latency efficiency. While Spring Boot provides a mature ecosystem for building complex enterprise applications, its architecture introduces overhead that affects latency under high-frequency workloads. Express.js, with its lightweight design, demonstrates higher efficiency in the tested CRUD-oriented scenario. These observations should, however, be interpreted in the context of the adopted workload, the single-machine environment, and the three-run measurement design.

5. Conclusions

This study compared Express.js and Spring Boot in CRUD-oriented web applications under a unified benchmarking procedure conducted on identical hardware and with the same workload profile. The results indicate that Express.js achieved lower global latency, lower p95 latency, and lower run-to-run variability than Spring Boot. CRUD-level measurements also showed lower latency values for Express.js across all analysed operations, with the most pronounced difference observed for Delete requests.

Both frameworks achieved comparable throughput under the applied constant-arrival-rate workload, which means that the adopted test profile exposed differences mainly in latency and stability rather than in raw throughput capacity. System-level measurements were also consistent with these findings, as Express.js required less CPU and substantially less memory than Spring Boot under the same conditions.

From a practical perspective, the results suggest that Express.js is better suited for latency-sensitive and resource-constrained CRUD workloads, whereas Spring Boot remains a valid choice when architectural consistency, extensive tooling, and enterprise-level integration are prioritised over raw latency performance.

This study also has important limitations. The experiments were conducted on a single physical machine, under a single workload level and a fixed CRUD request distribution, and the workload was inspired by TPC-C rather than being a full benchmark specification. In addition, the analysis was based on three benchmark repetitions, which improves comparability but still does not fully characterise long-term runtime variability. Future work should therefore include experiments under different workload levels, alternative request distributions, broader infrastructure configurations, and containerised or distributed environments, as well as comparisons with additional backend frameworks.

References

[1] J. Dean, L. A. Barroso, The Tail at Scale, Communications of the ACM 56(2) (2013) 74–80, https://doi.org/10.1145/2408776.2408794.

[2] L. Erickson, Application Monitoring 101: Averages Lie, Percentiles Clarify, https://www.scoutapm.com/blog/application-monitoring-101-averages-lie-percentiles-clarify, [12.04.2026].

[3] Node.js Foundation, Node.js Documentation, https://nodejs.org/en/docs/, [12.04.2026].

[4] Express.js Team, Express.js - Fast, unopinionated, minimalist web framework for Node.js, https://expressjs.com/, [12.04.2026].

[5] Spring Team, Spring Boot Reference Documentation, https://docs.spring.io/spring-boot/docs/3.2.5/reference/html/, [12.04.2026].

[6] Oracle Corporation, Java Virtual Machine Specification, https://docs.oracle.com/javase/specs/, [12.04.2026].

[7] K. Huppler, The Art of Building a Good Benchmark, In: Performance Evaluation and Benchmarking, LNCS 5895, Springer (2009) 18–30, https://doi.org/10.1007/978-3-642-10424-4_3.

[8] A. Silberschatz, H. F. Korth, S. Sudarshan, Database System Concepts, McGraw-Hill, New York, 2010.

[9] M. Wicha, B. Pańczyk, Performance Analysis of REST API Technologies Using Spring and Express.js Examples, Journal of Computer Sciences Institute 29 (2023) 352–359, https://doi.org/10.35784/jcsi.3796.

[10] D. Choma, K. Chwaleba, M. Dzieńkowski, The Efficiency and Reliability of Backend Technologies: Express, Django, and Spring Boot, Informatyka, Automatyka, Pomiary w Gospodarce i Ochronie Środowiska 13(4) (2023) 73–78, https://doi.org/10.35784/iapgos.4279.

[11] M. Szewczyk, M. Skublewska-Paszkowska, Performance comparison of development frameworks in selected environments in REST API architecture, Journal of Computer Sciences Institute 35 (2025) 121–128, https://doi.org/10.35784/jcsi.7041.

[12] E. Shkodra, E. Jajaga, M. Shala, Development and Performance Analysis of RESTful APIs in Core and Node.js using MongoDB Database, Proceedings of WEBIST 2021 (2021) 227–234, https://doi.org/10.5220/0010621200003058.

[13] A. Georges, D. Buytaert, L. Eeckhout, Statistically Rigorous Java Performance Evaluation, ACM SIGPLAN Notices 42(10) (2007) 57–76, https://doi.org/10.1145/1297105.1297033.

[14] TPC-C Benchmark Standard Specification Revision 5.11, Transaction Processing Performance Council, https://www.tpc.org/tpcc/, [12.04.2026].

[15] k6 Team, k6 - Modern Load Testing Tool, https://k6.io/docs/, [12.04.2026].

[16] Oracle Corporation, Garbage Collection in Java, https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/, [12.04.2026].