JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 39 (2026) 183–187
Received: 18 April 2026
Accepted: 10 May 2026

Comparative Performance Analysis of Spring Boot and Ktor for a Ticket Reservation REST API on the JVM

Miłosz Serej*, Kamil Kopciński, Jakub Smołka

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

Abstract
This paper presents an experimental comparison of Spring Boot and Ktor as backend frameworks for REST API development on the JVM platform. The study was motivated by the need for a controlled evaluation of these frameworks, since available comparisons are often based on non-uniform assumptions and do not clearly separate framework-related effects from other implementation factors. To ensure comparability and reduce the influence of external variables, two reference applications implementing the same ticket reservation business scenario and sharing the same PostgreSQL database were developed. The evaluation was conducted with load tests covering write, read and mixed workloads. The analysis included response times, system stability, the number of successful and unsuccessful responses, as well as CPU and RAM usage. The results show that both frameworks maintained full technical correctness, while Ktor achieved lower response times and lower average CPU usage.

Keywords: Spring Boot; Ktor; REST API; load testing; JVM; performance

*Corresponding author
Email address: milosz.serej@gmail.com (M. Serej)

1. Introduction

Modern information systems increasingly rely on distributed architectures in which backend services exposing REST APIs play a fundamental role [1]. In such systems, the application framework affects not only the implementation of business logic, but also performance, stability and maintainability. In the JVM ecosystem, Spring Boot remains one of the most widely used solutions due to its broad ecosystem and extensive auto-configuration mechanisms [2, 3]. An alternative is Ktor, a framework designed for Kotlin and based on a more explicit and modular configuration model [4-6].

Although many comparisons of these frameworks can be found in the literature and industry sources, a large part of them is descriptive in nature or based on non-uniform assumptions. In many cases, the compared applications do not share the same business scenario, the same data model or the same testing environment, which makes the interpretation of results less reliable [7-12].

Therefore, the aim of this paper was to compare Spring Boot and Ktor under a controlled experimental setup. Two reference applications implementing the same business scenario – a ticket reservation system – were developed and evaluated under load. The research hypothesis assumed that Ktor could achieve lower response times and lower CPU usage than Spring Boot in a lightweight JVM REST API scenario.

1.1. Background

Spring Boot is one of the most established frameworks in the JVM ecosystem and is commonly used in enterprise systems due to its extensive ecosystem, auto-configuration mechanisms and broad integration capabilities [2, 3]. Its popularity makes it a natural reference point in comparative studies concerning backend application development.

Ktor, in contrast, represents a lighter and more explicit approach. As a Kotlin-first framework, it promotes modularity and direct configuration of application components. Because of these characteristics, Ktor is often considered a promising alternative in projects where low overhead and fine-grained control over the application setup are desirable [4-6].

1.2. Research gap

Despite the popularity of both frameworks, many available comparisons are limited to simplified benchmarks or qualitative descriptions. In many cases, the compared applications do not share the same business scenario, the same persistence layer or the same load profile. This reduces the reliability of conclusions and makes it difficult to distinguish framework-related effects from implementation-related differences [7-12].

1.3. Objective and hypothesis

The objective of this study was to compare Spring Boot and Ktor in a controlled experimental setup using the same ticket reservation scenario, the same PostgreSQL database and the same load generation tool. The study assumed that Ktor could achieve lower response times and lower CPU usage than Spring Boot in a lightweight JVM REST API system, while preserving full technical correctness.

2. Materials and Methods

The subject of the study was a comparative analysis of two JVM backend frameworks: Spring Boot and Ktor. To ensure comparability, two reference applications implementing the same ticket reservation system were developed. Both implementations shared the same PostgreSQL relational database running in a Docker container [13-16].

Tests were performed locally on the same machine and in a sequential manner, meaning that only one tested application was active at a time. This reduced the risk of mutual interference in system and database resources and improved the comparability of the results. For technical separation, the Spring Boot application was exposed on port 8080, whereas the Ktor application was exposed on port 8081.

2.1. Business scenario and API

Both applications exposed the same REST API [1]. The main endpoint was POST /reservations, responsible for creating a reservation. This endpoint implemented the business logic required to verify whether an event existed, check ticket availability, perform a consistent update of the ticket pool, and store the reservation result. The second endpoint was GET /reservations/{id}, used to retrieve a previously created reservation. The HTTP response logic was unified as well. Code 201 indicated a successful reservation, code 409 indicated a reservation rejected due to the limited ticket pool, and code 404 indicated a missing resource, such as a non-existing event or reservation.

2.2. Data model

The system used a shared PostgreSQL database with two tables: events and ticket_reservations. The events table stored event information and the current number of available tickets, whereas the ticket_reservations table stored the history and outcomes of reservation operations. This data model was designed to reflect the core logic of a ticket reservation system while keeping the experimental setup sufficiently simple and comparable across both frameworks.

The events table acted as the source of the current resource state. In particular, the available_tickets field represented the shared value modified during reservation requests. The ticket_reservations table stored information about completed reservation attempts, including the reservation identifier, event identifier, user identifier, requested number of tickets, ticket type, operation status and failure reason. As a result, the database model supported both functional verification and performance-oriented analysis.

From the API perspective, the POST /reservations endpoint required reading the selected event, checking ticket availability, updating the available ticket pool and storing the outcome of the reservation request. The GET /reservations/{id} endpoint was based on reading a previously stored reservation record. This means that the adopted data model supported both write-heavy and read-heavy workloads, which was important for the comparative analysis.

The use of a relational database was important because the reservation scenario required transactional consistency and protection against race conditions [13, 14]. This made it possible to evaluate not only performance but also the correctness of concurrent write operations.

2.3. Test dataset

The experiment was based on synthetic test data prepared specifically for the ticket reservation scenario. Initial event records and ticket pool values were defined before each test series, while reservation requests were generated automatically by the load testing tool. Such an approach improved repeatability and ensured that the obtained results reflected the behavior of the compared frameworks rather than the properties of an external dataset.

The complete dataset covered 30 test runs, corresponding to 5 repetitions for each of the 3 load scenarios and 2 frameworks. Under the default load parameters, a single run generated approximately 22,150 virtual users. Depending on the scenario, this resulted in about 22,150 request records for the POST test, 88,600 request records for the GET test, and approximately 39,870 request records for the MIXED test. In total, the detailed dataset contained approximately 1.5 million records describing individual HTTP requests. In addition, 30 aggregated run-level records and about 2,640 CPU and RAM measurement samples were collected.external dataset.

2.4. Experimental method

Gatling was used as the load generation tool. Three test scenarios were defined:

- post – focused on write-heavy operations,
- get – focused on read-heavy operations,
- mixed – a mixed workload with 80% GET operations and 20% POST operations.

Each scenario followed the same load profile consisting of four phases:

- Ramp – gradual increase of load,
- Steady – constant load,
- Spike – sudden traffic increase,
- Soak – prolonged load.

Each scenario was repeated five times for each framework. The analysis included the number of requests, successful and unsuccessful responses, median response time (P50), P95, P99, mean response time, maximum response time, and average CPU and RAM usage. CPU and RAM measurements were recorded automatically during each test run by a background monitoring task launched before the start of the load test. Resource samples were collected every 5 seconds and stored in a CSV file. The recorded values included total CPU usage of the whole system, available memory, and committed memory. Therefore, the reported CPU and RAM values describe the overall resource usage of the test environment while the tested application was active, rather than the consumption of the Spring Boot or Ktor process alone.

During the measurements, the test machine was left without user interaction, and only the software required for the experiment was active, including the tested application, the database container and the development environment.

2.5. Experimental setup

All tests were executed on the same local machine and under the same operating conditions. Only one tested application was active at a given time, which reduced the risk of mutual interference in CPU, RAM or database access. The shared PostgreSQL instance was deployed in a Docker container to maintain a consistent persistence environment [15, 16].

2.6. Repetition and result consistency

Each scenario was repeated five times for each framework. Repeated runs were necessary because single measurements may be influenced by incidental factors such as JVM warm-up effects, operating system background activity or temporary fluctuations in resource usage. The repeated measurements made it possible to assess not only average results but also their stability.

2.7. Metrics

The analysis included the total number of requests, the number of successful and unsuccessful responses, P50, P95, P99, mean response time, maximum response time, and average CPU and RAM usage. In the presented results, successful responses are marked as OK, whereas KO denotes responses classified by the load testing tool as unsuccessful. These metrics were selected to assess both the typical response behavior and the stability of the system under load [17, 18].

3. Results

The POST scenario was the most demanding one because it involved state-changing operations and updates to a shared resource in the database. The aggregated results for this scenario are presented in Table 1.

Table 1: Aggregated results for the POST scenario

Metric	Ktor	Spring Boot
Requests	22150.0	22150.0
OK	22150.0	22150.0
KO	0.0	0.0
P50 [ms]	3.0	4.0
P95 [ms]	4.0	5.0
P99 [ms]	4.2	5.2
Mean [ms]	3.0	4.0
Max [ms]	226.6	222.0
CPU avg [%]	4.44	4.85
RAM avg [MB]	15408.18	15260.58

As shown in Table 1, both frameworks processed the same number of requests and produced no KO responses, that is, no responses classified as unsuccessful by the load testing tool. Ktor achieved lower P50, P95, P99 and mean response time values. It also showed slightly lower CPU usage, while Spring Boot consumed slightly less RAM.

The POST scenario was the most demanding one because it combined request processing, database access and state modification. From the business perspective, it was also the most sensitive operation, since each successful request changed the shared ticket pool.

The results presented in Table 1 show that both frameworks maintained full technical correctness. No KO responses were recorded, which indicates that neither implementation failed under the adopted load profile. This is important because the tested operation involved concurrent access to a shared resource and could therefore reveal consistency-related problems.

In terms of latency, Ktor achieved lower values for all major timing metrics. The difference was visible not only in the median response time, but also in P95 and P99, which means that Ktor performed better not only in typical responses but also in slower and more demanding cases. This suggests a more stable latency profile under the tested write-heavy workload.

Although the maximum response time was slightly lower in Spring Boot, this metric should be interpreted with caution, as it reflects only a single extreme observation. In contrast, the percentile values presented in Table 1 represent a more general view of system behavior and therefore provide a more reliable basis for comparison.

3.1. Get scenario

The GET scenario focused on read operations and served as a reference against the write-intensive scenario. The aggregated results for this scenario are presented in Table 2.

Table 2: Aggregated results for the GET scenario

Metric	Ktor	Spring Boot
Requests	88600.0	88600.0
OK	88600.0	88600.0
KO	0.0	0.0
P50 [ms]	1.0	1.0
P95 [ms]	3.0	4.0
P99 [ms]	4.0	5.0
Mean [ms]	1.0	2.0
Max [ms]	163.2	164.2
CPU avg [%]	5.63	6.51
RAM avg [MB]	15381.49	15202.63

As shown in Table 2, both frameworks maintained full technical correctness. Ktor achieved lower P95, P99 and mean response time values, indicating a more favorable performance profile in read operations.

The GET scenario represented a read-heavy workload and did not involve modification of the shared ticket pool. For this reason, it can be treated as a less conflict-prone scenario and as a useful reference point for assessing pure retrieval performance.

Again, both frameworks processed the same number of requests without technical failures. The absence of KO responses confirms that both implementations remained stable under the adopted load profile.

The difference between frameworks became visible in higher-percentile response times and in the mean response time. Although the median was identical, Ktor achieved lower P95 and P99 values. This indicates that the Ktor-based application handled slower requests more efficiently and maintained better latency stability in the read-dominant workload.

3.2. MIXED scenario

The MIXED scenario combined read and write operations and best reflected a realistic business workload. Its aggregated results are presented in Table 3.

Table 3: Aggregated results for the MIXED scenario

Metric	Ktor	Spring Boot
Requests	39864.2	39889.8
OK	39864.2	39889.8
KO	0.0	0.0
P50 [ms]	2.0	3.0
P95 [ms]	3.0	5.0
P99 [ms]	4.0	5.0
Mean [ms]	2.0	3.0
Max [ms]	164.0	165.8
CPU avg [%]	4.39	5.18
RAM avg [MB]	15521.0	15372.0

As shown in Table 3, Ktor again achieved lower response times and lower CPU usage, while Spring Boot consumed slightly less RAM. This result is particularly important because the MIXED scenario most closely reflects real system traffic.

The MIXED scenario is particularly important because it combines read and write requests and therefore reflects a more realistic application workload than the isolated POST or GET cases. In this setup, the majority of traffic consisted of reads, while writes still influenced the global system behavior.

Both frameworks again maintained full technical correctness, as no KO responses were recorded. This confirms that the implementations remained stable even when handling heterogeneous traffic patterns. Ktor also preserved its advantage in response times also in this scenario. Lower P50, P95, P99 and mean values indicate that the framework performed well in this mixed workload.

3.3. Cross-scenario comparison

Across all scenarios, Ktor consistently achieved lower response times than Spring Boot. The difference was present in POST, GET and MIXED workloads, which indicates that the observed behavior was not limited to a single access pattern. At the same time, both frameworks showed full technical correctness, as no KO responses were recorded.

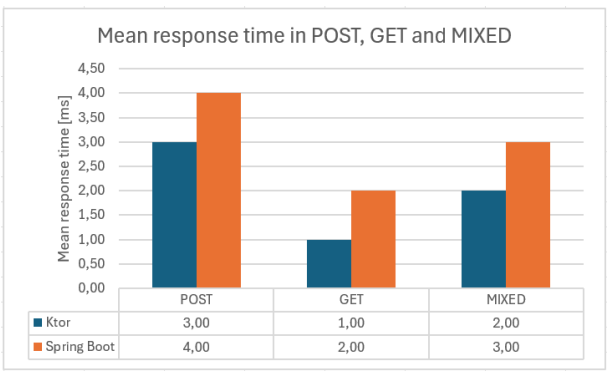

Figure 1: Mean response time in the POST, GET and MIXED scenarios

Figure 1 confirms that Ktor achieved lower mean response times in all tested scenarios. The largest relative difference was observed in the GET and MIXED workloads, while the POST scenario also showed a consistent advantage of Ktor over Spring Boot.

The CPU measurements also showed a consistent tendency: Ktor used less CPU on average in all tested scenarios. RAM usage showed the opposite trend, with Spring Boot consuming slightly less memory. This indicates that the performance advantage of Ktor was accompanied by a modest increase in memory usage.

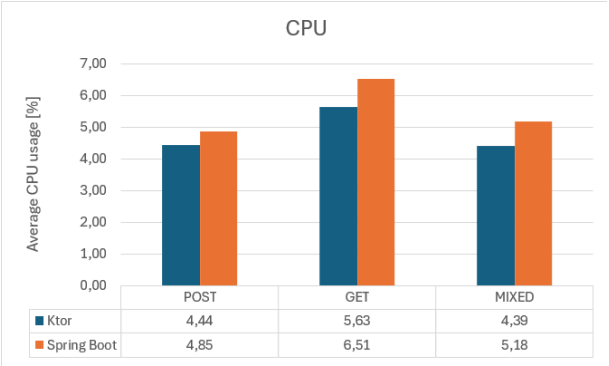

Figure 2: Average CPU usage in the POST, GET and MIXED scenarios

As shown in Figure 2, Ktor maintained lower average CPU usage in all three scenarios. This suggests that the lower response times were accompanied by favorable processor efficiency rather than increased CPU load.

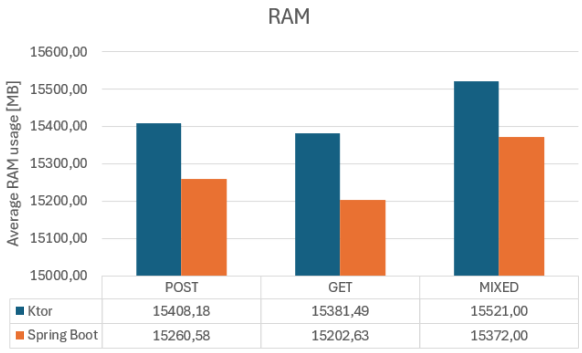

Figure 3: Average RAM usage in the POST, GET and MIXED scenarios

Figure 3 shows that Spring Boot used slightly less RAM than Ktor in all tested scenarios. However, the observed differences were moderate and did not affect the technical correctness or stability of the applications.

4. Discussion and Conclusions

The obtained results indicate that both frameworks maintained full technical correctness under the adopted load profile. In all test runs, 0 KO responses were recorded, which means that both Spring Boot and Ktor correctly implemented the ticket reservation logic without technical instability.

At the same time, the results did not confirm performance equivalence between the two solutions. In all three scenarios, Ktor achieved lower response times than

Spring Boot. This difference was visible not only in median values, but also in the higher percentiles P95 and P99, which indicates that Ktor maintained better latency stability under load [7–12].

The resource usage analysis showed that Ktor used less CPU on average in all scenarios, whereas Spring Boot consumed slightly less RAM. This means that Ktor's time advantage was not achieved solely at the cost of increased resource usage, but rather reflected a more favorable overall performance profile [8–12].

The obtained results therefore support the research hypothesis. In the tested scenario, Ktor achieved lower response times and lower CPU usage than Spring Boot while preserving full technical correctness. In the adopted experimental setup, Ktor proved to be the more efficient solution in terms of latency and processor load.

At the same time, the study did not demonstrate an unconditional advantage of one framework in every aspect. Spring Boot consistently showed slightly lower RAM usage. Therefore, the comparison should be interpreted as a trade-off between different runtime characteristics rather than as an absolute superiority of one framework over the other.

Several limitations of the study should be emphasized. The experiment was based on a single business scenario, a shared PostgreSQL database, and a local execution environment. The applications did not include additional enterprise-level concerns such as security mechanisms, asynchronous messaging, caching, or integration with external services. Therefore, the results should be treated as valid for the tested case rather than universally representative for all JVM backend systems.

Future work should include more complex business logic, external integrations, cloud or container-orchestrated environments, and a broader range of framework configurations. It would also be valuable to compare the analyzed frameworks under larger-scale distributed deployment conditions.

Based on the conducted experiment, the following conclusions can be formulated:

1. Both frameworks maintained full technical correctness and stability under the tested load profile.
2. Ktor achieved lower response times than Spring Boot in all analyzed scenarios.
3. Ktor used less CPU on average, while Spring Boot consumed slightly less RAM.

References

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

[2] Spring, Spring Boot Reference Documentation, https://docs.spring.io/spring-boot/reference/index.html, [08.04.2026].

[3] Spring, Spring Boot project page, https://spring.io/projects/spring-boot, [08.04.2026].

[4] JetBrains, Ktor Documentation, https://ktor.io/docs/welcome.html, [08.04.2026].

[5] JetBrains, Creating a server – Ktor Documentation, https://ktor.io/docs/server-create-and-configure.html, [08.04.2026].

[6] JetBrains, Kotlin Coroutines Overview, https://kotlinlang.org/docs/coroutines-overview.html, [08.04.2026].

[7] M. Jeleń, M. Dzieńkowski, The comparative analysis of Java frameworks: Spring Boot, Micronaut and Quarkus, Journal of Computer Sciences Institute 21 (2021) 287–294, https://doi.org/10.35784/jcsi.2724.

[8] Ł. Wyciślik, Ł. Latusik, A. M. Kamińska, A Comparative Assessment of JVM Frameworks to Develop Microservices, Applied Sciences 13(3) (2023) 1343, https://doi.org/10.3390/app13031343.

[9] P. Plecinski, N. Bokla, T. Klymkovych, M. Melnyk, W. Zabierowski, Comparison of Representative Microservices Technologies in Terms of Performance for Use for Projects Based on Sensor Networks, Sensors 22(20) (2022) 7759, https://doi.org/10.3390/s22207759.

[10] K. X. Carmo, F. Ferreira, E. Figueiredo, Performance Evaluation of Back-End Frameworks: A Comparative Study, Proceedings of the 20th Brazilian Symposium on Information Systems 43 (2024) 1–9, https://doi.org/10.1145/3658271.3658314.

[11] O. C. Novac, D. Ghiurău, M. C. Novac, C. E. Gordan, M. Oproescu, G. Bujdoso, Comparison of Node.Js and Spring Boot in web development, In 2023 15th International Conference on Electronics, Computers and Artificial Intelligence (ECAI) (2023) 1–7, https://doi.org/10.1109/ECAI58194.2023.10194025.

[12] S. du Plessis, B. Mendes, N. Correia, A Comparative Study of Microservices Frameworks in IoT Deployments, In 2021 International Young Engineers Forum on Electrical and Computer Engineering (YEF-ECE) (2021) 86–91, https://doi.org/10.1109/YEF-ECE52297.2021.9505049.

[13] PostgreSQL Global Development Group, PostgreSQL Documentation: Concurrency Control, https://www.postgresql.org/docs/current/mvcc.html, [08.04.2026].

[14] PostgreSQL Global Development Group, PostgreSQL Documentation: Transactions, https://www.postgresql.org/docs/current/tutorial-transactions.html, [08.04.2026].

[15] Docker Inc., What is Docker?, https://docs.docker.com/get-started/docker-overview/, [08.04.2026].

[16] E. Casalicchio, V. Perciballi, Measuring Docker Performance: What a mess!!!, Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion (2017) 11–16, https://doi.org/10.1145/3053600.3053605.

[17] Gatling Corp., Gatling Documentation, https://docs.gatling.io/, [08.04.2026].

[18] Gatling Corp., Load testing concepts, https://docs.gatling.io/testing-concepts/, [08.04.2026].